NSL チュートリアル

簡単な NSL 回路

次の例題は、NSL で記述する回路です。この例題は、シミュレーション専用の文法（_から始まる要素）を利用しているため、コンパイルしても、その部分は電子回路とはなりません。シミュレーションメッセージを出力するための記述です。この回路を tut0.nsl という名前のテキストファイルとして用意してください。

```
declare tut0 simulation { }

module tut0 {
   _finish("Hello World");
}
```

NSL の回路は、1 つ以上のモジュールから構成します。モジュールは、入出力仕様を記述する declare 文と動作本体の記述である module 文から構成します。どちらも、記述内容を {} で囲みます。この例題の declare 文には修飾語(simulation)がついています。この修飾語は、このモジュールがシミュレーション専用であり、論理合成の対象にならないことをコンパイラに伝えます。

モジュールにはモジュール名が必要です。モジュール名は、declare 文と module 文で同じ名前にします。 モジュールを外部から見た使い方は、declare 文によって定義します。例題のモジュールは、外部に対する入出力は持っていませんが、モジュール定義のための declare 文は省略できません。

この回路には実行文がひとつあります。

```
_finish("Hello World");
```

C の記述と同様に、実行文の後ろには、セミコロン ; を書きます。(C と同様に、複合文の後ろには不要です。) モジュール内に、複数の実行文を記述すると、それらの実行文は、並列に動作します。モジュールの実行文は、ハードウェアとして実現するため、電源投入とともに、動作を開始し、電源断まで、永続的に動作します。この点は、呼び出されたときにだけ動作する、プログラム言語の関数とは大きく異なるこ とに注意してください。ただし、この例題では、モジュール内の実行文は、シミュレーションを停止する文一つだけなので、実行を開始すると、すぐにシミュ レーションが停止します。(シミュレーションが停止しても、ハードウェアは消滅するわけではありません。)

_finish()は、メッセージをコンソールに出力し、シミュレーションを停止する関数です。_から始まる関数群は、シミュレーション専用であり、実回路にはならないことに注意してください。シミュレーション専用の関数、変数には次のものがあります。これらの関数や変数は、VerilogHDL に対応する関数・変数があり、VerilogHDL に変換するときに、引数をそのまま引き渡します。

_finish(string, arg1, arg2,...)： メッセージ出力後、シミュレーションを終了します。
_stop(string, arg1, arg2,...)： メッセージ出力後、シミュレーションを停止します。
_display(string, arg1, arg2,...)：メッセージを出力します。
_monitor(string, arg1,arg2,...)：引数に指定した端子が変化したときにメッセージを出力します。
_readmemh(file, mem)：ファイルの内容を16進数として、メモリに読み込みます。
_readmemb(file, mem)：ファイルの内容を2進数として、メモリに読み込みます。
_random：乱数を値として返します。(32ビット)
_time：シミュレーションタイムを返します。(64ビット)

_init { }：simulationモジュール内のみに使える順次実行ブロック。シミュレーションの開始1クロック後から、動作を順次実行します。シミュレーションモジュールの他のブロックの内側でない場所に記述可能。
_delay(数値)：_initブロック内で、指定する数値クロックの遅延をします。

注） シミュレーション用に開発されたVerilogHDLと異なり、NSLには、他のモジュールの内部要素を直接HDLとして記述する方法はありません。これを補うものとして、シミュレーションの結果の波形ファイルには、全信号を出力することができます。

NSLコンパイラは、モジュールが他のモジュールのインスタンスを階層的に利用する場合 、declare文を読み、そ のモジュールの入出力仕様を判定します。例題では、declare文とmodule文を同一のファイルに記述していますが、declare文と module文の二つを分離したファイルとして、declare文は、複数のモジュールのものをまとめて、ヘッダファイルとして用意しておくと、大規模な 回路を作成する場合に便利です。

シミュレーションを実行するために、この回路を、NSL COREでコンパイルします。
ここでは、VerilogHDLコンパイラ Icarus Verilogを用いた例を出しますが、SystemCへの合成を行ってSystemCコンパイラによるシミュレーションも可能です。

```
> nsl2vl tut0.nsl -verisim2 -target tut0
```

この操作で、tut0.vというVerilogHDLのファイルができます。
次に、Icarus Verilogで、コンパイルします。

```
> iverilog -o tut0.vvp tut0.v
```

それでは、実行してみましょう。

```
> vvp tut0.vvp

VCD info: dumpfile tut0.vcd opened for output.
Hello World
```

NSL 内に記述したメッセージを出力してシミュレーションが終了ましたね。VCD info から始まる行は Verilog コンパイラが出しているメッセージです。

SystemC への合成は次のコマンドで行います。

```
> nsl2sc tut0.nsl -scsim2 -target tut0 -split
```

最後のオプション -split に注意してください。SystemC コンパイラはモジュールの宣言順序に制約条件があるため、NSLCORE はモジュールごと別々のファイルに合成し、インクルードの順序を制御して SystemC の要求する制約条件を満たします。このとき、シミュレーションテストベンチは、指定したモジュール名に_sim を付加したファイル名となります。(この場合には tut0_sim.sc)SystemC 環境をどこにおくかによって、シミュレーション実行イメージのコンパイル方法は変わりますが、例として、LiveCygwin の場合を示します。g++コンパイラは .sc の拡張子のファイルをコンパイルできないので、ファイル名を修正して .cpp にしておきます。

```
> cp tut0_sim.sc tut0_sim.cpp
>  g++  -I/usr/local/systemc-2.2.0/include  -L/usr/local/systemc-2.2.0/lib-cygwin
-lsystemc -o  tut0_sim.exe tut0_sim.cpp
```

端子とレジスタ（構成要素）

次の例題は、NSL で記述する論理回路です。この例題も、シミュレーション専用であり、コンパイルしても、電子回路とはなりません。この回路を tut1.nsl という名前のテキストファイルとして用意してください。

```
declare tut1 simulation { }
module tut1 {
   reg count[8] = 0;
   count++;
   if(count==100) _display("Hello World");
   if(count==200) _finish("bye");
}
```

module 文の中に、内部で用いる端子やレジスタなどの構成要素と実現する回路を記述します。例題では、構成要素として 8 ビットのレジスタ count を定義して、その初期値を 0 としています。また、レジスタはクロックごとにカウントアップし、カウント値が 100 の時と 200 の時に、それぞれ_display 文と_finish 文を実行します。

大規模な回路を開発する場合、論理回路の各モジュールを同期回路として設計します。そこで、NSL は、通常、クロック信号を明示的には書きません。クロックやリセットの信号は、いわば上水道や下水道などのインフラにあたり、設計者が通常意識する必要がないからです。後述するように、必要に応じて、これらの信号を用い、同期回路同士を異なるクロックやリセットの信号で動かすこともできます。

プログラム言語における変数に相当する演算のための要素として、ハードウェア設計者は、回路中のネット（配線）に名前をつけます。この名前のついたネットを端子と呼び、演算に

おいて、端子名を記述することで、その値を参照します。NSL では、モジュールの内外を接続するための、入力端子や出力端子のほかに、モジュール内部の演算処理のための内部端子を用います。ネットの信号は、駆動されている間だけ電位を保ちますが、記憶素子（レジスタ、メモリ）の出力ネットだけは、電源が入っている間中、記憶内容を保ち続けるため、データを保持する特別の存在として扱います。

NSL においてモジュール内部で宣言する、ハードウェアに展開する基本的な構成要素は、次の 3 種類です。(他モジュールのインスタンスや整数、変数、構造体などの構成要素は別途説明します。) すべての構成要素は、実行文の前に宣言する必要があります。宣言では、構成要素の種類を示すキーワードに続けて、要素名とビット幅を指定します。

1. レジスタ　: レジスタは reg　レジスタ名[ビット幅]; と、名前とビット幅を指定します。セミコロンの前に = 値 として初期値を設定可能です。ビット幅を省略すると 1 ビットとみなします。レジスタはデータを記憶する素子です。レジスタに記憶されたデータは、明示的に変更されるまで同じ値を保ちます。

2. 内部端子 : 内部端子は wire 内部端子名[ビット幅]; と、レジスタ同様に名前とビット幅を指定します。ビット幅を省略すると 1 ビットとみなします。内部端子は、転送を受けたクロックサイクルの間だけ、データを保持します。

3. メモリ : メモリは、mem メモリ名[ワード数][ビット幅]; と、名前とワード数、ビット幅を指定します。
 mem mm[256][8] = { 3, 2, 5 }
 のように、初期値を設定可能です。メモリはレジスタを配列上に並べて、その番号で値を読み書きする素子です。

内部端子への値の転送は、転送が記述された条件のクロック内に行い、レジスタ、メモリへの値の転送は、次のクロックの立ち上がりに同期して実行します。 静的な主記憶領域を前提としたソフトウェアでは、変数の型の主な目的は主記憶上の記憶領域の大きさを指定することです。ハードウェアは、ソフトウェアと異なり、物理的な回路の大きさと、回路の機能を指定する必要があります。そこで、NSL は、回路機能として、レジスタ、内部端子、メモリの 3 つを提供します。

ハードウェアの特徴として、記述した回路は同時並列に動作します。そこで、回路の動作を条件によって選択的に行うための条件実行の構文を用意しました。条件実行は、条件付実行文のほかに、関数や手続き、状態があります。これらのの制御機構によって、回路動作の振る舞いを調整し、必要な機能を実現します。特に関数には、モジュール内部に完結する制御のほか、他のモジュールと連携を取るための外部からの／への制御があります。

ハードウェアに設計では、クロックの遷移する瞬間の動的なデータ変化が重要です。設計者は、内部構成要素のデータが、どのクロックで有効になるのかを常に把握しておく必要があります。

NSL には、回路として実現する記述の他に、論理シミュレーションを支援するための構文があります。この例題の declare 文に書いた simulation の修飾は、このモジュールが論理合成対象ではなく、シミュレーション時にのみ有効であることを示します。また、_display と _finish は、どちらもシミュレーション支援の組込み関数です。

シミュレーションを実行するために、この回路を、NSL CORE でコンパイルします。

```
> nsl2vl tut1.nsl -verisim2 -target tut1
```

この操作で、tut1.v という VerilogHDL のファイルができます。

次に、Icarus Verilog で、コンパイルします。

```
> iverilog -o tut1.vvp tut1.v
```

それでは、実行してみましょう。

```
> vvp tut1.vvp

VCD info: dumpfile tut1.vcd opened for output.
Hello World
bye
```

この回路記述が出力しているのは、Hello World と bye の 2 行になります。

NSL は同時並列実行を行っているので、メッセージを出力する順序は、順序を作るための回路より生成します。例題では、count というレジスタをクロックごとにカウントアップして、その値によって実行する文を切り替えます。プログラム言語は、記述した順序で実行しますが、ハードウェアは並列動作を基本とすることから、このような手続きが必要になります。（VerilogHDL のようにシミュレーションのためだけに、順序実行のための記述を持つ HDL もあります）

メモリ例題

次の例題はメモリと、メモリの初期化を用いた例題になります。この例題で用いる文法の一部は、まだ説明していませんので、チュートリアルの最後にもう一度読み直してみてください。

```
declare tut20 simulation { }
module tut20 {
    reg count[8] = 0;
    mem m[256][8];
    func_self readm();

    _init {
        readm();
        _display("Hello World:m[1]=%x", m[1]);
        _finish("bye");
     }

function readm {
```

```
        _display("readmem:%d", count);
    _readmemh("tut20.hex",m);
}
}
```

数値表現と整数表現

NSL では、演算に用いる数値表現にはビット幅を明示します。これは、ハードウェアの演算では、数値のビット幅が回路規模に大きな影響を与えるからです。NSL の式やレジスタ、端子の値は符号なし 2 進数として扱います。設計者にとって自明なビット幅となる演算を行う場合には、ビット幅を明示する記述は冗長なので、NSL では、明らかにビット幅が推定可能な部分においては、整数を指定することができます。自明な場合とは、レジスタや端子への値の転送、同一ビット幅同士の 2 項演算です。シフト演算におけるシフト数の指定も整数での表記が可能です。

ビット数を明示する数値表現には次の 7 種類があります。

1. 2 進数： 0b から始まる 2 進数は、表記したビット数を有する数値として扱います。例：0b100

2. 8 進数： 0o から始まる 8 進数は、表記した桁×3 ビットの数値として扱います。例：0o13

3. 16 進数： 0x から始まる 16 進数は、表記した桁×4 ビットの数値として扱います。例：0x123

4. 2 進数： 数値'b から始まる 2 進数は、数値で示すビット幅を持つ数値として扱います。例： 4'b1

5. 8 進数： 数値'o から始まる 10 進数は、数値で示すビット幅を持つ数値として扱います。例：8'o25

6. 10 進数： 数値'd から始まる 10 進数は、数値で示すビット幅を持つ数値として扱います。例：8'd20

7. 16 進数： 数値'h から始まる 16 進数は、数値で示すビット幅を持つ数値として扱います。例：8'h3

NSL では、整数は、32 ビット符号付の範囲を持ちます。レジスタやメモリの初期値や、ビット幅を推定可能な部分においても、32 ビット符号付整数で表せる数よりも大きな数値を必要とする場合には、整数による初期化ではなく、明示的にビット数を指定した数値表現を使います。

注：式や端子の値は符号なしに対して、整数は符号付なので、両者を混在して計算を行う場合、十分注意してください。

浮動小数点数

信号処理などで用いる数式の係数計算など、回路機能の記述を容易にするために浮動小数点数を使うことができます。浮動小数点数は、論理合成対象ではありませんが、コンパイル時に評価し、値を端子やレジスタ、構造体に定数として転送することができます。また、_int 関数で整数に変換すれば、値を整数定数として演算式において利用可能です。

浮動小数点数は、小数点を含む数値列、および、浮動小数点数 E 指数の形で表記します。浮動小数点数の演算は、加減乗除のほかに、次の初等関数が利用可能です。

_real,_int,_acos,_asin,_atan,_atan2,_ceil,_cos,_cosh,_exp,_fabs
_floor,_fmod,_log,_log10,_pow,_sin,_sinh,_sqrt,_tan,_tanh

式は、整数との混在はできません。_real を用いて浮動小数点数に変換するか、浮動小数点数だけを用いてください。

回路に計算結果を転送する場合、値を整数値として扱う場合には、_int(浮動小数点式)として転送します。IEEE754 形式のビット列として転送する場合には、式を転送の右辺に記述します。転送先は、32 ビットまたは 64 ビットの、wire, reg もしくは、構造体でなければなりません。この転送において、浮動小数点数は、IEEE754 浮動小数点単精度もしくは倍精度のビット列に変換されます。メモリ、レジスタなどの初期値としても利用できます。

```
例  Z = _sin(45./180.);     /* IEEE754 形式ビット列として転送 */
    Y = _int(_sin(45./180.))*127+128; /* 計算結果の整数を転送 */
```

if 文と比較演算

例題では if 文を用いてレジスタ count の値がある値になったら、対応する動作を行わせています。NSL の if 文は C 言語と同様に () の中に示される式が 0 の時には偽、0 以外の時には真として動作条件を判定します。比較のための演算子を示します。これらの演算子は条件が真の時に 1 ビットの 1 を返し、偽のときに 0 を返します。

== : 同一値のときに真
!= : 異なる値のときに真
> : 左の式が右の式より大きいときに真
< : 左の式が右の式より小さいときに真
>= : 左の式が右の式以上のときに真
<= : 左の式が右の式以下のときに真

比較演算とともに、論理演算子を使って複雑な比較をすることができます。論理演算子には、==, ||, !があります。NSL は、すべての値の評価を並列に行います。この点は、C 言語と異なるので、副作用のある関数呼び出記述する場合には、十分、その影響を確認してください。また、C 言語と異なり、値を転送する文（C 言語では代入文、NSL では転送文）は値を持たないので、転送を式に含めることはできません。複数の比較を優先順位をつけて評価する必要がある場合には、後述の alt 文を利用してください。

C 言語と同様に、if 文には else 節をつけることができます。

```
if(x > 0 || x<=10)     _display("ok:%d",x);
else                        _finish("bye");
```

if と else を組み合わせて複雑な動作条件を持つ回路を構成できます。しかし、複雑に入り組んだ if 文は分かりにくく、バグの温床になります。条件が複雑すぎると思うときには、多くの場合、NSL の持つ多方向分岐書式である any と alt を用いることで、見通しのよい回路を作ることができます。

if 文は、NSL の条件実行文の特別な形式です。一般形式は、any 文もしくは alt 文になります。これらの文は、条件 : 実行文の形で、複数の条件に対する多方向分岐処理を行います。また、これらの文には、条件として else を指定することもできます。any 文は、すべての条件を同時に評価し、条件が真となるすべての文を実行します。alt 文は、条件は上位の文から評価され、最初に真となる文だけを実行します。例題の二つの if 文と同じ動作をする any 文の例を示します。C 言語の switch 文と似ていますが、コロン : の左には論理式を記述することに注意してください。また、C 言語と異なり、条件に一致した場合には、コロン : の右の 1 文のみを実行します。複数の動作を行いたい場合には、後述の複合文を使います。

```
any {
    count==100: _display("Hello World");
    count==200: _finish("bye");
 }
```

alt 文を記述すると、コンパイルした結果のハードウェア中には、alt 文の動作原理に従い、プライオリティエンコーダが生成されます。プライオリティエンコーダはハードウェア性能のボトルネックとなる場合が多いので、alt を用いる回路を作成するときに、alt が本当に必要なケースであるかを十分吟味してください。また、同じ理由で、else の利用も性能上の理由から推奨しません。

転送と演算、および複合文

NSL の構成要素である、レジスタと端子には、値を転送できます。転送を受けると、端子やレジスタは、ビット幅を持った値を持ちます。レジスタはクロックをまたいで値を保持するのに対し、端子は、転送が起きたクロックの間だけ有効な値を持ちます。そこで、この二つの転送を区別するために、NSL では、レジスタへの転送記号(:=)と端子への転送記号(=)を異なる記号としています。また、値を記憶する構成要素であるメモリは、レジスタと同じ記号を持ちます。

レジスタへの転送は、転送を行ったクロックサイクルの次のクロックサイクルのはじめに値の変化が起きます。つまり、レジスタはリーディングエッジトリガフリップフロップで実現します。

レジスタや端子の値を用いて、演算を行うことができます。NSL には算術演算と論理演算があります。NSL の演算は、すべて、同一クロックにおいて結果を生じます。

算術演算は、次の 3 つを用意しています。除算は、実現方法が多数あり、かつ、1 クロックで値を返す実装を用いることはほとんどないため、演算子として用意していません。

＋: 同一ビット幅同士の加算を行います。結果は同じビット幅の値となります。第 2 項に整数を指定すると第 1 項のビット幅に推定します。

－:　同一ビット幅同士の減算を行います。結果は同じビット幅の値となります。第 2 項に整数を指定すると第 1 項のビット幅に推定します。

＊: 乗算を行います。ビット幅の制限はありません。結果は、第 1 項と第 2 項の和のビット幅を持ちます。

論理演算には、対応するビットごとに演算を行う演算と、1 語のビット方向に演算を行うリデュース演算があります。リデュース演算は、値の前に前置演算子を置き、演算結果は 1 ビットとなります。ビットごとの演算は、同一ビット幅同士の演算を行い、結果のビット幅は変わりません。ビットごとの演算では、ビット幅の推定が可能なので、第 2 項には整数が使えます。次の演算の中で否定以外の演算はリデュース演算子にも利用できます。リデュース演算は 1 ビットの信号には適用できません。

&:論理積
|:論理和
^:排他的論理和
~:否定

論理演算の例を示します。

```
a = 8'h55 & 170;
b = |a;
```

論理演算と比較演算の違いに十分注意してください。リデュース演算を用いた複雑な式を書く場合には、混乱しないよう、適切に括弧を用いてください。

その他の演算として、シフト、ビット連結、ビット切り出し、リピート、ビット幅変更、サイン拡張、条件演算があります。

>>	論理右シフト。結果のビット幅は変わりません。第 2 項は整数にしてください。
<<	論理左シフト。結果のビット幅は変わりません。第 2 項は整数にしてください。
{ 式 , 式, ... }	{ } で囲む複数の式を、並び順にビット連結した値を返します。
数値 { 式 }	式を数値回数リピートして連結した値を返します。
式[数値 1:数値 2]	式を数値 1 ビット目から数値 2 ビット目まで切り出した値を返します。
式[式 x]	式から式 x ビットの 1 ビットを切り出した値を返します。
数値 '(式)	式のビット幅を数値で示す幅に変更した値を返します。
数値#(式)	式を数値ビットにサイン拡張した値を返します。
if (式) 式 1 else 式 2	式が真のとき、式 1 を、偽のときに式 2 を値として返します。

NSL は、演算のビット数を厳しくチェックします。そのため、設計者は、式の途中で、任意の式もしくは数値に対して、ビット数を変更したい場合が生じるでしょう。その場合には、次のようにビット数の変更を行ってください。（ここで、y は 8 ビットの端子とします）

```
y = 8'(12);
```

演算式の一部だけを条件によって変更した場合、if 演算を用います。if 演算は、式 1 もしくは式 2 を値とします。たとえば、端子 x に a と b のうち小さいほうを転送する場合、次のように記述します。

```
x = if(a<b) a else b;
```

このほかに、記憶素子であるレジスタに対しては、++と--の演算子があります。++は、レジスタの値を 1 増やし、--は、レジスタ の値を 1 減らします。これらの演算子を、レジスタに限定しているのは、レジスタは、値が反映されるタイミングが次のクロックの立ち上がりであるため、レジス タの値を用いた結果を、そのレジスタに転送しても正しく演算できますが、端子の値を用いた演算結果を、同じ端子に転送すると、組合せ回路だけからなるループ回路が生成されてしまい、回路動作が正しく行われないからです。この演算子は、式として使うと、前置と後置の二通りの使い方ができます。

```
w = r++;
```

後置の場合には、もとのレジスタの値を式の値とし、その後、レジスタを加算もしくは減算します。

この例では、端子 w にレジスタ r の値を転送します。

```
y = --q;
```

前置の場合には、加算もしくは減算した結果を式の値として、その結果をレジスタに転送します。 この例では、端子 y にレジスタ q-1 の値を転送します。

また、演算を、動作として単独で記述することもできます。

```
s--;
```

この場合、レジスタ s の値を 1 だけ減算します。

中括弧 {} を用いて、複数の文からなる複合文を作ることができます。複合文には次の形式があります。

{ } : 複合文中のすべての文を同時に実行します。

```
if(x==0) {
   y=1;
   z=2;
}
```

x が 0 のとき、端子 y に 1 を、端子 z に 2 をそれぞれ同時に転送する。

seq { } : 1 クロックずつ順次実行するシーケンスブロックを記述します。関数の処理としてのみ記述可能です。それぞれの文はパイプラインを構成します。すなわち、実行された文が終了する前に、さらに次の起動をした場合、後続の処理は先行する処理の終了を待たずに実行を開始します。

```
func foo seq {
   a=1;
   a=2;
   a=3;
}
```

foo が呼ばれると、端子 a に 1, 2, 3 を順次転送します。

次の例題を tut2.nsl という名前のテキストファイルとして用意してください。

```
declare tut2 simulation { }

module tut2 {
   reg count[5]=0;
   wire x[5],y[5];
   any {
       count < 10 : {
               _display("x=%d,y=%d,count=%d",x,y,count);
               count := x;
               y = x + 1;
               x = count + 1;
               }
       count >= 10:    _finish("End");
   }
}
```

例題 tut2 は少し意地悪な順序で記述を並べています。先を読む前にどのような結果が出てくるか考えてみてください。

ヒントは、複合文の各文は、すべて同時に実行されるということです。文の記述順序は実行状況に関係しません。

シミュレーションを実行するために、この回路を、NSL CORE でコンパイルします。

```
> nsl2vl tut2.nsl -verisim2 -target tut2
```

この操作で、tut2.v という VerilogHDL のファイルができます。

次に、Icarus Verilog で、コンパイルします。

```
> iverilog -o tut2.vvp tut2.v
```

それでは、実行してみましょう。

```
> vvp tut2.vvp

VCD info: dumpfile tut2.vcd opened for output.
x= 1,y= 2,count= 0
x= 1,y= 2,count= 0
x= 2,y= 3,count= 1
x= 3,y= 4,count= 2
x= 4,y= 5,count= 3
x= 5,y= 6,count= 4
x= 6,y= 7,count= 5
x= 7,y= 8,count= 6
x= 8,y= 9,count= 7
x= 9,y=10,count= 8
x=10,y=11,count= 9
End
```

複合文の中はすべて同時に実行し、記述順には関係しないので、x と y への転送と、count への転送が同時に起こります。しかし、レジスタの値が変化するのは、次のクロックの立ち上がり時点なので、x よりも 1 クロック遅く変化が発生します。count が 0 の行が 2 行出ているのは、この回路には、リセット信号が入っていて、リセットがかかっている間、count が 0 のままとなるからです。

左辺ビット連結演算 .{}によって複数の端子やレジスタに、まとめて値を転送することができます。

```
.{ 端子 1, 端子 2, ... } = 値 ;
.{ レジスタ 1, レジスタ 2, ... } := 値 ;
```

右辺は、整数値もしくは、左辺の連結したビット数に対応する値とします。

関数（制御端子）と引数を持つ回路

NSL は、ハードウェアの機能を関数として呼び出して実行する文法を持ちます。 NSL では、関数名を、制御端子として参照できます。通常のデータ信号は、転送されていない場合、ハザード ('bx)が与えられるのに対して、制御端子は、関数が呼び出されたクロックにおいて 1 となり、呼び出されていない間は 0 となります。そこで、制御系の信号として、設定されないときに 0 となる必要がある信号として使うこともできます。モジュール内部に対する制御を行う内部関数（制御内部端子)、モジュール外からの制御を受ける入来関数（制御入力端子)、モジュール外に制御を発行する出立関数（制御出力端子）の 3 種類があります。それぞれの関数（制御端子）には仮引数と戻り値を定義できます。ソフトウェアと異なり、仮引数と戻り値は、ハードウェアの端子となります。そこで、関数（制御端子）の宣言の前に、あ

らかじめ仮引数となる端子を宣言しておきます。関数（制御端子）の宣言では関数（制御端子）名と仮引数となる端子名、ならびに必要ならば戻り値を返す端子名を示します。(仮引数と戻り値は省略可能です。)

自モジュールが出力する出立関数では、処理は外部のモジュールで行うため、モジュール内に動作の記述は行いません。ただし、サブモジュールの出立関数の動作を親モジュールに記述することはできます。

内部関数定義

func_self 関数名(引数内部端子リスト) : 返り値端子 ;

入来関数定義

func_in 関数名(引数入力端子リスト) : 返り値出力端子 ;

出立関数定義

func_out 関数名(引数出力端子リスト) : 返り値入力端子 ;

関数が呼び出されたときの動作は、func 文に記述します。

func 関数名 動作

関数の動作には、実行文として許されている文法がすべて記述できます。関数の値は、関数を呼び出したクロック中だけ有効です。

複合文である、seq 文は、関数の実行文としてのみ記述可能で、順次ステップ実行します。seq 文が動作を継続している場合でも、関数（制御端子）自体は、起動されたクロックだけ有効であり、関数の戻り値を利用する場合には、起動クロック以外の値を受け取れないことに注意が必要です。複数クロックで処理を行った演算結果を利用する場合、演算結果をレジスタに記憶し、その値を読み出す関数を別途作成してください。(演算終了状態を返す関数もあわせて作成しておくと便利です。)

```
input a,b;
output ret;
func_in  do_calc(a,b);
func_in  get_value(): ret;
```

次の例題は、内部関数を用いた論理回路です。この回路を tut3.nsl という名前のテキストファイルとして用意してください。

```
declare tut3 simulation { }

module tut3 {
   wire value[8], ret[8];
   func_self  start(value) : ret;
```

```
    if(_time>=100) _finish("Value = %d",start(1));
    func start {
        return ( value + 3 );
    }
}
```

_time は、シミュレーション時間を値として持つ組込み変数です。シミュレーション時間はクロックの立ち上がりと立下りの両方でカウントされるので、クロックの立ち上がりで動作する論理回路の時間と、必ずしも、一致するものではないことに注意してください。(比較演算子==で、_time の一致判定をすると、一致しない可能性があります。)

シミュレーションするために、この回路を、NSL CORE でコンパイルします。

```
> nsl2vl tut3.nsl -verisim2 -target tut3
```

こ の操作で、tut3.v という VerilogHDL のファイルができます。
次 に、Icarus Verilog で、コンパイルします。

```
> iverilog -o tut3.vvp tut3.v
```

それでは、実行してみ ましょう。

```
> vvp tut3.vvp

VCD info: dumpfile tut3.vcd opened for output.
Value = 4
```

順序実行、while と for による繰り返し処理回路

関数の処理には、順序実行を行うシーケンスブロックを定義できます。シーケンスブロック内では、記述した実行文を 1 クロックずつ順次実行します。シーケンスブロックの中で、繰り返し処理を記述するために while 文と for 文を用意しました。

while は括弧の中の式が 0 でない間、実行文を 1 クロックずつ順次実行します。ループの終了判定は実行文の実行に先立って行われます。そこで、ループからの脱出は、終了判定が偽 (0) となった後で行われることになります。

```
        while(count<loop) {
             _display("loop = %d, count = %d", loop, count);
        }
```

tut4 に while を用いた例題を示します。レジスタ count と、仮引数となる端子 value を定義し、さらに制御内部端子 start に仮引数の value を指定して定義します。

```
declare tut4 simulation { }

module tut4 {
```

```
    reg count[8] = 0;
    wire value[8];
    func_self  start(value);
    count++;
    if(count==100) start(110);

    func start seq {
        reg loop[8]=0;
        loop:=value;
        while(count<loop) {
             _display("loop = %d, count = %d", loop, count);
        }
        _finish("bye: count = %d", count);
    }
}
```

この回路では、count の値が 100 になったところで、110 を実引数として内部関数 start を呼び出します。count の値は、仮引数 value に転送されて、同じクロック中に start を起動します。start が起動されたときに実行する処理は、func の実行文として記述します。例題では、シーケンスブロックが起動され、レジスタ loop への転送が start の起動クロックで実行されます。ついで、count の値が loop より小さい場合に、while 文がメッセージ出力を出す _display 文を起動します。count の値が loop 以上になると while は終了し、_finish を実行します。内部関数 start の仮引数を、いったん、レジスタに転送しなおしている理由は、端子である仮引数の値は呼び出されたクロックでのみ有効ですが、while は複数クロックの間実行を行うため、while の条件はレジスタの値で判定する必要があるからです。

シミュレーションするために、この回路を、NSL CORE でコンパイルします。

```
> nsl2vl tut4.nsl -verisim2 -target tut4
```

この操作で、tut4.v という VerilogHDL のファイルができます。

次に、Icarus Verilog で、コンパイルします。

```
> iverilog -o tut4.vvp tut4.v
```

それでは、実行してみましょう。

```
> vvp tut4.vvp

VCD info: dumpfile tut4.vcd opened for output.
loop = 110, count = 101
loop = 110, count = 102
loop = 110, count = 103
loop = 110, count = 104
loop = 110, count = 105
loop = 110, count = 106
```

```
loop = 110, count = 107
loop = 110, count = 108
loop = 110, count = 109
bye: count = 111
```

この回路は、並列処理と順序処理が並行して動作しています。count レジスタをカウントアップする処理、if 文による条件動作、それと、func 文のシーケンスブロックによる順序処理です。func 文の start 制御は、if 文の実行文によって起動され、その後、seq ブロックの動作を 1 つずつ実行しています。ハードウェアでは、通常、すべての処理が同時に並行して動作するので、プログラム言語と異なり、start 制御を呼び出しても、呼び出した元の回路の動作が終了するわけではないことに注意してください。

注意深い方は、最後の count が 111 であることに疑問をもたれたかもしれません。この理由は、while の実現方法にあります。while の合成は、次のステップで行います。

1. 条件式を評価する計算を合成します。式の値が 0 であれば、while ブロックの次の処理に進む goto 文を合成します。

2. while ブロック内の処理を順次合成します。最初の処理が、手続き、goto 文、ラベルでない場合には、式の評価と最初の文を同時に実行する合成をします。

3. 条件式の評価に戻る goto 文を合成します。ブロックの最後の文が、、手続き、goto 文、ラベルでない場合には、最後の文と式の評価への goto 文を同時に実行する合成をします。

例題 tut4 では、count<loop が条件式になります。count の値が 110 になったとき、条件式が偽となり、while を抜けて、次の文に進みます。次の文の実行（ここでは、_finish()）は、1 クロック後に起動されるため、count の値は 111 に変化しています。条件式が偽となるまで待つのは、言語仕様であり、これ以上はタイミングを詰められません。

ループ処理を使って、記述をコンパクトにしたいけれど、タイミングをきっちりと詰めたい場合には、次の例題（tut5）に示す数え上げ形の for 文を使ってください。

```
declare tut5 simulation { }

module tut5 {
   reg count[8] = 0;
   func_self  start();
   count++;
   if(count==100) start();

   func start seq {
       reg loop[8];
       for(loop:=0,9) {
            _display("loop = %d, count = %d", loop, count);
       }
       _finish("bye: count = %d", count);
```

```
    }
 }
```

数え上げ型の for 文は、レジスタ変数の値を指定した数値の数だけ 1 ずつ増加もしくは減少させて、ブロック中の文を実行します。変数が最終値に達したところで、ループを終了するため、条件が偽となってからループを終了する while 文等と異なり、判定のためのクロックが不要になり、1 クロック早く次の文が実行可能です。変数の範囲には整数値だけが指定できます。変数範囲を loop:=9,0 とすると、9 から逆順に数え上げます。

```
> nsl2vl tut5.nsl -verisim2 -target tut5
> iverilog -otut5.vvp tut5.v
> vvp tut5.vvp

VCD info: dumpfile tut5.vcd opened for output.
loop =   0, count = 101
loop =   1, count = 102
loop =   2, count = 103
loop =   3, count = 104
loop =   4, count = 105
loop =   5, count = 106
loop =   6, count = 107
loop =   7, count = 108
loop =   8, count = 109
loop =   9, count = 110
bye: count = 111
```

for 文に関しては、C 言語の for 文に慣れている設計者のため、C と似た形式を用意しています。

```
for(式 1; 式 2; 式 3) { 実行文 }
```

式 1 は、ループの開始前に実行し、式 2 が真の場合、実行文を順次実行し、その後式 3 のを実行します。式 1、式 3 は、並列ブロックにより、複数の実行文を同時に指定することができます。、ループの評価については、while と同じく、条件式が偽となってからループを脱出するのですが、式 3 が、レジスタ++もしくは、レジスタ--の形の場合だけ、条件評価をループの最後において、式 3 の変更後の値を用いて評価するように合成していますので、典型的な例（tut6）では、while よりも 1 クロック早くループの脱出が可能です。

```
declare tut6 simulation { }

module tut6 {
   reg count[8] = 0;
   wire value[8];
   func_self  start(value);
   count++;
   if(count==100) start(5);
```

```
    func start seq {
        reg loop[8]=0,lend[8]=0;
        for({loop:=0;lend:=value;};loop<lend; loop++) {
            _display("loop = %d, count = %d", loop, count);
        }
        _finish("bye: count = %d", count);
    }
}
```

シミュレーションを実行してみましょう。

```
> nsl2vl tut6.nsl -verisim2 -target tut6
> iverilog -otut6.vvp tut6.v
> vvp tut6.vvp

VCD info: dumpfile tut6.vcd opened for output.
loop =   0, count = 101
loop =   1, count = 102
loop =   2, count = 103
loop =   3, count = 104
loop =   4, count = 105
bye: count = 106
```

例題 tut6 では、内部関数 start の引数を用いてループの制御をしています。 内部関数の引数は端子に与えられるため、for 文のような複数クロックの動作を行う文で利用する場合には、いったん、レジスタに受けなおす必要があります。この例では、for 文の先頭において、ループ変数 loop の初期化とともに、レジスタ lend へ仮引数 value の値を転送しています。

階層構造

分割して統治する。これが、大規模なハードウェアの開発の常道です。実用的な規模の回路開発では、必ずと言っていいほど、階層構造が使われます。階層構造の利点は、モジュール性の向上です。モジュール性の高い設計は、開発効率が高いだけでなく、保守性もよくなります。NSL の回路をモジュールと呼びますが、モジュールは、他モジュールを構成要素として持つことができます。モジュールに読み込んだ他モジュールを、サブモジュールと呼びます。サブモジュールを構成要素として利用する場合には、その入出力関係を示した declare 文が、あらかじめ定義されている必要があります。サブモジュールの module 文自体はなくても、NSL のコンパイルは可能です。もちろん、回路として実現する場合には、全部のモジュールが定義されている必要がありますが、NSL コンパイラは、分割コンパイルのために、declare 文をプロトタイプ文として利用しています。

NSL には、もう一つ、構造を抽象化する構成要素として、構造体があります。構造体は、declare 文同様、あらかじめ、struct 文を用いて、定義されている必要があります。

1. declare 文： モジュールの入出力を規定します。また、モジュールの属性に、interface と simulation があります。前者は、クロック信号を含め、モジュールの、すべての入出力ピ

ンを declare で定義することを示し、後者は、このモジュールが論理シミュレーション用であり、実回路には展開しないことを意味します。

```
declare モジュール名 モジュール属性 {
  入出力ピン定義
入来関数定義
出立関数定義
}
```

モジュール名以外の定義は、対応するものがなければ、省略可能です。

2. struct 文： 構造体のビット割り当てを規定します。C 言語の構造体と異なり、構造体の宣言に型を指定しません。これは、構造体のビット割り当てをレジスタに使うのか、端子に使うのかはユーザが決めることであり、かつ、同じ構造体を双方に使う場合もあるからです。C 言語では、型はメモリ上に占める大きさの情報でしかありませんが、NSL のレジスタと端子は、実体としてまったく異なるものなので、構造体の定義には型を含めないことにしました。構造体名の宣言には、後ろにセミコロンが必要です。現在の言語仕様では、本来、このセミコロンの必要性は、ないのですが、後の拡張のために言語仕様に取り入れています。

```
struct 構造体名 {
 メンバー名 1[ビット数];
 メンバー名 2[ビット数];
 .....
} ;
```

モジュールの中で、構成要素として、サブモジュールや構造体を利用する場合には、次のように定義します。

1. サブモジュール　: サブモジュールは、サブモジュール名 インスタンス名[多重度] ; と、記述し、多重度は省略可能です。

2. 構造体 : 構造体は、 構造体名 型 構造体インスタンス名 = 初期値 ; と宣言します。型は、レジスタ型(reg)もしくは端子型(wire)です。レジスタ型では、初期値が設定可能です。初期値の設定は、省略可能です。

サブモジュールや構造体を用いる例題は、どうしても、多少複雑になってしまいます。例題を注意深く読んでください。tut7 は、4 ビット全加算器のモジュール fadd4 を二つ用いて、8 ビットの加算をする例題です。8 ビットの信号を、二つの 4 ビット信号に分解するために、構造体を用いています。

```
declare tut7 simulation {}

declare fadd4 {
       input a[4];
       input b[4];
       input ci;
       output q[4];
```

```
        output c;
        output o;
        func_in ex(a,b,ci) : q;
}

struct byte_nibble {
        hi[4];
        lo[4];
} ;

module tut7 {
        byte_nibble reg count = 0 ;
        fadd4 sm[2];
        count++;
        if(count>60) {
                byte_nibble wire res;
                res.lo=sm[0].ex(count.lo,count.lo,0);
                res.hi=sm[1].ex(count.hi,count.hi,sm[0].c);
                _display("count:%d, res:%d, ovf:%d", count, res,sm[1].o);
        }
        if(count==67) _finish("bye");
}

module fadd4 {
        func ex {
                wire qs[4],qe[2];
                qs = {0b0,3'(a)} + {0b0,3'(b)} + 4'(ci);
                qe = (2'(a[3])+2'(b[3])+2'(qs[3]));
                c = qe[1];
                o = qe[1] ^ qs[3];
                return {qe[0],3'(qs)};
        }
}
```

構造体 byte_nibble は、4 ビットのデータ二つからなる 8 ビットの構造体を定義しています。tut7 モジュールは、レジスタ型 count と端子型 res の二つの構造体インスタンスを生成しています。count は 0 に初期化し、クロックごとに 1 つずつ加算していきます。加算は、構造体全体（つまり 8 ビット）に対して行われることに注意してください。count の値が 60 より大きい場合、二つの全加算器を用いて、加算を実行します。count の値は、構造体のメンバーを用いて、hi と lo に分解し、全加算器の呼び出しに用います。下位の 4 ビットからのキャリーを上位の加算器のキャリー入力にいれ、それぞれの加算器の結果は、端子型構造体 res のメンバーに転送します。これらの計算結果と、上位の加算器のオーバーフロー出力を計算結果として表示します。NSL では、数値を符号なし 2 進数として扱うので、オーバーフローを起こしても、符合はマイナスには見えません。しかし、8 ビットの 2 の補数表記で考えれば、負の値になっていることが分かると思います。

4 ビット全加算器 fadd4 は、オーバーフローの計算のため、下位 3 ビットの計算と、MSB の計算を分離しています。キャリーを得るため、MSB の計算を 2 ビットで行っています。また、キャリーと、3 ビット目からの桁上げの排他的論理和をオーバーフローとしています。

それでは、実行してみましょう。ここでは、Windows 版 NSL CORE を用いて、シミュレーション用の VerilogHDL を作成します。Language が Simulation となっていることを確認し、右下の Target に tu7 を記入した上で、tut7.nsl を VerilogHDL シミュレーションファイルに変換してください。その後、Icarus Verilog でコンパイル、シミュレーションを行います。

```
> iverilog -otut7.vvp tut7.v
> vvp tut7.vvp

VCD info: dumpfile tut7.vcd opened for output.
count: 61, res:122, ovf:0
count: 62, res:124, ovf:0
count: 63, res:126, ovf:0
count: 64, res:128, ovf:1
count: 65, res:130, ovf:1
count: 66, res:132, ovf:1
bye
count: 67, res:134, ovf:1
```

count の値を 2 倍した数値が res に返ってきているのが分かります。8 ビットの 2 の補数表記では、127 までが正の数で、それ以上の数は負になります。そこで、128 以上の結果は、本来の符号と異なる結果を返していることになり、オーバフローとなります。符号なしの演算を行うときには、オーバフローは無視します。count が 67 になったときには、二つの_display 文が同時に動作するので、シミュレータにより、この二つの順序は変化します。

構造体で宣言した端子やレジスタと同一のビット幅を持つ式の一部を構造体メンバーとして参照したい場合には、キャスト演算により、対象ビットの切り出しができます。

たとえば、

```
struct inst {
  op[8];
  r1[4];
  r2[4];
} ;
```

と宣言された全 16 ビットの構造体があります。16 ビットのレジスタ opreg から、構造体のメンバー名を用いて op メンバーの値を取り出したいときには、

```
   workop = (inst)(opreg).op;
```

と、(構造体名)(式).メンバー名　の形式で式を構造体にキャストしてメンバーの参照ができます。ただし、キャストは転送の左辺には使えないことに注意してください。

プリプロセッサ

NSL は、プリプロセッサディレクティブとして、次のものを用意しました。

マクロ定義
#define マクロ名 定義値: マクロを定義します。ソースコード中に書かれたマクロ名は、定義した値に変換します。識別子中に、マクロ名を利用する場合、%%で囲みます。
#undef マクロ名: マクロ定義を取り消します。
条件コンパイル
#ifdef マクロ名: マクロが定義されている場合、以降#endif もしくは#else までのソースコードをコンパイルします。
#ifndef マクロ名: マクロが定義されていない場合、以降#endif もしくは#else までのソースコードをコンパイルします。
#if 数値 : 数値が 0 以外の場合、以降#endif もしくは#else までのソースコードをコンパイルします。
#else : #ifdef/#ifndef の逆の条件で、以降#endif までのソースコードをコンパイルします。
#endif : 条件コンパイルの停止
インクルード
#include〈ファイル名〉: NSL_INCLUDE 環境変数のパスから、ファイル名を検索し、ソースコードとして挿入します。
#include"ファイル名": カレントディレクトリもしくは、コンパイル時に指定したパスから、ファイル名を検索し、ソースコードとして挿入します。

プリプロセッサである、nslpp.exe の出力は、C 言語のプリプロセッサと互換性があります。そこで、さらに高度なマクロを利用したい場合、gcc などを NSL のフロントエンドとして使うこともできます。

ラベルと goto 文

ソフトウェアの世界では、goto 文は排除する方が見通しのよいプログラムとなります。ところが、1 クロックの隙間も惜しむハードウェア設計では、for や while などの構造化機構だけでは、詰め切れないタイミングを調整するために goto 文が必要となることがあります。NSL の goto 文は、シーケンスブロックの制御に用います。後述する、ステートマシンの状態遷移にも goto 文を使いますが、両者はまったく別のものです。

goto 文で実行制御を切り替える先にはラベルが必要です。ラベルは、label_name 文で、シーケンスブロックの先頭部分において宣言し、ラベル名 : の形で、切り替え先を定義します。

次に示す例は、goto 文を用いて、ある条件を待ち合わせます。ここで、if 文で判定した結果が偽の時に、else の文が実行されるため、同じクロック内で、値を取り込むことができます。この形式の回路は、出力タイミングが不明な外部ブロックとの待ち合わせなどに用いることができます。

```
func ex seq {
```

```
 label_name l1;
 reg value;
 l1: if(!condition) goto l1;
     else value:=something;
}
```

この例は、関数 ex の処理の中で、condition が偽の間は、処理を停止し、真となったときに、その値をレジスタ value に取り込みます。この関数は、シーケンスブロックを用いて、複数クロックでの動作をするので、関数から値を返すための return 文は利用できないことに注意ください。 return 文は、記述可能ですが、関数を呼び出したクロックと異なるタイミングで値が返るので、呼び出した側が受け取る手段が限られます。

次の例題は、割り算回路のサブモジュールを呼び出します。割り算回路が出立関数で結果を報告したときに、その値を表示します。

```
declare tut8 simulation {}
#define N 10
#define M 8

declare divu_%N%_%M% {
input A[N],B[M];
output Q[N],R[M];
func_in divu_do(A,B);
func_out divu_done(Q,R);
func_out divu_error;
}

module divu_%N%_%M% {
reg QB[M], QQ[N+M];
wire sub_i1[N+1], sub_i2[N], minus;
func_self sub(sub_i1,sub_i2);

func sub {
wire sub_o[N+1];
 sub_o = {sub_i1} - {0b0,sub_i2};
 minus=sub_o[N];
}

func divu_do  {
   if(B==M'b0) divu_error();
   else seq {
   reg bitcount[M];
   for( {bitcount:=0; QB:=B; QQ:={M'b0,A};} ;
             bitcount < N ; bitcount++) {
        if(sub(QQ[N+M-1:M-1],(N'(QB)<<(N-M))).minus) {
         QQ := (QQ << 1) ;
        }
```

```
        else {
         QQ := {(sub_o << 1)[N:N-M],(QQ[N-2:0]<<1)} | (N+M)'b1;
        }
         }
   divu_done(QQ[N-1:0],QQ[(N+M-1):N]);
   }
   }
}

module tut8 {
   divu_%N%_%M%  divid;
   reg  a[N], b[M];
   func_self go();
   reg count[16]=0;
   count++;
   if(count == 10) go();
   if(divid.divu_error) _finish("divid error");

   func go seq {
       label_name wait_res;
       {
           a:=N'(_random);
           b:=M'(_random);
       }
       {
           divid.divu_do(a,b);
           _display("start %d/%d",a,b);
       }
       wait_res:
       if(!divid.divu_done) goto wait_res;
       else _display("result = %d : %d",divid.Q, divid.R);
       _finish();
   }
}
```

それでは、実行してみましょう。ここでは、Windows 版 NSL CORE を用いて、シミュレーション用の VerilogHDL を作成します。Language が Simulation となっていることを確認し、右下の Target に tu8 を記入した上で、tut8.nsl を VerilogHDL シミュレーションファイルに変換してください。その後、Icarus Verilog でコンパイル、シミュレーションを行います。

```
> iverilog -o tut8.vvp tut8.v
> vvp tut8.vvp

VCD info: dumpfile tut8.vcd opened for output.
start  911/ 18
result =   50 :  11
```

goto 文を使わずに、サブモジュールの出立関数の関数文を用いて、tut8 と同様の処理を行うこともできます。この例では、こちらの方が、goto 文を用いるより、構造的になっているので、分かりやすいでしょう。この例のように、ほとんどの goto 文は、関数を適切に記述することで、置き換え可能です。設計者は、goto 文を使わずに可能な方法がないのか、回路記述の見通しを悪くしないように慎重に検討してください。

```
module tut8_alt {
   divu_%N%_%M%  divid;
   reg  a[N], b[M];
   func_self go();
   reg count[16]=0;
   count++;
   if(count == 10) go();
   if(divid.divu_error) _finish("divid error");

   func go seq {
       {
           a:=N'(_random);
           b:=M'(_random);
       }
       {
           divid.divu_do(a,b);
           _display("start %d/%d",a,b);
       }
   }

   func divid.divu_done seq {
       _display("result = %d : %d",divid.Q, divid.R);
       _finish();
   }
}
```

手続き

複雑な論理回路は、多くの場合、パイプライン制御や状態制御を行います。NSL は、これらの制御構造を陽に表す文法を持ちます。その一つが手続きです。手続き(proc)は、呼び出しを受けたら、クロックに同期して起動を行い、他の手続きを呼び出すか、自ら終了を宣言する（finish()の実行）まで動作を続けます。手続きの宣言には、レジスタ型の仮引数を指定できます。手続きを呼び出すときに指定した値が、仮引数に転送された上で、手続きが起動します。

手続きが終了するクロックにおいて、他の手続き（もしくは自分自身）から起動を受けると、終了処理を実行しながら、次のクロックで再度起動を行います。これにより、パイプライン動作を円滑に記述可能です。

手続きの大きな用途の一つは、パイプラインステージの記述です。この場合、仮引数にはパイプラインレジスタを指定します。手続きが動作している状態を状態遷移マシンの状態とみ

なして回路を構成することも可能です。他の手続きを呼び出すことで、状態を遷移できます。また、関数の中で、まとまった処理を別途行わせる用途にも使います。シーケンスブロックの（他のブロックや if 文の下にない）トップレベルに手続き呼び出しを記述すると、シーケンスブロックは、手続きが終了するまで、実行を保留します。手続きが終了するクロックにおいて、手続き呼び出しの次の文が、手続きの終了時処理と同時に実行されます。

手続きは、構成要素の宣言として記述します。

proc_name　手続き名(仮引数 1, 仮引数 2, ...);

仮引数は、省略可能です。（その場合にも括弧は必要です。）

手続きを呼び出す場合には、

手続き名(実引数 1, 実引数 2, ...);

とします。手続き中で、他の手続きを呼び出すと、呼び出した側の手続きは終了します。また、明示的に終了する場合、

finish();

手続き名.finish();

を記述します。上の形式は、手続き内のみで利用可能で、記述した手続き自身を終了します。下の形式は、手続き外からも記述可能で、指定した手続きを終了させます。

次の例題は、5 個の命令を実装する、8 ビットの小さな CPU と、そのシミュレーション用のモジュールです。

```
#define ADD 0
#define LD  1
#define ST  2
#define JMP 3
#define JZ  4

declare cpu {
   inout data[8];
   output address[8];
   func_out mread(address) : data;
   func_out mwrite(address,data);
}

module cpu {
   reg count[8]=0, pc[8], op[8], im[8], acc[8]=0;
   proc_name ift(pc), imm(op), exe(im);

   any {
```

```
        count < 20: count++;
        count == 10: ift(0);
    }

    proc ift {
        imm(mread(pc++));
    }

    proc imm {
        exe(mread(pc++));
    }

    proc exe {
        wire nextpc[8];
        any {
            op == ADD: acc:=acc+im;
            op == LD:  acc:=mread(im);
            op == ST:  mwrite(im,acc);
        }
        any {
            op == JMP: nextpc=im;
            (op == JZ) && (acc == 0): nextpc=im;
            else: nextpc=pc;
        }
        ift(nextpc);
    }
}


declare tut9 simulation {}

module tut9 {
    mem mainmem[256][8] = {ADD, 2, JZ, 10, ST, 32, ADD, -1, JMP, 2, ST, 255};
    cpu tut9cpu;

    func tut9cpu.mread {
        _display("READ:        ADDRESS:%x,         DATA:%x",         tut9cpu.address,
mainmem[tut9cpu.address]);
        return mainmem[tut9cpu.address];
    }

    func tut9cpu.mwrite {
        _display("WRITE: ADDRESS:%x, DATA:%x", tut9cpu.address, tut9cpu.data);
        mainmem[tut9cpu.address] := tut9cpu.data;
        if(tut9cpu.address == 255) _finish("SIM STOP");
    }
}
```

モジュール CPU が、CPU の回路となり、tut9 がシミュレーションモジュールになります。

CPU は、8 ビットの双方向データバス、8 ビットのアドレス バスを持ち、メモリの読み込み(mread)、書き込み(mwrite)の出立関数で CPU の外に置くメモリとの入出力を行います。CPU は、3 つの手続き(ift, imm, exe)を持ち、それぞれ、8 ビットの仮引数レジスタ(pc, op, im)を指定します。これらの手続きは、命令読み出し、即値読み出し、命令実行を行う手続きになります。

シミュレーションモジュール tut9 は、CPU をインスタンス tut9cpu として定義し、命令をあらかじめ書き込んだ主記憶 mainmem を構成要素として持ちます。CPU の二つの出立関数に対するメモリとしての振る舞い応答を行っています。

CPU は、内部カウンタレジスタ count が 10 となったところで、0 番地から命令実行を開始します。ここでは、Windows 版 NSL CORE を用いて、シミュレーション用の VerilogHDL を作成します。Language が Simulation となっていることを確認し、右下の Target に tu9 を記入した上で、tut9.nsl を VerilogHDL シミュレーションファイルに変換してください。その後、Icarus Verilog でコンパイル、シミュレーションを行います。

```
> iverilog -otut9.vvp tut9.v
> vvp tut9.vvp
VCD info: dumpfile tut9.vcd opened for output.
READ: ADDRESS:00, DATA:00
READ: ADDRESS:01, DATA:02
READ: ADDRESS:02, DATA:04
READ: ADDRESS:03, DATA:0a
READ: ADDRESS:04, DATA:02
READ: ADDRESS:05, DATA:20
WRITE: ADDRESS:20, DATA:02
READ: ADDRESS:06, DATA:00
READ: ADDRESS:07, DATA:ff
READ: ADDRESS:08, DATA:03
READ: ADDRESS:09, DATA:02
READ: ADDRESS:02, DATA:04
READ: ADDRESS:03, DATA:0a
READ: ADDRESS:04, DATA:02
READ: ADDRESS:05, DATA:20
WRITE: ADDRESS:20, DATA:01
READ: ADDRESS:06, DATA:00
READ: ADDRESS:07, DATA:ff
READ: ADDRESS:08, DATA:03
READ: ADDRESS:09, DATA:02
READ: ADDRESS:02, DATA:04
READ: ADDRESS:03, DATA:0a
READ: ADDRESS:0a, DATA:02
READ: ADDRESS:0b, DATA:ff
SIM STOP
```

```
WRITE: ADDRESS:ff, DATA:00
```

動作の状況を波形ビュアで見てみましょう。

```
> gtkwave tut9.vcd
```

図で見るように、3 つの手続き、ift,imm,exe が順次起動され、手続きに従って命令が実行されています。

ステートマシンと状態遷移

論理設計において、呼び出されたときに起動する手続きではなく、静的な状態変数を持つ回路を作りたい場合があります。NSL は、このようなケースに対応し、ステートマシンを作成する構文を持ちます。状態変数は、リセット時に初期化され、状態遷移によって、遷移先の状態に書き換えます。ステートマシンは、構成要素として、モジュール、複合文の中で定義できます。複合文で定義したステートマシンは、複合文の中だけの、局所的なステートマシンとなることに注意してください。ステートマシンの定義は、下記の構文で行います。先頭の状態が、ステートマシンの初期状態になります。

```
state_name 状態名 1, 状態名 2, 状態名 3, .... ;
```

ステートマシンを定義したら、各状態に対応する動作を記述します。動作の記述は、state 文を使います。

```
state 状態名 実行文
```

state 文の中で、他の状態に遷移する場合、goto 文を用います。シーケンスブロックの goto 文とは、全く異なることに注意してください。

```
goto 状態名 ;
```

次に、ステートマシンの例題を示します。

```
declare tut10 simulation {}

module tut10 {
   state_name state1, state2, state3;
   state state1 {
       reg c1[2]=0;
       if(c1++ == 3) goto state2;
       _display("in state1 %d",c1);
   }

   state state2 {
       reg c2[2]=0;
       if(c2++ == 3) goto state3;
       _display("in state2 %d",c2);
```

```
  }

  state state3 {
      reg c3[2]=0;
      if(c3++ == 3) {goto state1; _finish();}
      _display("in state3 %d",c3);
  }
}
```

状態変数は静的であり、state 文は、実行の条件がそろった上で、状態変数が指定した状態になっている場合に、実行を行うことに注意してください。たとえば、関数や手続き内にステートマシンを作成したとき、state 文を記述しても、その関数もしくは手続きが動作していなければ、状態変数が一致していても、実行は行われません。

それでは、実行してみましょう。ここでは、Windows 版 NSL CORE を用いて、シミュレーション用の VerilogHDL を作成します。Language が Simulation となっていることを確認し、右下の Target に tu10 を記入した上で、tut10.nsl を VerilogHDL シミュレーションファイルに変換してください。その後、Icarus Verilog でコンパイル、シミュレーションを行います。

```
> iverilog -otut10.vvp tut10.v
> vvp tut10.vvp

VCD info: dumpfile tut10.vcd opened for output.
in state1 0
in state1 0
in state1 1
in state1 2
in state1 3
in state2 0
in state2 1
in state2 2
in state2 3
in state3 0
in state3 1
in state3 2
in state3 3
```

シミュレーションで、最初に state1 の 0 の値が 2 行出ているのは、リセット時にもクロックが入力されていて、(_display 文など) リセット信号が入っていない回路が動作しているからです。

ステートマシンの利用上、注意していただきたい点があります。状態変数の初期化はリセットのときに行われますが、通常の動作中は行われないので、手続きに記述した状態を、手続きの呼出しごとに初期状態に戻したい場合、手続きの終了前に、初期状態に遷移する goto 文を記述する必要があります。

整数変数と一時変数および構造展開

NSL の多くの構文は、生成するハードウェアと一対一に対応しています。これは、ハードウェア設計者に見通しのよい言語を提供するためです。しかし、似たような構造を複数作成する場合には、記述量の圧縮のため、構造を展開する構文が便利に使えます。 NSL は、構造展開のために、整数変数、一時変数および構造展開構文を提供します。これらの構文は、コンパイル時に評価し、値が確定します。評価は、NSL 文の記述順に行われ、実行時の動作順序とは無関係となることに注意してください。

整数変数(integer)は、整数を記述できる部分に使えます。また、整数変数同士および整数の演算式を利用できます。ただし、整数と整数変数の演算は、左優先の優先順位となるので、演算を行う場合、明示的に括弧で優先順を指定してください。

integer 変数名 ;

特別なケースとして、if 文の条件に整数もしくは整数変数からなる式を指定した場合、コンパイル時に、真もしくは偽のいずれかの実行文のみ生成します。プリプロセッサによる条件コンパイルよりも、細やかな条件制御を可能としますが、多用すると、可読性の低下をもたらすことになるので、注意してください。

一時変数(variable)は、幅を持ち、端子と同様に用います。ただし、端子と異なり、同一クロックで複数回の転送が可能であり、かつ、部分的な転送も行えます。これは、一時変数は、転送が出現するたびに、新たな端子を自動作成し、必要な演算を行うからです。一時変数の評価は記述順に行い、展開後の端子への転送は並列に実行します。

variable 変数名[ビット幅] ;

整数変数と一時変数を用いて回路を生成するための、generate 文があります。

generate(整数変数=初期値 ; 整数変数式 ; 整数変数更新式) 実行文

次の例題は、generate 構文を使い、巡回符号による擬似乱数生成回路を作成しています。一時変数 v の各ビットへの値を generate 文を用いて設定し、生成した結果をまとめてレジスタ r に転送します。

```
declare glfsr {
   input seed[16];
   output q[16];
   func_in set_seed(seed);
   func_in next_rand : q;
}

module glfsr {
   reg r[16] = 0x39a5;
   variable v[16];
   integer i;
```

```
   func next_rand {
       generate (i=0;i<15;i++) {
           if((i == 13) || (i == 12) || (i == 10)) v[i] = r[i+1] ^ r[0];
           else v[i] = r[i+1];
           }
       v[15] = r[0];
       r:=v;
       return r;
   }

   func set_seed r:=seed;
}

declare tut11 simulation {}

module tut11 {
   glfsr rmod;
   reg count[16]=0;
   count++;
   any {
       3'(count) == 7: {
           _display("set seed:%d",count+0x9876);
           rmod.set_seed(count+0x9876);
       }
       count==10:    _finish("finished");
       else:    _display("random generate %d", rmod.next_rand());
   }
}
```

この例題を実行すると、下記のようになります。ここでは、Windows 版 NSL CORE を用いて、シミュレーション用の VerilogHDL を作成します。Language が Simulation となっていることを確認し、右下の Target に tu11 を記入した上で、tut11.nsl を VerilogHDL シミュレーションファイルに変換してください。その後、Icarus Verilog でコンパイル、シミュレーションを行います。

```
> iverilog -otut11.vvp tut11.v
> vvp tut11.vvp

VCD info: dumpfile tut11.vcd opened for output.
random generate 14757
random generate 14757
random generate 43218
random generate 21609
random generate 40500
random generate 20250
random generate 10125
random generate 42950
```

```
set seed:39037
random generate 39037
random generate 63550
finished
```

パラメータ

パラメータには整数パラメータ(param_int)と、文字列パラメータ(param_str)があります。NSL では、パラメータは二つの使い方をします。一つ目は、構造展開を行う時の制御変数としての使い方です。制御変数として用いる場合には、パラメータに値を設定します。この場合、パラメータは、モジュールの外に定義し、大域変数扱いになります。

```
param_int パラメータ名 = 整数;

param_str パラメータ名 = 文字列;
```

if 文の条件式において、パラメータ名を用いて比較演算を行うことができます。整数パラメータの比較は、整数変数と同一の演算子を用います。文字列の比較には、一致（==）と不一致（!=）が利用可能です。

```
param_str MODE = "SIM";

module cpu {
 if(MODE == "SIM") {
   _display("PC: %4X", pc);
 }
```

他の使い方として、NSL 以外の言語（VerilogHDL や VHDL）で作成されたモジュールへのパラメータ設定があります。他言語のモジュールをインスタンスとして利用する場合に、場合によって、それらのモジュールに指定されたパラメータを与える必要があります。このため、他言語のモジュールのプロトタイプを指定する declare 文において、パラメータを宣言することができます。

```
param_int パラメータ名;
param_str パラメータ名;
```

サブモジュールのインスタンス宣言において、パラメータに値をセットすることで、生成された VerilogHDL もしくは VHDL にパラメータを渡します。

```
モジュール名 サブモジュールインスタンス名(パラメータ名=値, パラメータ名=値) ;
```

初期値を設定した整数パラメータと、文字列パラメータは、整数もしくは文字列として、if 文の構造展開の条件式に用いることができます。

次の記述は、Xilinx 社の FPGA のクロックモジュール(DCM)とグローバルバッファ(BUFG)を用いて、クロック配線を作る例題です。DCM には、ユーザが与えなくてはならないパラメータがあり、これを NSL から設定しています。この例題は、論理合成専用の例題で、一部のモジ

ュールは declare しかないしシミュレーション実行部分がないので、実行して確かめることはできませんが、記述方法の参考にしてください。

```
declare DCM interface {
param_int CLKDV_DIVIDE;
param_str CLK_FEEDBACK;
input RST, PSINCDEC, PSEN, PSCLK, CLKIN, CLKFB;
output PSDONE, CLK0, CLK90, CLK180, CLK270,
 CLK2X, CLK2X180, CLKDV, CLKFX, CLKFX180,
 LOCKED, STATUS[8];
}
declare BUFG interface {
input I;
output O;
}

declare sample {
input samplein;
output sampleout;
}

declare tut12 {}
module tut12 {
BUFG buff;
DCM dcm2(CLKDV_DIVIDE=4, CLK_FEEDBACK="1X");
sample target;
dcm2.RST = p_reset;
dcm2.CLKIN = m_clock;
dcm2.CLKFB = dcm2.CLK0;
dcm2.PSEN = 0;
dcm2.PSCLK = 0;
dcm2.PSINCDEC =0;
buff.I = dcm2.CLKDV;
target.m_clock = buff.O;
target.p_reset = p_reset;
}
```

浮動小数点加算器例題

IEEE754 浮動小数点形式を入力とする浮動小数点加算器の例題を示します。この加算器はパイプライン構造となっていて、1クロックごとに演算結果を出力することができます。

```
/*
  IEEE 754 type Single Precision Floating Point Pipeline Adder
  Copyright (c) 2011 Naohiko Shimizu, IP ARCH, Inc.

  This circuit is provided only for demonstration of NSL description.
```

```
*/

/*
  Adder interface declaration.
  We will have two inputs 'a' and 'b' those must be normalized
  single precision IEEE754 numbers.
  To start the circuit, you can invoke 'exe' with two arguments.
  When the circuit finish the calculation, it will send the result
  with done signal.
  The circuit is completely pipelined.
  And you can invoke it in every cycle.
*/

declare Ieee754SpAdd {
     input a[32];
     input b[32];
     output result[32];
     func_in exe(a,b);
     func_out done(result);
}

/*
   The struct Ieee754Sp defines single precision format of IEEE754.
   It has Sign bit, 8bit Exponent, 23bit Mantissa.
*/

struct Ieee754Sp {
     Sign;
     Exponent[8];
     Mantissa[23];
};
```

```
/*
  We will use Barrel Shifter for pre-shift the mantissa.
  Though the required length is not 32bit, we will use 32bit shifter.
  Logic synthesizer will do the compaction of unused net.
*/

declare BarrelShift {
     input a[32];
     input sa[8];
     output f[32];
     func_in exe(a,sa):f;
}

/*
  After the mantissa calculation, we need suppress leading zeros
  of the result.
  LeadingZeroShift provide the function.
  It will return the result and shifted amount.
*/

declare LeadingZeroShift {
     input a[32];
     output shamt[8];
     output f[32];
     func_in exe(a):f;
}

/*
  Barrel Shifter body.
  We will decode each bit of shift amount and
  determine to shift the contents or not.
*/

module BarrelShift {
     wire t0[32],t1[32],t2[32],t3[32],t4[32];
     func exe {
          if(sa[0]) t0=a>>1;
          else      t0=a;
          if(sa[1]) t1=t0>>2;
          else      t1=t0;
          if(sa[2]) t2=t1>>4;
          else      t2=t1;
          if(sa[3]) t3=t2>>8;
          else      t3=t2;
          if(sa[4]) t4=t3>>16;
          else      t4=t3;
          if(sa[7:5]==0)return t4;
```

```
          else      return 0;
     }
}

/*
  Leading Zero Shifter body.
  We will evaluate continuous zero from MSB to LSB.
*/

module LeadingZeroShift {
     wire t0[32],t1[32],t2[32],t3[32],t4[32];
     wire a0,a1,a2,a3,a4;

     func exe {
          if(a[31:16]==0)         { a4 = 1; t0 = a<<16; }
          else           { a4 = 0; t0=a; }
          if(t0[31:24]==0)    { a3 = 1; t1 = t0<<8; }
          else           { a3 = 0; t1=t0; }
          if(t1[31:28]==0)    { a2 = 1; t2 = t1<<4; }
          else           { a2 = 0; t2=t1; }
          if(t2[31:30]==0)    { a1 = 1; t3 = t2<<2; }
          else           { a1 = 0; t3=t2; }
          if(t3[31]==0)       { a0 = 1; t4 = t3<<1; }
          else           { a0 = 0; t4=t3; }
          shamt = 8'({a4,a3,a2,a1,a0});
          return t4;
     }

}

/*
  Floating Adder body.
  It has 4 stages 'stageA' through 'stageD.'
  At 'stageD' we will get the result.
  exe:    invoke the circuit and calculate exponents differences.
  stageA: pre-shift for adder operation.
  stageB: Mantissa addition.
  stageC: Leading zero shift.
  stageD: Return result.
*/

module Ieee754SpAdd {
/*
  Resources defenitions.
*/
     BarrelShift bshft;
     LeadingZeroShift lzshft;
```

```
/*
  stageA resources
*/

     reg Aexdf[8];
     Ieee754Sp reg x, y;
     proc_name stageA(Aexdf, x, y);

/*
  stageB resources
*/

     reg Bm1[32], Bm2[32], Bs1, Bs2, Bexp[8];
     proc_name stageB(Bm1, Bm2, Bs1, Bs2, Bexp);

     wire s1, s2, x1[32], x2[32], r1[32];
     func_self madd(s1, s2, x1, x2) : r1;

/*
  stageC resources
*/

     reg Cm[32], Cs, Cexp[8];
     proc_name stageC(Cm, Cs, Cexp);

/*
  stageD resources
*/

     Ieee754Sp reg z;
     proc_name stageD(z);

/*
  func exe(a,b) is the starting point of this adder.
*/

     func exe {
          wire wdiff[9];

/*
          In IEEE 754, the exponent is biased binary.
          Therefore, negative value of subtraction will
          show that 'b.Exponent > a.Exponent.'
          We will select pre-shift argument depending on the result.
*/
          wdiff = 9'((Ieee754Sp)(a).Exponent) - 9'((Ieee754Sp)(b).Exponent);
```

```
        if(wdiff[8]) {
            stageA( -wdiff[7:0], b, a );
        }
        else {
            stageA(  wdiff[7:0], a, b );
        }
    }

/*
    proc_name stageA(Aexdf, x, y);

    We will pre-shift 'y' for addition.
    Because IEEE754 suppress MSB's '1' in mantissa, we will add it here.
    But if the exponent part is zero, it may be un-normalized value.
    Therefore, we will not add '1'.
*/
    proc stageA {
        wire xmsb[3],ymsb[3];

        if(x.Exponent==0)
            xmsb=0;
        else
            xmsb=3'b1;

        if(y.Exponent==0)
            ymsb=0;
        else
            ymsb=3'b1;

        stageB( {xmsb,x.Mantissa,6'b0},
            bshft.exe({xmsb,y.Mantissa,6'b0},Aexdf),
            x.Sign,
            y.Sign,
            x.Exponent );
    }

/*
    proc_name stageB(Bm1, Bm2, Bs1, Bs2, Bexp);

    Add two mantissa values.
    In IEEE754, the mantissa does not have sign it self.
    Therefore, if the addition shows negative value,
    we will make 2's complement of it.
*/

/*
    s1,s2 is sign bit for x1,x2.
```

```
     When the value is negative, we will make 2's complement.
*/
     func madd {
          return (32#s1^x1) + 32'(s1) + (32#s2^x2) + 32'(s2);
     }

     proc stageB {
          wire m3[32];

          m3 = madd(Bs1,Bs2,Bm1,Bm2);
          if(m3[31])
               stageC( -m3, m3[31], Bexp );
          else
               stageC(  m3, m3[31], Bexp );
     }

/*
     proc_name stageC(Cm, Cs, Cexp);

     We now have mantissa 'Cm', sign 'Cs', exponent 'Cexp.'
     But we need to suppress leading zero of mantissa for normalize.
     If the result of mantissa calculation was 0, we will return 0.
     Or if the exponent was 0, it shows un-normalized value,
     then we will not do the leading zero shift.
*/
     proc stageC {
          if(Cm==0)
               stageD(0);
          else if(Cexp==0)
               stageD({Cs,Cexp,Cm[28:6]});
          else
               stageD( {Cs,
                    8'(Cexp - lzshft.shamt + 1),
                    lzshft.exe({Cm[30:0], 1'b0})[30:8]
                    } );
     }

/*
     proc_name stageD(z);
     return ack signal and the result from adder.
*/
     proc stageD {
          done(z);
          finish;
     }
}
```

チュートリアルのまとめ

NSL の言語仕様を例を交えて説明しました。50 行弱で記述する小さな CPU を含め、NSL は、コンパクトに見通しよく、設計者がまさに実現したい回路を書き下ろせます。また、既存の VerilogHDL や VHDL の資産を、そのまま生かして回路を開発できるため、今までの投資を無駄にしません。

次世代のハードウェア設計言語 NSL を活用し、すばらしい設計をしてください。