WAL Cheat Sheet

Essential Operations

Function	Syntax	Semantic
load	`(load file id)`	Load waveform from *file* as *id*
step	`(step id amount)`	Step trace *id* by *amount* (arguments are optional, default $id = all, amount = 1$)
time information	`INDEX`	Returns the current time index (starting at 0, 1, ...)
	`TS`	Returns the current time as specified in the waveform Depends on when data was dumped during simulation. Not necesarilly continous.
signal information	`SIGNALS`	Returns a list of all signals in the waveform
scope information	`SCOPES`	Returns a list of all scopes
signal renaming	`(alias name signal)`	Introduces an alias name for `signal`
signal renaming	`(unalias name)`	Removes alias `name`
signal, variable access	`symbol`	If symbol is a signal return signal value at current index If symbol is a bound variable return value of variable otherwise bind symbol to 0 and return 0
signal bit access	`(slice expr index)`	Evaluates *expr* and returns the bit at position *index*
signal slicing	`(slice expr upper lower)`	Evaluates *expr* and returns the slice of bits specified by the *upper* and *lower* arguments
relative evaluation	`(reval expr offset)`	evaluate *expr* at current index + *offset*

Advanced Operations

group detection	`(group` p_0 p_1 ... p_m`)`	returns a list of all partial signal names g for which $g + p_n$ is an existing signal
group capturing	`(in-group g expr)`	Captures the group g and then evaluates *expr*.
	`(in-groups g expr)`	Captures all groups from the list g and then evaluates *expr* for each of these groups.
group resolution	`(resolve-group p)`	Takes the current group g and appends p. If $g + p$ is an existing signal return value
scope capturing	`(in-scope s expr)`	Captures the scope s and then evaluates *expr*.
	`(in-scopes s expr)`	Captures all scopes from the list s and then evaluates *expr* for each of these scope.
scope resolution	`(resolve-scope p)`	Takes the current scope s and appends p. If $s + p$ is an existing signal return value
current group	`CG`	Returns the currently captured group
current cope	`CS`	Returns the currently captured scope
scope information	`SCOPES`	Returns a list of all scopes
Searching	`(find cond)`	Returns a list of all time indices where `cond` evaluates to true
Conditional Stepped-Evaluation	`(whenever cond expr)`	Steps through the waveform and executes `expr` when `cond` evaluates to true

Shorthand Syntax		Transformed Into
relative evaluation	`expr@sint`	`(reval expr sint)`
relative evaluation list	`expr@(`$sint_0$ ... $sint_n$`)`	`expr@`$sint_0$ ... `expr@`$sint_n$
scope resolution	`~symbol`	`(resolve-scope symbol)`
group resolution	`#symbol`	`(resolve-group symbol)`
bit extraction	`expr[int]`	`(slice expr int)`
slice extraction	`expr[`int_0`:`int_1`]`	`(slice expr` int_0 int_1`)`
expression quoting	`üexpr`	`(quote expr)`

Arithmetic and Comparison Operations

Syntax	Semantic
`(+ arg1 arg2 ... argn)`	Add arguments 1 to n
`(- arg1 arg2 ... argn)`	Subtract arguments 1 to n
`(* arg1 arg2 ... argn)`	Multiply arguments 1 to n
`(/ arg1 arg2)`	Divide arg_1 by arg_2
`(** arg1 arg2)`	$arg_1^{arg_2}$
`(! arg1)`	Negate arg_1
`(= arg1 arg2 ... argn)`	Test if arguments 1 to n are equal
`(!= arg1 arg2 ... argn)`	Test if arguments 1 to n are unequal
`(< arg1 arg2)`	Test if arg_1 is smaller than arg_2
`(<= arg1 arg2)`	Test if arg_1 is smaller or equal than arg_2
`(> arg1 arg2)`	Test if arg_1 is larger or equal than arg_2
`(>= arg1 arg2)`	Test if arg_1 is larger or equal than arg_2
`(&& arg1 arg2 ... argn)`	Logical-and arguments 1 to n
`(|| arg1 arg2 ... argn)`	Logical-or arguments 1 to n

General Purpose Operations

Syntax	Semantic
`(set (sym value)+`	Bind symbol `sym` to `value` globally for all tuples in the list
`(let (sym value)+ expr`	Bind symbol `sym` to `value` locally for all tuples in the list and evaluate `expr`
`(let (sym value)+ expr`	Like `let` but bindings can reference previous bindings
`(if cond expr1 expr2)`	If `cond` evaluates to true evaluate `expr1` else evaluate `expr2`
`(when cond expr)`	Evaluate `expr` if `cond` evaluates to true
`(unless cond expr)`	Evaluate `expr` if `cond` evaluates to false
`(cond (cond expr)+)`	For all `(cond expr)` tuples in the list. Evaluate first `expr` for which corresponding `cond` evaluates to true
`(case cond (value expr)+)`	Evaluate first `expr` for which corresponding `value` mathces result of evaluating `cond`
`(while cond expr)`	Evaluate `expr` until `cond` evaluates to false
`(do` $expr_1$ $expr_2$ ... $expr_n$`)`	Evaluate $expr_1$ to $expr_n$ and return result of $expr_n$
`(quote expr)`	Quote expression and return `expr` unevaluated.