

SIMSYNCH

Digital Circuit Simulator
Version 1c4

Aubrey Jaffer

This manual documents SIMSYNCH 1c4 (released August 2009), a digital logic simulator written in SCM.

Copyright © 1997, 1998, 1999, 2000, 2001, 2002, 2003 Aubrey Jaffer

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions, except that this permission notice may be stated in a translation approved by the author.

Table of Contents

1	Overview	1
2	Infrastructure	2
3	Blocks and Chips	4
4	Signals and Pins	6
5	Signal Names and Types	8
6	Macros for Signals and Pins	9
7	Models	11
8	Functional Design	12
9	Translation	16
10	Simulation Output	18
	Procedure and Macro Index	22
	Variable Index	23
	Concept and Feature Index	24

1 Overview

SIMSYNCH is a digital logic simulator. The design files are comprised of Scheme definitions and expressions. These design files can be run as a Scheme program at high speed. The design files can also be translated into formats suitable for logic compilers.

SIMSYNCH simulates blocks of synchronous logic, signals whose states change simultaneously on a clock signal transition. Each block also has a reset signal, which forces all signals to the state specified in the design file. SIMSYNCH can simultaneously simulate multiple blocks with different clocks and resets.

Blocks can contain multiple devices; Devices can contain multiple blocks.

For a list of the features that have changed since the last SIMSYNCH release, see the file 'ANNOUNCE'. For a list of the features that have changed over time, see the file 'ChangeLog'.

The author can be reached as 'agj @ alum.mit.edu'. The most recent information about SIMSYNCH can be found on SIMSYNCH's WWW home page:

<http://people.csail.mit.edu/jaffer/SIMSYNCH.html>.

2 Infrastructure

SIMSYNCH unpacks into a directory called ‘synch’. I set up design directories as peers of ‘synch’; that is, both are subdirectories of the same directory. I find it useful to put related designs in one directory. I use short design names which prefix all SIMSYNCH files which comprise that design. For instance, a design comprised of files ‘foo.scm’, ‘fooio.scm’, and ‘foojtag.scm’ having a *block* named ‘xp’ would compile into ‘foo-xp.pds’.

In order to load SIMSYNCH files, you need to create a local SLIB catalog (named ‘usercat’) within the design directory or your *HOME* directory. This catalog translates symbols to the pathnames of SIMSYNCH files.

```
;;; "usercat": SLIB catalog additions for SIMSYNCH.      --scheme--
(
  (simsynch      . "../synch/simsynch.scm")
  (run           . "../synch/run.scm")
  (models       . "../synch/models.scm")
  (logic        . "../synch/logic.scm")
  (machxl       . "../synch/scm2mach.scm")
  (verilog      . "../synch/scm2vrlg.scm")
  (vhdl         . "../synch/scm2vhdl.scm")
)
```

My design files can then load SIMSYNCH with the expression:

```
(require 'simsynch)
```

Time-stamp: *time-string* [Function]

The **Time-stamp:** procedure is named to exploit a feature of the *Emacs* text editor. The Emacs command ‘M-x **time-stamp**’ updates the string in the **Time-stamp:** expression to the current time and author.

If **Time-stamp:** is not called, the default time-stamp is #f; the default author is me.

Revision-stamp: *obj* [Function]

Configuration-stamp: *obj* [Function]

Company-stamp: *company-name-string* [Function]

The rest of these *stamps* are not supported by Emacs.

All of these information fields are used in creating the headers for logic compiler design files. This information can also be used to in creating identification fields for programmable logic devices.

```
(Time-stamp: "97/10/17 14:31:24 jaffer")
(Company-stamp: "Bipolar Technologies")
(Revision-stamp: 0)
```

create-board *board-name dbtype* [Function]

Creates an SLIB relational database of type *dbtype* as file ‘*board-name.db*’. The returned open database has empty tables for describing the pin and signal arrangements for a design.

After all calls have been made to **define-synchronous-system**, do the following to commit the database file:

```
(solidify-database *board*)
```

If the database file ‘*board-name.db*’ was not closed, solidified, or synced from a previous run, `create-board` will fail with the error:

```
; loading SIMSYNCH 1b7

;While loading "../synch/simsynch.scm", line 219:
;loaded from "uart.scm", line 24:
;ERROR: "uart.db" locked by "jaffer@aubrey.jaffer.3837:1056316195"
```

In order to proceed, delete the ‘*board-name.db*’ and/or ‘*~\$board-name.db*’ files:

```
36 Jun 22 17:09 .#uart.db -> jaffer@aubrey.jaffer.3837:1056316195
162 Jun 22 17:09 ~$uart.db
```

design [Variable]

This symbol names the design. ***design*** should be set before loading SIMSYNCH files. The design name is used in constructing filenames. The default value for ***design*** is `test`.

board [Variable]

When loaded, the file ‘`simsynch.scm`’ sets the identifier ***board*** to the database created by (`create-board *design* 'alist-table`).

`comment string1 ...` [Function]

Appends `string1 ...` to the strings given as arguments to previous calls `comment`.

`comment` [Function]

Returns the (appended) strings given as arguments to previous calls `comment` and empties the current string collection.

`#:text-till-end-of-line` [Read syntax]

Behaves as (`comment "text-till-end-of-line"`).

`synch:register-ptag`, `synch:register-block`, `synch:define-pin`, and `synch:define-signal` capture the documentation strings specified before them by ‘`#;`’ or calls to `comment`.

3 Blocks and Chips

`synch:register-block` *block-name* [Function]

`synch:register-block` *block-name clock-name* [Function]

`synch:register-block` *block-name clock-name reset-name* [Function]

The arguments *block-name*, *clock-name*, and *reset-name* are symbols. *clock-name* names the signal on which all the registered signals in *block-name* change to their next values. The default value of *clock-name* is *block-name-clock*.

reset-name names the signal which disables all the registered signals in *block-name*. Until *reset-name* deasserts, all the registered signals have their initial values. The default value of *reset-name* is *block-name-reset*.

If *reset-name* is a list with first element ‘synchronous’, then the reset state is unspecified if no clock is present during reset; and the second element of *reset-name* otherwise.

Subsequent SIMSYNCH definitions until the next `synch:register-block` will pertain to the block *block-name*.

`synch:register-ptag` *tag device-type* [Function]

`synch:register-ptag` *tag device-type signature* [Function]

Registers the symbol *tag* as a component of the string *device-type*. The optional *signature* argument associates data with *tag* for use when compiling to firmware, hardware, or microcode.

`define-synchronous-system` *block-name* ... *body* [Macro]

`define-synchronous-system` collects all the signals defined in *block-name* and defines *block-name-sim* to a procedure of one argument, the number of cycles to simulate.

When the returned procedure, *block-name-sim*, is called, it sets the queued count for block *block-name* to the argument if it is larger than that already queued. *block-name-sim* then calls `simulate!`, which runs a cycle of each non-reset block with queued cycles.

`define-synchronous-system` also defmacros *block-name* to an accessor for all registered signals in *block-name*:

`<block-name>` *registered-signal* [Macro]

Returns the value of identifier *registered-signal* in block `<block-name>` if that block is running; `eval` of *registered-signal*’s initial-value-expression if `<block-name>` is reset.

The *block-name* macro provides signal connections between blocks. In addition, the *block-name* macro provides easy access to simulation state from interactive sessions.

When the simulation is run from an interactive Scheme session, inputting any character will stop the simulation with a breakpoint which can be resumed by (`bk`).

`synch:register-ptag` and `synch:register-block` store the documentation strings specified before them by `'#;` or calls to `comment`.

4 Signals and Pins

These functions are the “pure” interface to defining signals and pins. They were created after the macro interface of the next section.

synch:insert-check *expr* [Function]

Adds the scheme expression *expr* to the body of the **block** simulator. Check expressions are evaluated once per clock cycle of their block while the block’s reset is deasserted.

Check expressions are evaluated in the order in which they are defined by **synch:insert-check**.

synch:define-pin *ptag signal pin-type pin-name output-enable* [Function]
paired-input

Associates all of the physical attributes of the symbol *signal*. Describing the signal flow of *signal*, *pin-type* is one of

- output
- input/output
- wire-and
- wire-or
- clock
- input
- unused
- internal

synch:define-signal *signal defining-ptag signal-type reset-state* [Function]
next-function

Associates the initial state and transitions for *signal*.

reset-state is the value held by this signal while *reset-name* for this block is asserted. *reset-state* should be a boolean for single signals and an integer for vectorized signals. If *reset-name* is a list with first element ‘**synchronous**’, then the reset state is unspecified if no clock is present during reset; and the second element of *reset-name* otherwise. If *reset-name* is ‘***xxxx***’, then the reset state is unspecified if no clock is present during reset; *next-function* otherwise.

The scheme expression *next-function* determines the values for *signal* when the block reset is deasserted.

signal-type is one of

registered

signal will take the value *next-function* at the next rising edge of this block’s clock signal.

registered-input

signal will take the value *next-function* at the next rising edge of this block’s clock signal. But when compiling into the hardware description language, *signal*’s input is the pin named *signal*.

combinatorial

signal follows the value *next-function* before the next rising edge of this block's clock signal. **combinatorial** is not yet implemented.

hidden

macro *signal* follows the value *next-function* before the next rising edge of this block's clock signal. **hidden** is used to specify the input side of input/output pins.

synch:define-pin and **synch:define-signal** store the documentation strings specified before them by '#' or calls **comment**.

synch:define-bus *name float-limit clause1 clause2 . . .* [Function]

Each *clause* is a (quoted) list of two expressions, an output-enable signal and a data value. That data value becomes the value of the bus if its output-enable signal is non-false.

If the last *clause* is (**#t value**), then *value* is returned whenever no other clause is active.

synch:define-bus instantiates checks whether *name* is being driven from multiple sources to different values. It also creates a check whether *name* is undriven for *float-limit* cycles. If the feature *check-turnaround* is provided when 'simsynch.scm' is loaded, **synch:define-bus** will also check the at least one cycle separates driving of *name* from different sources.

5 Signal Names and Types

Boolean (simple) signals take values of **#t** and **#f**. Boolean identifiers start with an alphabetic character and followed by characters which are alphabetic, numeric, or

`! $ % & * + - / : < = > ? @ ^ _ ~`

Vectorized signals represent multiple bits and assume integer values.

Vectorized signal names are written as boolean names followed with two integers separated by `..` or `:` and encased by square brackets (`[31..0]`). The two integers specify an inclusive range. For example, `fooCnt[0..3]` specifies a four-bit signal. The indexes can be in either order, but little-endian values must not be assigned to big-endian signals and vice versa.

The even/odd bit of vectorized values is always the lowest bit of the index pair. The designer is responsible for shifting vectorized signals to align as desired.

Aggregate signals group multiple vectorized signals into arrays.

6 Macros for Signals and Pins

This is the original form of `simsynch`. It grew over time, so it is poorly organized. But I have a lot of modules written this way.

`synch:defcheck expr` [Macro]

Adds the scheme expression *expr* to the body of the **block** simulator. Check expressions are evaluated once per clock cycle of their block while the block's reset is deasserted.

Check expressions are evaluated in the order in which they are defined by `synch:defcheck`.

`synch:pre signal` [Macro]

`synch:pre` is a macro that saves typing. `synch:pre` retrieves the *next-function* expression of *signal*. Using `synch:pre` has the same effect of inserting the second argument of the `synch:set!` specifying *signal*.

The following defmacros treat their *pin-name*, *signal*, *name* argument specially. If the symbol contains a colon (':'), the part before the colon is the ptag and the part after is the signal name. If the colon is the last character, then the name is `#f`, which is used for signals which do not connect to pins. If the part after the colon is a number, then the (pin's) name is that number.

`synch:set!`, `synch:set/reset!`, and `synch:pre` do not accept ':' encoded signal-names.

If the *pin-name* argument is ? (question-mark), this indicates that the fitter should assign a pin(s) for this signal. The assignments for bussed signals can be a range of pin names (eg. 'addr[12..0]') or a parenthesized list of pin names or numbers (eg. '(34 25 61 12)').

`synch:defmacro name expression` [Macro]

Defines *name* to be a signal of type `macro` whose value is *expression*. *name* will translate to a macro in the hardware description language which replaces *name* with *expression*.

`synch:defshare ptag:name` [Macro]

`synch:defshare pin-name ptag:name` [Macro]

Identifies *pin-name* (or *name* if single argument) of chip *ptag* with an already defined signal *name*. `synch:defshare` is used when connecting inputs in one package to outputs on another. These outputs need not originate from chips which are generated; they may also come from hidden models.

`synch:defineinput pin-name name expression` [Macro]

Defines a pin used only for input. *expression* is computed as a *hidden* macro.

`synch:defineinput pin-name name expression` [Macro]

Defines a pin used only for input, but whose signal is delayed by one clock cycle.

`synch:define pin-name signal reset-state pin output-enable` [Macro]

synch:define *pin-name signal reset-state pin output-enable next-function* [Macro]

Defines a synchronous *signal* and *pin*. **synch:define** defines all of the information **synch:define-pin** and **synch:define-signal** do. If *next-function* is absent, then it must be supplied by use of **synch:set!**.

output-enable is a boolean valued expression which controls whether the *signal* drives the pin(s). *pin* is #f or a symbol which names the signal at the pin. Note that the values of *pin* will necessarily match the values of *signal* only while *output-enable* is #t. *signal* and *pin* can be the same symbol if you have no need to access the (possibly undriven) *signal*.

```
(synch:define #f XP:tl-raw #t #f #f)
```

defines an internal signal with initial value #t.

```
(synch:define 36 XP:dloe- #t dloe- #t)
```

defines an output named *dloe-*.

```
(synch:define (34 33 32) XP:stpsz-out[2..0] #.tpsz-tag stpsz[2..0] dloe-)
(synch:defmacro HID:stpsz[2..0]
  (cond (dloe- stpsz-out[2..0])
        ((not last-dloe-) dly:-fo-tag-reg)
        (else -1)))
```

defines bussed pins *stpsz[2..0]* sometimes driven by the vectorized signal *stpsz-out[2..0]*, which is always available to the design. *HID:stpsz[2..0]* emulates the signals at the pins which can be driven from various sources.

reset-state is the value held by this signal while *reset-name* for this block is asserted. *reset-state* should be a boolean for single signals and an integer for vectorized signals. If *reset-name* is a list with first element 'synchronous', then the reset state is unspecified if no clock is present during reset; and the second element of *reset-name* otherwise. If *reset-name* is '*xxxx*', then the reset state is unspecified if no clock is present during reset; *next-function* otherwise.

synch:set! *signal next-function* [Macro]

The scheme expression *next-function* determines the values for *signal* while the block's reset is deasserted.

synch:set/reset! *signal turn-on turn-off* [Macro]

is a variant of **synch:set!** which simulates a *clocked set/reset flip-flop*. The *turn-off* expression overrides *turn-on* when both are simultaneously true; in which case a warning is also generated.

synch:declare *signal flag* [Macro]

Declares a translation-target dependent attribute *flag* for *signal* in the current block.

```
(synch:declare caa[7:0] synchronous-reset)
```

declares that signal *caa[7:0]* is synchronous reset (in the HDL translation).

7 Models

(require 'models)

'models.scm' contains some simple models of memory. These `make-` functions should be called as initial values of signals or as top-level defines. If defined at top level and used (as initial value) from two different blocks, the memory allocated is shared between the blocks.

`make-ram-array` *prototype length* . . . [Function]

Creates a RAM memory of size *length*. The ram-array memory will be of width sufficient to hold *prototype*.

`make-ram` *length* [Function]

`make-ram` *length prototype* [Function]

Creates a RAM of size *length*. If an integer *prototype* is supplied, the RAM will be of width sufficient to hold *prototype*.

`make-fifo` *length* [Function]

`make-fifo` *length prototype* [Function]

Creates a FIFO memory of size *length*. If an integer *prototype* is supplied, the FIFO memory will be of width sufficient to hold *prototype*.

`fifo:clear` *fifo* [Function]

Returns an empty *fifo*.

`synch:fifo` *name length data-source write-strobe read-strobe data-first fullness empty? full?* [Macro]

Creates a FIFO of size *length*, whose (expression) input is *data-source* enabled by (expression) *write-strobe* and emptied by (expression) *read-strobe*. `synch:fifo` defines macros or signals with names *data-first*, *fullness*, *empty?*, and *full?*.

8 Functional Design

The simultaneous assignment of signals within a block is managed by `define-synchronous-system`. The *next-function* expressions are functional as opposed to imperative; these expressions should not use `set!` or other mutators.

See [Chapter 7 \[Models\]](#), page 11 for how to create (RAM) Arrays and FIFOs.

`array-ref array index1 index2 . . .` [Function]

The procedure `array-ref` returns the contents of the `array` location selected by `index1 index2 . . .`.

`array-set array new-value index1 index2 . . .` [Function]

The procedure `array-set` returns a copy of `array` with the location selected by `index1 index2 . . .` replaced by `new-value`.

`fifo:first fifo` [Function]

Returns the first item of `fifo`. This item will be discarded by a call to `fifo:remove-first`.

`fifo:remove-first fifo` [Function]

Returns `fifo` with its first item discarded. If `fifo` is already empty, an error is signalled.

`fifo:insert-last fifo datum` [Function]

Returns `fifo` with `datum` added. If `fifo` is already full, an error is signalled.

`fifo:empty? fifo` [Function]

Returns `#t` if `fifo` is empty; otherwise, `#f`.

`fifo:full? fifo` [Function]

Returns `#t` if `fifo` is full; otherwise, `#f`.

`fifo:fullness fifo` [Function]

Returns the number of items `fifo` is currently holding.

Bitwise Operations

`logand n1 n1` [Function]

Returns the integer which is the bit-wise AND of the two integer arguments.

Example:

```
(number->string (logand #b1100 #b1010) 2)
⇒ "1000"
```

`logior n1 n2` [Function]

Returns the integer which is the bit-wise OR of the two integer arguments.

Example:

```
(number->string (logior #b1100 #b1010) 2)
⇒ "1110"
```

number-or *name k1 ...* [Function]

The arguments to **logior** must be integers. The *k1 ...* arguments to **number-or** can be either integers or **#f**. All the arguments to **number-or** are evaluated. If more than one *k1 ...* argument is an integer, a warning using name *name* is issued.

```
(number-or 'ls[2..0]
           (and cc-litx1 #.ls-data)
           (and cc-litx2 #.ls-data-x2)
           (and cc-litx3 #.ls-data-x3))
```

number-check *name k* [Function]

Issues a warning if *k* is not a number or negative. **number-check** returns **#t**.

logxor *n1 n2* [Function]

Returns the integer which is the bit-wise XOR of the two integer arguments.

Example:

```
(number->string (logxor #b1100 #b1010) 2)
⇒ "110"
```

lognot *n* [Function]

Returns the integer which is the 2s-complement of the integer argument.

Example:

```
(number->string (lognot #b10000000) 2)
⇒ "-10000001"
(number->string (lognot #b0) 2)
⇒ "-1"
```

bitwise-if *mask n0 n1* [Function]

Returns an integer composed of some bits from integer *n0* and some from integer *n1*. A bit of the result is taken from *n0* if the corresponding bit of integer *mask* is 1 and from *n1* if that bit of *mask* is 0.

logtest *j k* [Function]

```
(logtest j k) ≡ (not (zero? (logand j k)))
```

```
(logtest #b0100 #b1011) ⇒ #f
(logtest #b0100 #b0111) ⇒ #t
```

= *j k* [Function]

Returns **#t** if and only if integer *j* equals integer *k*.

zero? *j* [Function]

Returns **#t** only if integer *j* is 0.

Bit Within Word

logbit? *index j* [Function]

Returns **#t** if bit *index* of *j* is 1. Bit 0 is always the low order bit; (**logbit? 0 data[31..24]**) will be **#t** only if *data[31..24]*'s value is odd.

`copy-bit` *index from bit* [Function]
 Returns an integer the same as *from* except in the *index*th bit, which is 1 if *bit* is *#t* and 0 if *bit* is *#f*.

Example:

```
(number->string (copy-bit 0 0 #t) 2)    ⇒ "1"
(number->string (copy-bit 2 0 #t) 2)    ⇒ "100"
(number->string (copy-bit 2 #b1111 #f) 2) ⇒ "1011"
```

Fields of Bits

`bit-field` *n start end* [Function]
 Returns the integer composed of the *start* (inclusive) through *end* (exclusive) bits of *n*. The *start*th bit becomes the low order (even/odd) bit in the result. Note that the *end* index is one more than the high order bit index. Thus, `(bit-field data[7..0] 0 8)` and `(bit-field data[31..24] 0 8)` are identity functions.

This function was called `bit-extract` in SYNCH1a1.

`copy-bit-field` *to from start end* [Function]
 Returns an integer the same as *to* except possibly in the *start* (inclusive) through *end* (exclusive) bits, which are the same as those of *from*. The 0-th bit of *from* becomes the *start*th bit of the result.

Example:

```
(number->string (copy-bit-field #b1101101010 0 0 4) 2)
  ⇒ "1101100000"
(number->string (copy-bit-field #b1101101010 -1 0 4) 2)
  ⇒ "1101101111"
(number->string (copy-bit-field #b110100100010000 -1 5 9) 2)
  ⇒ "110100111110000"
```

`ash` *int count* [Function]

`arithmetic-shift` *int count* [Function]
 Returns an integer equivalent to `(inexact->exact (floor (* int (expt 2 count))))`.

Example:

```
(number->string (ash #b1 3) 2)
  ⇒ "1000"
(number->string (ash #b1010 -1) 2)
  ⇒ "101"
```

Bits shifted below the low-order bit disappear; bits shifted above the high order bit *do not* disappear. Remember to mask unused bits using `logand`.

`booleans-to-number` *bits* [Function]
bits should be booleans, taken in order they are the binary representation of the integer returned.

Example:

```
(number->string (booleans-to-number #t #f #f))  
  ⇒ "100"  
(number->string (booleans-to-number #t #t #f #t))  
  ⇒ "1101"
```

9 Translation

SIMSYNCH can translate its *register-transfer-level* designs to *MACHXL*, *Verilog*, or *VHDL* formats. The bulk of the work is performed by the files ‘*scm2mach*’, ‘*scm2vrlg*’, and ‘*scm2vhdl*’. The file ‘*run.scm*’ defines `translate`, which drives the conversion.

`translate` *design target-language* [Function]

The symbol *design* should correspond to a file named ‘*design.scm*’, which contains the definitions.

The symbol *target-language* can be:

`machxl` MACHXL 2 is a format devised by AMD for their PLDs. MACHXL 3, 4, and 5 are an unrelated commercial product. MACHXL 2 was available for free at one time.

`translate` creates a file named ‘*design.pds*’ with the MACHXL definitions.

`verilog` Verilog is a popular *High level Design Language*. Translate creates a file named ‘*design.v*’ with the Verilog definitions; and ‘*design.acf*’ with pin-name assignments. The *acf* format is used by the Altera *MAXPLUS2* Version 8 fitter. ACF is not yet supported.

`vhdl` VHDL is a popular *High level Design Language*. Translate creates files named ‘*design-tag.vhd*’ with the VHDL definitions.

SIMSYNCH translates the following functions:

- array-ref
- array-set
- ash
- arithmetic-shift
- bit-field
- bitwise-if
- copy-bit
- copy-bit-field
- logand
- logbit?
- logior
- lognot
- logtest
- logxor
- make-ram
- zero?

SIMSYNCH translates the following syntaxes. Remember that you are limited to this set only for signals which will be translated for the benefit of a logic compiler.

- and** One vectorized expression can appear as the last argument to **and**.
- or** Is recommended for booleans only.
- number-or** Use **number-or** for combining clauses which return only vectorized values or **#f**.
- if** The form is (**if** <test> <consequent> <alternate>).
- case**
- qase** **qase** is an extension of standard Scheme **case**: Each clause of a **case** statement must begin with a list of literal datums, the corresponding list in a **qase** statement may contain either literal datums or the names of symbolic constants preceded by a comma. A **qase** statement is equivalent to a **case** statement in which all symbolic constants preceded by commas have been replaced by the values of the constants. This use of comma, (or, equivalently, **unquote**) is similar to that of **quasiquote** except that the unquoted expressions must be symbolic constants.
- defconst** Symbolic constants are defined using **defconst**, their values are substituted in the head of each **qase** clause during macro expansion. In practice **defconst** constants should be defined before use.
- synch:fifo** Instantiates a synchronous FIFO memory using **make-ram**.
- synch:pre** Expands to the setter of its argument.

10 Simulation Output

`synch:info arg1 ...` [Function]

`synch:info` prints a one-line message of all `arg1 ...`.

If `arg1` is a *printf* format string, then `printf` is applied to `arg1 ...`; otherwise `arg1 ...` are printed, numbers in hexadecimal.

`synch:error arg1 ...` [Function]

`synch:error` prints an error message of all `arg1 ...` and (up to) 50 lines of the most recent timing/state for the block calling `synch:error`, and then stops the simulation.

If `arg1` is a *printf* format string, then `printf` is applied to `arg1 ...`; otherwise `arg1 ...` are printed, numbers in hexadecimal.

The simulation can be unstopped by a call to `(synch:reset)` or `(bk)`.

`synch:warn arg1 ...` [Function]

`synch:warn` prints a warning message of all `arg1 ...` and (up to) 50 lines of the most recent timing/state for the block calling `synch:warn`. *print-timing* is then enabled for at least the next 50 states.

If `arg1` is a *printf* format string, then `printf` is applied to `arg1 ...`; otherwise `arg1 ...` are printed, numbers in hexadecimal.

`check= name j k` [Function]

Will print a `synch:warn` message each cycle when `j` and `k` are not equal. `check=` returns `k`.

`print-chat` [Variable]

While set to `#t`, all *chat* messages will be printed. When set to `#f`, chat messages are not printed.

`synch:chat msg` [Function]

Queues the string `msg` for printing when `print-chat` is true or when timing/state output is generated.

`synch:count` [Variable]

Within the body of `define-synchronous-system`, the identifier `synch:count` is bound to the number of cycles which that block has completed.

This variable can be used to turn timing output on and off at specified times.

```
(cond ((eqv? synch:count trigger-start)
      (set! print-timing #t))
      ((eqv? synch:count trigger-stop)
      (set! print-timing #f)))
```

`print-timing` [Variable]

While set to `#t`, each simulation block will generate state/timing diagrams as specified by its `synch:print` command, as well as chat messages.

While `print-timing` is asserted, a line is output along with all the *chat* messages since the last line was printed. A line is printed only when if it is different from the last line printed.

Warning and *Error* messages are always printed, along with one line of state/timing diagrams.

`synch:print` *signals* . . . [Macro]

Defines the format of the mixed state and timing diagrams (such as produced by logic analyzers) from the `define-synchronous-system` in whose body it appears.

Each of the *signals* can a literal string or character, or a scheme expression. Literal characters appear on each line of the timing diagram and are used as visual separators. A literal string is used to give name to the expression which immediately follows it in *signals*. If a string does not precede an expression (such as a symbol), then the printed representation of the (unevaluated) expression serves as its name. Every 50 cycles of printed output, a header composed of the expression names is printed with the names rotated 90 degrees. Long expressions without preceding strings can make for awkward looking output.

While *print-timing* is asserted, each scheme expression is evaluated every cycle, and its value printed in the order of *signals*.

- Boolean values print as vertical bars ('|'). For `#f`, the bar is in the same column as the name; for `#t`, the bar is one column to the right from the name.
- Numeric values are printed as two-digit hexadecimal numbers.
- Characters and strings are *displayed*.
- All other values are *written*.

There are a couple of things to notice in this example of the use of `synch:print`. The character `#\-` provides a boundary. State information can be printed as two-character strings or symbols. The use of untranslatable scheme code is not a problem because `synch:print` forms are not translated into logic-compiler languages.

```
(define-synchronous-system Foo
  (synch:print
    "STATE" named-state
    "TL-X" (booleans-to-number
            #f tl-x6 tl-x5 tl-x4 tl-x3 tl-x2 tl-x1 tl-x0)
    "AMREN" pci-amren
    "RDFIFO#" pci-rdfifo-
    "RDMT" pci-rdempty
    "RDFL" pci-rdfull
    "DLOE#" dloe-
    "DLRE#" dlre-
    "DLWE#" dlwe-
    "DL-FULL" dly:-fullness
    "look-saf" look-safe
    dqoe
    hungry
```

```

"TPSZ/s" (string
  (if dloe-
    (char-upcase (tpsz-char stpsz[2..0]))
    (tpsz-char stpsz[2..0]))
  (cond (av14 #\4)
        (av13 #\3)
        (av12 #\2)
        (av11 #\1)
        (xping #\0)
        (else #\space)))
#\-
"lucy-st" lucy-state))

```

Here is some output like Foo generates. The text to the right of the ‘lucy-st’ column is produced by calls to `synch:chat`. In the first chat line, one call had the argument ‘f601f601’, while the second called with ‘runcd 1’; both appear in the same cycle. The line starting with ‘WARN’ is produced by a call to `synch:warn`.

```
96475
```

```

S T A R R R D D D D L D H T L
T L M D D D L L L L O Q U P U
A - R F M F O R W - O O N S C
T X E I T L E E E F K E G Z Y
E N F # # # U - R / -
      O L S Y S S
      # L A T
          F

```

```

e0 79 | | | | | | | 2a | | | 42 - po
e0 79 | | | | | | | 2a | | | 41 - pd f601f601 runcd 1
e0 7a | | | | | | | 29 | | | 44 - po
e0 7a | | | | | | | 29 | | | 43 - pd runcd 2
e0 7b | | | | | | | 29 | | | 42 - rn
e0 7c | | | | | | | 29 | | | 42 - po
e0 7c | | | | | | | 29 | | | 41 - pd fdfdfbf8 runcd 2
e0 7d | | | | | | | 28 | | | 44 - rn
e0 7e | | | | | | | 28 | | | 44 - po
e0 7e | | | | | | | 28 | | | 43 - pd 4bit*
e0 7f | | | | | | | 28 | | | 42 - bt
sd 40 | | | | | | | 28 | | | 42 - po
sd 40 | | | | | | | 28 | | | 22 - po
sd 40 | | | | | | | 28 | | | 22 - po dqoe
td 40 | | | | | | | 28 | | | 22 - po dqoe
td 40 | | | | | | | 28 | | | 2 - po
td 00 | | | | | | | 28 | | | - - po
td 00 | | | | | | | 28 | | | 4 - po
td 00 | | | | | | | 27 | | | ? - po
td 00 | | | | | | | 26 | | | a - po

```

```

tg 00 | | | | | | | 25 | | | A - po
tg 00 | | | | | | | 25 | | | - - po
tg 00 | | | | | | | 25 | | | t - po
tg 00 | | | | | | | 25 | | | t - po *      Z 3Y Tile
e1 00 | | | | | | | 24 | | | 10 - po
e1 00 | | | | | | | 24 | | | 10 - po
e1 00 | | | | | | | 24 | | | 10 - po ffffff9e
e1 00 | | | | | | | 23 | | | 41 - po f0f6fff0
e1 00 | | | | | | | 22 | | | 44 - pd 2xlit 31
e1 01 | | | | | | | 22 | | | 43 - x1
WARN: >>>> First Underrun Occured Here <<<<
e1 02 | | | | | | | 22 | | | 43 - x2
e1 03 | | | | | | | 22 | | | 42 - x1
e1 04 | | | | | | | 22 | | | 42 - x2
e1 05 | | | | | | | 22 | | | 41 - x1
e1 06 | | | | | | | 22 | | | 41 - x2
e1 07 | | | | | | | 22 | | | 40 - x1 fff0f6ff
e1 08 | | | | | | | 21 | | | 44 - x2
e1 09 | | | | | | | 21 | | | 43 - x1
e1 0a | | | | | | | 21 | | | 43 - x2
e1 0b | | | | | | | 21 | | | 42 - x1
e1 0c | | | | | | | 21 | | | 42 - x2
e1 0d | | | | | | | 21 | | | 41 - x1
e1 0e | | | | | | | 21 | | | 41 - x2
e1 0f | | | | | | | 21 | | | 40 - x1 f6fff0f6
e1 10 | | | | | | | 20 | | | 44 - x2
e1 11 | | | | | | | 20 | | | 43 - x1
e1 12 | | | | | | | 20 | | | 43 - x2
e1 13 | | | | | | | 20 | | | 42 - x1
e1 14 | | | | | | | 20 | | | 42 - x2
e1 15 | | | | | | | 20 | | | 41 - x1

```


Variable Index

*			
<code>*board*</code>	3		
<code>*design*</code>	3		
P			
<code>print-chat</code>	18		
		S	
		<code>print-timing</code>	18
		<code>synch:count</code>	18

Concept and Feature Index

B

block 2

E

Emacs 2

H

HOME..... 2

S

SLIB 2

stamp 2

synch 2

T

time-stamp 2