Digital Circuit Simulator
User Manual

1 Overview

This simulator supports the creation and simulation of digital circuits, interconnecting a variety of modules, from simple digital gates (e.g., ANDs, NOTs) up to processors, memories, and peripherals, enabling the simulation of very simple digital circuits up to complete processor-based systems, programmed in assembly language and even in C.

It was developed as the experimental support to the book "Arquitetura de Computadores", 5th edition, ISBN 978-972-722-789-1, by José Delgado and Carlos Ribeiro (book in Portuguese). However, it can be used to explore and to experiment with the behavior of digital circuits, in general.

The simulator is written in the Java language and requires a Java environment to be installed (see section 2.2). Given the portability of Java, the simulator can be used in a wide variety of operating systems: Windows, macOS and Linux. It is distributed as a Java archive (jar) file and requires no installation other than the Java environment.

A digital circuit consists of modules with pins, interconnected by connections. All modules can be configured (parameterized, according to its functionality) and many have a simulation-time user interface that allows its state and behavior to be seen in real time by the user, in addition to providing a means for the user to interact with the simulated circuit. Connections can also provide information on the pins they connect to and on their current value.

The simulator includes a variety of predefined modules that cover the most used cases. It is possible to define new ones, but that requires changing the program and generating a new jar.

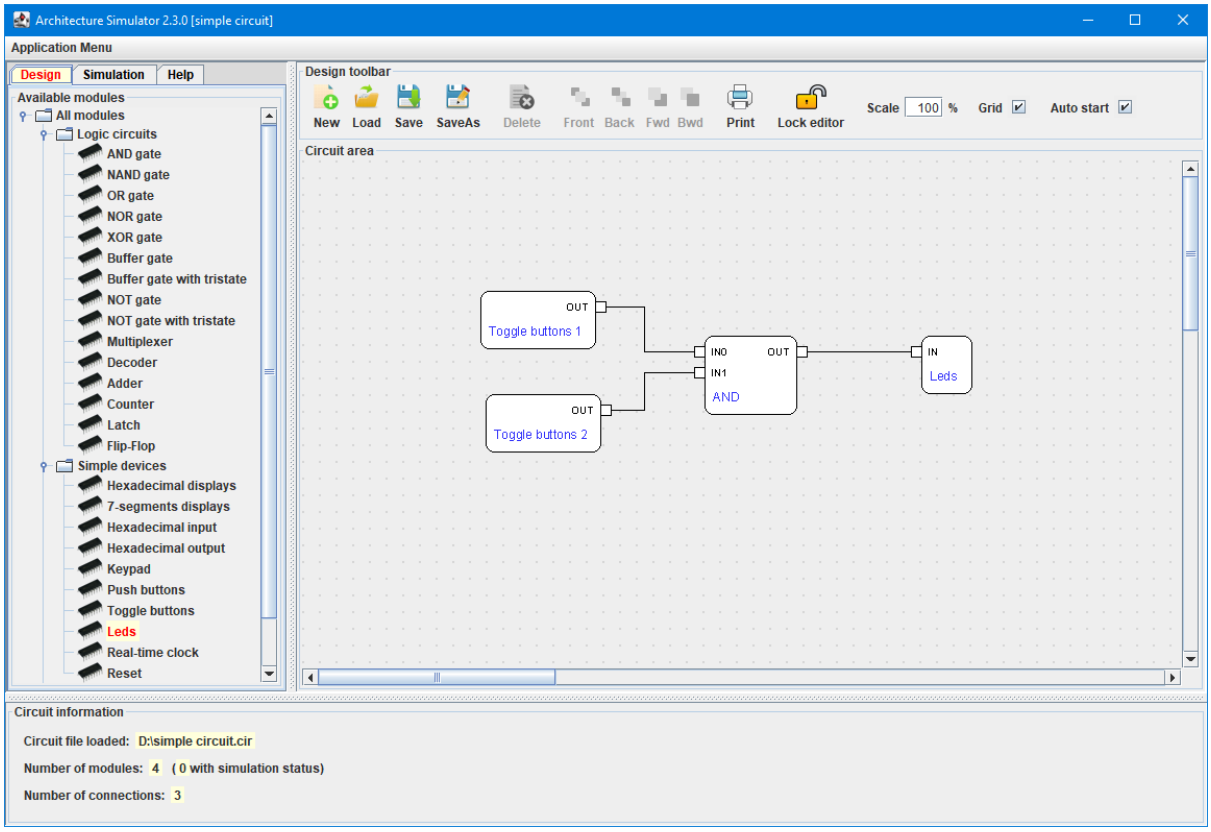
This manual covers essentially the following aspects:

- How to install Java and how to run the simulator;
- How to use the simulator;
- The characteristics of each of the predefined modules and how to use them.

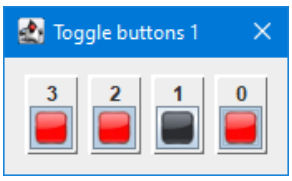
The simulator has 3 modes:

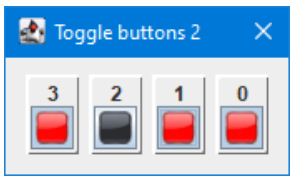
- **Design** – Allows creating and editing digital circuits;
- **Simulation** – Runs a simulation of the circuit;
- **Help** – Displays this manual.

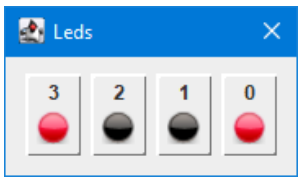
The following figure depicts the simulator in Design mode, showing a simple circuit with an AND gate, with inputs connected to two toggle buttons and the output connected to leds. All four modules have been configured for 4 bits.



Switching to Simulation mode, the behavior of the circuit is simulated. Clicking the buttons and the leds modules shows their simulation interfaces, as the following figure illustrates as an example. Only when the corresponding button bits are both 1 (red) does the corresponding bit in the leds becomes 1 (red). Clicking each button bit icon in the buttons toggles its state and may affect the corresponding led bit icon.



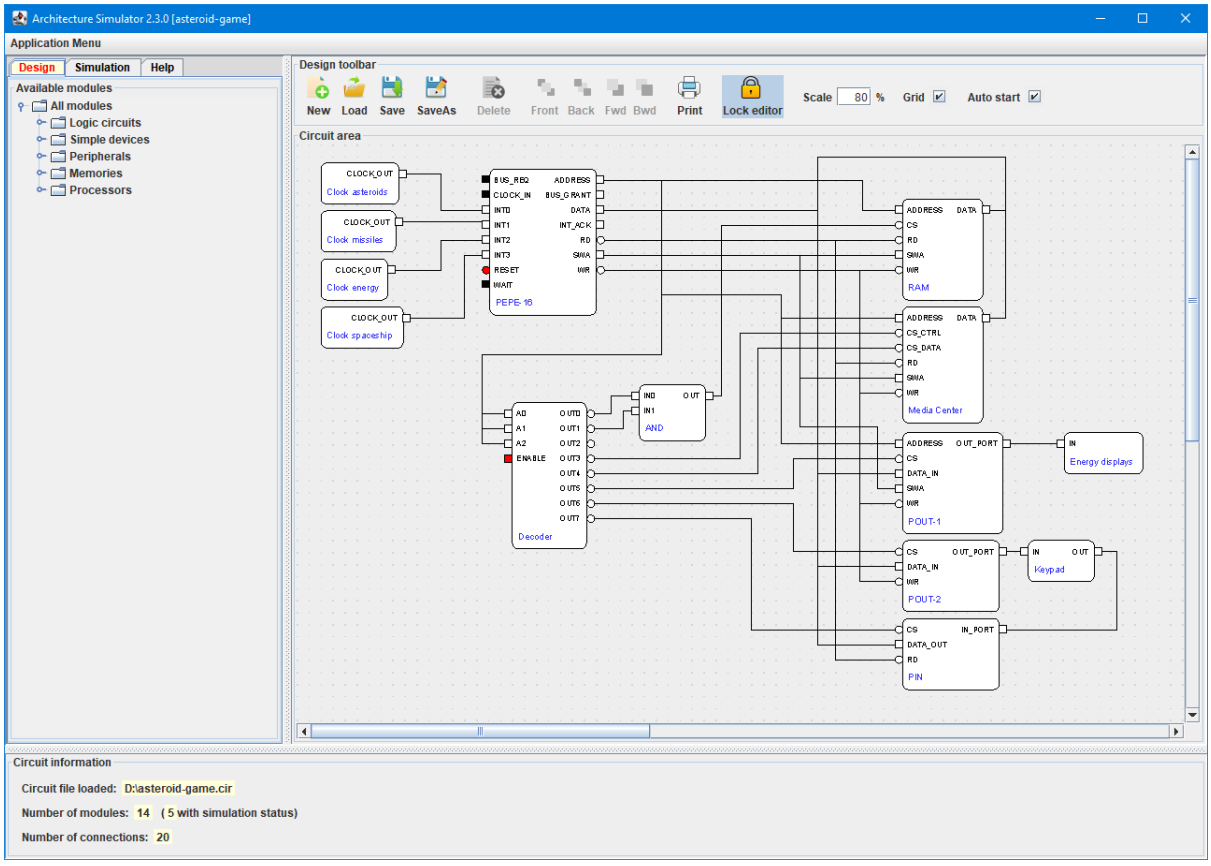




The following figures illustrate a much more complex application:

- A processor-based circuit (with a processor, a memory bank, several peripherals, and several timers), shown in Design mode. The processor can be programmed in assembly language or in the C language, typically with some lower-level code segments in assembly;
- The main user interface (in Simulation mode) of a game written in assembly language, using the circuit above. This window is the interface of the MediaCenter module, which provides a graphics screen with a background video and various objects displayed in the foreground. Interaction with the game is done via a keypad (user interface not shown) connected to a peripheral and read by the program;

- The game shows the dashboard of a spaceship navigating in space, sending missiles to destroy asteroids that get in the way, before the spaceship hits them and gets destroyed (if that happens, the game ends);
- The background scenario is provided by a video and all the objects shown are drawn by the program, pixel by pixel. The program moves these objects by erasing them in one position and drawing them in a new position.



2 Installing and running the simulator

2.1 Supported platforms

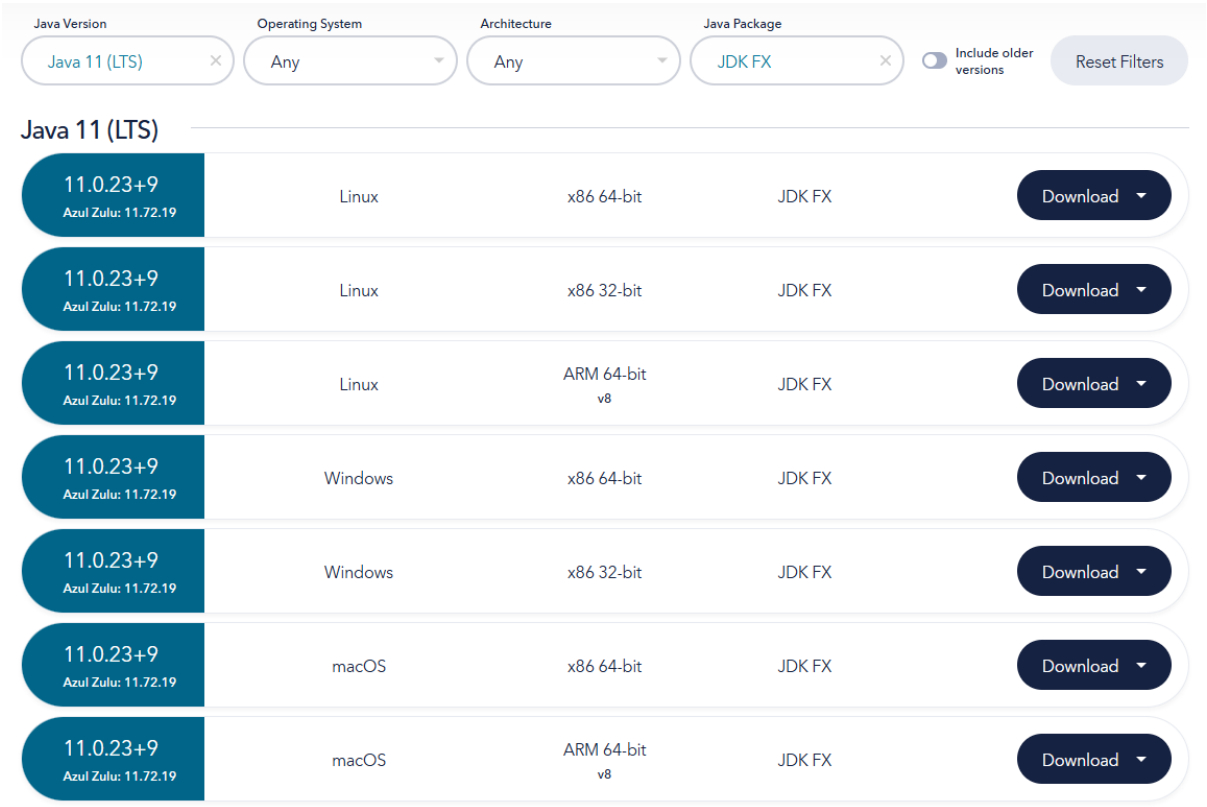
The simulator can be run on Windows, macOS and Linux, as long as an appropriate Java runtime is installed, as described below.

2.2 Installing the simulator

A Java runtime environment (version 11 LTS, 17 LTS, or 21 LTS, 64 bits) must be installed on your computer. LTS versions have Long Term Support and are more stable.

Any distribution of OpenJDK 11, 17, or 21 can be used. The recommended distribution is Azul Zulu, which can be obtained from https://www.azul.com/downloads/. It already includes the JavaFX libraries used by the MediaCenter, as long as the JDK FX package is selected.

Opening the above link, scroll down and choose the Java version 11, 17, or 21 (LTS), the operating system, the architecture (64 bits, x86 or ARM), and the JDK FX package (be sure to select JDK FX and no other!). The following figure illustrate this download page with the 11 version selected.



Then download the selected installer and install it by double-clicking it. Correct Java installation can be verified by typing **java –version** on a command terminal window.

NOTE
- Install Java versions 11, 17, or 21 (LTS) and not the most recent one;
- Select the JDK FX package;
- Select the appropriate 64-bit architecture and operating system for your computer.

2.3 Simulator versions

The simulator itself is a single file Java Archive (with a ".jar" extension). Its current version is 2.3.0.

Java is backward compatible, which means that the simulator should equally run on Java versions 11, 17, and 21. The same may be not exactly true in what graphics and multimedia libraries are concerned, however.

Therefore, there are three versions of the simulator, each compiled for Java version 11, 17, and 21. These versions use a multimedia library with a version number that also increases and requires increasing Java version numbers.

The names of the simulator jar files include the Java version and the version of the simulator:

- simulator-j11-v2.3.0.jar
- simulator-j17-v2.3.0.jar
- simulator-j21-v2.3.0.jar

Use the jar with the highest java version that is not higher than the version of the Java runtime environment installed on your computer.

Further versions may be available in the future as the Java technology evolves.

2.4 Running the simulator

The most practical way is just to double click the simulator jar file, as any executable application.

If double clicking the simulator jar file does not open its Design mode interface, open a command terminal window and execute a **java –jar *pathOfSimulatorFile*** command, in which *pathOfSimulatorFile* must be replaced by the concrete path (including the ".jar" extension) of the simulator jar file.

If it works, then you must select the Java 11 (or 17, or 21) version as the default application to open jar files, so that the simulator can be run by double clicking it.

Also, verify that the installation directory of Java 11 (or 17, or 21) is listed firstmost in the system environment variable PATH.

If it still does not work, verify that the Java runtime environment is correctly installed by executing the command **java –version** in the terminal window. The installed Java version should be reported.

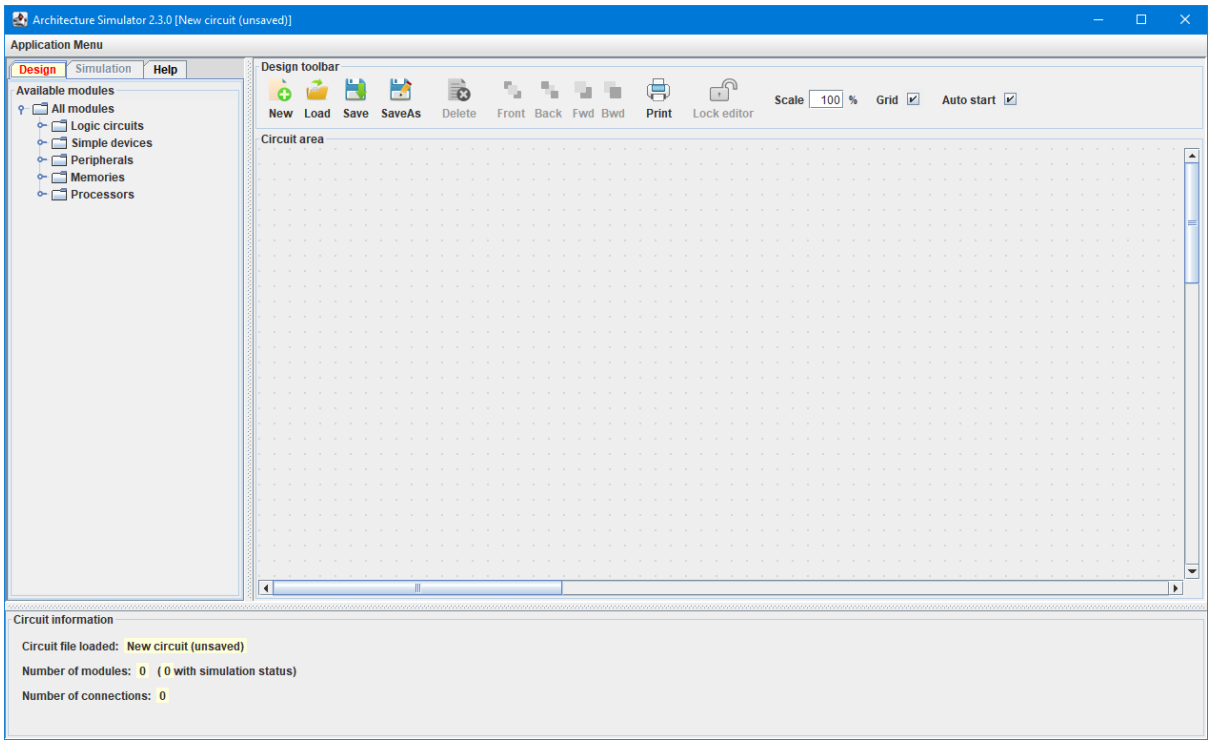
Even if double clicking works, running the simulator from the command line terminal has also the advantage of getting any error messages that might occur if some problems arise with the Java installation or if the simulator does not work properly.

Eventual simulator bugs may be reported to jose.delgado@tecnico.ulisboa.pt.

3 Editing digital circuits

3.1 The Design mode interface

The simulator starts in Design mode, showing the circuit edited in the previous run. If no circuit has been defined yet, it shows an empty circuit, as depicted in the following figure.



The Design mode window has 4 component groups:

- **Available modules**. This is a tree-shaped list of the modules available in the simulator, organized into 5 categories. These can be expanded by clicking the symbol at each category's line;
- **Design toolbar**, offering the following commands (all buttons display what they do by hovering the mouse cursor over it):
  - **New**, to change from a previous circuit to an empty one;
  - **Load**, **Save** and **SaveAs**, to load a previously saved circuit and to save any changes made to the current circuit. By default, circuit files are saved with the ".cir" extension;
  - **Delete**, to eliminate a module or connection (which must be selected first);
  - **Front**, **Backward**, **Forward**, and **Backward**. Four commands that act on the selected modules or connections to change their viewing position relative to the user. This is important particularly in complex circuits, to bring to the front the most important elements, such as modules in front of connections;
  - **Print**, which prints the circuit to pdf or to a physical printer;
  - **Lock editor**, a toggle button as a safety guard against accidental changes to the circuit when not editing it;

- **Scale**, allowing to enlarge or to shrink the circuit for a better viewing perspective (10% to 500%). The scale can be changed by specifying a new value or by pressing the CTRL key (or Command Key, in mac computers) and rolling the mouse's wheel;
- **Grid**, allowing to turn on or off a grid of small dots to ease the alignment of circuit elements;
- **Auto start**. If checked, the circuit starts running its simulation immediately upon selecting the Simulation mode;

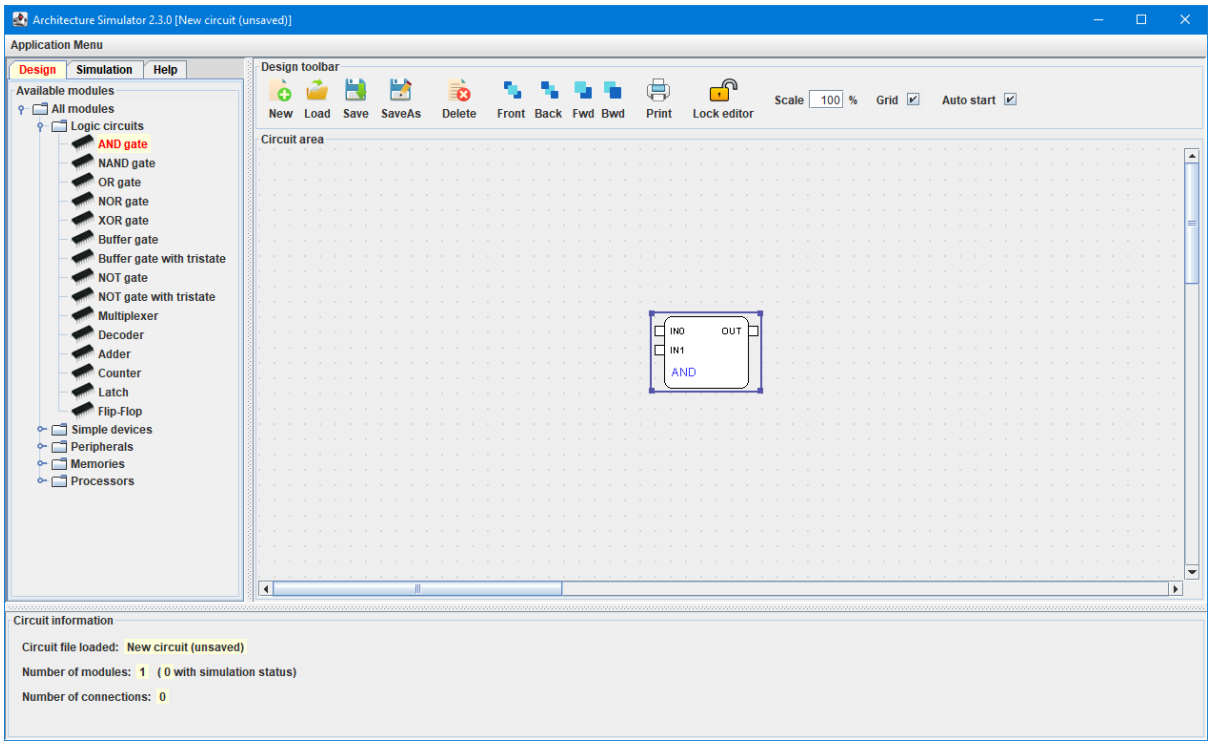
- **Circuit area**, in which modules and connections are placed and edited;
- **Circuit information**. Provides information on the circuit currently loaded.

The vertical and horizontal bars dividing these component groups can be dragged to change the relative areas of these component groups.

3.2 Adding modules to the circuit

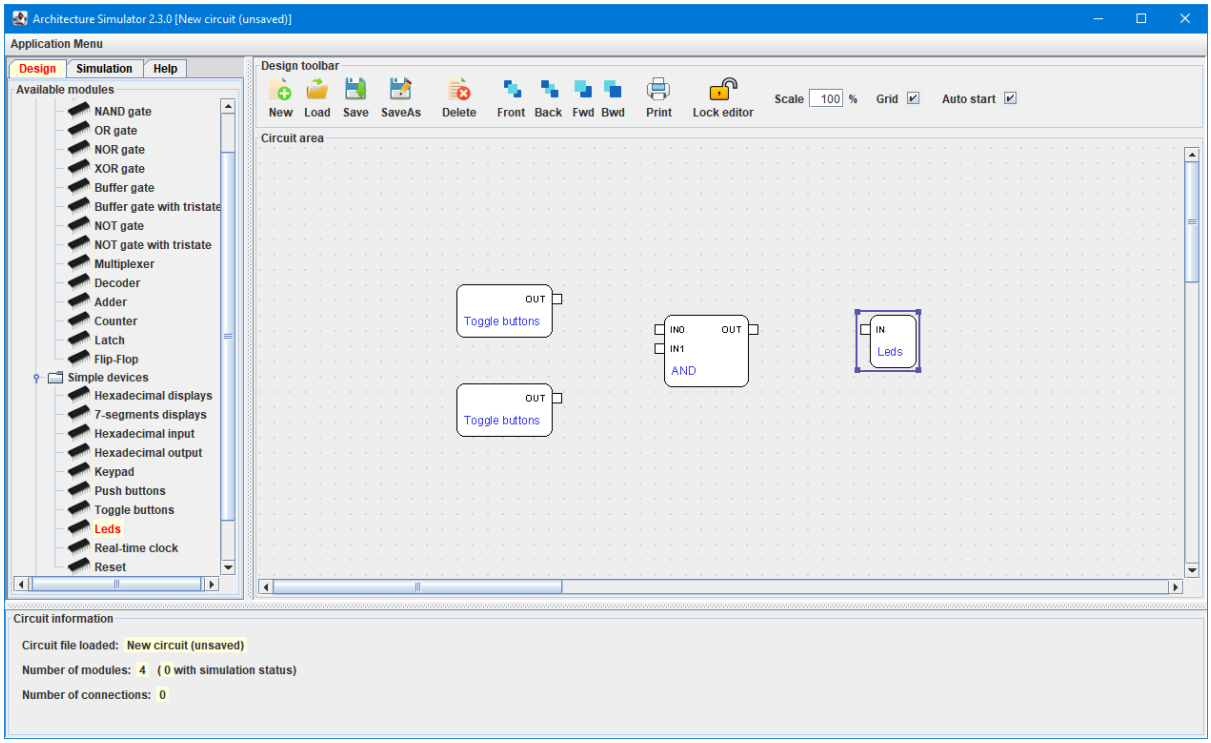
The first thing to do is to open the category (by clicking the symbol) where the desired module is defined. Then click the module and, without releasing, drag (a chip icon accompanies the cursor) and drop it wherever it should be placed in the circuit. Editing must be unlocked.

The module appears as a rectangle with named pins and also its name, as shown below.



After adding it to the circuit, a module can be moved by clicking and dragging it to a new position.

Now more modules can be added by repeating the process, as the following figure illustrates. The last element created is the one selected, but any other module can be selected by clicking it. To select more than one, press CTRL (Command key in mac computers) and then, without releasing the key, click each of the modules to select. To select all elements, press CTRL a (or CTRL A).

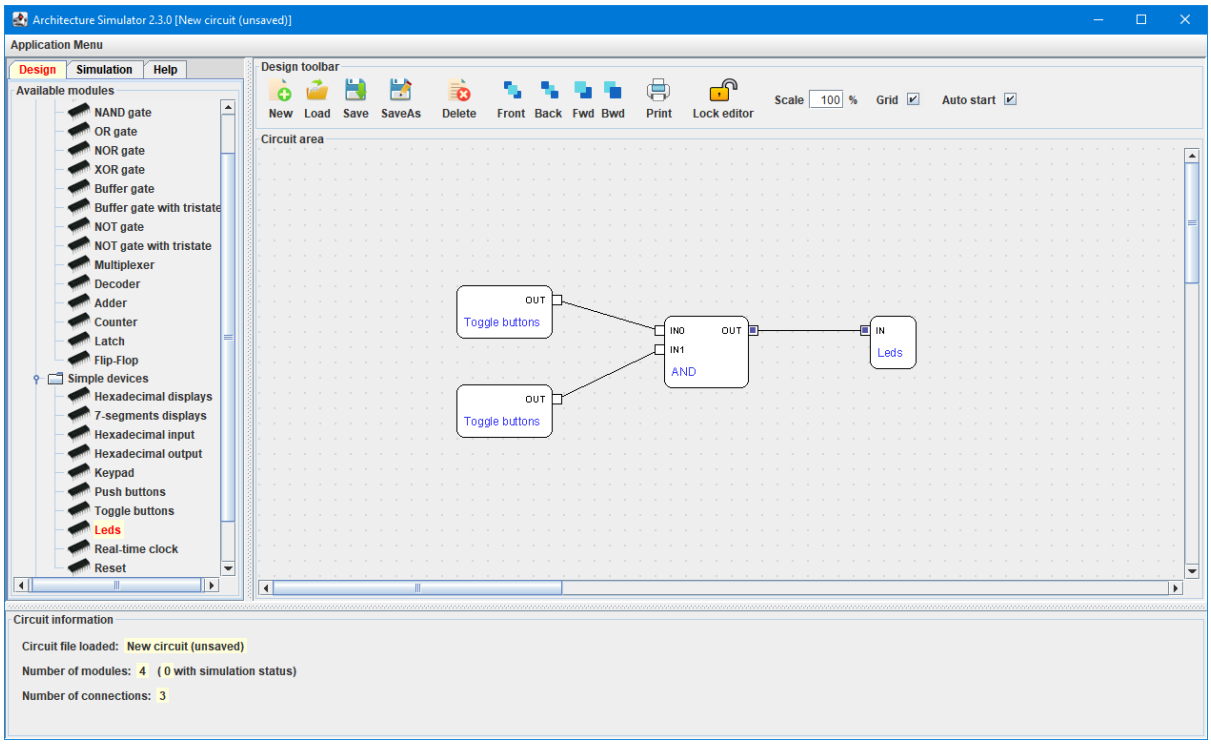


3.3 Adding connections to the circuit

A circuit is built by creating modules and then drawing connections between their pins.

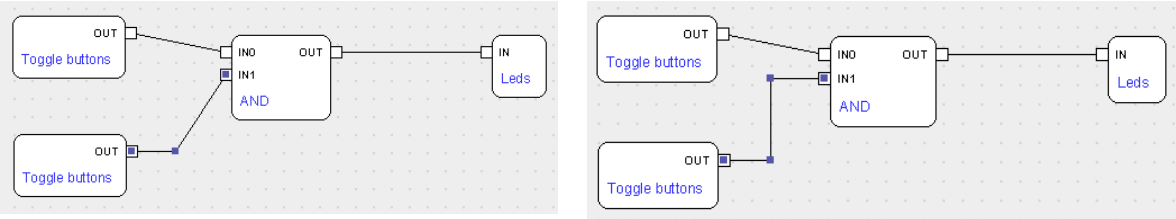
There is no need to create all modules before creating connections. As soon as the pins to interconnect are visible, a connection between them can be created (even in the same module).

Click one pin and drag to another pin, releasing the mouse over that pin. A connection is drawn, as illustrated below. Repeat for the other connections. The last one created is selected.



A connection is represented by a wire, but it can be several bits wide. Its width is automatic and depends on the width in bits of the pins that it interconnects (it is as wide as required by the pins).

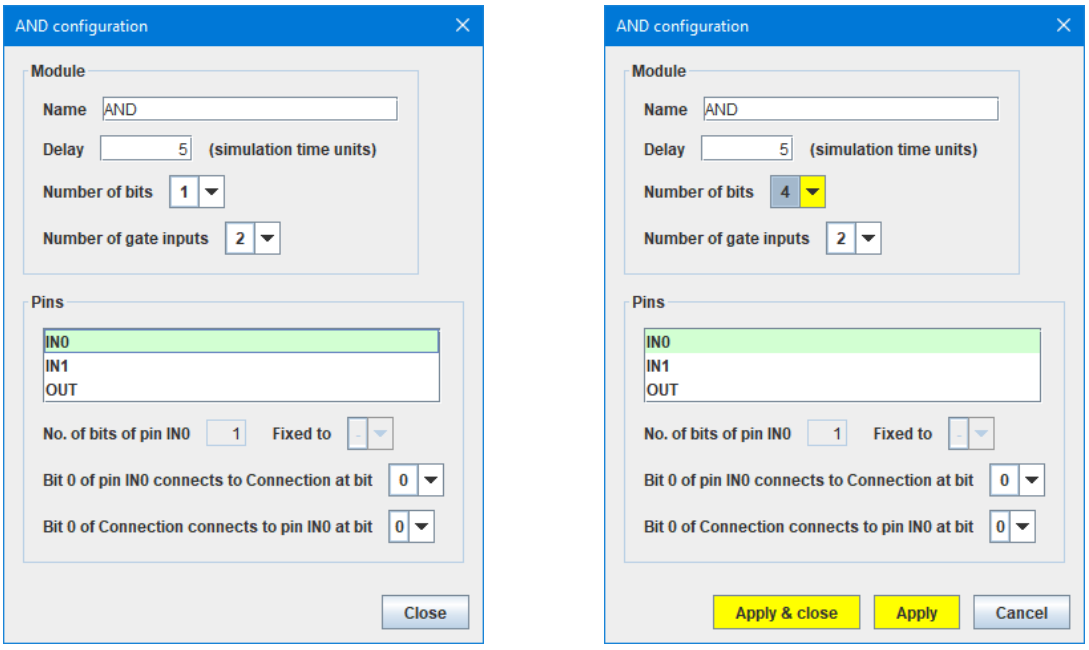
To improve the layout, connections can be broken up in horizontal and vertical segments. Click some point in a connection and drag. This will create a breaking point, becoming two line segments instead of just one. Then do the same in the diagonal segment and drag to produce two more segments, one vertical and the other horizontal, as illustrated below.



3.4 Configuring modules and their pins in Design mode

All modules support some form of configuration in Design mode. Some of the modules also support configuration in Simulation mode. The details for all the modules are described in section 6.

To configure a module in Design mode, double click it. A window appears with the parameters of the module, which can be configured. For example, the configuration window of the AND module is:

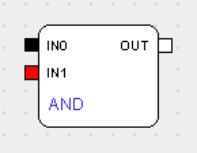


Each module has a specific configuration window, although the name of the module and the configuration of the pins are supported by all modules. If some parameter is changed (e.g., the number of bits of the AND gate, in this example), it appears in yellow and the **Apply & close**, **Apply**, and **Cancel** buttons appear. The first two make the changes effective and the Cancel button returns to the previous configuration.

The name allows several modules of the same type to be distinguished. The Pins group lists the pins of the module. Selecting one allows to:

- See its width in bits (information only);

- If it is an input pin that is not connected to a connection, its value can be fixed to 0 or to 1 (only in this case the **Fixed to** option becomes active). If fixed to 0, the pin appears black, and, if fixed to 1, it appears red. This is illustrated in the following figure:



- Specify the misalignment between a pin and the connection. Usually, the pin and the connection have the same bit width, in which case each bit of the pin connects to the corresponding bit in the connection, starting at bit order 0. However, in some cases, when pins of different bit widths are interconnected by the same connection, the pin may connect (from its bit 0) to a higher order bit of the connection, or the connection may connect (from its bit 0) to a higher order bit of the pin. These situations can be configured here.

IMPORTANT – When the number of bits of a module is changed, all or some of its pins may have their pin width changed accordingly.

– When interconnecting several modules, such as in the previous example, the various modules should have their bit widths configured consistently. If the AND module is configured for 4 bits, the two buttons and the leds should also be configured for 4 bits. The connections automatically become 4-bit wide.

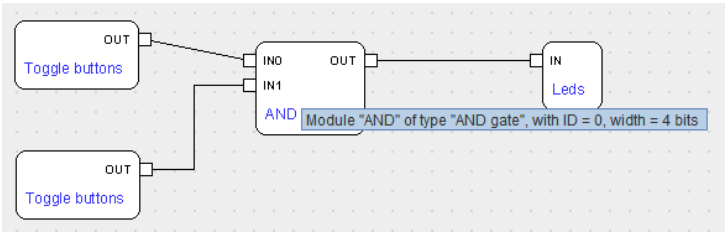
– When a connection cannot supply all the bits required by an input (e.g., connecting a 1-bit button to a 4-bit gate), the non-supplied bits are generated randomly, to reproduce what could happen in real circuits when some of the bits of an input are left unconnected and their values become highly sensitive to noise.

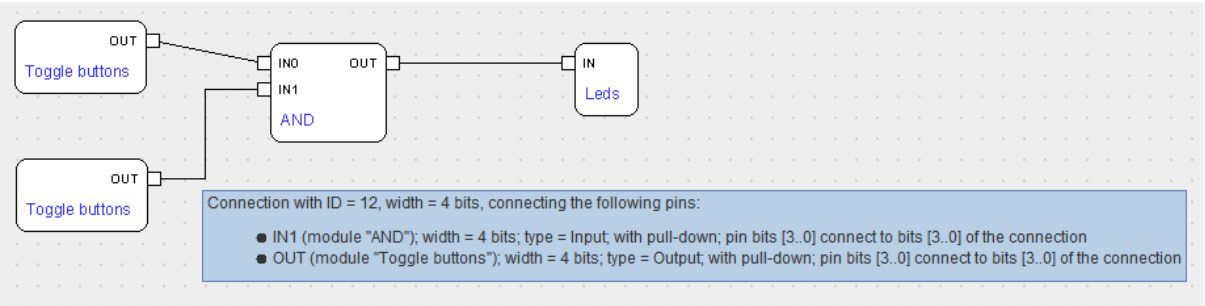
– Two output pins should not be interconnected! That can result in a value conflict. An error message will appear during simulation, if that occurs.

3.5 Obtaining information on modules and connections in Design mode

Hovering the cursor for a bit over a circuit element (module or connection) in Design time, without clicking it, displays information on that element, as illustrated by the following figures (in which all modules have already been configured for 4 bits).

Each circuit element has a unique internal ID, assigned automatically, which can be useful in some cases, particularly for error reporting.

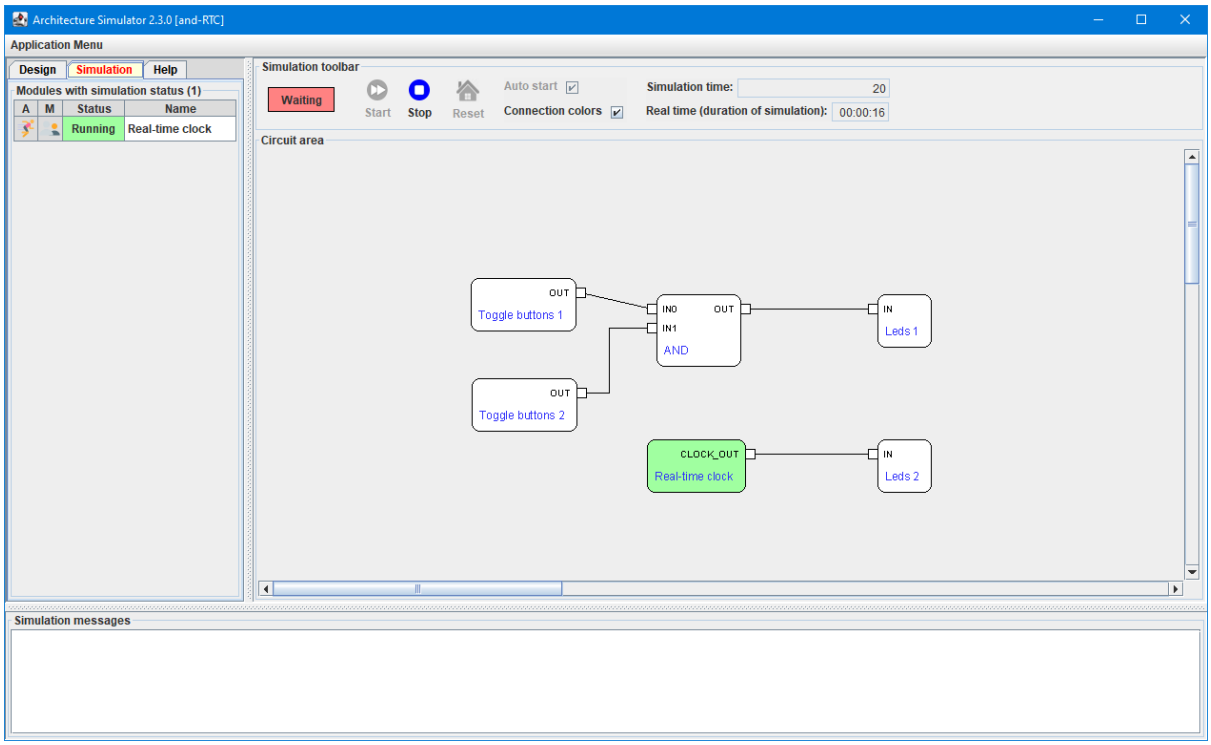



OUT
Toggle buttons
IN0
IN1
AND
OUT
IN
Leds
OUT
Toggle buttons
Connection with ID = 12, width = 4 bits, connecting the following pins:
IN1 (module "AND"); width = 4 bits; type = Input; with pull-down; pin bits [3..0] connect to bits [3..0] of the connection
OUT (module "Toggle buttons"); width = 4 bits; type = Output; with pull-down; pin bits [3..0] connect to bits [3..0] of the connection

4 Simulating digital circuits

4.1 The Simulation mode interface

The following figure depicts the simulator window upon switching to Simulation mode, by clicking the corresponding tab in the upper left corner. The example is still the one of the previous section, with modules configured for 4 bits, but now with the addition of a real-time clock module and a 1-bit led connected to it.



The Simulation mode window has 4 component groups:

- **Modules with simulation status**. Most modules just react immediately (upon a configurable simulation time delay) to changes in its input pins, i. e., they are only reactive. Only the real-time clock and the processor modules have the capacity to generate events and the ability to being paused and single-stepped, exhibiting a simulation status that can be Runnning, Paused, Stepping, and so on. These modules are listed in this group (in this example, there is only 1), with the following information:
  - **A**, an icon indicating whether this module is configured as auto-start (i.e., whether it starts running automatically upon switching to Simulation mode);
  - **M**, an icon indicating whether this module is configured as master (i.e., whether pausing or stopping it also pauses or stops the other modules; in a sense, whether it controls the simulation);
  - **Status**, indicating the current simulation status of this module. Each status has its color, with green indicating status Running. The color that fills the real-time clock module is synchronized with its status;
  - **Name** of the module, so that it can be identified;

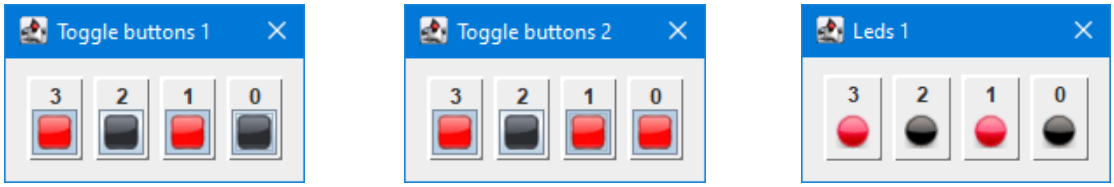
- **Simulation toolbar**, offering the following information and commands (all buttons display what they do by hovering the mouse cursor over it):
  - **Status** of the whole simulation, namely Ready, Running, Waiting and Stopped. This simulator is event-based, which means that it advances simulation when an event occurs, such as clicking a button or the clock changes its value. It is normally Waiting, switching briefly to Running when an event occurs. Stopped is reached if the Stop button is clicked and Ready is displayed if the Reset button is clicked when stopped;
  - **Start**, used to start simulation when auto-start is not checked, or upon clicking the Reset button, in which case the button is named Restart;
  - **Stop**, to stop simulation without leaving the Simulation mode (switching mode with the simulation running also stops it);
  - **Reset**, active when the simulation status is Stopped, changing it to Ready;
  - **Auto start**. If checked, the circuit starts running its simulation immediately upon selecting the Simulation mode (identical to the same setting in Design mode);
  - **Connection colors**. If checked, the connections display colors dynamically (black if the value is 0, red if not 0, and blue if the connection is in high-impedance);
  - **Simulation time**. Displays the current time, in simulation time units. This time advances only when there are simulation events and depends mainly on the delays configured for each module (simulating the delays incurred by real circuit components);
  - **Real time**. Real time duration of the simulation run, since it started. Advances once at every second, even if there are no simulation events;
- **Circuit area**. Shows the circuit, as in Design mode, but now with colors in the modules with simulation status and in connections (if the connection colors setting is checked). In Simulation mode, the circuit cannot be changed;
- **Simulation messages**. This group displays messages, if any, caused by the simulation run, namely errors reported by specific modules or by connections detecting value conflicts.

As in Design mode, the vertical and horizontal bars dividing the component groups can be dragged to change the relative areas of these component groups.

4.2 User interface of modules in Simulation mode

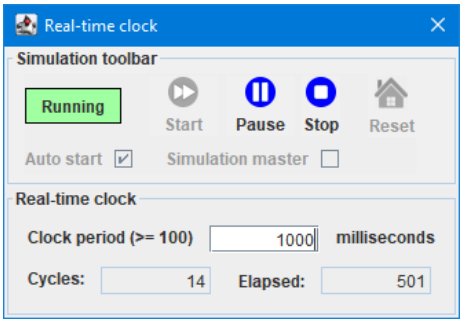
All modules have a configuration interface in Design mode (section 3.4). However, not all modules support a user interface in Simulation mode. For example, during a simulation the AND module has no information to display nor can it be configured. However, the buttons and leds need an interface to interact with the user, and the period of the real-time clock can be configured during the simulation. The details for all the modules are described in section 6.

To open the interface of a module in Simulation mode, just click it. If the module supports a simulation time interface, a window appears with information on that module and the parameters that can be configured during the simulation. For example, the simulation interfaces of the buttons and leds of the example above (with 4-bit nodules) are as follows:



Clicking each individual bit of the button windows changes its state (red is 1, black is 0). The leds window just displays the state of each bit, determined by the AND module.

The real-time clock's simulation interface is more complex, not only displaying the period (1000 milliseconds, in this example) and the current elapsed time in the current period, but also exhibiting a toolbar that shows the current state of the module (currently Running) and buttons that enable to start, pause, and stop the clock.



Each module type has its own simulation-time user interface window. The details for all the modules are described in section 6.

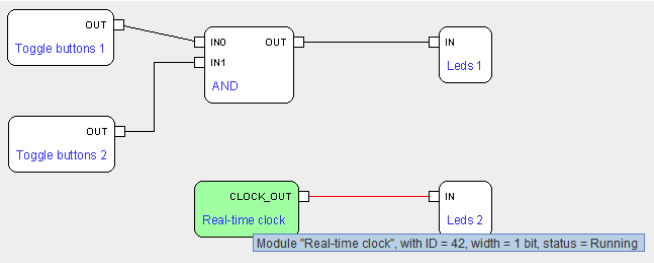
NOTE – When switching from Simulation to Design or Help modes, all opened simulation interfaces are closed. When switching to Simulation mode again, the simulation interfaces that were opened are automatically opened again;

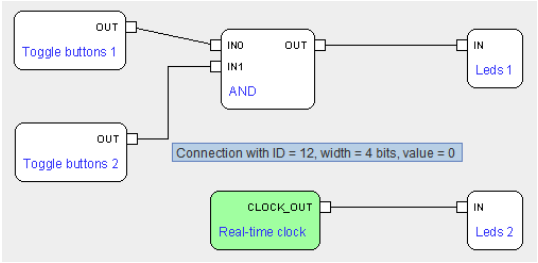
– Saving a circuit memorizes which simulation interfaces were open in Simulation mode. When loaded again, switching to Simulation mode opens the simulation interfaces that were previously opened.

4.3 Obtaining information on modules and connections in Simulation mode

Hovering the cursor for a bit over a circuit element (module or connection) in Simulation time, without clicking it, information on that element is displayed, as illustrated by the following figures. This is similar to what happens in Design mode (section 3.5).

Information on modules is identical, but including the simulation status if they support it. Information on connections includes their current value, instead of the pins they connect to.

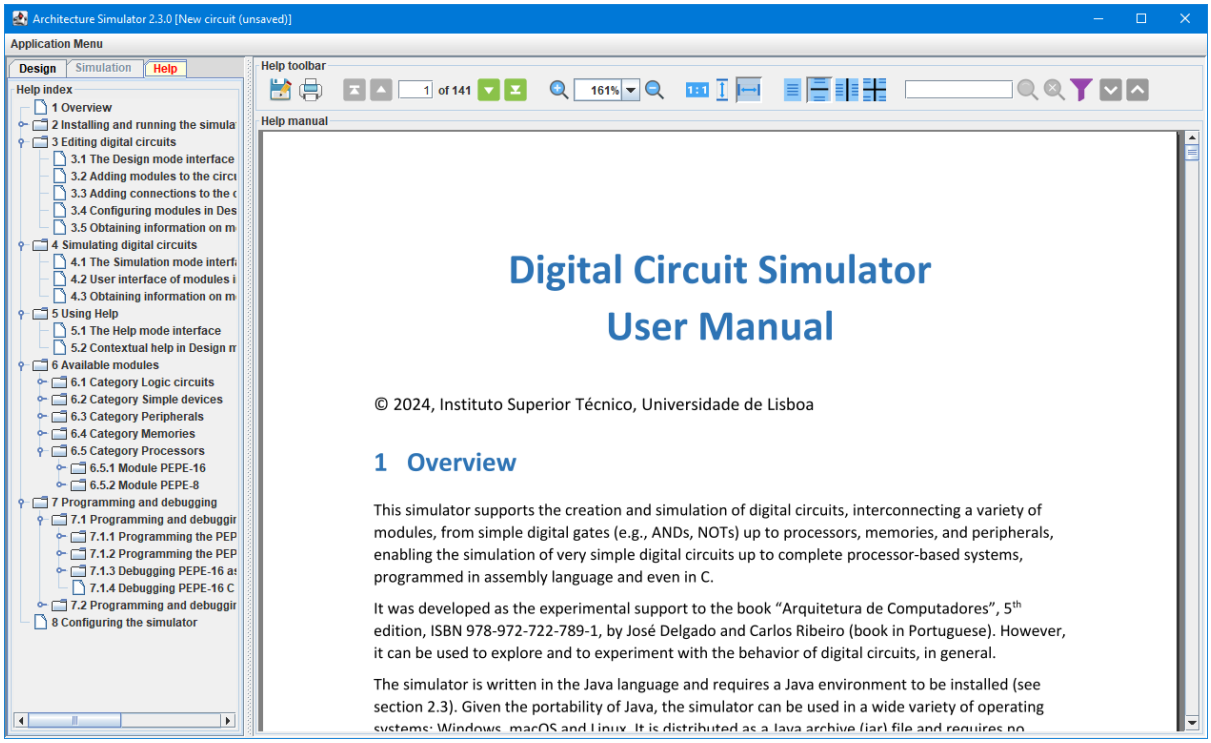




5 Using Help

5.1 The Help mode interface

The following figure depicts the simulator window upon switching to Help mode, by clicking the corresponding tab in the upper left corner. This displays help information on the simulator.



The Help mode window has 3 component groups:

- **Help index**. A tree-based representation of the headings of all the sections of the help manual. A section with sub-sections has a  symbol that can be clicked to open it and to display its sub-sections. Headings can be clicked to position the help manual at the beginning of the corresponding section;
- **Help toolbar**, offering the following information and commands (all buttons display what they do by hovering the mouse cursor over it):
  - **Save**, to save a copy of the help manual, as a PDF file;
  - **Print**, which prints the help manual (or a specific range of its pages) to a PDF file or to a physical printer;
  - **Navigation**, with green buttons to place the help manual at the first page, previous page, next page and last page. In the middle, the number of the current page and the total number of pages are displayed. The number of the current page can be changed to go directly to a page with a given number;
  - **Zoom scale**, with one button to increase zooming and another to decrease it. Between them, the current scale is displayed (10% to 2400%). It can be changed (or one of the predefined scale values can be selected) to set a given zoom scale

directly. The zoom scale can also be changed by pressing the CTRL key (or Command Key, in mac computers) and rolling the mouse's wheel;

- **Fit mode**, with 3 buttons to choose how each page of the help manual fits the display area: actual size, fit to height and fit to width;
- **Viewing mode**, with 4 modes:
  - **Single page**, in which the vertical scroll bar scrolls only over the current page. However, continuing to roll the mouse wheel, past the beginning or the end of the current page, switches to the previous or next page, respectively;
  - **Single page continuous**, in which the vertical scroll bar scrolls over all the pages, in a continuous fashion. This is the default mode;
  - **Double page**, similar to single page, but in which two pages are seen simultaneously, side by side;
  - **Double page continuous**, similar to single page continuous, but in which two pages are seen simultaneously, side by side. Given the continuous scroll, parts of four pages can usually be seen;
- **Search text**, supporting finding specific text in the help manual. This part of the help toolbar has the following components:
  - **Search text box**, in which the text to search can be specified;
  - **Search**, a button to perform the search. Search can also be performed by pressing the Enter key after specifying the text in the search box. The results are highlighted in the help manual, which is positioned in the first result;
  - **Clear search**, a button to clear the results (clears highlighted text and the search text box);
  - **Search filter**, allowing to specify how the search is to be performed. Text case can be considered or ignored. The search text can match only whole words or parts of words, and the text search box can be interpreted as a regular expression (pattern) instead of the actual text to search. The whole word and regular expression filter settings are mutually exclusive. The rules for regular expressions are those of Java and are too complex to be described here (but information on them can be easily obtained by searching online);
  - **Search results navigation**, with two buttons to show the next and previous match results. In addition, if a search produces match results, an indication is provided on the number of matches found and the number of the match result currently displayed;

- **Help manual**. Displays the help manual contents.

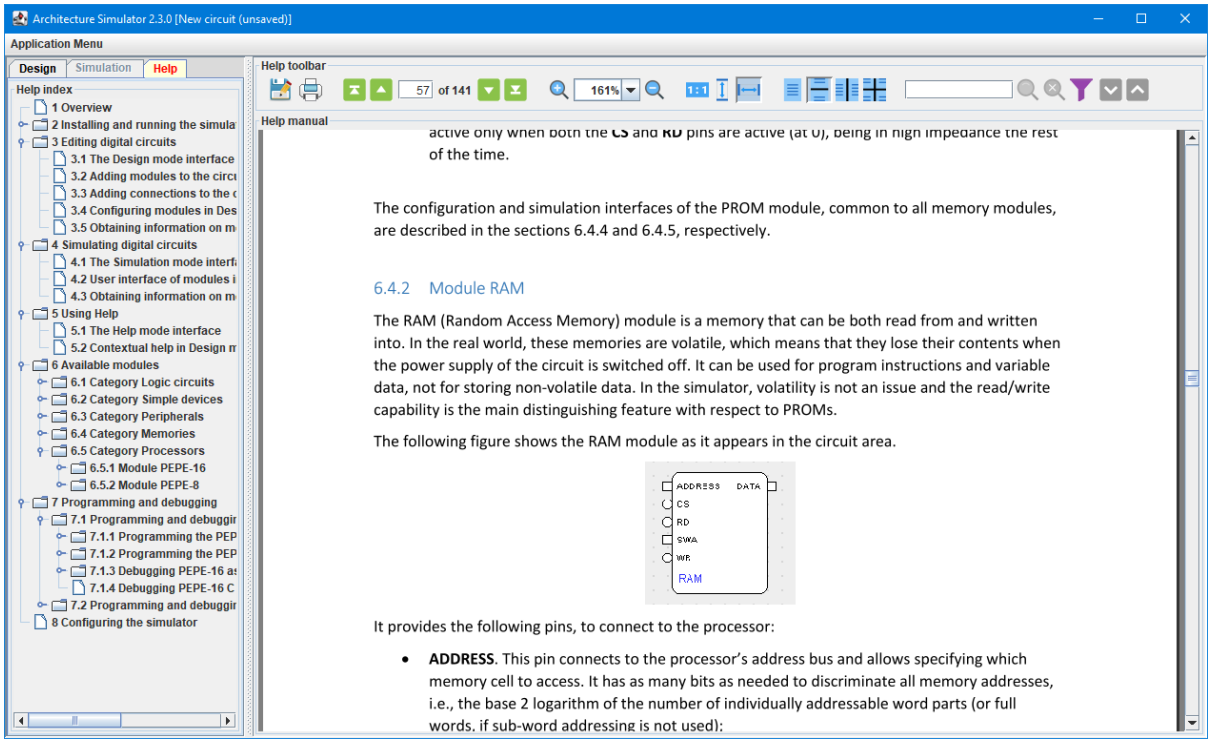
The vertical bar dividing the help index group from the other component groups can be dragged to change the relative areas of these component groups.

Changing from Simulation mode to Help mode stops the simulation, if it is running.

5.2 Contextual help in Design mode

The simulator supports contextual help on modules while in Design mode. Double clicking a module, in the module tree of the Available modules, switches from Design to Help mode and positions the help manual so that information on that module becomes visible.

The following figure shows the result of double clicking the RAM memory module in Design mode.



6 Available modules

The following sections provide detailed information on the modules supported by the simulator, organized in the categories and in the order visible in Design mode, in the Available modules component group.

6.1 Category Logic circuits

This category includes basic digital circuit modules, both combinational and sequential, and each one of them can be configured in Design mode.

None of these modules has simulation interface, since during a simulation they have no internal information to show and no need to interact with the user. All the interaction with them is done through their pins.

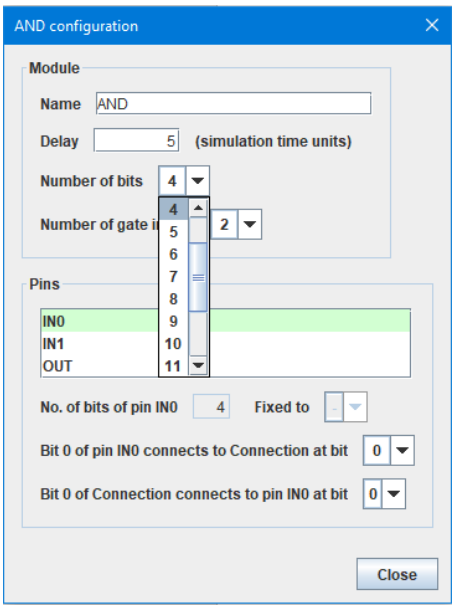
6.1.1 Module AND gate

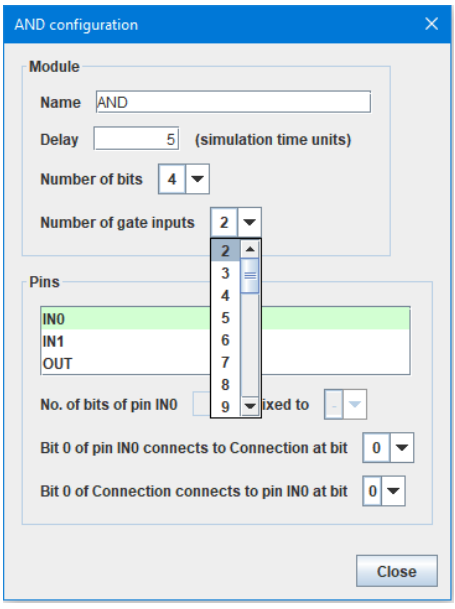
The AND gate module is a basic digital logic gate that implements logical conjunction (∧). By default, it is a 2-input, 1-bit gate. Its output is 1 only when all inputs are also 1.

It can be configured in two main ways, using the configuration interface (opened by double clicking the AND gate module in Design mode):

- **Number of bits** (1 to 16) in each input and in the output pin. The bits order N in each of the inputs are ANDed and the result yields bit order N in the output (1 only if the bits order N in all the inputs are 1);
- **Number of gate inputs** (2 to 32), indicating how many connections are to be ANDed (each of which can be several bits wide).

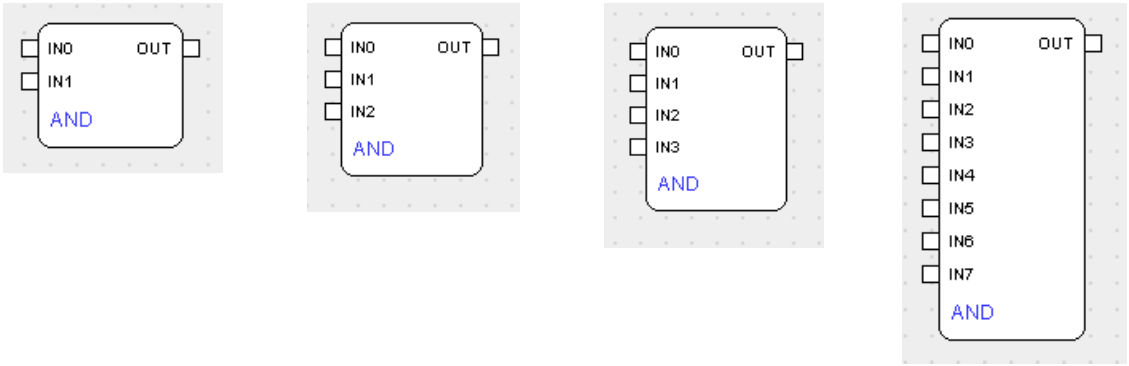
The following figures illustrate how to configure these values, by clicking the arrow in the corresponding dropdown box.





Changing the number of bits does not change the circuit visually, since all connections are represented by a wire connected to pins, in a bit-width independent way.

However, the number of inputs can easily be seen. The following figure shows four AND module configurations, with 2, 3, 4 and 8 inputs (up to 32 inputs). Note that all inputs must always be connected to a connection, or fixed to 0 or 1.



The configuration interface also allows configuring:

- **Name** of the module;
- **Simulation delay**, in simulation time units. This allows to simulate the propagation delays of real world logical gates;
- **Pins**, as described in section 3.4.

6.1.2 Module NAND gate

The NAND gate module is similar to the AND gate module, with the difference that it performs the negation of the conjunction operation, meaning that (for a 1-bit gate) the output is 0 only when all the inputs are 1.

See the description of AND gate module (section 6.1.1) for configuration details.

6.1.3 Module OR gate

The OR gate module is similar to the AND gate module, with the difference that it performs the logical disjunction operation ($\vee$), meaning that (for a 1-bit gate) the output is 1 only when at least one of the inputs is 1.

See the description of AND gate module (section 6.1.1) for configuration details.

6.1.4 Module NOR gate

The NOR gate module is similar to the OR gate module, with the difference that it performs the negation of the disjunction operation, meaning that (for a 1-bit gate) the output is 0 only when at least one of the inputs is 1.

See the description of AND gate module (section 6.1.1) for configuration details.

6.1.5 Module XOR gate

The XOR gate module is similar to the OR gate module, with the difference that it performs the exclusive disjunction operation (⊕), meaning that (for a 1-bit gate) the output is 1 only when just one of the inputs is 1.

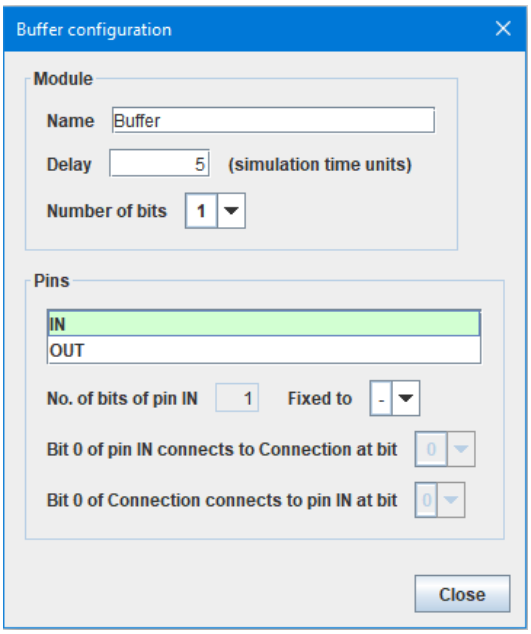
See the description of AND gate module (section 6.1.1) for configuration details.

6.1.6 Module Buffer gate

The buffer gate module has just one input and the output repeats the input value:

Its basic usefulness in a simulator is in separating connections in which an output connect to many inputs, making the layout simpler. In real world circuits, its main use is to increase the fanout, i.e., the number of inputs that may connect to an output, providing more power to drive the capacitances of all the inputs than what a normal output is able to provide. This is not an issue here.

The number of bits of the input and output can be configured. The configuration window is identical to that of the AND gate module (section 6.1.1), with the absence of the configuration of the number of inputs.

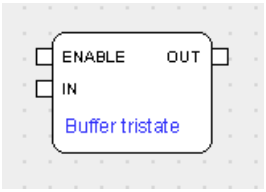


6.1.7 Module Buffer gate with tristate

The buffer gate with tristate module is similar to the buffer gate module (section 6.1.6), with the difference that the output can be disconnected, meaning that it has the capability of not enforcing a value to the output connection, independently of the input value.

Output pins that can enforce a 0 or 1, or enforce no value (as if it were disconnected from the connection) are designated tristate pins, since they can have these three states. The third state is usually designated *high impedance* or *high-Z*.

There is a pin (**ENABLE**) to control whether to enforce a value (0 or 1, depending on the input) or to not enforce a value at all. This enables several outputs to be interconnected without conflict, as long as only one of them is enabled and the others are in high impedance.



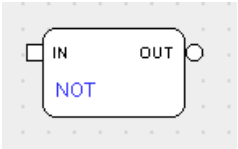
ENABLE:

- 0 – output is disabled
- 1 – output enforces the value present at the input (IN pin)

The configuration window is similar to that of the buffer gate module (section 6.1.6), with the addition of the **ENABLE** pin.

6.1.8 Module NOT gate

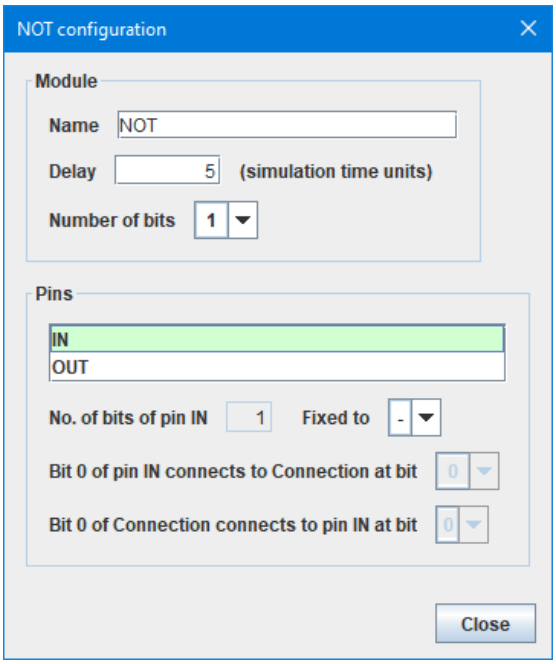
The NOT gate module is similar to the buffer gate module (section 6.1.6), with the difference that the output produces the logical negation of the input value: it is 1 when the input is 0 and it is 0 when the input is 1.



Note the output pin, a circle instead of a square, providing an indication of the negation function.

A NAND gate, for example, is equivalent to an AND gate followed by a NOT gate.

The number of bits of the input and output can be configured. The configuration window is identical to that of the AND gate module (section 6.1.1), with the configuration of the number of inputs.

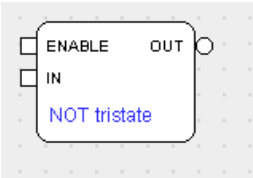


6.1.9 Module NOT gate with tristate

The NOT gate with tristate module is similar to the NOT gate module (section 6.1.8), with the difference that the output can be disconnected, meaning that it has the capability of not enforcing a value to the output connection, independently of the input value.

Output pins that can enforce a 0 or 1, or enforce no value (as if it were disconnected from the connection) are designated tristate pins, since they can have these three states. The third state is usually designated *high impedance* or *high-Z*.

There is a pin (**ENABLE**) to control whether to enforce a value (0 or 1, depending on the input) or to not enforce a value at all. This enables several outputs to be interconnected without conflict, as long as only one of them is enabled and the others are in high impedance.



ENABLE:

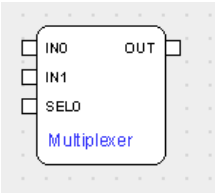
- 0 – output is disabled
- 1 – output enforces the value present at the input (IN pin)

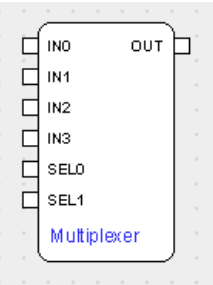
The configuration window is similar to that of the NOT gate module (section 6.1.8), with the addition of the **ENABLE** pin.

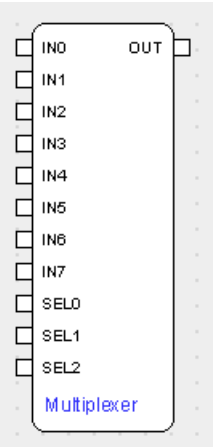
6.1.10 Module Multiplexer

The multiplexer module has one or more selection pins, several inputs and one output. The output reflects the value of the input selected by the combined value of the selection pins.

The number of inputs is the power of 2 of the number of selection pins, which can configured between 1 and 5. The number of inputs will thus vary between 2 and 32. The following figures illustrate three possible configurations, with 1, 2 and 3 selection pins.

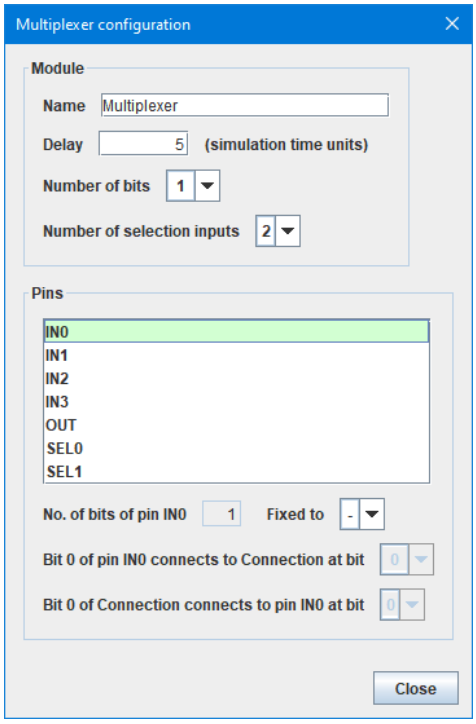






The **IN** pins are the inputs and the **SEL** pins are the selection pins. If all selection pins are 0, the first input (**IN0**) is selected and its value appears at the output. If all selection pins are 1, the last input is selected. Intermediate combinations of the selection pins select the corresponding input.

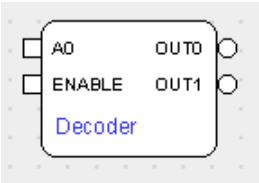
The following figure illustrates the configuration window of the multiplexer module, in which what is configurable is the number of selection input pins (1 to 5), which in turn determines the number of inputs (2 to 32).

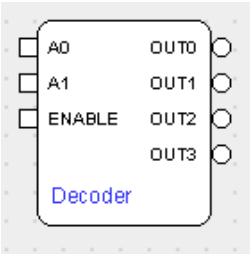


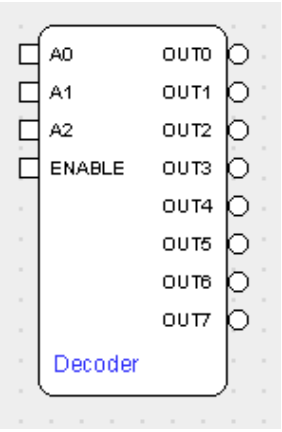
6.1.11 Module Decoder

The decoder module has one or more address pins, several outputs and one **ENABLE** pin. All outputs enforce 1, except one that enforces 0. That output is chosen by the address pins, but only if the **ENABLE** pin is 1 (if 0, all outputs enforce 1).

The number of outputs is the power of 2 of the number of address pins, which can configured between 1 and 5. The number of outputs will thus vary between 2 and 32. The following figures illustrate three possible configurations, with 1, 2 and 3 address pins. Note that output pins are circles, to emphasize that the addressed output is active at 0.



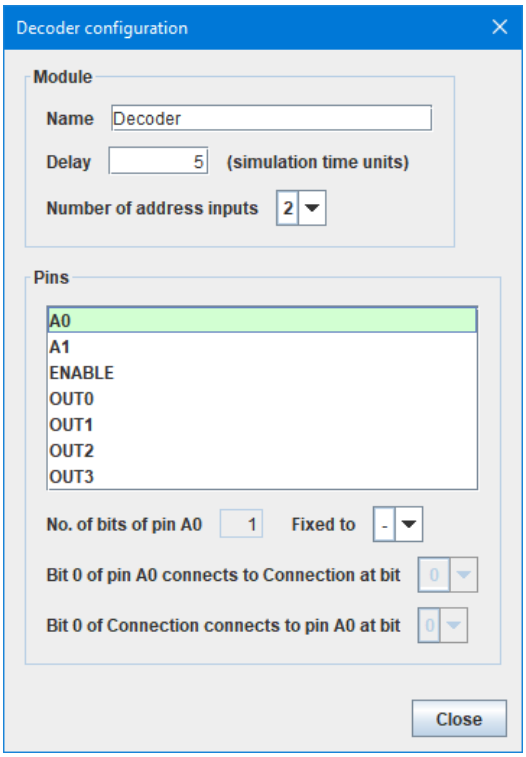




If all address pins are 0, the first output (**OUT0**) becomes 0 and the others stay at 1. If all address pins are 1, the last output is set to 0, with all the others at 1. Intermediate combinations of the address pins select the corresponding output.

Decoders are very useful in addressing schemes of processor-based systems, to discriminate between system devices (memories and peripherals).

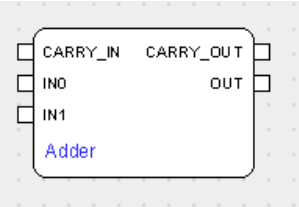
The following figure illustrates the configuration window of the decoder module, in which what is configurable is the number of address input pins (1 to 5), which in turn determines the number of outputs (2 to 32).



6.1.12 Module Adder

The adder module implements the addition function between 2 or more (up to 32) inputs, including carry. The bit width N of each input and of the output can be configured from 1 to 16.

The following figure depicts an adder module with 2 inputs (and the description of the pins):



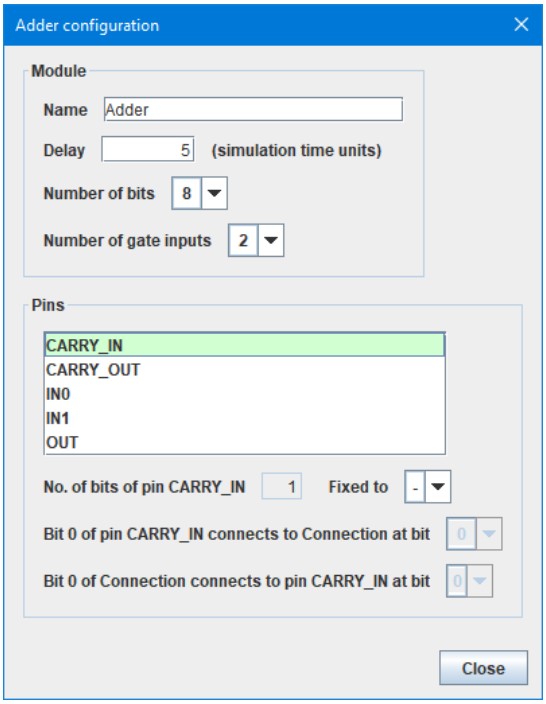
- **CARRY_IN** (1 bit) – Carry from a previous adder (0 if none)
- **IN0** to **IN1** (N bits) – The values to add (operands)
- **CARRY_OUT** (1 bit) – Carry for the next adder (unconnected if none)
- **OUT** (N bits) – The result of adding the operands and **CARRY_IN**

The **CARRY_IN** pin behaves as an additional operand, with a value of 0 or 1.

The carry pins allow several adders to be cascaded, supporting the addition of numbers with more bits than the bit width of the adders. However, the most common case is to configure the number of bits of the adder module to the bit width of the numbers to be added.

The adder module works with both unsigned and signed operands, as long as the latter use the two's complement number representation.

The following figure illustrates the configuration window of the adder module, in Design mode, configured for adding two 8-bit wide numbers:



6.1.13 Module Counter

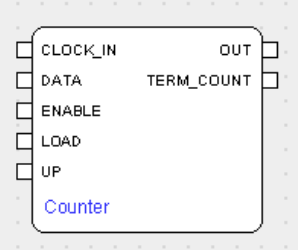
Unlike the previous modules, that are combinational circuits (their output values depend only on the values of the inputs), the counter module is a sequential circuit (the output values depend on both the values of the inputs and on the previous internal state).

Sequential circuits usually have a 1-bit clock pin, which upon a transition (0 to 1, typically) causes the circuit to change from one state to the next.

The internal state of the counter module consists of a number, to which it adds 1 on each clock transitions, therefore counting clocks. The number of bits N of this module is configurable and determines the counting range (between 0 and 2^N-1). When the maximum value is reached, the next value is 0 and the counting range repeats.

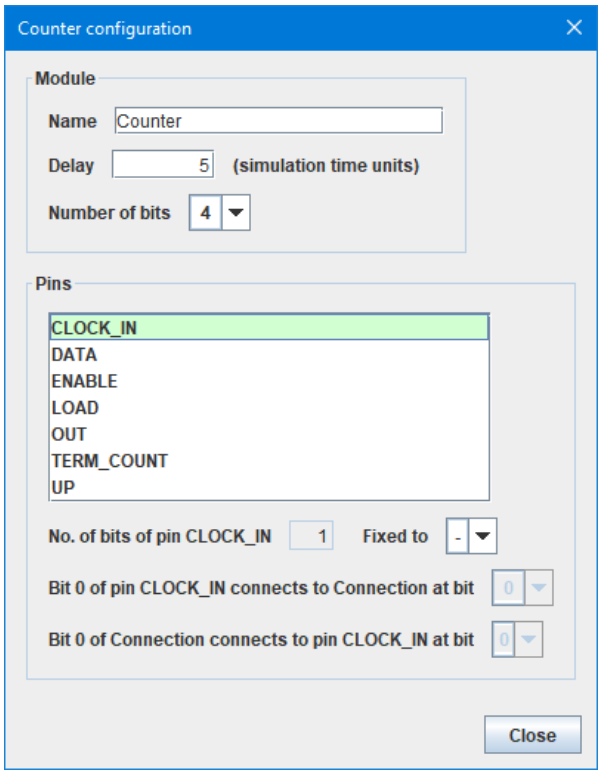
The counter module also supports loading a given value, from which to start counting, and counting downwards (subtracting 1) instead of counting upwards (increasing 1).

The following figure depicts the counter module, and the description of its pins:



- **CLOCK_IN** (1 bit) – The clock signal
- **DATA** (N bits) – The value to load as a start counting value
- **ENABLE** (1 bit) – The module counts only when this pin is 1
- **LOAD** (1 bit) – The value of DATA is stored in the module when this pin is 1. To count, this pin must be 0
- **UP** (1 bit) – The module counts upwards if this pin is 1
- **OUT** (N bits) – The new count value after the clock
- **TERM_COUNT** (1 bit) – Becomes 1 when the OUT pin reaches the maximum or minimum value (depending on the UP pin)

The following figure illustrates the configuration window of the counter module, in Design mode. In this example, the counter module has been configured with 4 bits, which means that it will count between 0 and 15.



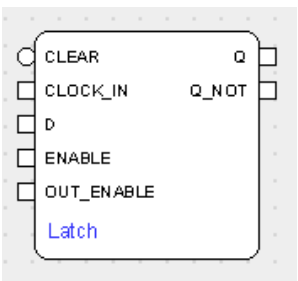
6.1.14 Module Latch

The latch module is a memory. It can store (memorize) N bits simultaneously, with N configurable in Design mode.

It has a clock pin, just like the counter module, but instead of producing a new value on each clock transition, it just stores the value present at its input. After that, the input value can change without affecting the output value, which keeps exhibiting the memorized value.

The output pins have tristate capability (see section 6.1.7 for a description of tristate outputs).

The following figure depicts the latch module, and the description of its pins:

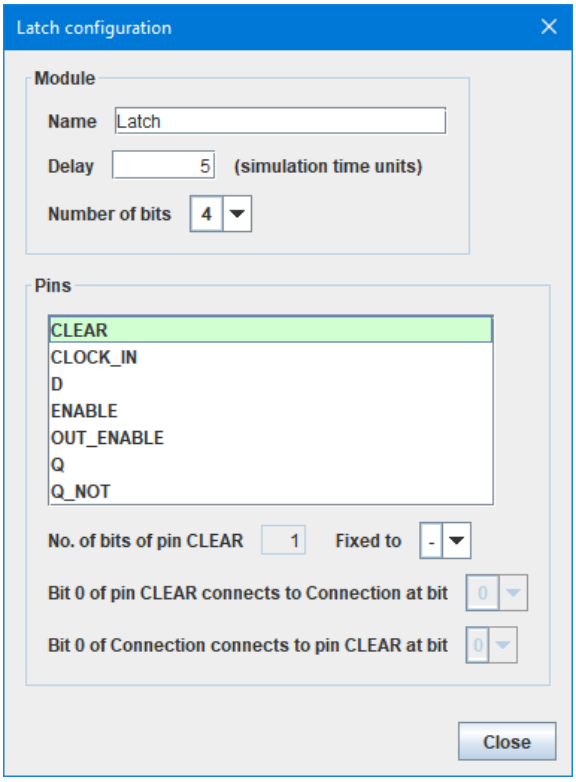


- **CLEAR** (1 bit) – The output value is reset to 0 when this pin is 0
- **CLOCK_IN** (1 bit) – The clock signal
- **D** (N bits) – The value to store (memorize)
- **ENABLE** (1 bit) – The module memorizes only when this pin is 1
- **OUT_ENABLE** (1 bit) – The outputs Q and Q_NOT is in high impedance when this pin is 0
- **Q** (N bits) – Output value
- **Q_NOT** (N bits) – The logical negation of the output value (bit by bit)

Memorization of values by the latch module works like this (assuming that both **CLEAR** and **ENABLE** pins are 1), with **Q_NOT** always the logical negation of **Q**:

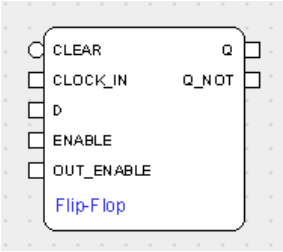
- When the **CLOCK_IN** pin is 1, the latch module is transparent, i.e., the value is not memorized yet and the output **Q** exhibits the same value as the **D** pin (if it changes its value, the output **Q** pin will change as well);
- When the **CLOCK_IN** pin is 0, the latch module has memorized the last value of **D** before the **CLOCK_IN** pin changed to 0. That value appears at the output **Q** and will not change, even if the **D** pin changes its value, as long as the **CLOCK_IN** pin remains at 0.

The following figure illustrates the configuration window of the latch module, in Design mode. In this example, the latch module has been configured with 4 bits, which means that it will memorize 4 bits at once.



6.1.15 Module Flip-Flop

The flip-flop gate module is similar to the latch gate module (section 6.1.14), with the difference that the output pins (**Q** and **Q_NOT**) will memorize the value of the input (**D**) only in the instant of the transition of the clock pin (**CLOCK_IN**) from 0 to 1. When the clock pin value is either 0 or 1, or changes from 1 to 0, the value of the output pins will not change, even if the input pin changes.



6.2 Category Simple devices

This category includes simple devices that are useful to interact with the user or to generate events.

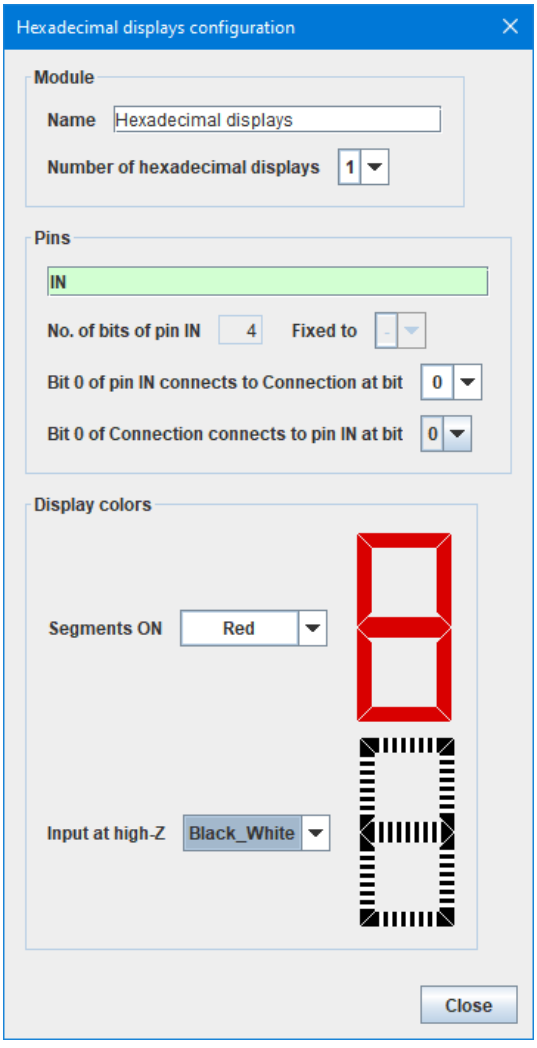
In addition to the Design mode configuration interface, most of these modules have a Simulation mode interface to support the interaction with the user.

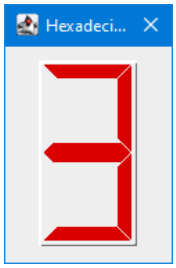
6.2.1 Module Hexadecimal displays

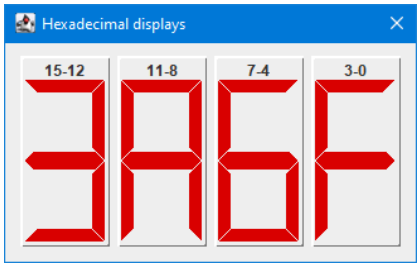
The hexadecimal displays module have just one input pin (**IN**) and show the corresponding value in hexadecimal (0..9, A..F) in up to four hexadecimal displays, during simulation.

The following figures illustrate the configuration interface of a hexadecimal displays module (left side) and the simulation interface (right side), configured with 1, 2, and 4 displays. 3 displays are also possible. When there is more than one display, the bit range of each display is shown.

During simulation, when the value of the input changes, the value displayed changes immediately.







The configuration interface allows specifying the following aspects, specific to this module:

- **Number of hexadecimal displays** (1 to 4). Each display requires 4 bits (hexadecimal values), and the number of bits of the IN pin is adjusted accordingly (4, 8, 12 or 16 bits);
- **Color of the display segments** that are active, depending on the value displayed in each segment. Non-active segments are transparent;
- **Color of high impedance state**, affecting all the segments of each displays. This state is reached when the pin is left unconnected or is connected to an output pin that has tristate capability and is in high impedance.

See section 6.2.3 for additional information on this module.

6.2.2 Module 7-segments displays

The 7-segments displays module is similar to the hexadecimal displays (section 6.2.1), with the difference that all 7 segments can be individually controlled, instead of using just 4 bits per display.

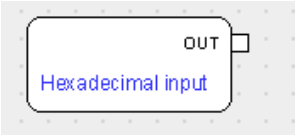
The advantage is that more segment combinations are available (not just the 16 hexadecimal digits), enabling some letters and symbols to be drawn in the simulation interface. The following figures present some examples.

The disadvantage is that, occupying 7 bits per display, the module supports just one or two displays. If more are needed, several modules must be used and their simulation interfaces placed side to side on the computer screen.

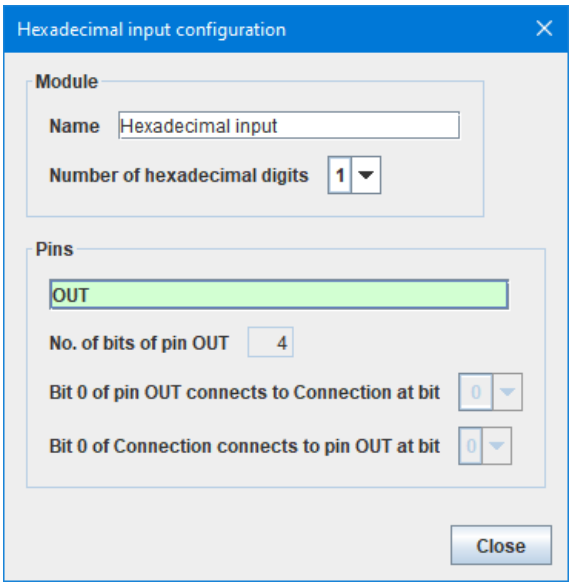
6.2.3 Module Hexadecimal input

The hexadecimal input module can be used by the user to supply, during simulation and using the computer keyboard, up to 4 hexadecimal digits (16 bits). It is equivalent to using a toggle buttons module (section 6.2.7), but instead of having to click each button icon for each individual bit, each hexadecimal digit generates 4 bits at once.

The hexadecimal input module has just one output pin (**OUT**), with a bit width configurable in multiples of 4 bits (4, 8, 12, or 16), corresponding to 1, 2, 3, or 4 hexadecimal digits.

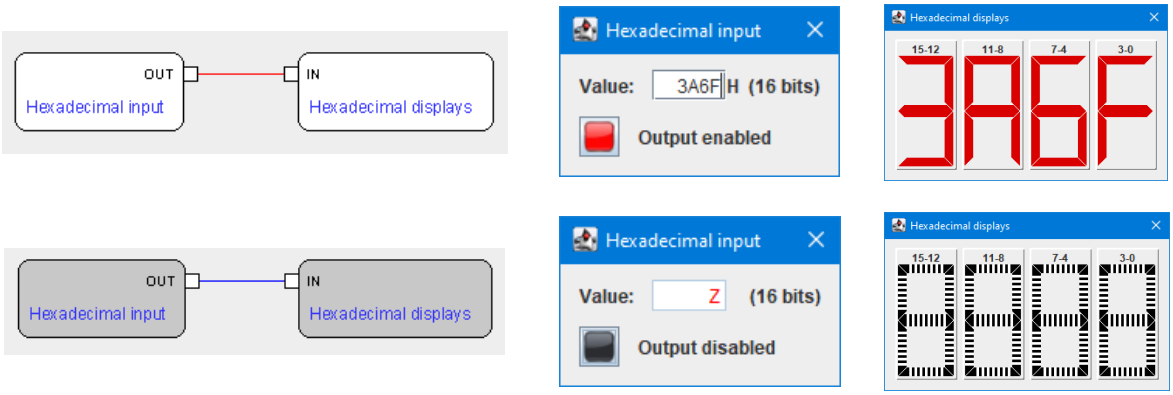


The configuration window, illustrated by the following figure, is simple and the only parameter specific to this module that can be configured is the number of hexadecimal digits (1 to 4).



The following figures illustrate the simulation window and behavior of the hexadecimal input module, connected to a hexadecimal displays module (section 6.2.1), both configured for 4 hexadecimal digits.

Circuit representation | **Hexadecimal input** | **Hexadecimal displays**



The leftmost figures show the representation of the modules as they appear in the circuit area, during simulation. The middle and rightmost figures represent the simulation windows of the hexadecimal input module and hexadecimal displays modules, respectively.

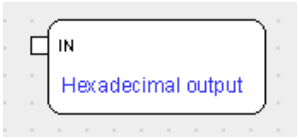
The hexadecimal input module's simulation window has a button that allows the output pin to be set into a high impedance state:

- The topmost figures represent the normal situation (output pin enabled, or active);
- The bottommost figures represent the situation in which the output pin is disabled (in high impedance). Note the color of the connection (blue), the color of both modules (gray) and of the hexadecimal displays, stating the high impedance state, and the Z indication in the hexadecimal output's simulation window.

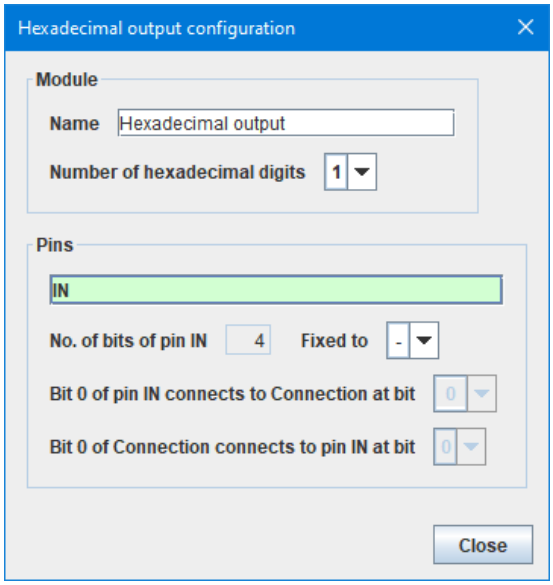
6.2.4 Module Hexadecimal output

The hexadecimal output module can be used by the user to visualize, during simulation, the value of a connection in hexadecimal format, with up to 4 hexadecimal digits (16 bits). It is equivalent to using a leds module (section 6.2.8), but instead of each led icon showing the value of each individual bit, each hexadecimal digit shows 4 bits at once.

The hexadecimal output module has just one input pin (**IN**), with a bit width configurable in multiples of 4 bits (4, 8, 12, or 16), corresponding to 1, 2, 3, or 4 hexadecimal digits.

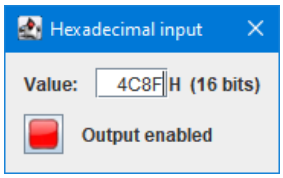


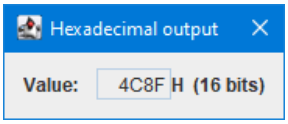
The configuration window, illustrated by the following figure, is simple and the only parameter specific to this module that can be configured is the number of hexadecimal digits (1 to 4).

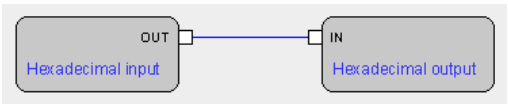


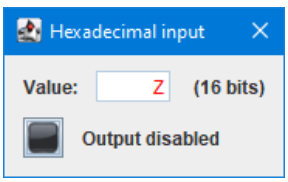
The following figures illustrate the simulation window and behavior of the hexadecimal output module, connected to a hexadecimal input module (section 6.2.3), just to provide a value, with both modules configured for 4 hexadecimal digits.

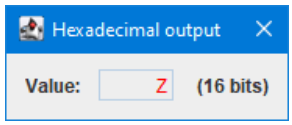
Circuit representation | **Hexadecimal input** | **Hexadecimal output**











The leftmost figures show the representation of the modules as they appear in the circuit area, during simulation. The middle and rightmost figures represent the simulation windows of the hexadecimal input module and hexadecimal output module, respectively.

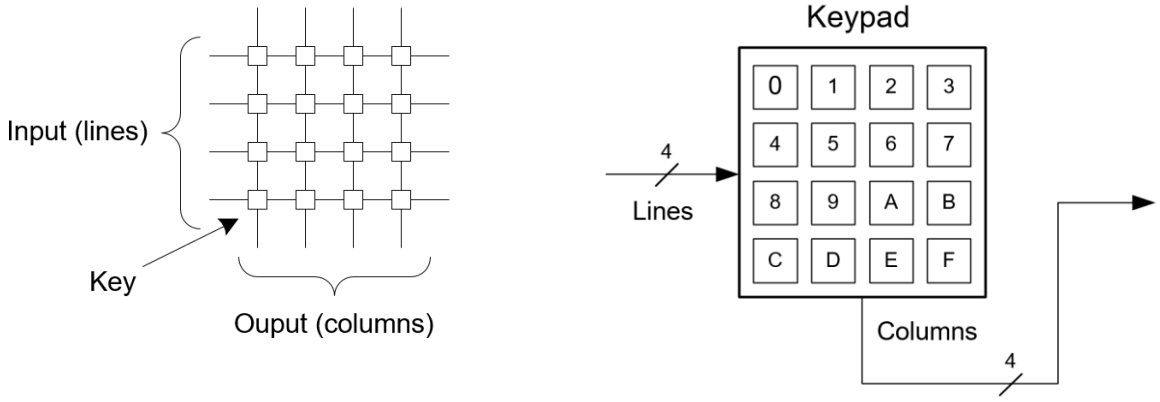
The hexadecimal input module's simulation window has a button that allows its output pin to be set into a high impedance state, illustrating how the hexadecimal output module behaves when connected to a high impedance connection:

- The topmost figures represent the normal situation (connection with a specific value);
- The bottommost figures represent the situation in which the connection is in high impedance). Note the color of the connection (blue), the color of both modules (gray), and the Z indication in the hexadecimal output's simulation window, stating the high impedance state of its input pin.

6.2.5 Module Keypad

The keypad module simulates a physical keypad, with several keys that the user can click (emulating a finger pressing the key). Typically, this is used in processor-based systems, in which the processor scans the keypad to detect any pressed key.

Physically, a real world keypad is nothing more than a set of vertical and horizontal wires, with buttons (one for each key) connecting to both one horizontal and one vertical wire. Pressing a key produces an electrical connection between the line and the column wires that connect to that key. With a 4 x 4 keypad (16 keys), for example, the typical way to organize the keys is with a 4-line, 4-column matrix, as shown in the following figures.



To detect that the user pressed a button, the processor injects a 1 in just one of the lines (0 on the others), one line at a time (hence the use of the term "scan") and reads the columns. If it reads all 0s, there is no key pressed at that line. If it reads at least one 1, at least a key has been pressed at that line and by analyzing the bit order of the 1s detected, and knowing which line has been injected with a 1, the processor is able to figure out which key has been pressed.

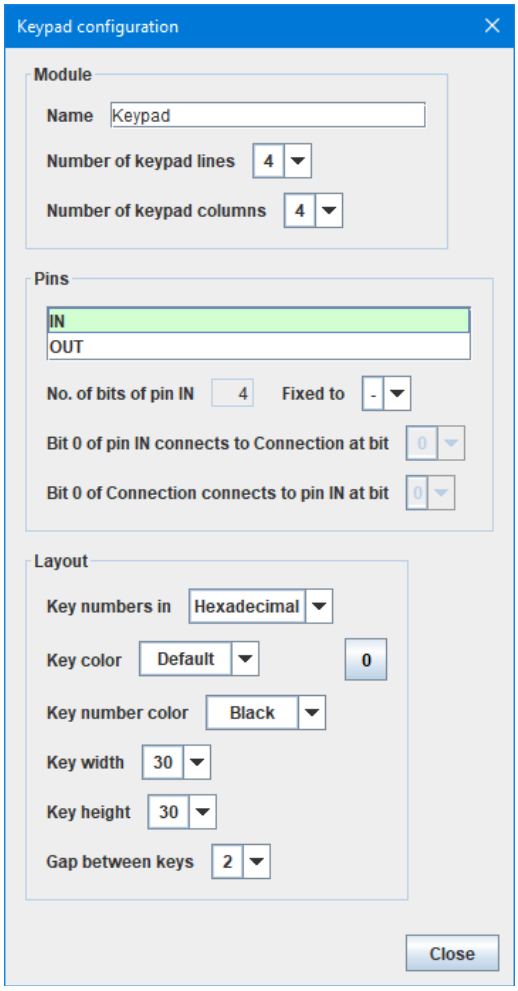
The scanning should be done repeatedly and frequently, so that pressing and releasing a key is detected immediately, at least in a human time scale.

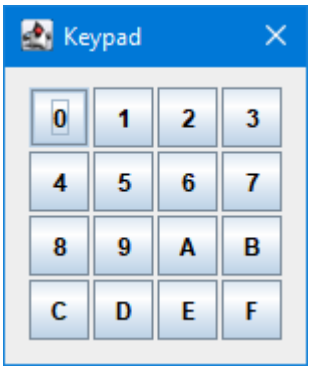
The keypad module has therefore one input (**IN**) and one output (**OUT**) pin, corresponding to the lines and columns, respectively.

The configuration interface allows specifying the number of lines and columns independently, which correspond to the number of bits of the input and output pins, respectively. It is also possible to configure several aspects of the layout, including numbering of the keys (decimal or hexadecimal), color of the keys and of their numbers, and size of the keys and of the gap between them (in pixels). A sample key is provided, so that the result of the layout configuration can be seen immediately.

The simulation interface shows a matrix of buttons, according to the configuration. These buttons can be pressed by clicking them. Only one key can thus be pressed at a time.

The following figures illustrates both interfaces, assuming a 4 x 4 keypad.





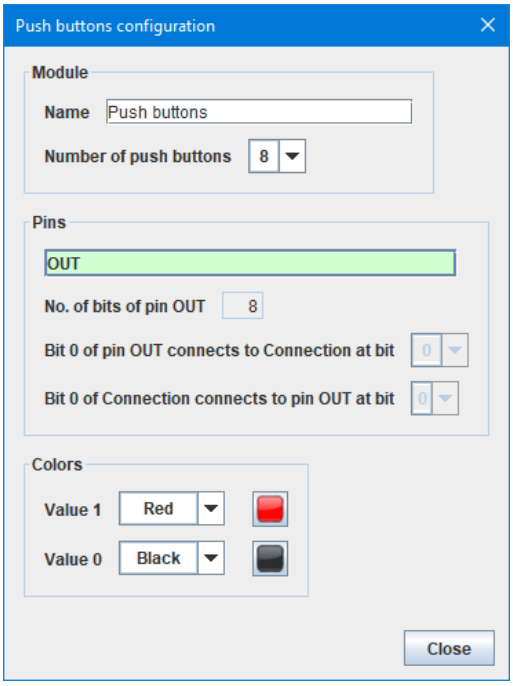
6.2.6 Module Push buttons

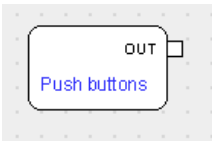
The push buttons module has just one output pin (**OUT**) and during simulation it provides a window with a group of button icons, one for each bit of the output pin, which the user can click, one at a time, to change the value of one bit in the circuit.

Clicking a button icon sets the corresponding bit to 1 in the output pin, which maintains this value until the user releases the mouse, upon which the value of the bit returns to 0 (to usual value).

Hence the name “push button”. The number of buttons is configurable (1 to 16), as well as the colors of the buttons in the 1 (pressed) and 0 (released) states.

The following figures show the configuration interface, the module representation in the circuit area and the simulation interface, illustrated for 8 buttons (each showing its bit order).





6.2.7 Module Toggle buttons

The toggle buttons module is similar to the push buttons module (section 6.2.6), with the difference that buttons toggle their corresponding bit value. Clicking a button icon toggles its state (from 0 to 1 or 1 to 0), which is maintained when the mouse is released. When clicked again, the original value is restored. Each click toggles the button state.

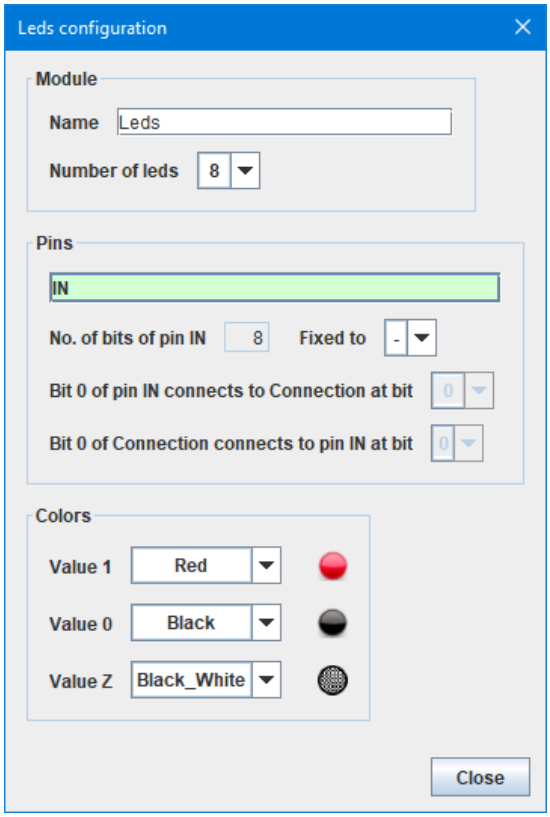
This means that, unlike in the push buttons module, several buttons can be 1 at the same time, providing more control since its new state is memorized.

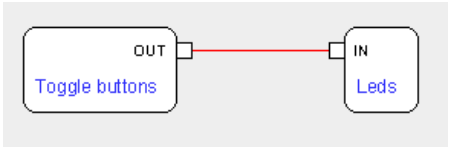
The following figure illustrates the simulation interface of the toggle buttons module, configured for 8 buttons (each showing its bit order) and several at value 1 (red, but the colors can be configured).

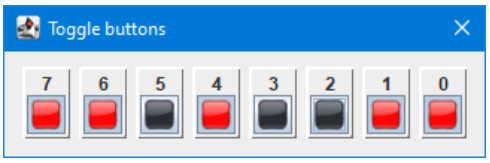
6.2.8 Module Leds

The leds module has just one input pin (**IN**). During simulation, it provides a window with a group of leds, one for each bit of the input pin, providing a means for the user to inspect visually the values of the corresponding bits.

The following figures show the configuration interface, the module representation in the circuit area (connected to a toggle buttons module, just to provide a value) and the simulation interface, illustrated for 8 leds (each showing its bit order) .







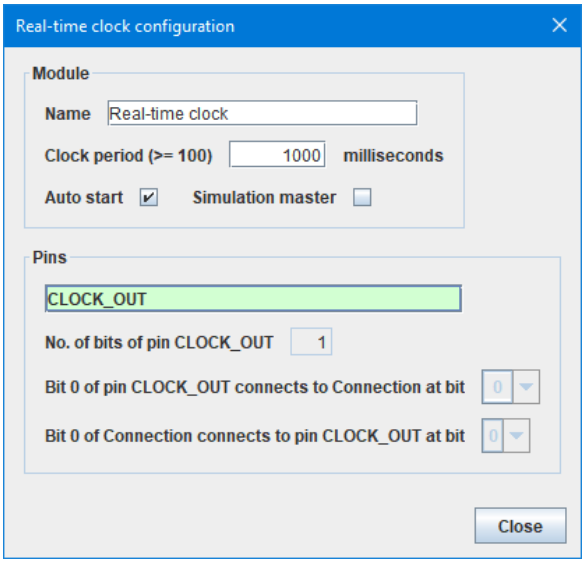
The configuration interface allows to configure not only the number of leds (1 to 16) but also their colors, for the 0, 1 and high impedance (Z) states.

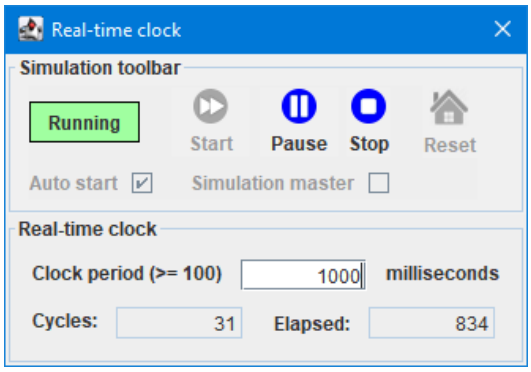
6.2.9 Module Real-time clock

The real-time clock module is an event generator. It has just one 1-bit wide output pin (**CLOCK_OUT**) that provides a square wave, oscillating between 0 and 1 with a configurable period.

It is essential in providing real time timings, since the speed at which the simulator runs depends on the processing power of the underlying computer and on the workload it undertakes at a given time. A module generating events (transitions in a bit from 0 to 1, for example) with precise timings (a clock signal) is therefore a basic requirement in any real world system.

The following figures illustrate the configuration and simulation interfaces of the real-time clock module.





In Design mode, it is possible to configure:

- The period of the clock in milliseconds (half at 0, half at 1);
- Whether the clock starts automatically upon changing to Simulation mode (if not, it must be started manually);
- Whether this module is a simulation master (if it is, pausing or stopping during simulation pauses or stops the simulation of the other modules).

In Simulation mode, this mode has a simulation status, which means that it can be Running, Paused, Stopped, or Ready, after clicking the Start, Pause, Stop and Reset buttons, respectively. With auto start, there is no need to click the Start button.

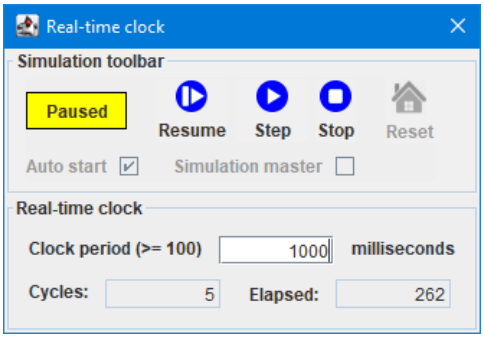
The color of the module in the circuit area reflects the color of the simulation status.

When Running or Paused, the simulation interface shows the time elapsed in the current clock period, as well as the number of cycles (periods) already executed since the last time the clock was started.

When Stopped or Ready, the simulation interface also allows configuring the auto start and simulation master settings.

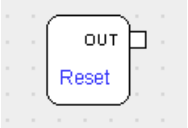
At any simulation status, the simulation interface also allows configuring the clock period.

In the Paused status, as illustrated by the following figure, it is possible to Resume or to Step, in which case only one cycle is executed, and then the clock pauses again (showing a Stepping status in the interim).

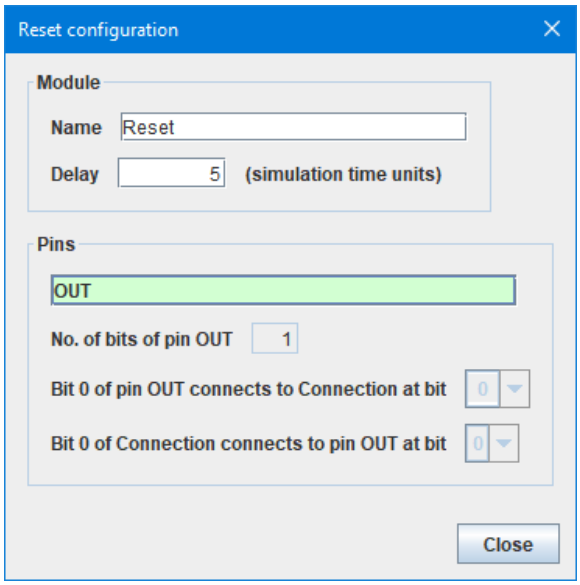


6.2.10 Module Reset

The reset module has just one output pin (**OUT**), with just 1 bit, and provides a delayed transition from 0 to 1 immediately after starting simulation, remaining at 1 for the rest of the simulation. This is a means to maintain the reset pin of a module activated for some time, allowing other modules to start their simulation first.



The configuration interface allows to configure the delay (in simulation time units), from the start of simulation until the output pin changes to 1. The reset signal is therefore active at 0 and actuates only once, at the start of the simulation.



6.3 Category Peripherals

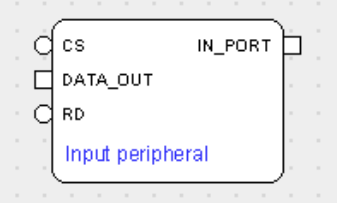
This category includes modules that are used by the processors as peripherals, to interact with the outside world, namely modules of the Simple devices category. Therefore, they include pins to connect to a processor, including addressing, data transfer and bus access control.

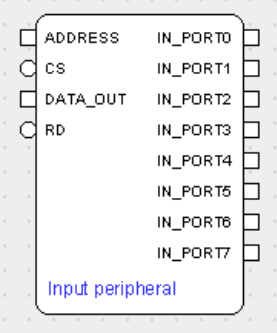
6.3.1 Module Input peripheral

The input peripheral module provides a means for a processor to read bits from the system, typically produced by user input modules in the Simple devices category (e.g., buttons, keypad). It includes pins to interface the processor and to connect to the bits to read from:

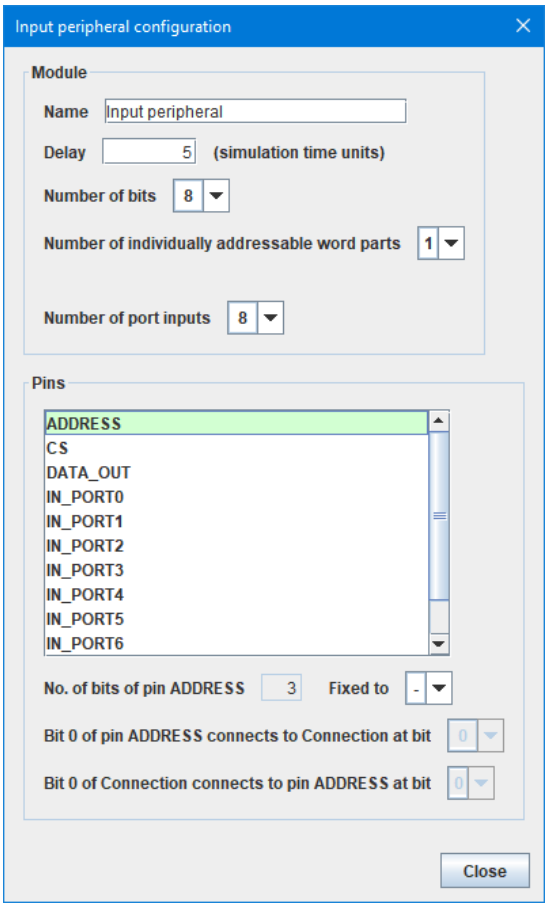
- **CS** (chip select). This pin indicates that the processor wants to access this peripheral. It is active at 0 (hence the round pin in the module representation, shown in the following figures);
- **DATA_OUT**. This is an output pin, to be read by the processor (connects to its data bus), and conveys the bits read by the peripheral;

- **RD** (read). In addition to the **CS** pin, the **RD** pin must also be active (at 0) for the processor to perform the read operation. This distinguishes read from write operations. This peripheral supports read operations only, but the processor is able to write into other devices (e.g., the output peripheral module, described in section 6.3.2);
- **IN_PORT** (input port). This pin connects to the bits to read from the system. The number of bits is configurable (1 to 16). It is also possible to configure more than one input port, up to 32, in which case there are several of these pins, each with the same name suffixed with its number, as illustrated in the following figure, and an additional pin, **ADDRESS**;
- **ADDRESS**. This pin is available only when there are several input ports defined, and allows the processor to specify which port it wants to read from. It connects to the processor's address bus.



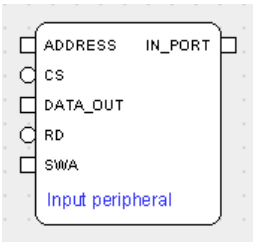


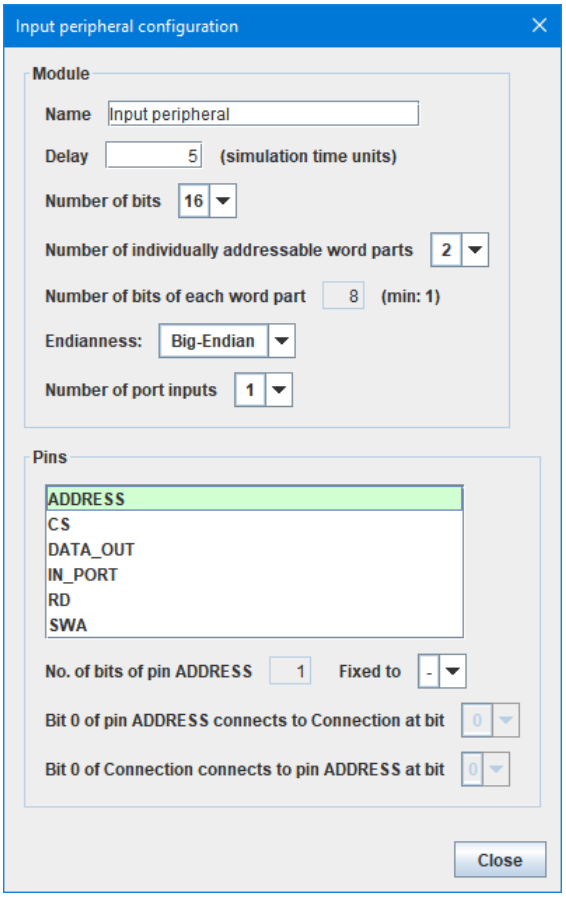
The configuration interface allows configuring the number of bits of each input port (1 to 16) and the number of input ports (1 to 32, in powers of 2).



In addition, the input peripheral module allows reading from each port all the bits at once (a full word) or, if the number of bits of the word is a power of 2, in individually addressable word parts. The number of bits of each part, and the number of word parts, must also be a power of 2. This is provided to allow a 16-bit processor, for example, to read a 16-bit input port in one single operation, or to read each of the two 8-bit halves individually. This can be important in some applications, is known as sub-word addressing, and is supported by the 16-bit processor available in this simulator, the PEPE-16 (section 6.5.1).

If the number of individually addressable word parts is configured to be higher than 1, a new pin (**SWA**) appears and the configuration interface has additional features. This is shown in the following figures, which illustrate a 16-bit input peripheral module with 2 individually addressable word parts, which means that the processor can read all 16 bits (one word) at once or each of its two halves (bytes) individually. In this case, one part has an even port address and the other has an odd port address.





The new pin is:

- **SWA** (sub-word addressing). This pin specifies how many word parts to read. This pin connects to the processor, which must provide support for sub-word addressing. In this example (16-bit ports with 2 individually addressable word parts), the **SWA** pin has only 1 bit, to specify whether to read one part (one byte) or two parts (full word).

The configuration interface shows how many bits each word part has (8 bits, in this example), and the *endianness* (referring to which part, or which end, of the word comes first in the addressing scheme, or located at the lowest address), which can be:

- **Big-Endian**: the highest order word part (the one with the highest order bits in the word) is assigned the lowest address;
- **Little-Endian**: the lowest order word part (the one with the lowest order bits in the word) is assigned the lowest address.

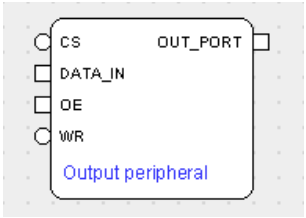
The processor that this module connects to must support sub-word addressing and have the same endianness setting. See section 6.5.1.2 for further details on sub-word addressing.

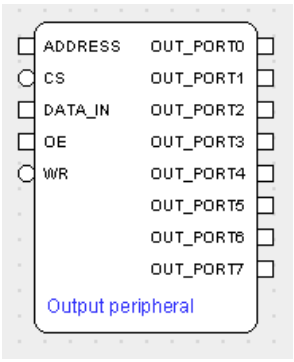
This module has no simulation interface.

6.3.2 Module Output peripheral

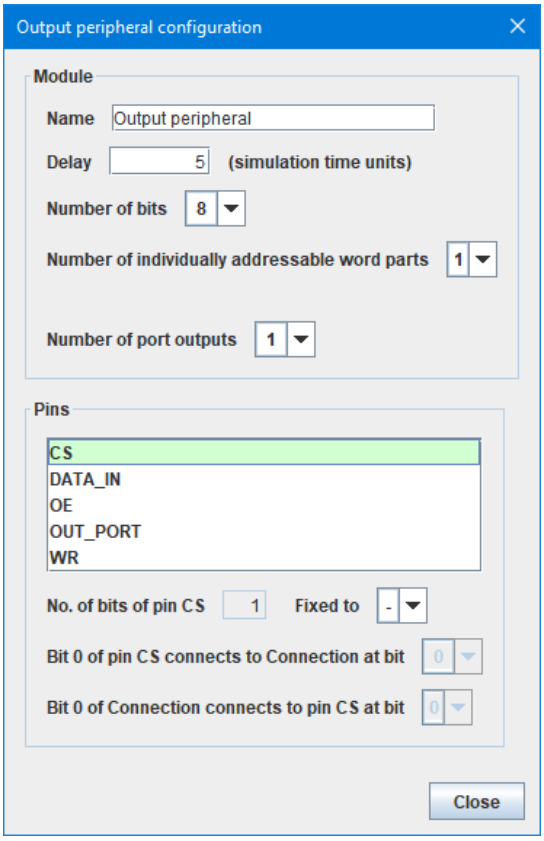
The output peripheral module provides a means for a processor to write bits into the system, typically read by visualization modules in the Simple devices category (e.g., hexadecimal displays, leds). It includes pins to interface the processor and to connect to the bits to write into:

- **CS** (chip select). This pin indicates that the processor wants to access this peripheral. It is active at 0 (hence the round pin in the module representation, shown in the following figures);
- **DATA_IN**. This is an input pin, which the processor writes to (connects to its data bus), and conveys the bits to write in the peripheral;
- **OE** (output enable). The output port pins have tristate capability and the **OE** pin controls whether the output pins are active (**OE** = 1) or in high impedance (**OE** = 0);
- **WR** (write). In addition to the **CS** pin, the **WR** pin must also be active (at 0) for the processor to perform the write operation. This distinguishes write from read operations. This peripheral supports write operations only, but the processor is able to read from other devices (e.g., the input peripheral module, described in section 6.3.1);
- **OUT_PORT** (output port). This pin connects to the bits to set in the system. The number of bits is configurable (1 to 16). It is also possible to configure more than one output port, up to 32, in which case there are several of these pins, each with the same name with its number appended, as illustrated in the following figure, and an additional pin, **ADDRESS**;
- **ADDRESS**. This pin is available only when there are several output ports defined, and allows the processor to specify which port it wants to write into. It connects to the processor's address bus.





The configuration interface allows configuring the number of bits of each output port (1 to 16) and the number of output ports (1 to 32, in powers of 2).



In addition, the output peripheral module allows writing into each port all the bits at once (a full word) or, if the number of bits of the word is a power of 2, in individually addressable word parts. The number of bits of each part, and the number of word parts, must also be a power of 2. This is provided to allow a 16-bit processor, for example, to write a 16-bit input port in one single operation, or to write each of the two 8-bit halves individually. This can be important in some applications, is known as sub-word addressing, and is supported by the 16-bit processor available in this simulator, the PEPE-16 (section 6.5.1).

If the number of individually addressable word parts is configured to be higher than 1, a new pin (**SWA**) appears and the configuration interface has additional features. This is shown in the following figures, which illustrate a 16-bit output peripheral module with 2 individually addressable word parts, which means that the processor can write all 16 bits (one word) at once or each of its two halves (bytes) individually. In this case, one part has an even port address and the other has an odd port address.

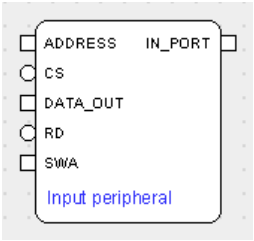
The new pin is:

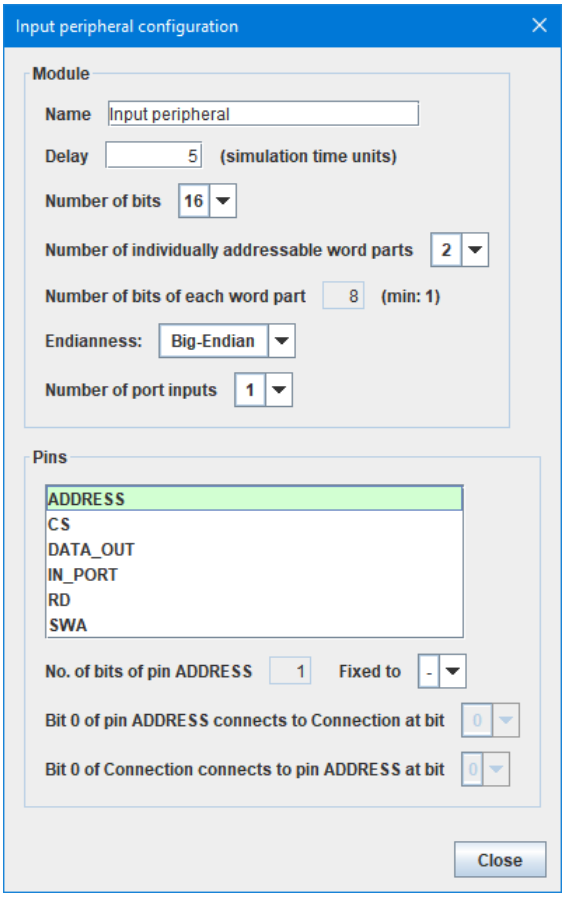
- **SWA** (sub-word addressing). This pin specifies how many word parts to write. This pin connects to the processor, which must provide support for sub-word addressing. In this example (16-bit ports with 2 individually addressable word parts), the **SWA** pin has only 1 bit, to specify whether to write one part (one byte) or two parts (full word).

The configuration interface shows how many bits each word part has (8 bits, in this example), and the *endianness* (referring to which part, or which end, of the word comes first in the addressing scheme, or located at the lowest address), which can be:

- **Big-Endian**: the highest order word part (the one with the highest order bits in the word) is assigned the lowest address;

- **Little-Endian**: the lowest order word part (the one with the lowest order bits in the word) is assigned the lowest address.





The processor that this module connects to must support sub-word addressing and have the same endianness setting. See section 6.5.1.2 for further details on sub-word addressing.

This module has no simulation interface.

6.3.3 Module MediaCenter

6.3.3.1 General description

The MediaCenter module is one of the most complex modules in the simulator and supports, in Simulation mode, a wide range of multimedia features, including video, sound effects, and pixel screens, enabling drawing and animation of pixel-based objects. It is essential to support programming of multimedia-based applications, such as the game briefly described in section 1.

This module includes:

- A memory to hold the pixels shown in the pixel screens. Pixels can be drawn by writing in this memory, which can also be read (section 6.3.3.4). However, it is easier to draw pixels by using commands (section 6.3.3.5);
- A set of input and output ports, which do not connect to external bits but rather control the functionality of the MediaCenter module, either to change its state or to obtain information on it. Each port has a different address and corresponds to a different command:

- The value written (into an output port) is an argument to a write command that changes the state of the MediaCenter module;
- The value read (from an input port) is the information on some aspect of the state of the MediaCenter module that is retrieved by a read command.

Just like the input and output peripheral modules (sections 6.3.1 and 6.3.2, respectively), the MediaCenter module supports sub-word addressing. This means that it allows accessing each memory cell or command port by transferring all the bits at once or in individually addressable word parts (the width of a word is the number of bits of each memory cell or command port), each with a number of bits that must be a power of 2. This is provided to allow a 16 bit processor, for example, to write a 16-bit output port in one single operation, or to write the two 8-bit halves individually. This can be important in some applications and is supported by the 16-bit processor available in this simulator, the PEPE-16 (section 6.5.1).

If the number of individually addressable word parts is configured to be higher than 1, a new pin (SWA) appears and the configuration interface has additional features. The following figure depicts the representation of the MediaCenter module in the circuit area, without (left) and with (right) sub-word addressing.

This module has the following pins:

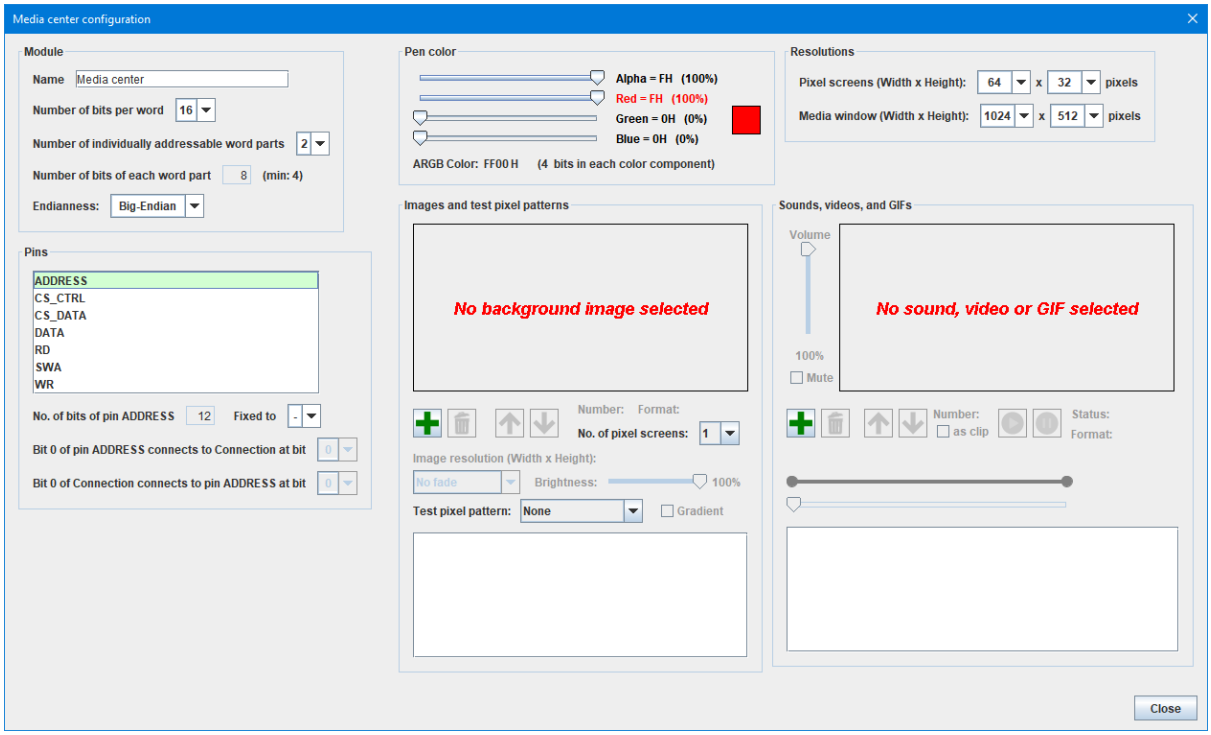
- **ADDRESS**. This pin connects to the processor's address bus and allows the processor to specify which memory cell or port (command) it wants to access;
- **CS_CTRL** (chip select for control commands). This pin indicates that the processor wants to access one of the ports of the MediaCenter module, or to execute one of its commands. It is active at 0 (hence the round pin in the module representation) and must not be active simultaneously with the **CS_DATA** pin;
- **CS_DATA** (chip select for memory data accesses). This pin indicates that the processor wants to access the internal memory of the MediaCenter module. It is active at 0 (hence the round pin in the module representation) and must not be active simultaneously with the **CS_CTRL** pin;
- **RD** (read). In addition to the **CS_CTRL** and **CS_DATA** pins, the **RD** pin must also be active (at 0) for the processor to perform a read operation. This distinguishes read from write operations, in either memory and command accesses;
- **SWA** (sub-word addressing). This pin specifies how many word parts to access and appears only when the number of individually addressable parts is greater than 1. This pin connects to the processor, which must provide support for sub-word addressing. In the example used in the following section (16-bit MediaCenter module with 2 individually addressable word parts), the **SWA** pin has only 1 bit, to specify whether to access one part (one byte) or two parts (full word). See section 6.5.1.2 for further details on sub-word addressing;

- **WR** (write). In addition to the **CS_CTRL** and **CS_DATA** pins, the **WR** pin must also be active (at 0) for the processor to perform a write operation. This distinguishes write from read operations, in either memory and command accesses;
- **DATA** (data bus). This pin connects to the data bus of the processor and is used to transfer the values to write or to read. The number of bits is configurable, but must be multiple of 4 (4, 8, or 16). This restriction stems from the fact that each data value must be wide enough to hold one pixel, and each pixel has 4 components (3 color and 1 opacity), each of which can be 1, 2, or 4 bits wide.

To access this peripheral, the processor must perform a read or a write access, specifying the address to access (in the **ADDRESS** pin). In addition, there must be an address decoding circuit to activate the adequate chip select pin (**CS_CTRL** for commands or **CS_DATA** for memory accesses).

6.3.3.2 Configuration interface

The following figure illustrates the rather complex configuration interface of the MediaCenter module, with typical settings (but others can be set). Some of these settings can be changed during simulation by using commands (see section 6.3.3.5), but others can only be changed in Design mode.



This interface includes the component groups described below, with the configurable parameters and information shown.

Module

It is possible to configure:

- **Name** of the module;

- **Number of bits per word** (pixel in the module's internal memory or command): 4, 8, or 16;
- **Number of individually addressable word parts**: 1, 2, or 4. If greater than 1, the number of bits per word part is shown and the endianness can also be configured:
  - **Big-Endian**: the highest order word part (the one with the highest order bits in the word) is assigned the lowest address;
  - **Little-Endian**: the lowest order word part (the one with the lowest order bits in the word) is assigned the lowest address.

The example shown above illustrates a configuration adequate to connect this module to the 16-bit processor, PEPE-16, which supports 2 word parts (each byte can be accessed individually) in a Big-Endian setting. See section 6.5.1.2 for further details on sub-word addressing.

Pins

This group is identical to all modules (apart from the set of pins). See section 3.4 for details.

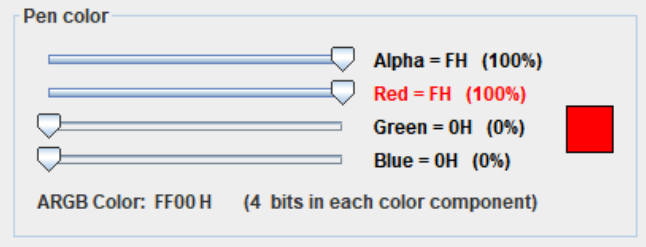
Pen color

Objects are drawn in the screen by drawing each of their individual pixels. It is possible to draw each pixel with its own color, but in some cases it is easier to just specify whether a given pixel is off (transparent) or on (in which case the pen color that has been defined is used).

The pen color is specified with the help of 4 sliders, one for each color component: Alpha (opacity), R (red), G (green), and B (blue). With 16 bits per pixel, as in this example, each component has 4 bits and 16 possible values. A pixel color value of 0000H is all colors zero (black) but completely transparent (opacity zero). An opaque black pixel must have color F000H. An opaque white pixel has color FFFFH.

This group contains a sample pixel that shows the currently selected color. Its hexadecimal value is also shown, as well as the number of bits of each color component.

The following figure provides an example of this group.



Resolutions

This group configures the width and height of the simulation interface of this module, in pixels. There are two separate resolutions:

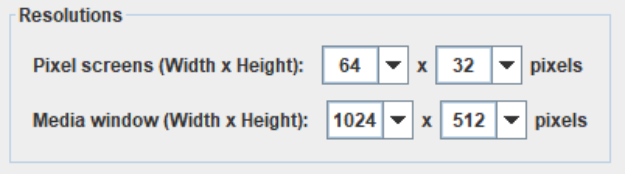
- **Pixel screens**. This refers to the number of pixels in each line (width) and column (height) that can be drawn in each pixel screen. These are not the pixels of the computer screen;

- **Media window**. This refers to the width and height of the simulation interface window, in computer screen pixels. Therefore, this determines the dimension of the window;

Typically, the resolution of the media window is higher than that of the pixels screens. This means that each pixel drawn in a pixel screen occupies several pixels in the computer screen.

Pixel screens, images, videos, and GIFs are automatically stretched to fit the media window resolution. To avoid distortions, the form factor of all these graphic elements should be similar.

The following figure provides an example of this group, with a form factor of 2 (width double of height). In this case, each pixel drawn in the MediaCenter module occupies 16 x 16 pixels in the computer screen.



Images and test pixel patterns

The simulation interface of the MediaCenter module (section 6.3.3.3) is a superposition of several windows, all with the same dimensions (width and height).

Some of these windows can display an image, allowing to create a background scenario. The image formats supported are JPG, PNG, BMP, and GIF without animation (in this case, the image displayed is the first one of the GIF). For animated GIFs, use the next group.

This group allows specifying a list of images that can be selected to be displayed during simulation, among other features. This group has the following components (many of which disabled when no image is defined, as shown by the figure with the configuration interface, above):

- A scrollable area (at the bottom) to display a list of the pathnames of all the defined image files (up to 64). The currently selected one is highlighted;
- A rectangle to display the image currently selected. If there are no images defined, a text appears in this rectangle to emphasize that;
- A button to add an image to the list, another to remove it, and two arrows to move them up or down in the list. Each image is referred to in commands by their order number, starting at 0. The order numbers of images and of playable media (next group) are independent;

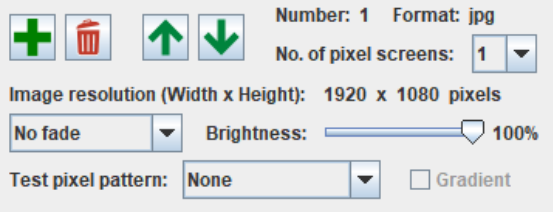
  NOTE – An image can also be added by dragging a file and dropping it on the scrollable area with the list of image file pathnames;

- The order number and format (e.g., JPG) of the image currently selected. GIFs can be used, but only its first image appears;
- The resolution in pixels of the image currently selected;
- The number of pixel screens to use (1 to 16). This module allows several of these screens to be used independently. Since they are superposed in the simulation interface (section 6.3.3.3), drawing and moving pixel-based objects on different screens avoids destroying one

when drawing another one that it partially overlaps (just like windows in an operating system). The number of pixels screens required must be configured before using them;

- Fade setting, to be used when changing an image or a video (fade-out and fade-in). This can be disabled or set from very slow to very fast;
- Brightness, from 100% to 0%, allowing to dim the entire simulation interface of this module;
- Test pixel pattern. It is possible to superimpose on the image currently displayed one of 11 predefined pixel patterns, drawn by using the pen color;
- Gradient. If a pattern is selected, it is possible to draw it with a gradient.

The following figure illustrates some of the components of this group, with some images defined in this configuration interface.



Sounds, videos, and GIFs

In addition to images, the simulation interface of the MediaCenter module (section 6.3.3.3) also supports playable media (sounds, videos, and GIFs).

This group allows specifying a list of sounds, videos, and GIFs that can be selected to be played during simulation, among other features. The audio-only formats supported are AIF, AIFF, MP3, and WAV. The video formats supported are MP4, M4A, and M4V. In this group, GIFs can be animated, just like if they were videos.

This group has the following components (many of which disabled when no media file is defined, as shown by the figure above):

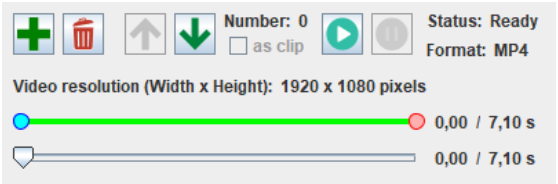
- A scrollable area (at the bottom) to display a list of the pathnames of all the defined playable media files (up to 64). The currently selected one is highlighted;
- A volume control, with a slider and a mute checkbox;
- A rectangle to display the playable media file currently selected. If there are no files defined, a text appears in this rectangle to emphasize that;
- A button to add a file to the list, another to remove it, and two arrows to move them up or down in the list. Each playable media file is referred to in commands by their order number, starting at 0. The order numbers of images (previous group) and of playable media are independent;

  NOTE – A playable media file can also be added by dragging a file and dropping it on the scrollable area with the list of playable media file pathnames;

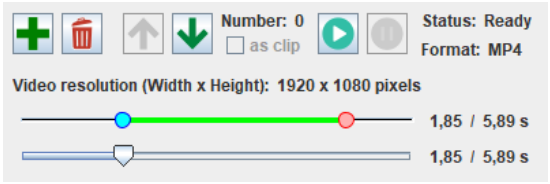
- Two buttons, one to play/stop the currently selected playable media file, and another to pause/resume;

- The order number, format (e.g., MP4) and status (UNKNOWN, READY, PLAYING, PAUSED, STOPPED, ENDED, STALLED, and MEDIA_ERROR) of the file currently selected. The status becomes STOPPED if the button to stop is clicked, and ENDED if playing reaches the end of the media;
- An "as clip" checkbox, available only when the selected media file is audio only. If checked, the full file is loaded into memory prior to playing. This improves latency, starting to play almost instantly upon receiving the play command for this file, but uses a lot of memory. It should mainly be used for short audio-only media files, e.g, short sound effects;
- The resolution in pixels of the playable media currently selected (only for videos and GIFs);
- A media-range slider, with two circles that can be dragged to specify a start and end playing time points in the media file, allowing therefore to play only a part of it. The start and end playing time points (in seconds) are also displaying next to the slider. By default, these time points are the start and end of the playable media. This information, illustrated by the following figures, is available only when a valid playable media file is selected;
- A play slider with a cursor, which advances automatically when the playable media is played (between the start and end time points defined by the media-range slider). The current playing time and duration (in seconds) is also displaying next to the slider. This information, illustrated by the following figures, is available only when a valid playable media file is selected.

Sliders without timing restrictions



Sliders with restricted start and stop times



IMPORTANT – When saving a circuit that includes a MediaCenter module with media files (images, sounds, videos, or GIFs), make sure that these files are located in the same directory (or sub-directory thereof) as the circuit file. This ensures that the pathnames of the files are stored with a path relative to the circuit file and are therefore relocatable. If the MediaCenter module refers to media files located elsewhere, these will be stored with an absolute path and will not be found if the saved circuit file is transferred to another computer.

6.3.3.3 Simulation interface

The simulation interface of the MediaCenter module can be opened by clicking the module in Simulation mode. This window is a stack of several superposed multimedia panels, all with the same width and height, usually partially transparent, so that objects can be animated on a background scenario (image, video, or GIF). These panels are, in back-to-front order:

- One background image panel, behind everything else;

- Zero or more video/GIF panels, which can be played simultaneously. They are visible only when playing, hiding the background image. The latest created is the frontmost. When a video/GIF finishes, the latest created video/GIF that happens to be still playing is shown. When all videos/GIFs finish, the background image is shown again;
- One brightness panel, transparent by default but that can be made as opaque as required, simulating a brightness control of the image, video, or GIF that is visible behind it;
- One or more colored pixel images (pixel screens), which can be written pixel by pixel, forming figures that can be moved and animated (by erasing them in one place and redrawing them in another). By default, pixel screens are created with all pixels transparent, but pixels can be redrawn in any color. The pixel screens are shrunk or stretched as needed to match the resolution of the images/videos, so that their overall sizes are the same. The number of pixels screens to use must be configured in Design mode (see section 6.3.3.2). Pixel screen 0 is the frontmost;
- One foreground image, typically mostly transparent, used for banners or other front-side information that needs to be shown over the previous panels.

Sound files are played without affecting these panels, which can be changed while the sound is playing. Several sound files can be played and heard simultaneously.

The following figure illustrates the simulation interface using the game mentioned in section 1. The background image is in fact a video that was playing and the text belongs to the foreground image. The objects seen were drawn in pixel screens.

Pixels can be drawn by accessing the MediaCenter module as a memory (section 6.3.3.4) or by using commands (section 6.3.3.5).

In both cases, commands are needed to change the background image, play, pause, or stop playing sounds, videos, or GIFs, etc. Therefore, it is easier to do everything just with commands. Section 6.3.3.5 describes the list of available commands.

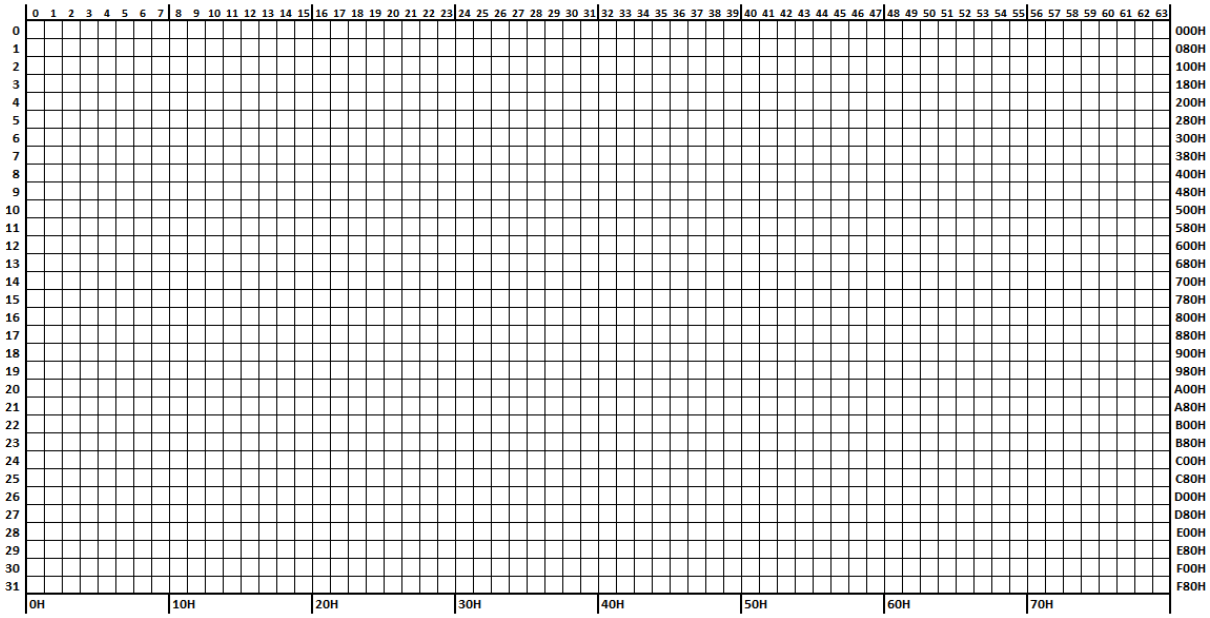
6.3.3.4 Accessing the MediaCenter as a memory

To access the MediaCenter module's internal memory directly, the processor must perform a read or a write access (activating the **RD** or **WR** pin, respectively) and specify the module's internal address (**ADDRESS** pin) corresponding to the memory cell to access. In addition, an address decoding circuit must activate the **CS_DATA** chip select pin (and not the **CS_ CTRL** chip select pin). This memory holds the pixels of all the pixel screens declared in the configuration interface (section 6.3.3.2, in the "Images and test pixel patterns" group).

Each pixel screen must be able to store the pixels declared in the pixel screen resolution. In the example of the configuration interface presented in section 6.3.3.2, "Resolutions" group, that resolution is 64 x 32 pixels (64 columns x 32 lines). Each pixel in the same example is 16 bits, which is the number of bits per word declared in the same example ("Module" group). Each word can be accessed in 2 parts, addressed individually, as also declared in the "Module" group.

The following figure illustrates this configuration for each pixel screen, with column 0 and line 0 at the top left corner. Each small square is a pixel and occupies 2 addresses. There are 64 x 32 = 2048 pixels. Their internal addresses vary between 000H and FFEH (each pixel must start at an even address). The first pixel (column 0, line 0) occupies addresses 000H and 001H and the last pixel (column 63, line 31) occupies addresses FFEH and FFFH.

Line numbers are shown on the left side and column numbers on the top side. The bottom side shows the address of each pixel relative to the address of the first pixel in each line (just every 8 pixels, to avoid cluttering), which is shown on the right side.



Therefore, there is a direct correspondence between the pixel screen image (which holds the pixels displayed in the simulation interface) and the memory (with words that hold the color of each pixel).

This is repeated for each pixel screen, if more than one is used, and address of the first pixel of the following pixel screen is just after the address of the last pixel of the previous pixel screen. This means that the pixels of the first pixel screen occupy the address range of 000H .. FFFH, the second occupies the range, 1000H .. 1FFFH, and so on. This, of course, for the case illustrated in section 6.3.3.2. Other settings (different pixel screen resolution or number of addressable parts per word) will imply different addressing ranges.

To draw a pixel or to read its current value, the processor needs only to write or read, respectively, the MediaCenter module, using the **CS_DATA** chip select pin and specifying the address of the intended pixel, which can be determined by the following formula:

*pixel address = module base address + pixel index * number of parts per word*

*pixel index = pixel screen number * number of lines *number of columns +*
*pixel line * number of columns +*
pixel column

The index of pixels in the memory are therefore sequentially numbered, throughout the various pixels screens. Multiplying this by the number of parts of each word yields the start internal address of each pixel. Since the MediaCenter module belongs to a processor-based system, the internal addresses must be added to the module's base address in the system.

NOTE – The MediaCenter module supports sub-word addressing, allowing to access each word part individually, but to draw pixels it is better to write one full pixel at a time, i.e., to write a full word. However, it is important to keep in mind that all word parts have individual addresses. In the example presented above, with 2 parts per word, the address of each pixel must start at an even address. Odd addresses are just the second part of each pixel's color;

– In addition to drawing and reading the value of pixels, the MediaCenter module offers many more features, which can only be accessed using commands, described in the next section. Therefore, it is easier to use commands for everything concerning this module, including drawing and reading pixels.

6.3.3.5 Controlling the MediaCenter with commands

6.3.3.5.1 How to issue a command

To access the MediaCenter module's features using commands, the processor must perform a write or a read access (activating the **WR** or **RD** pin, respectively) and specify the module's internal address (**ADDRESS** pin) corresponding to the command number to execute. In addition, an address decoding circuit must activate the **CS_CTRL** chip select pin (and not the **CS_DATA** chip select pin).

The access operation can be of two types:

- **Write access**. The value written by the processor through its data bus is not stored anywhere, but rather used as an argument to the write command, which can change the state of the module. For commands that do not require an argument, its value is irrelevant;
- **Read access**. This inquires the state of some aspect of the module, providing a value as a response that is read by the processor through its data bus.

From the point of view of the processor, this is a normal memory access, but from the point of view of the MediaCenter module no memory access takes place. The internal address specified is converted into a command number (as described below) and that command is executed.

The MediaCenter module does not use all the bits of the processor's address bus, but only those required to specify all the commands are used. These are the lower order bits of the address bus.

The activation of the **CS_CTRL** pin ensures that these address bits are being used by the processor to execute a command and not to access the module's internal memory.

Note that a write and a read command can have the same address and still refer to different commands. This discrimination is done with the **WR** and **RD** pins. Write and read accesses by the processor are different operations.

IMPORTANT – The MediaCenter module supports sub-word addressing, so that each word part of the internal memory can be accessed individually. However, the values used as arguments to write commands or returned by read commands may not fit into a word part (a byte, in the example used above). Therefore, all the commands must be executed by using full-word addressing;

– Therefore, the address to execute a command is not the number of that command, but rather that number multiplied by the number of parts. In the example used above, with 2 parts per word, all addresses used to execute commands must be even (e.g., 00H, 02H, 04H, and so on, corresponding to command numbers 0, 1, 2, and so on);

6.3.3.5.2 List of write commands

The list of write commands available in the MediaCenter module is presented in the following table. These correspond to write operations, performed by the processor on the MediaCenter module, which change the state of the module and must specify:

- An address that activates the **CS_CTRL** chip select pin (section 6.3.3.5.1). The lower bits of this address that connect to the **ADDRESS** pin of the MediaCenter module must specify a value equal to a write command number multiplied by the number of word parts. In the example used above, with 2 parts per word, this address value must be even (e.g., 00H, 02H, 04H, and so on, corresponding to command numbers 0, 1, 2, and so on). The table below illustrates the correspondence, valid for this example only;
- A value to write, which provides a parameter value to that command. This is a full-word value and is limited by the number of bits of each word of the MediaCenter module (configured as shown by section 6.3.3.2). For example, commands that must specify the number of a pixel (e.g., commands 7, 11, and 12) do not support the higher pixel screen resolutions, requiring separate selection of the line and column if those resolutions are used.

NOTE – The "address" column in the table below is just an example, valid for 2 word parts only. In general, it must be equal to the command number multiplied by the number of word parts;

– Commands that specify lines, columns, or pixels refer to the currently selected pixel screen only. Since all pixel screens are superposed, the currently selected pixel screen can be changed without changing the currently selected line, column, or pixel;

– If auto-increment is set to ON (see command 8), commands marked with (*) advance automatically the currently selected pixel to the next one.

Command		Description	Argument (value to write)
number	address		
0	00H	Erase all pixels of the specified pixel screen	0 .. no. pixel screens - 1
1	02H	Erase all pixels of all the pixel screens	---
2	04H	Select the specified pixel screen	0 .. no. pixel screens - 1
3	06H	Show the specified pixel screen	0 .. no. pixel screens - 1
4	08H	Hide the specified pixel screen	0 .. no. pixel screens - 1
5	0AH	Select the specified line, to use in subsequent commands	0 .. no. lines - 1
6	0CH	Select the specified column, to use in subsequent commands	0 .. no. columns - 1
7	0EH	Select the specified pixel, to use in subsequent commands. Equivalent to commands 5 + 6	0 .. no. pixels - 1
8	10H	Sets auto-increment. If ON, commands marked with (*) advance automatically the currently selected pixel to the next one	0 – OFF; 1 – ON (OFF by default)
9	12H	(*) Set the color of the currently selected pixel	Word in ARGB format
10	14H	Set the color of the pixel pen	Word in ARGB format
11	16H	Set the color of the specified pixel with the pen color	0 .. no. pixels - 1
12	18H	Clear (set to 0) the color of the specified pixel	0 .. no. pixels - 1
13	1AH	(*) Change the color of the currently selected pixel, either clearing or setting it to the pen color	0 – clear; 1 – pen color
14	1CH	(*) Change the color of P pixels (P = word part width). Each bit behaves as in command 13. Only the P lowest order bits of the parameter are used. If the currently selected column C is not a multiple of P, the pixels are drawn starting at the column that is the highest multiple of P not greater than C	Each bit of word part: 0 –clear; 1 – pen color
15	1EH	(*) Change the color of N pixels (N = word width). Each bit behaves as in command 13. If the currently selected column C is not a multiple of N, the pixels are drawn starting at the column that is the highest multiple of N not greater than C	Each bit of full word: 0 – clear; 1 – pen color
16	20H	Draw a pattern (one of the 12, including no pattern, which can be tested in the MediaCenter ’s configuration interface)	0 .. 11
17	22H	Same as command 16, but with a color gradient	0 .. 11
---	Command numbers 18 .. 31 reserved for future expansion		---
32	40H	Clear the background image (and also the "No background image selected" banner)	---
33	42H	Show the specified image in background	0 .. no. images - 1
34	44H	Clear the foreground image	---
35	46H	Show the specified image in foreground	0 .. no. images - 1
36	48H	Select the specified playable media file (sound, video, or GIF), to use in subsequent commands	0 .. no. media files - 1
37	4AH	Set the sound volume level (percentage) of the selected media file	0 .. 100
38	4CH	Mute (no sound) the specified media file	0 .. no. media files - 1
39	4EH	Set the sound volume of the specified media file to the previous level before muting it	0 .. no. media files - 1

Command		Description	Argument (value to write)
number	address		
40	50H	Mute (no sound) all the media files that are playing	---
41	52H	Set the sound volume of all playing media files to the respective previous levels before muting them	---
42	54H	Set the brightness level (percentage) of the background image and videos	0 .. 100
43	56H	Set a fading pattern for video transitions (one of the five that can be tested in the configuration interface)	0 – no fading; 1 – very slow; 2 – slow; 3 – fast; 4 – very fast
44	58H	Set the number of times a media file is to be reproduced, once it has started playing. A value of 0 will play repeatedly until one a stop commands is executed	>= 0
45	5AH	Start playing the specified media file	0 .. no. media files - 1
46	5CH	Play the specified media file continuously until stopped	0 .. no. media files - 1
47	5EH	Pause the specified media file	0 .. no. media files - 1
48	60H	Resume playing the specified media file	0 .. no. media files - 1
49	62H	Pause all the media files that are playing	---
50	64H	Resume all the media files that are paused	---
51	66H	Stop playing the specified media file	0 .. no. media files - 1
52	68H	Stop all the media files that are playing	---

6.3.3.5.3 List of read commands

The list of read commands available in the MediaCenter module is presented in the following table. These correspond to read operations, performed by the processor on the MediaCenter module, which obtain information on the state of the module and must specify an address that activates the **CS_CTRL** chip select pin (section 6.3.3.5.1). The lower bits of this address that connect to the **ADDRESS** pin of the MediaCenter module must specify a value equal to a read command number multiplied by the number of word parts. In the example used above, with 2 parts per word, this address value must be even (e.g., 00H, 02H, 04H, and so on, corresponding to command numbers 0, 1, 2, and so on). The table below illustrates the correspondence, valid for this example only.

The module provides a value with the result of the command, which the processor will read. This is a full-word value and is limited by the number of bits of each word of the MediaCenter module (configured as shown by section 6.3.3.2). For example, command number 6, which obtains the number of the currently selected pixel, does not support the higher pixel screen resolutions, requiring to separately read the currently selected line and column, if those resolutions are used.

NOTE
- The “address” column in the table below is just an example, valid for 2 word parts only. In general, it must be equal to the command number multiplied by the number of word parts;
- Commands that specify lines, columns, or pixels refer to the currently selected pixel screen only. Since all pixel screens are superposed, the currently selected pixel screen can be changed without changing the currently selected line, column, or pixel;
- If auto-increment is set to ON (which can be inspected by command 7), commands marked with (*) advance automatically the currently selected pixel to the next one.

Command		Description	Result (value read)
number	address		
0	00H	Obtain the number of columns of the pixel screens (power of 2)	1 .. 2048
1	02H	Obtain the number of lines of the pixel screens (power of 2)	1 .. 1024
2	04H	Obtain the number of the currently selected pixel screen	0 .. no. pixel screens - 1
3	06H	Obtain the visibility of the currently selected pixel screen	0 – hidden; 1 – visible
4	08H	Obtain the number of the currently selected line	0 .. no. lines - 1
5	0AH	Obtain the number of the currently selected column	0 .. no. columns - 1
6	0CH	Obtain the number of the currently selected pixel	0 .. no. pixels - 1
7	0EH	Obtain the auto-increment setting. If ON, commands marked with (*) advance automatically the currently selected pixel to the next one	0 – OFF; 1 – ON
8	10H	(*) Obtain the color of the currently selected pixel	Word in ARGB format
9	12H	Obtain the current color of the pixel pen	Word in ARGB format
10	14H	(*) Obtain the color status of the currently selected pixel	0 – transparent; 1 – non-zero color
11	16H	(*) Obtain the color status of P pixels (P = word part width). Each bit behaves as in command 10. Only the P lowest order bits are used, with the rest of the bits set to 0. If the currently selected column C is not a multiple of P, the pixels are read starting at the column that is the highest multiple of P not greater than C	Each bit of word part: 0 – transparent; 1 – non-zero color
12	18H	(*) Obtain the color status of N pixels (N = word width). Each bit behaves as in command 10. If the currently selected column C is not a multiple of N, the pixels are drawn starting at the column that is the highest multiple of N not greater than C	Each bit of full word: 0 – transparent; 1 – non-zero color
---	Command numbers 13 .. 31 reserved for future expansion		---
32	40H	Obtain the number of defined images	0 .. 64
33	42H	Obtain the number of the image shown as background	-1 .. no. images - 1 (-1 if none)
34	44H	Obtain the number of the image shown as foreground	-1 .. no. images - 1 (-1 if none)
35	46H	Obtain the number of defined playable media files (sounds, videos, and GIFs)	0 .. 64
36	48H	Obtain the number of the currently selected playable media file	-1 .. no. media files - 1 (-1 if none set)
37	4AH	Obtain the sound volume level (percentage) of the selected media file	0 .. 100
38	4CH	Obtain the mute status the selected media file	0 – normal; 1 – muted
39	4EH	Obtain the brightness level (percentage) of the background image and videos	0 .. 100
40	50H	Obtain the current fading pattern for video transitions (one of the five that can be tested in the configuration interface)	0 – no fading; 1 – very slow; 2 – slow; 3 – fast; 4 – very fast

Command		Description	Result (value read)
number	address		
41	52H	Obtain the status of the currently selected playable media file	0 - Unknown; 1 - Ready; 2 - Paused; 3 - Playing; 4 - Stopped; 5 - Error
42	54H	Obtain the number of media files that are playing	0 .. no. media files - 1
43	56H	Obtain the number of times that the currently selected media file is to be reproduced, once it has started playing. A value of 0 means that it will play repeatedly until a stop command is executed	>= 0

6.4 Category Memories

This category includes modules that processors use as memories, to hold programs and data.

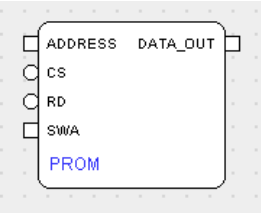
These modules offer the same basic structure and user interfaces, differing mainly on the operations they support and on how they connect to the processor.

They all support sub-word addressing, allowing to access all the bits of a memory cell at once or in individually addressable word parts (the width of a word is the number of bits of each memory cell), each part with a number of bits that must be a power of 2. This is provided to allow a 16 bit processor, for example, to access a 16-bit memory in one single operation, or to access the two 8-bit halves individually. This can be important in some applications and is supported by the 16-bit processor available in this simulator, the PEPE-16 (section 6.5.1).

6.4.1 Module PROM

The PROM (Programmable Read-Only Memory) module is a memory that can only be read from. In the real world, these memories are non-volatile, which means that they retain their contents when the power supply of the circuit is switched off. They are used for program instructions and permanent data, not for storing variable data. They are programmable if their contents can be changed (infrequently). In the simulator, volatility is not an issue, but the read-only characteristic is maintained. The PROM's contents can be changed by the user or by loading data from a file.

The following figure shows the PROM module as it appears in the circuit area.



It provides the following pins, to connect to the processor:

- **ADDRESS**. This pin connects to the processor's address bus and allows specifying which memory cell to read. It has as many bits as needed to discriminate all memory addresses, i.e., the base 2 logarithm of the number of individually addressable word parts (or full words, if sub-word addressing is not used);

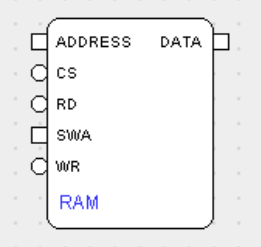
- **CS** (chip select). This pin indicates that the processor wants to access this memory. It is active at 0 (hence the round pin in the module representation);
- **RD** (read). In addition to the **CS** pin, the **RD** pin must also be active (at 0) for the processor to perform the read operation. This distinguishes read from write operations. This memory supports read operations only, but the processor is able to write into other devices (e.g., the RAM module, described in section 6.4.2);
- **SWA** (sub-word addressing). This pin specifies how many word parts to read and appears only when the number of individually addressable parts is greater than 1. This pin connects to the processor, which must provide support for sub-word addressing. With a 16-bit PROM module with 2 individually addressable word parts, for example, the **SWA** pin has only 1 bit, to specify whether to read one part (one byte) or two parts (full word). See section 6.5.1.2 for further details on sub-word addressing;
- **DATA_OUT**. This is an output pin, to be read by the processor (connects to its data bus), and conveys the bits read from the memory cell. Note that this pin has tristate capability and is active only when both the **CS** and **RD** pins are active (at 0), being in high impedance the rest of the time.

The configuration and simulation interfaces of the PROM module, common to all memory modules, are described in the sections 6.4.4 and 6.4.5, respectively.

6.4.2 Module RAM

The RAM (Random Access Memory) module is a memory that can be both read from and written into. In the real world, these memories are volatile, which means that they lose their contents when the power supply of the circuit is switched off. It can be used for program instructions and variable data, not for storing non-volatile data. In the simulator, volatility is not an issue and the read/write capability is the main distinguishing feature with respect to PROMs.

The following figure shows the RAM module as it appears in the circuit area.



It provides the following pins, to connect to the processor:

- **ADDRESS**. This pin connects to the processor's address bus and allows specifying which memory cell to access. It has as many bits as needed to discriminate all memory addresses, i.e., the base 2 logarithm of the number of individually addressable word parts (or full words, if sub-word addressing is not used);
- **CS** (chip select). This pin indicates that the processor wants to access this memory. It is active at 0 (hence the round pin in the module representation);

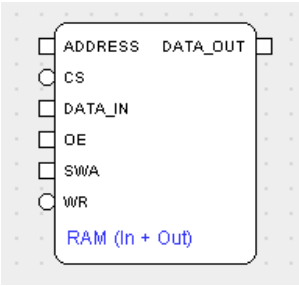
- **RD** (read). In addition to the **CS** pin, the **RD** pin must also be active (at 0) for the processor to perform the read operation. This distinguishes read from write operations. The **RD** and **WR** pins must not be active (at 0) simultaneously;
- **SWA** (sub-word addressing). This pin specifies how many word parts to access and appears only when the number of individually addressable parts is greater than 1. This pin connects to the processor, which must provide support for sub-word addressing. With a 16-bit RAM module with 2 individually addressable word parts, for example, the **SWA** pin has only 1 bit, to specify whether to access one part (one byte) or two parts (full word). See section 6.5.1.2 for further details on sub-word addressing;
- **WR** (write). In addition to the **CS** pin, the **WR** pin must also be active (at 0) for the processor to perform the write operation. This distinguishes write from read operations. The **WR** and **RD** pins must not be active (at 0) simultaneously;
- **DATA**. This is an input/output pin, to be read from or written into by the processor (connects to its data bus), and transfers the bits read from or written into the memory cell. Note that this pin is normally set as an input, being set as output only when both the **CS** and **RD** pins are active (at 0), i.e., when the processor is reading a memory cell.

The configuration and simulation interfaces of the RAM module, common to all memory modules, are described in the sections 6.4.4 and 6.4.5, respectively.

6.4.3 Module RAM (In + Out)

The RAM (In + Out) module is a memory that can be both read from and written into. It is identical to the RAM module (section 6.4.2), with the difference that it has separate data buses, one to read data from, and the other to write data into, a memory cell. In addition, there is only one pin to specify whether to write or to read. It is useful for single-cycle processors that require separate data buses, such as the 8-bit processor available in this simulator, the PEPE-8 (section 6.5.2).

The following figure shows the RAM (In + Out) module as it appears in the circuit area.



It provides the following pins, to connect to the processor:

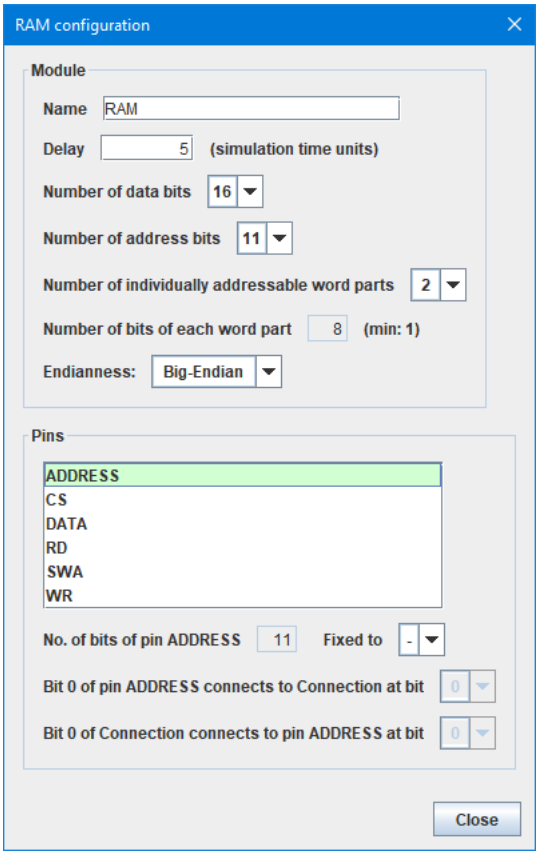
- **ADDRESS**. This pin connects to the processor's address bus and allows specifying which memory cell to access. It has as many bits as needed to discriminate all memory addresses, i.e., the base 2 logarithm of the number of individually addressable word parts (or full words, if sub-word addressing is not used);
- **CS** (chip select). This pin indicates that the processor wants to access this memory. It is active at 0 (hence the round pin in the module representation);
- **DATA_IN**. This is an input pin, which the processor writes into (connects to its data output bus), and conveys the bits to write in the memory;

- **OE** (output enable). The **DATA_OUT** is an output pin with tristate capability and the **OE** pin controls whether it is active (**OE** = 1) or in high impedance (**OE** = 0);
- **SWA** (sub-word addressing). This pin specifies how many word parts to access and appears only when the number of individually addressable parts is greater than 1. This pin connects to the processor, which must provide support for sub-word addressing. With a 16-bit RAM (In + Out) module with 2 individually addressable word parts, for example, the **SWA** pin has only 1 bit, to specify whether to access one part (one byte) or two parts (full word). See section 6.5.1.2 for further details on sub-word addressing;
- **WR** (write). This pin distinguishes between write (**WR** = 0) and read (**WR** = 1) operations, when the **CS** pin is active at 0;
- **DATA_OUT**. This is an output pin, which the processor reads from (connects to its data input bus), and conveys the bits to read from in the memory. This pin has tristate capability and the **OE** pin controls whether it is active (**OE** = 1) or in high impedance (**OE** = 0). If **CS** is not active (**CS** = 1), the value of this pin is not valid (albeit active, if **OE**=1).

The configuration and simulation interfaces of the RAM (In + Out) module, common to all memory modules, are described in the sections 6.4.4 and 6.4.5, respectively.

6.4.4 Configuration interface of memories

All the memory modules share the same configuration interface, with differences in the set of pins as described in the respective sections. This section describes the configuration interface using the RAM module as an example, using a word with of 16 bits and 2 word parts (each memory byte addressable individually), as illustrated by the following figure.



This interface includes the component groups described below, with the configurable parameters and information shown.

Module

It is possible to configure:

- **Name** of the module;
- **Simulation delay**, in simulation time units. This allows to simulate the propagation delays of real world memories, between activating the **CS**, **RD**, and **WR** pins and actually performing the read or write operation;
- **The number of data bits**, which is the number of bits in each full word: 1 .. 16. This is just the width of the word as is orthogonal to the number of memory cells;
- **The number of address bits**, which is the number of bits required to address all individually addressable memory cells: 1 .. 16. This setting determines the memory's capacity, in word parts ($2^{\text{no. address bits}}$). In full words, it is given by $2^{\text{no. address bits}}$ / *no. individually addressable word parts*. Note that the number of address bits needed to address a memory is equal to the base 2 logarithm of the memory's capacity in full words only if the number of individually addressable word parts is 1 (no sub-word addressing). Otherwise, it is higher;
- **Number of individually addressable word parts**: a power of 2 that is a divisor of the number of data bits. If greater than 1, the number of bits per word part is shown and the endianness can also be configured:
  - **Big-Endian**: the highest order word part (the one with the highest order bits in the word) is assigned the lowest address;
  - **Little-Endian**: the lowest order word part (the one with the lowest order bits in the word) is assigned the lowest address.

The example shown above illustrates a configuration adequate to connect this memory module to the 16-bit processor, PEPE-16, which supports 2 word parts (each byte can be accessed individually) in a Big-Endian setting, with 11 address bits, yielding a capacity of 2^{11} = 2048 bytes (2^{10} = 1024 words with 16 bits, or 2 bytes). See section 6.5.1.2 for further details on sub-word addressing.

Pins

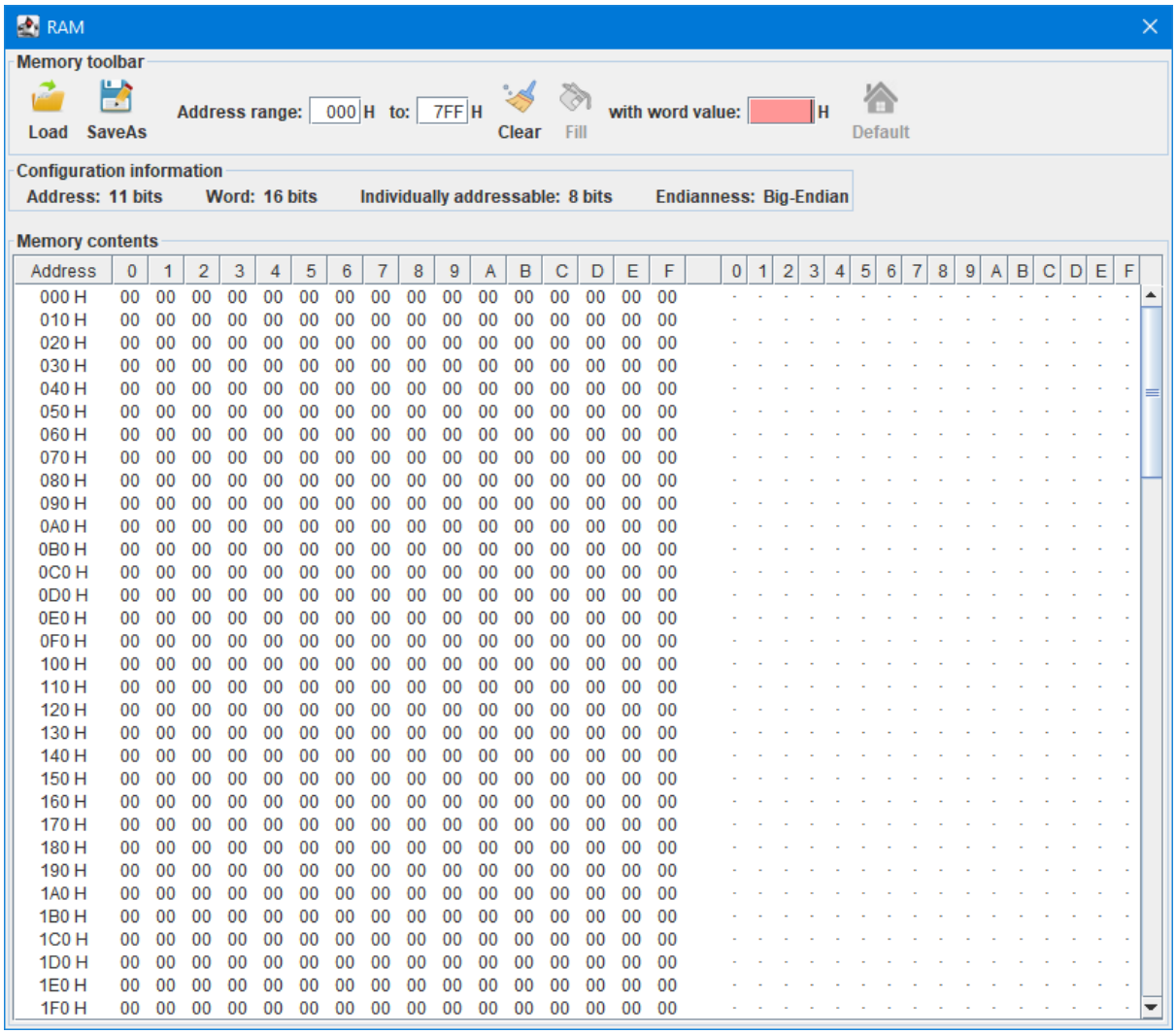
This group is identical to all modules (apart from the set of pins). See section 3.4 for details. The pins for each memory module have already been described in the respective sections.

6.4.5 Simulation interface of memories

In Simulation mode, it is possible:

- To see the value of each individually addressable memory cell;
- To change its value, or to fill a cell range with a value, or to clear a cell range;
- To save the current data contents to a file, or to load data from a file.

To open the memory module's simulation interface, click it in Simulation mode. The interface for the RAM module example used in the previous section is illustrated by the following figure.



This interface includes the component groups described below.

Memory toolbar

This toolbar offers the following commands (all buttons display what they do by hovering the mouse cursor over it) and value fields:

- **Load** and **SaveAs**, to load from a file previously saved data and to save to a file the current data contents. By default, memory content files are saved with the ".dat" extension;
- **Address range**, with initial and final address fields, to be used when clearing or filling words. Note that these fields refer to word part addresses (if sub-word addressing is used, as in the example above), but only full words can be cleared or filled. Therefore, the start address must be a multiple of the number of word parts and the final address must be a multiple of the number of word parts minus one. This means that the address range must refer to a set of complete words. These fields are yellow if they have values different from the default values (first and last address in the memory) and are red if the specified value is not valid;

- **Clear**, to set to zero the memory words specified in the address range;
- **Fill**, to set the memory words specified in the address range to the word value specified in the next field (disabled if no value has been specified yet);
- **Word value**, a word value to be used when filling memory words. This field is red if empty (default) or invalid, and yellow if a value has been specified (therefore, not the default);
- **Default**, a button to reset the address range and word value fields to their default values (disabled if the fields already have their default values).

Configuration information

This group displays information on the configuration of the memory, to make it easier to understand the data displayed:

- **Number of address bits**. 2 raised to this number yields the capacity of the memory, in word parts (bytes, in this example). The address range fields (in the previous group) give the first and last word part addresses;
- **Word width**, in bits (16 in this example);
- **Width of each individually addressable word part**, in bits (8, in this example, since there are 2 word parts);
- **Endianness**, indicating which word part (highest or lowest order) is stored in the lowest address (multiple of the number of word parts). In this example, Big-Endian is used, which means that the highest order part (bits 15 to 8) is stored at an even address and the lowest order part (bits 7 to 0) is stored in the next address (odd).

Memory contents

This group displays all the individually addressable word parts (bytes, in this example), organized in a table with 16 word part values in each row. This is merely an organizing option, since conceptually they are a vector of sequential values, each with its own address.

Each row of the table displays the address of the first word part, which means it is increased by 16 (10H) as each new row begins. In the table's header, the numbers 0 to F are to be added to the address at the beginning of each row to determine the actual address of each value.

Since each word part in this example is a byte, each value has 2 hexadecimal digits.

At the right, in each row, the text representation of each value is displayed, using ASCII (American Standard Code for Information Interchange) encoding, if the value corresponds to a printable character (letters, digits, and punctuation symbols. Otherwise, a dot (".") is displayed instead.

This table can be scrolled to see all the memory values.

The user can manually change any value, by clicking it and typing in a new value. However, the new value is assumed only when the user clicks the enter key or clicks on another value.

6.5 Category Processors

This category includes modules that can be programmed, offering the capabilities of processors. This simulator offers two:

- **PEPE-16**, a 16-bit processor that can be programmed in assembly language and C;
- **PEPE-8**, an 8-bit processor that can be programmed in assembly language (with a much simpler instruction set than that of PEPE-16).

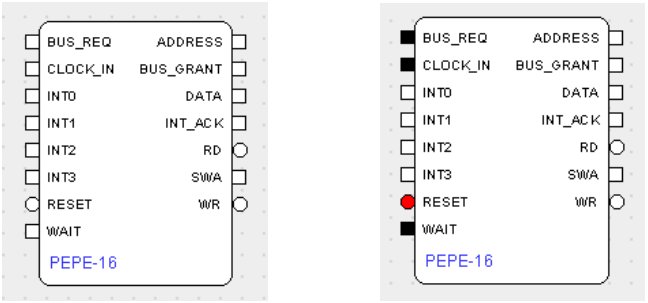
6.5.1 Module PEPE-16

The following sections describe only the features of the PEPE-16. Sections 7.1 and 7.2 describe how to program and to debug it, respectively, both in assembly and C languages.

6.5.1.1 General description

The PEPE-16 processor module is one of the most complex modules in the simulator and supports programming and debugging in assembly language and C. It has been conceived to implement complete processor-based systems, in conjunction with memory and peripheral modules. In particular, when controlling a MediaCenter module it can support programming of multimedia-based applications, such as the game briefly described in section 1.

The followings figures illustrate the PEPE-16 module, as it appears in the circuit area (left) and with some input pins forced to 0 or 1 (right), as described below.



The PEPE-16 module has the following pins:

- **BUS_REQ** (bus request). The PEPE-16 has been designed to support DMA (Direct Memory Access), but requires a DMA controller module, which has not been developed yet. Therefore, this input pin is not used and should be forced to 0;
- **CLOCK_IN**. This pin can be used to exercise the processor with an external clock. If not used (the default case), it should be forced to 0;
- **INT0 .. INT3**. These input pins are used to request interrupts to the processor by the external hardware (including timers). Interrupts are described in section 6.5.1.6;
- **RESET**. This input pin maintains the PEPE-16 in the reset state as long as it is active (at 0). The PEPE-16 performs an auto reset when the simulation starts, but if some external hardware is slow to initialize and needs to delay the start of the processor, this pin can be connected to a Reset module (section 6.2.10) configured with a suitable delay. This pin can also be used to reset the processor during the simulation, if needed. The normal case, however, is just to use the auto reset feature and to force this pin to 1 (not active);

- **WAIT**. This input pin prolongs the memory/peripheral access cycles (reads and writes), as long as it is active (at 1). It is provided to support slow memory or peripheral devices, if needed. The most usual case is to configure the delays of the processor and of the devices so that no additional delays are needed. Therefore, this pin is forced to 0 (not active);
- **ADDRESS**. This output pin allows the processor to specify which memory cell or peripheral port (command, in the case of the MediaCenter) it wants to access. Usually, only the lowest order address bits connect to the memory or peripheral modules. The highest order address bits connect to the address decoding sub-system, to generate the chip select signals to notify the devices that the processor is accessing them;
- **BUS_GRANT**. The PEPE-16 has been designed to support DMA (Direct Memory Access), but requires a DMA controller module, which has not been developed yet. Therefore, this output pin is not used and should be left unconnected;
- **DATA** (data bus). This input/output pin is the 16-bit wide data bus of the processor and connects to the data buses of the memory and peripheral modules, to transfer the values to write into or to read from these modules;
- **INT_ACK** (interrupt acknowledge). The PEPE-16 supports more than 4 interrupt inputs, but requires an interrupt controller module, which has not been developed yet. Therefore, this output pin is not used and should be left unconnected;
- **RD** (read). This pin will be active (at 0) during a read operation by the processor. This distinguishes read from write operations, in either memory and peripheral accesses;
- **SWA** (sub-word addressing). This output pin has only 1 bit, since the number of individually addressable word parts in the PEPE-16 is 2, and specifies how many word parts to access. If 0, only one part (byte) will be accessed. If 1, a full word (16 bits) will be accessed. See section 6.5.1.2 for further details on sub-word addressing;
- **WR** (write). This pin will be active (at 0) during a write operation performed by the processor. This distinguishes write from read operations, in either memory or peripheral accesses.

6.5.1.2 Sub-word addressing

Most processors support accessing not only full words (reading or writing all the bits in a word at once) but also parts of words (usually multiple of 8 bits). A 32-bit processor, for example, is typically able to access data in 32-bit, 16-bit, and 8-bit (byte) chunks. The PEPE-16 processor is able to access 16-bit (full) words and 8-bit (byte) word parts.

This is known as sub-word addressing and requires that:

- Every individual word part must have its own address, so that the processor is able to specify which word part it wants to access;
- There must be a way for the processor to specify how many word parts it wants to access (from 1 to all, always a power of 2);
- The address that the processor specifies must be a multiple of the number of parts it wants to read. For example, the 32-bit processor above must specify an address multiple of 4 when

accessing full 32-bit words, multiple of 2 when accessing 16-bit word parts and multiple of 1 (any address) when accessing 8-bit word parts. In the case of PEPE-16, the address of a full word must be multiple of 2 and any address (even or odd) can be used when accessing each byte (8 bits) individually;

- The processors, memories, and peripherals in a system must support the same sub-word addressing capabilities.

In this simulator, all the memory and peripheral modules support sub-word addressing, but only the PEPE-16 processor supports it (PEPE-8 can only access 8-bit data and has no need for it).

Each of the modules with support for sub-word addressing has a **SWA** pin (output on the processor and input on the other modules), used to indicate how many word parts the processor wants to access.

The value at the **SWA** pin is the base 2 logarithm of the number of word parts to access simultaneously (0 for 1 part, 1 for 2 parts, 2 for 4 parts, and so on). Therefore, it ranges from 0 to the base 2 logarithm of the number of individually addressable word parts (i.e., from 1 word part to the number of parts in a full word). The number of word parts must always be a power of 2.

This pin has as many bits as required to support all possible combinations of how many word parts to access. The 32-bit processor example above would require 2 bits, since there are 3 possibilities: 1 word part (8 bits, **SWA**=0), 2 word parts (16 bits, **SWA**=1), and 4 word parts (full 32-bit word, **SWA**=2). The PEPE-16 has only 16 bits and requires a **SWA** pin with only 1 bit, since there are only 2 possibilities: 1 word part (8 bits, **SWA**=0) and 2 word parts (full 16-bit word, **SWA**=1).

The address specified by the processor must be consistent with the value of the **SWA** pin. This means that the address must be a multiple of the number of word parts indicated by the **SWA** pin.

With sub-word addressing, with each word part individually addressable, the address space assigns one address to each word part, just as if each word part were independent from the others. The start address of each word is a multiple of the number of word parts.

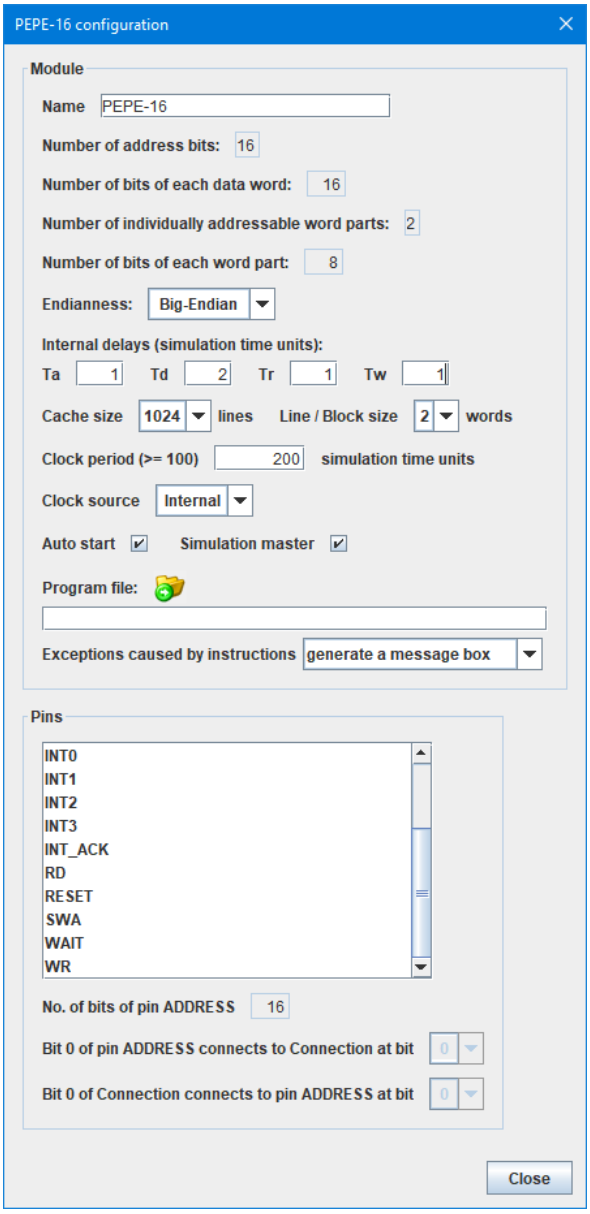
This raises a problem, known as *endianness*: which part, or which end, of the word comes first in the addressing scheme (i.e., located at the lowest address)? There are two possibilities:

- **Big-Endian**: the highest order word part (the one with the highest order bits in the word) is assigned the lowest address;
- **Little-Endian**: the lowest order word part (the one with the lowest order bits in the word) is assigned the lowest address.

Both possibilities are valid and the choice is irrelevant if all interconnected devices (processor, memory, and peripherals) use the same endianness setting and perform full word accesses. However, accessing just a part of the word will behave differently in big-endian and in little-endian configurations. Care must be taken when programming the processor with sub-word addressing instructions.

6.5.1.3 Configuration interface

The following figure illustrates the configuration interface of the PEPE-16 module, with typical settings (but others can be set).



This interface includes the component groups described below, with the configurable parameters and information shown. Some of these settings can also be changed during simulation, by the user.

Module

It is possible to configure or to obtain information on:

- **Name** of the module;
- **Number of address bits** (information only): 16 bits;
- **Number of bits per word** (information only): 16 bits;
- **Number of individually addressable word parts** (information only): 2;
- **Number of bits per word part** (information only): 8 bits. This means that the processor can access a full 16-bit word or individual bytes;

- **Endianness**, which can be configured as:
  - **Big-Endian** (default): the highest order byte (bits 15 to 8) is assigned the lowest address;
  - **Little-Endian**: the lowest order byte (bits 7 to 0) is assigned the lowest address.
- **Internal delays**, pertaining to write and read cycles, which are described in detail in section 6.1.6.2 of the book underlying this simulator, "Arquitetura de Computadores", 5th edition, ISBN 978-972-722-789-1, by José Delgado and Carlos Ribeiro (book in Portuguese). It should be stressed that, unlike many other processor simulators, which just simulate the functionality of the instruction set, this simulator includes the hardware communication protocol between the processor and memories/peripherals, just like a real circuit:
  - **Ta**: Reaction delay from the rising edge of the clock until the address bus is stable;
  - **Td**: Reaction delay between the edges of the clock and the activation/deactivation of the data bus;
  - **Tr**: Reaction delay between the edges of the clock and the edges of the **RD** pin, in read bus cycles;
  - **Tw**: Reaction delay between the edges of the clock and the edges of the **WR** pin in write bus cycles.
- **Cache size**. The processor includes a direct-mapped cache. This is the number of lines of that cache, configurable between 0 (no cache) and at most 4096 (with 1 word per line);
- **Line/Block size**, the number of words in each line. Note that the product of the number of lines (cache size) and the line size cannot exceed 4096;
- **Clock period**, in simulation time units, relevant only if the clock is configured as internal. This should be high enough to allow for the delays in various modules, such as the address decoding subsystem, memories, and peripherals;
- **Clock source**, which can be configured as:
  - Internal, generated internally with the above clock period;
  - External, in which case a clock generation circuit must be connected to the **CLOCK_IN** pin;
- **Auto start**. If checked, the processor will start executing the program as soon as the simulator gets into Simulation mode. Otherwise, the user must start the program manually;
- **Simulation master**. If checked, pausing or stopping the processor also pauses or stops the other modules, i.e., controls the simulation. Otherwise, pausing or stopping affects only this module;
- **Program file**. Clicking on the button opens a dialog that allows to specify a file with the program to run (its path will be shown in the text box). The file can also be specified by drag and drop of a source program file to the text box. When changing to Simulation mode, that program is loaded into the memory connected to the processor;
- **Exceptions caused by instructions**, such as division by zero, cause an exception. This parameter can be configured to generate a message box, warning the user of the occurrence, or to invoke an exception routine (the program needs to set up this routine);

Pins

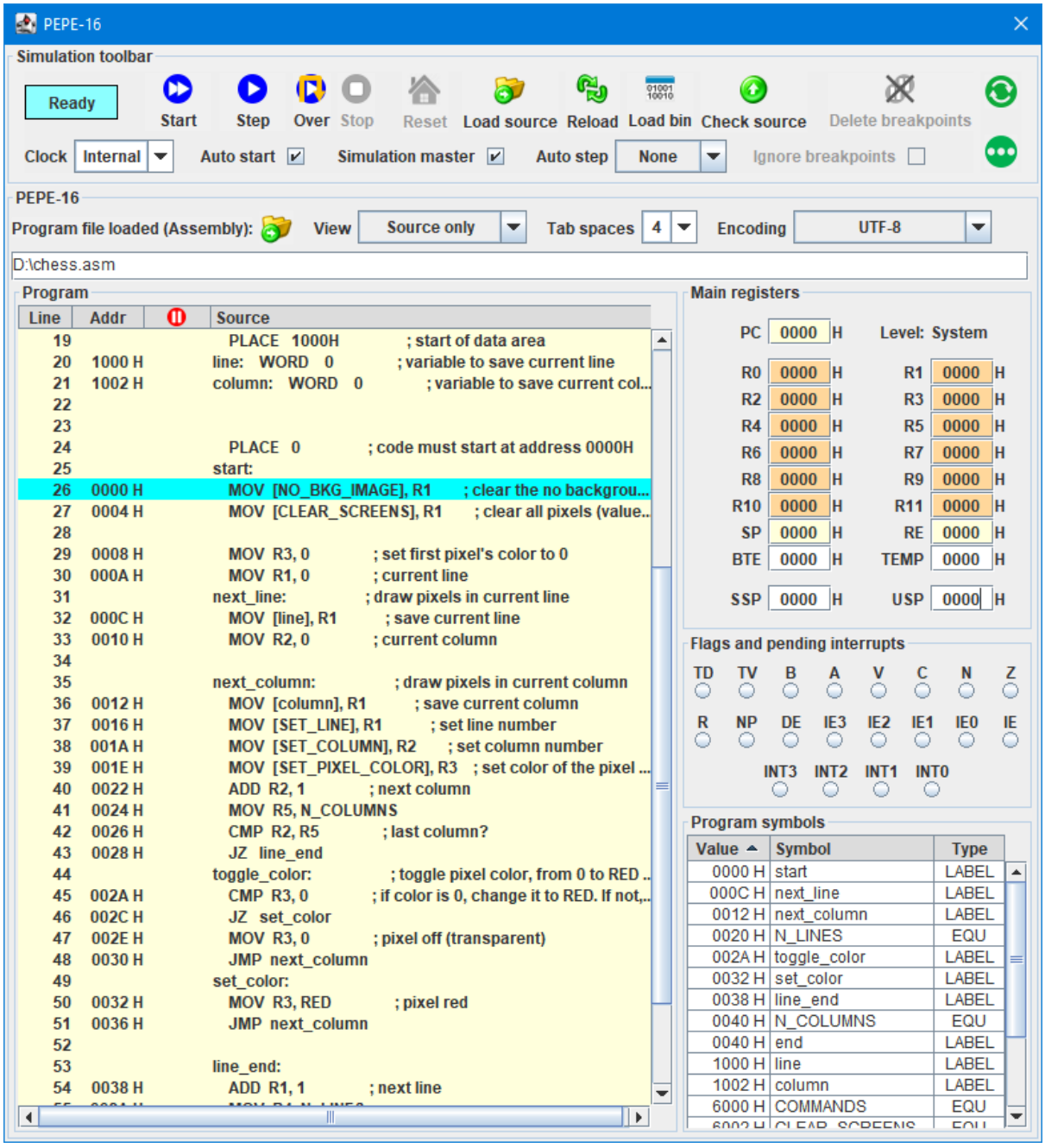
This group is identical to all modules (apart from the set of pins). See section 3.4 for details.

6.5.1.4 *Simulation interface*

The simulation interface of the PEPE-16 module can be opened by clicking the module in Simulation mode.

The following figure illustrates this interface, showing part of a small program in PEPE-16 assembly language that produces a chess pattern in the simulation interface of a MediaCenter module (see section 7.1.1.3). This includes three main groups of components:

- **Simulation toolbar**, with controls that act on the processor, enabling several operations;
- **PEPE-16**, with information on the program and state of the processor;
- **Additional panel**, usually hidden, which includes additional features of the PEPE-16 (not shown in the following figure, see section 6.5.1.4.3).



6.5.1.4.1 Simulation toolbar

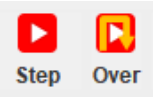
This toolbar includes the following components:

- **Status**, indicating whether the processor is Not Ready, Ready, Stopped, Running, Stepping, Waiting, or Paused. Each status has its own color, also reflected in the module in the circuit area;
- **Start** executing the program;
- **Step**, to execute a single instruction. When executing the program, this button changes to **Pause** and enables pausing execution;
- **Over**, similar to Step but executes CALL instructions as if they were a single instruction (effectively, they step over the routine);
- **Stop** the execution of the program;
- **Reset**, a button enabled when Stopped to reset the state of the processor, changing its status to Ready;
- **Load source**, to select a file with a source program (in PEPE-16 assembly or C languages) and load it into memory;
- **Reload**, which reloads the currently loaded program. This is useful when the program file has been edited and changed, to load the new version by just clicking this button;
- **Load bin**, to select a file with a previously compiled program (binary code) and load it into memory;

IMPORTANT – When loading a program, in other source program or compiled code formats, the processor does not verify that enough memory is present. If, when running the program, error messages occur for no apparent reason, check that there is memory available at all the addresses used by the program for instructions, variables, and stack area. All the program views except source view (see next section) provide an indication of the addresses used by the program (except peripherals).

- **Check source**, to select a file with a source program (in PEPE-16 assembly or C languages) and compile it without loading it into memory. Useful to check for compilation errors, and also to save the result as a compiled program, binary file;
- **Delete breakpoints**, to delete any breakpoint that has been set;
- **Refresh** (). A toggle button that, if selected, periodically refreshes the information below on the PEPE-16, such as the position of the blue bar, which indicates the instruction to be executed next, and the values of the registers and flags. During program execution, this information varies too fast for the window components to keep up with. This periodic refresh enables the user to at least have an idea of what the processor is doing;
- **Clock** source, which can be internal (the default) or external. This setting is also available in the configuration interface;
- **Auto start**. If checked, the processor will start execution automatically when changing the simulator to Simulation mode. Otherwise, the user must start the program manually. This setting is also available in the configuration interface;

- **Simulation master**. If checked, pausing or stopping the processor also pauses or stops the other modules, i.e., controls the simulation. Otherwise, pausing or stopping affects only this module. This setting is also available in the configuration interface;
- **Auto step**, which performs single step or step over operations automatically, at a specific rate that can be selected (from very slow to very fast). This is useful to execute a program at a pace much slower than continuous execution, so that the user can check in detail how the program evolves, in instruction addresses and register values, without having to click the step buttons repeatedly. If the setting is not None, the step buttons become red. Once clicking one of these buttons, stepping starts, pausing briefly at every instruction, and executing CALL instructions as if they were a single instruction (if Over was clicked):



- **Ignore breakpoints**. If checked, all defined breakpoints are ignored. Execution does not stop at breakpoints, but since these are not deleted, they are available again by unchecking this check box;
- **More** (). A toggle button that, if selected, extends the simulation interface by opening up a panel with additional information on the processor (see section 6.5.1.4.3).

6.5.1.4.2 Information on PEPE-16

This is where the most relevant information on the state of the processor is presented. It includes the following groups:

Program file and view

The text box displays the path name of the file with the source program currently loaded into memory. The open button () allows browsing to load another source program file. The file can also be specified by drag and drop of a source program file to the text box.

The program loaded is displayed in the Program group below. There are 4 possible views:

- **Source only** (illustrated in the simulation interface above), a copy of the source file, but with the start address of each instruction. Source line indentation is not perfect;
- **Source with code**, in which for each source instruction is followed by the corresponding assembly instructions generated by the compiler, ordered by source line number. This is more useful in C language programs, since it shows both the C statements and the assembly instructions it generates, but it also works in assembly language programs. Section 7.1.2.9 provides an example of this view with a C program;
- **Code with source**. Identical to Source with code, but ordered by the address of generated instructions instead of source line number. The difference is more noticeable in programs that manually place variables or code with PLACE directives with starting addresses that appear in decreasing order;
- **Code only**. Only the assembly instructions are shown, without source information (namely, comments). It is a more compact view.

Regarding the source program text, shown in the Program group (except in Code only view), it is possible to specify:

- The number of blank spaces inserted for each tab in the source program text. Spacing and indentation are not identical to those of the source text (the font is not monospaced to provide a more compact view), so this setting can help in improving the source text layout;
- Encoding. Several text encodings are available to match that of the source program file, so that the source text appears correctly in the Program group.

Program

This group displays the program instructions, according to the view selected, above. Depending on that view, the following columns may appear:

- **Line**. This is the number of each source line, as it appears in the source program file. Some lines will only include comments, or just a new line;
- **Addr**. This is the start address of each instruction, in hexadecimal. Most instructions occupy 2 addresses, but a few more complex ones occupy 4. Only the rows with instructions that generate code or data will have an address;
- ⏸ This is a column where breakpoints can be defined (see section 7.2.1.3);
- **Source**. The text of each source program line;
- **Label**. Each assembly language line can start with a label, so that it can be referenced in control flow instructions;
- **Mnemonic**. Each assembly instruction has a mnemonic, to indicate what it does.
- **Operands**. Some assembly instructions have operands, such as registers and constants.

If the program is not running, a colored bar indicates the instruction that will be executed next. The color is cyan if the program is stopped or paused, and magenta or green if the program reached a code or data breakpoint, respectively (see section 7.2.1.3).

Main registers

This group displays the values of the main registers of PEPE-16:

- **PC** (Program Counter). Contains the address of the instruction to be executed next;
- **R0** through **R15**. Of these, the last four (R12 to R15) are used internally by the PEPE-16 (therefore, should not be used by the program) and have specific names:
    - **SP** (R12, Stack pointer). This is the address of the last used word of the stack;
    - **RE** (R13, State register, or flags). Each bit of this register is a flag, namely used in conditional jumps (such as the Zero and Carry flags) and to control interrupts. These are detailed in the “Flags and pending interrupts” group);
    - **BTE** (R14, base of the exception table). This register is used as the base of the table with the addresses of the exception routines;
    - **TEMP** (R15, temporary register). This register is used to hold temporary values by some of the most complex instructions;

- **SSP** (System Stack Pointer) and **USP** (User Stack Pointer). PEPE-16 has two privilege levels, System and User, each with its own stack pointer. The SP register is actually an alias to one of these registers, depending on the privilege level selected (System, upon reset) by the NP flag of the **RE** register. The current privilege is also indicated explicitly, right next to the **PC** register.

When the program is not running, these registers can be changed manually by the user.

Section 6.5.1.5 describes the main registers of PEPE-16 in detail.

Flags and pending interrupts

This group displays two types of flags, which can be changed manually (if the program is not running) by clicking them:

- **State flags**: each of the 16 bits of the RE register. If changed manually, the value of the RE register is updated accordingly;
- **Pending interrupt flags**, which if set indicate that the respective interrupt has been requested and is pending. The processor will service it if that interrupt is enabled, by the respective interrupt enable flag (IE0 to IE3) and by the global interrupt enable flag (IE). If enabled manually, it is possible to simulate a hardware interrupt request without actual hardware connected to the interrupt pin.

Program symbols

This group displays a table with a row for each symbol defined in the program, with the following columns:

- **Value**: the value of the symbol, in hexadecimal. This is an address for labels and a numeric constant value for constants defined with an EQU directive;
- **Symbol**: the name of the symbol;
- **Type**: Label or EQU, according to whether the symbol was defined by a label or by an EQU directive.

Any of these columns can be sorted, in ascending or descending order, by clicking the header cell and then the small triangle next to the column name.

6.5.1.4.3 Additional panel

Clicking the **More** () toggle button extends the simulation interface, opening up a panel with additional information on the processor. This panel has three selectable tabs:

- **Processes**, to display information on processes;
- **Cache**, to expose statistics on the cache;
- **Auxiliary registers**, to display the values of additional registers.

The Processes tab

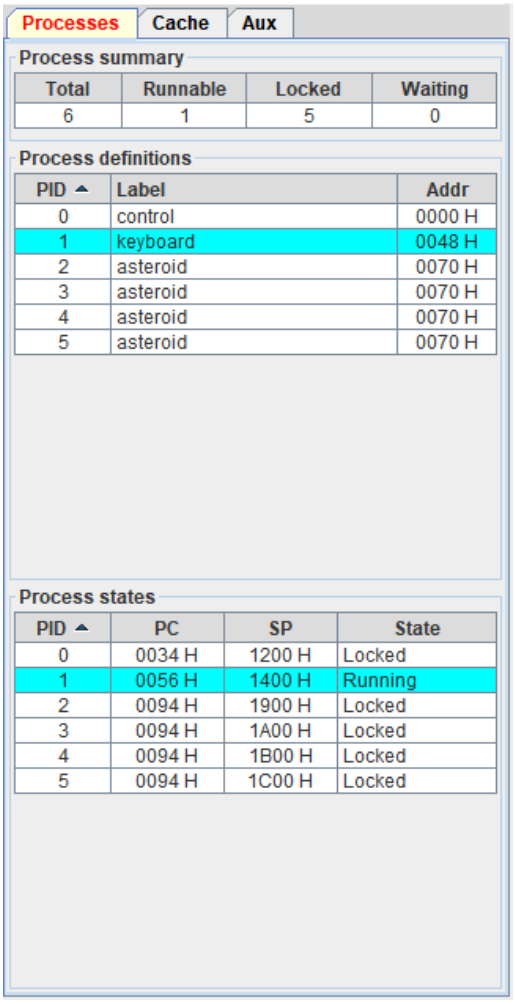
This tab displays information on the processes that have been created, in 3 tables:

- **Process summary table**. This is a one row table, with the total number of processes and the number of those that are Runnable (including the one currently running), Locked, and Waiting;
- **Process definitions table**, with information pertaining to the creation of processes: PID (process ID, a unique number for each process), label of the routine which the process executes, and the address of that routine (at which the process starts execution);
- **Process states table**, which provides the state of each process (using its PID): the current PC and SP register values, as well as the current status: Runnable, Running (only one process can have this status), Locked, or Waiting.

The last table appears only when the Refresh button is active. In this case, a blue bar indicates the current process.

Any of the columns of the process definitions and process states tables can be sorted, in ascending or descending order, by clicking the header cell and then the small triangle next to the column name.

The following figure illustrates this tab. Note that there are 4 processes with the same name. These are instances of processes that run the same routine, but each has its own stack area, and their current SP values are different.

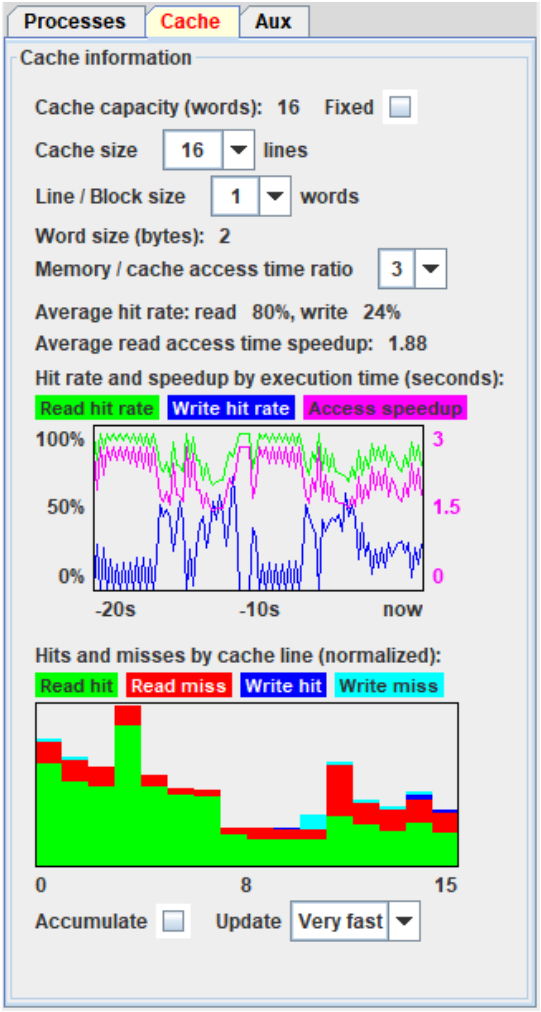


The Cache tab

The PEPE-16 processor includes a direct-mapped cache. This tab provides statistical information on it, as well as allowing to change some of its configuration settings. This includes:

- **Cache capacity (words)**. This is the product of the number of cache lines by the number of words in each line (or block). This capacity can be specified to be fixed, in which case (after checking the check box) increasing one number automatically decreases the other;
- **Cache size**: the number of cache lines;
- **Line/Block size**: the number of words in each line (or block);
- **Word size**, in bytes. In PEPE-16, this is 2;
- **Memory/cache access time ratio**: the number of times that a memory access is slower than a cache size (the higher the ratio, the more advantage in having a cache);
- **Average hit ratio** (in percentage), for read and for write operations. This depends on how the program goes, and on the size of the cache (the higher the size, the greater the hit ratio);
- **Average read access time speedup**. This is the average speedup gain of using the cache, which stems from read accesses. Write accesses are actually slower than a write access without cache, since a write operation updates both the cache and the memory. However, reads are much more frequent than writes;
- **Hit rate and speedup by execution time**. A chart showing the evolution, along the last 20 seconds of program execution, of the read and write hit rates, and of the access speedup;
- **Hits and misses by cache line (normalized)**. A chart showing the number of access hits and misses (normalized to the maximum value) in each line, during the last update period. Each type of access has a different color;
- **Accumulate**. If checked, the hits and misses charts shows accumulated statistics, instead of just those of the last update period;
- **Update**. The update frequency can be defined here, from slow to very fast, or even none (which freezes the chart).

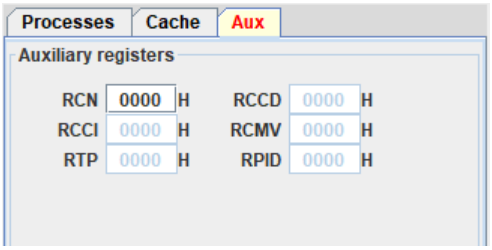
The following figure illustrates the cache tab.



The Auxiliary registers tab

In addition to the main registers, always shown in the simulation interface, the PEPE-16 processor includes a few auxiliary registers, described in detail in section A.2.2 of the book underlying this simulator, “Arquitetura de Computadores”, 5th edition, ISBN 978-972-722-789-1, by José Delgado and Carlos Ribeiro (book in Portuguese).

However, in the current implementation only the RCN auxiliary register is used. The others appear disabled in this tab:

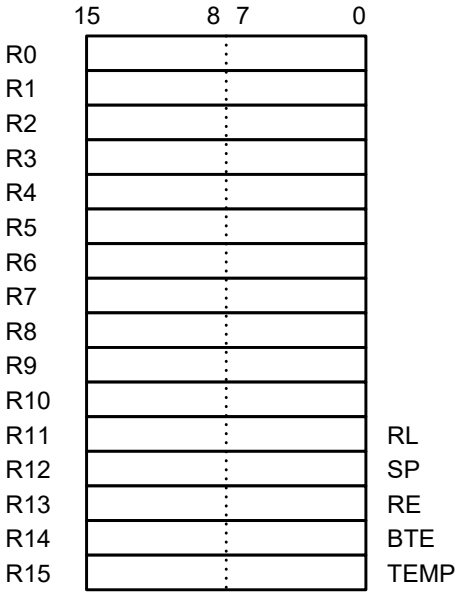


Section 6.5.1.7 describes the RCN register.

6.5.1.5 Main registers

The register bank of the PEPE-16 has 16 registers, each with 16 bits. Some instructions can access these registers in two halves: byte high (bits 15..8) and byte low (bits 7..0).

Some of these registers have special roles and names, as shown in the following figure. R0 .. R10 are completely general-purpose and can be used freely in programs. R11 can also be used, unless CALLF and RETF instructions are used. Even if instructions allow equal access to all 16 registers, the R12..R15 should not be explicitly used by programs, since they are used to implement instructions.



The PC register is not included in the bank register and is accessed only by specific instructions, such as calls and jumps. The following table describes the registers in further detail.

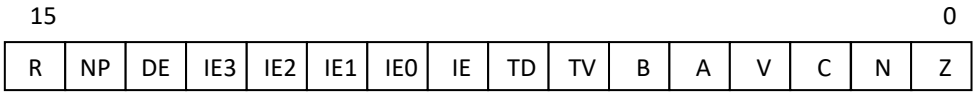
Abbreviation	Description
PC	Program Counter, the address of the next instruction to execute.
R0 a R10	General-purpose registers
RL or R11	Link register, used to store the PC between CALLF and RETF instructions. When these instructions are not used, can also be used as a general-purpose register
SP or R12	Stack Pointer, used to address the last used word in the stack. Used in instructions such as CALL, RET, PUSH, and POP.
RE or R13	State register (flags, described in detail below)
BTE or R14	Base of the exception table (see section 6.5.1.6)
TEMP or R15	Temporary register, used in the implementation of some instructions (not to be used explicitly in the program)

The value of the PC register must always be even, since all instructions occupy one word (2 bytes). Upon reset, the processor starts execution at address 0000H, so the PC is initialized with this value.

The R11 register (link register) can be used to optimize a call to a routine that does not invoke another. The return address is stored in RL (R11) and the RETF instruction returns from the routine by reading the RL. This mechanism is seldom used, therefore R11 can usually be added to the set of general-purpose registers when CALLF and RETF instructions are not used.

The SP register must be initialized with the even address immediately following the area reserved for the stack. CALL and PUSH instructions decrease the value of SP by 2 and RET and POP instructions increase the value of SP by 2. A correct program always matches the instructions that increase and decrease the SP.

The following figure and table detail the state register and its bits (flags).



Bit	Abbrev.	Description
0	Z	Zero. This bit is set to 1 by operations that yield a zero result.
1	N	Negative. This bit is set to 1 by operations that yield a negative result (highest order bit at 1).
2	C	Carry. This bit is set to 1 by operations that produce a carry.
3	V	Overflow. This bit is set to 1 by operations that yield a result too large to be represented correctly with 16 bits.
4	A	Auxiliary flag to be freely used by the programmer.
5	B	Auxiliary flag to be freely used by the programmer.
6	TV	Trap on overflow. If set to 1, the EXCESS exception is generated on the instruction that produced the overflow. If set to 0, only the V flag is updated.
7	TD	Trap on division by zero. If set to 1, the DIV0 exception is generated on the instruction that produced the division by zero. If set to 0, the division is not executed and the situation is ignored.
8	IE	Global interrupt Enable. If this bit is 0, all interrupts are disabled and interrupt requests are ignored.
9	IE0	Interrupt Enable for interrupt 0. If this bit is 0, interrupt 0 is disabled and the respective interrupt requests are ignored.
10	IE1	Interrupt Enable for interrupt 1. If this bit is 0, interrupt 1 is disabled and the respective interrupt requests are ignored.
11	IE2	Interrupt Enable for interrupt 2. If this bit is 0, interrupt 2 is disabled and the respective interrupt requests are ignored.
12	IE3	Interrupt Enable for interrupt 3. If this bit is 0, interrupt 3 is disabled and the respective interrupt requests are ignored.
13	DE	DMA Enable. The current implementation does not support DMA and this bit is ignored.
14	NP	Privilege level. Sets the current privilege level: 0 = System; 1 = User.
15	R	Reserved for future use.

6.5.1.6 *Exceptions*

Exceptions are (typically infrequent) events that constitute alterations to the normal flow of instructions of a program.

The cause of exceptions can be external (activation of a processor pin) or internal (due to the execution of specific instructions, such as division by zero).

Interrupts are external exceptions caused by the activation of one of the interrupt pins of the PEPE-16 (**INT0** to **INT3**). These are asynchronous exceptions, since they can occur in any instruction executed by the program. They correspond to exceptions 0 to 3, respectively.

The occurrence of an exception causes the processor to invoke a routine that must deal with the exception. In the case of interrupts, this occurs only if the global enable interrupt flag (IE) and the specific enable interrupt flag (IE0 to IE3) are set to 1. When the exception routine finishes, it returns to the instruction following the last one executed before invoking the exception routine.

Checking whether an interrupt has been requested occurs between instructions only (never in the middle of an instruction).

There must be a table with the addresses of all the exception routines. The BTE register must be initialized with the base of the exception table, which is the address of the interrupt 0 routine.

If several interrupts have been requested and are pending when the processor is able to execute an interrupt routine, interrupt 0 has the highest priority (therefore, serviced first) and interrupt 3 has the lowest priority (therefore, serviced last).

The execution of interrupt routines cannot be interrupted by another interrupt request, unless interrupts are explicitly enabled before returning, with the EI instruction. This is a design feature, which automatically sets IE=0 when the interrupt routine is invoked. This way, an interrupt routine, which usually performs critical operations, has the assurance that it cannot be interrupted, unless it wants to.

The following table describes the most relevant exceptions supported by the PEPE-16. The last three cannot be disabled (masked).

Number	Exception	Cause	Occurs at	Maskable
0	INT0	Pin INT0 is activated (com IE=1, IE0=1).	Anytime	Yes
1	INT1	Pin INT1 is activated (com IE=1, IE1=1)	Anytime	Yes
2	INT2	Pin INT2 is activated (com IE=1, IE2=1)	Anytime	Yes
3	INT3	Pin INT3 is activated (com IE=1, IE3=1)	Anytime	Yes
4	EXCESS	An arithmetic operation yields an overflow (with TV=1 in RE)	Instruction execution	Yes
5	DIV0	A division operation fails because the quotient is zero (with TD=1 in RE)	Instruction execution	Yes
6	COD_INV	The control unit finds an invalid opcode when decoding an instruction	Instruction decoding	No
7	D_UNALIGNED	A 16-bit (full word) memory data access is made with an odd address (unaligned data address)	Instruction execution	No
8	I_ UNALIGNED	A 16-bit (full word) memory instruction fetch is made with an odd address (unaligned instruction address)	Instruction fetch	No

6.5.1.7 Auxiliary registers

RCN, the only auxiliary register available in this implementation of the simulator, supports the configuration (during program execution, by instructions) of some aspects of the processor, described in the following table.

The most useful aspect is how the interrupt pins recognize a request for an interrupt, either by a pin value change (rising or falling edge) or by a specific pin value level (0 or 1). Edge-triggered requests are memorized until serviced, but only once (multiple pending requests are not distinguishable). Level-triggered requests are not memorized and are serviced as long as they are enabled and the pin stays at the required value level.

Bits	Abbrev.	Description
1..0	NSI0	Interrupt request mode for pin INT0: 00=rising edge; 01=falling edge; 10=level 1; 11=level 0
3..2	NSI1	Interrupt request mode for pin INT1: 00=rising edge; 01=falling edge; 10=level 1; 11=level 0
5..4	NSI2	Interrupt request mode for pin INT2: 00=rising edge; 01=falling edge; 10=level 1; 11=level 0
7..6	NSI3	Interrupt request mode for pin INT3: 00=rising edge; 01=falling edge; 10=level 1; 11=level 0
8	FR	Processor's clock source: 0=internal clock; 1=external clock, connected to the CLOCK_IN
9	E	Pipeline processing (not available in this implementation of the simulator)
15..10		Reserved for future use

6.5.1.8 PEPE-16 instruction set (assembly language)

6.5.1.8.1 The structure of instructions

The PEPE-16 is a 16-bit processor, which means that it is able to deal with 16-bit words in a single operation or data transfer. Its registers and data bus are 16 bits wide. Since addresses can also be used as data, they also have 16 bits.

In addition, the processor supports sub-word addressing, being able to deal with just 8-bit (one byte) values. This means that most instructions deal with 16-bit data, but some deal with 8-bit data. All memory space addresses refer to each byte individually, which means that 16-bit values occupy 2 addresses. By default, PEPE-16 is Big-Endian, i.e., the higher order byte of each word is stored at the lower (even) address and the lower order byte of each word is stored at the higher (odd) address.

To simplify the processor's internal hardware, all instructions occupy exactly one full word (16 bits), and therefore occupy 2 addresses. There are just a few data transfer instructions that need more, since they contain a 16-bit constant value. The user still sees only one instruction in the assembly language program, but during machine code generation these are automatically divided into two one-word instruction, each dealing with one of the two bytes of the 16-bit constant.

Each instruction must have at least a mnemonic, which identifies it and is related to what it does. Depending on the instruction, it may have zero, one or two operands, either a register or a constant.

With two operands, the first (the destination, or target, of a data transfer) must be a register. There are also cases in which one operand is obtained by the sum of two registers.

6.5.1.8.2 Addressing modes

An addressing mode indicates how to obtain an operand, to be used in an instruction. The following table describes the addressing modes supported by the PEPE-16, in which Ri and Rj designate any register (from R0 to R11, at most; do not use R12 to R15) and the square brackets notation refers to a memory cell contents, using the value between brackets as its address.

Addressing mode	How to obtain the operand	Operand used as	No. of bits used in the instruction	Examples of instructions
Immediate	Constant	Data	4	ADD R1, 3
			16	MOV R2, 3456H
Register	Ri	Data	4	ADD R1, R2
Direct	[Constant]	Address	16	MOV R2, [3456H]
Indirect	[Ri]	Address	4	MOV R1, [R2]
Based	[Ri + constant]	Address	4 + 4	MOV R1, [R2+6]
Indexed	[Ri + Rj]	Address	4 + 4	MOV R1, [R2+R3]
Relative	Constant	Address	8	JZ 100H
			12	JMP 100H, CALL 100H
Implicit	SP, [SP]	Address	0	PUSH, POP
	SP, [SP], PC	Address	0	RET, CALL

NOTE – The Direct addressing mode was not supported at the time the book mentioned in section 1 was written. Now it can be used;

– The number of bits used by the operand is important. The more operands an instruction has (and the more bits each operand uses), the less bits are available to specify the mnemonic, and everything has to fit into 16 bits. Direct addressing and immediate addressing with 16 bits need to separate into two instructions, each dealing with one byte of the 16-bit constants;

– In addition, the number of bits used by the operand determines its value range. For example, a 4-bit operand can only specify values between -8 and +7.

These modes can be defined in the following manner:

- **Immediate**. The operand is specified as a data constant in the instruction itself;
- **Register**. The operand is the value of the specified register;
- **Direct**. The operand is an address constant between square brackets, with up to 16 bits, to cover the whole address space;
- **Indirect**. The operand is the value of the specified register, which is used as an address (between square brackets) to access the memory cell where the actual operand to use is;

- **Based**. The operand is like in the Indirect mode, but with a constant added to a register. This is useful to access tables, in which the register contains the table's base address and the constant is used as a fixed offset to address subsequent elements. The only instructions that use this mode access full words, so the address must be even. Since there are 4 bits to code the constant value, it can vary between -16 and +14 (restricted to even values), doubling the range compared to use any value between -8 and +7. The implies that the value of the register must be even as well, which is not a problem since all full-word tables start at an even address;
- **Indexed**. The operand is like in the Based mode, but with a register added to another register. This is useful to access tables, in which one of the registers contains the table's base address and the other register is used as an index, or variable offset, to address subsequent elements. The only instructions that use this mode access full words, so the address (sum of two registers) must be even;
- **Relative**. This is used in jump and call instructions, in which what is encoded in the machine code instruction is the difference between the constant specified in the assembly source instruction and the address at which the instruction is located. When the instruction is executed, that difference is added to the PC (Program Counter) register to obtain the actual target address which to jump to or to call. This way, it is not necessary to use 16-bit constants, at the expense of a limited jump or call address range (other instructions exist to solve this limitation). Note that conditional jumps have only 8 bits to specify the constant, which means that they cannot jump to a very far instruction address;
- **Implicit**. The operand is not explicitly specified in the instruction, but rather assumed by the instruction's semantics to obtain it from a predetermined source, namely the SP or PC registers.

6.5.1.8.3 Data transfer instructions

These constitute the most used type of instructions, transferring (16- or 8-bit) data between data holders (registers and memory). Although most of the instructions have a mnemonic derived from "move", they have a copy semantics. The data source is never destroyed, unless it is used explicitly as data destination as well.

The PEPE-16 is a processor with a load-store architecture, which means that operations on data are only performed with register data. Therefore, any operation on memory data requires transferring them to registers first, performing the operations, and then transferring them back to memory. This greatly simplifies the architecture.

The following table illustrates the most common data transfer instructions, in which Rs, Rd, and Ri designate registers with the role of source, destination, and index, respectively.

Instruction	Addressing mode	Data transfer	Typical usage
MOV Rd, constant	Immediate	Copy constant to register	Initializing registers
MOV Rd, Rs	Register	Copy one register to another	Saving one register in another
MOV Rd, [constant]	Direct	Memory read (16 bits)	Data transfer(16 bits) between memory and registers
MOV Rd, [Rs]	Indirect		
MOV Rd, [Rs + constant]	Based		
MOV Rd, [Rs + Ri]	Indexed		
MOV [constant], Rs	Direct	Memory write (16 bits)	
MOV [Rd], Rs	Indirect		
MOV [Rd + constant], Rs	Based		
MOV [Rd + Ri], Rs	Indexed		
MOVB Rd, [Rs]	Indirect	Memory read (8 bits)	Individual byte processing (e.g., string operations)
MOVB [Rd], Rs	Indirect	Memory write (8 bits)	
SWAP Rd, Rs	Register	Atomic swap of two register values	Data swap, concurrent programming semaphores
SWAP Rd, [Rs] SWAP [Rs], Rd	Indirect	Atomic data swap (16 bits) between registers and memory. Even with cache, the memory access is enforced	
PUSH Rd	Implicit (SP)	Stack write	Save a register value in the stack to retrieve later on
POP Rd	Implicit (SP)	Stack read	Retrieve a value stored in the stack

NOTE – Data transfer instructions do not affect the flags, since they do not entail an operand transforming operation.

Section 6.5.1.8.9 provides the complete set of instructions.

6.5.1.8.4 Arithmetic instructions

These instructions perform arithmetic operations on their operands, setting up the Z, N, C, and V flags of the RE register according to the value of the result. Operand and result values of arithmetic instructions are always considered in two's complement notation.

The following table provides a simplified description of the arithmetic instructions. The result is always stored in a register (Rd). Note that this register is also an operand.

Instruction		Description
ADD	Rd, Rs	Adds a register (Rd) to another register (Rs) or to a constant. The constant can only have 4 bits (-8 .. +7)
ADD	Rd, constant	
ADDC	Rd, Rs	Adds two registers and the value of the C (carry) flag (1 or 0)
INC	Rd	Increments register Rd by 1
SUB	Rd, Rs	Subtracts a register (Rs) or a constant from a register (Rd). The constant can only have 4 bits (-8 .. +7)
SUB	Rd, constant	
SUBB	Rd, Rs	Subtracts a register (Rs) and the value of the C (carry) flag (1 or 0) from a register (Rd). The carry flag works as a borrow.
DEC	Rd	Decrements register Rd by 1
CMP	Rd, Rs	Same as SUB, but does not affect the value of Rd, only the flags. Typically used for tests, in conjunction with conditional jumps
CMP	Rd, constant	
MUL	Rd, Rs	Multiplies the two registers
DIV	Rd, Rs	Obtains the quotient of the integer division of Rd by Rs
MOD	Rd, Rs	Obtains the remainder of the integer division of Rd by Rs
NEG	Rd	Obtains the negation of the value of Rd (if positive, becomes negative with the same modulus, and vice-versa)

Section 6.5.1.8.9 provides the complete set of instructions.

6.5.1.8.5 Logic instructions

These instructions implement boolean logic operations, bit by bit.

The following table provides a simplified description of the logic instructions. The result is always stored in a register (Rd). Note that this register is also an operand.

Instruction		Description
AND	Rd, Rs	Performs the AND operation on two registers, bit by bit
OR	Rd, Rs	Performs the OR operation on two registers, bit by bit
NOT	Rd	Performs the NOT operation on each of the bits of one register
XOR	Rd, Rs	Performs the exclusive-or operation on two registers, bit by bit
TEST	Rd, Rs	Same as AND, but does not affect the value of Rd, only the flags. Typically used for testing masks, in conjunction with conditional jumps

Section 6.5.1.8.9 provides the complete set of instructions.

6.5.1.8.6 Bit manipulation instructions

These instructions implement bit operations on a specific bit of a register.

The following table provides a simplified description of the bit manipulation instructions. The result is always stored in a register (Rd). Note that this register is also an operand.

Instruction	Description
BIT Rd, n	Sets the Z flag to 1 if bit n of register RD is 0 (and vice-versa). Typically used for tests, in conjunction with conditional jumps
SET Rd, n	Sets bit n of register RD to 1, leaving the others unchanged
EI	Same as SET with n equal to the number of the IE flag in RE
EI0	Same as SET with n equal to the number of the IE0 flag in RE
EI1	Same as SET with n equal to the number of the IE1 flag in RE
EI2	Same as SET with n equal to the number of the IE2 flag in RE
EI3	Same as SET with n equal to the number of the IE3 flag in RE
SETC	Same as SET with n equal to the number of the C flag in RE
CLR Rd, n	Sets bit n of register RD to 0, leaving the others unchanged
DI	Same as CLR with n equal to the number of the IE flag in RE
DI0	Same as CLR with n equal to the number of the IE0 flag in RE
DI1	Same as CLR with n equal to the number of the IE1 flag in RE
DI2	Same as CLR with n equal to the number of the IE2 flag in RE
EI3	Same as CLR with n equal to the number of the IE3 flag in RE
CLRC	Same as CLR with n equal to the number of the C flag in RE
CPL Rd, n	Complements bit n of register RD (1 to 0, or 0 to 1), leaving the others unchanged
CPLC	Same as CPL with n equal to the number of the C flag in RE

Section 6.5.1.8.9 provides the complete set of instructions.

6.5.1.8.7 Shift and rotate instructions

These instructions perform shift and rotate operations on a register, up to 15 bits at a time, leftward or rightward. Shift instructions lose bits on one side of the register, with new bits entering on the other side. Rotate instructions make lost bits on one side reenter on the other side.

The following table provides a simplified description of the shift and rotate instructions. The result is always stored in a register (Rd). Note that this register is also an operand.

If n is 0, no change is made to Rd, but the Z and N flags are still updated, based on the value of Rd.

Instruction	Description
SHR Rd, n	Shifts the value of Rd by n bits do the right. The C flag is set with the last bit lost on the right. The n leftmost bits of Rd are set to 0. Note that negative numbers become positive, since the leftmost bit is set to 0.
SHL Rd, n	Shifts the value of Rd by n bits do the left. The C flag is set with the last bit lost on the left. The n rightmost bits of Rd are set to 0. Equivalent to the multiplication of Rd by 2^n.
SHRA Rd, n	Shift right arithmetic, considering that Rd has a number in two's complement notation. Same as SHR, but the n leftmost bits of Rd are set to the value of bit 15 of the original value of Rd. Therefore, the signal bit is maintained. Equivalent to the integer division of Rd by 2^n.
SHLA Rd, n	Same as SHL. Available only for symmetry with SHRA.
ROR Rd, n	Shifts the value of Rd by n bits do the right, with each bit moved out on the right entering again on the left. The C flag is set with the last bit moved out on the right.
ROL Rd, n	Shifts the value of Rd by n bits do the left, with each bit moved out on the left entering again on the right. The C flag is set with the last bit moved out on the left.
RORC Rd, n	Rotate right through the flag C. Same as ROR, but considering that Rd has 17 bits, with the C concatenated on the right side of Rd (bit 0)
ROLC Rd, n	Rotate left through the flag C. Same as ROL, but considering that Rd has 17 bits, with the C concatenated on the left side of Rd (bit 15)

Section 6.5.1.8.9 provides the complete set of instructions.

6.5.1.8.8 Control flow instructions

Unlike other instructions, which perform an operation and proceed to execute the instruction at the next address, control flow instructions can determine that the next instruction to execute next is not located at the next address. Jumps and calls to routines belong to this group

The main result of these instructions is to set the PC (Program Counter) register with either the next address (the normal instruction execution sequencing) or another address, specified by the instruction through a label (see section 7.1.1.1).

Instructions that deal with routines, such as calls, however, have other side effects.

One of the most common type of control flow instructions is conditional jumps, which depend on the value of one or more of the Z, N, C, and V flags, in the RE register. These instructions can just test the value of a given flag, but their true usefulness lies in implementing tests after a CMP instruction (section 6.5.1.8.4), which compares two registers and affects the flags according to their relative values.

A CMP instruction followed by a conditional jump, both using the flags, is the equivalent in assembly language of if statements in higher level programming languages. If some combination of the values of these flags is true, the next instruction to execute will the one at the address indicated by the

label (a jump is made). If the condition is false, there is no jump and the instruction at the next address will be executed after the conditional jump.

The following table provides a simplified description of the most common conditional jump instructions.

Instruction	Description	Use to
JZ label	Jump if zero (Z=1)	Test the value of a flag or the sign of a signed value (in two's complement notation), by comparing it with 0
JNZ label	Jump if not zero (Z=0)	
JN label	Jump if negative (N=1)	
JNN label	Jump if not negative (N=0)	
JP label	Jump if positive (N=0 and Z=0)	
JNP label	Jump if not positive (N=1 or Z=1)	
JC label	Jump if carry (C=1)	
JNC label	Jump if not carry (C=0)	
JV label	Jump if overflow (V=1)	
JNV label	Jump if not overflow (V=0)	
JA label	Jump if above (C=0 and Z=0)	Compare two unsigned values
JAE label	Jump if above or equal (C=0)	
JB label	Jump if below (C=1)	
JBE label	Jump if below or equal (C=1 or Z=1)	
JEQ label	Jump if equal (Z=1)	Compare two signed values (in two's complement notation)
JNE label	Jump if not equal (Z=0)	
JLT label	Jump if less than (N≠V)	
JLE label	Jump if less or equal (N≠V or Z=1)	
JGT label	Jump if greater than (N=V and Z=0)	
JGE label	Jump if greater or equal (N=V)	

NOTE – The machine code instruction can only use 8 bits to specify the label constant. It uses relative addressing mode (section 6.5.1.8.2) and actually encodes in the instruction is the difference between the value of label and the address at which the conditional jump is located. There are only 8 bits to specify this difference.

The following table provides a simplified description of other commonly used control flow instructions. Note that the functionality reflects the result and not the actual operations performed.

Instruction	Description	Equivalent functionality
JMP label	Jumps unconditionally to the address specified by label, which cannot differ from the instruction address by more than 2^{12} addresses, since there are only 12 bits to encode the difference	PC ← label
JMP Rs	Jumps unconditionally to the address specified by the register Rs. This instruction enables jumping to any address in the address space	PC ← Rs
CALL label	Invokes the routine at the address specified by label, which cannot differ from the instruction address by more than 2^{12} addresses, since there are only 12 bits to encode the difference	save return address in stack PC ← label
CALL Rs	Invokes the routine at the address specified by the register Rs. This instruction enables calling a routine at any address in the address space	save return address in stack PC ← Rs
RET	Returns from a routine, to the return address saved by the latest executed CALL instruction	PC ← return address (from stack)
RFE	Returns from an exception routine, to the address of the instruction that would be executed if the exception routine had not been invoked	RE ← saved RE (from stack) PC ← return address (from stack)

Section 6.5.1.8.9 provides the complete set of instructions.

6.5.1.8.9 Instruction set reference table

This section provides a description of the complete instruction set of the PEPE-16 processor.

The notational conventions that were adopted are described in the following table.

Notation	Meaning
Rd, Rs, Ri	Main register (R0 a R15). Specific names can also be used, e.g., PC, SP, RE, BTE, TEMP, USP
Ad, As	Auxiliary register
Z, N, C, V, IE, IE0 to IE3, etc.	Flags (bits of the state register, RE)
K	Constant, with 4, 8, 12, or 16 bits (depending on the instruction)

Notation	Meaning
XXXX	4-bit field with bits that are ignored
ANI	Address of the next instruction. Not a register, simply a notation that represents the address of the current instruction increased by 2 (since each instruction occupies 2 addresses)
Mw[*address*]	16-bit memory cell located at *address* and *address*+1 (*address* must be even). PEPE-16 is a Big_endian processor, which means that the highest order byte is located at *address* and the lower order byte is located at *address*+1
Mb[*address*]	8-bit memory cell located at *address* (which can be even or odd)
(i)	Bit *i* of a register or of a memory cell
(i..j)	Contiguous bits *i* to *j* of a register or of a memory cell (*i* >= *j*)
<n>	Repetition *n* times, e.g., 0<4> is equivalent to 0000
dest ← *expr*	Assignment of the value of an expression (*expr*) to a register or memory cell (*dest*)
expr : *action*	Execute *action* if the boolean expression *expr* evaluates to true
∧, ∨, ⊕	Bitwise logic operations: AND, OR, Exclusive-OR
||	Bit sequence concatenation (the bits at the left of the operator stay at the left, or with a higher bit order)

The following table describes each of the available instructions, including:

- The assembly syntax;
- Each of the four 4-bit fields (bits 15..12, 11..8, 7..4, and 3..0) in which each instruction is divided, providing the detailed encoding of each instruction into binary form. This can be helpful when inspecting code in memory. In these columns, the names of registers and of flags must be replaced by their index in the register bank and in the RE register, respectively;
- The detailed actions executed by each instruction, using the conventions in the table above;
- The flags affected by the instruction;
- Any relevant comments.

NOTE – Some MOV instructions generate two machine code instructions, so that they are able to deal with 16-bit constants. This is automatic and done by the compiler. In the PEPE-16's simulation interface (section 6.5.1.4) only one instruction appears, even in the views that show code, but in the address column each of these instructions occupies 4 bytes, not 2. The actual generated instructions can only be seen by inspecting the contents of the memory.

Assembly syntax		Bits 15..12	Bits 11..8	Bits 7..4	Bits 3..0	Actions	Affected flags	Comments
ADD	Rd, Rs	0101	0000	Rd	Rs	Rd ← Rd + Rs	Z, N, C, V	
ADD	Rd, K	0101	0001	Rd	K	Rd ← Rd + K	Z, N, C, V	K ∈ [-8 .. +7]
ADDC	Rd, Rs	0101	0010	Rd	Rs	Rd ← Rd + Rs + C	Z, N, C, V	
INC	Rd	0000	1111	0000	Rd	Rd ← Rd + 1	Z, N, C, V	
SUB	Rd, Rs	0101	0011	Rd	Rs	Rd ← Rd – Rs	Z, N, C, V	
SUB	Rd, K	0101	0100	Rd	K	Rd ← Rd - K	Z, N, C, V	K ∈ [-8 .. +7]
SUBB	Rd, Rs	0101	0101	Rd	Rs	Rd ← Rd – Rs - C	Z, N, C, V	
DEC	Rd	0000	1111	0001	Rd	Rd ← Rd - 1	Z, N, C, V	
CMP	Rd, Rs	0101	0110	Rd	Rs	(Rd – Rs)	Z, N, C, V	Rd is not changed
CMP	Rd, K	0101	0111	Rd	K	(Rd – K)	Z, N, C, V	K ∈ [-8 .. +7] Rd is not changed
MUL	Rd, Rs	0101	1000	Rd	Rs	Rd ← Rd * Rs	Z, N, C, V	Rs is changed
DIV	Rd, Rs	0101	1001	Rd	Rs	Rd ← quotient (Rd / Rs)	Z, N, C, V←0	Integer division
MOD	Rd, Rs	0101	1010	Rd	Rs	Rd ← remainder (Rd / Rs)	Z, N, C, V←0	Integer division
NEG	Rd	0101	1011	Rd	XXXX	Rd ← –Rd	Z, N, C, V	Two's complement (negation) V←1 if Rd=8000H
AND	Rd, Rs	0110	0000	Rd	Rs	Rd ← Rd ∧ Rs	Z, N	
OR	Rd, Rs	0110	0001	Rd	Rs	Rd ← Rd ∨ Rs	Z, N	
NOT	Rd	0110	0010	Rd	XXXX	Rd ← Rd ⊕ FFFFH	Z, N	One's complement
XOR	Rd, Rs	0110	0011	Rd	Rs	Rd ← Rd ⊕ Rs	Z, N	Exclusive or
TEST	Rd, Rs	0110	0100	Rd	Rs	(Rd ∧ Rs)	Z, N	Rd is not changed
BIT	Rd, n	0110	0101	Rd	n	Z ← Rd(n) ⊕ 1	Z	Rd is not changed
SET	Rd, n	0110	0110	Rd	n	Rd(n) ← 1	Z, N, or another if Rd is RE	n ∈ [0 .. 15] If Rd=RE, affects only RE(n)

Assembly syntax	Bits 15..12	Bits 11..8	Bits 7..4	Bits 3..0	Actions	Affected flags	Comments
EI	0110	0110	RE	IE	RE(IE) ← 1	EI	Enable interrupts (global)
EI0	0110	0110	RE	IE0	RE (IE0) ← 1	EI0	Enable interrupt 0
EI1	0110	0110	RE	IE1	RE (IE1) ← 1	EI1	Enable interrupt 1
EI2	0110	0110	RE	IE2	RE (IE2) ← 1	EI2	Enable interrupt 2
EI3	0110	0110	RE	IE3	RE (IE3) ← 1	EI3	Enable interrupt 3
SETC	0110	0110	RE	C	RE (C) ← 1	C	Set Carry flag
EDMA	0110	0110	RE	DE	RE (DE) ← 1	DE	Enable DMA
CLR Rd, n	0110	0111	Rd	n	Rd(n) ← 0	Z, N, or another if Rd is RE	n ∈ [0 .. 15] If Rd=RE, affects only RE(n)
DI	0110	0111	RE	IE	RE (IE) ← 0	EI	Disable interrupts (global)
DI0	0110	0111	RE	IE0	RE (IE0) ← 0	EI0	Disable interrupt 0
DI1	0110	0111	RE	IE1	RE (IE1) ← 0	EI1	Disable interrupt 1
DI2	0110	0111	RE	IE2	RE (IE2) ← 0	EI2	Disable interrupt 2
DI3	0110	0111	RE	IE3	RE (IE3) ← 0	EI3	Disable interrupt 3
CLRC	0110	0111	RE	C	RE (C) ← 0	C	Clear Carry flag
DDMA	0110	0111	RE	DE	RE (DE) ← 0	DE	Disable DMA
CPL Rd, n	0110	1000	Rd	n	Rd(n) ← Rd(n) ⊕ 1	Z, N, or another if Rd is RE	n ∈ [0 .. 15] If Rd=RE, affects only RE(n)
CPLC	0110	1000	RE	C	RE (C) ← RE(C) ⊕ 1	C	Complement Carry flag
SHR Rd, n	0110	1001	Rd	n	n>0 : C ← Rd(n-1) n>0 : Rd ← 0<n> || Rd(15..n)	Z, N, C	n ∈ [0 .. 15] If n=0, affect Z and N (not C)
SHL Rd, n	0110	1010	Rd	n	n>0 : C ← Rd(15-n+1) n>0 : Rd ← Rd(15-n..0) || 0<n>	Z, N, C	n ∈ [0 .. 15] If n=0, affect Z and N (not C)
SHRA Rd, n	0101	1100	Rd	n	n>0 : C ← Rd(n-1) n>0 : Rd ← Rd(15)<n> || Rd(15..n)	Z, N, C	n ∈ [0 .. 15] If n=0, affect Z and N (not C)

Assembly syntax		Bits 15..12	Bits 11..8	Bits 7..4	Bits 3..0	Actions	Affected flags	Comments
SHLA	Rd, n	0101	1101	Rd	n	n>0 : C ← Rd(15-n+1) n>0 : Rd ← Rd(15-n..0) || 0<n>	Z, N, C, V	n ∈ [0 .. 15] If n=0, affect Z and N (not C) V ← 1 if any of the outgoing bits differs from Rd(15) after execution
ROR	Rd, n	0110	1100	Rd	n	n>0 : C ← Rd(n-1) n>0 : Rd ← Rd(n-1..0) || Rd(15..n)	Z, N, C	n ∈ [0 .. 15] If n=0, affect Z and N (not C)
ROL	Rd, n	0110	1101	Rd	n	n>0 : C ← Rd(15-n+1) n>0 : Rd ← Rd(15-n..0) || Rd(15..15-n+1)	Z, N, C	n ∈ [0 .. 15] If n=0, affect Z and N (not C)
RORC	Rd, n	0110	1110	Rd	n	n>0 : Rd || C ← Rd(n-2..0) || C || Rd(15..n-1)	Z, N, C	n ∈ [0 .. 15] If n=0, affect Z and N (not C)
ROLC	Rd, n	0110	1111	Rd	n	n>0 : C || Rd ← Rd(15-n+1..0) || C || Rd(15..15-n+2)	Z, N, C	n ∈ [0 .. 15] If n=0, affect Z and N (not C)
MOV	Rd, [Rs + offset]	0111	Rd	Rs	offset/2	Rd ← Mw[Rs + offset]	None	offset ∈ [-16 .. +14] (even)
MOV	Rd, [Rs]	0111	Rd	Rs	0000	Rd ← Mw[Rs + 0000]	None	
MOV	Rd, [Rs + Ri]	1000	Rd	Rs	Ri	Rd ← Mw[Rs + Ri]	None	
MOV	[Rd + offset], Rs	1001	Rs	Rd	offset/2	Mw[Rd + offset] ← Rs	None	offset ∈ [-16 .. +14] (even)
MOV	[Rd], Rs	1001	Rs	Rd	0000	Mw[Rd + 0000] ← Rs	None	
MOV	[Rd + Ri], Rs	1010	Rs	Rd	Ri	Mw[Rd + Ri] ← Rs	None	
MOVB	Rd, [Rs]	1011	0000	Rd	Rs	Rd ← 0<8> || Mb[Rs]	None	Bits 15..8 of Rd set to 0
MOVB	[Rd], Rs	1011	0001	Rd	Rs	Mb[Rd] ← Rs(7..0)	None	The byte adjacent to Mb[Rd] is not affected
MOVBS	Rd, [Rs]	1011	0010	Rd	Rs	Rd ← Mb[Rs](7)<8> || Mb[Rs]	None	Same as MOVB Rd, [Rs], but with sign extension to 16 bits
MOV	Rd, [K]	1100 1110	TEMP Rd	K(7..0) K(15..8)		TEMP ← K(7)<8> || K(7..0) Rd ← Mw[K(15..8) || TEMP(7..0)]	None	K ∈ [-32768 .. +32767] Automatically generates two instructions

Assembly syntax		Bits 15..12	Bits 11..8	Bits 7..4	Bits 3..0	Actions	Affected flags	Comments
MOV	[K], Rs	1100 1111	TEMP Rs	K(7..0) K(15..8)		TEMP ← K(7)<8> || K(7..0) Mw[K(15..8) || TEMP(7..0)] ← Rs	None	K ∈ [-32768 .. +32767] Automatically generates two instructions
MOV	Rd, K	1100	Rd	K		Rd ← K(7)<8> || K	None	If K ∈ [-128 .. +127], only one instruction is generated (K is sign extended to 16 bits)
		1100 1101	Rd Rd	K(7..0) K(15..8)		Rd ← K(7)<8> || K(7..0) Rd(15..8) ← K(15..8)	None	If K ∈ [-32768 .. -129] or K ∈ [+128 .. +32767], two instructions are generated
MOV	Rd, Rs	1011	0011	Rd	Rs	Rd ← Rs	None	
MOV	Ad, Rs	1011	0100	Ad	Rs	Ad ← Rs	None	Allowed in System privilege level only
MOV	Rd, As	1011	0101	Rd	As	Rd ← As	None	Allowed in System privilege level only
MOV	Rd, USP	1011	0110	Rd	XXXX	Rd ← USP	None	Allowed in System privilege level only
MOV	USP, Rs	1011	0111	XXXX	Rs	USP ← Rs	None	Allowed in System privilege level only
SWAP	Rd, Rs	1011	1000	Rd	Rs	TEMP ← Rd Rd ← Rs Rs ← TEMP	None	
SWAP	Rd, [Rs] or [Rs], Rd	1011	1001	Rd	Rs	TEMP ← Mw[Rs] Mw[Rs] ← Rd Rd ← TEMP	None	
PUSH	Rs	1011	1010	Rs	XXXX	Mw[SP-2] ← Rs SP ← SP – 2	None	
POP	Rd	1011	1011	Rd	XXXX	Rd ← Mw[SP] SP ← SP + 2	None	

Assembly syntax	Bits 15..12	Bits 11..8	Bits 7..4	Bits 3..0	Actions	Affected flags	Comments
PUSHC	1011	1100	XXXX	XXXX	Mw[SP-2] ← Ri SP ← SP – 2	None	Performs these actions for each of the registers R0 to R11
POPC	1011	1101	XXXX	XXXX	Ri ← Mw[SP] SP ← SP + 2	None	Performs these actions for each of the registers R0 to R11
JZ label	0001	0000	K = (label – ANI)/2		Z=1: PC ← ANI + (2*K)	None	label ∈ [ANI - 256 .. ANI + 254]
JNZ label	0001	0001	K = (label – ANI)/2		Z=0: PC ← ANI + (2*K)	None	label ∈ [ANI - 256 .. ANI + 254]
JN label	0001	0010	K = (label – ANI)/2		N=1: PC ← ANI + (2*K)	None	label ∈ [ANI - 256 .. ANI + 254]
JNN label	0001	0011	K = (label – ANI)/2		N=0: PC ← ANI + (2*K)	None	label ∈ [ANI - 256 .. ANI + 254]
JP label	0001	0100	K = (label – ANI)/2		(N∨Z)=0: PC ← ANI + (2*K)	None	label ∈ [ANI - 256 .. ANI + 254]
JNP label	0001	0101	K = (label – ANI)/2		(N∨Z)=1: PC ← ANI + (2*K)	None	label ∈ [ANI - 256 .. ANI + 254]
JC label	0001	0110	K = (label – ANI)/2		C=1: PC ← ANI + (2*K)	None	label ∈ [ANI - 256 .. ANI + 254]
JNC label	0001	0111	K = (label – ANI)/2		C=0: PC ← ANI + (2*K)	None	label ∈ [ANI - 256 .. ANI + 254]
JV label	0001	1000	K = (label – ANI)/2		V=1: PC ← ANI + (2*K)	None	label ∈ [ANI - 256 .. ANI + 254]
JNV label	0001	1001	K = (label – ANI)/2		V=0: PC ← ANI + (2*K)	None	label ∈ [ANI - 256 .. ANI + 254]
JA label	0001	1010	K = (label – ANI)/2		(C∨Z)=0 : PC ← ANI + (2*K)	None	label ∈ [ANI - 256 .. ANI + 254]
JAE label	0001	0111	K = (label – ANI)/2		C=0 : PC ← ANI + (2*K)	None	label ∈ [ANI - 256 .. ANI + 254]
JB label	0001	0110	K = (label – ANI)/2		C=1 : PC ← ANI + (2*K)	None	label ∈ [ANI - 256 .. ANI + 254]
JBE label	0001	1011	K = (label – ANI)/2		(C∨Z)=1: PC ← ANI + (2*K)	None	label ∈ [ANI - 256 .. ANI + 254]
JEQ label	0001	0000	K = (label – ANI)/2		Z=1: PC ← ANI + (2*K)	None	label ∈ [ANI - 256 .. ANI + 254]
JNE label	0001	0001	K = (label – ANI)/2		Z=0: PC ← ANI + (2*K)	None	label ∈ [ANI - 256 .. ANI + 254]
JLT label	0001	1100	K = (label – ANI)/2		N⊕V =1 : PC ← ANI + (2*K)	None	label ∈ [ANI - 256 .. ANI + 254]
JLE label	0001	1101	K = (label – ANI)/2		((N⊕V)∨Z)=1 : PC ← ANI + (2*K)	None	label ∈ [ANI - 256 .. ANI + 254]
JGT label	0001	1110	K = (label – ANI)/2		((N⊕V)∨Z)=0 : PC ← ANI + (2*K)	None	label ∈ [ANI - 256 .. ANI + 254]
JGE label	0001	1111	K = (label – ANI)/2		N⊕V =0 : PC ← ANI + (2*K)	None	label ∈ [ANI - 256 .. ANI + 254]
JMP label	0010	K = (label – ANI)/2			PC ← ANI + (2*K)	None	label ∈ [ANI-4096 .. ANI+4094]
JMP Rs	0000	0111	XXXX	Rs	PC ← Rs	None	Jumps to any address

Assembly syntax	Bits 15..12	Bits 11..8	Bits 7..4	Bits 3..0	Actions	Affected flags	Comments
CALL label	0011	K = (label – ANI)/2			Mw[SP-2] ← ANI PC ← ANI + (2*K) SP ← SP - 2	None	label ∈ [ANI-4096 .. ANI+4094]
CALL Rs	0000	0010	XXXX	Rs	Mw[SP-2] ← ANI PC ← Rs SP ← SP - 2	None	Calls a routine at any address
CALLF label	0100	K = (label – ANI)/2			RL ← ANI PC ← ANI + (2*K)	None	label ∈ [ANI-4096 .. ANI+4094]
CALLF Rs	0000	0011	XXXX	Rs	RL ← ANI PC ← Rs	None	Calls a routine at any address Uses RL to store return address
RET	0000	0100	XXXX	XXXX	PC ← Mw[SP] SP ← SP + 2	None	Gets return address from stack
RETF	0000	0101	XXXX	XXXX	PC ← RL	None	Gets return address from RL
SWE K	0000	0001	K		TEMP ← RE RE ← 0 Mw[SP-2] ← ANI Mw[SP-4] ← TEMP PC ← Mw[BTE+2*K] SP ← SP - 4	All flags set to zero	K ∈ [0 .. 255] Invokes a routine through the exception table. Allowed in System privilege level only
RFE	0000	0110	XXXX	XXXX	TEMP ← Mw[SP] PC ← Mw[SP+2] SP ← SP + 4 RE ← TEMP	All flags restored	Return from exception routine Allowed in System privilege level only
NOP	0000	0000	XXXX	XXXX		None	No operation (does nothing)

6.5.2 Module PEPE-8

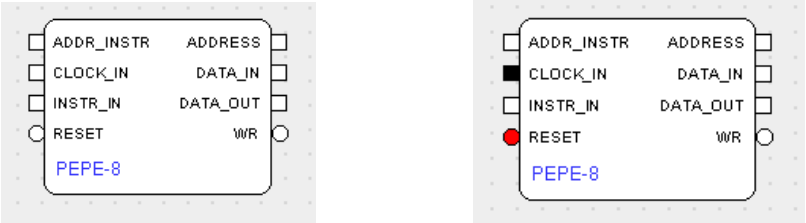
The following sections describe only the features of the PEPE-8. Sections 7.3 and 7.4 describe how to program and to debug it, respectively.

6.5.2.1 General description

The PEPE-8 processor module is a very simple, 8-bit processor, designed to illustrate the most basic principles of processor-based systems. It can only support very simple applications.

Each instruction is executed in a single clock cycle, which imposes that there must be two memories: code (for instructions) and data (for variables). In addition, the data memory needs to have separate input and output data pins.

The followings figures illustrate the PEPE-8 module, as it appears in the circuit area (left) and with some input pins forced to 0 or 1 (right), as described below.



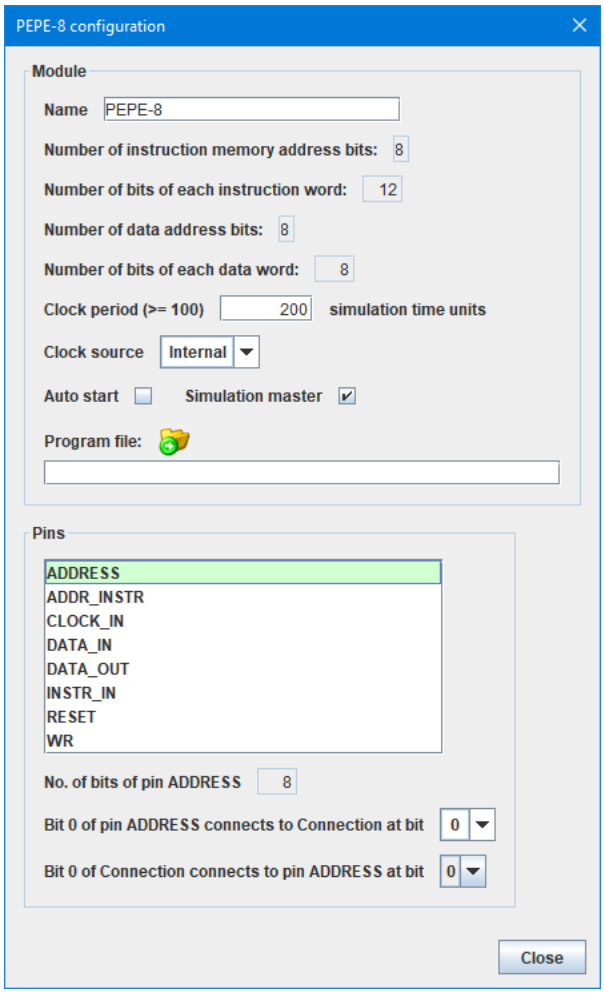
The PEPE-8 module has the following pins:

- **ADDR_INSTR**. This 8-bit output pin allows the processor to specify which instruction memory cell is to be read, providing the instruction to be executed;
- **CLOCK_IN**. This pin can be used to exercise the processor with an external clock. If not used (the default case), it should be forced to 0;
- **INSTR_IN**. This input pin is 12-bit wide and reads instructions from the instruction memory. Each instruction requires 4 bits to specify the action to perform and 8 bits to specify the instruction operand;
- **RESET**. This input pin maintains the PEPE-8 in the reset state as long as it is active (at 0). The PEPE-8 performs an auto reset when the simulation starts, but if some external hardware is slow to initialize and needs to delay the start of the processor, this pin can be connected to a Reset module (section 6.2.10) configured with a suitable delay. This pin can also be used to reset the processor during the simulation, if needed. The normal case, however, is just use the auto reset feature and to force this pin to 1 (not active);
- **ADDRESS**. This 8-bit output pin allows the processor to specify which data memory cell to access;
- **DATA_IN**. This 8-bit input pin connects to the output data bus of the data memory, so that a memory cell can be read;
- **DATA_OUT**. This 8-bit output pin connects to the input data bus of the data memory, so that a memory cell can be written;
- **WR**. This pin distinguishes read from write data memory accesses. It will be active (at 0) during a write operation by the processor, and inactive (at 1) during read operations.

NOTE – The PEPE-8 does not support sub-word addressing, to keep it simple and because its data word is only 8-bit wide (one byte). All data and instruction addresses increase one by one.

6.5.2.2 Configuration interface

The following figure illustrates the configuration interface of the PEPE-8 module, with typical settings (but others can be set).



This interface includes the component groups described below, with the configurable parameters and information shown. Some of these settings can also be changed during simulation, by the user.

Module

It is possible to configure or to obtain information on:

- **Name** of the module;
- **Number of instruction memory address bits** (information only): 8 bits;
- **Number of bits of each instruction word** (information only): 12 bits;

- **Number of data address bits** (information only): 8 bits;
- **Number of bits of each data word** (information only): 8 bits;
- **Clock period**, in simulation time units, relevant only if the clock is configured as internal. This should be high enough to allow for the delays in various modules, such as the address decoding subsystem and memories and peripherals;
- **Clock source**, which can be configured as internal, generated internally with the above clock period, or external, in which case a clock generation circuit must be connected to the **CLOCK_IN** pin;
- **Auto start**. If checked, the processor will start executing the program as soon as the simulator gets into Simulation mode. Otherwise, the user must start the program manually;
- **Simulation master**. If checked, pausing or stopping the processor also pauses or stops the other modules, i.e., controls the simulation. Otherwise, pausing or stopping affects only this module;
- **Program file**. Clicking on the button, a file with the program to run can be specified (it will be shown in the text box). When changing to Simulation mode, that program is loaded into the instruction memory connected to the processor.

Pins

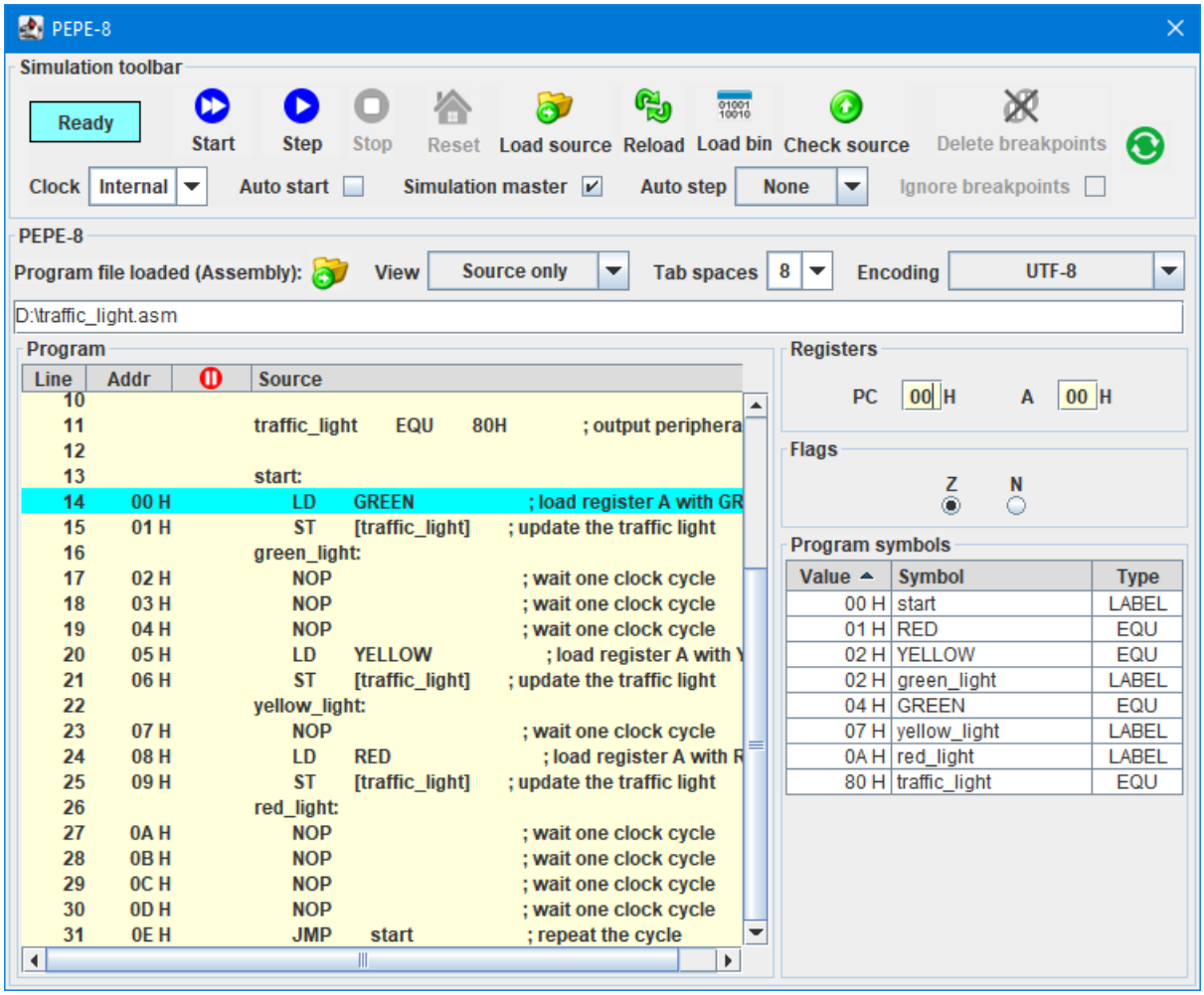
This group is identical to all modules (apart from the set of pins). See section 3.4 for details.

6.5.2.3 Simulation interface

The simulation interface of the PEPE-8 module can be opened by clicking the module in Simulation mode.

The following figure illustrates this interface, showing part of a small program in PEPE-8 assembly language that cycles a traffic light through green, yellow, and red (lights are emulated with leds). This includes two main groups of components:

- **Simulation toolbar**, with controls that act on the processor, enabling several operations;
- **PEPE-8**, with information on the program and state of the processor.



6.5.2.3.1 Simulation toolbar

This toolbar includes the following components:

- **Status**, indicating whether the processor is Not Ready, Ready, Stopped, Running, Stepping, Waiting, or Paused. Each status has its own color, also reflected in the module in the circuit area;
- **Start** executing the program;
- **Step**, to execute a single instruction. When executing the program, this button changes to **Pause** and enables pausing execution;
- **Stop** the execution of the program;
- **Reset**, a button enabled when Stopped to reset the state of the processor, changing its status to Ready;
- **Load source**, to select a file with a source program (in PEPE-8 assembly language) and load it into memory;
- **Reload**, which reloads the currently loaded program. This is useful when the program file has been edited and changed, to load the new version by just clicking this button;

- **Load bin**, to select a file with a previously compiled program (binary code) and load it into memory;
- **Check source**, to select a file with a source program (in PEPE-8 assembly language) and compile it without loading it into memory. Useful to check for compilation errors, and also to save the result as a compiled program, binary file;
- **Delete breakpoints**, to delete any breakpoint that has been set;
- **Refresh** (). A toggle button that, if selected, periodically refreshes the information below on the PEPE-8, such as the position of the blue bar, which indicates the instruction to be executed next, and the values of the registers. During program execution, this information varies too fast for the window components to keep up with. This periodic refresh enables the user to at least have an idea of what the processor is doing;
- **Clock** source, which can be internal (the default) or external. This setting is also available in the configuration interface;
- **Auto start**. If checked, the processor will start execution automatically when changing the simulator to Simulation mode. Otherwise, the user must start the program manually. This setting is also available in the configuration interface;
- **Simulation master**. If checked, pausing or stopping the processor also pauses or stops the other modules, i.e., controls the simulation. Otherwise, pausing or stopping affects only this module. This setting is also available in the configuration interface;
- **Auto step**, which performs single step operations automatically, at a specific rate that can be selected (from very slow to very fast). This is useful to execute a program at a pace much slower than continuous execution, so that the user can check in detail how the program evolves, in instruction addresses and register values, without having to click the Step button repeatedly. If the setting is not None, the Step button becomes red. Once clicked, stepping starts, pausing briefly at every instruction:

- **Ignore breakpoints**. If checked, all defined breakpoints are ignored. Execution does not stop at breakpoints, but since these are not deleted, they are available again by unchecking this check box.

6.5.2.3.2 Information on PEPE-8

This is where the most relevant information on the state of the processor is presented. It includes the following groups:

Program file and view

The text box displays the path name of the file with the source program currently loaded into memory. The open button () allows browsing to load another source program file.

The program loaded is displayed in the Program group below. There are 4 possible views:

- **Source only** (illustrated in the simulation interface above), a copy of the source file, but with the start address of each instruction;
- **Source with code**, in which for each source instruction is followed by the corresponding assembly instructions generated by the compiler, ordered by source line number;
- **Code with source**. Identical to Source with code, but ordered by instruction address instead of source line number;
- **Code only**. Only the assembly instructions are shown, without source information (namely, comments). It is a more compact view.

Regarding the source program text, shown in the Program group (except in Code only view), it is possible to specify:

- The number of blank spaces inserted for each tab in the source program text. Spacing and indentation is not identical to those of the source text (the font is not monospaced to provide a more compact view), so this setting can help in improving the source text layout;
- Encoding. Several text encodings are available to match that of the source program file, so that the source text appears correctly in the Program group.

Program

This group displays the program instructions, according to the view selected, above. Depending on that view, the following columns may appear:

- **Line**. This is the number of each source line, as it appears in the source program file. Some lines will only include comments, or just a new line;
- **Addr**. This is the start address of each instruction, in hexadecimal. Note that PEPE-8 does not support sub-word addressing and instruction addresses increase one by one. Only the rows with instructions that generate code will have an address;
- ⏸ This is a column where breakpoints can be defined (see section 7.4.1);
- **Source**. The text of each source program line;
- **Label**. Each assembly line can start with a label, so that it can be referenced in control flow instructions;
- **Mnemonic**. Each assembly instruction has a mnemonic, to indicate what it does.
- **Operands**. Almost all assembly instructions have constants as operands.

If the program is not running, a colored bar indicates the instruction that will be executed next. The color is cyan if the program is stopped or paused, and magenta if the program reached a breakpoint.

Registers

This group displays the values of the two registers of PEPE-8 (both 8-bit wide):

- **PC** (Program Counter). Contains the instruction memory address of the instruction to be executed next;
- **A** (Accumulator). The PEPE-8 has just one data register. Simple, albeit very limitative.

When the program is not running, these registers can be changed manually by the user.

Flags

The simulation interface shows two flags, which:

- **Z** (zero): this flag is 1 if the value of register A is zero, 0 otherwise;
- **N** (negative): this flag is 1 if the value of register A is negative, 0 otherwise.

NOTE – These flags are not the result of a previous operation. They just reflect the current value of register A.

Program symbols

This group displays a table with a row for each symbol defined in the program, with the following columns:

- **Value**: the value of the symbol, in hexadecimal. This is an address for labels and a numeric constant value for constants defined with an EQU directive;
- **Symbol**: the name of the symbol;
- **Type**: Label or EQU, according to whether the symbol was defined by a label or by an EQU directive.

Any of these columns can be sorted, in ascending or descending order, by clicking the header cell and then the small triangle next to the column name.

6.5.2.4 PEPE-8 instruction set (assembly language)

6.5.2.4.1 The structure of instructions

The PEPE-8 is an 8-bit processor, which means that it is able to deal with 8-bit data words in a single operation or data transfer. Its registers and buses on the data side are 8 bits wide. However, 8 bits are not enough to encode instructions with 8-bit operands. Therefore, the instruction memory is 12-bit wide and each instruction has 12 bits, in two fields:

- **Opcode** (4 highest order bits). This operation code allows to encode 16 different instructions (enough for the PEPE-8);
- **Operand** (8 lowest order bits). This is used as an address or as a data constant, depending on the instruction, defined by the opcode.

Each instruction must have at least a mnemonic, which identifies it and is related to what it does. Almost all the PEPE-8 instructions have one operand, and there is only one with no operands (in practice, ignores de Operand field).

The operand has always 8 bits and, depending on the instruction, can be interpreted at:

- **Data**, in the range [-128 .. +127];
- **Address**, in the range [0 .. +255].

6.5.2.4.2 Addressing modes

An addressing mode indicates how to obtain an operand, to be used in an instruction. The following table describes the addressing modes supported by the PEPE-8, in which the square brackets notation refers to a data memory cell contents, using the value between square brackets as its address.

Addressing mode	How to obtain the operand	Operand used as	No. of bits used in the instruction	Examples of instructions
Immediate	Constant	Data	8	LD 56H
Direct	[Constant]	Address	8	ADD [56H]
Implicit	A	Data	0	LD, ADD
	PC	Address	0	JZ, JMP

These modes can be defined in the following manner:

- **Immediate**. The operand is specified as a data constant in the instruction itself;
- **Direct**. The operand is an address constant between square brackets, with 8 bits, to cover the whole address space;
- **Implicit**. The operand is not explicitly specified in the instruction, but rather assumed by the instruction's semantics to obtain it from a predetermined source (the A or PC registers).

6.5.2.4.3 Data transfer instructions

Since there is only one general-purpose register, there are just a few data transfer instructions, illustrated by the following table.

Instruction		Addressing mode	Data transfer	Typical usage
LD	constant	Immediate	Load register A with *constant*	Initializing register A
LD	[constant]	Direct	Copy data memory at address *constant* to register A	Read memory value
ST	[constant]	Direct	Copy register A to data memory at address *constant*	Save value in register A to memory

Section 6.5.2.4.7 provides the complete set of instructions.

6.5.2.4.4 Arithmetic instructions

These instructions perform arithmetic operations on their operands. Operand and result values of arithmetic instructions are always considered in two's complement notation.

The following table provides a simplified description of the arithmetic instructions. The result is always stored in register A. Note that this register is also an operand.

Instruction		Description
ADD	constant	Adds *constant* to the value of register A
ADD	[constant]	Adds the value of the data memory at address *constant* to the value of register A
SUB	constant	Subtracts *constant* from the value of register A
SUB	[constant]	Subtracts the value of the data memory at address *constant* from the value of register A

Section 6.5.2.4.7 provides the complete set of instructions.

6.5.2.4.5 Logic instructions

These instructions implement boolean logic operations, bit by bit.

The following table provides a simplified description of the logic instructions. The result is always stored in register A. Note that this register is also an operand.

Instruction		Description
AND	constant	Bitwise AND between *constant* and the value of register A
AND	[constant]	Bitwise AND between the value of the data memory at address *constant* and the value of register A
OR	constant	Bitwise OR between *constant* and the value of register A
OR	[constant]	Bitwise OR between the value of the data memory at address *constant* and the value of register A

Section 6.5.2.4.7 provides the complete set of instructions.

6.5.2.4.6 Control flow instructions

Unlike other instructions, which perform an operation and proceed to execute the instruction at the next address, control flow instructions can determine that the next instruction to execute next is not located at the next address. These are conditional and unconditional jumps.

The main result of these instructions is to set the PC (Program Counter) register with either the next address (the normal instruction execution sequencing) or another address, specified by the instruction through a label (see section 7.3.1).

The following table provides a simplified description of the control flow instructions.

Instruction		Description
JMP	label	Jump unconditionally to *label*
JZ	label	Jump to *label* if register A is zero (Z=1)
JN	label	Jump to *label* if register A is negative (N=1)

Section 6.5.2.4.7 provides the complete set of instructions.

6.5.2.4.7 Instruction set reference table

This section provides a description of the complete instruction set of the PEPE-16 processor.

The notational conventions that were adopted are described in the following table.

Notation	Meaning
A, PC	Registers A and PC
K	Constant, with 8 bits
XXXX	4-bit field with bits that are ignored
M[*address*]	8-bit data memory cell located at *address*
dest ← *expr*	Assignment of the value of an expression (*expr*) to register A or to a memory cell (*dest*)
expr : *action*	Execute *action* if the boolean expression *expr* evaluates to true
∧, ∨	Bitwise logic operations: AND, OR

The following table describes each of the available instructions, including:

- The assembly syntax;
- The 4-bit field with the instruction opcode (bits 11..8);
- The 8-bit field with the instruction operand (bits 7..0), which some instructions interpret as data and others as an address;
- The detailed actions executed by each instruction, using the conventions in the table above.

Assembly syntax		Bits 11..8	Bits 7..0	Actions
LD	K	0000	K	A ← K
LD	[K]	0001	K	A ← M[K]
ST	[K]	0010	K	M[K] ← A
ADD	K	0011	K	A ← A + K
ADD	[K]	0100	K	A ← A + M[K]
SUB	K	0101	K	A ← A - K
SUB	[K]	0110	K	A ← A - M[K]
AND	K	0111	K	A ← A ∧ K
AND	[K]	1000	K	A ← A ∧ M[K]
OR	K	1001	K	A ← A ∨ K
OR	[K]	1010	K	A ← A ∨ M[K]
JMP	label	1011	K	PC ← endereço
JZ	label	1100	K	(A=0) : PC ← endereço
JN	label	1101	K	(A<0) : PC ← endereço
NOP		1110	XXXX XXXX	None

7 Programming and debugging

The simulator processors (PEPE-16 and PEPE-8) support programming and debugging in their respective assembly languages.

In addition, PEPE-16 also supports programming and debugging in the C language.

Sections 6.5.1 and 6.5.2 describe the PEPE-16 and PEPE-8 processor modules, respectively, including their hardware features, instruction sets, and user interfaces during simulation.

The following sections describe how to develop programs to run on these processors, using the languages they support.

7.1 PEPE-16: Programming

7.1.1 Assembly language

IMPORTANT – Assembly language source program files MUST have an "asm" extension (the file name must end with ".asm").

Section 6.5.1.8 describes the assembly language instructions. The following sections describe the assembly language instruction format and directives.

An example of a simple assembly language program is presented in section 7.1.1.3.

IMPORTANT – When loading a program the processor does not verify that enough memory is present. If, when running the program, error messages occur for no apparent reason, check that there is memory available at all the addresses used by the program for instructions, variables, and stack area. All the program views except source view (see section 6.5.1.4.2) provide an indication of the addresses used by the program (except peripherals).

7.1.1.1 Instruction format

Each line in the source program text has just one assembly instruction (there are no multi-line instructions) and has at most three parts. Any of them can be omitted, including all, but those present must appear in the following order:

- **Label**. A name followed by a colon (":"), providing a way to refer to the instruction. This name (without the colon) is used in jumps and calls, for instance;
- **Instruction**. This specifies actions to be performed. The syntax of all the available instructions is described in section 6.5.1.8.9);
- **Comment**. A semicolon (";") followed by text until the end of line. This is ignored by the compiler and provides a way to provide a textual explanation of what the instruction does and, most importantly, why. Assembly language programs are hard to understand without abundant comments.

A name is a set of alphanumeric characters and the underscore ("_"), in which the first character must be a letter.

The designations of registers and flags are reserved names and cannot be given any other use, such as labels. In addition, names are unique in the program and can only be defined once.

Some of the instructions accept a constant operand, which can be:

- A **symbolic constant**: a user defined named constant (a label or a name defined by an EQU directive – see section 7.1.1.2.1);
- A **literal constant**, a numeric value specified in one of three bases:
  - **Decimal**: a sequence of decimal digits, optionally preceded by a minus ("-") or plus ("+") sign, and optionally followed by a "d" or "D" character (to make the decimal base explicit, although it is the default base);
  - **Hexadecimal**: a sequence of hexadecimal digits, in which the first must be a decimal digit, followed by a "h" or "H";
  - **Binary**: a sequence of binary digits (bits), followed by a "b" or "B".

NOTE – There are instructions that accept literal constant operands with 4, 8, and 16 bits. When fewer hexadecimal digits or bits than necessary are specified (in hexadecimal or binary, respectively), the literal constant is extended with 0s at the left. This means that it is considered a positive value;

– Negative numbers in hexadecimal and binary are not specified with a minus ("-") sign, but rather by specifying all hexadecimal digits or bits, respectively, required by the instruction, with the higher order bit at 1. In hexadecimal constants, the first digit of a negative number must be from 8 to "F". If that first digit is a letter, a 0 must precede it, to make clear that it is a number and not a name, but this does not convert it into a positive number if all required digits are specified. As an example, 0FH in the instruction ADD R1, 0FH adds the value of -1 to R1, since ADD accepts only 4-bit constants. However, to specify -1 in a MOV instruction requires MOV R1, 0FFFFH. All 4 hexadecimal digits need to be specified, and the 0 before just avoids confusion with the name FFFFH and does not make the number positive because all hexadecimal digits are there. A constant such as 0FFFH would be positive, since after the 0 only 3 "F" are specified.

7.1.1.2 Assembly language directives

Directives, also known as pseudo-instructions, are commands that are recognized by the compiler but are not instructions, which means that they do not generate code in the compiled program. However, they may reserve space for variables and provide useful definitions and configurations.

The list of available directives is presented below, with the source program format and a description. Optional items appear inside square brackets [...] and items than can be specified zero or more times appear inside curly brackets { ... }.

NOTE – The directives STACK, PROCESS, YIELD, WAIT, and LOCK were not supported at the time the book mentioned in section 1 was written. Support for them have since been added and they can be used in this simulator.

7.1.1.2.1 EQU

Format: *name* EQU *constant*

The EQU directive defines the symbolic constant *name* and assigns it the value provided by *constant*. After this, *name* can be used in the program wherever a constant is expected or may be used.

NOTE – Name is not a label declaration. Therefore, a colon (":") must NOT be used.

– The value of constant can span the entire 16-bit range. Negative hexadecimal and binary numeric literals need to have all 16 bits specified, with the highest order bit at 1.

7.1.1.2.2 PLACE

Format: PLACE *constant*

The compiler maintains an internal address counter and increases it in each generated instruction or data definition. Each new item generated (code or data) is placed at the address provided by that counter. This sequential placement can be broken by the PLACE directive.

When it appears, the address counter is initialized with the value of *constant* and the subsequent sequential placement starts at that address. This is used to control where to place code and data blocks (instructions and data declarations, respectively).

By default (until a PLACE directive appears), placement starts at address 0000H. This is consistent with the fact that the PEPE-16 processor starts execution, after a reset, at address 0000H, where the first instruction to execute must be located.

However, a good programming practice is to declared variables (data) at the beginning of the program, and instructions (code) afterwards. Therefore, it is usual to start the program with a PLACE directive specifying some address high enough to fit all the code before it, specify data declarations, and then use a PLACE 0000H directive before finally specifying the instructions. When the program is loaded, both data and code are loaded from the respective addresses onward, and the processor has the first instruction at address 0000H, as required.

This separation and ordering of data and code areas with PLACE directives is illustrated by the assembly language program example shown in section 7.1.1.3.

7.1.1.2.3 WORD

Format: [*label*:] WORD *constant* { , *constant* }

The WORD directive allows to reserve space for one or more 16-bit words in memory, each initialized with the value specified by each 16-bit *constant*, which can be a 16-bit numeric literal or a symbolic constant, defined previously by an EQU directive. Note that negative hexadecimal and binary numeric literals need to have all 16 bits specified, with the highest order bit at 1.

Words are located contiguously in memory, starting at the address assigned to *label* by the compiler. All words will be placed at even addresses.

The WORD directive is the usual way of declaring 16-bit variables.

7.1.1.2.4 BYTE

Format: [*label*:] BYTE *constant* { *, constant* }

The BYTE directive allows to reserve space for one or more bytes, located contiguously in memory, starting at the address assigned to *label* by the compiler (which can be even or odd). The total number of bytes allocated depends on the constants specified, which can be:

- A symbolic constant, defined previously by an EQU directive, with a value that fits into 8 bits;
- 8-bit literal constant (in decimal, hexadecimal, or binary base). Its value is considered unsigned (not negative). If negative values are required, the WORD directive must be used;
- Character, with a single character enclosed by single quote marks (e.g., 'a'). Escape characters (e.g., '\t') are also supported;
- String, a sequence of characters enclosed by double quote marks (e.g., "a4$"). A string allocates as many bytes as the number of characters

Bytes are initialized by the order stemming from the declaration of constants. The BYTE directive is the usual way of declaring 8-bit variables.

7.1.1.2.5 TABLE

Format: [*label*:] TABLE *constant* { *, constant* }

The TABLE directive allocates one table for each specified *constant*, with a number of words indicated by the value of the respective *constant*.

The first table starts at the address assigned to *label* by the compiler and the ensuing tables are placed consecutively.

NOTE – The constants indicate the size in words of the tables, not values to initialize the words of the tables. These are all initialized to zero and it is up to the program to write meaningful values in them.

7.1.1.2.6 STACK

Format: [*label*:] STACK *constant* { *, constant* }

The STACK directive is similar to the TABLE directive, with the difference that it turns on a protection mechanism that generates an error if an instruction that involves the stack (e.g., CALL, RET, PUSH, POP, and interrupts) tries to use a memory address that was not allocated with a STACK directive.

The recommended practice is to use TABLE to declare data areas (to access with MOV instructions) and to use STACK to declare stack areas (to access with stack manipulating instructions).

NOTE – The PEPE-16 decreases the value of the SP register in CALL and PUSH instructions. Therefore, the SP must be initialized NOT with the value of *label* but rather with the address immediately following the declared stack area. To avoid calculations, the best practice is to specify only one constant in the STACK directive and to place a label immediately following the directive. That label can then be used to initialize the SP register.

7.1.1.2.7 PROCESS

Format: PROCESS *constant*

The PROCESS directive should be specified immediately before a routine, indicating that a CALL to that routine creates an executable process (which will execute when its turn arrives) instead of invoking that routine, as a normal CALL does.

This is a way of declaring that a routine is a process and of creating an executable instance of that process without needing a specific instruction to do it.

The *constant* is the address with which the SP register is initialized when the process instance is created and should be obtained from a STACK declaration. Note that this must be the address immediately following the stack area declaration (see the STACK directive). However, nothing prevents the process from reinitializing its SP, after being created, and in fact this must be done when several instances of the same process routine are created.

NOTE – Each CALL to a process routine generates a new process instance! This is not a normal routine call. The same routine process can have several instances (see the Processes tab in section 6.5.1.4.3 for an example);

– Each process instance has its own stack and an independent set of processor registers. This means that each process routine can be programmed as if it were the only one running in the program, eliminating restrictions stemming from sharing registers with other routines.

A process instance is generally a long executing execution thread, in a loop, and the processor executes each instance of a process until it reaches a switch point, in which the processor changes execution to the next executable instance of a process (see the YIELD, WAIT, LOCK directives).

If the process instance execution reaches the RET instruction in the routine, that instance terminates, without affecting the others.

7.1.1.2.8 YIELD

Format: YIELD

The YIELD directive is to be used before an instruction in the process code and marks a process switch point, in which the process instance can pause its execution, so that another process instance can resume its execution.

This is a way of controlling the points in which a process instance can switch to another, thereby ensuring that a process switch does not occur in an uncontrolled manner.

NOTE – A long-running loop in a process routine must have at least a switch point, otherwise it can block all other process instances, preventing them from being executed. The WAIT and LOCK directives provide alternative ways to specify a switch point.

7.1.1.2.9 WAIT

Format: WAIT

The WAIT directive is similar to the YIELD directive, allowing to specify a switch process point. However, it allows optimizing the use of the computer's CPU time by the simulator. Whenever execution reaches a WAIT directive and there are no other process instances that are executable (can be blocked, see the LOCK directive), the processor is put to sleep.

The processor wakes up automatically when a relevant event occurs in the system, such as a value change in a bit of an input peripheral or an interrupt request is made.

This mechanism is useful in processes that read peripherals in polling mode, i. e., read continuously a peripheral until the required value is found. This is the case of the keypad module, for example (see section 6.2.5). Functionally, nothing is lost, since any relevant event wakes up the processor, but until those events occur the processor is not running a loop of instructions in a uselessly manner.

7.1.1.2.10 LOCK

Format: [*label*:] LOCK *constant*

The LOCK directive is similar to the WORD directive, allocating a word in memory and initializing it with a value (only one word per directive). The difference lies on how it deals with processes:

- If an executing process instance reads a LOCK variable, it will become blocked and the processor switches to another process instance;
- When a process instance writes a LOCK variable, all the process instances that are blocked on that variable (because they tried to read it) will be unblocked, becoming executable. Their reading attempts will complete by reading the value just written on the LOCK variable.

LOCK variables are typically used for communication (using the value written) and synchronization (processes are unblocked when another writes the LOCK) between:

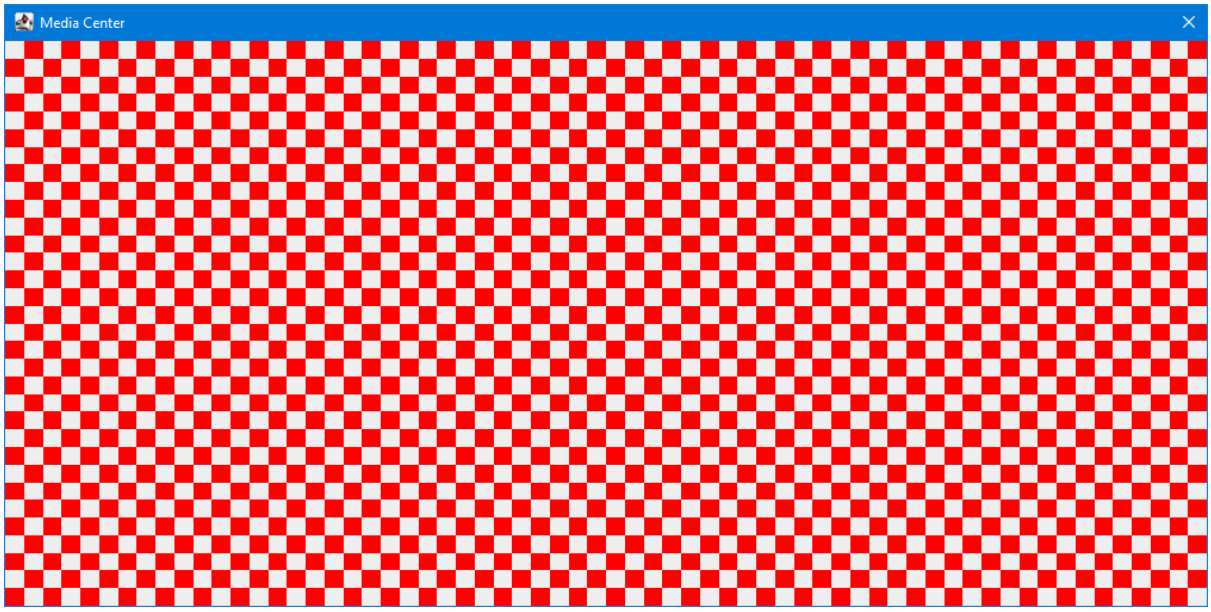
- Process instances;
- Interrupt routines and process instances. Interrupts constitute a very low level mechanism and interrupt routines should not perform application level actions. Therefore, the correct approach is for a high level process to be blocked on a LOCK variable, which an interrupt routine writes when that interrupt occurs. The process is then unblocked and performs whatever is needed to deal with the interrupt event, at the level of the application.

NOTE – Care must be taken when dealing with LOCK variables, to not fall into a process *deadlock*. If a process instance X reads LOCK A before writing LOCK B, and a process instance Y reads LOCK B before writing a LOCK A, both process instances will be blocked indefinitely!

7.1.1.3 A simple PEPE-16 assembly language program example

The following listing illustrates some of the assembly language instructions and directives. This simple program assumes that a MediaCenter module is available at address 6000H (for any other address, simply change the COMMANDS constant in the program) and produces the chess pattern shown after the program listing.

```
; File:    chess.asm
;
; Description:  This program illustrates pixel drawing with commands.
;               It draws a chess pattern

COMMANDS        EQU  6000H          ; base address of MediaCenter commands

SET_LINE        EQU COMMANDS + 0AH  ; address of the command to set the line
SET_COLUMN      EQU COMMANDS + 0CH  ; address of the command to set the column
SET_PIXEL_COLOR EQU COMMANDS + 12H  ; address of the command to draw a pixel
NO_BKG_IMAGE    EQU COMMANDS + 40H  ; address of the command to clear background
CLEAR_SCREENS   EQU COMMANDS + 02H  ; address of the command to clear all pixels

N_LINES         EQU  32             ; number of screen lines (height)
N_COLUMNS       EQU  64             ; number of screen columns (width)

RED             EQU 0FF00H          ; pixel color

     PLACE   1000H                  ; start of data area
line:           WORD 0              ; variable to save current line
column:         WORD 0              ; variable to save current column

     PLACE   0                      ; code must start at address 0000H
start:
     MOV  [NO_BKG_IMAGE], R1        ; clear the no background image banner
     MOV  [CLEAR_SCREENS], R1       ; clear all pixels (value of R1 not relevant)

     MOV  R3, 0                     ; set first pixel's color to 0
     MOV  R1, 0                     ; current line
next_line:                          ; draw pixels in current line
     MOV  [line], R1                ; save current line
     MOV  R2, 0                     ; current column

next_column:                        ; draw pixels in current column
     MOV  [column], R1              ; save current column
     MOV  [SET_LINE], R1            ; set line number
     MOV  [SET_COLUMN], R2          ; set column number
     MOV  [SET_RED], R3             ; set color of the pixel at current line and
column
     ADD  R2, 1                     ; next column
     MOV  R5, N_COLUMNS
     CMP  R2, R5                    ; last column?
     JZ   line_end
toggle_color:                       ; toggle pixel color (0 to red or vice-versa)
     CMP  R3, 0                     ; is pixel color 0?
     JZ   set_color
     MOV  R3, 0                     ; set pixel color 0 (transparent)
     JMP  next_column
set_color:
     MOV  R3, RED                   ; set pixel color red
     JMP  next_column

line_end:
     ADD  R1, 1                     ; next line
     MOV  R4, N_LINES
     CMP  R1, R4                    ; last line?
     JNZ  next_line

end:
     JMP  end                       ; stop program
```



7.1.2 C language

IMPORTANT – C language source program files MUST have a “c” extension (the file name must end with “.c”).

The following sections describe just the C language features specific to the PEPE-16 C compiler, as well as the unsupported standard C language features. Previous knowledge of C language programming is required.

An example of a simple C language program is presented in section 7.1.2.8.

IMPORTANT – When loading a program the processor does not verify that enough memory is present. If, when running the program, error messages occur for no apparent reason, check that there is memory available at all the addresses used by the program for instructions, variables, and stack area. All the program views except source view (see section 6.5.1.4.2) provide an indication of the addresses used by the program (except peripherals).

7.1.2.1 Limitations of the PEPE-16 C compiler

PEPE-16 is a simple processor and its C compiler implements only the features required to implement simple applications. It is at the level of the first C language standard (C89), with some limitations. In particular, the PEPE-16 C compiler does NOT support:

- Separate compilation. The whole program must be contained in a single “.c” file;
- Header files (“.h” files). The whole program must be contained in a single “.c” file;
- Libraries (e.g., stdio). With this C compiler, input and output must be performed by using peripherals, located at addresses that can be specified with pointers;

- Data types beyond **int**, such as **long**, **float** and **double**;
- Structures, unions, and enumerations;
- The following keywords (and the associated functionality):
  - **auto**
  - **double**
  - **enum**
  - **extern**
  - **float**
  - **long**
  - **register**
  - **struct**
  - **typedef**
  - **union**
  - **unsigned**
  - **volatile**

Nevertheless, it supports important features such as pointer arithmetic, multidimensional arrays, type casts, and forward function definitions.

In addition, it includes several features destined to deal with the PEPE-16 hardware, namely support for:

- Interrupts;
- Placement of code, data, and stack areas (to start at specific addresses);
- Processes (with various process instances, as well as Locks and Yield and Wait directives);
- Inline expansion of assembly language instructions within a C function, with the ability to access C variables from those instructions).

7.1.2.2 Data types

The PEPE-16 C compiler supports the following data types:

- **int** (16-bit, signed);
- **short** (8-bit, signed);
- **char** (8-bit, unsigned);
- **void** (for functions that do not return a value);
- multidimensional arrays;
- pointers to **int**, **short**, **char**, arrays, and functions;

Pointer arithmetic is supported.

Array and function identifiers can be used as pointers, thus enabling arrays of pointers to functions, for example (useful to invoke a function chosen at runtime from an array of pointers to functions).

The address operator (**&**) can be applied to an Lvalue (typically, a global or static variable or a function) to obtain its address, usually to initialize a pointer variable.

Variables can be declared constant (with **const**) and local variables can be declared static (with **static**).

As usual in C, pointers and the values they point to can be declared constant independently (e.g., const int* const p – the first **const** makes constant the value pointed to and the second one makes constant the pointer itself).

The **sizeof** keyword can be applied to an expression to obtain the number of bytes (individually addressable) occupied by its result data type. This is particularly useful with arrays.

The **sizeof** keyword can also be applied to a type declaration, which must be enclosed by parentheses.

7.1.2.3 Initialization of global and static variables

It is typical to initialize variables directly in the declaration (e.g., int x = 2;).

However, it is important to understand that global variables (defined outside any function) and static variables (local variables qualified as "static") generate assembly language variable declarations directly (e.g., WORD, and BYTE directives).

These directives include an initial value, which is set **only** when the program is loaded and **not** when it is restarted. This can produce unexpected results when these variables are changed by the program, the user stops the program for some reason and then restarts it. The global and static variables will not be initialized again.

Therefore, it is recommended to simply declare global and static variables without initializing them (e.g., int x;), and then perform their initialization in some function (e.g., x = 2;), since code is always executed in each time the program is run.

Global and static constants (e.g., const int x = 2,) must be initialized in the declaration, but since they cannot be changed this problem does not arise.

Arrays can also be declared and initialized in the declaration (e.g., int y[4] = {3, 5, 7, 9};), but if they are non-constant global or static variables and are changed by the program they have the same problem, so the recommendation is to declared them first and initialize them by code. Unfortunately, arrays cannot be assigned as a whole in C, therefore a loop must be used to initialize an array, one element at a time.

Local variables (declared inside some function) not marked as static are stored in the stack (to support recursion) and are initialized by code. Therefore, they can be safely initialized in the declaration.

If global or static variables (constant or not) are initialized in their declaration, the initializer can be an expression, but it must have a constant value, computable at compile-time.

7.1.2.4 Accessing specific hardware addresses

The C compiler takes care of placement of variables and code. However, access to specific hardware addresses is needed to access input and output peripherals, since these are located at addresses defined by the address decoding hardware of the system with the PEPE-16.

Accessing a user-defined address (either a memory cell or a peripheral port) is done by setting a pointer with the required address value, which can be done in several ways, described in the following sections.

7.1.2.4.1 Cast of an address value to a pointer

This is the simplest way. Just convert a value (with the address to access) to a pointer, which enables to use pointer access in C. However, note that the access can be in 16 or 8 bits, for which int* or char* pointers are used, respectively, as shown in the following example.

```
const int addr = 0x6000;                // the address to access
int* const pw = (int *) addr;           // pw now points to the 16-bit (word) cell or port to access
char* const pb = (char *) (addr + 1);   // pb now points to the 8-bit (byte) cell or port to access
                                        // what is constant is the pointer, not the cell or port
int x;                                  // 16 bit variable
char y;                                 // 8 bit variable
void main(){
        *pw = 3;                        // write the 16-bit cell or port at address 0x6000
        x = *pw;                        // read the 16-bit cell or port at address 0x6000
        *pb = 3;                        // write the 8-bit cell or port at address 0x6001
        y = *pb;                        // read the 8-bit cell or port at address 0x6001
}
```

NOTE – Pointers to **int** must have an even value, a restriction imposed by the PEPE-16 addressing. Pointers to **char** do not have this restriction, since they refer to bytes (8 bits), which PEPE-16 can address individually.

7.1.2.4.2 Pointer arithmetic

When accessing a range of cells or peripheral ports (e.g., commands for the MediaCenter module, see section 6.3.3.5), it is convenient to define a base address (from an **int** with a cast) and then add a value to it, acting like an index in PEPE-16's indexed addressing mode (see section 6.5.1.8.2). This is illustrated by the following example.

```
const int base = 0x6000;                // the base of the address range to access
int* const pw = (int *) base;           // pw now points to the first 16-bit (word) cell or port
char* const pb = (char *) base;         // pb now points to the first 8-bit (byte) cell or port
                                        // what is constant is the pointer, not the cell or port
const int i = 5;                        // index
int x;                                  // 16 bit variable
char y;                                 // 8 bit variable
void main(){
        *(pw + i) = 3;                  // write the 16-bit cell or port at address 0x600A
        x = *(pw + i);                  // read the 16-bit cell or port at address 0x600A
        *(pb + i) = 3;                  // write the 8-bit cell or port at address 0x6005
        y = *(pb + i);                  // read the 8-bit cell or port at address 0x6005
}
```

NOTE – Pointer arithmetic takes the size of the pointed type into account. Since **int**s occupy 2 bytes, adding 5 to pw corresponds in fact to adding 10. Adding 5 to pb adds only 5, since each **char** occupies just 1 byte.

7.1.2.4.3 Pointer to array

When accessing a range of cells or peripheral ports (e.g., commands for the MediaCenter module, see section 6.3.3.5), an alternative to pointer arithmetic (the previous case) is to consider those cells or ports as an array and then define a pointer to that array (initialized from an **int** with a cast). This is illustrated by the following example.

```
const int base = 0x6000;                    // the base of the address range to access
int (* const pw)[] = (int (*)[]) base;      // pw is a pointer to an array of ints, initialized via a cast
                                            // pw has the address of the first 16-bit (word) cell or port
char (* const pb)[] = (char (*)[]) base;    // pb is a pointer to an array of chars, initialized via a cast
                                            // pb has the address of the first 8-bit (byte) cell or port
const int i = 5;                            // index
int x;                                      // 16 bit variable
char y;                                     // 8 bit variable
void main(){
        (*pw)[i] = 3;                       // write the 16-bit cell or port at address 0x600A
        x = (*pw)[i];                       // read the 16-bit cell or port at address 0x600A
        (*pb)[i] = 3;                       // write the 8-bit cell or port at address 0x6005
        y = (*pb)[i];                       // read the 8-bit cell or port at address 0x6005
}
```

This example illustrates the equivalence between array indexing and pointer arithmetic. Again, the number of bytes occupied by each element of the array is taken into account.

NOTE – The commands of the MediaCenter are all 16-bit, so they require the **int** version.

7.1.2.5 Interrupts

To enable a given PEPE-16 interrupt (in the range 0 .. 3), a function must be declared as being the interrupt handler, i.e, the function to invoke when a given interrupt occurs. This is done in C with a pragma interrupt directive, preceding the function declaration and specifying the number of the interrupt (0 .. 3) it is associated with, as illustrated by the following example.

```
#pragma INTERRUPT  0
void int_0(){
        // process interrupt 0
}
. . .       // declaration of functions for interrupts 1 and 2

#pragma INTERRUPT  3
void int_3(){
        // process interrupt 3
}
```

Up to 4 functions can be associated with interrupts, using this directive. Only those required need to be specified and can appear in any order. The compiler automatically generates the enable interrupt instructions and the exception table.

The names of the functions can be any valid identifier. The previous ones are just an example.

If no interrupt functions are specified, no interrupts are enabled.

7.1.2.6 Support for processes

Support in the C compiler for the PEPE-16 processes is provided with additional pragma directives.

7.1.2.6.1 PROCESS

To declare a process, a function must be declared with a pragma process immediately preceding its declaration, using the following syntax:

```
#pragma PROCESS  instances  @  stack size
void name (arguments){
        while (1){
                // process statements
        }
}
```

The following remarks are in order:

- Functions marked as a process must return void;
- Invoking the function name with its arguments, if any, creates a process that becomes runnable, instead of actually executing its code immediately;
- The same process function can be invoked several times, each time creating a new process instance, with the function code but with a separate stack area;
- *instances* is to be replaced by a constant expression that specifies the maximum number of allowed process instances, i.e., the number of times the function can be invoked. This is required in order to reserve stack space for all the process instances. If *instances* is omitted, the default value of 1 is assumed. If specified and is greater than 1, then it is mandatory that the function has at least one argument and the first argument must specify the number of the instance (0 .. *instances*-1) when the function is invoked. The program will misbehave if the value of the first argument is outside this range when the function is invoked;
- *stack size* is to be replaced by a constant expression that specifies the number of 16-bit words to be reserved for each process instance. It will be multiplied by the number of allowed instances to reserve enough space for all the instances of this process. If *stack size* (and the "@") is omitted, the default value of stack size is assumed. This default size can be changed with the STACK_SIZE directive (section 7.1.2.7.2);

- The process function body typically includes an infinite loop, since a process normally implements some long-running activity. If the process function reaches its end or executes a return statement, the process instance terminates.

NOTE

– Although the “main” function cannot be preceded by the PROCESS directive, it actually is the main body of the main process, one that exists even if no other processes are created. This can be seen in the simulation window of the PEPE-16 processor (Processes tab of the additional panel – see section 6.5.1.4.3);

– This main process has only one instance and its stack size is the default process stack size (section 7.1.2.7.2);

– In addition, the YIELD and WAIT directives can be used inside the “main” function and it can use LOCK variables as any other process;

– If the “main” function returns, the main process blocks itself by reading a LOCK that is never written by any process. The processes that it has created will continue, until they terminate or get blocked indefinitely in a deadlock (waiting for a lock that no other process writes onto).

7.1.2.6.2 YIELD and WAIT

These directives specify points of the process code in which the scheduler can switch to another process and generate a YIELD and WAIT directives, respectively, in assembly language (sections 7.1.1.2.8 and 7.1.1.2.9, respectively). This allows processes to give others the opportunity to run when their turn arises.

They can only be specified inside a process function, preceding the statement at which the process switch can be made. Their syntax is very simple, as in the following example:

```
#pragma PROCESS
void name (arguments){
        while (1){
                #pragma YIELD           // or #pragma WAIT (may switch to another instance here)
                // process statements
        }
}
```

7.1.2.6.3 LOCK

The LOCK directive allows to specify variables that can be used for communication and synchronization between process instances. A process instance reading a lock variable is automatically blocked until some other instance writes it with some value. When this occurs, all process instances blocked at that lock are unblocked (becoming runnable) and read the value written into the lock. In terms of assembly language, this C directive generates a LOCK declaration in assembly language (instead of WORD). See section 7.1.1.2.10 for details.

The LOCK directive must immediately precede the declaration of a variable, as shown in the following example:

```
#pragma LOCK int x;
```

LOCK can only be applied to variables of the type int, pointer, or array of int or pointer.

LOCK variables must be global (declared outside any function).

IMPORTANT – Beware of multiple tests to a LOCK variable! This is typical of chained **if**s. Each test of the variable's value implies reading it and the process gets blocked each time the variable is read, which is not what is intended in a chained if sequence. The correct procedure is to read the LOCK variable only once, placing its value in an auxiliary, non-LOCK variable, and then performing multiple tests on this auxiliary variable. This way, the process only gets blocked once.

7.1.2.7 Additional directives

Additionally, the C compiler supports the directives described below.

7.1.2.7.1 STACK_INIT

Usually, the C compiler takes care of placing the areas reserved for the various processes (or just the main program, if no processes are declared). However, this can constitute a substantial amount of memory, and the user might want to place stack areas at a different memory module than the program and its variables.

This directive, with the syntax

#pragma STACK_INIT *address*;

allows to specify the *address* where the overall stack area must start (must be even).

The usual place to locate this directive is at the beginning of the program.

7.1.2.7.2 STACK_SIZE

By default, the main program and each process instance (if any) is allocated 0x100 words of stack area (0x200 bytes). This may be too much or too little, depending on the program and on the memory available.

This directive, with the syntax

#pragma STACK_SIZE *default_stack_size_in_words*;

allows to specify the *default stack size in words* allocated by default to the main program to each process instance (if no other value is specified in the process declaration – see section 7.1.2.6.1).

The usual place to locate this directive is at the beginning of the program.

7.1.2.7.3 PLACE

Usually, the C compiler takes care of placing the assembly-level variables and instructions generated from the variable declarations and statements in C. However, the user may want to have a better control of the ranges of addresses at which variables and/or code should be placed.

This directive, with the syntax

#pragma PLACE *address*;

allows to specify the address at which the next generated items (variables or code) should start to be placed (must be even).

This directive can be used wherever the sequential and increasing order of addresses generated by the compiler is to be broken and to start at another value. It corresponds to the assembly level directive PLACE (section 7.1.1.2.2).

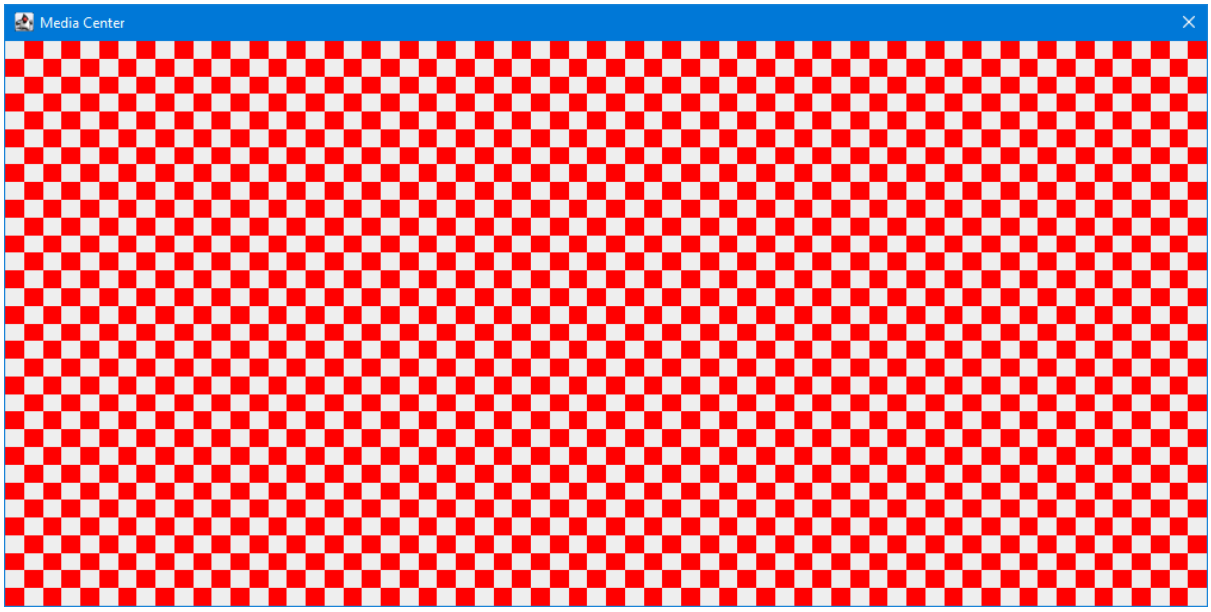
7.1.2.8 A simple PEPE-16 C language program example

The following listing illustrates C language programming in the PEPE-16 processor and is equivalent to the program in section 7.1.1.3. This simple program assumes that a MediaCenter module is available at address 0x6000 (for any other address, simply change the BASE constant in the program) and produces the chess pattern shown after the program listing.

NOTE – The execution of this program takes more than twice the time needed by the assembly program of section 7.1.1.3, although the result is the same. This is the price of programming in a high-level language. Assembly programming is lower level and harder, but is much more efficient.

```
/*
File:    chess.c

Description:    This program illustrates pixel drawing with commands.
                It draws a chess pattern
*/

const int BASE = 0x6000;        // base address of the commands for the MediaCenter
int (* const commands)[] = (int(*)[])BASE;     // pointer to an array of commands

const int SET_LINE         = 5;       // number of the command to set the line
const int SET_COLUMN       = 6;       // number of the command to set the column
const int SET_PIXEL_COLOR  = 9;       // number of the command to draw a pixel
const int NO_BKG_IMAGE     = 0x20;    // number of the command to clear background
const int CLEAR_SCREENS    = 1;       // number of the command to clear all pixels

const int N_LINES          = 32;      // number of screen lines (height)
const int N_COLUMNS        = 64;      // number of screen columns (width)

const int RED = 0xFF00;               // pixel color

int line = 0;
int column = 0;
int color = 0;

void main() {
     (*commands)[NO_BKG_IMAGE] = 0;       // clear the no background image banner
     (*commands)[CLEAR_SCREENS] = 0;      // clear all pixels

     for (line = 0; line < N_LINES; line++) {
          for (column = 0; column < N_COLUMNS; column++) {
               (*commands)[SET_LINE] = line;              // set line
               (*commands)[SET_COLUMN] = column;          // set column
```

```
                (*commands)[SET_PIXEL_COLOR] = color;      // set color of the pixel
                color = color == 0 ? RED : 0;         // toggle color at every pixel
            }
            color = color == 0 ? RED : 0;         // toggle color of first pixel in
                                                  // each new line
        }
}
```



7.1.2.9 *Inspecting the assembly code generated by the C compiler*

The PEPE-16 C compiler analyzes the C source file, verifies its correctness (syntax and semantics), and, if no errors are detected, generates PEPE-16 assembly language instructions and directives, in order to implement the functionality specified by the C source statements and directives. The assembly language compiler (assembler) will then be invoked to generate machine code, which the PEPE-16 actually executes.

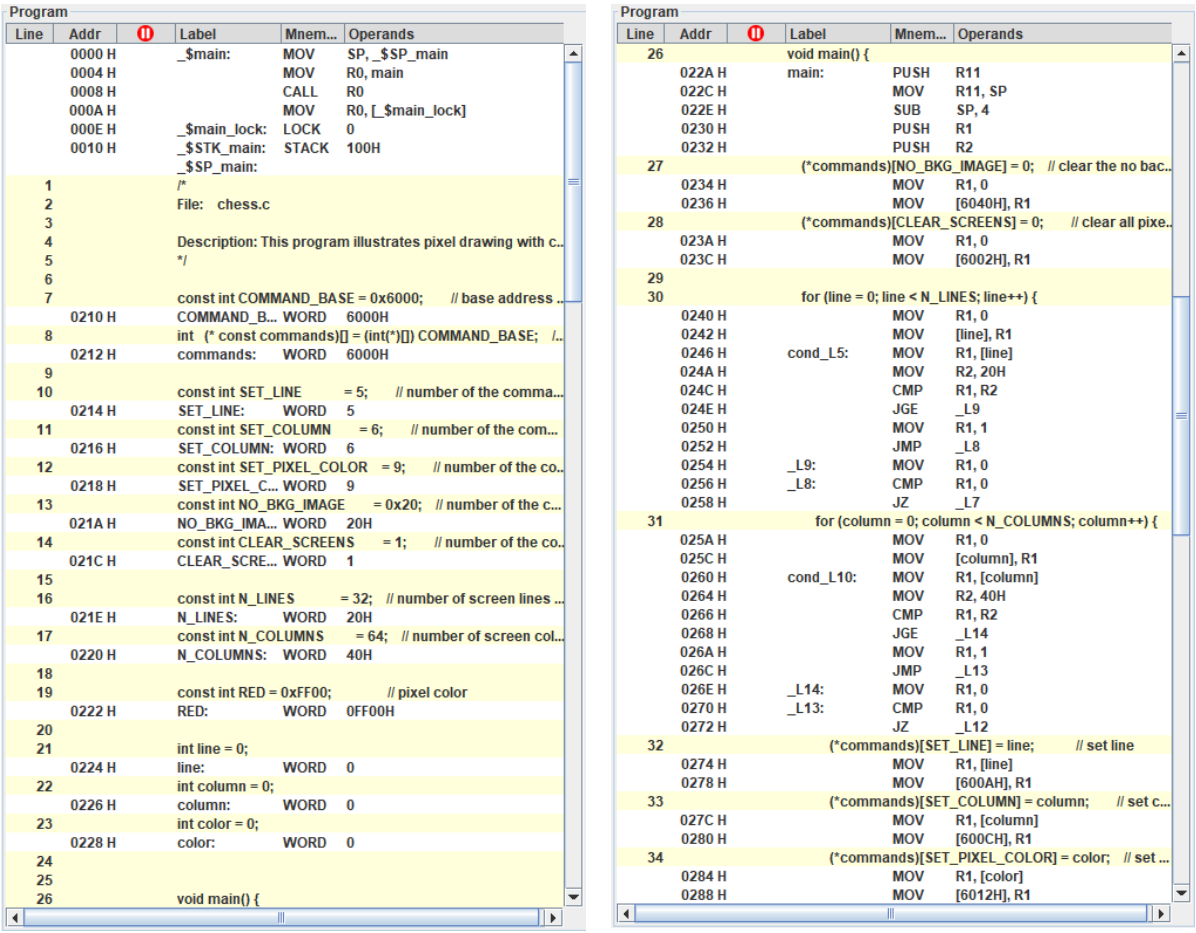
It is useful to be able to inspect the assembly instructions generated by the compiler. This not only is instructive, to have an idea of how a compiler works, but also enables to check what is actually generated at which addresses, which may prove useful to detect run-time errors.

In particular, it is possible to debug a C program not only at the source level but also at the assembly instruction level, including checking the values of registers (see section 7.2.2).

The following figure shows the program chess.c of the previous section in the simulation interface of the PEPE-16, using the source with code view, in which each source line is followed by the assembly code that it generates. For brevity, only the variable declarations and part of the statements of the main function are shown, in two columns (obtained from the PEPE-16 interface by scrolling).

Note the initial code automatically inserted by the compiler, before the first source line. It initializes the stack pointer and calls the main function (using a register in the CALL to avoid the range limitations of the CALL label). The main program ends by reading a Lock variable, so that it suspends execution (putting PEPE-16 into a Waiting state) rather than using active waiting by an endless loop. There is also the declaration of the stack area.

The rest shows declaration of variables and the assembly instructions generated by each C statement, including the addresses at each variable and instruction.



```
Program
Line  Addr     Label        Mnem...  Operands
      0000 H   _$main:      MOV      SP, _$SP_main
      0004 H                MOV      R0, main
      0008 H                CALL     R0
      000A H                MOV      R0, [_$main_lock]
      000E H   _$main_lock: LOCK     0
      0010 H   _$STK_main:  STACK    100H
               _$SP_main:
 1                          /*
 2                          File:   chess.c
 3
 4                          Description: This program illustrates pixel drawing with c...
 5                          */
 6
 7                          const int COMMAND_BASE = 0x6000;      // base address ...
      0210 H   COMMAND_B... WORD     6000H
 8                          int  (* const commands)[] = (int(*)[]) COMMAND_BASE;   /...
      0212 H   commands:    WORD     6000H
 9
10                          const int SET_LINE        = 5;      // number of the comma...
      0214 H   SET_LINE:    WORD     5
11                          const int SET_COLUMN      = 6;      // number of the com...
      0216 H   SET_COLUMN:  WORD     6
12                          const int SET_PIXEL_COLOR = 9;      // number of the co...
      0218 H   SET_PIXEL_C... WORD   9
13                          const int NO_BKG_IMAGE    = 0x20;   // number of the c...
      021A H   NO_BKG_IMA... WORD    20H
14                          const int CLEAR_SCREENS   = 1;      // number of the co...
      021C H   CLEAR_SCRE... WORD    1
15
16                          const int N_LINES         = 32;   // number of screen lines ...
      021E H   N_LINES:     WORD     20H
17                          const int N_COLUMNS       = 64;   // number of screen col...
      0220 H   N_COLUMNS:   WORD     40H
18
19                          const int RED = 0xFF00;           // pixel color
      0222 H   RED:         WORD     0FF00H
20
21                          int line = 0;
      0224 H   line:        WORD     0
22                          int column = 0;
      0226 H   column:      WORD     0
23                          int color = 0;
      0228 H   color:       WORD     0
24
25
26                          void main() {
```

```
Program
Line  Addr     Label        Mnem...  Operands
26                          void main() {
      022A H   main:        PUSH     R11
      022C H                MOV      R11, SP
      022E H                SUB      SP, 4
      0230 H                PUSH     R1
      0232 H                PUSH     R2
27                          (*commands)[NO_BKG_IMAGE] = 0;   // clear the no bac...
      0234 H                MOV      R1, 0
      0236 H                MOV      [6040H], R1
28                          (*commands)[CLEAR_SCREENS] = 0;      // clear all pixe...
      023A H                MOV      R1, 0
      023C H                MOV      [6002H], R1
29
30                          for (line = 0; line < N_LINES; line++) {
      0240 H                MOV      R1, 0
      0242 H                MOV      [line], R1
      0246 H   cond_L5:     MOV      R1, [line]
      024A H                MOV      R2, 20H
      024C H                CMP      R1, R2
      024E H                JGE      _L9
      0250 H                MOV      R1, 1
      0252 H                JMP      _L8
      0254 H   _L9:         MOV      R1, 0
      0256 H   _L8:         CMP      R1, 0
      0258 H                JZ       _L7
31                              for (column = 0; column < N_COLUMNS; column++) {
      025A H                MOV      R1, 0
      025C H                MOV      [column], R1
      0260 H   cond_L10:    MOV      R1, [column]
      0264 H                MOV      R2, 40H
      0266 H                CMP      R1, R2
      0268 H                JGE      _L14
      026A H                MOV      R1, 1
      026C H                JMP      _L13
      026E H   _L14:        MOV      R1, 0
      0270 H   _L13:        CMP      R1, 0
      0272 H                JZ       _L12
32                                  (*commands)[SET_LINE] = line;        // set line
      0274 H                MOV      R1, [line]
      0278 H                MOV      [600AH], R1
33                                  (*commands)[SET_COLUMN] = column;        // set c...
      027C H                MOV      R1, [column]
      0280 H                MOV      [600CH], R1
34                                  (*commands)[SET_PIXEL_COLOR] = color;   // set ...
      0284 H                MOV      R1, [color]
      0288 H                MOV      [6012H], R1
```

7.1.2.10 Using assembly language in the C source program

The PEPE-16 C compiler offers the possibility of including PEPE-16 assembly instructions embedded in the C source file. These instructions will be inserted at the point in which they appear and will be treated as if they were generated by the C compiler.

The block of assembly instructions can appear wherever a C statement is allowed and the syntax is simple:

asm {
    assembly instructions
}

The assembly instructions follow the normal syntax of assembly instructions, including comments.

asm statements suffer from some caveats and restrictions that need to be taken into account:

- Comments consist of ";" until the end of line, not "//";
- Hexadecimal constants use the assembly notation (e.g., 1000H, not 0x1000);

- EQU is the only directive allowed (all others are forbidden in **asm** statements);
- Use of registers R11, SP, RE, BTE, and TEMP is not allowed;
- Jumping to a label outside the **asm** statement generates an error. Jumping through a register is not checked;
- Calling a C function is allowed (even if outside the **asm** statement), but parameters are not taken care of (the compiler passes function arguments by the stack);
- Some specific semantic errors such as a constant out of range (e.g., ADD R1, 25) are detected only after generating the assembly instructions for the entire program, which means that the reported line of the error will not match its line number in the C source file.

It is possible to access the C variables from inside the **asm** statement, with the following rules:

- Only identifiers can be used (no expressions);
- An identifier preceded by "$" accesses its content (both read and write). Includes all variables (global, static, local, and arguments);
- An identifier preceded by "&" accesses its address (read only). Includes all variables (global, static, local, and arguments) and functions;
- The types of the identifiers must be **int** or pointer (array and function identifiers are considered pointers with their initial address). Types **char** and **short** are not allowed;
- Only MOV instructions can be used to access identifiers in this way;
- Labels corresponding to identifiers of global or static variables, or functions, can be used as usual by assembly instructions. In that case, accessing variable content (equivalent to "$") requires square parentheses, as illustrated below.

The following example illustrates some of these features:

```
int x = 5;
int * p = (int *) 0x2000;
int w[5];

void f(){
}

void main(){
     int y = 3;                    ; local variable, allocated in the stack

     asm{
          addr EQU 1000H
               MOV  R1, addr
               MOV  R3, $p         ; equivalent to MOV R3, [p]
               MOV  $p, R1         ; equivalent to MOV [p], R1
               CALL f              ; could also use CALL &f
               MOV  R2, $x         ; equivalent to MOV R2, [x]
               MOV  R2, &x         ; equivalent to MOV R2, x
               MOV  R2, x          ; same as previous
               MOV  R2, &w         ; equivalent to MOV R2, w
               MOV  R2, &y         ; not equivalent to MOV R2, y
                                   ; y is local and resides in the stack
                                   ; generates code to get its address
     }
}
```

In all, using assembly instructions within a C function should be reserved for very low-level, special cases, and not as a general solution. In most cases, C alone is enough to build a complete program.

7.2 PEPE-16: Debugging

7.2.1 Debugging assembly programs

7.2.1.1 Debugging mechanisms

When developing a program, most likely it will not work in the first attempts, and the programmer needs to understand what the program is doing and how to correct errors. This is known as program debugging.

There are three main mechanisms to debug a program:

- **Refresh** the simulation interface periodically, by clicking the refresh button () in the simulation toolbar of the PEPE-16 (section 6.5.1.4.1). This is a very soft mechanism, since it is periodic and not continuous, but allows having an idea about which zones of instructions the program is going through (the blue bar indicates the instruction that will be executed next). It is particularly useful when the program is blocked somewhere;
- **Step** execution, with single step or auto step. The most controllable way of executing instructions while debugging, but also the slowest. Useful to inspect localized program segments;
- **Breakpoints**, executing the program at full speed but pausing at previously set points (the breakpoints). Useful to run programs up to some point, from which it can be executed in step mode.

7.2.1.2 Stepping programs

The following figure illustrates a program being stepped. The blue bar indicates which instruction will be executed next. Clicking the **Step** button executes that instruction and the blue bar changes to the next instruction to execute. This works even in jump and call instructions.

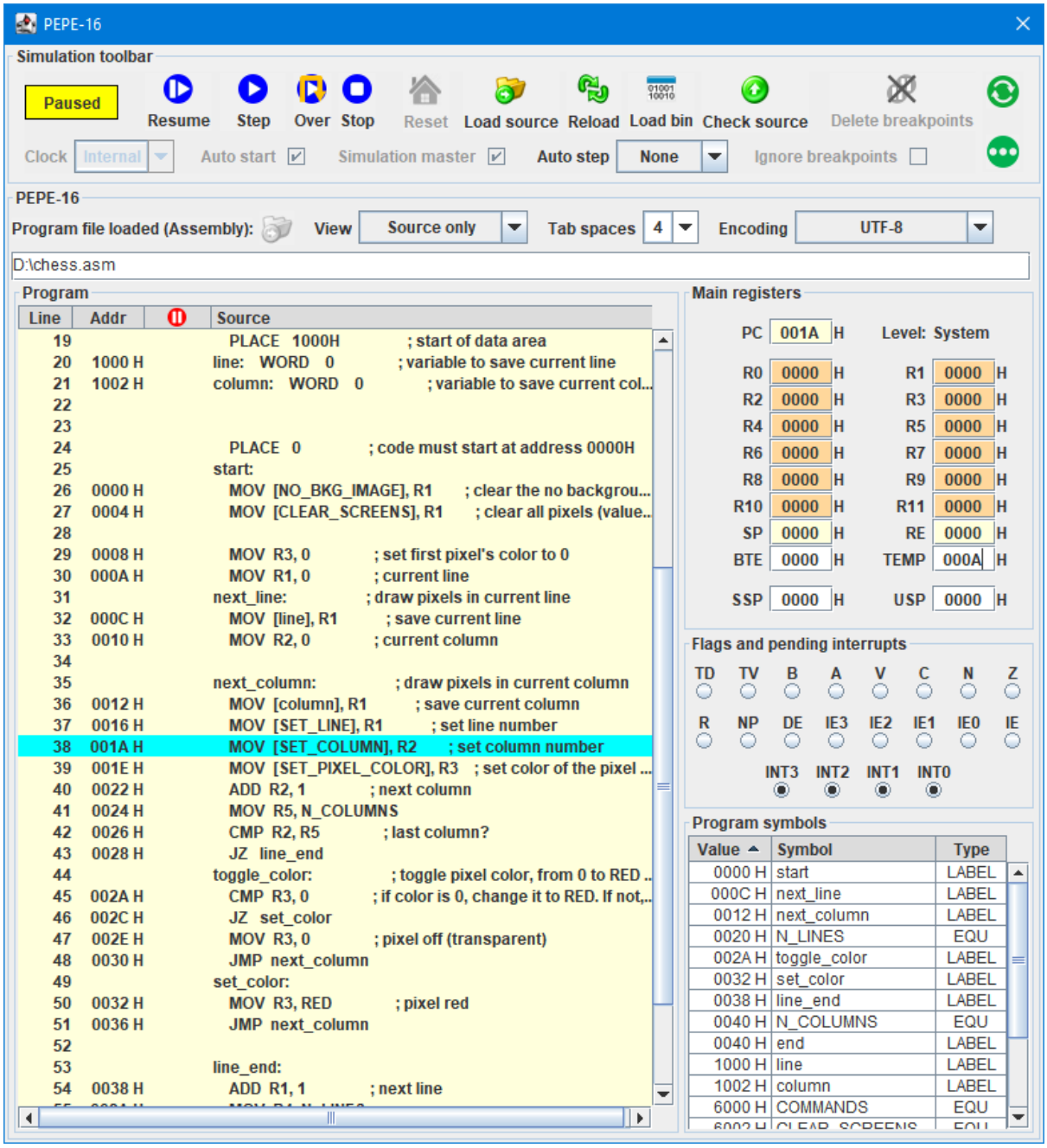
The program status in this mode is Paused, briefly passing by Stepping when the **Step** button is clicked. If needed, the continuous execution can be resumed by clicking the **Resume** button.

When the program is executing (status Running), execution can be paused by clicking the **Pause** button. After that, execution can proceed by stepping or by resuming execution.

The **Over** button also steps instructions, but CALL instructions are executed as if they were just one instruction. The routine is invoked, its instructions are executed at full speed (including eventual routines that it may invoke) and returns, pausing at the instruction after the CALL. This is useful to avoid entering lower level routines and distracting the programmer from the flow of control of interest. For all other instructions, **Over** is identical to **Step**.

Therefore, upon reaching a CALL, the programmer can decide whether to step into the routine (**Step**) or over it (**Over**).

At any point, each time execution pauses after a step, the registers and memory can be inspected.



NOTE – If the PEPE-16 is a simulation master (selectable in the simulation toolbar), pausing it causes the suspension of other modules with simulation status, if they exist. A step resumes them briefly, but when the stepped instruction finishes they are suspended again.

Stepping many instructions can become laborious and tiresome. The auto step feature can ease this problem. In the simulation toolbar, a speed of step mode execution can be selected in the Auto step combo box (from Very slow to Very fast). In this case, the Step and Over buttons become red:

Clicking one of these buttons will start stepping autonomously, at the selected pace, stepping over CALLs if Over was clicked. At any time, execution can be paused.

7.2.1.3 Using breakpoints

A breakpoint is an indication that execution must be paused if the processor tries to execute an instruction, or to read or write a variable, located at an address at which a breakpoint has been set.

There are two types of breakpoints:

- **Code breakpoints**. Execution pauses when the processor reaches the instruction with the breakpoint (before executing it). These can only be set in instructions, not in directives;
- **Data breakpoints**. Execution pauses when any MOV instruction tries to access the variable with the breakpoint (before accessing it). Data breakpoints can only be set in BYTE, WORD, and LOCK directives. These can be setup to distinguish the type of access: read, write, or both.

A breakpoint can be set by clicking, in the Program group, the source line in which the instruction or variable appears. Clicking successively, the breakpoint indication circulates among the various possibilities. In the case of instruction breakpoints, this is just set or not set. In the case of data breakpoints, there are more possibilities.

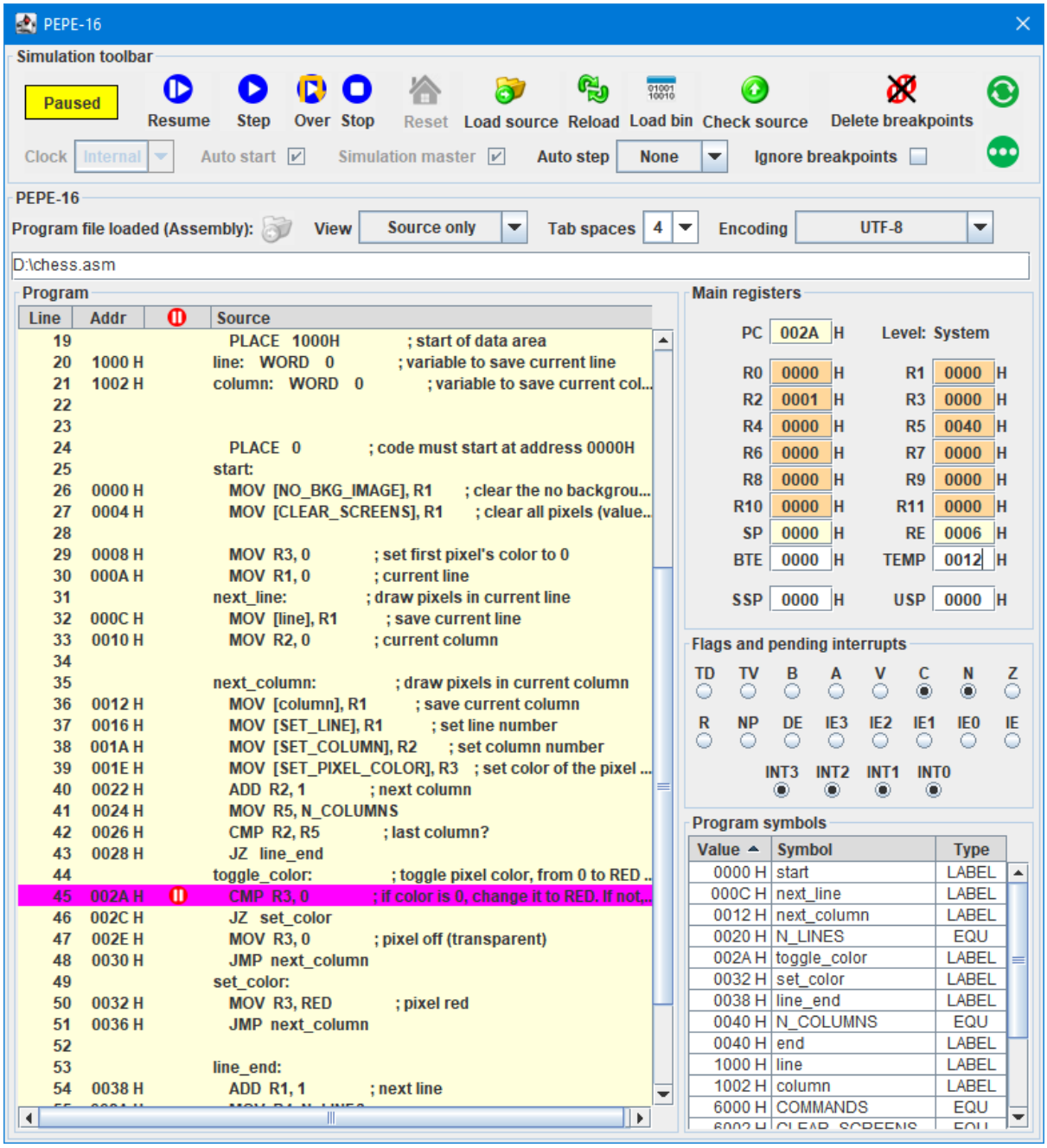
The breakpoint indication, if set, appears in the breakpoint column (). The following table shows the various possible types of breakpoint indication.

Breakpoint indication	Breakpoint type
	Instruction breakpoint
R	Data read breakpoint
W	Data write breakpoint
RW	Data read or write breakpoint

In addition:

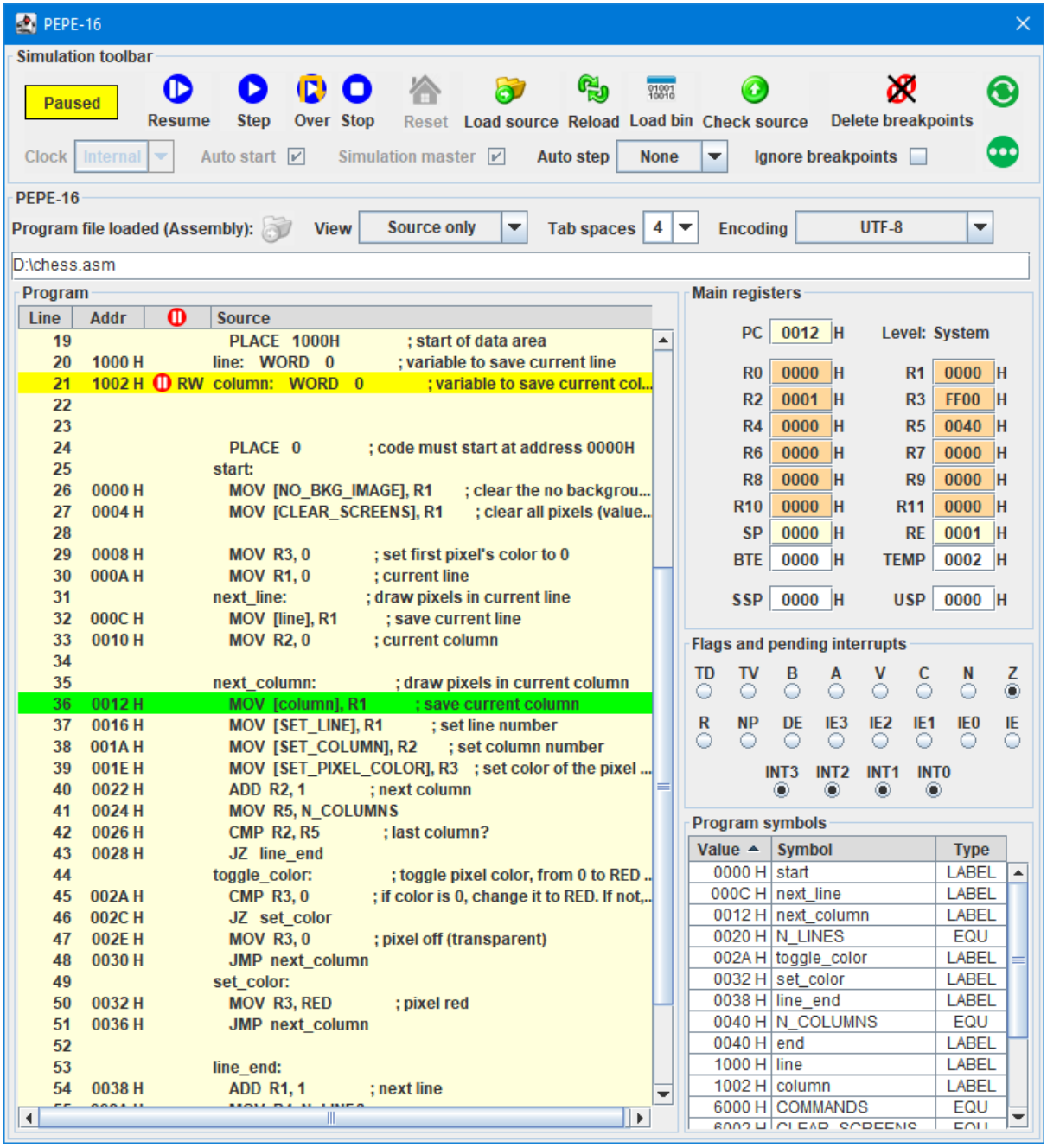
- Right clicking the instruction or variable declaration, a popup menu appears with further options;
- Any of the breakpoints can be individually disabled, in which case it appears grayed out. This enables ignoring (not pausing on) a specific breakpoint without deleting it;
- The simulation toolbar includes a checkbox which allows to ignore all breakpoints and a button to delete all breakpoints that have been set.

When the execution pauses at an instruction breakpoint, the respective line appears with a magenta color, instead of blue. This is illustrated in the following figure.



When the execution pauses at a data breakpoint, two lines appear colored: the line of the instruction that tried the access appears green, and the line of the data accessed appears yellow.

This is illustrated by the following figure, in which a data write breakpoint was set in variable **column** and execution was paused at the instruction that attempted to write that variable.



NOTE – When stepping, processing will not stop at breakpoints. This is more relevant when stepping over. Even if breakpoints are defined and active in the code executed inside a CALL, execution will only stop at the instruction following the CALL.

7.2.2 Debugging C programs

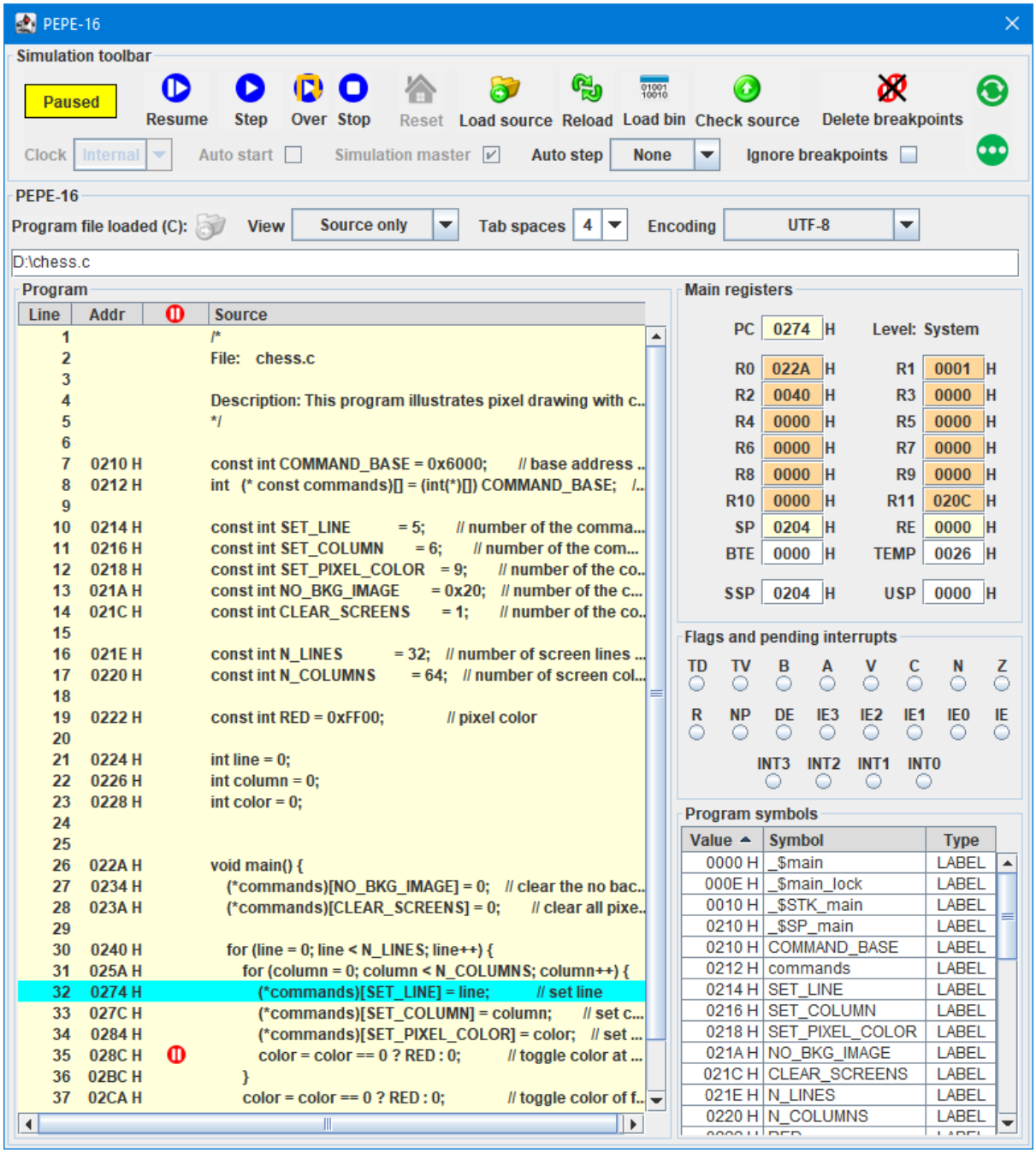
Debugging C programs is almost identical to debugging assembly language programs, as described in section 7.2.1.

However, the point to keep in mind is that each C source statement typically corresponds to several assembly level instructions. This means that it is possible to debug a C program at the source level (source view) or at the assembly language level (all other views). Program views are described in section 6.5.1.4.2).

In source only view:

- Only the C source statements are visible;
- When the program is loaded but not executed (being ready to start), the blue bar appears in the heading of the “main” function, since execution starts there;
- Each step executes one C statement, including all the assembly instructions that it generates;
- An **asm** statement counts as a single C statement, even if many assembly language instructions are visible inside it;
- A step over executes a complete invocation of a function, including execution of its complete body, even if it invokes other functions;
- Breakpoints can be set at the start of all C statements in which execution can pause, including function headings (execution will pause after invocation but before executing the function body);
- There is no support to inspect the value of variables in this view, but data breakpoints can be set in variables.

The following figure illustrates debugging of the chess program of section 7.1.2.8 in source only view, with the blue bar indicating the C source statement to be executed next, in source line 32. A breakpoint can also be seen, set in the source line 35.



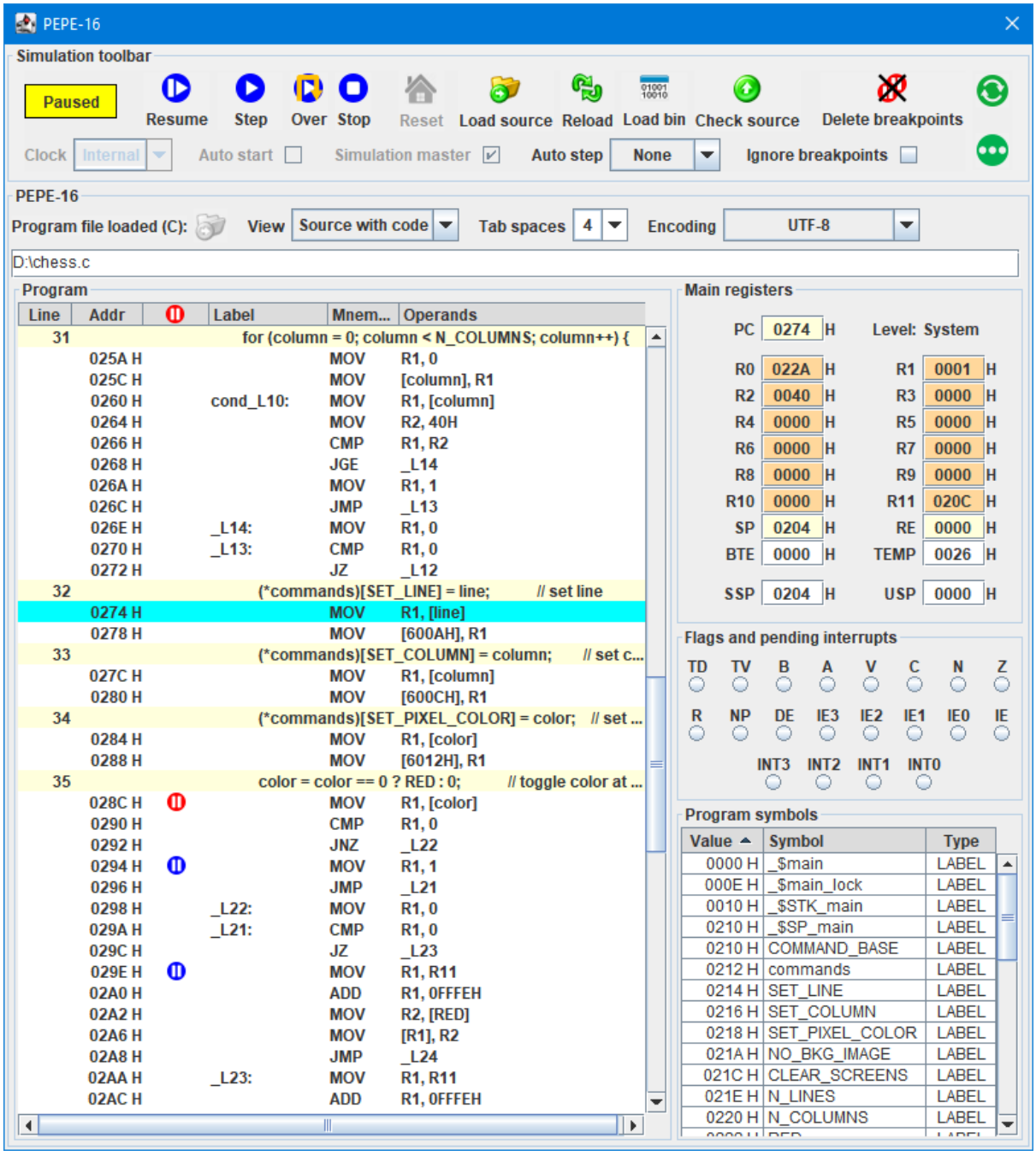
In all other views, the assembly level instructions will be shown. The most used is the source with code view, in which assembly level instructions appear after the C source statement that generated them, ordered by C source line number.

In views that expose assembly level instructions:

- Both C source statements and assembly instructions are visible (except in code only view);
- When the program is loaded but not executed (being ready to start), the blue bar appears in address 0000H, where the processor starts execution. The “main” function is invoked after some initializations, which can be stepped;
- A breakpoint can no longer be set in a C source statement, but rather on the first assembly level instruction that it generates. This is just more detail than a breakpoint in source view;

- However, now it is possible to set breakpoints in subsequent assembly instructions of the same C source statement, providing more execution debugging control;
- These intermediate breakpoints appear in blue (⏸) instead of red (⏸), precisely to make explicit that these do not correspond to the beginning of a C source statement.

The following figure illustrates debugging of the same chess program (section 7.1.2.8), now in source with code view, with C source statements and the assembly instructions they generate. The blue bar still indicates the C source statement to be executed next, in source line 32. The breakpoint of source line 35 can still be seen (in the first assembly instruction generated by the C source line 35), but two other breakpoints have been set (shown in blue) inside the set of assembly instructions generated by the C source line 35. This allows debugging in greater detail (if needed).



7.3 PEPE-8: Programming

The PEPE-8 processor supports only its assembly language.

IMPORTANT – Assembly language program files MUST have an “asm” extension (the file name must end with “.asm”).

Section 6.5.1.8 describes the assembly language instructions. The following sections describe the assembly language instruction format and directives.

7.3.1 Instruction format

Each line in the source program text has just one assembly instruction (there are no multi-line instructions) and has at most three parts. Any of them can be omitted, including all, but those present must appear in the following order:

- **Label**. A name followed by a colon (“:”), providing a way to refer to the instruction. This name (without the colon) is used in jump instructions;
- **Instruction**. This specifies actions to be performed. The syntax of all the available instructions is described in section 6.5.2.4.7);
- **Comment**. A semicolon (“;”) followed by text until the end of line. This is ignored by the compiler and provides a way to provide a textual explanation of what the instruction does and why. Assembly language programs are hard to understand without abundant comments.

A name is a set of alphanumeric characters and the underscore (“_”), in which the first character must be a letter. Names are unique in the program and can only be defined once.

Most of the instructions accept a constant operand, which can be:

- A **symbolic constant**: a user defined named constant (a label or a name defined by an EQU directive – see section 7.3.2.1);
- A **literal constant**, a numeric value specified in one of three bases:
  - **Decimal**: a sequence of decimal digits, optionally preceded by a minus (“-“) or plus (“+”) sign, and optionally followed by a “d” or “D” character (to make the decimal base explicit, although it is the default base);
  - **Hexadecimal**: a sequence of hexadecimal digits, in which the first must be a decimal digit, followed by a “h” or “H”;
  - **Binary**: a sequence of binary digits (bits), followed by a “b” or “B”.

NOTE – The literal constant operands are expected to have 8 bits. When fewer hexadecimal digits or bits than necessary are specified (in hexadecimal or binary, respectively), the literal constant is extended with 0s at the left. This means that it is considered a positive value;

– Negative numbers in hexadecimal and binary are not specified with a minus (“-“) sign, but rather by specifying all hexadecimal digits or bits, respectively, required by the instruction, with the higher order bit at 1. In hexadecimal constants, the first digit of a negative number must be from 8 to “F”. If that first digit is a letter, a 0 must precede it, to

make clear that it is a number and not a name, but this does not convert it into a positive number if all required digits are specified. As an example, 0FH in the instruction ADD 0FH adds the value +15 to register A. However, to specify -1 as the value to add, the constant 0FFH must be used. Both hexadecimal digits need to be specified, and the 0 before just avoids confusion with the name FFH and does not make the number positive because all hexadecimal digits are there.

7.3.2 Assembly language directives

Directives, also known as pseudo-instructions, are commands that are recognized by the compiler but are not instructions, which means that generate no code in the compiled program. However, they may provide useful definitions and configurations.

The PEPE-8 assembly language includes only two directives: EQU and WAIT.

7.3.2.1 EQU

Format: *name* EQU *constant*

The EQU directive defines the symbolic constant *name* and assigns it the value provided by *constant*. After this, *name* can be used in the program wherever a constant is expected and may be used.

NOTE – Name is not a label declaration. Therefore, a colon (":") must NOT be used.

– The value of constant can span the entire 8-bit range. Negative hexadecimal and binary numeric literals need to have all 8 bits specified, with the highest order bit at 1.

7.3.2.2 WAIT

Format: WAIT

The WAIT directive is to be used before an instruction and marks a point in which the program can pause its execution. It allows to optimize the use of the computer's CPU time by the simulator. Whenever execution reaches a WAIT directive, the PEPE-8 processor is put to sleep.

The processor wakes up automatically when a relevant event occurs in the system, such as a value change in a bit of an input peripheral.

This mechanism is useful in programs that read peripherals in polling mode, i. e., read continuously a peripheral until the required value is found. This is the case of the keypad module, for example (see section 6.2.5). Functionally, nothing is lost, since any relevant event wakes up the processor, but until those events occur the processor is not running a loop of instructions in a uselessly manner.

7.3.3 A simple PEPE-8 assembly language program example

The following listing illustrates some of the assembly language instructions and directives. This simple program implements a fixed cycle light traffic controller.

Timings are implemented with NOP instructions, assuming an external clock source with a 1 Hz frequency clock. This means that each instruction takes 1 second to execute.

```
; File:    traffic_light.asm
;
; Description:  This program implements a traffic light controller

RED             EQU     01H             ; RED light (led connects to bit 0)
YELLOW          EQU     02H             ; YELLOW (led connects to bit 1)
GREEN  EQU      04H                     ; GREEN (led connects to bit 2)

traffic_light   EQU     80H             ; output peripheral address

start:
        LD      GREEN                   ; load register A with GREEN
        ST      [traffic_light]         ; update the traffic light
green_light:
        NOP                             ; wait one clock cycle
        NOP                             ; wait one clock cycle
        NOP                             ; wait one clock cycle
        LD      YELLOW                  ; load register A with YELLOW
        ST      [traffic_light]         ; update the traffic light
yellow_light:
        NOP                             ; wait one clock cycle
        LD      RED                     ; load register A with RED
        ST      [traffic_light]         ; update the traffic light
red_light:
        NOP                             ; wait one clock cycle
        NOP                             ; wait one clock cycle
        NOP                             ; wait one clock cycle
        NOP                             ; wait one clock cycle
        JMP     start                   ; repeat the cycle
```

7.4 PEPE-8: Debugging

7.4.1 Debugging mechanisms

When developing a program, most likely it will not work in the first attempts, and the programmer needs to understand what the program is doing and how to correct errors. This is known as program debugging.

There are three main mechanisms to debug a program:

- **Refresh** the simulation interface periodically, by clicking the refresh button () in the simulation toolbar of the PEPE-8 (section 6.5.2.3.1). This is a very soft mechanism, since it is periodic and not continuous, but allows having an idea about which zones of instructions the program is going through (the blue bar indicates the instruction that will be executed next). It is particularly useful when the program is blocked somewhere;

- **Step** execution, with single step or auto step. The most controllable way of executing instructions, but also the slowest. Useful to inspect localized program segments;
- **Breakpoints**, executing the program at full speed but pausing at previously set points. Useful to run programs up to some point, from which in can be executed in step mode.

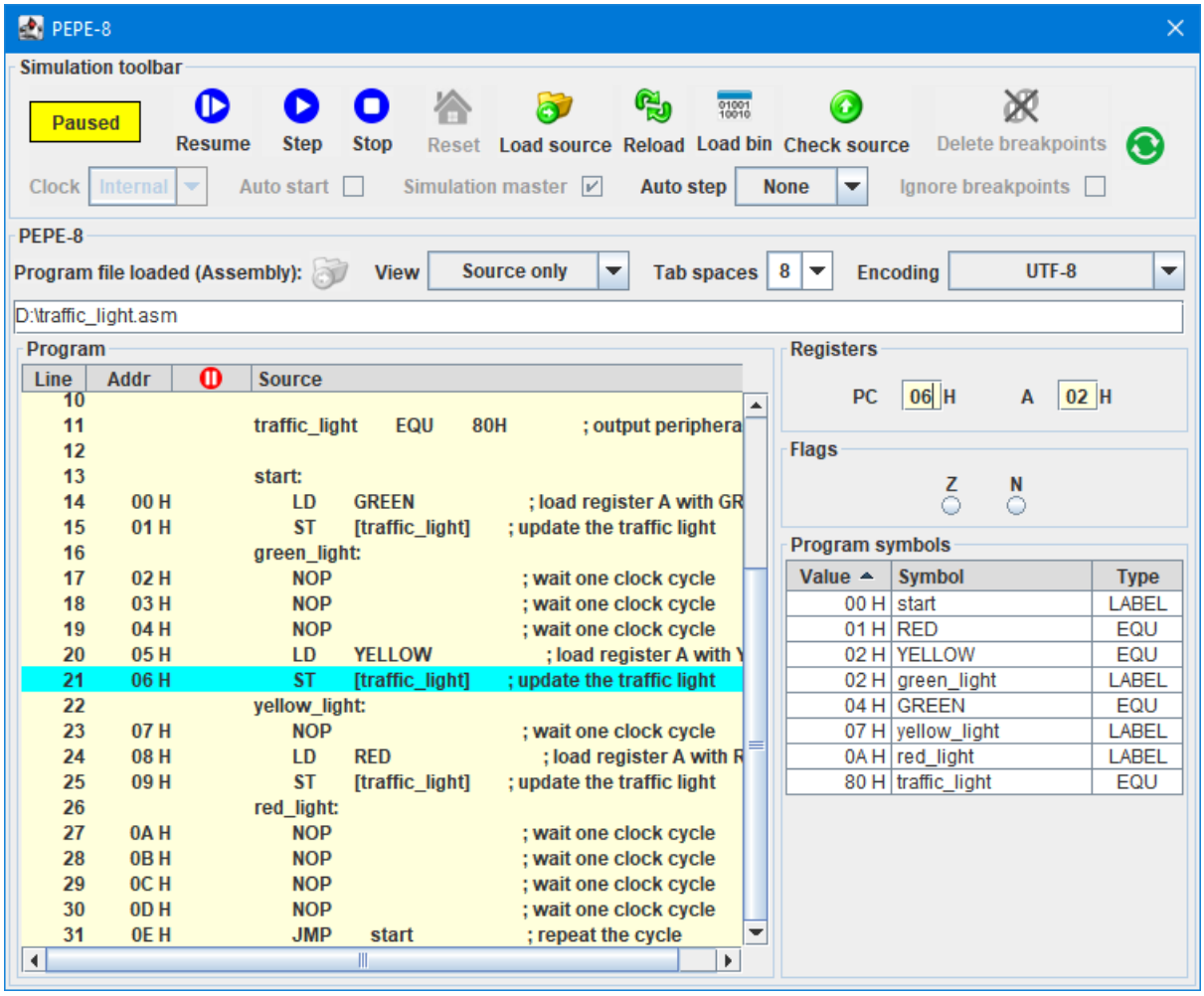
7.4.2 Stepping programs

The following figure illustrates a program being stepped. The blue bar indicates which instruction will be executed next. Clicking the **Step** button executes that instruction and the blue bar changes to the next instruction to execute. This works even in jump and call instructions.

The program status in this mode is Paused, briefly passing by Stepping when the **Step** button is clicked. If needed, the continuous execution can be resumed by clicking the **Resume** button.

When the program is executing (status Running), execution can be paused by clicking the **Pause** button. After that, execution can proceed by stepping or by resuming execution.

At any point, each time execution pauses after a step, the registers and memory can be inspected.



NOTE – If the PEPE-8 is a simulation master (selectable in the simulation toolbar), pausing it causes the suspension of other modules with simulation status, if they exist. A step resumes them briefly, but when the stepped instruction finished they are suspended again.

Stepping many instructions can become laborious and tiresome. The auto step feature can ease this problem. In the simulation toolbar, a speed of step mode execution can be selected in the Auto step combo box (from Very slow to Very fast). In this case, the Step button becomes red:

Clicking this button will start stepping autonomously, at the selected pace. At any time, execution can be paused.

7.4.3 Using breakpoints

A breakpoint is an indication that execution must be paused if the processor tries to execute an instruction located at a specific address. Execution pauses when the processor reaches the instruction with the breakpoint (before executing it). Breakpoints can only be set in instructions, not in directives.

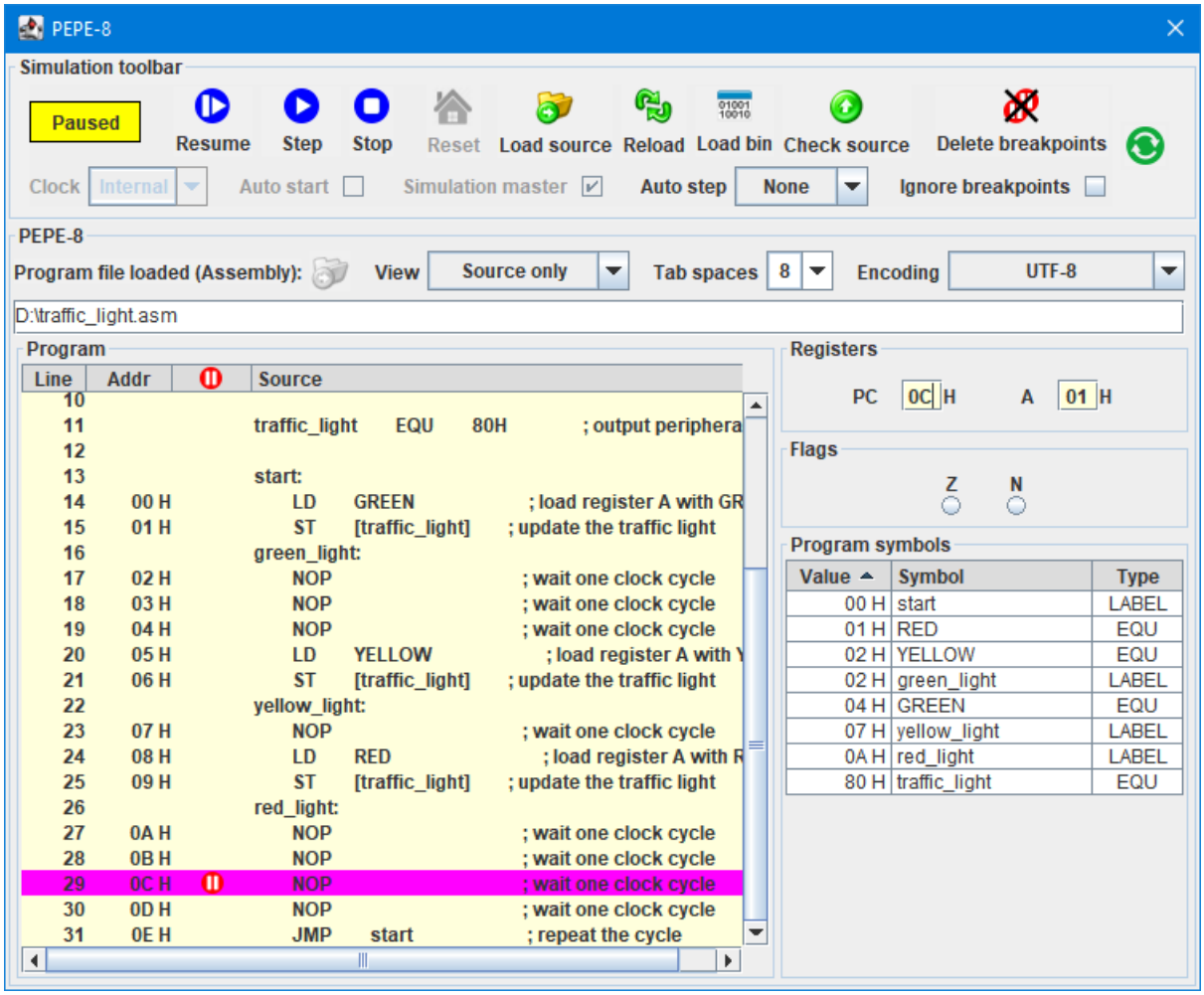
A breakpoint can be set by clicking, in the Program group, the source line in which the instruction appears. Clicking successively, the breakpoint indication circulates between set and not set.

The breakpoint indication, if set (), appears in the breakpoint column.

In addition:

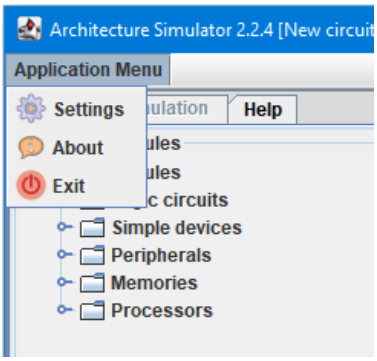
- Right clicking the instruction or data, a popup menu appears with further options;
- Any of the breakpoints can be individually disabled, in which case it appears grayed out. This enables ignoring (not pausing on) a specific breakpoint without deleting it;
- The simulation toolbar includes a checkbox which allows to ignore all breakpoints and a button to delete all breakpoints that have been set.

When the execution pauses at an instruction breakpoint, the respective line appears with a magenta color, instead of blue. This is illustrated in the following figure.


PEPE-8
Simulation toolbar
Paused
Resume
Step
Stop
Reset
Load source
Reload
Load bin
Check source
Delete breakpoints
Clock
Internal
Auto start
Simulation master
Auto step
None
Ignore breakpoints
PEPE-8
Program file loaded (Assembly):
View
Source only
Tab spaces
8
Encoding
UTF-8
D:\traffic_light.asm
Program
Line Addr Source
10
11 traffic_light EQU 80H ; output peripher
12
13 start:
14 00 H LD GREEN ; load register A with GR
15 01 H ST [traffic_light] ; update the traffic light
16 green_light:
17 02 H NOP ; wait one clock cycle
18 03 H NOP ; wait one clock cycle
19 04 H NOP ; wait one clock cycle
20 05 H LD YELLOW ; load register A with Y
21 06 H ST [traffic_light] ; update the traffic light
22 yellow_light:
23 07 H NOP ; wait one clock cycle
24 08 H LD RED ; load register A with R
25 09 H ST [traffic_light] ; update the traffic light
26 red_light:
27 0A H NOP ; wait one clock cycle
28 0B H NOP ; wait one clock cycle
29 0C H NOP ; wait one clock cycle
30 0D H NOP ; wait one clock cycle
31 0E H JMP start ; repeat the cycle
Registers
PC 0C H
A 01 H
Flags
Z
N
Program symbols
Value Symbol Type
00 H start LABEL
01 H RED EQU
02 H YELLOW EQU
02 H green_light LABEL
04 H GREEN EQU
07 H yellow_light LABEL
0A H red_light LABEL
80 H traffic_light EQU

8 Configuring the simulator

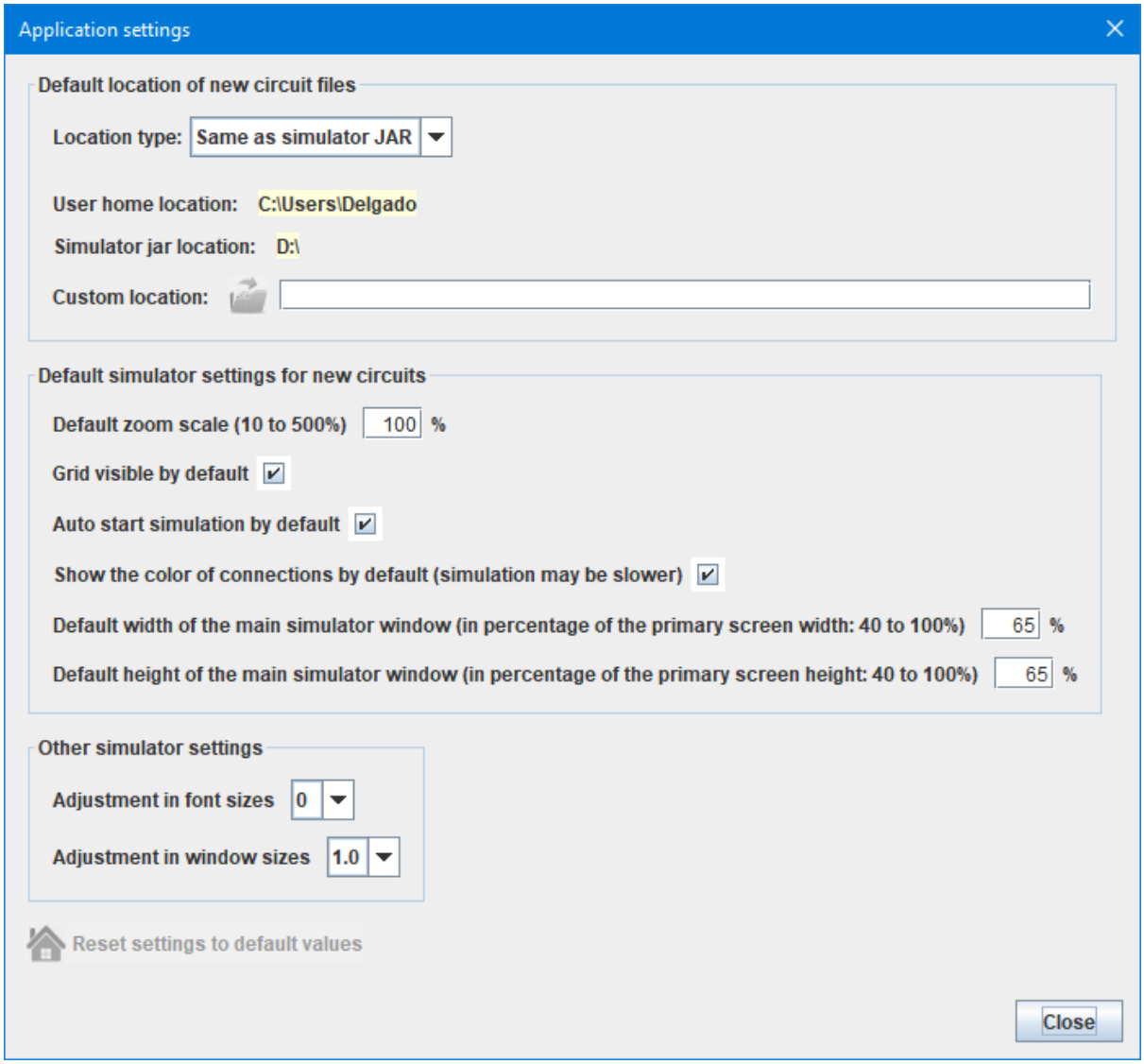
The simulator application has an additional menu that can be invoked by clicking the “Application Menu”, at the top left corner of the application window, as illustrated by the following figure.



It includes three buttons:

- **Settings**. This allows configuring several aspects of the simulator, described below;
- **About**. This opens up a window with the following information:
  - Version of the simulator (also displayed in the window’s title) and the date and time on which the jar has been produced;
  - Version of the Java runtime environment (JRE) required to run the simulator (or higher);
  - Version of the Java compiler used to compile the simulator Java archive;
  - Version of the JavaFX library used to build the simulator Java archive (for multimedia features);
  - List of contributors to versions 2 and 1 of the simulator;
- **Exit**. Terminates the simulator application. This can also be done by closing the simulator application’s main window.

Clicking the Settings button (enabled on Design mode only), a simulator configuration interface window appears, as illustrated by the following figure.



This window allows to configure:

- **The default location** for new circuit files;
- **The default simulator settings** for new circuits, when the New button is clicked in Design mode;
- **Other simulator settings**, which include increasing the font size and window size of the various simulator windows. This enables adjusting the size of the interface to the resolution of the computer used, preventing windows from being too small in high resolutions and vice-versa. It also enables adjusting manually the fonts and sizes well beyond what would be required for computer screen viewing. In particular, increasing the font size is very helpful to a professor projecting a computer screen on an overhead projector.

There is also a button to reset all these settings to factory default values.