BreadBoard Documentation

Table of Contents

text

Language

At its core, the BreadBoard simulator features microcontrollers that interact with other various components to create complex processes. Being a very low-level simulation environment, these microcontrollers must be programmed with a compact assembly language unique to the BreadBoard simulator, though many features are shared with other assembly languages.

It is important to note when developing BreadBoard programs that the code is run via a built-in interpreter, and thus exceptions will only be thrown when they are encountered. Just because a simulation begins successfully does not mean there are no lexical or syntactic errors in any of the programs.

BreadBoard programs are run in parallel line by line, though at a vastly delayed speed than the computer is capable of. Every “tick” of the simulation executes one and only one line from each microcontroller in the simulation. Thus, it is important to understand a few basic principles of parallel processing if using more than one microcontroller, especially avoiding race conditions and volatile memory sharing.

When a BreadBoard simulation begins, all user code is copied and processed into a more efficient format, thus increasing execution speed for the duration of the simulation. This process will be referred to as “compiling” though it is not strictly-speaking accurate.

The assembly language used in the BreadBoard simulator is case sensitive (e.g. “not” is a valid command, “NOT” is not).

Leading and trailing white space (including spaces, tabs, and carriage returns) and empty lines are ignored when compiling a BreadBoard program. Additionally, internal white space between arguments to BreadBoard commands are ignored in the compiled format, though at one piece of white space must be present between arguments to separate them (e.g. “add 5” and “add 5” are valid and identical commands, but “add5” is not valid).

When specifying literal values, you may simply specify the number in base 10 for default numeric parsing (e.g. “47” will have a decimal value of 47). However, the BreadBoard interpreter also allows you to specify numeric literals in other bases by appending a suffix to the expression (e.g. “101b” has a decimal value of 3).

|  |  |
| --- | --- |
| b | Specifies a binary literal |
| o | Specifies an octal literal |
| d | Specifies a decimal literal (default functionality) |
| x | Specifies a hexadecimal literal |

As an additional abstraction, you may specify a standard Unicode character wrapped in single quotes (‘) to specify the numerical value associated with said character (e.g. ‘A’ is equivalent to decimal 65).

To create a comment line, begin the line with the comment character (#). Comment lines are ignored during the compilation process, and as such do not impact execution whatsoever.

To create a label, specify the name of the label, and end the line with the label character (:). Labels are left out of the compiled form of BreadBoard programs, but a reference to the position of the label in the compiled program is preserved.

Labels can contain only alphanumeric characters as well as underscores, and must begin with an alphabetic character or an underscore.

When used in place of a numeric literal, labels are evaluated to be the line they point to in the compiled form of the BreadBoard program. In this way, the BreadBoard interpreter is capable of mimicking the effects of function pointers, as the line number of a jump can be moved into an internal register and jumped to at a later point.

Additionally, the line escape character (%) can be used in place of a valid numeric value, and will be evaluated to be the very next line in execution. The most direct use of this functionality is the equivalent of interrupts, wherein execution jumps to a label, then returns to the calling line.

Reference

|  |  |
| --- | --- |
| <d> | Any valid destination address (register or bus) |
| <v> | Any valid numeric value (literal or evaluated) |

Manipulation

|  |  |
| --- | --- |
| mov <v> <d> | Moves the specified value to the specified destination |
| slp <v> | Halts execution for the specified number of execution cycles |
| stop | Halts execution indefinitely |

Arithmetic

|  |  |
| --- | --- |
| add <v> | Adds the specified value to the accumulator |
| sub <v> | Subtracts the specified value from the accumulator |
| mul <v> | Multiplies the accumulator by the specified value |
| div <v> | Divides the accumulator by the specified value |
| mod <v> | Sets the value of the accumulator to the modulus of the accumulator and the specified value |
| or <v> | Sets the value of the accumulator to the bitwise or of the accumulator and the specified value |
| and <v> | Sets the value of the accumulator to the bitwise and of the accumulator and the specified value |
| xor <v> | Sets the value of the accumulator to the bitwise exclusive of the accumulator and the specified value |
| not | Sets the value of the accumulator to the bitwise inverse of the accumulator |
| bsl <v> | Performs a bitwise left shift by the specified value |
| bsr <v> | Performs a bitwise right shift by the specified value |

Control

|  |  |
| --- | --- |
| jmp <v> | Jumps execution to the specified value |
| jeq <v> <v> <v> | Jumps execution to the third value if the first value is equal to the second value |
| jne <v> <v> <v> | Jumps execution to the third value if the first value is not equal to the second value |
| jgt <v> <v> <v> | Jumps execution to the third value if the first value is greater than the second value |
| jlt <v> <v> <v> | Jumps execution to the third value if the first value is less than the second value |
| jge <v> <v> <v> | Jumps execution to the third value if the first value is greater than or equal to the second value |
| jle <v> <v> <v> | Jumps execution to the third value if the first value is less than or equal to the second value |

Components

The BreadBoard simulator offers a variety of components to combine and interact with to create complex behaviors. While these components can have very different and unique behaviors, each has at least one thing in common: connectivity.

Connectivity in the BreadBoard simulator is allowed via the orange (simple) and blue (X) bus ports on a component’s side. Each component offered in the simulator has at least one port to allow input/output operations and manipulation.

The simple bus allows for continuous reading/writing of signal values on the range of a standard unsigned 8-bit integer [0 to 255]. The values of simple bus signals are continuous: once the value of a simple bus has been set, it will continue to output that value continuously until instructed to do otherwise.

When reading simple bus signals, the highest value is returned (e.g. if two simple buses are outputting values of 75 and 65 respectively into the same simple bus cable line, a third simple bus connected to said cable line would read a value of 75).

The X bus ports allow for blocking reading/writing of data packet values on the range of a standard signed 64-bit integer [–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807]. Where the simple bus is meant to send control signals to components such as LEDs, the X bus is meant to relay data.

Read/write operations on X bus ports are blocking (e.g. if you write a value to an X bus port, execution will not resume until that value is read by another X bus port and vice versa). Because of this property, X bus data packet passing is an effective way to synchronize the processes of various microcontrollers operating in parallel, or even to create user-defined sub-processes programmed into separate chips.

It should also be noted that it is not possible to connect different types of buses together (i.e. you cannot connect an X bus cable line to a simple bus). Additionally, though you may connect more than 2 X bus ports together with the same cable line, caution should be taken, as this may result in unpredictable behavior, such as one port reading all the incoming packets, effectively preventing any other ports from receiving a packet and unblocking.

Microcontroller

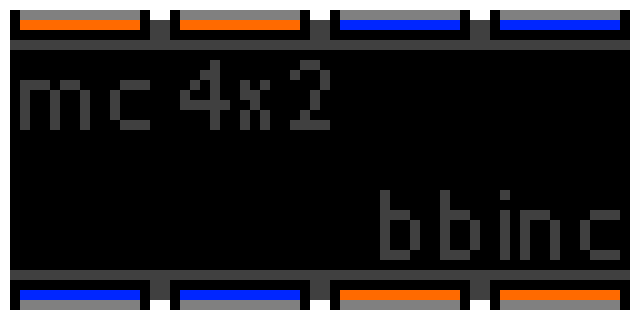
Microcontrollers are at the core of the BreadBoard simulator, as they are the primary means of creating complex user-defined component interaction.

Microcontrollers work with data in the form of 64-bit signed integers, thus allowing for complete coverage of the X bus data packet range. Additionally, all microcontrollers have a number of registers, which act as internal memory for performing operations and calculations.

All microcontrollers contain an accumulator register with the address “acc”. This register acts as the destination to which all mathematical and logical operations are stored after being calculated by the chips’ onboard ALU (Arithmetic Logic Unit). Though many microcontrollers offer additional registers, only the accumulator register is guaranteed to be on every model of chip. It is therefore important to know your hardware specifications when designing BreadBoard programs.

Unless instructed to do otherwise, a microcontroller in the BreadBoard simulator will loop the code given to them, returning to the beginning of the program upon reaching the end.

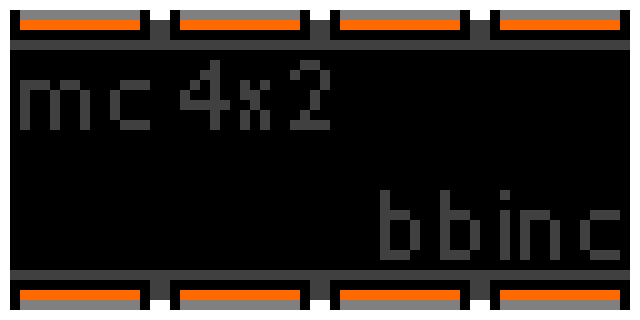
s0 s1 x0 x1



x3 x2 s3 s2

Registers: acc r0 r1 r2 r3

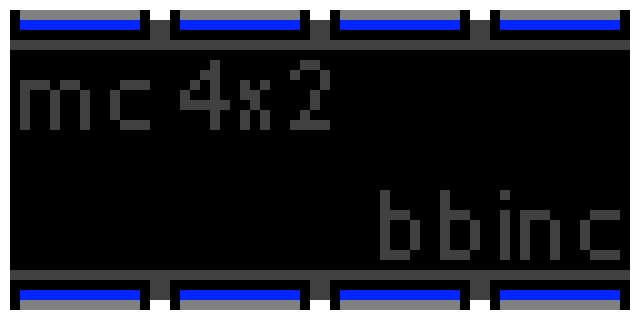
s0 s1 s2 s3



s7 s6 s5 s4

Registers: acc r0 r1 r2 r3

x0 x1 x2 x3



x7 x6 x5 x4

Registers: acc r0 r1 r2 r3

Memory

Memory (commonly referred to as RAM – Random Access Memory) provides a means of storing a large amount of information in a small area, though at the expense of not having the information directly available onboard the microcontroller processing said information (i.e. in a register).

Memory cards in the BreadBoard simulator are addressed for blocks of 8 bytes (as opposed to real memory, in which every byte is addressed). This is to simplify the reading/writing processes between microcontrollers and memory, as otherwise writing a 64-bit value to memory would require 8 separate writes, each to different memory addresses, and would require the user to disassemble and reassemble these 8-bit values to and from 64-bit values.

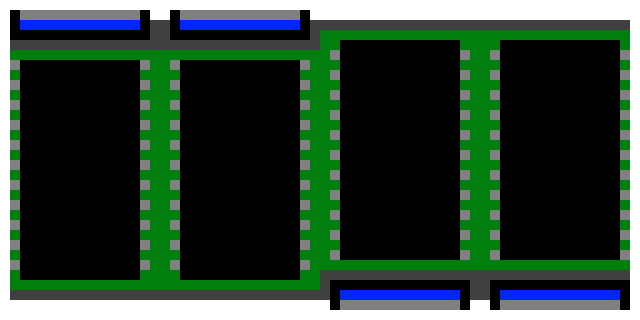
Each memory card has one or more pairs of X bus ports. In each pair, one port, designated with an “xp” is the pointer bus. The other bus in each pair is designated “xd” and is the data port.

Writing to the pointer bus will set the current address that pair’s data bus points to. Reading/writing from/to the data bus will get/set the value in memory at the address specified by its corresponding pointer bus.

In this way, any address in memory can be accessed at any time with full read/write privileges. However, this type of memory sharing can lead to data overwriting if care is not taken, as the memory cards have no existing data organization structure.

Attempting to set a pointer beyond the allowed address range (zero based) of a memory card will result in an exception, thus stopping the simulation.

xp0 xd0



xd1 xp1

Size: 1024

Numeric Display

The numeric display, quite simply, allows you to display a numeric value to the user. Of note is that the numeric display accepts only x bus input, and as such simple bus signals will been to be converted into x bus signals if you wish to display them.

The numeric display ports are write only. You cannot read the value being displayed on the screen.

A numeric display may have one or more ports. Each port on the device has the same function (i.e. reading values and displaying them). Ports on the device are designated “xi”.

Each numeric display has a certain range of values that can be displayed. Values lower than the minimum will be displayed as the minimum. Likewise, values larger than the maximum will be displayed as the maximum. Both the minimum and maximum are typically a series of 9’s sufficient to fill the display area, though this is not guaranteed to be the case.

Range: -9999999 to 9999999

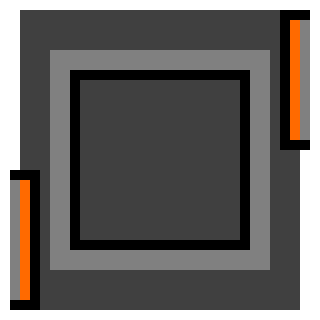
xi1

xi0



Push Button

The push button is among the simplest user input devices featured in the BreadBoard simulator. While the simulation is running, the user can click and hold the push button to cause it to output a simple bus signal of 255 for as long as the button is held. At all other times, the buses output a value of 0.

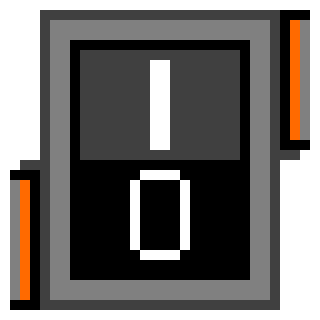


so1

so0

Toggle Switch

The toggle switch behaves much the same as the push button, but with one exception. When clicked on, the toggle switch will toggle between on and off. While on, each of its simple buses will output a signal of 255. At all other times, the buses output a value of 0.



so0

so1