Verilog-A概要

1. はじめに

1.1 Verilog-Aとは何か

Verilog-Aは、アナログモデルを定義するための言語です。これは、半導体デバイスの非常に詳細なモデルだけでなく、高い抽象レベルをもつ動作モデルを定義するのに適しています。

Verilog-Aおよび他の同様の言語（VHDL-AMSやMASTなど）の導入以前は、そのようなモデルの定義は、制御されたソース、任意のソース、およびさまざまな半導体デバイスのサブ回路を使用してのみ実現できました。この方法は柔軟性がなく、ぎこちなく、通常は非常に非効率的です。

さらに、SIMetrix Verilog-Aはコンパイラ型言語です。つまり、組み込みデバイスモデルが実装されているのと同じ方法で、Verilog-Aコードがバイナリ実行可能プログラムにコンパイルされます。これにより、Verilog-Aモデルは非常に高速です。

Verilog-AのSIMetrix実装では、コンパイラを使用して、Verilog-Aソースを'C'言語を使用するプログラムコードに変換します。これは、DLLにコンパイルされ、SIMetrixメモリイメージにロードされます。次に、モデルとインスタンス行を使用して、ネットリストレベルでVerilog-Aの記述にアクセスします。

Verilog-Aを使用するために'C'コンパイラをインストールする必要はありません。SIMetrix Verilog-Aには、MinGW拡張機能を使用したオープンソースの'C'コンパイラgccが付属しています。このために必要な必須ファイルのみを含むgccの簡易バージョンを使用しました。

SIMetrix Verilog-Aコンパイラは、当社によって開発されました。サードパーティ製品にライセンスを与えていませんし、オープンソースソフトウェアに基づくものでもありません。これは、当社がこの製品を熟知しており、すべての製品で常に提供してきたのと同じ高レベルのサポートを提供できることを意味します。

1.2 Verilog-A言語リファレンスマニュアル

SIMetrix実装は、もともとバージョン2.2 Language Reference Manualに基づいて開発され

ました。バージョン8.3から、ドキュメンテーションとコンパイラメッセージを更新して、最新バージョンであるLRM 2.4を参照するようにしました。ほとんどの領域で、LRM 2.4はLRM 2.2のスーパーセットであり、ほとんどすべての場合、LRM 2.2用に記述されたVerilog-Aモジュールは、変更を必要とせずに2.4実装で同様に機能します。

バージョン2.4 Language Reference Manualは、*Verilog-A Language Reference Manual*から入手できます。

2. Verilog-Aコンパイラの使用

2.1 SIMetrix回路図でVerilog-Aを使用する

SIMetrixには、Verilog-A定義で使用する回路図シンボルを作成するシンプルな機能があります。この機能は、コンパイルプロセスの最初の部分のみを実行するように指示するオプションを使用して、Verilog-Aコンパイラを呼び出します。これにより、スクリプトは、モジュール名やポート名などのVerilog-Aファイルに関する情報を学習できます。スクリプトは、各ピンの配置場所を尋ね、その後、シンボルを作成して回路図に配置します。シンボルは、Verilog-Aモデルをシミュレータに接続するために必要なすべてのプロパティで修飾されます。

スクリプトを使用するには、Verilog-A定義を作成した後、回路図メニューの**Verilog-A | Construct Verilog-A Symbol**を実行します。Verilog-Aファイル（拡張子.va）に移動して閉じます。要求に応じてピンの位置を選択します。シンボルの画像が配置用に表示されます。

シンボルは、**Place | From Symbol Library**を使用して、将来使用するために見つけることができます。そして、“Auto Created Symbols -> Verilog-A Symbols”に移動します。

2.2 ネットリストでのVerilog-Aファイルの定義

シミュレータ文「.LOAD」を使用して、Verilog-Aソースファイルを指定します。例えば下記のとおりです。

```
.LOAD resistor.va
```

これにより、Verilog-Aコンパイラ（va.exe）が呼び出され、1つの共通‘C’ファイルと、Verilog-Aファイル内のモジュール文ごとに1つの‘C’ファイルが作成されます。‘C’ファイルは、gccを使用してコンパイルおよびリンクされ、拡張子.sxdevを持つ最終DLLが生成されます。

これらのファイルはすべて下記のディレクトリに配置されます。
%APPDATAPATH%\SIMetrixvvv\vacache
ここで、%APPDATAPATH%はユーザのアプリケーションデータディレクトリです。（vvvは製品バージョン、たとえばバージョン8.3の場合は830）

vaファイルをコンパイルすると、.LOADは.sxdevファイルをSIMetrixメモリイメージにロードします。次に、コードをシミュレータのモデルテーブルにマッピングして、新しいデバイスをすぐに使用できるようにします。

Verilog-Aモジュール文で定義された新しいデバイスを使用するには、.MODEL文を指定する必要があります。これは、.LOAD文の*後に*配置する必要があります。.MODEL文の形式は次のとおりです。

```
.MODEL modelname va-mod-name parameters
```

*modelname*がインスタンスで参照されるモデル名である場合（下記参照）、*va-mod-name*はVerilog-Aソースファイル内のモジュールの名前で、*parameters*はVerilog-Aパラメータキーワードを使用して定義されたパラメータです。

新しいデバイスのインスタンスを作成するには、'N'、'P'、'W'、'U'、'Y'のいずれかの文字で始まるインスタンス行（または適切なプロパティを持つ回路図シンボル）を作成します。端子の数がその文字の元の使用と互換性がある限り、他の文字を使用できます。たとえば、MOSデバイスのようにデバイスに4つの端子がある限り、'M'の文字を使用できます。ただし、端子が4つより多い場合や端末が1つしかない場合は、'N'、'P'、'W'、'U'、'Y'のいずれかを使用する必要があります。'Q' の使用は避けることをお勧めします。このデバイスは3つまたは4つの端子を使用できるため、あいまいさが生じる可能性があります。

新しいシミュレーションを開始すると、前回の実行でロードされたsxdevファイルはすべてアンロードされ、モデルテーブルエントリは削除されます。

2.3 メッセージ

シミュレーションを実行すると、コマンドシェルにいくつかのメッセージが表示されます。これらは、Verilog-AコンパイラとMAKEユーティリティによって出力されます。

まれに、'C'コンパイラまたはリンカによって警告またはエラーが生成される場合があります。これらはテクニカルサポートに報告する必要があります。

この処理中に、Verilog-Aコンパイラによるエラーまたは警告出力が表示されることがあります。これは次の形式になります。

```
*** ERROR *** (@'verilog-a-filename',linenum), error-message
```

問題が構文にある場合、メッセージには*** SYNTAX ERROR ***と表示されます。

注：警告またはエラーメッセージで参照される場合、Verilog-Aコードで使用する識別子（変数、パラメータ、ポートなど）がアンダースコアとともに前に付くかもしれません。

編集せずに.VAファイルを2回目以降に実行すると、Verilog-Aコンパイラからのメッセージは表示されません。

2.4 .LOADの完全な構文

```
.LOAD file [instparams=parameter_list] [nicenames=0|1]
[goiters=goiters] [ctparams=ctparams] [suffix=suffix]
[warn=warnlevel]
```

*file*は、Verilog-Aファイルまたは.SXDEVファイルのいずれかを指定できます。拡張子が.SXDEVの場合、コンパイルは実行されず、指定されたファイルが直接ロードされます。この場合、上記の残りのオプションは認識されません。それ以外の場合は、上記のビルドシーケンスが開始されます。パスは、現在の作業ディレクトリに相対的です。**スペースを含む.VAファイル名を使用しないでください。**

*parameter_list*は、コンマで区切られたパラメータ名のリストです。このリストにはスペースを入れないでください。このリストの各パラメータは、インスタンスパラメータとして定義されます。詳細については、 “Instance Parameters”を参照してください。

*goiters*は、グローバルオプティマイザの反復回数を指定します。デフォルトは3です。数値を大きくすると、コンパイル時間が長くなりますが、コードの実行速度が向上する場合があります。実際には、これは非常に大きなVerilog-Aファイルに対してのみ顕著な効果があります。値をゼロに設定すると、グローバルオプティマイザが無効になります。これにより、実行速度が少し遅くなる可能性があります。グローバルオプティマイザは、‘C’ファイル内の冗長な文をクリーンアップするアルゴリズムです。

*ctparams*は「コンパイル時パラメータ」を定義し、*name = value*形式のパラメータ名/値のペ

アのコンマ区切りのリストです。リストされたパラメータは、Verilog-Aコードにリテラル定数として入力されたかのように、コンパイル中に定義された定数値に置き換えられます。この機能は、配列サイズやベクトル化されたポートサイズなどの項目に特に役立ちます。そのような項目の値がコンパイル時にわかっている場合、かなり効率的な結果が生成されます。

*warnlevel*は、警告メッセージのフィルタを設定します。設定をゼロにすると、警告は表示されません。設定を2にすると、すべての警告が表示されます。デフォルトは1で、ほとんどの警告が表示されますが、それほど深刻ではないものは省略されます。

nicenames=0|1は、デバッグを目的とした高度な機能です。可能であれば、'C'ファイルで意味のある名前を使用するようコンパイラに指示します。そうでない場合は、短い名前が使用されます。このオプションをオンにすると、'C'ファイルで名前が衝突するリスクが非常に小さくなります。

2.5 Verilog-Aキャッシュ

SIMetrixは、ソースファイルが変更されていない場合、再コンパイルせずに既存のVerilog-Aバイナリファイルを再利用します。ソースファイルのMD5チェックサムを計算し、これを.sxdevファイルに保存されている値と比較することにより、ファイルが変更されたかどうかを判断します。この方法は、ファイルの日付をチェックする従来の方法よりも低速ですが、よりロバストで信頼性があります。

VA定義が大きい場合、このキャッシュメカニズムにより時間を大幅に節約できます。たとえば、hicumモデルのコンパイルには約6秒かかります。

回路図メニュー**Verilog-A | Clear Cache**を使用して、いつでもキャッシュをクリアできます。これにより、キャッシュディレクトリ内のすべてのファイルが削除されます。

2.6 永続的な.SXDEVインストール

.sxdevファイルはplugins\devicesディレクトリに再配置される場合があり、その場合は組み込みデバイスになります。バージョン間のバイナリ互換性は保証されていませんが、古いSIMetrixバージョンを使用して作成された.sxdevファイルは、ほとんどの場合、新しいSIMetrixバージョンで正しく動作します。.sxdevファイルをロードするには、Verilog-Aライセンスが必要です。

3. Verilog-Aコードの作成

3.1 概要

いくつかの例を示して、Verilog-Aを紹介します。各例では、新しい概念または言語機能を紹介しています。これは言語の定義的なリファレンスではありませんが、最も一般的に使用される機能を示したいと考えています。次の表は、このマニュアルで使用されている例を、実行する回路図とVerilog-A定義ファイルを見つけることができるファイルのパスとともに示します。

Example	File Location
Hello World!	Examples/Verilog-A/Manual/Hello-world
A Simple Device Model	Examples/Verilog-A/Manual/Gain-block
A Resistor	Examples/Verilog-A/Manual/Resistor
A Soft Limiter	Examples/Verilog-A/Manual/Soft-limiter
A Capacitor	Examples/Verilog- A/Manual/Capacitor
A Voltage Controlled Oscillator	Examples/Verilog-A/Manual/Vco
Digital Gate	Examples/Verilog-A/Manual/Gates
Butterworth Filter	Examples/Verilog-A/Manual/Butterworth-filter
RC Ladder - Loops, Vectored Nodes and genvars	Examples/Verilog-A/Manual/RC-ladder
Indirect Assignments	Examples/Verilog-A/Manual/Indirect-assignment

3.2 Verilog-Aテキストエディタ

SIMetrixには、Verilog-A構文を強調表示するために構成された、組み込みのVerilog-Aテキストエディタが含まれています。新しいVerilog-Aデザインを開くには、メニューから**File | New Verilog-A**を選択します。既存のファイルを開くには、ファイルをコマンドシェルメッセージウィンドウにドラッグアンドドロップするか、メニューから**File| Open...**を選択します。右下のドロップダウンボックスからVerilogAファイルを選択します。

3.3 Hello World!

どのコンピュータ言語も“Hello world”プログラムを紹介することが慣習になりました。これは、単に“Hello world”と印刷するプログラムです。Verilog-Aはこの種のタスクを実行するようには設計されていませんが、それでもこのようなプログラムを作成することは可能です。以下に例を示します。

```
module hello_world ;

    analog
    begin

        @(initial_step)
            $strobe("Hello World!") ;
    end
endmodule
```

次の手順を使用してこれを試すことができます。

1. メニューの**File | New Verilog-A**を使用して、新しいVerilog-Aデザインを開きます。次に、新しいVerilog-Aで上記の行を入力します。（これはPDFからコピーアンドペーストしてOKですが、一般的にPDFからASCIIテキストをコピーアンドペーストすると奇妙な問題が発生する可能性があることに注意してください。特に、「-」文字に注意してください。これらは必ずしも見た目とは限りません。）
2. hello_world.vaというファイルに保存します。
3. 空の回路図シートを開きます。
4. 回路図を任意のファイル名で保存します。
5. メニューの**Verilog-A | Construct Verilog-A Symbol**を選択します。
6. 上記の手順2で作成したファイルに移動します。
7. OKを選択します。
8. 作成されたシンボルを配置します。これはピンのない単なる箱です。
9. 任意の停止時間で過渡解析を設定します。
10. シミュレーションを実行します。

これを初めて実行すると、コンパイル手順に関連するメッセージが表示されます。その後、メッセージ “Hello World!”がコマンドシェルに表示されます。

エラーメッセージが表示された場合は、入力したコードを確認してください。エラーメッセージは、問題が発生した行を指し示すはずです。示された行番号が正確でない場合があるこ

とに注意してください。構文解析ツールが何かが間違っていることを検出するポイントは、問題の実際の原因の1〜2行後に発生する可能性があります。たとえば、$strobe呼び出しを含む行で「;」を省略した場合、次の行またはその後の行についてもエラー「Unexpected token 'end'」が報告されます。「;」が欠落している場合、'end'トークンは予期されませんが、'end'は次の行にあります。

私たちのhello worldプログラムはそれほど多くを行いませんが、Verilog-Aの多くの概念を紹介しています。

1. Modules。モデルおよびインスタンスとしてインスタンス化できるすべてのデバイスは、モジュールとして定義されます。上記の例では、モジュールの名前は`hello_world`です。この名前は、このモジュールにアクセスするために、SIMetrixネットリストの関連する.MODEL文で使用されます。
2. キーワード`analog`で示される*アナログブロック*。これは、Verilog-A定義の本体が配置される場所です。
3. `@(initial_step)`で示される初期ステップ*イベント*。これに続く文は、シミュレーションの最初のステップ、つまりDC動作点フェーズでのみ実行されます。この行を削除すると、どうなるかを確認することができます。次のようにスラッシュで「コメントアウト」することで簡単に試すことができます。
   `//@(initial_step)`
4. $strobe。これは、Verilog-A LRM（言語リファレンスマニュアル）では*システムタスク*として知られています。$ strobeは、メッセージをコマンドシェルに出力します。また、さまざまな形式で値を出力でき、'C'言語のprintf関数と同様に動作します。これについては後で詳しく説明します。

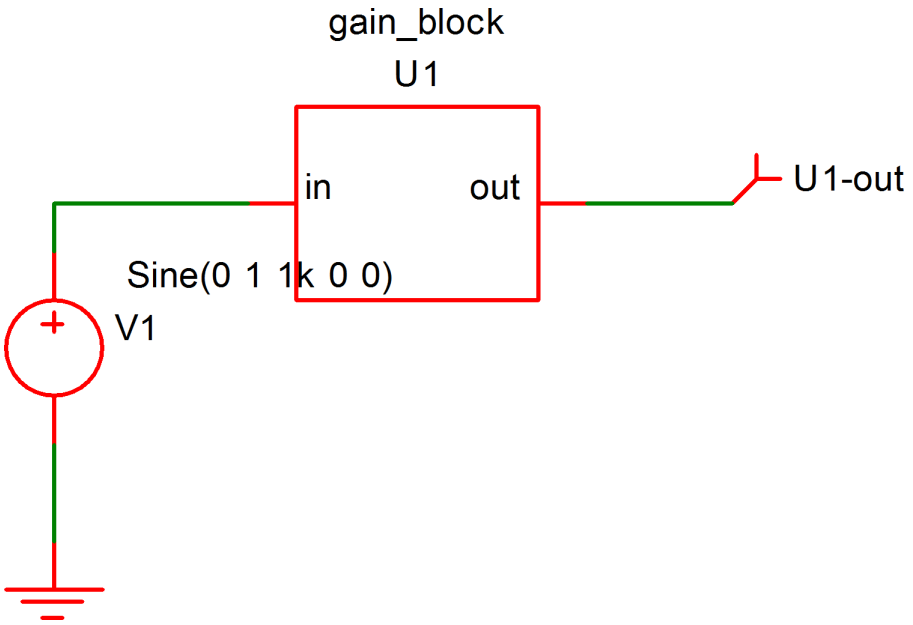
3.4 簡単なデバイスモデル

次に、簡単なゲインブロックを作成する方法を示します。Verilog-Aのデザインは次のとおりです。

```
`include "disciplines.vams"
module gain_block(in, out) ;

    electrical in, out ;
    parameter real gain=1.0 ;

    analog
        V(out) <+ V(in)*gain ;
```

```
endmodule
```

前のhello worldの例と同じ方法で上記の例を入力できます。次の回路を入力することをお勧めします。



上記の定義のためにメニューの**Verilog-A | Construct Verilog-A Symbol**を使用する場合、デバイスのピンの場所を尋ねるダイアログボックスが表示されます。hello worldの例ではデバイスにピンがないため、ダイアログボックスは表示されませんでした。ここでは、‘in’に ‘left’、‘out’に‘right’を選択します。

上記のビルド済みを提供しています。Examples/Manual/gain-blockを参照してください。このマニュアルで使用されているすべての例は、Examples/Manualから入手できます。ただし、コードと回路図を手動で入力する方が有益な場合があります。しかし、一部の文字が正しくコピーされない可能性があるため、このドキュメントからコピーアンドペーストすることはお勧めしません。手書き入力をお勧めします。`include`の前の引用文字は「back tick」であることに注意してください。通常は、US,、UK、その他のキーボードの数字列の左端のキーです。

通常の方法で上記を実行します。入力に続く出力が表示されます。それは1V 1kHzの正弦波です。

3.4.1 モジュールポート

上記の定義では、モジュール定義に2つの「モジュールポート」を導入しています。これらは接続端子を定義し、生成されたシンボルはこれらをピン‘in’およびピン‘out’として示します。

3.4.2 ブランチコントリビューション

下記の行は、モジュールのポートoutとポートinの間の関係を定義します。

```
V(out) <+ V(in)*gain ;
```

これは*ブランチコントリビューション*として知られ、Verilog-Aの最も重要な概念の1つです。ブランチコントリビューションは、シミュレータが右側の「プローブ」（上記のV(in)）と左側の「ソース」（上記のV(out)）の間で維持する必要がある関係を定義します。これらは、任意のソースデバイスと同じように動作します。たとえば、上記は次のようなSIMetrixネットリストの行と同等です。

```
B1 out 0 v = V(in) * gain
```

ただし、ブランチコントリビューションは、加法的であるという点で任意のソースとは異なります。同じ左側の連続したブランチコントリビューションは、互いに加算されます。これは、電圧源と電流源の両方に適用されます。 例えば、

```
V(out) <+ V(in)*gain ;
V(out) <+ 1.0 ;
```

は下記と同じです。

```
V(out) <+ V(in)*gain + 1.0 ;
```

上記のV()関数は、アクセス関数として知られています。アクセス関数には、それぞれポートまたは内部ノードを参照する必要がある1つまたは2つの引数があります。2つの引数が指定されている場合、V()は2つのノード間の電位にアクセスします。 単一のノードのみが提供される場合、そのノードとグランド間の電位にアクセスします。

アクセス関数I()は、2つのノード間を流れる電流にアクセスします。電圧アクセス関数と同様に、1つのノードのみが提供される場合、2番目のノードは暗黙に接地されます。

アクセス関数V()およびI()は、言語キーワードとして定義されていませんが、実際にはdisciplines.vamsファイル内に含まれるelectricalディシプリンによって定義されています。

3.4.3 パラメータ

下記の行はパラメータを定義し、デフォルト値1.0を指定します。

```
parameter real gain=1.0 ;
```

この値は、ネットリストレベルで編集できます。生成されたシンボルまたはビルド済みの例を使用した場合は、デバイスU1をダブルクリックして、次のように入力します。

```
gain=5
```

次に、回路図を再実行します。出力振幅が5Vピークに増加することに注意してください。

3.4.4 ディシプリン

最後に、hello worldの例にはない他の2つの行があります。

下記の行は、この場合「electrical」のモジュールポートディシプリンを定義します。

```
electrical in, out ;
```

Verilog-Aは、熱、機械、回転などの他のディシプリンをサポートしており、電気的および電子的以外の物理的プロセスのシミュレーションが可能です。これらの他のディシプリンの定義は、次の行を使用してインクルードされるdisciplines.vamsファイルで定義されています。

```
`include "disciplines.vams"
```

ほとんどすべてのVerilog-A定義は、ファイルの先頭にこの行を含みます。Hello Worldの例ではこの行は必要ないので、除外しました。

3.5 抵抗

この例では、簡単な抵抗を定義します。抵抗は、電流が端子間の電圧差に比例するデバイスです。これは、次のようにブランチコントリビューションを使用してVerilog-Aで定義されま

す。

```
I(p,n) <+ V(p,n)/resistance ;
```

これは、シミュレータがノードpおよびノードnで維持する必要がある電流/電圧の関係を定義します。I(p,n)はポートpからポートnに流れる電流を表し、V(p,n)はノードpとノードnの間で測定された電位差を表します。

完全な定義は次のとおりです。

```
`include "disciplines.vams"
module va_resistor(p,n) ;

    parameter real resistance = 1000.0 from (0.0:inf] ;
    electrical p, n ;

    analog
        I(p,n) <+ V(p,n)/resistance ;

endmodule
```

上記では、ゼロ以下の抵抗値を防ぐために、resistanceパラメータに値の範囲制限が与えられています。resistanceがゼロの場合、ゼロ除算エラーが発生します。

値がゼロの抵抗をブロックする代わりに、ゼロ電圧のコントリビューションを使用してゼロ抵抗を実装できます。これは、次のようにします。

```
`include "disciplines.vams"
module va_resistor(p,n) ;

    parameter real resistance = 1000.0 ;
    electrical p, n ;

    analog
    begin
        if (resistance!=0.0)
            I(p,n) <+ V(p,n)/resistance ;
```

```
        else
            V(p,n) <+ 0.0 ;
    end
endmodule
```

if (resistance!=0.0)で始まる条件文に注意してください。また、analogブロックはキーワードbeginとendで囲まれていることに注意してください。上記の場合では、これらは実際には必要ありませんが、analogブロックに複数の文がある場合に必要です。

3.6 ソフトリミッタ

これはソフトリミッタデバイスの定義です。これは、入力信号をある限界まで変更せずに通過させ、それを超えると、次の形式で指数関数的に減衰します。

```
1 - exp( -(v-vlim) )
```

下限についても同じことが逆に生じます。完全な定義は次のとおりです。

```
`include "disciplines.vams"
module soft_limiter(in, out) ;

    electrical in, out ;
    parameter real    vlow=-1.0,
                      vhigh=1.0,
                      soft=0.1 from (0:1.0) ;

    localparam real   band = (vhigh-vlow)*soft,
                      vlow_1 = vlow+band,
                      vhigh_1 = vhigh-band ;

    real vin ;

    analog
    begin

        @(initial_step)
```

```
            if (vhigh<vlow)
            begin
                $strobe("Lower limit must be less than higher
                limit") ;
                $finish ;
            end

        vin = V(in) ;

        if (vin>vhigh_1)
            V(out) <+ vhigh_1+band*(1.0-exp(-(vin-vhigh_1)/band));
        else if (vin<vlow_1)
            V(out) <+ vlow_1-band*(1.0-exp((vin-vlow_1)/band)) ;
        else
            V(out) <+ vin ;
    end
endmodule
```

Examples/Manual/Soft-limiterの例を参照してください。

上記の例では、次の新しい概念を紹介しています。

1. 変数。V(in)の値を保持するためにvinを使用します。この例では、単にコードを少し読みやすくするために使っています。ただし、変数には任意の値または式を格納でき、はるかに幅広い用途があります。
2. $finishシステムタスク
3. exp関数
4. localparamキーワードを使用するローカル変数
5. fromキーワードを使用したパラメータ値の範囲制限（前の抵抗でも使用）

ソフトリミットの例では、前の抵抗の例で最初に見たifおよびelseを使用した条件文も使用します。

3.6.1 変数

例のvinなどの変数は最初に宣言する必要があります。上記の例では、この宣言は次の行です。

```
real vin ;
```

これにより、変数は「real」と宣言されます。これは、コンピューティングの意味で「実数」であり、値は浮動小数点演算を使用して格納され、非整数の値を取ることができます。代わりの宣言はintegerで、変数が整数を格納することを意味します。パラメータ宣言などの変数宣言は、module - endmoduleのブロック内に配置する必要があります。上記の例のようにanalogブロックの外側で宣言するか、名前付きのbegin - endブロックの内側で宣言できます。例えば次のとおりです。

```
begin : main
    real vin ;
    ...
end
```

このように宣言された場合、変数は宣言されたbegin - endブロック内でのみ使用できます。

3.6.2 $finish

$finishシステムタスクはシミュレーションを無条件に中止します。

3.6.3 関数

Verilog-Aには、さまざまな数学関数が組み込まれています。上記の例では、exp関数を使用しました。 完全なリストについては、 “Verilog-A Functions”を参照してください。

3.6.4 ローカルパラメータ

ローカルパラメータは、ユーザが.MODEL文またはその他の手段で変更できないパラメータです。ローカルパラメータは、変数とは異なり、宣言以外では代入することができないため、定数値を定義する方法です。この例では、bandローカルパラメータを次のように宣言しました。

```
localparam real band = (vhigh-vlow)*soft
```

これを行うための変数を単純に定義することもできます。ただし、ローカルパラメータを使用することで、後で変更できないと知ることができます。これにより読みやすさが向上しますが、より重要なことは、コンパイラに変更できないことを伝え、結果を効果的に最適化できるよう

になることです。

3.6.5 パラメータ制限

パラメータには、上限と下限を指定できます。これは、fromキーワードを使用して行われます。上記の例では、次の行はsoftに0〜1.0の*exclusive*な制限を定義します。

```
soft=0.1 from (0:1.0)
```

つまり、0より大きく1.0より小さい値は受け入れられますが、0と1.0の値は許可されません。丸括弧の代わりに角括弧を使用して、*inclusive*な制限を定義することもできます。たとえば、次の場合、1.0は許可されています。

```
soft=0.1 from (0:1.0]
```

3.6.6 条件文

条件文の形式は次のとおりです。

```
if (conditional-expression)
    statement ;
else
    statement ;
```

*statement*は、ブランチコントリビューションなどの単一の文でも、beginとendで囲まれた文の集まりでもかまいません。

3.7 キャパシタ

キャパシタを実装するには、時間微分関数が必要です。Verilog-Aでは、これはddtアナログ演算子を使用して実現されます。キャパシタは、次のブランチコントリビューション文を使用して定義できます。

```
I(p,n) <+ capacitance * ddt( V(p,n)) ;
```

抵抗と同様に、これはシミュレータがノードpおよびノードnで維持しなければならない電流/電圧関係を定義します。ただし、この定義には時間依存性があります。

キャパシタの完全な定義は次のとおりです。

```
`include "disciplines.vams"

module va_capacitor(p,n) ;

    parameter real capacitance = 1n;
    electrical p, n ;

    analog
        I(p,n) <+ capacitance * ddt(V(p,n)) ;

endmodule
```

Examples/Manual/Capacitorを参照してください。初期条件のパラメータを持つキャパシタには別の定義capacity_with_ic.vaがあることに注意してください。これは、初期条件の指定を可能にする時間積分演算子idtを使用します。

3.8 電圧制御発振器

Verilog-Aは、信号ソースの作成に使用できます。ここでは、電圧制御発振器の作成方法を示します

```
`include "disciplines.vams"
`include "constants.vams"

module vco(in, out) ;

    parameter real  amplitude = 1.0,
                    centre_frequency = 1K,
                    gain = 1K ;

    parameter integer steps_per_cycle=20 ;

    localparam real   omegac = 2.0 * `M_PI * centre_frequency,
                      omega_gain = 2.0 * `M_PI * gain ;
```

```
    electrical in, out ;

    analog
    begin : main

        real vin, instantaneousFreq ;

        vin = V(in) ;
        V(out) <+ amplitude*sin(idt(vin*omega_gain+omegac,0.0)) ;

        // Use $bound_step system task to limit time step
        // This is to ensure that sine wave is rendered with
        // adequate detail.
        instantaneousFreq = centre_frequency + gain * vin ;
        $bound_step (1.0 / instantaneousFreq / steps_per_cycle) ;
    end
endmodule
```

これは、Examples/Manual/Vcoにあります。

このモデルは、idtアナログ演算子を使用して、周波数を積分して位相を取得します。周波数は、定数項であるomegacと、電圧制御項であるvin*omega_gainから計算されます。

正弦波信号の問題は、適切な分解能を得るために、時間ステップをサイクル時間の制御された割合に制限する必要があることです。上記では、パラメータsteps_per_cycleを使用して、サイクルごとの最小ステップ数を定義しています。これは、$bound_stepシステムタスクを使用して実装されます。これにより、次の時点で使用できる最大時間ステップがシミュレータに通知されます。必要に応じて小さなステップを使用できますが、大きなステップを使用しないでください。

非常に多くのサイクルを実行し続けると、上記は問題が発生する可能性があります。idt演算子からの戻り値は継続的に増加し、最終的にこの値のサイズは利用可能な計算精度に影響を与え、不正確になります。この問題は、idtmod演算子を使用して解決できます。

3.9 デジタル素子 - 概要

Verilog-Aは、アナログデバイスだけでなくデジタルデバイスもモデル化できます。これは、いくつかの簡単な論理関数が主にアナログ素子とインターフェイスする状況で役立ちます。1つの例は、位相ロックループの位相検出器です。

入力の少なくとも1つはしばしばアナログソースからのものであり、その出力は通常アナログコンポーネントで実装されたローパスフィルタを駆動します。一部の位相検出器のデザインでは、通常デジタルイベントドリブンシミュレータに適したデジタル状態マシンを採用しています。ただし、アナログデバイスとインターフェイスする場合は、インターフェイスブリッジをアナログ信号に接続する必要があります。これにより、シミュレーションが複雑になり、速度が低下します。Verilog-Aを使用すると、アナログ領域で機能全体を効率的に実装できます。

位相検出器の例を提供しました。Examples/phase_detectorを参照してください。

このセクションでは、いくつかの簡単な論理要素を作成する方法を示します。

3.10 デジタルゲート

ANDゲートの定義は次のとおりです。

```
`include "disciplines.vams"

module and_gate(in1, in2, out);

    electrical in1, in2, out ;

    parameter real    digThresh = 2.0,
                      digOutLow = 0.0,
                      digOutHigh = 5.0,
                      trise=10n,
                      tfall=10n ;

    analog
    begin : main

        integer dig1, dig2, logicState ;
```

```
        // Detect in1 threshold
        @ (cross(V(in1)-digThresh, 0, 1n))
            if (V(in1)>digThresh)
                dig1 = 1 ;
            else
                dig1 = 0 ;

        // Detect in2 threshold
        @ (cross(V(in2)-digThresh, 0, 1n))
            if (V(in2)>digThresh)
                dig2 = 1 ;
            else
                dig2 = 0 ;

        logicState = dig1 && dig2 ? digOutHigh : digOutLow ;
        V(out) <+ transition(logicState , 0.0, trise, tfall) ;
    end
endmodule
```

この例では、2つの新しい概念を紹介します。

1. crossイベント
2. transitionアナログ演算子

3.10.1 cross()モニタイベント

crossイベント関数は、入力信号が論理閾値を超えたことを検出するために使用されます。次の行を考えてみましょう。

```
@ (cross(V(in1)-digThresh, 0, 1n))
```

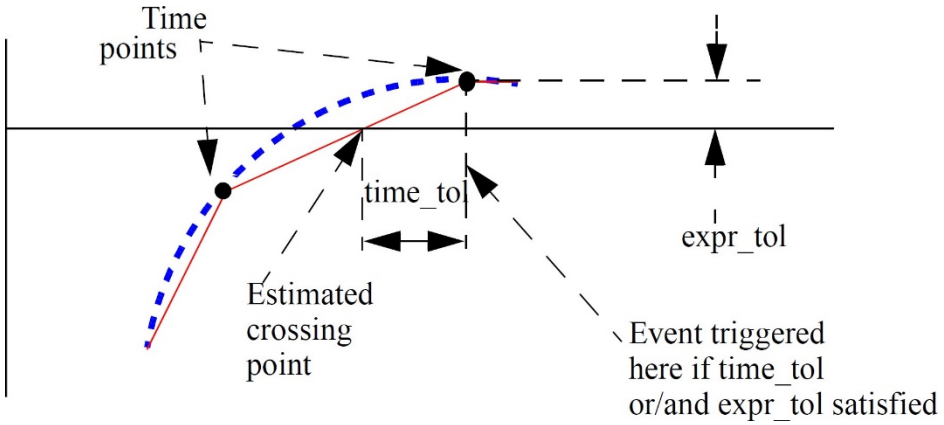
この行は、イベントを定義すると同時に、トリガーされたときにイベントに応答します。引数はイベントを定義しますが、それに続く文はイベントがトリガーされたときに実行されるアクションです。

この関数の形式は次のとおりです。

```
cross( expr, edge, time_tol, expr_tol )
```

最初の引数のみが必須です。

expr	テストする式。式がゼロとクロスするとイベントがトリガーされます。
edge	エッジを示す0、+1、または-1。+1は、exprが上昇しているときにのみイベントが発生することを意味し、-1は下降中にのみ発生することを意味し、0はいずれかのエッジで発生することを意味します。省略した場合のデフォルトは0です。
time_tol	ゼロクロッシング検出の時間許容値。入力が正確に線形でない限り、シミュレータがクロッシングポイントの正確な位置を予測することはできません。ただし、推定を行ってから、タイムステップをカットまたは延長して、定義された許容範囲内でヒットさせることができます。*time_tol*は、この推定の時間許容値を定義します。イベントは、現在のタイムステップと推定クロッシングポイントの差が*time_tol*未満になるとトリガーされます。省略するか、ゼロまたは負の場合、タイムステップ制御は適用されず、イベントはクロッシングポイントの後の最初の自然な時点でトリガーされます。このパラメータの意味の説明については、下の図を参照してください。
expr_tol	S*time_tol*に似ていますが、代わりに入力式の許容値を定義します。下の図を参照してください。



Cross Event Function Behaviour

3.10.2 transition()アナログ演算子

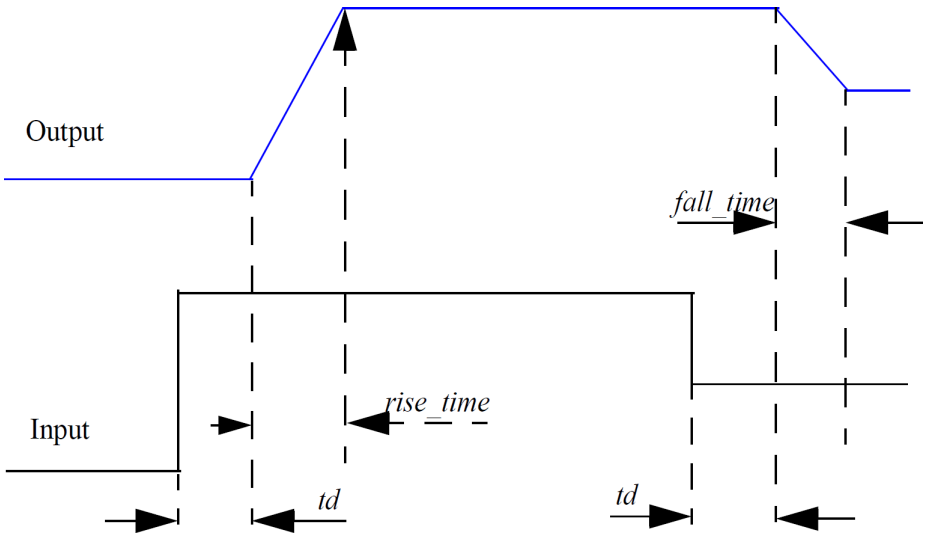
最後のtransition関数は、アナログ演算子と呼ばれる関数のクラスの1つです。前述のddt関数とidt関数もアナログ演算子です。transitionアナログ演算子は、ロジックデバイスやデジタルアナログコンバータの出力など、離散ステップで変化する信号を処理するように設計されています。上記のANDゲートの例では、出力ロジックレベルは瞬時に変化しますが、実際のデバイスの出力は通常、指定された立ち上がり時間または立ち下がり時間に従います。transitionアナログ演算子は、指定された立ち上がり時間と立ち下がり時間を使用して、離散的な入力値を連続的な出力値に変換します。この関数の形式は次のとおりです。

```
transition(expr, td, rise_time, fall_time, time_tol)
```

expr	入力式。
td	遅延時間。これは、*転送*または*格納*の遅延です。つまり、遅延時間中に入力が複数回変化した場合でも、指定された遅延時間後にすべての変化が出力に忠実に再現されます。これは、遅延よりも短い持続時間のアクティビティを吸収する*内部*遅延とは対照的です。 デフォルト= 0。
rise_time	入力の変化に応じた出力の立ち上がり時間。
fall_time	入力の変化に応じた出力の立ち下がり時間。
time_tol	無視されます。LRMは何をするべきかを明示的に述べておらず、許容範囲パラメータの目的は見当たりません。

*fall_time*が省略され、*rise_time*が指定されている場合、*fall_time*はデフォルトで*rise_time*になります。どちらも指定されていないか、ゼロに設定されている場合、非ゼロで最小の立ち上がり/立ち下がり時間が使用されます。これは、最小ブレークポイント値であるMINBREAKの値に設定されます。MINBREAKの詳細については、*Simulator Reference Manual/Command Reference/.OPTIONS*のOPTIONSを参照してください。

transitionアナログ演算子は、入力値を連続的に変化させるためには使用しないでください。代わりに、slewアナログ演算子またはabsdelayアナログ演算子を使用してください。



Transition Analog Operator Waveforms

3.11 バターワースフィルタ

ここでは、任意の次数のバターワースフィルタを示します。SIMetrixにはすでにこのようなものが組み込まれていますが、配列、ループ構造、およびLaplaceアナログ演算子を表すVerilog-Aバージョンを示します。

このデザインにより、ユーザはモデルパラメータを使用してフィルタの次数を指定できます。フィルタ自体は、アナログ演算子laplace_ndを使用して実装されます。この演算子は、分子と分母の多項式係数で定義されたラプラス伝達関数を提供します。指定した順序の係数を計算するには、forループを使用して分母の係数の配列を作成します。配列は一度だけ計算する必要があるため、initial_stepイベントに応答してこの計算を行います。（実際には、DC動作点の反復ごとに再計算されますが、効率はそれほど高くありません。これは、今後の改訂で対処したい分野です。）

```
`include "disciplines.vams"
`include "constants.vams"

module laplace_butter(in,ref,out) ;
    real res ;
    electrical in, ref, out ;
    parameter freq=1.0 ;
    parameter integer order=5 ;
```

```
        real scale, bPrev ;
        // Denominator array size
        real den[order:0] ;
        integer k ;

        analog
        begin

            //     Calculate Butterworth coefficients
            @ (initial_step)
            begin
                scale = 1.0/freq/2/`M_PI ;
                bPrev = 1.0 ;
                den[0] = 1.0 ;

                for (k=1 ; k<order+1 ; k=k+1)
                begin
                    bPrev = scale*cos((k-1.0)/order*(`M_PI*0.5))/
                            sin((k*0.5)/order*`M_PI) * bPrev ;
                    den[k] = bPrev ;

                    $strobe("den coeff \%d = \%g", k, den[k]) ;
                end
            end

            // Actual Butterworth filter
            res = laplace_nd( V(in,ref), {1.0}, den) ;
            V(out,ref) <+ res ;
        end
    endmodule
```

Examples/Manual/Butterworth-filterを参照してください。

このデザインでは、次の言語機能が紹介されています。

1. 配列変数
2. forループ
3. lapalace_ndアナログ演算子

3.11.1 配列

Verilog-Aは、変数とパラメータの両方の配列をサポートしています。上記の例では、配列を使用してlaplace_ndアナログ演算子の分母の係数を保存します。配列変数は、次の構文を使用して、許可されたインデックスの範囲で宣言する必要があります。

```
type array_name[low_index:high_index] ;
```

ここで

type	realまたはinteger
array_name	配列名
low_index	許可された最小インデックス
high_index	許可された最大インデックス

*low_index*および*high_index* は、配列内の要素の数を次のように決定します。

high_index − *low_index* + 1

3.11.2 forループ

forループは、'C'言語に似た構文を使用します。これは次のとおりです。

```
for (initial_assignment ; test_expression ; loop_assignment )
    statement
```

initial_assignment	ループに入るときに1回だけ実行する代入文（*変数* = *式*の形式）。通常、これはループカウンター変数に定数値を代入します。例では、変数kに1を代入します。
test_expression	式は、*statement*の前のループの各反復の開始時に評価されます。評価の結果が非ゼロの場合、*statement*が実行されます。そうでない場合、ループは終了します。
loop_assignment	*statement*の後に実行される代入文。通常、これはループカウンター変数をインクリメントまたはデクリメントします。上記では、kを1インクリメントします。

3.11.3 laplace_nd関数

laplace_nd関数は、ラプラス伝達関数を実装します。これは次の形式です。

$$H(s) = \frac{n_0 + n_1 s + n_2 s^2 + \cdots + n_m s^m}{d_0 + d_1 s + d_2 s^2 + \cdots + d_m s^m}$$

ここで、$d_0, d_1, d_2, \ldots, d_m$は分母の係数、$n_0, n_1, n_2, \ldots, n_m$は分子の係数、次数は$m$です。

laplace_nd関数の形式は次のとおりです。

```
laplace_nd (expr, num_coeffs, den_coeffs, ε)
```

ここで

expr	入力式
num_coeffs	分子の係数。これは、配列変数または配列初期化子として入力できます。配列初期化子は、「{」と「}」で囲まれた一連のコンマ区切り値です。 例：{1.0, 2.3, 3.4, 4.5}。 値は定数である必要はありません。
den_coeffs	分母の係数で、形式は分子と同じです（上記を参照してください）。例では、これは配列denとして提供されます。denの値はforループで計算されます。
ε	許容値のパラメータですが、現在は使用しません。

分母の定数項（上記の方程式のd0）がゼロの場合、ラプラス関数はクローズドフィードバックループ内に存在する必要があります。分母がゼロの場合、DCゲインは無限大です。関数をループ内に配置することにより、シミュレータは入力をゼロに維持して有限の出力を提供できます。そうでなければ、特異行列エラーが発生します。

3.12 RCラダー- ループ, ベクトル化ノード, およびgenvars

Verilog-Aでは、ノードのベクトルを使用して定義された繰り返し要素を、定義に含めることができます。ここでは、任意の数の要素を持つRCネットワークを定義する例を示します。

```
`include "discipline.h"

/* Model for an n-stage RC ladder network */
```

```
module rc_ladder(inode[0], inode[n]) ;

    electrical [0:n] inode ;

    /* The compile_time attribute is a SIMetrix extension and is
       not part of the Verilog standard. compile_time parameters
       must be defined at the time the module is compiled. Their
       values can be specified on the .LOAD line in the netlist
       using the "ctparams" parameter. E.g. ctparams="n=8"
       If not specified on the .LOAD line, the default value
       specified here will be used. */
    (* type="compile_time" *) parameter integer n=16 ;
    parameter r=1k ;
    parameter c=1n ;

    genvar i ;

    analog
    begin
        for (i=0 ; i<=n-1 ; i=i+1)
        begin
            I(inode[i],inode[i+1]) <+(V(inode[i],inode[i+1]))/r;
            I(inode[i+1]) <+ ddt(V(inode[i+1])*c) ;
        end
    end
endmodule
```

このデザインでは、次の言語機能が紹介されています。

1. ノードのベクトル
2. アナログforループおよびgenvars
3. コンパイル時パラメータ（これはSIMetrix拡張機能であり、Verilog-A仕様の一部ではありません）

3.12.1 ノードのベクトル

Verilog-Aでは、ノードをベクトルとして指定できます。これを使用して、複数の入力または

出力を持つデバイス（ADCやDACなど）や、上記の例のような複数の内部要素を持つデバイスを実装できます。

Verilog-Aの仕様では、ベクトル化されたノードのサイズを、実行時に代入することができるパラメータとして指定できます。SIMetrixはいくつかの単純なケースでこれを許可しますが、上記の例では受け入れられません。ただし、通常、ベクトル化されたノードサイズ（上記の例ではn）は、コンパイル時に使用できる定数として指定されます。これにはいくつかの方法があります。

1. 次のようなプリプロセッサ定数として。

   ```
   `define n 16
   ```

   これは、その後、バックティック文字を使用して、つまり`nのようにアクセスする必要があります。
2. 定数パラメータlocalparamとして。これはユーザが設定することができないため、コンパイル時に値が固定されます。
3. コンパイル時パラメータとして。 詳細は以下を参照ください。

ノードのベクトルは、ノードディシプリン宣言で指定できます。 上記の例では、これは次の行です。

```
electrical [0:n] inode ;
```

ノードは、配列変数が使用されるのと同じ方法で、定数式を囲む角括弧を使用してアクセスされます。たとえば、inode[0]はベクトル化されたノードinodeの最初のノードであり、inode[n]は最後のノードです。

3.12.2 アナログforループおよびgenvars

バターワースフィルタの例でforループを見ました。アナログforループは構文的には同一ですが、通常の変数の代わりに*genvar*と呼ばれる特別なタイプの変数を使用します。アナログforループは、ベクトル化されたノードを反復処理できる唯一のループです。また、アナログ演算子を使用できる唯一のループです。

*genvars*は、*generate*文と呼ばれるVerilog-Aバージョン1.0の概念から継承されています。generate文は、制御変数（*generate variable*または*genvar*）を増減しながら、文を何度でも複製する方法を定義します。コンピューターサイエンスでは、この手法はしばしば*ループ展開*と呼ばれます。generate文は廃止されたと見なされ、アナログforループに置き換えられま

したが、機能は似ています。

Verilog-A言語仕様は、アナログforループを展開することを規定していませんが、コンパイル時にすべての定数値が利用できる限り、genvarsの使用にいくつかの制限を課して展開を可能にします。ベクトル化されたノードを参照するループの展開は、実行時の評価よりもはるかに効率的です。

SIMetrixは、可能であればアナログforループを展開します。forループの1つ以上の値をコンパイル時に評価できなかったために展開できない場合でも、デザインの実装を試みます。しかし、このプロセスが失敗すると、エラーメッセージが表示されます。成功すると、レベル2の警告が表示され、一部の変数が一定である場合にデザインがより効率的になることを通知します。

3.12.3 コンパイル時パラメータ

コンパイル時パラメータはSIMetrix拡張機能であり、言語仕様の一部ではありません。

コンパイル時パラメータは、ネットリストの.LOAD文で割り当てるか、Verilog-Aコードの属性を使用して定義するか、その両方で行うことができます。この概念はまだ初期段階にあり、さらに発展させたいと考えています。コード内の属性（これは`(* type="compile_time" *)`がパラメータキーワードの前に付けられます）は、パラメータをコンパイル時として宣言し、デフォルト値を提供します。値は、ネットリストの.LOAD文で上書きできます。

Examples/DACのDACの例も参照してください。これには、実行時にモデルパラメータで指定できるサイズの、ベクトル化されたモジュールポートがあります。

3.13 間接代入

間接代入は、解くべき方程式を定義する方法です。ここでは、間接代入の2つの例を示します。完全な仮想アースと1次微分方程式の解です。

3.13.1 仮想アース

間接分岐代入のこの例を考えてみましょう。

```
V(out) : V(in) == 0.0 ;
```

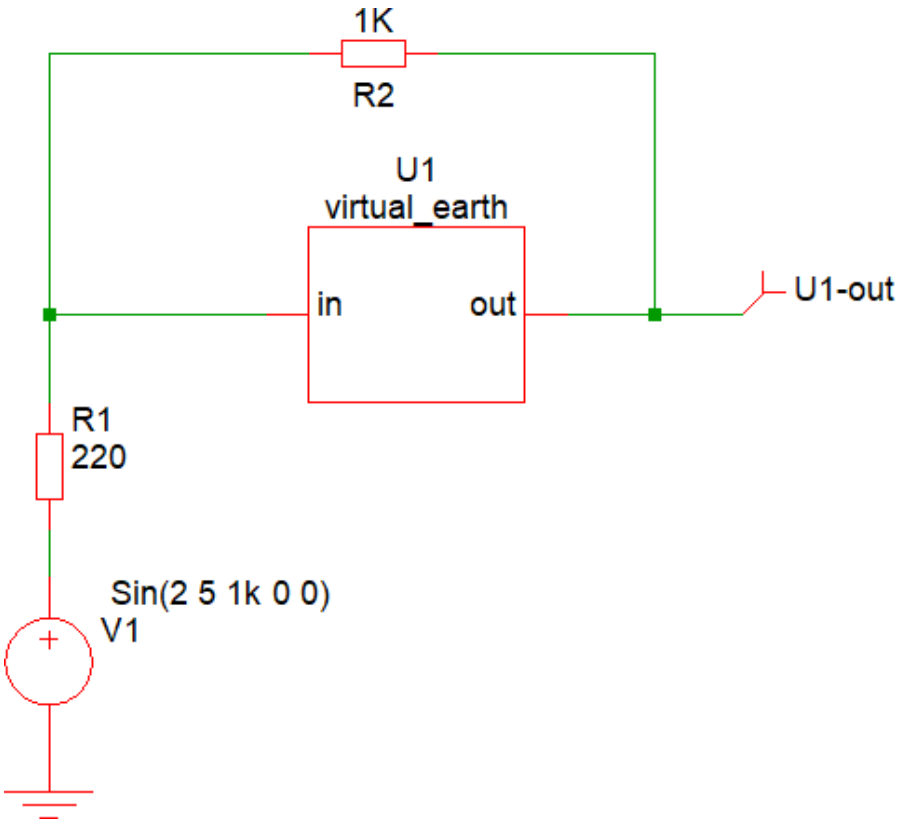
これは「V(in)がゼロに等しくなるようにV(out)を駆動する」と表現できます。これにより、

vinに仮想アースが作成されます。

完全なVerilog-Aモジュールは次のとおりです。

```
`include "disciplines.vams"

module virtual_earth(in, out) ;

    electrical in, out;

    analog
    begin

        V(out) : V(in) == 0.0 ;

    end
endmodule
```

上記は、次の回路で接続できます。



上記の回路では、U1の*in*ノードは正確に0ボルトに維持されるため、ゲインが1k/220の理想的な反転増幅器が作製されます。

3.13.2 微分方程式

次のVerilog-Aモジュールを考えてみましょう。

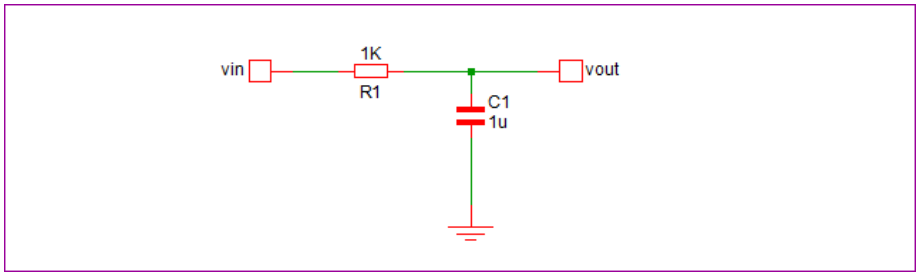
```
`include "disciplines.vams"

module differential_equation(in, out) ;

    electrical in, out;

    analog
    begin

        V(out) : ddt(V(out)) == (V(in) - V(out))/1m ;
```

```
        end
    endmodule
```

これは次の微分方程式を解きます。

$$\frac{d\ vout}{dt} = \frac{(vin - vout)}{0.001}$$

これは次の回路を支配する方程式です。



この回路図で試すことができます。