CodeSqueezer Floating

ご紹介と使用の手引

シグナル・プロセス・ロジック株式会社

CodeSqueezer Floating の機能と特徴

複数の演算機能からなる論理を自動作成するIPツールです

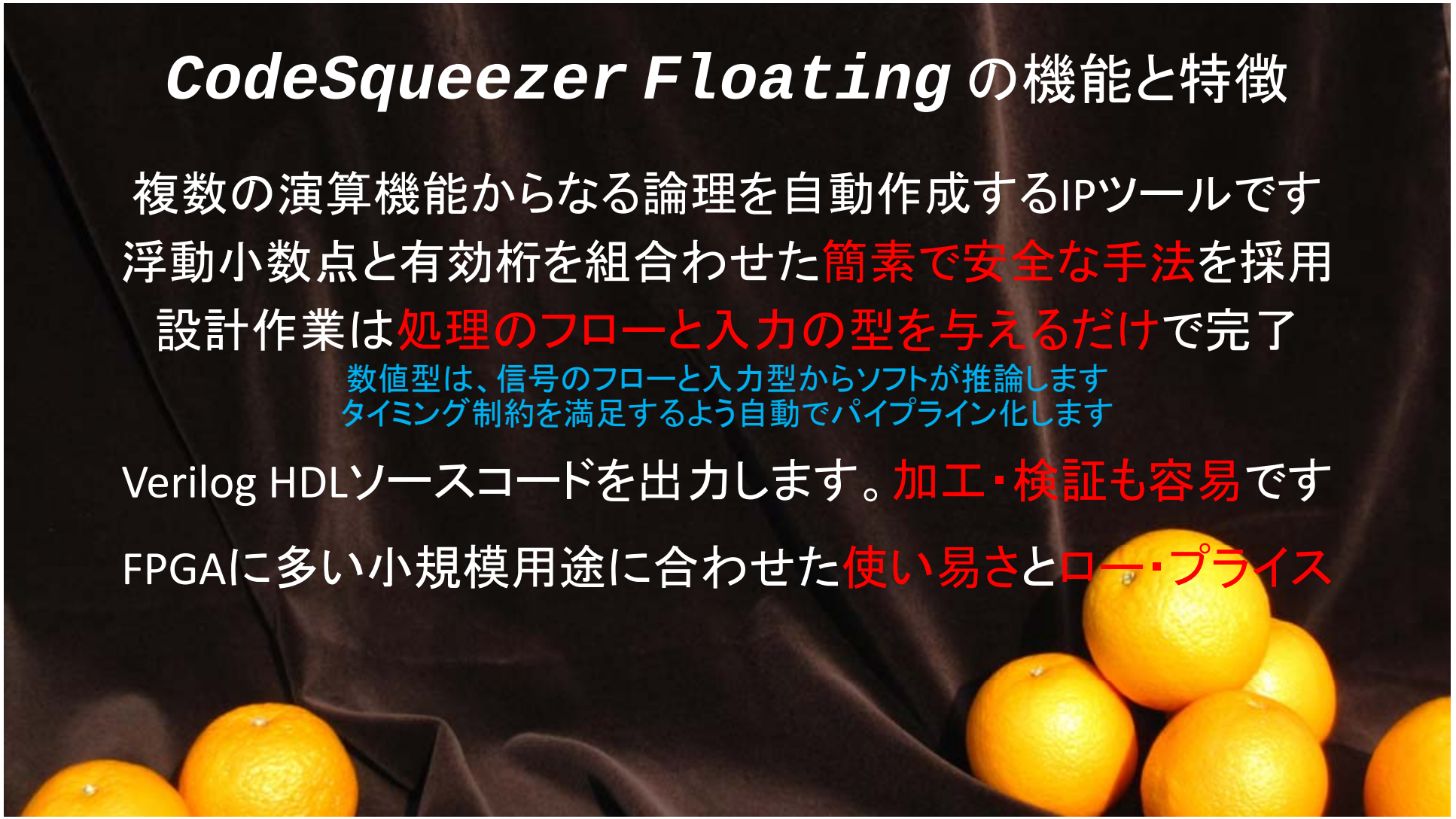
浮動小数点と有効桁を組合わせた簡素で安全な手法を採用

設計作業は処理のフローと入力の型を与えるだけで完了

数値型は、信号のフローと入力型からソフトが推論します
タイミング制約を満足するよう自動でパイプライン化します

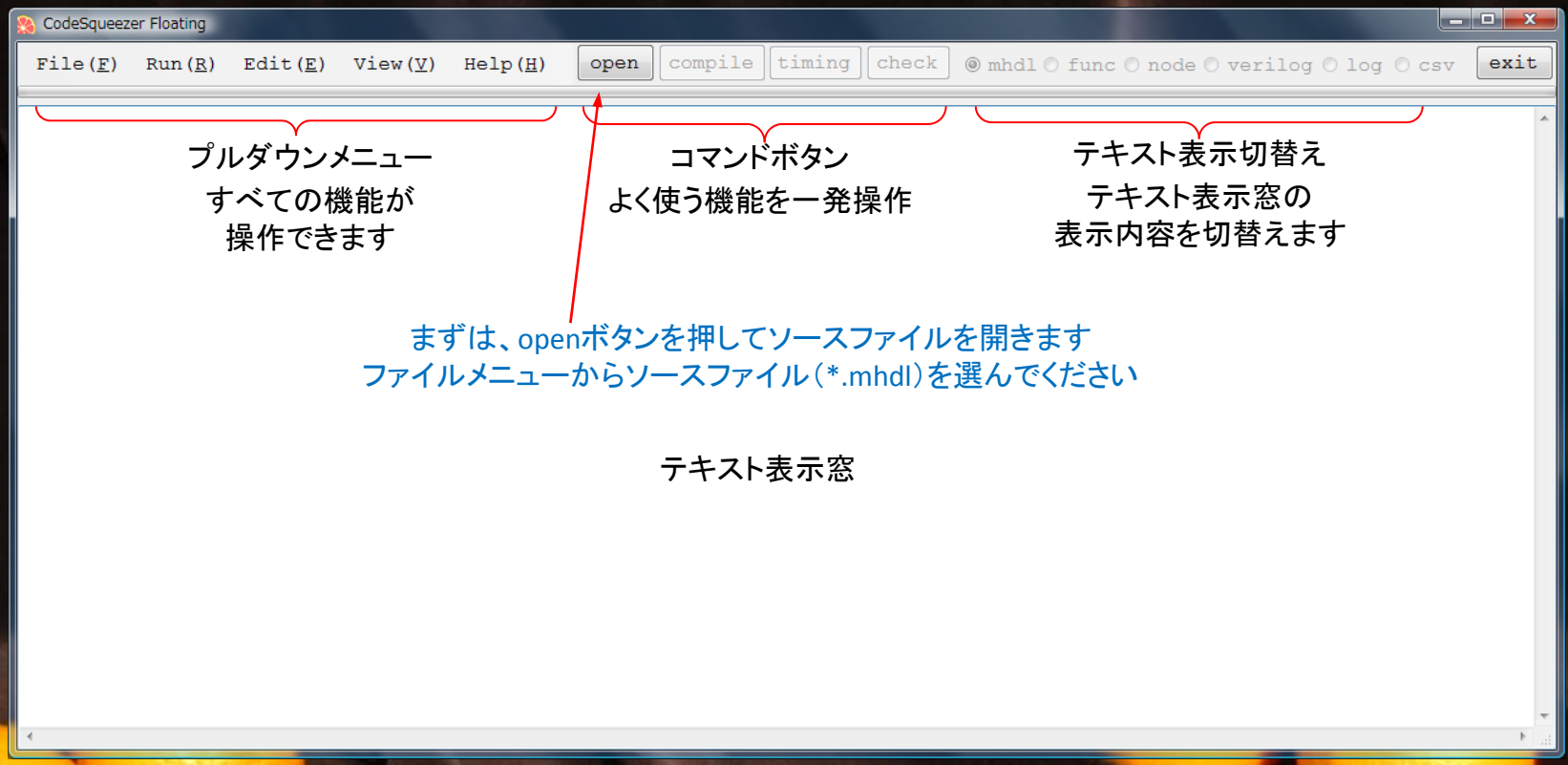
Verilog HDLソースコードを出力します。加工・検証も容易です

FPGAに多い小規模用途に合わせた使い易さとロー・プライス



起動画面

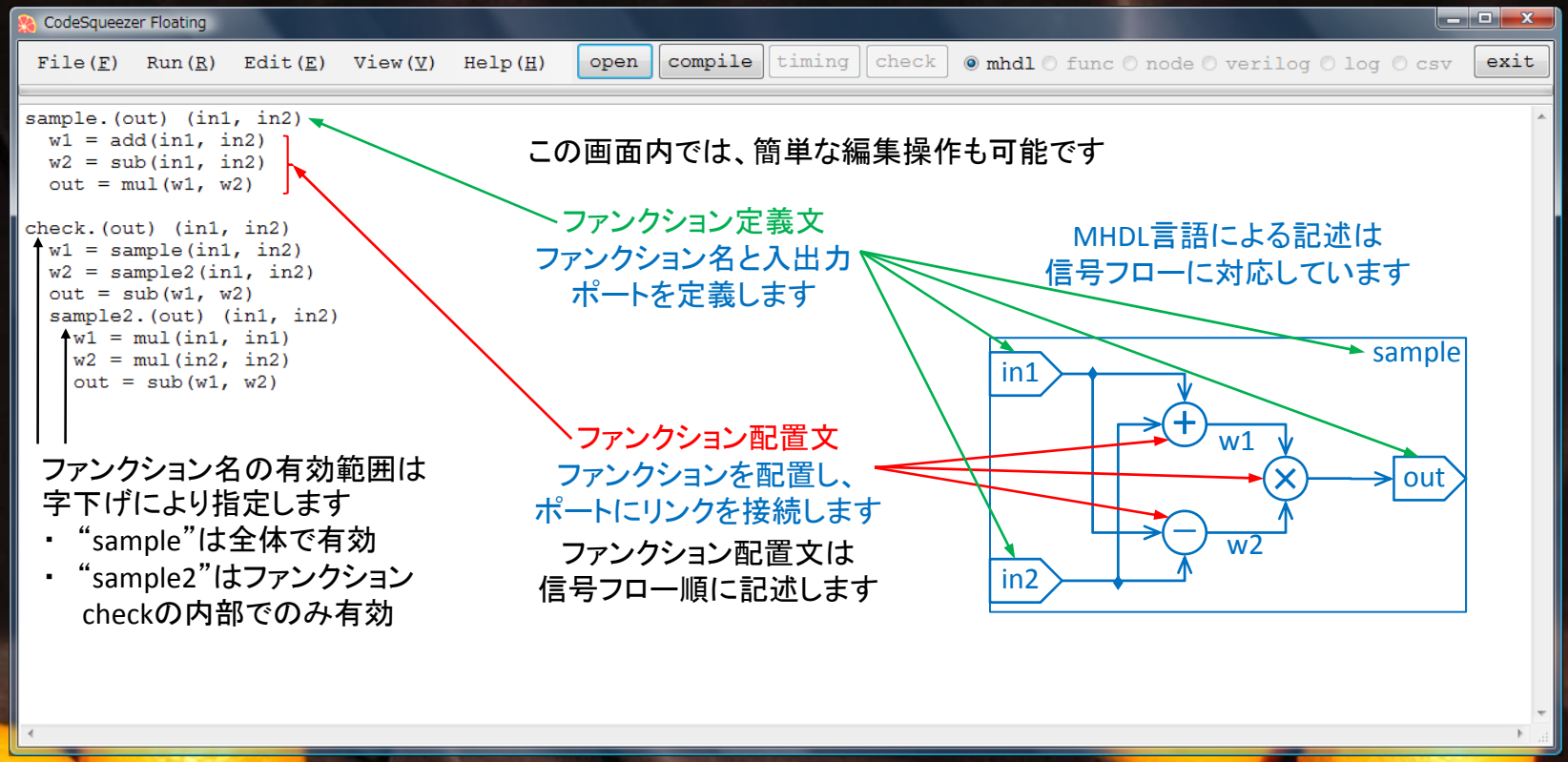
アイコン  をダブルクリックして起動します



ソースファイルの表示

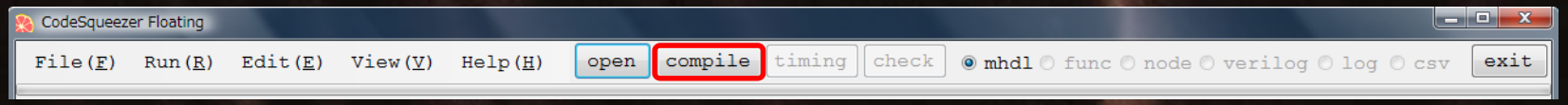
ソースファイルは MHDL 言語形式*で記述します。言語の概要は下記の通りです

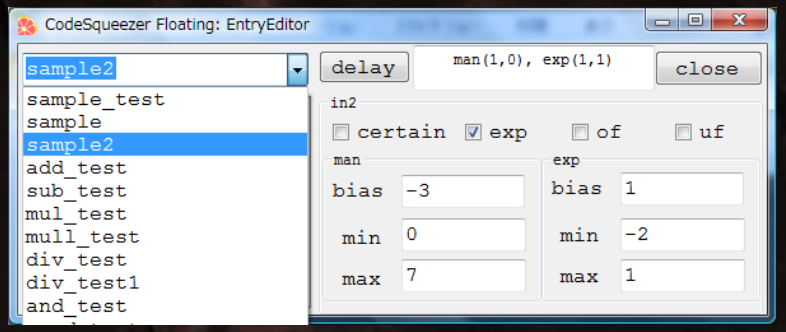
＊：言語の詳細につきましては本文書の後半をご参照ください



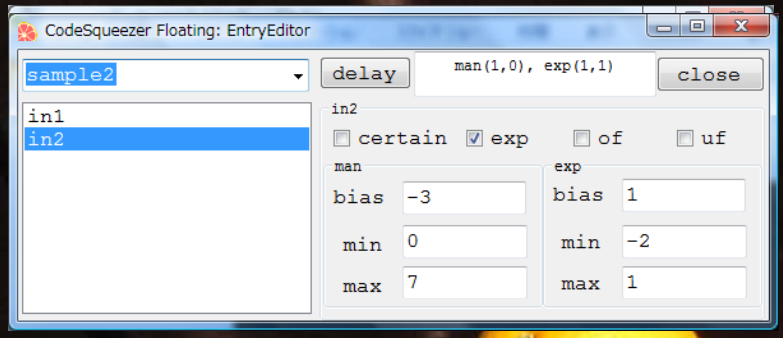
コンパイル

compileボタンを押すとコンパイル条件を設定するウィンドウが開きます





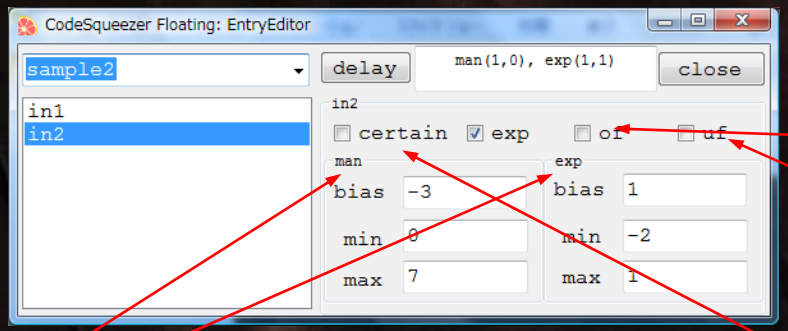
まず、コンパイルする最上位ファンクションを指定します
次に、入力ポートごとに数値型を指定します（右図）



指数部がある場合は、
仮数部と指数部の各々
について型を指定します

数値の型と信号の値

信号の値は　man × 2 ^{exp} で表されます（man: 仮数部、exp: 指数部）



異常状態は二つのフラグ（of, uf）で表示します

チェックされたフラグに対して信号線を形成します

オーバーフローフラグは、値が最大値以上であることを示します

アンダーフローフラグは、値が最小値以下であることを示します

双方のフラグが立っている場合は数値化不能を示します

いずれかの異常フラグが立っている場合、信号値は無効です

man, expはいずれも整数で、負数は2の補数で表します

整数型（ビット幅と符号の有無）は最小値と最大値で規定します

この値に定数値（bias）を加算した値が man, exp の値となります

min = max = 0 なら定数、指数部が定数なら固定小数点数です

manとexpがともに定数の場合、信号線は形成されません

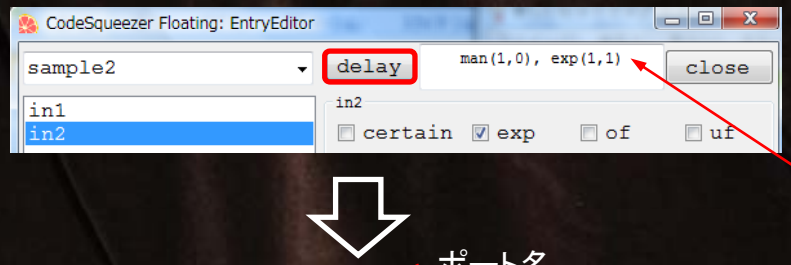
certain フラグは、誤差のない数値であることを示します
（この情報は型の決定に使用します）

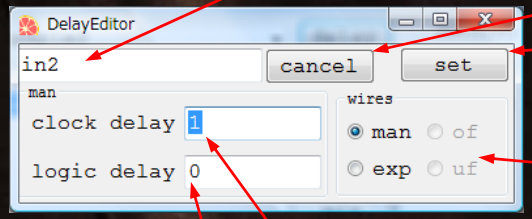
close ボタンを押すとコンパイルが始まります

入力信号線の遅延条件*

各信号線の遅延条件が上部の窓に表示されます

delay ボタンを押すと遅延設定ウィンドウ（下図）が開きます





*) 入力信号線の遅延条件はタイミング制約を解決する際に使用されます

信号線名（クロック遅れ, クロックに対する論理遅延時間）, ...

ポート名

変更を取り消してウィンドウを閉じます

表示内容を設定してウィンドウを閉じます

信号線切り替えラジオボタン

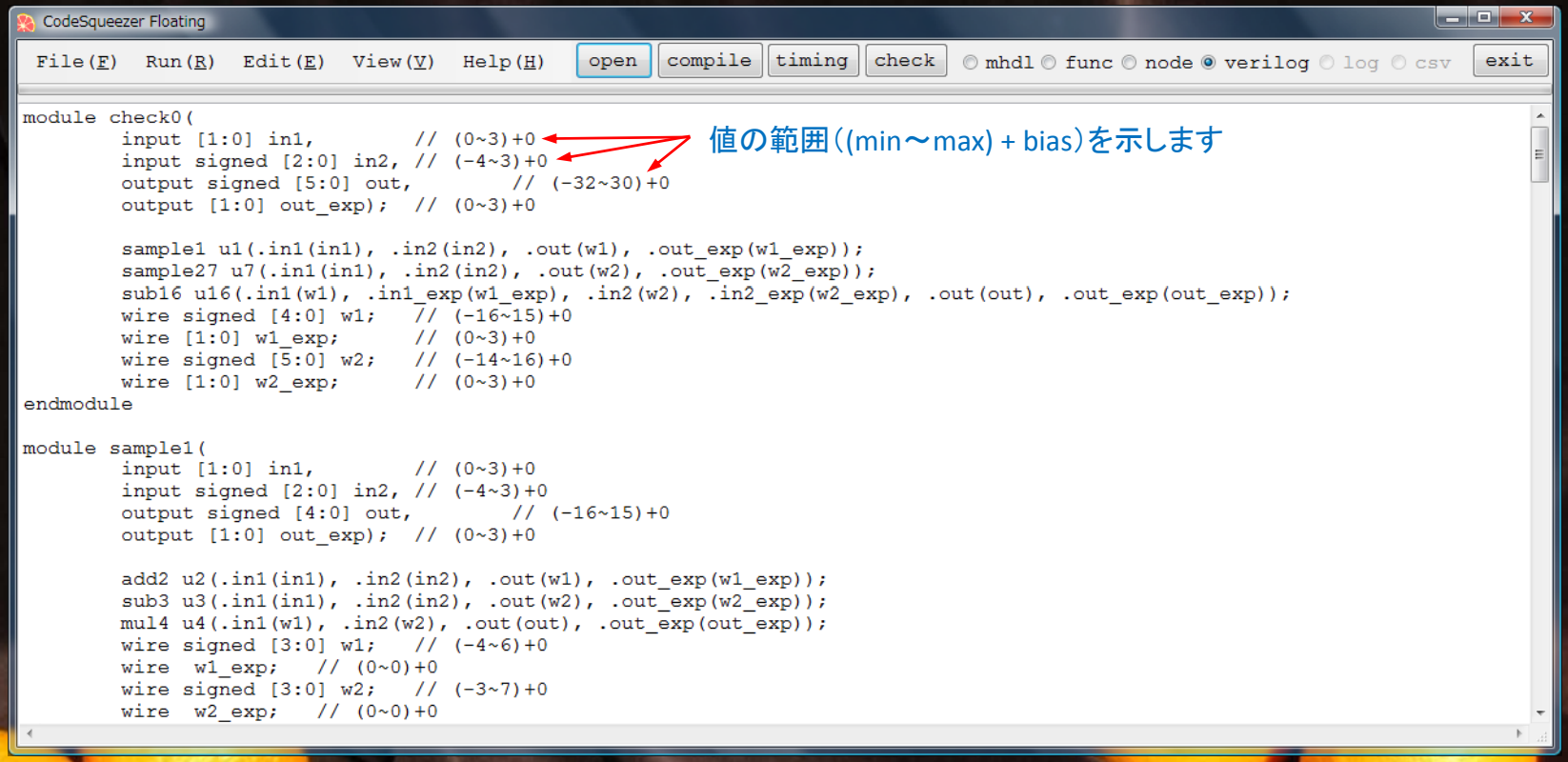
クロック遅れ（クロック数、相対値）

論理遅延時間（クロックに対する遅れ、任意単位）

変換結果：非同期コード

変換が完了すると Verilog HDL のソースコード（非同期コード*）が表示されます

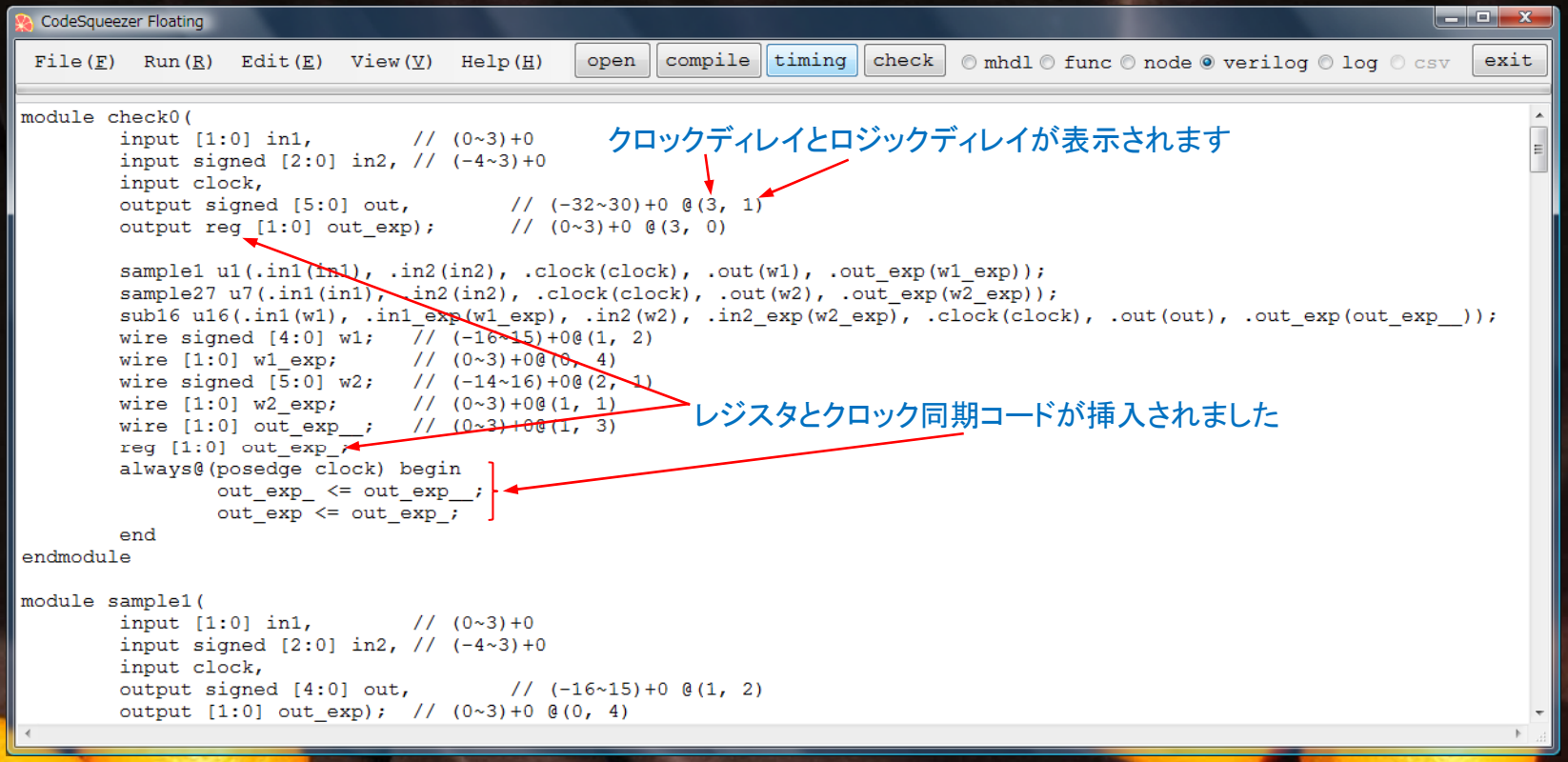
＊：非同期コードとは、レジスタを含まないコードで、入力の変化がそのまま出力に現れます




```
module check0(
        input [1:0] in1,         // (0~3)+0
        input signed [2:0] in2, // (-4~3)+0
        output signed [5:0] out,         // (-32~30)+0
        output [1:0] out_exp);  // (0~3)+0

        sample1 u1(.in1(in1), .in2(in2), .out(w1), .out_exp(w1_exp));
        sample27 u7(.in1(in1), .in2(in2), .out(w2), .out_exp(w2_exp));
        sub16 u16(.in1(w1), .in1_exp(w1_exp), .in2(w2), .in2_exp(w2_exp), .out(out), .out_exp(out_exp));
        wire signed [4:0] w1;    // (-16~15)+0
        wire [1:0] w1_exp;       // (0~3)+0
        wire signed [5:0] w2;    // (-14~16)+0
        wire [1:0] w2_exp;       // (0~3)+0
endmodule

module sample1(
        input [1:0] in1,         // (0~3)+0
        input signed [2:0] in2, // (-4~3)+0
        output signed [4:0] out,         // (-16~15)+0
        output [1:0] out_exp);  // (0~3)+0

        add2 u2(.in1(in1), .in2(in2), .out(w1), .out_exp(w1_exp));
        sub3 u3(.in1(in1), .in2(in2), .out(w2), .out_exp(w2_exp));
        mul4 u4(.in1(w1), .in2(w2), .out(out), .out_exp(out_exp));
        wire signed [3:0] w1;    // (-4~6)+0
        wire  w1_exp;    // (0~0)+0
        wire signed [3:0] w2;    // (-3~7)+0
        wire  w2_exp;    // (0~0)+0
```

変換結果：同期コード

timing ボタンを押すと、タイミング制約を満足するようにレジスタが挿入されます

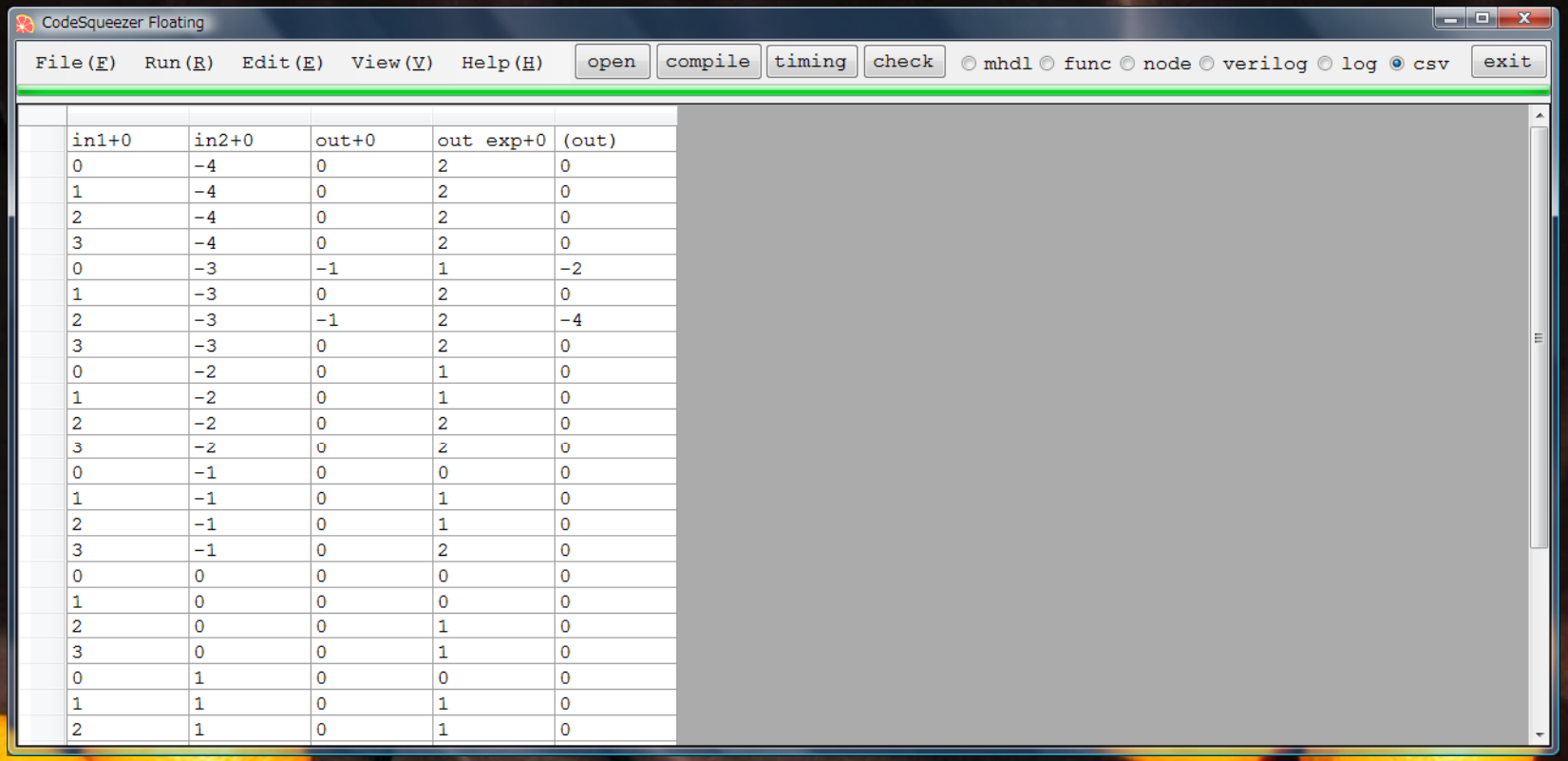



```
module check0(
        input [1:0] in1,        // (0~3)+0
        input signed [2:0] in2, // (-4~3)+0
        input clock,
        output signed [5:0] out,        // (-32~30)+0 @(3, 1)
        output reg [1:0] out_exp);      // (0~3)+0 @(3, 0)

        sample1 u1(.in1(in1), .in2(in2), .clock(clock), .out(w1), .out_exp(w1_exp));
        sample27 u7(.in1(in1), .in2(in2), .clock(clock), .out(w2), .out_exp(w2_exp));
        sub16 u16(.in1(w1), .in1_exp(w1_exp), .in2(w2), .in2_exp(w2_exp), .clock(clock), .out(out), .out_exp(out_exp__));
        wire signed [4:0] w1;   // (-16~15)+0@(1, 2)
        wire [1:0] w1_exp;      // (0~3)+0@(0, 4)
        wire signed [5:0] w2;   // (-14~16)+0@(2, 1)
        wire [1:0] w2_exp;      // (0~3)+0@(1, 1)
        wire [1:0] out_exp__;   // (0~3)+0@(1, 3)
        reg [1:0] out_exp_;
        always@(posedge clock) begin
                out_exp_ <= out_exp__;
                out_exp <= out_exp_;
        end
endmodule

module sample1(
        input [1:0] in1,        // (0~3)+0
        input signed [2:0] in2, // (-4~3)+0
        input clock,
        output signed [4:0] out,        // (-16~15)+0 @(1, 2)
        output [1:0] out_exp);  // (0~3)+0 @(0, 4)
```

チェッカー

check ボタンを押すと、自動形成したテストベクターを用いてコードを解釈実行します

最大約2048点となるよう間引きされます。biasを加算した値が表示されます。()を付した項は浮動小数点に変換したものです



in1+0	in2+0	out+0	out exp+0	(out)
0	-4	0	2	0
1	-4	0	2	0
2	-4	0	2	0
3	-4	0	2	0
0	-3	-1	1	-2
1	-3	0	2	0
2	-3	-1	2	-4
3	-3	0	2	0
0	-2	0	1	0
1	-2	0	1	0
2	-2	0	2	0
3	-2	0	2	0
0	-1	0	0	0
1	-1	0	1	0
2	-1	0	1	0
3	-1	0	2	0
0	0	0	0	0
1	0	0	0	0
2	0	0	1	0
3	0	0	1	0
0	1	0	0	0
1	1	0	1	0
2	1	0	1	0

Excel を用いた検証

テスト結果は .csv ファイルにも出力されます

=((B2+C2)*(B2-C2)-(B2*B2-C2*C2)-F2)/POWER(2,E2)

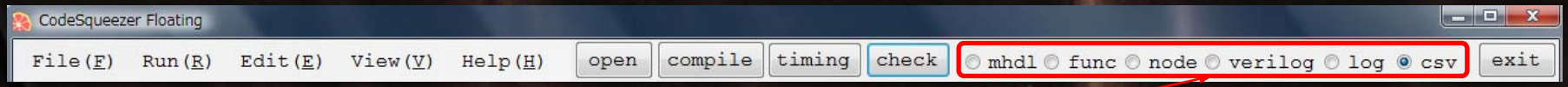
B	C	D	E	F	G
in1 +0	in2+0	out+0	out_exp+0	(out)	
0	-4	0	2	0	0
1	-4	0	2	0	0
2	-4	0	2	0	0
3	-4	0	2	0	0
0	-3	-1	1	-2	1
1	-3	0	2	0	0
2	-3	-1	2	-4	1
3	-3	0	2	0	0
0	-2	0	1	0	0
1	-2	0	1	0	0
2	-2	0	2	0	0
3	-2	0	2	0	0
0	-1	0	0	0	0
1	-1	0	1	0	0
2	-1	0	1	0	0
3	-1	0	2	0	0
0	0	0	0	0	0
1	0	0	0	0	0
2	0	0	1	0	0
3	0	0	1	0	0
0	1	0	0	0	0
1	1	0	1	0	0
2	1	0	1	0	0
3	1	0	2	0	0
0	2	0	1	0	0
1	2	0	1	0	0
2	2	0	2	0	0
3	2	0	2	0	0
0	3	-1	1	-2	1
1	3	0	2	0	0
2	3	-1	2	-4	1
3	3	0	2	0	0

Excel の演算機能を用いれば演算結果の検証も容易です

出力にクロックディレイがある場合は、タイミングの異なるデータを用いて評価します

演算誤差を power(2, out_exp) で割った結果が1前後に収まれば結果は妥当です。演算段数が多い場合は、この値は多少大きくなります。

表示の選択とファイル出力



ラジオボタンで表示を選択することができます

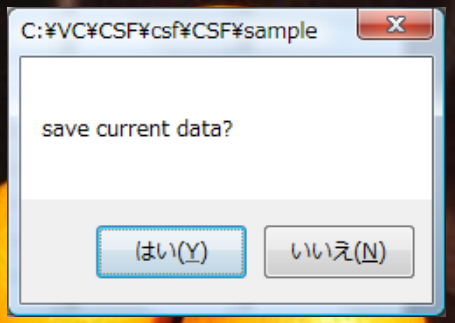
表示内容とファイルセーブの際の拡張子は下表の通りです

終了時などの操作により表示内容をファイルに出力できます

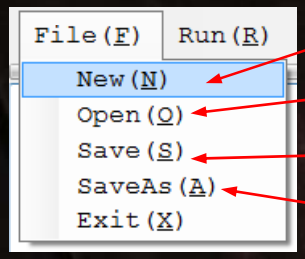
ラベル	拡張子	内容
mhdl	mhdl	MHDLソーステキスト
func	func	コンパイル対象ソースコード
node	node	コンパイル中間結果*
verilog	v	Verilog ソースコード
log	log	エラー情報
csv	csv	チェッカーの出力

＊：コンパイル・エラーの一部もここに表示されます

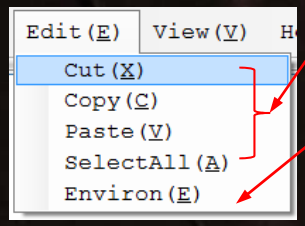
終了やファイルオープンの際、問合せウィンドウがポップアップします（下図）
「はい」を選択するとこれまでの結果がファイルに格納されます
チェック結果（csv ファイル）はチェック実行時に自動的に作成されます



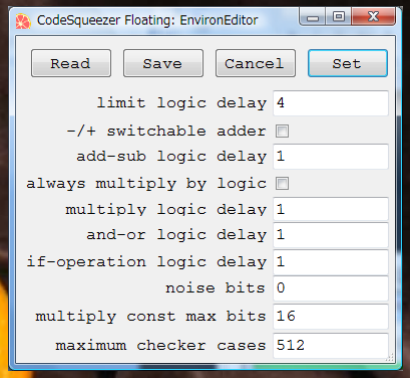
プルダウンメニュー

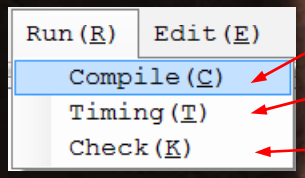


新規に仕様を作成します
既存の仕様ファイルを開きます
ファイルに上書き保存します
別のファイルに保存します

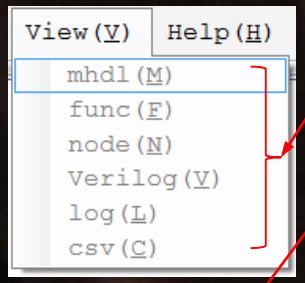


テキストの編集コマンドです
ショートカットキーも使用できます
各種処理条件を設定します
次頁参照

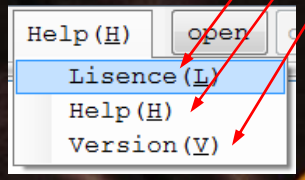




コンパイルします
タイミング制約を解決します
チェッカーを起動します



表示を切り替えます



ライセンス条件を表示します
操作方法を表示します
バージョン情報を表示します

処理条件の設定 （Edit→Environ）

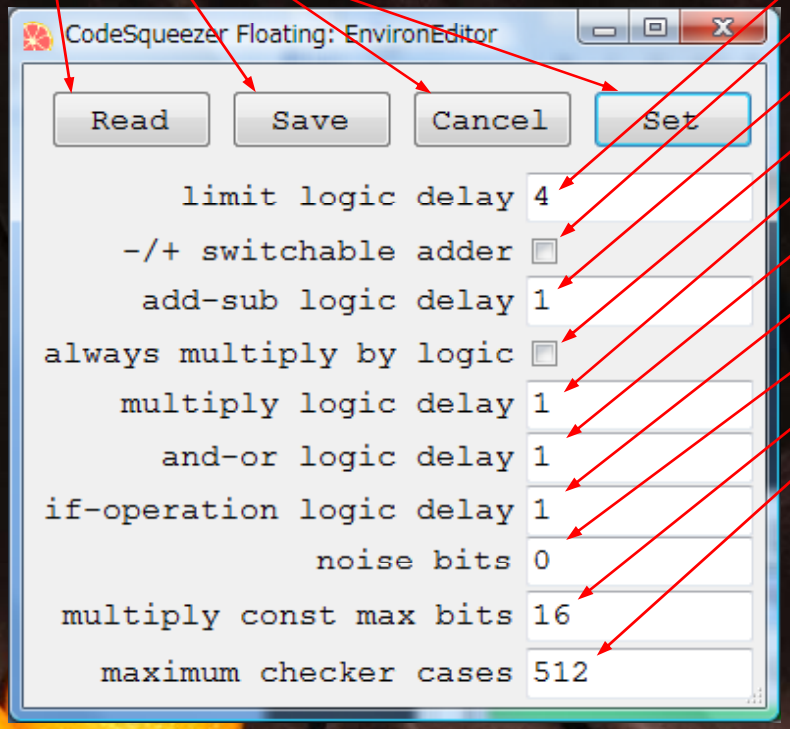
起動時にファイル“environ.env”から読み込みます

条件をファイルから読み込みます

条件をファイルにセーブします

条件を変更せずに終了します

条件を変更して終了します



最大許容論理遅延時間（タイミング制約）

加減算切り替え機能の有無

加減算の論理遅延時間

乗算論理を形成する（ハードウエア乗算器を使わない）

Verilog乗算（ハードウエア乗算器使用）の論理遅延時間

論理演算の論理遅延時間

if 式演算の論理遅延時間

ノイズとみなす下位ビット数

定数乗算最適化のブロックサイズ

チェッカーの最大テストケース数

MHDL言語仕様

- 文と行
  - 文には、ファンクション宣言文とファンクション配置文があります
  - 行末が文の区切りとなります。ただし、閉じかっこ以外の区切り記号で終わる行は次行に継続します
  - 行頭の空白の数を文のレベルと呼びます。ファンクション宣言文の範囲は、よりレベルの深い文が続く限りです
- コメント
  - “//”以降行末までの間をコメントとみなし、コンパイラはこれを無視します
  - “/*” 以降“*/”までの間もコメントとみなされます
- リンク
  - 入出力ポート間を接続する信号線で、数値型はコンパイル段階で与えられます
  - リンクの入力側は一つの出力ポートのみ、リンクの出力側は任意の数の入力ポートに接続されます
  - 入力リンクリスト中のリンクは、それ以前の行で出力ポートに接続されていなければいけません
    信号は、ソースコードの記述順に従い、上から下へと伝達されます
  - リンク名の有効範囲はファンクションの内部に限られます

MHDL言語仕様（その2）

- リンク名
  - リンク名は英字に始まる英数字の並びです
  - アンダースコア“_”は処理システム内部で使用されているため、リンク名には使用しないでください

- 定数
  - 入力リンク記述部分には、リンク名の代わりに定数を記述することもできます
  - 書式：[+|-] *man_value* [e|d|b [+|-] *exp_value*] [c]
    - *man_value* ： 仮数部数値。10進数で小数点を含めることができます
    - e|d|b ： 指数表現を示す符号。e,dは10、bは2を底とする指数表示であることを表します
    - *exp_value* ： 指数部数値。10進数で、整数に限ります
    - c ： 最後にこれを付加した場合、定数は誤差を含まないことを示します
- リンクリスト
  - 書式：(*link_name, ...*)
  - 入出力ポートが複数ある場合、リンクリストの形式でポート名や接続されるリンク名を記述します
  - 名前が一つの場合、出力ポート側の“()”は省略可能ですが、入力ポート側の“()”は省略できません

MHDL言語仕様（その3）

- ファンクション宣言文
  - 書式：***fanction_name*.*outport_list* (*inport_list*)**
    fanction_name：宣言されるファンクション名
    outport_list：このファンクションが備える出力ポートのリスト。ピリオッド（“.”）に続けて記述します
    出力ポートリストがない場合（ピリオッドのない場合）はファンクション名と同じ出力ポートが定義されます
    inport_list：このファンクションが備える入力ポートのリスト。“( )”内に記述します。
    入出力ポートには、ポートと同名のリンクが自動的に接続されます
  - ファンクション宣言文に続けて、このファンクションに含まれるファンクション配置文とファンクション宣言文を記述します
  - ファンクションの内部で宣言されたファンクション名は、宣言を行ったファンクションの内部でのみ通用します
  - 他のファンクションに含まれないファンクションのみがコンパイルの際の最上位ファンクションとなり得ます

- ファンクション配置文
  - 書式： ***outlink_list = fanction_name(inlink_list)***
    outlink_list：配置されるファンクションの出力ポートに接続されるリンクのリスト
    fanction_name：配置されるファンクション名（宣言されたファンクションまたは標準ファンクション）
    inlink_list：配置されるファンクションの入力ポートに接続されるリンクのリスト

標準ファンクション

- 四則演算標準ファンクション
  - いずれの演算も、入力に誤差が含まれる場合は有効桁のみを演算して後段に伝達します
  - add(in1, in2) ： in1 + in2 を演算します
  - sub(in1, in2) ： in1 - in2を演算します
  - mul(in1, in2)／mull(in1, in2) : in1 ×in2を演算します
    mulを用いた場合、乗算論理の形成はVerilogに委ねられます
    mullを用いた場合は、本システムが乗算論理を作成します
  - div(in1, in2) : in1 ÷ in2を演算し商を返します
    双方誤差なしの場合は仮数部の除算を行い、商と余りを返します
    in2が0の場合は数値化不能異常となります
- 論理演算標準ファンクション
  - すべての論理演算では、入力される信号が0でない場合を真、0または検出限界以下の場合を偽として扱います
  - 論理演算の出力は真で1、偽で0をとる誤差なし整数で、定数となる場合と変数となる場合があります
  - 以下の論理演算は論理積、論理和、排他論理和およびこれらの否定を返します
    and(in1, in2)／nand(in1, in2)／or(in1, in2)／nor(in1, in2)／exor(in1, in2)／exnor(in1, in2)
  - not(in) : inの否定を返します

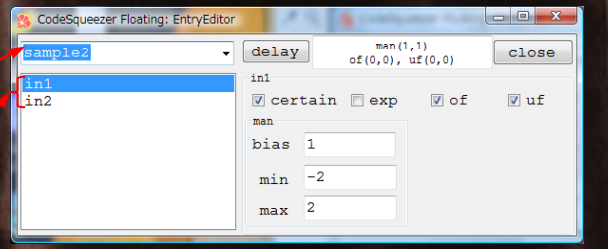
標準ファンクション（その2）

- スイッチ（マルチプレクサ）
  - if(cond, in1, in2)：condが0でないときにin1、0または検出限界以下のときにin2を返します
- 信号の加工
  - reduce(in)：inの有効桁数を一つ減じたものを出力します
  - to_int(in)：inを整数に変換します。オーバーフローしない充分なビット幅がとられます
  - to_fix(in1, in2)：in1をin2の型と同じ固定小数点数に変換します。in2の指数部は定数でなければなりません
    整数のビット幅を制限する場合にも to_fix をご利用ください。オーバーフローが生じる場合はフラグがセットされます
  - clip(in)：inがオーバーフローした場合、最大値または最小値を返します。数値化不能異常の場合は値を0に固定します
    異常フラグは解除されます

標準ファンクション（その3）

- 異常の検出
  - of(in)／uf(in)／nan(in)：inがオーバーフロー／アンダーフロー／数値化不能異常であるとき1を返します
- 数値の論理化
  - positive(in)／negative(in) ：inが正／負である場合に1となります。検出限界以下の場合は0を返します
  - nonzero(in)／undetectable(in)：inが0でない場合／0または検出限界以下である場合に1となります
- 検出限界の設定
  - 検出限界は誤差（ノイズなど）とみなされるビット数（noise bits）で指定します
  - noise bitsの設定は、プルダウンメニューの Edit → Environ でポップアップする Environ Editorで行います
  - 出荷状態では noise bits は0となっています（誤差は仮数部の1未満であるとみなします）

MHDLコンパイラ指示



- #で始まる行はコンパイラに対する指示で、コンパイル・ウィンドウでの操作を代行します
- コンパイラ指示には次の3つがあります
- **#entry** ***function_name***:コンパイルする最上位ファンクションを指定します
  - 指定されたファンクションが存在しない場合は無視されます
- **#inport** ***inport_name type_option*** …:入力ポートの数値型を指定します
  - ***inport_name***:数値型を指定する入力ポート名
  - ***type_option***:数値型の指定で、次のオプションがあります。必要なものを空白で区切って並べます
    man(*minimum, maximum, bias*)／exp(*minimum, maximum, bias*):
    仮数部／指数部の数値型指定で、“( )”内にはそれぞれ最小値、最大値、バイアス値をコンマで区切って並べます
    certain:誤差を含まないことを指定します
    of／uf／nan:オーバーフロー／アンダーフロー／数値化不能の異常状態をとりえることを指定します
    man_delay(*clock_delay, logic_delay*) :入力信号線の遅延を指定します。manの他に、exp, of, ufも指定可能です
  - manは必ず指定しますが、それ以外は必要なもののみを指定します
  - #entry が指定されていない場合は #inport 指示は無効です
- **#nonstop**:コンパイルウィンドウを表示せず直ちにコンパイルを開始します
- コンパイラ指示は、コンパイル開始時点でmhdlソース・テキストから読み込まれます
  コンパイラ指示の変更はmhdlソース・テキストを編集して行います

ご利用に際して

CodeSqueezer Floating は、FPGAで信号処理を行う際などに必要となる複雑な数値演算を含む論理モジュールを簡単に作成することを狙いとしています

最終的にFPGAに組み込むモジュールには、数値演算以外に各種の制御コードが含まれるのが一般的ですが、***CodeSqueezer Floating*** は数値演算論理の作成に特化しており、制御論理を作成するための機能は含まれておりません

実用的なFPGA組込モジュールの作成にあたっては、複雑な数値演算モジュールを***CodeSqueezer Floating*** で作成し、Verilog等で作成した制御モジュールと組み合わせてご使用ください

バグ情報、最新版等につきましては弊社HPをご参照ください。

http://signal-process-logic.com

シグナル・プロセス・ロジック株式会社