



PROXIMA

A Virtual Space Agency

MANUAL

Logic Blocks



BASIC OPERATIONS

Here's a list with the basic operations you'll need in order to work on your missions.

Add block to diagram

Drag block from palette into the diagram.

Select block

Click on the block you want to select.

Cancel Selection

Use ESC to cancel the current selection.

Delete block

Click on the block you want to delete, then click BACKSPACE.

The selected block and all of its connections will be deleted.

Connect Pins

Click on one of the pins then drag the connection and click on the pin you want to connect to.

Input pins can be connected to only one output pin; output may be connected to multiple inputs.

Delete Connection

Click on the pin that has the connection you want to delete.

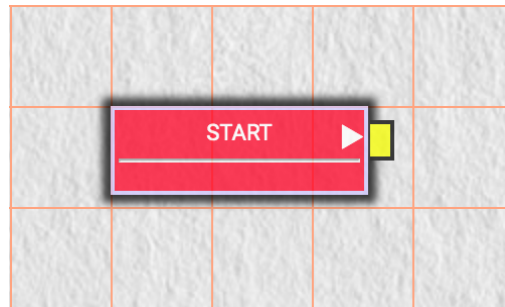
Warning: all wires connected to that pin will be deleted.



BASIC BLOCKS

START

Start operation.



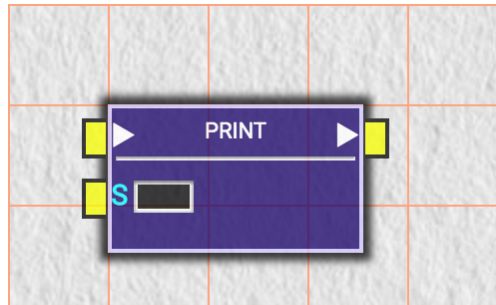
This block is activated when you start the simulation.

It is executed only once, and its only function is to trigger the first block in your control diagram.



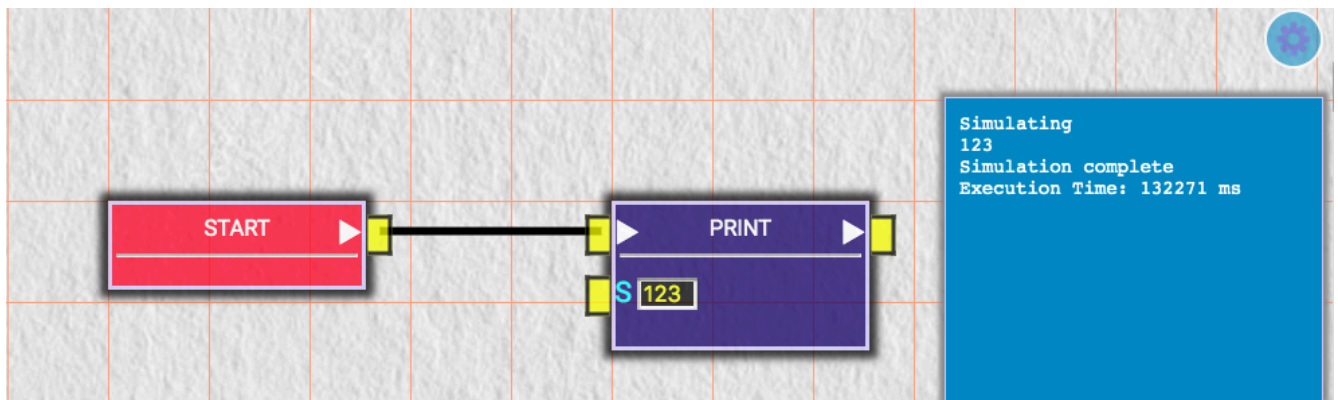
PRINT

Print a value on the debug area.



The value of input S is printed.

In the example below, we print number 123 when simulation begins.



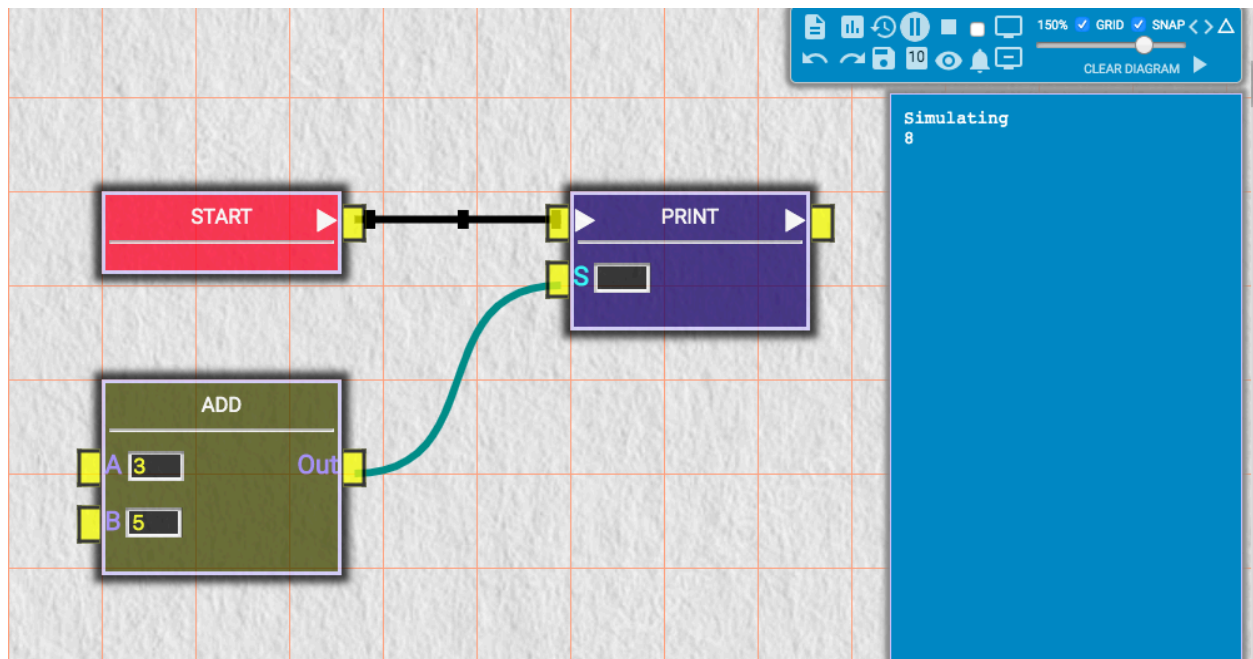


ARITHMETIC BLOCKS

ADD

Adds two numbers and outputs the result.

In the example below, we add 3 to 5 and print the result.



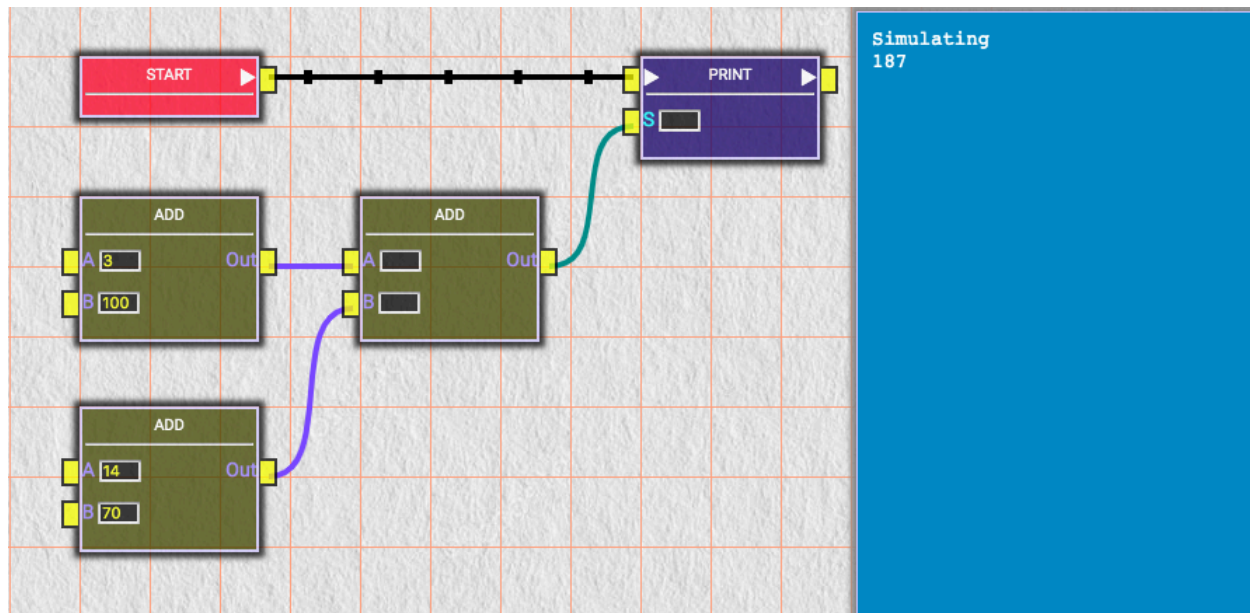


We can also combine multiple blocks.

In the example below, we are adding 4 numbers, by combining 2 ADD blocks.

The first block adds 3 and 100, the second block adds 14 and 70.

Then we use a third block to add the outputs of the first and second blocks, to get the final result, 187.





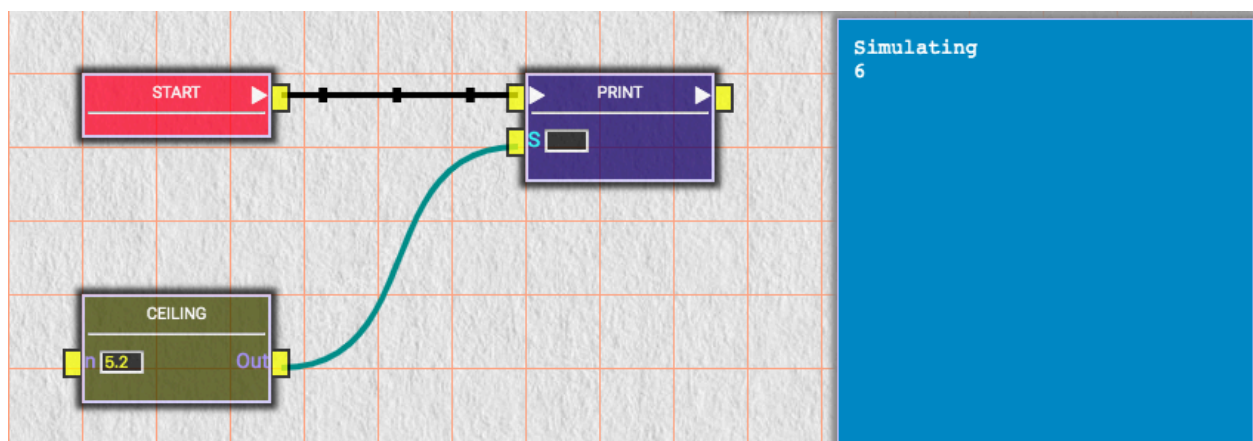
CEILING

This logic block outputs the mathematical ceiling of the value of its input “n”.

In mathematics, the ceiling function is used to round a value up.

If the input is an integer number, that is, if it doesn't have a fractional part, in other words, if it doesn't have significant digits after the decimal point, then its ceiling value is itself.

If the input value is not an integer value, then this block returns the next integer number up.



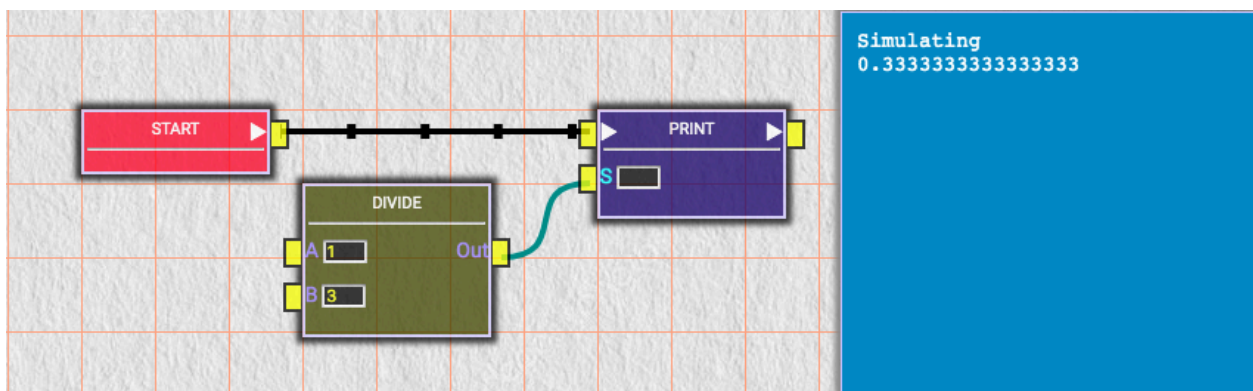


DIVIDE

This logic block divides the number in its “A” input by the number in its “B” input, sending the result to its output.

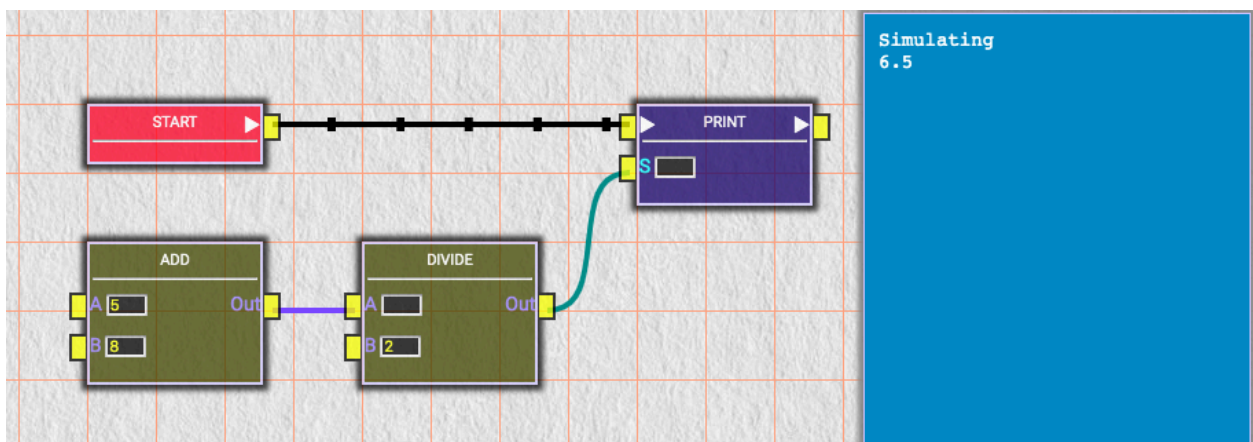
In the example, we’re dividing 1 by 3, and printing the result, 0.333...

Obviously, the order of the parameters is important: this block always divides A by B - never B by A.



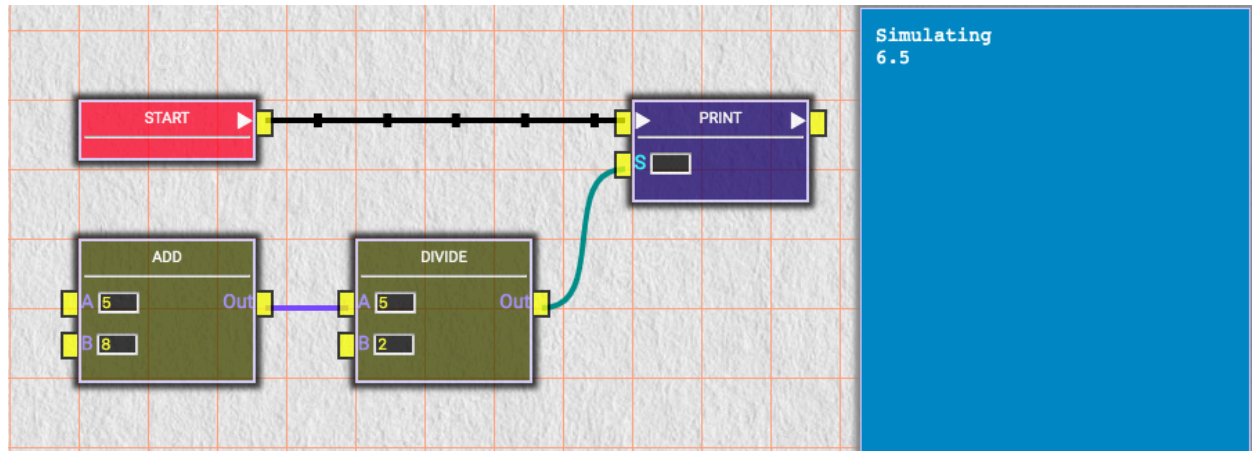
We can also combine different blocks, to build complex expressions.

In the following example, we are computing the result of $(5 + 8) / 2$.





One important thing to know is that when an input has a value typed in, but it is also connected to the output of another logic block, the value coming from the other block has priority over the literal value - the value typed in is ignored and the incoming value is used, instead (see example below).

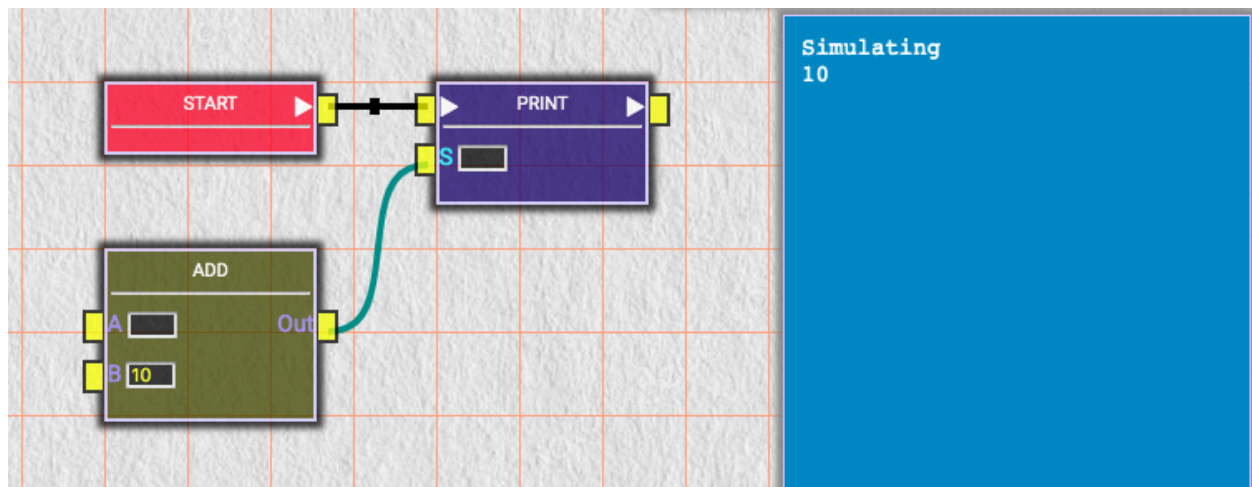


As you can see in the image above, the “5” in the A input of the DIVIDE block is ignored.

If 5 had been used, the PRINT block would print 2.5, that is, $5 / 2$.

Instead, the DIVIDE block used the 13, output of the ADD block as its A parameter.

If an input is not connected to the output of another block and you leave the input box empty, the logic block assumes the default value 0 (zero).





EXPR

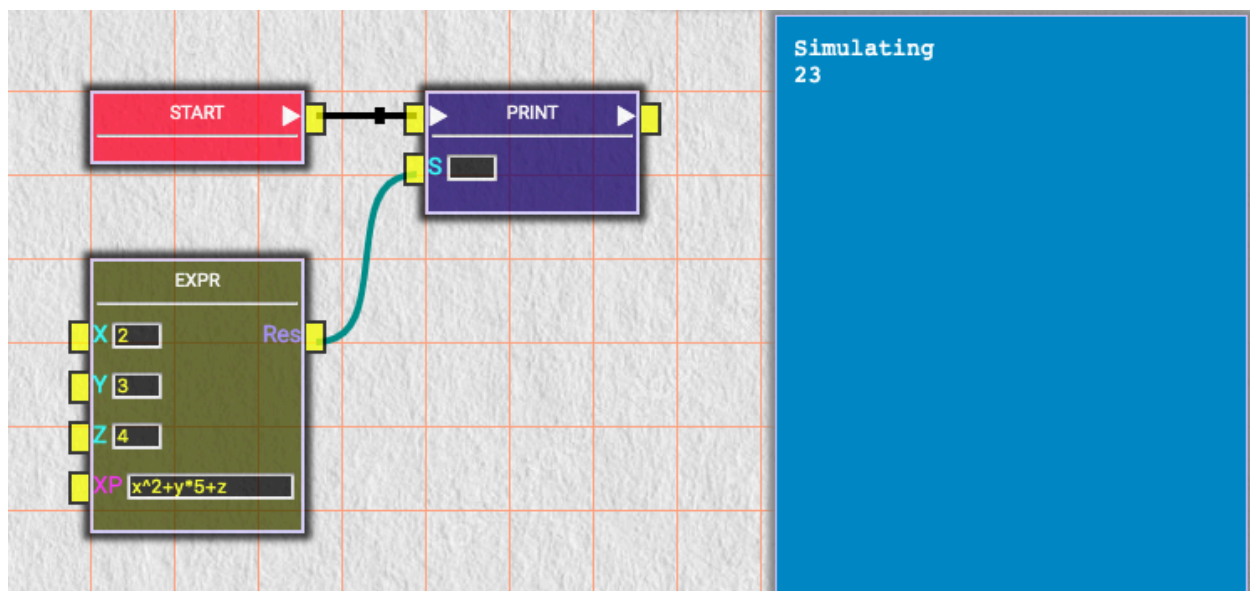
Computes the result of a mathematical expression.

Its inputs X, Y and Z are used as inputs for the function specified in its XP input.

In the example below, we want to print the result of $X^2 + 5.Y + Z$.

Instead of using multiple individual blocks, we are using an EXPR block.

Note that we don't need to use all three inputs, X, Y and Z - we can use EXPR to compute expressions that use only two - or even just one - of its inputs.





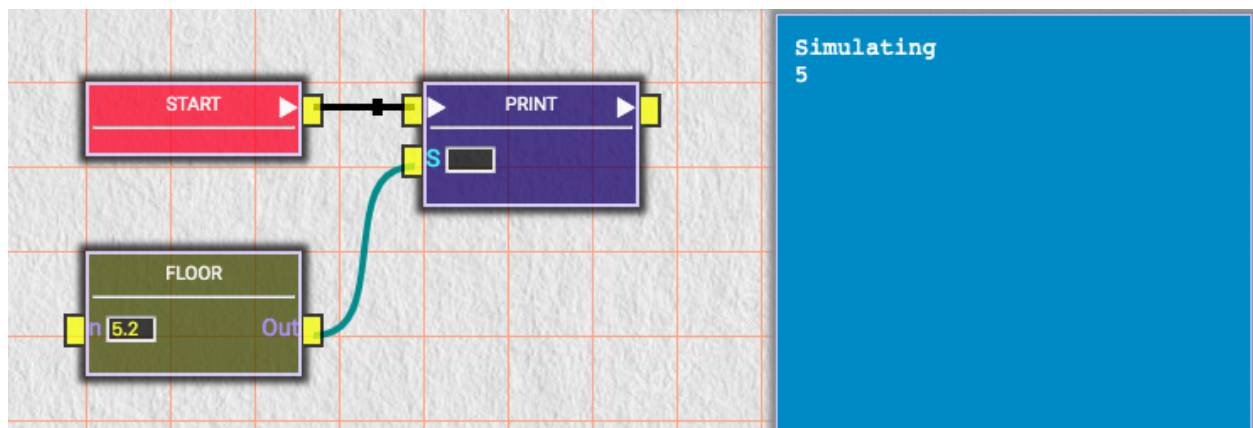
FLOOR

This logic block outputs the mathematical floor of the value of its input “n”.

In mathematics, the floor function is used to round a value down.

If the input is an integer number, that is, if it doesn't have a fractional part, in other words, if it doesn't have significant digits after the decimal point, then its floor value is itself.

If the input value is not an integer value, then this block returns the next integer number down.





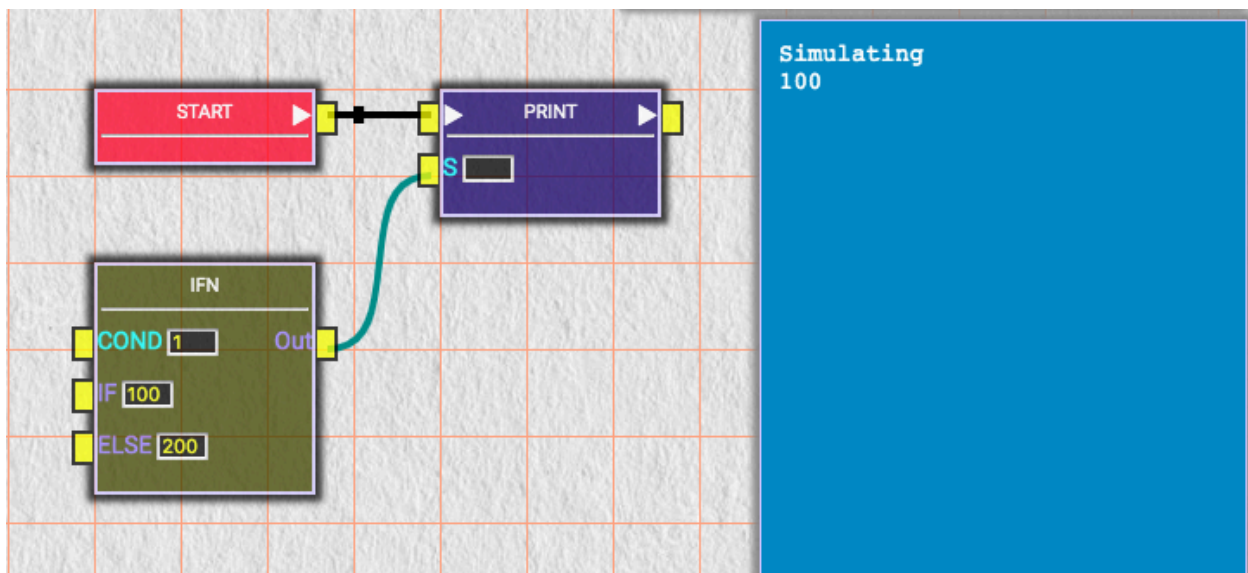
IFN

Selects one of two values, based on its “COND” input.

The selected value is then send to its output.

This block is used mainly to build conditional expressions, making it a bit difficult to understand without context, but that’s actually all it does: if “COND” has nay value other than 0 (zero), the block outputs the value of its “IF” input; if it is 0 (zero), then it outputs the value from its “ELSE” input.

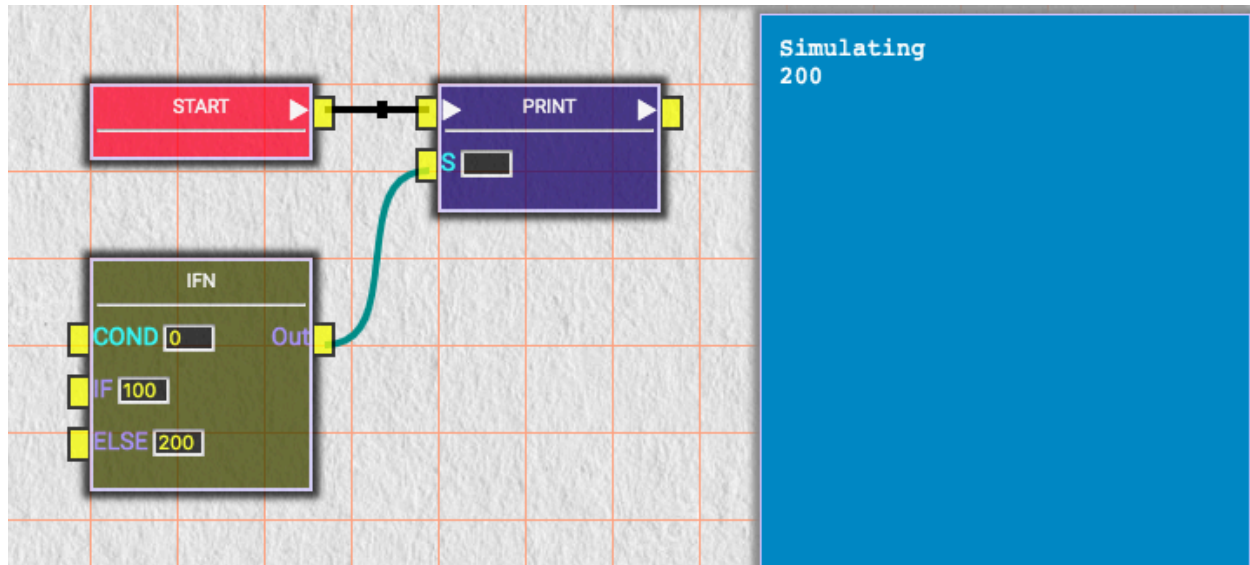
If COND is NOT zero



COND is 1, so the block outputs the value of its IF input, that is, 100.



If COND IS zero / Si COND ES cero / Se COND É zero

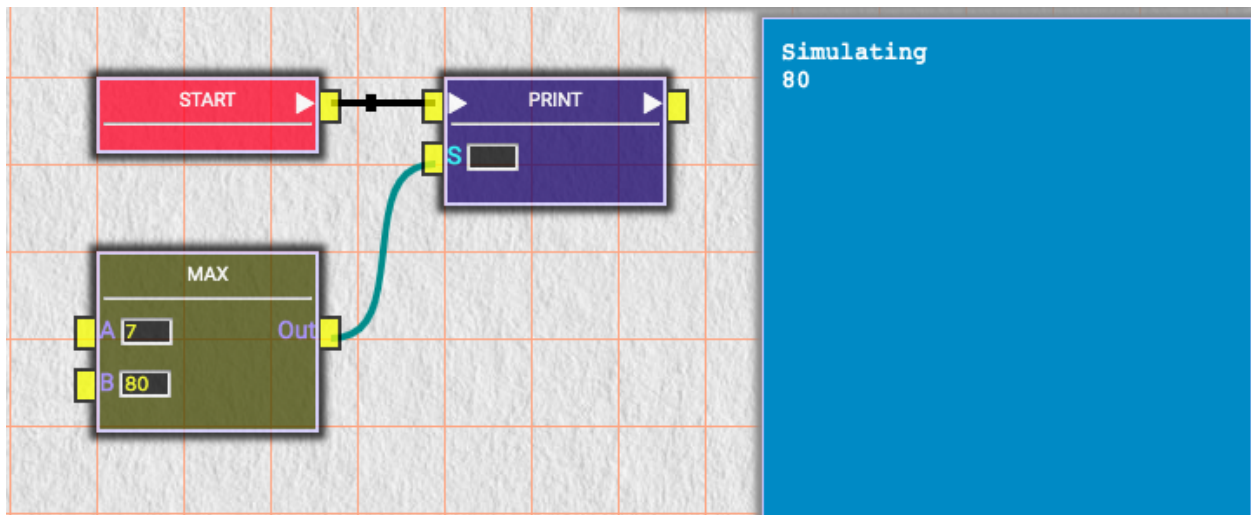


COND is 0, so the block outputs the value of its ELSE input, that is, 200.

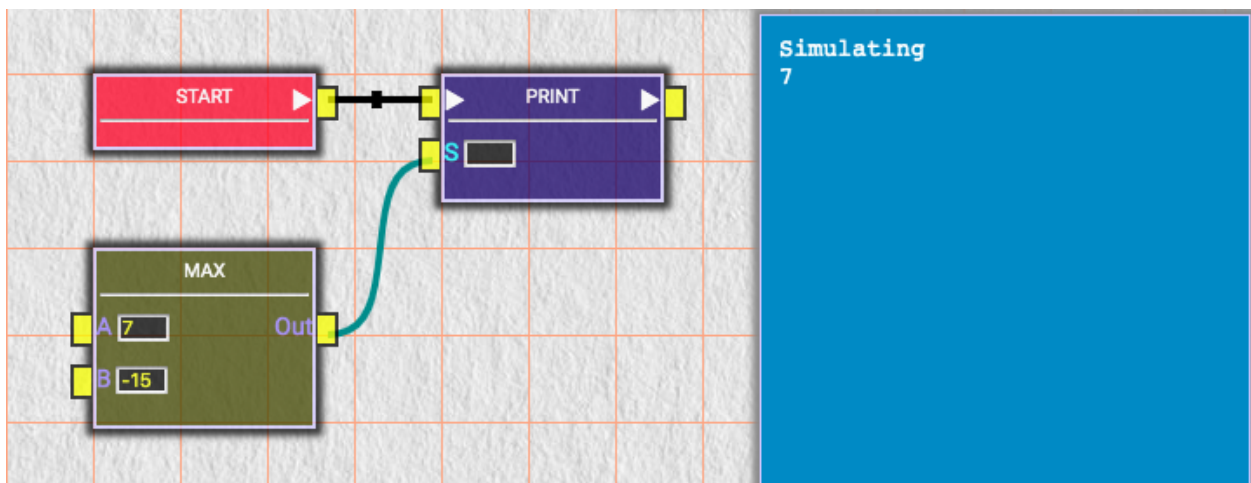


MAX

Compares the two values from its A and B inputs and outputs the biggest one.



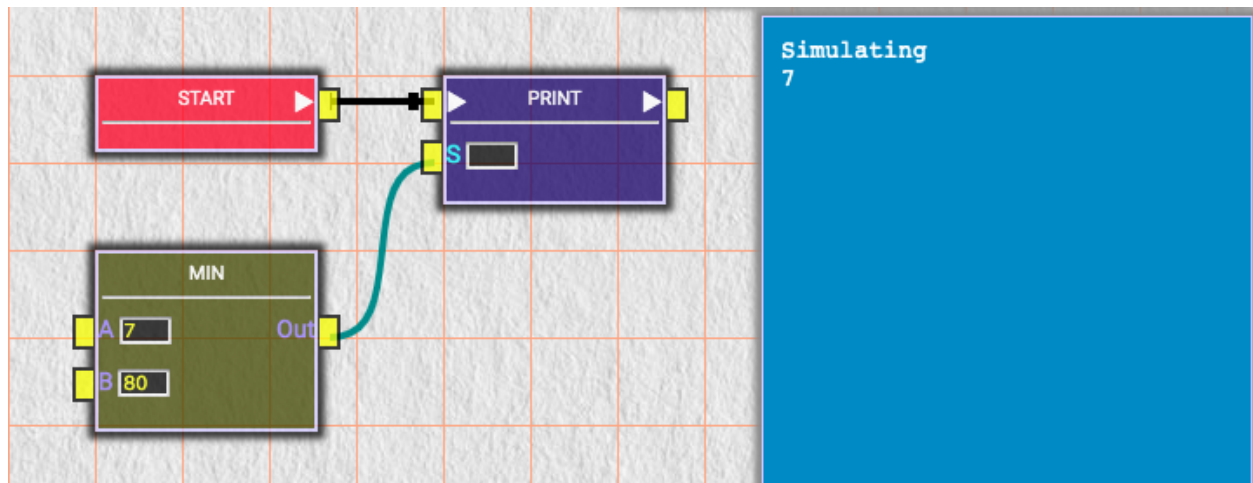
MAX also handles negative numbers.



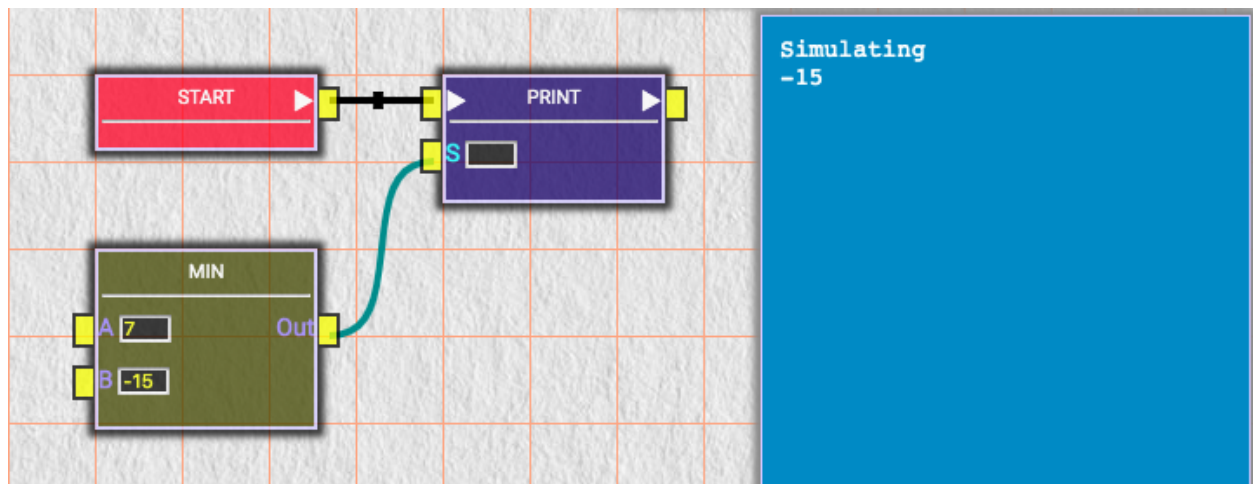


MIN

Compares the two values from its A and B inputs and outputs the smallest one.



MIN also handles negative numbers.

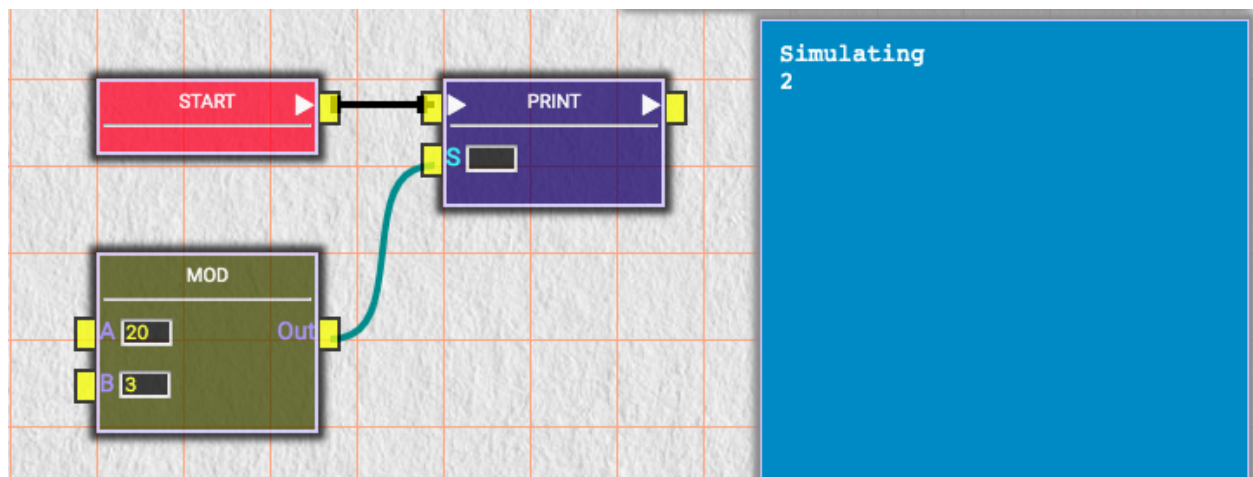




MOD

This logic block computes the mathematical “modulus” operation between the values of its inputs A and B and sends the result to its output.

The modulus operation returns the remainder of the division of A by B - ad, just like in the case of a division, the order of the parameters influences the result.



Whenever A is a multiple of B, the remainder of the division is zero.

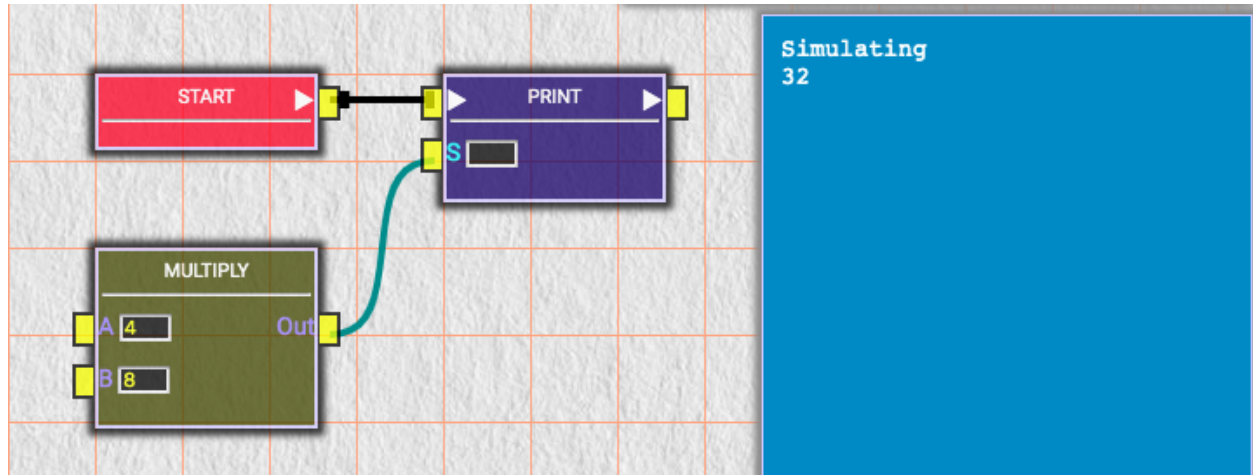
This is very useful and allows for a very easy way to check if a number A is a multiple of a number B; this can be used for testing if a number is even or odd, for example.



MULTIPLY

Multiplies two numbers and outputs the result.

In the example below, we multiply 4 to 8 and print the result.

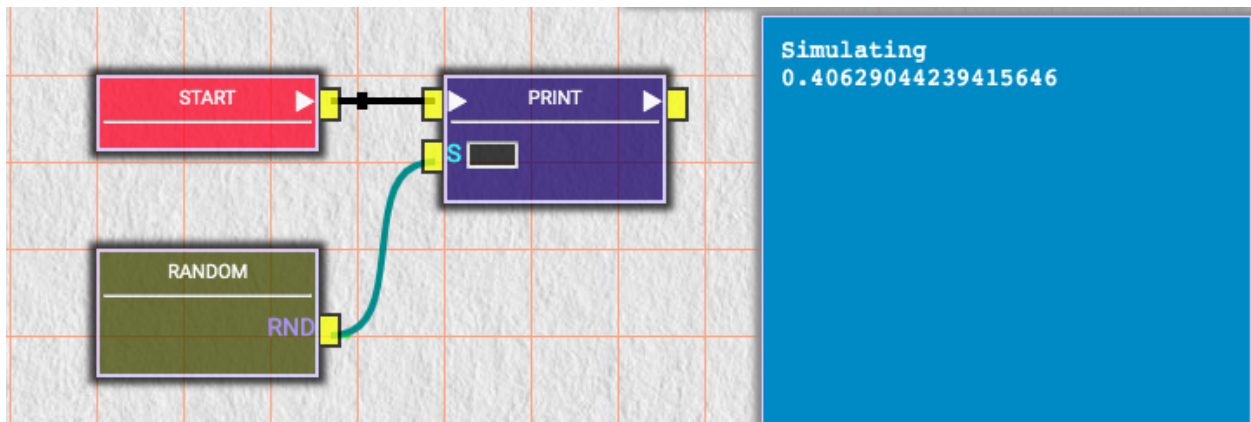




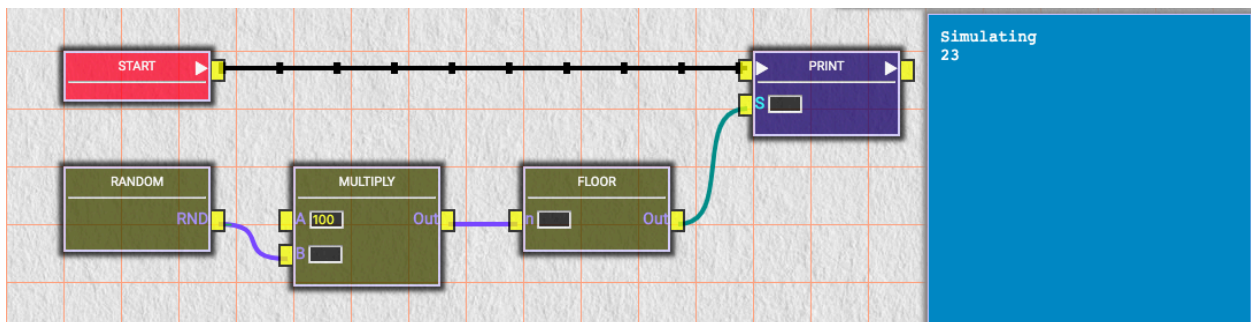
RANDOM

This block outputs a random number between 0.000000 and 0.999999.

This number can be used for simulating noise, or for performing statistical analysis, etc.



Example: you can convert this number to a value between 0 and 99, by combining RANDOM with MULTIPLY and FLOOR.

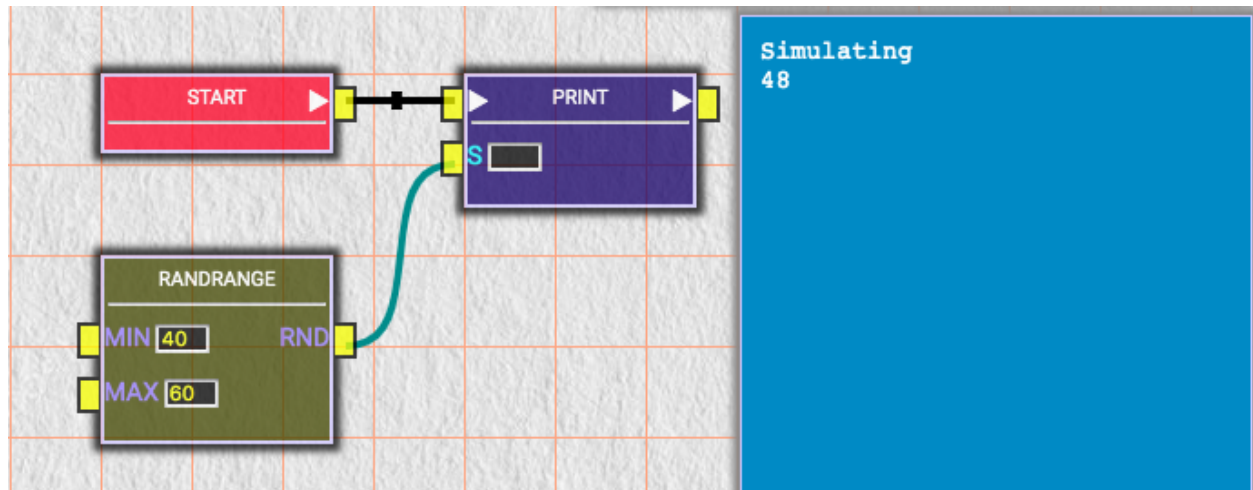




RANDRANGE

This block generates a random integer number between the values specified in the block's MIN and MAX inputs (the randomly generated number includes MIN and MAX).

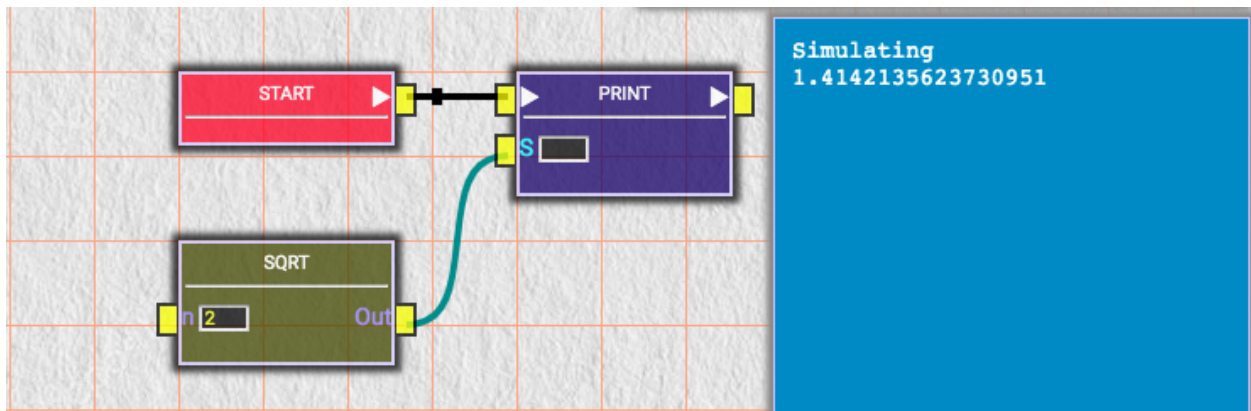
Note that MIN and MAX are also included in the range, and can also be generated.





SQRT

This block takes the value from its “n” input and sends its square root to the output. The value must be positive - the logic block does not handle complex numbers.

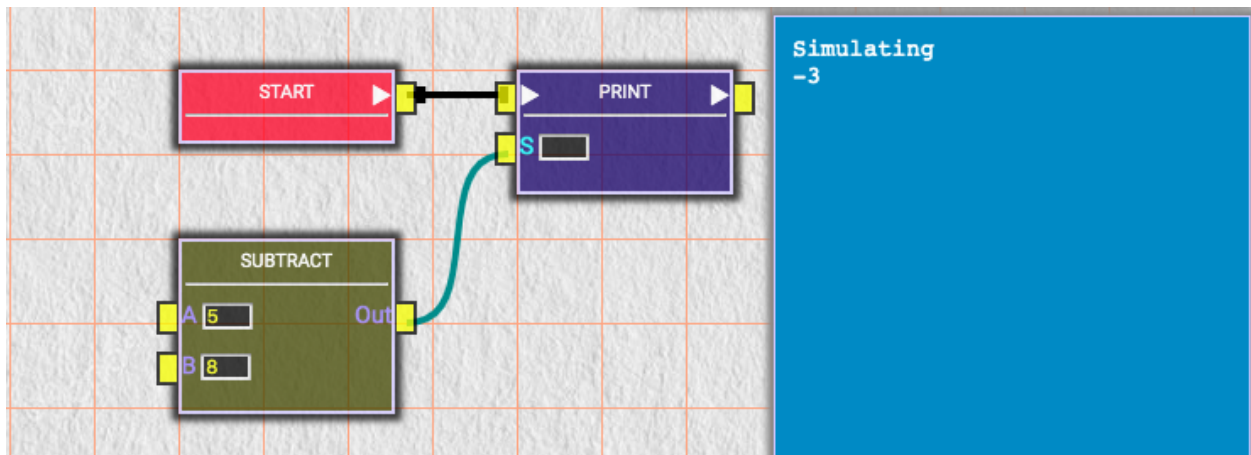




SUBTRACT

Subtracts two numbers and outputs the result.

In the example below, we subtract 8 from 5 and print the result.





BOOLEAN BLOCKS

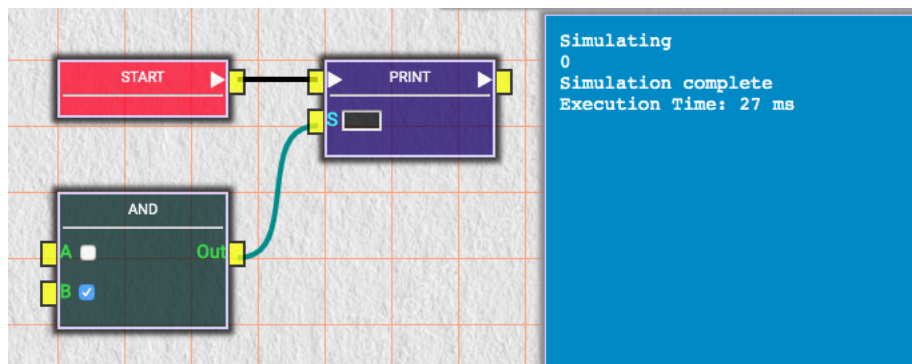
AND

The AND block sends a TRUE to its output when both of its inputs are TRUE.

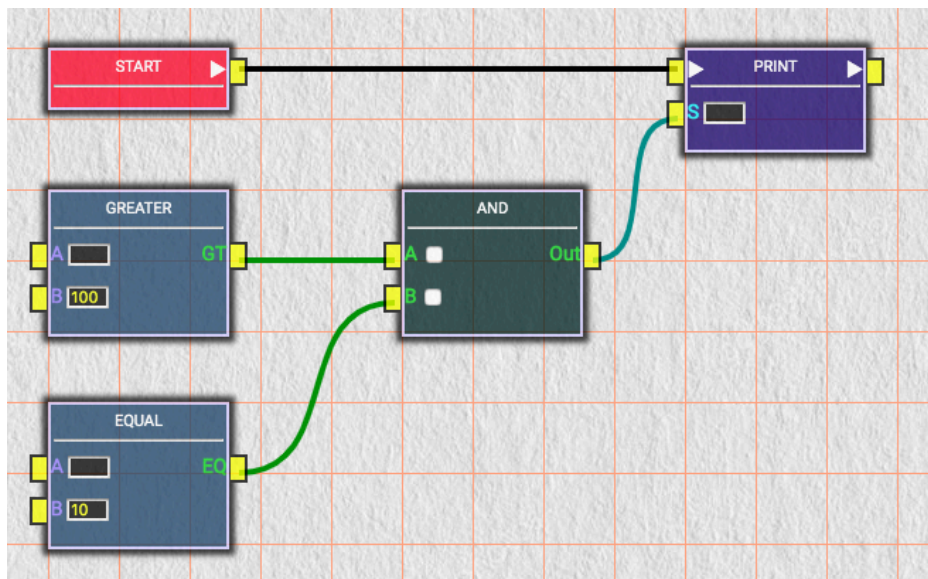
So, if either A or B (or both) is 0, the AND output is set to 0 (FALSE).

Only when A **AND** B are both TRUE (non zero), the output is set to 1 (TRUE), hence the name AND.

A and B can be directly set, by checking the input's checkboxes - checked sets its input to 1, unchecked sets the input to 0 - but when an input is connected to the output of another block, the checkbox state is ignored.



Logic blocks like AND are used to combine comparisons and conditions, such as “X is greater than 100 AND Y equal to 10” - in this example, you would use a GREATER block and an EQUAL block to test for the conditions and would combine them with the AND block.





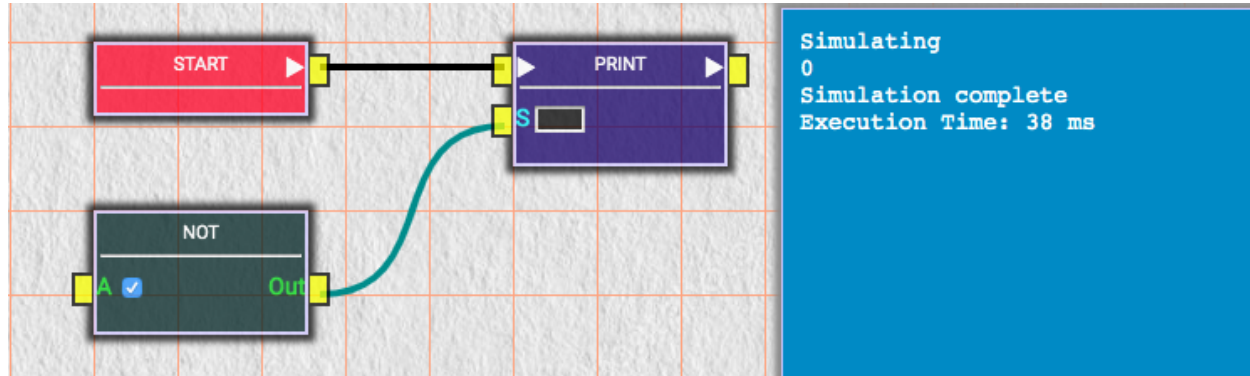
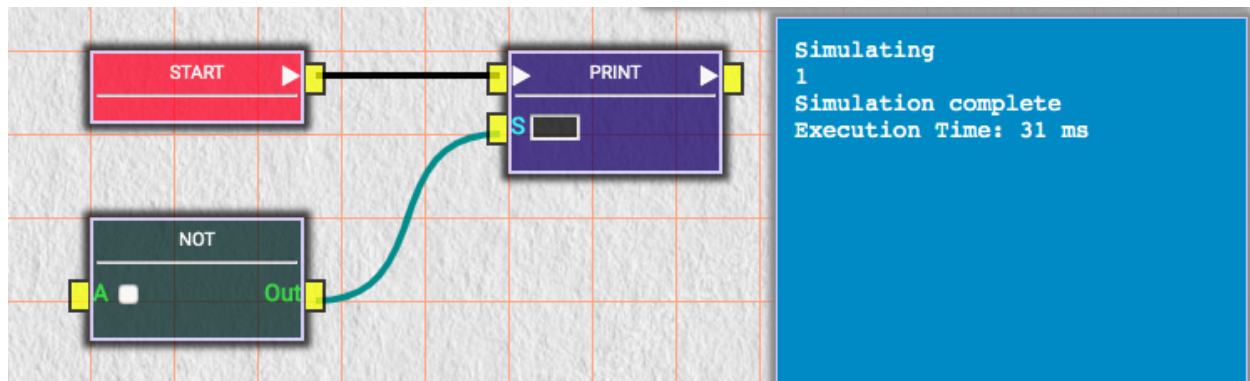
NOT

The NOT block is used to “invert” the logic state of a value.

When its input is FALSE, its output is set to TRUE.

When its input is TRUE, its output is set to FALSE.

In logic diagrams, a 0 (zero) is interpreted as FALSE, and any other value (either positive or negative) is interpreted as TRUE.





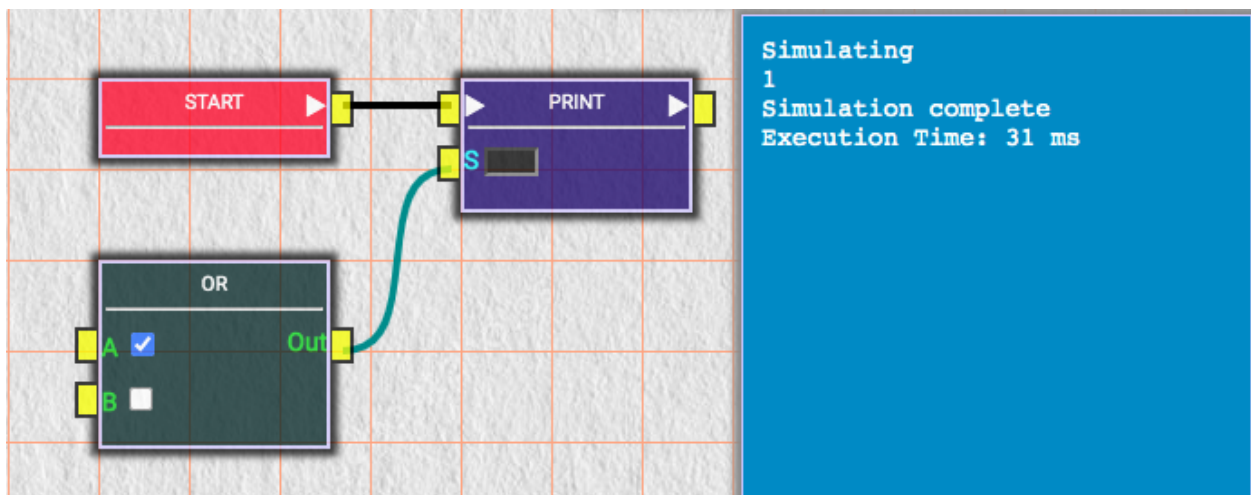
OR

The OR block sends a TRUE to its output when at least one of its inputs are TRUE.

OR outputs a FALSE only if both inputs are FALSE.

If either input A **OR** input B is TRUE, the output is set to 1 (TRUE), hence the name OR.

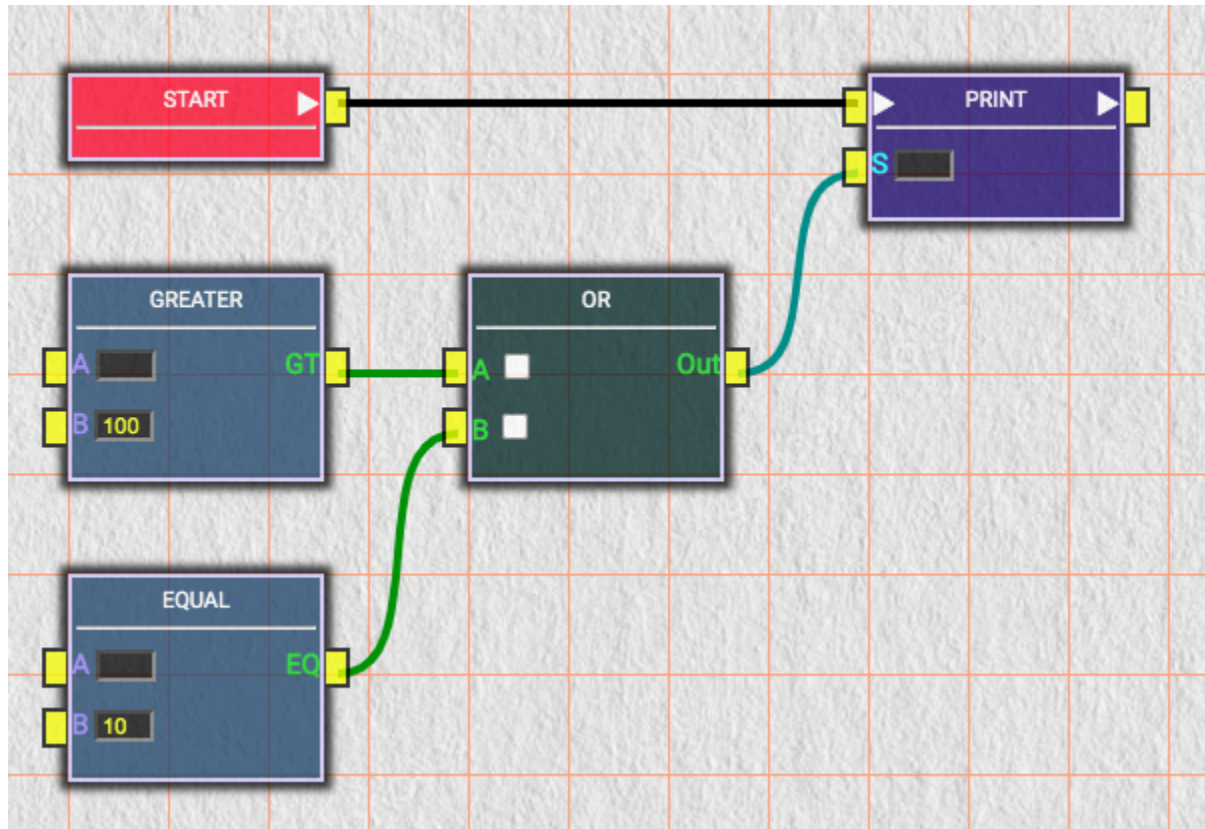
A and B can be directly set, by checking the input's checkboxes - checked sets its input to 1, unchecked sets the input to 0 - but when an input is connected to the output of another block, the checkbox state is ignored.





Logic Blocks

Logic blocks like OR are used to combine comparisons and conditions, such as “X is greater than 100 OR Y equal to 10” - in this example, you would use a GREATER block and an EQUAL block to test for the conditions and would combine them with the OR block.





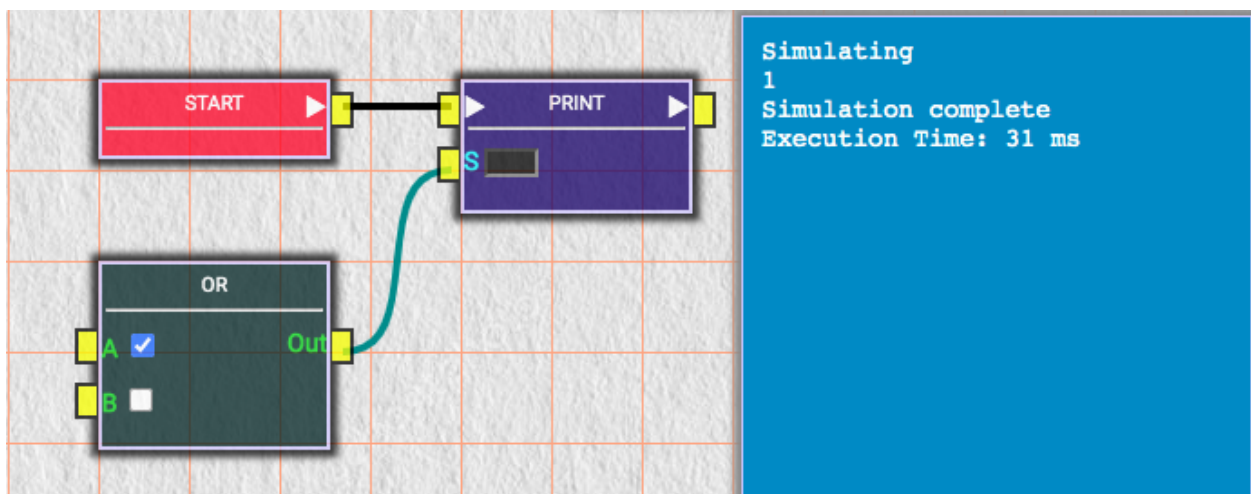
XOR

The XOR (also known as “EXCLUSIVE-OR”) block sends a TRUE to its output when only one of its inputs are TRUE.

XOR outputs a FALSE if both inputs are TRUE or if both inputs are FALSE.

In other words, XOR only outputs TRUE if its inputs are different from one another..

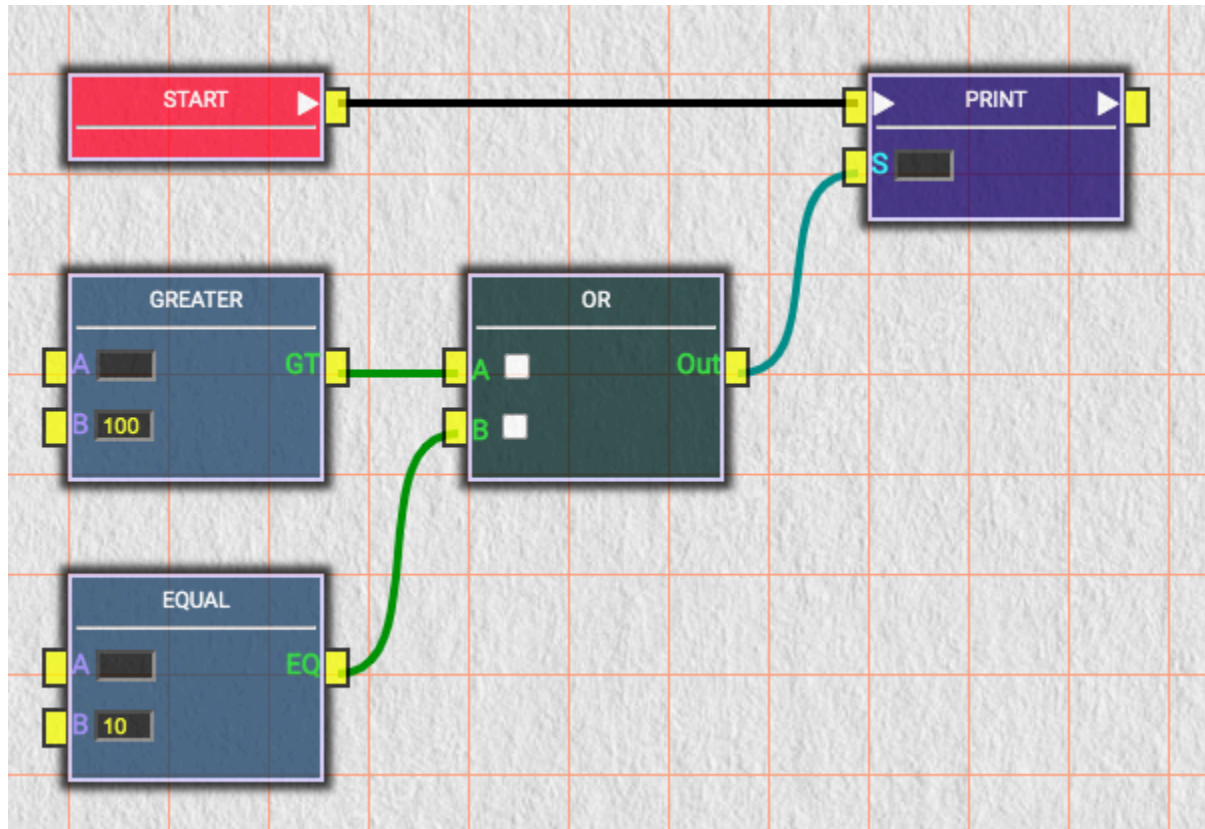
A and B can be directly set, by checking the input's checkboxes - checked sets its input to 1, unchecked sets the input to 0 - but when an input is connected to the output of another block, the checkbox state is ignored.





Logic Blocks

Logic blocks like XOR are used to combine comparisons and conditions, such as “if either, but not both, X is greater than 100 or Y equal to 10” - in this example, you would use a GREATER block and an EQUAL block to test for the conditions and would combine them with the XOR block.





COMPARATORS

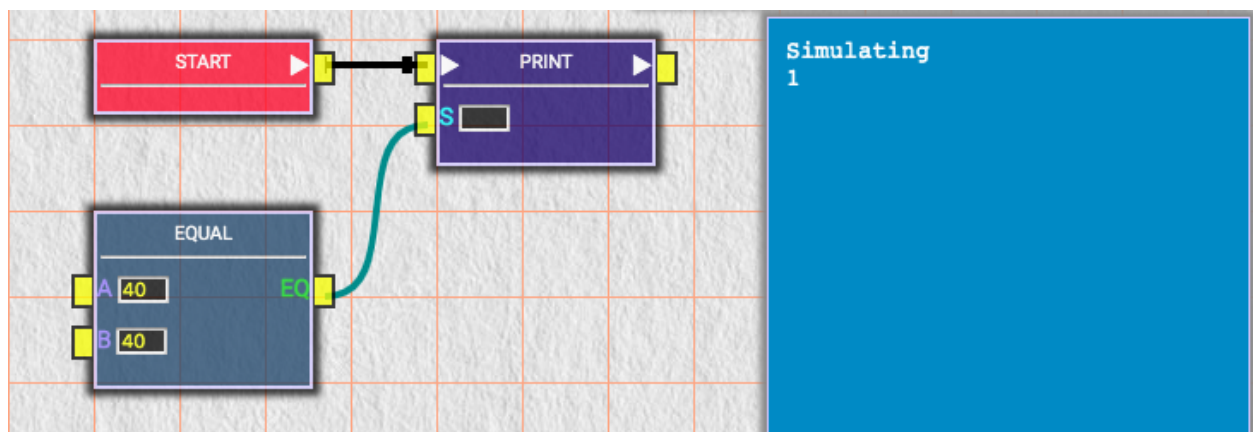
EQUAL

Compare the values in inputs A and B.

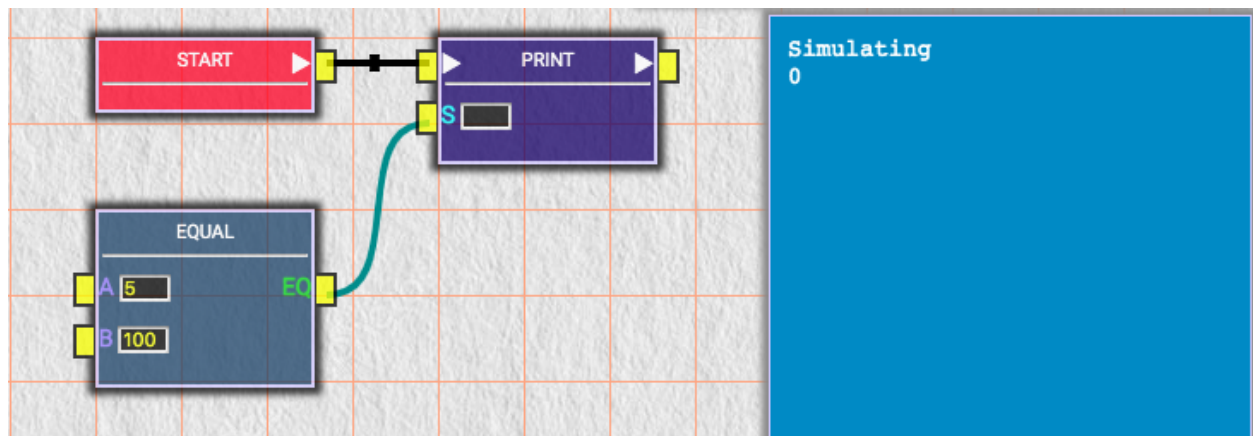
If A and B are equal, this block will output a 1 (one).

If A and B are different, a 0 is sent to the output.

Comparators output a 1 to indicate TRUE.



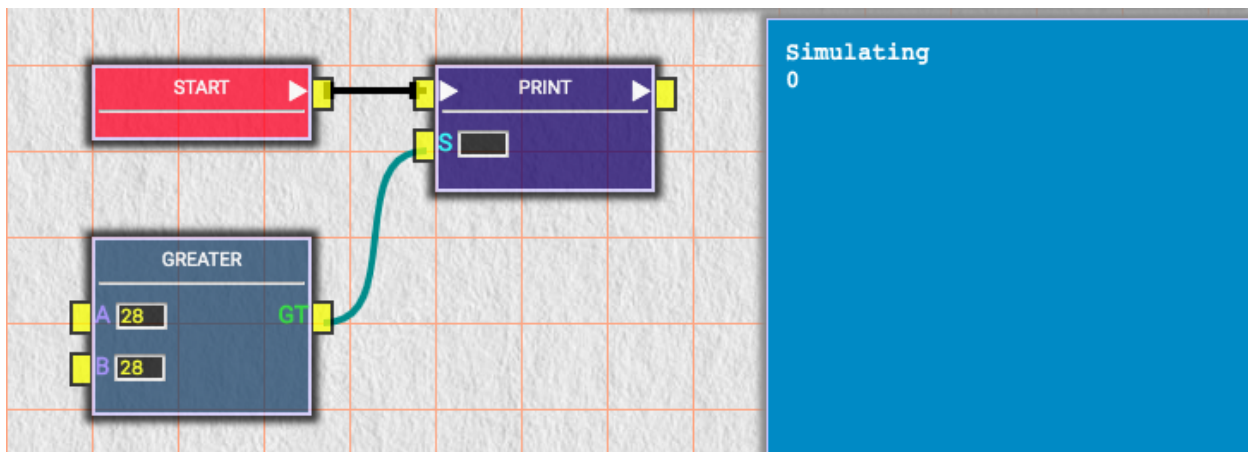
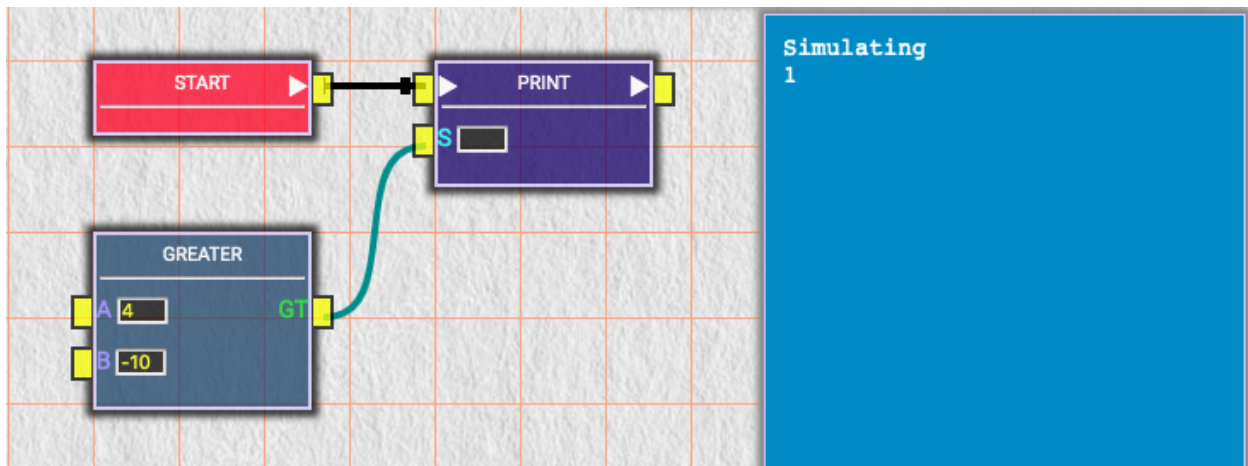
Comparators output a 0 to indicate FALSE.





GREATER

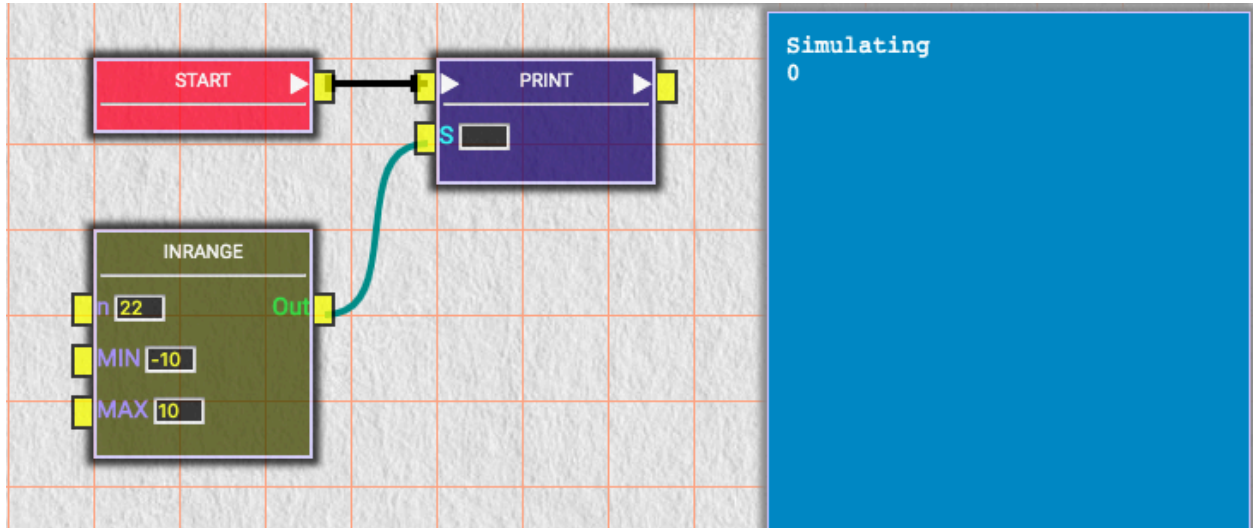
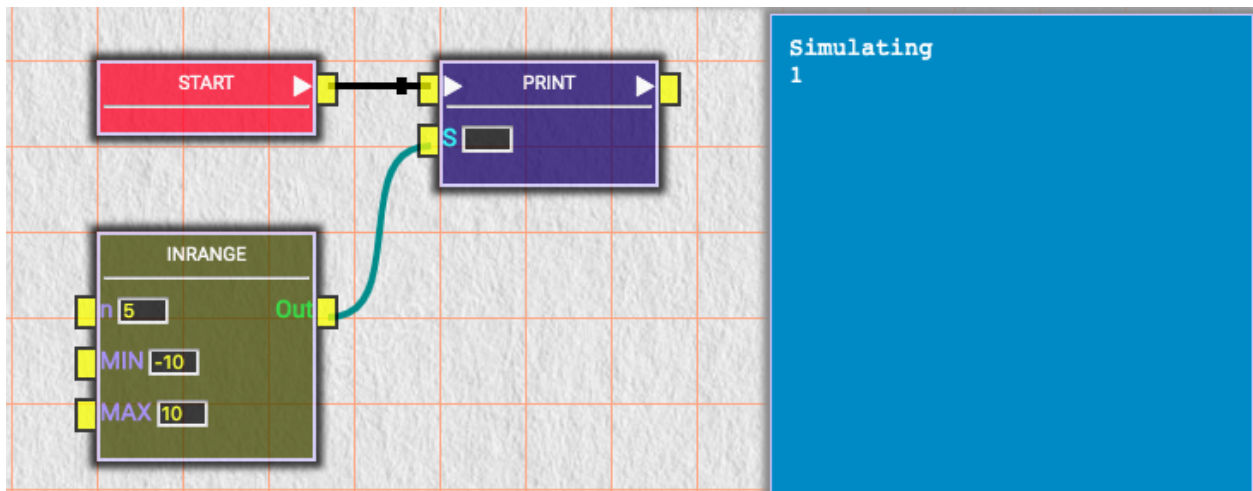
This block checks if value in input A is greater than the value in input B.
If A is greater than B, a 1 is sent to its output; a 0 is sent, otherwise.





INRANGE

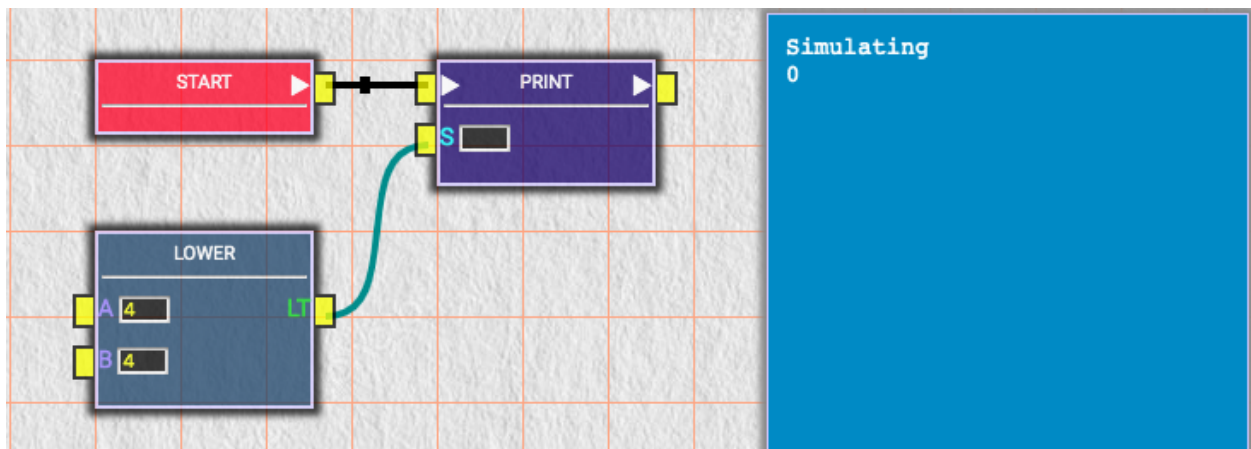
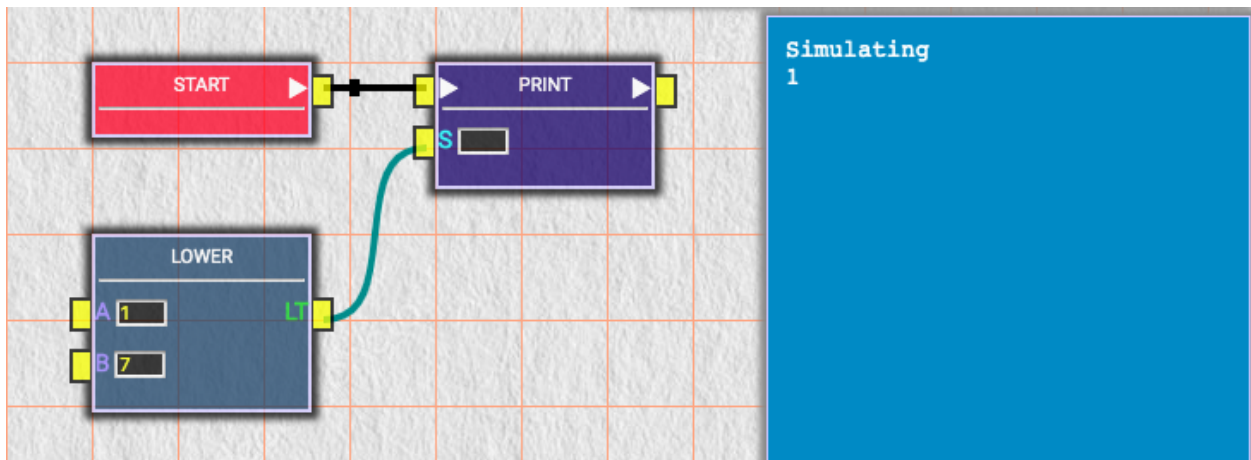
This block checks if the value in its “n” input is within the range specified by its MIN and MAX. If n is within the specified range, a 1 is sent to the output, otherwise a 0 is sent.





LOWER

This block checks if the value on input A is lower than value on input B.
A 1 is sent to the output if A is less than B, otherwise a 0 is sent, instead.





CONTROL BLOCKS

IF

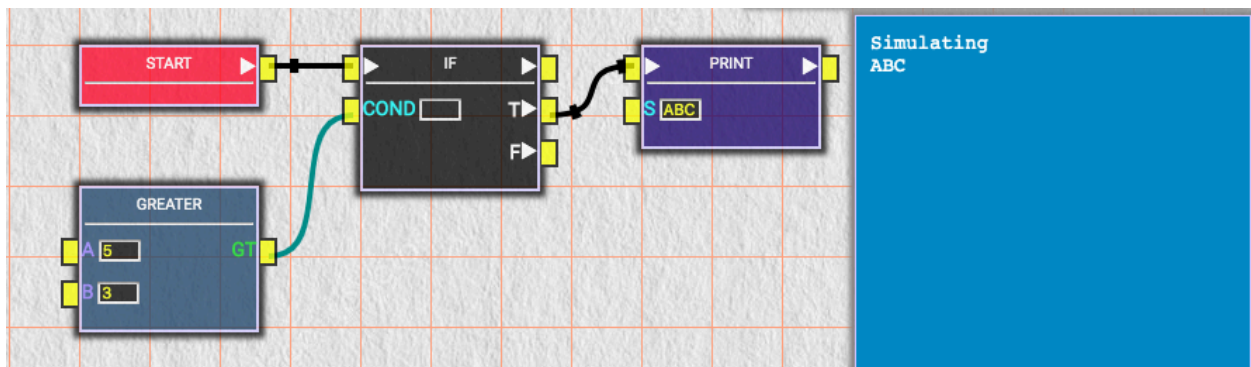
This block tests the value in its “COND” input and triggers one of its outputs, depending on whether that value is TRUE or FALSE.

If the condition is TRUE, it triggers its “T” output (T stands for TRUE).

If the condition is FALSE, it triggers its “F” output (F stands for FALSE).

The key concept here is: what is TRUE and what is FALSE?

In block diagrams (and in many textual programming languages), 0 (zero) is treated as FALSE, and any other value (even negative values) is treated as TRUE.



In the previous example, the block GREATER is comparing numbers 5 and 3.

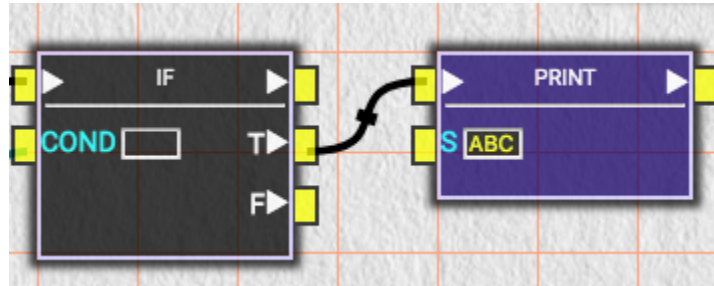
Since 5 IS greater than 3, this block outputs a 1, meaning TRUE (“yes, 5 is greater than 3”).





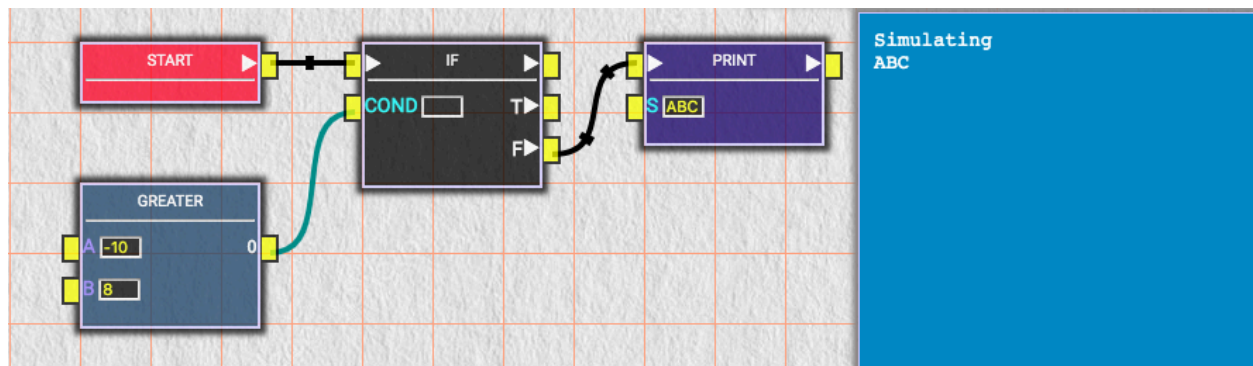
Then this “1” is fed into the IF block.

The IF block, on its turn, sees a “1”, that is, a TRUE in its “COND” input, triggers its “T” output, which is connected to the PRINT block.



If the value on the COND input was FALSE, then the IF block would trigger its F output, instead, like in the example below.

In this example, GREATER is outputting 0 (because its A input is not greater than its B input),



One important detail, though, is that the IF block also has another trigger output, unnamed, which is executed after its TRUE or FALSE output is triggered.

So, in short, what IF does is: first, checks COND and triggers T if COND is not zero, else triggers F; after triggering either T or F (based on the value of COND), IF triggers the unnamed output.



LOOP

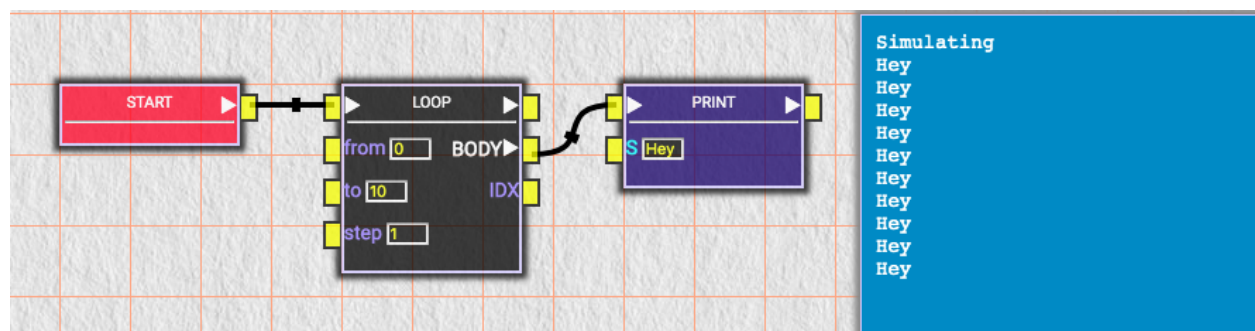
This block activates its BODY output a number of times - the number of times is specified by its “from”, “to” and “step” inputs.

When activated, this block counts, starting from its “from” value, up to the value specified in its “to” input, adding “step” each time.

Every step of the count, the BODY output is activated.

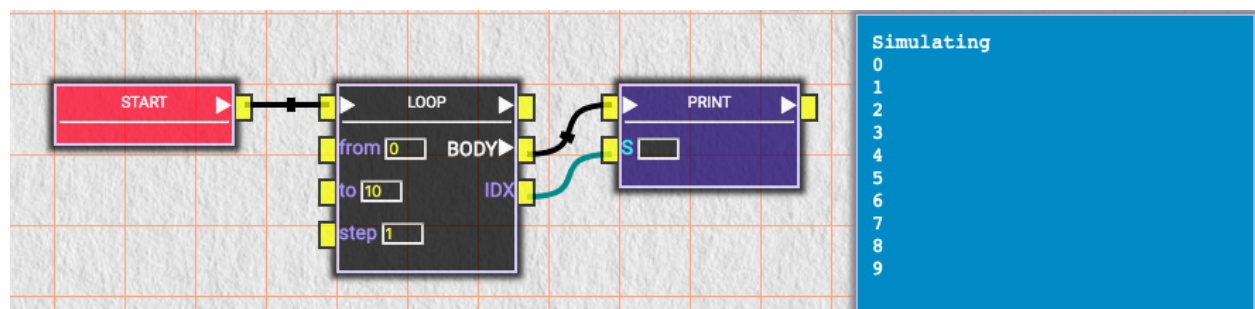
In the following example, from is 0, to is 10 and step is 1. This tells the LOOP block to count from 0 to 10, 1 step at a time (e.g.: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9), activating the BODY output each time (for a total of 10 times).

Since the BODY output is connected to the trigger of the PRINT block, which prints “HEY”, the end result of this logic diagram is to print “HEY” ten times.



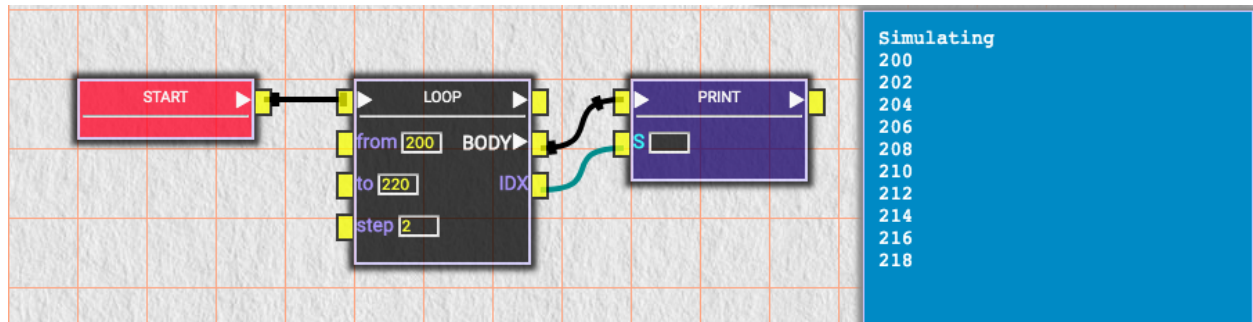
The LOOP block also offers a very valuable output, IDX, which tells us the current value of its internal count - this value can be sent to other blocks, where it can be used in computations, or just to inform the count value.

In the next example, we connect IDX to the PRINT input, so we can show the LOOP count.





We can easily configure LOOP to count in different ways - here, we set it to count from 200 to 220, 2 at a time:



You will notice in the previous example that the LOOP block didn't actually count up to 220.

That's because the "to" input tells the block **when to stop counting**.

This may sound weird, but it will make sense as you work with logic diagrams.

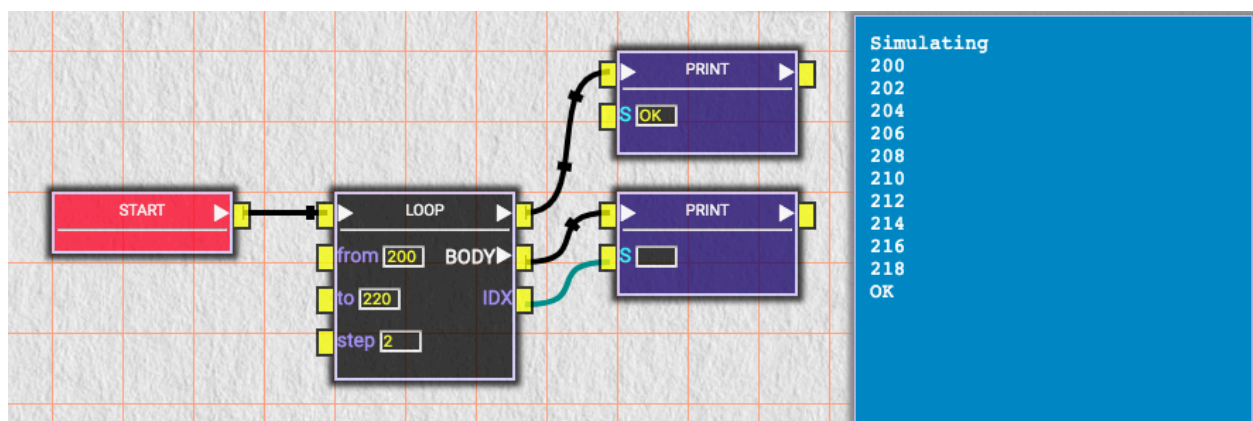
That is common to most programming languages - as is the fact that counting usually starts from zero, not one (that's another thing you'll get comfortable with, as you build your logic diagrams).

Another important thing to notice is that LOOP also has an unnamed trigger output, which is activated when the counting reaches its end.

In other words, this output is activated when the LOOP block has finished counting.

Below you'll find a modified version of the previous example - we just added a PRINT block and connected its input to the LOOP's end-of-count (unnamed) output pin.

So, this version counts from 200 to 220, 2 at a time, printing each value, and when it finishes counting, it prints a message, "OK".





REPN

This block is a simplified version of the LOOP block.

It only has one configuration pin, namely “n”.

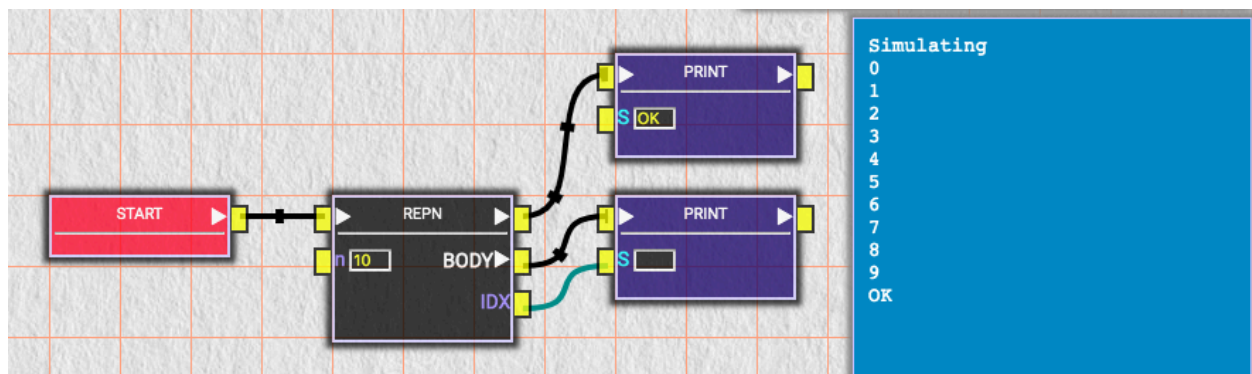
REPN counts from 0 to “n”, 1 step at a time.

“n” is not included in the counting, that is, “n” indicates when to stop counting.

Besides, REPN always counts one step at a time.

You can think of the REPN block as being a LOOP count where “from” is always set to 0 (zero) and “step” is always set to 1.

Other than that, REPN works exactly like LOOP - including the end-of-count signal being triggered when REPN reaches the specified end value.



REPN is less configurable than LOOP, but it is also smaller and uses up less space.

And, to be clear, most of the times you need a counter, you'll most likely need it to start from 0 and count 1 step at a time, anyway.



SEQUENCE

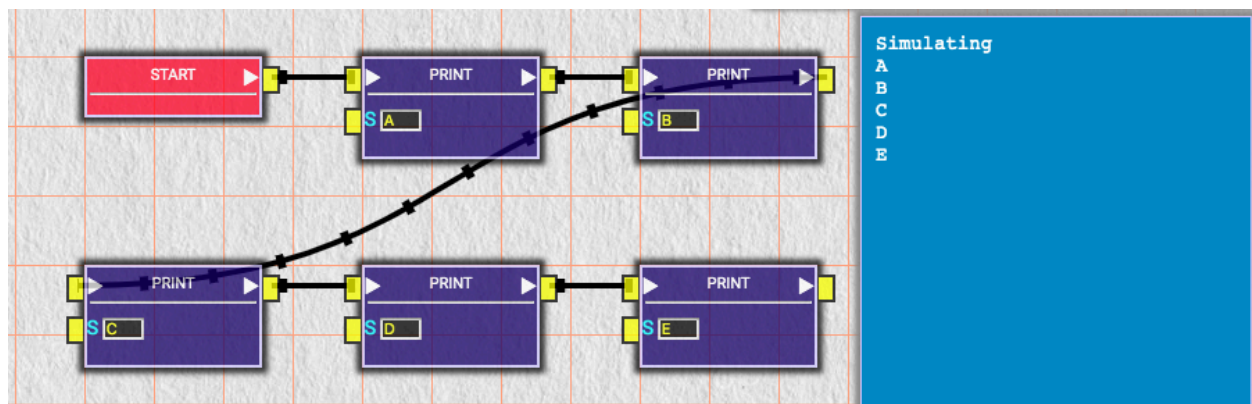
This block allows you to execute several other blocks, in a sequence.

To be clear, SEQUENCE is not really necessary, because all you have to do, in order to execute a sequence of blocks, is to connect one's trigger output to the next trigger input.

But, sometimes, you may want to organize a series of steps in some process, making it clear to other people who have to maintain your logic diagram that these processes are divided in some logical way.

So, all SEQUENCE does is help you “organize” your diagram.

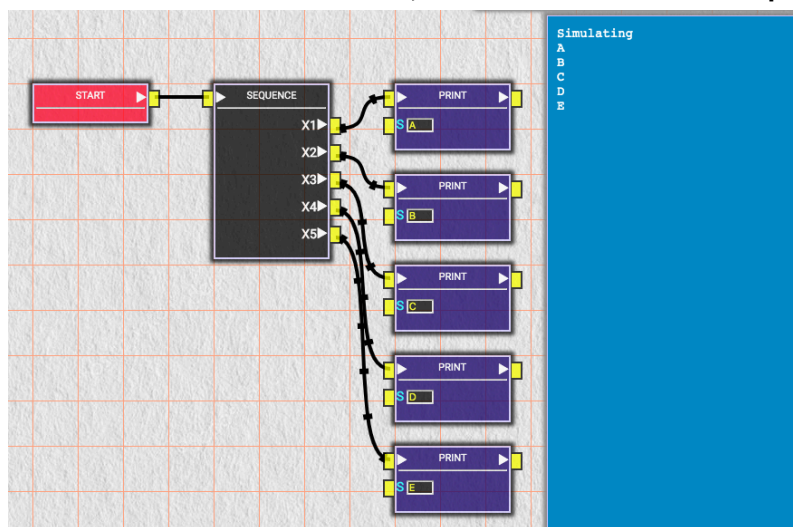
The next couple of diagrams have the exact same practical result, the only difference is we have used SEQUENCE to “organize” the second one in a different way.



Same thing, but using SEQUENCE to “organize” the diagram.

Lo mismo, pero usando SECUENCIA para "organizar" el diagrama.

A mesma coisa, mas usando o SEQUENCE para "organizar" o diagrama.





EVENT BLOCKS

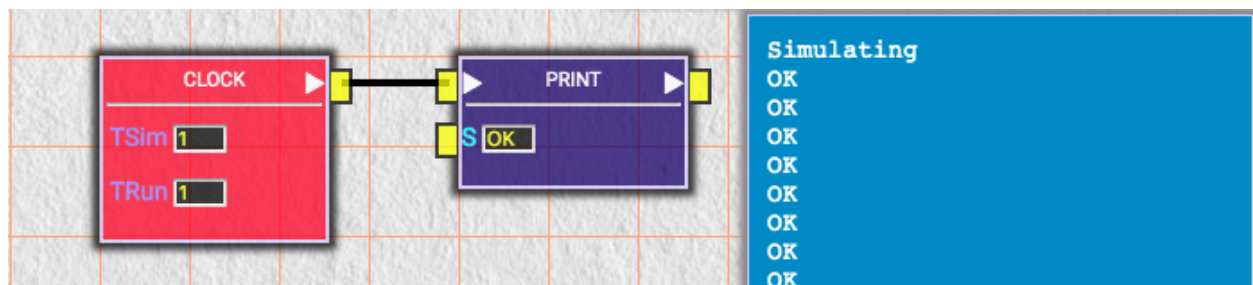
CLOCK

This block triggers the execution of the blocks that are connected to it, at a repeating interval (by default, this interval is set to 1s, so CLOCK activates the block that's connected to it every second).

On its inputs, CLOCK allows one to configure the simulation time, TSim and the run time, TRun.

TSim is used when you run the simulation on your computer and TRun is used when you run your logic diagram on actual hardware (on an Arduino or on a Raspberry Pi, for instance).

The following example prints "OK" every second.



In real world applications, the CLOCK block is used to activate your logic at regular intervals, to check inputs, activate outputs, compute your algorithms, etc...

In practice, a 1s interval is usually too big - a lot of things can happen in one second that will cause your rocket - or other system - to fail.

So, what you need is to configure CLOCK to repeat as quickly as possible - the problem with that is that if CLOCK is too fast, your logic diagram may not have enough time to perform all of its tasks before the next execution is started.



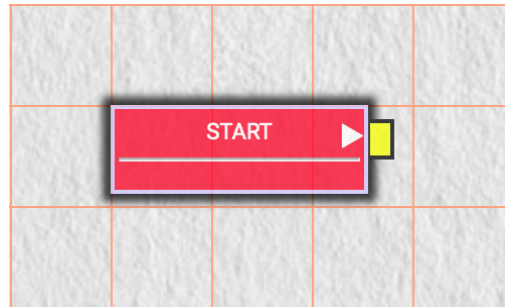
In your real world applications, you should start with a 0.1s CLOCK, which is frequent enough to update most practical systems, while still allowing plenty of time for your control logic to complete before the next execution - so, start with 0.1s and change it if/as needed.





START

Start operation.



This block is activated when you start the simulation.

It is executed only once, and its only function is to trigger the first block in your control diagram.

START is the starting point of execution of your logic diagram.

If you have a CLOCK block, you don't necessarily need to have a START block (and vice-versa), but if you have both, START will be executed before the first CLOCK cycle is activated.

Notice that you don't have to include both a START and a CLOCK block in your diagram, but you need at least one of them.

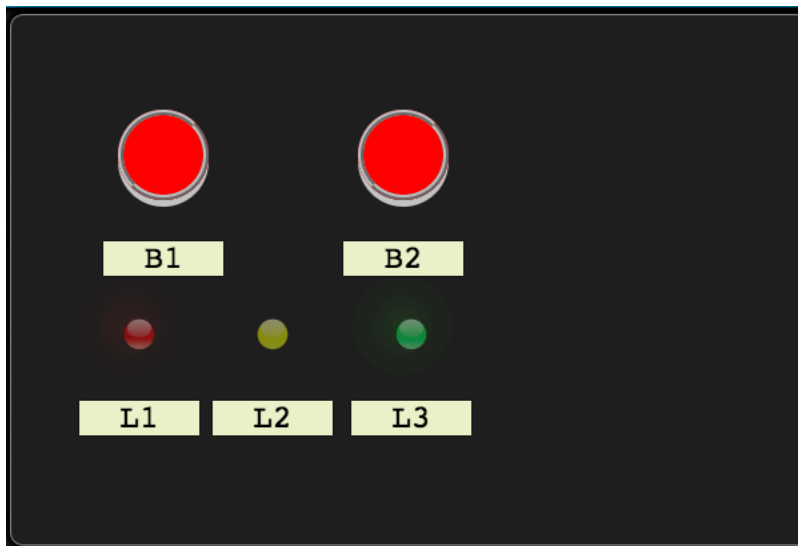


INTERFACE BLOCKS

In order to understand the following Logic Blocks, we are going to assume you're using the "TEST LAB" virtual mission.

The TEST LAB contains a few simulated devices you can play with: a couple of buttons, labeled "B1" and "B2" and three LED's, "L1" (red), "L2" (yellow) and "L3" (green).

These devices simulate real devices, so you can press the buttons and lit the LED's.





INP

This block reads a value from the simulation.

The value it reads depends on the type of simulation you are building your diagram to control.

It could be the current temperature of a rocket fuel tank in a rocket launch simulation, or perhaps the percentage of water level in a water tank in a water tower simulation, etc.

The value that is read from the simulation depends on what type of device is being simulated: a button, for instance, when read will return a value 0 or 1 (0 if not pressed, 1 if it is pressed); a temperature sensor will return a value between 0 and 800, indicating the temperature in Kelvin.

In the following example, we read the current state (on = 1 or off = 0) of a button labelled “L1”.





OUT

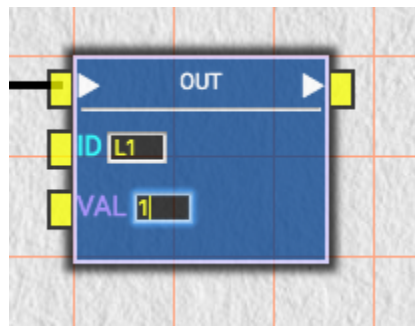
This block allows you to send values to simulated devices.

The type of device depends on the type of simulation your block diagram is designed to control.

A device could be a motor in a simulation of a robotic arm, or perhaps an LED to indicate some process has been completed.

The value you send will depend on the type of device you want to control and on what you want the device to do: you could send a 1 to an LED, to turn it on, or 0 to turn it off; or you could send a 50 to a motor, to set its speed to 50% of its maximum speed, etc.

In the following example, we are turning on the red LED (labelled “L1”) in the TEST LAB mission..



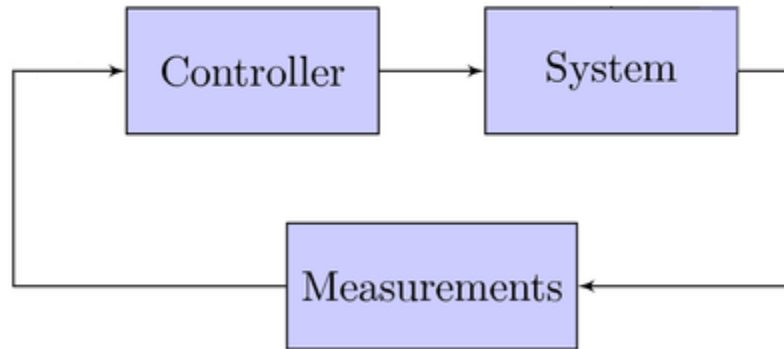
OUT and INP are the most important logic blocks in a logic diagram, because they are your means of connecting your logic to the simulation.

After all, what is a logic diagram? It is a combination of logic blocks that are connected in a way to monitor a system, to constantly read its state and to trigger various operations based on that state - and to send control information to devices that operate on that system.

This is what we call the “main loop”.



At the end of the day, all control systems are a “main loop” - a logic system that reads data from input devices, applies some control logic to those values and sends its results to control devices, in order to keep said system within the required ranges.

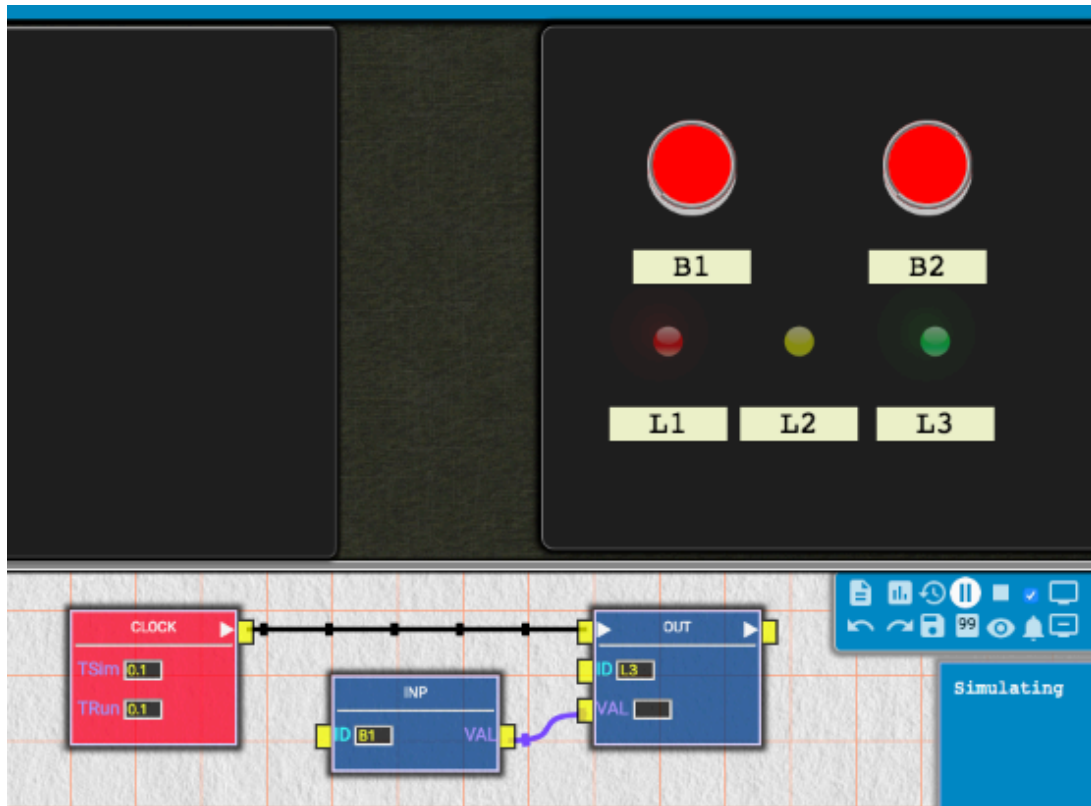


The main loop can be as complex as controlling the attitude actuators of a rocket, for a propelled landing, or as simple as controlling an LED with a button - and anything in between.

The following example you will find a simple logic diagram to use a button labelled “B1” to control the green LED (labelled “L3”) in the TEST LAB.



By running this simulation, you'll be able to turn the LED "L3" on and off, by pressing and releasing button "B1", respectively.



In the previous example, we are reading the value from button "B1" and sending it to the "L3" LED.

If the button is pressed, INP will receive a "1", which is sent to OUT - activating the LED.

If the button is not pressed, INP will get a "0", which is sent to OUT - deactivating the LED.

In other words, this logic diagram "uses button B1 to control LED L3".

But you will notice that OUT is being triggered by a CLOCK block, configured for a cycle of 0.1s (1 tenth of a second).

The CLOCK block defines how often the control loop is going to be executed - in most cases, 1/10 of a second works perfectly, but you may need to adjust it to run faster or slower, depending on the system you want to control.



TIME

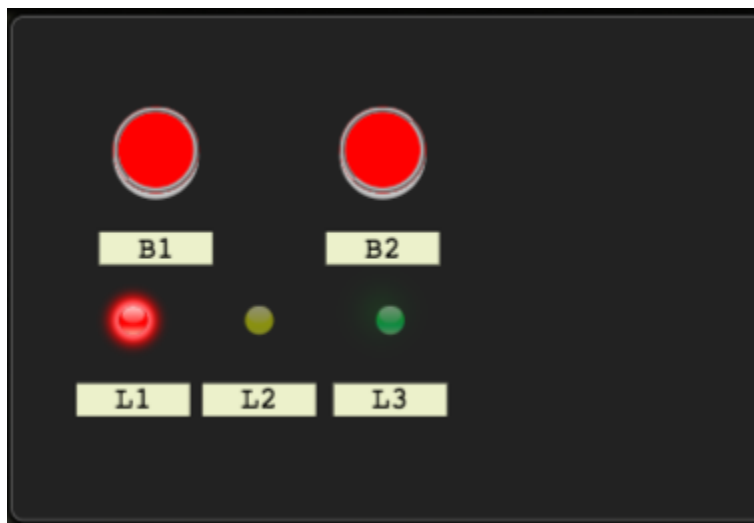
DELAY

This block pauses the execution for the specified time. Execution is resumed right after the specified time is elapsed.

IMPORTANT: all logic execution is blocked during the delay, so you should use this block carefully, usually for small delays - if you use a large delay, say 1s, everything will be stopped during the 1s, which in most applications will result in system failure.

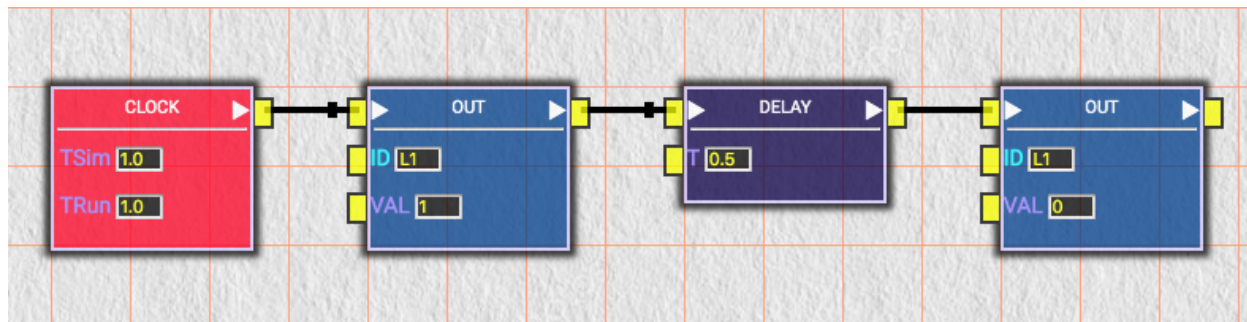
DELAY was designed for small delays, in the 10ms to 50ms that are sometimes required for performing actions like keyboard debouncing, sensor stabilizing time, etc.

You may use DELAY to pause execution for several seconds in systems that don't require quick response to changes in the application state, though - like in the following example, which blinks an LED.





Blinking LED using DELAY/LED parpadeante usando DELAY/LED piscando usando DELAY



First, we turn on LED labelled “L1”, then we pause execution for ½ a second, then we turn “L1” off.

Then execution stops, because there’s no other logic block in our diagram.

Then execution is restarted by CLOCK, after 1s - the result is an LED blinking every second.

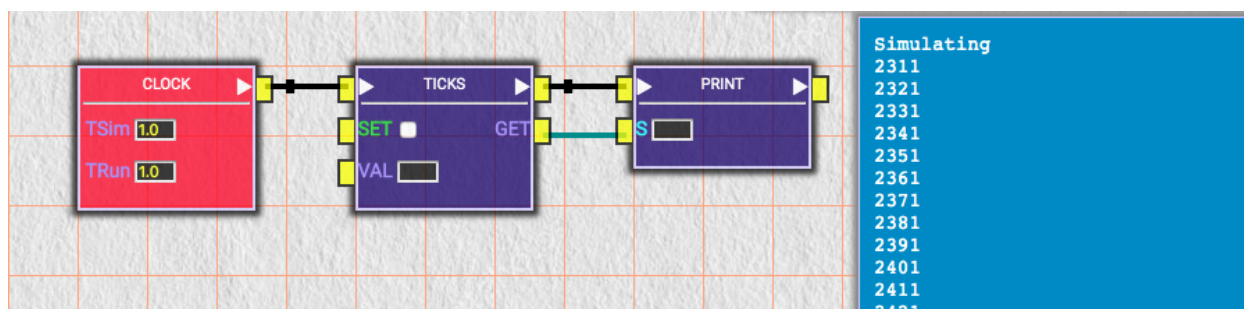
This block gives you the number of “ticks” the system clock has counted since simulation was started.

Ticks are incremented every 100ms (0.1s), so every second the system clock counts 10 ticks.

In ½ a second, the system will count 5 ticks, and so on.

This can be used to measure time between changes in the value of a sensor, or to control devices whose operation is based in time, instead of receiving digital values.

The following diagram prints the current system ticks every second.



You probably noticed that TICKS also has a couple of inputs, “SET” and “VAL”.

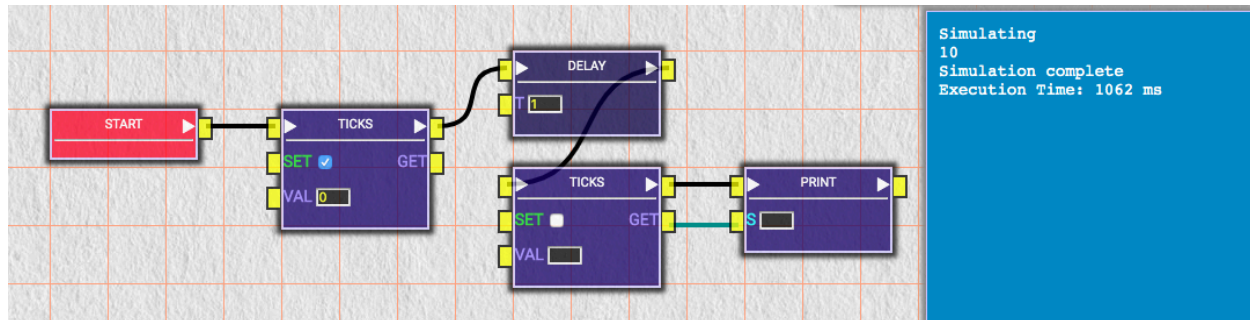
These can be used to set the ticks count.

Sometimes you want to count ticks, but want to start from a predefined value - like 0, for instance.

By ticking the “SET” input - or by connecting it to an output of another block which is outputting a “1” (that is, any non-zero value), the ticks counter is set to the value specified in its VAL input.



The following example “resets” the system tick counter, pauses execution for 2 seconds, then prints the number of ticks elapsed - since we have reset TICKS, the final tick count is the number of ticks elapsed during the execution of the DELAY block.



IMPORTANT!

In the previous diagram, you probably noticed that we are using two TICKS blocks.

That's because whenever a TICKS block is used, it gives you the number of system ticks at THAT time in the execution of your logic diagram.

The first TICKS is used to set system ticks to 0 - so, if we used its GET output, instead of using a new TICKS block, we would get the number of ticks as it was when the first TICKS was executed; in this case, a 0 (because we are resetting it!)

