NSL リファレンスマニュアル

Ver. 1.5.1

2016/07/08

オーバートーン株式会社

1 NSL 基本要項

1.1 用語解説

構文

NSL の文法に則って記述された文のこと。

モジュール (module)

入出力、内部構成が定義され、まとまって何らかの機能を担う動作記述

動作 (Action)

1 クロックで動作する部分のこと。

動作記述

モジュール内部の、動作を表している要素のこと。

共通動作記述

動作記述のうち、プロシージャ、ファンクション以外で、クロック毎に常に動作している部分のこと。

単位動作 (Atomic Action)

モジュールの動作記述部分で記述される、動作の最小単位のこと。

プロシージャ (proc : procedure)

動作を集約した処理手続きのこと。

プロシージャは起動すると、明示的に終了するかプロシージャ内から他のプロシージャを起動するまで、次クロック以降も動作を継続する。

ファンクション (func : function)

動作記述をある単位で集約して、モジュール外部や内部から制御端子を通じて呼び出すことができる処理のまとまりのこと。

プロシージャと違い、一連の処理を完了すると同時にファンクションは終了する。

インスタンス (instance)

サブモジュール構文において、下位となるモジュールを上位モジュール内で宣言した際の"実体"のこと。

入出力構成要素

モジュールへの入力またはモジュールからの出力を構成する要素。

内部構成要素

内部端子、レジスタやプロシージャなどモジュール自身を構成している要素。

端子

名前の付いたネット(配線)のこと

内部端子

モジュール内部の演算処理のために参照や値の転送を行う端子

外部端子

モジュール内外を接続するための端子

制御端子

モジュール内外に存在するファンクションを起動させるための端子のこと。

レジスタ (register)

電子回路や HDL 等で使用する記憶素子のこと。演算の補助や、値の記憶などに使用される。

メモリ

レジスタを配列とすることで、複数のデータを保存できるようにしたもの。

ブロック (block)

動作記述中で、{}により囲まれた部分のこと。 par,any,alt,if,seq などの種類があり、それぞれのブロックが別々の動作をする。

MSB/LSB (Most Significant Bit/Least Significant Bit)

NSL では左端を MSB(最上位ビット)、右端を LSB(最下位のビット)とする。

コンパイル、コンパイル操作

NSL 変換エンジンを使用し、NSL 言語ソースから下位言語ソースを生成することを指す。またはその操作。

1.2 値

NSL 言語では、それぞれの要素は以下の値をとることができます。

データ端子	“1”, ”0”, “Z”(Hi-Z), “U”(Undriven), “X”(不定)
制御端子	“1”, ”0”
レジスタ、メモリ	“1”, ”0”, “U”(Undriven)

1.3 ビット幅

多ビットのレジスタや端子は左端を最上位ビット（MSB）、右端を最下位ビット（LSB）として定義し、LSB ビット 0 とします。
例えば 6 ビットの信号線にデータ″101010″を出力した場合、ビット 0 は 0、ビット1は 1、ビット 2 は 0、ビット 3 は 1、ビット 4 は 0、ビット 5 は 1 となります。

また束線の任意のビットを切り出して他の端子に転送したり、束線の特定ビットのみを見て次の動作を決定したりといった処理も可能です。

ただし、例えば 4 ビットのレジスタを用意して、2 ビット目だけに 1 を書き込むといった、束線の特定ビットへの書き込みはできません。

1.4 数値表記法

NSL での数値表記は、VerilogHDL 型と、C 言語型の 2 種類があります。
VerilogHDL 型の表記方法では

<ビット幅 >'<基数を表す文字 ><値 >

と表記します。
例えば、10 進数の 12 を 4 ビット幅の 2 進数で表す場合は、

4'b1100

と表します。″b″は 2 進数を示します。
また、例として 4 ビット、10 進数の 5 を表記する場合は、

4'd5

となります。このように VerilogHDL 型において基数は、

2 進数:b、8 進数:o、10 進数:d、16 進数:h

とそれぞれ表記します。

また、C 言語型の表記法としては

0<基数を表す文字 ><値 >

と表記します。例えば、10 進数の 8 を 4 ビット幅の 2 進数で表す場合は、

0b1000

と表記します。この C 言語型の表記方法では

2 進数:b、16 進数:x
とそれぞれ表記します。

Verilog HDL 型では、"値"をビット幅に合わせて表記する必要はありませんが、 C 言語型ではビット幅は表記した "値"の幅で決定します。
例えば 0x00 では 8 ビットに、0b0000 では 4 ビットに決定されます。

値の表現に "_"(アンダースコア)を使うことができます。アンダースコアはコンパイル時には無視されます。多桁の数値表現の際に可読性を上げるために使うことができます。

例) 8'b01011010 → 8'b0101_1010
32'hA9876543 → 32'hA987_6543
0x12345678 → 0x1234_5678

1.5 演算時のビット幅推測

NSL における演算には、ビット幅の確定した値のみが許されます。例外として同一ビット数同士に演算オペランドが制約される一部の演算では、演算の第 2 項に整数 (および整数変数)を用いることが許されます。これは、NSL 処理系がビット数を推定し、ビット幅の確定した値に変換するからです。
同様な推定は信号、レジスタへの値の転送、レジスタ、メモリの初期値の設定、メモリアドレス指定においても行なわれるます。

ビット数が推定できる場合は、式の項に整数を使うこともできます。

- ✓ 端子やレジスタへの転送の右辺(proc や func_xxx の実引数を含む)は、転送先のビット幅が確定しているので、整数もしくは integer 変数を当該ビット幅の定数に変換して転送します。
- ✓ +,-,&,|,^の演算では、第 1 項のビット幅によって第 2 項のビット幅を推定します。そこで、第 2 項には、整数もしくは integer 変数が利用可能です。
  ただし、第 1 項、第 2 項ともに整数もしくは integer 変数の場合には、整数演算として扱い、結果はビット幅を持たない整数となります。
- ✓ if () else の条件演算は、その記述場所においてビット幅が確定している場合(端子やレジスタへの転送の右辺や上記同一ビット幅同士の演算の第 2 項)には、真、偽の値ともに整数もしくは integer 変数が利用可能です。

1.6 モジュール名や信号名に利用可能な文字

NSL ではモジュール名や信号名などの識別子に以下の文字が利用可能です。

- ✓ 半角アルファベット
- ✓ 数字 (ただし 2 文字目以降)
- ✓ アンダースコア "_" (ただし 2 文字目以降)

また、アンダースコアの二重表記"_"は禁止しています。

1.7　コメント

NSL のコメントは C 言語互換になっており、2 種類の表現が用意されています。
シングルラインコメントは

//コメント

エリアコメントは

/* コメント */

と記述します。
また、エリアコメント内にシングルラインコメントを記述することはできますが、エリアコメントのネストはできません。

1.8　一文の終了

NSL では、記述中において一文の終了をセミコロン″;″で表します。
要素の宣言や、動作の記述など、記述の最小単位である一文を終了する際には、

構成要素の宣言；
値の転送（動作の記述）;

のように、セミコロンを使用して終了を確定します。
同一行にセミコロンで区切って複数の記述を行うことは可能です。

1.9　クロック信号、リセット信号について

NSL はモジュールのクロック信号を自動で用意して単相クロックで動くモジュールを作成します。またクロック信号を用意すると同時に、回路のリセット信号も用意します。
何も指定しない場合、クロック信号は″m_clock″、リセット信号は″p_reset″という名前で自動合成しますが、リセット信号名・クロック信号名ともにコンパイルオプションで名前を変更することができます。

1.10 NSL 処理系の処理順序

NSL 処理系では、

- ✓ プリプロセッサによるディレクティブの展開
- ✓ 構造展開
- ✓ 構成要素による回路記述の合成

の 3 段階で処理を行います。
プリプロセッサによるディレクティブの展開は付録 1 で解説しています。

構造展開は回路記述の要素と関係なく、記述された順番に展開される構文です。
構造展開を利用することにより、同じような回路を複数生成する際の記述量を大幅に減らすことができます。詳細は第 6 章で解説します。
整数(および整数変数)は構造展開で値を決定します。

1.11 演算子

NSL の演算子は基本的には VerilogHDL 互換ですが、関係演算の一部及び除算を除いてあります。
また NSL 独自の演算としてビット幅拡張演算があります。表 1-1 に NSL で使用できる演算子を挙げます。
（以降、表中で使われるセミコロンは NSL 内で使われる構文とその意味を分ける記号として使用します。）

表 1-1　演算子

ビット演算

&	ビット毎の論理積
|	ビット毎の論理和
^	ビット毎の排他的論理和
~	ビット毎の論理否定

算術演算

+	加算
−	減算
*	乗算
++	インクリメント (*)
−−	デクリメント (*)

シフト演算

>>	右シフト
<<	左シフト

関係演算

==	等しい
!=	等しくない
>	左辺が大
<	左辺が小
>=	左辺が大または等しい
<=	左辺が小または等しい

論理演算

!	論理否定
&&	論理 AND
||	論理 OR

リダクション演算 (**)

&	AND
~&	NAND
|	OR
^|	NOR
^	EX-OR
~^	EX-NOR

その他

num#(sig)	符号付きビット幅拡張
num'(sig)	ビット幅指定(符号なしビット幅拡張)
sig[num]	ビット切り出し
sig[numA:numB]	ビット切り出し
numA{numB}	リピート演算
{sigA,sigB,,sigX}	ビット連結
if(条件文) sigA else sigB	条件演算

(*)インクリメント、デクリメントの結果が反映するのは、次のクロックです。

(**)リダクション演算は複数ビットの信号に対してのみ使用可能です (1 ビットの信号に対しては使えません)。

また、NSL の演算子には表 1-2 のとおりの優先順位が存在します。

一文の式中に複数の演算子を記述した場合は、演算子の優先順位の高いほうから順に演算が行われます。

式の優先順位を高くするには式中で"()"を使います。

表 1-2　**演算子優先順位**

優先順位	演算子	説明	例
1(高)	{sigA, sigB, …,sigX}	ビット連結	{0b0, 0b1, 0b01} → 0b0101
	numA{numB}	リピート演算	4{0b0} → 0b0000
	num#(sig)	符号付きビット幅拡張	8#(0b1101) → 0b11111101
	num'(sig)	ビット幅指定 (符号なしビット幅拡張)	8'(0b1101) → 0b00001101
	&	リダクション AND	&0b1111 → 1, &0b0111→0
	|	リダクション OR	|0b0000 → 0, |0b0111→1
	^	リダクション EX-OR	^0b1111 → 0, ^0b0111→1
	~	ビット演算 NOT	~0b0101 → 0b1010
	!	論理 NOT	if (!a)
2	*	算術乗算	q=a*b
3	+	算術加算	q=a+b
	-	算術減算	q=a-b
4	<<	左シフト	q=4'b1010, (q<<1)=4'b0100
	>>	右シフト	q=4'b1010, (q>>1)=4'b0101
5	<=	以下	if (a<=b)
	>=	以上	if (a>=b)
	<	小なり	if (a<b)
	>	大なり	if (a>b)
6	==	等しい	if (a==b)
	!=	等しくない	if (a!=b)
7	&	AND	a=4'b0110, b=4'b1100, a&b=4'b0100
	^	EX-OR	a=4'b0110, b=4'b1100, a^b=4'b1010
8	|	OR	a=4'b0110, b=4'b1100, a|b=4'b1110
9	&&	論理 AND	if (a&&b)
10	||	論理 OR	if (a||b)
11(低)	if ( ) else	条件演算	q=if (a>b) a else b

++(インクリメント)、 --(デクリメント)の結果が反映するのは次のクロックのため、この表からは除外してあります。

1.12 整数を含む演算の優先順位

前述の通り整数の値は構造展開時に決定しますが、式の値のビット幅を決定するため同一ビット幅同士で行われる演算(*)では第 1 項目を信号、第 2 項目を整数にする必要があります。3 項以上の場合も先頭 2 項を見て判断しますが、カッコ()でくくられた式がある場合は、そちらを優先して判断材料とします。

(*) +, −, <, <=, >, >=, ==, !=, |, ^, &の各演算子による演算

1.13 整数および整数変数のみの演算の優先順位

整数および整数変数のみの演算は、**演算子の優先順位とは関係なく左から演算します**。演算順序が関係する部分ではカッコ()を使うようにしてください。

例

```
integer    i;
variable   v[8];
i = 2 + 3 * 4 ;
v = i ;
```

この場合、v には 14 ではなく 20 が入ります。

2 基本構造

NSL の記述の基本構造は表 2-1 の体系で構成しています。

表 2-1 NSL 基本構造

```
< struct 構造体名{
  < 構造体メンバリスト >
} ; >

declare モジュール名 < interface > < simulation > {
  < パラメータ宣言リスト >
  < 入出力構成要素宣言リスト >
}

module モジュール名 {
  < 内部構成要素宣言 >
  < 動作記述 >
    -< 共通動作記述部分>
    -< ファンクション記述部分>
    -< プロシージャ記述部分>
}
```

NSL の回路は1つ以上のモジュールから構成されます。各モジュールは、入出力の仕様を記述する declare 文と、動作本体を記述する module 文から構成します。

declare 文では"入出力構成要素の宣言 "を行います。
"入出力構成要素の宣言 "は第 2 章で解説します。

module 文では "内部構成要素の宣言 "と "動作記述 "を行います。
"内部構成要素の宣言 "は第 2 章で、"動作記述 "は第 3 章以降で解説します。

declare 文と module 文は必ずしも続けて書く必要はありません。コンパイル時に両方セットで同一ファイル内に存在すれば、コンパイルは認められます。
そのため、declare 文のみをまとめて別ファイルとし、ヘッダファイルとして include することも可能です。

declare 文には interface または simulation という修飾子を付けることができます。
interface 修飾子は、サブモジュールとして呼び出すモジュールに対し、クロックとリセットの暗黙裡の生成を抑止し、明示的にクロックとリセットの信号名を指定したい場合に使います。
interface 修飾子のない declare 文記述が行われた場合は、クロックとリセットにそれぞれ m_clock、p_reset という信号名が使われます。詳細は第 9．1章を参照してください。
simulation 修飾子は、モジュール内でシミュレーションのためのシステムタスクを使う場合に使います。詳細は9．2章を参照してください。

複数の信号をまとめて扱う構造体の表記を使用する場合は、構造体の宣言を declare 文の前に記述します。構造体の詳細については8章に示します。

2.1 入出力構成要素の宣言

入出力構成要素とは、モジュールを外部と接続するための端子のことです。
NSL における入出力構成要素を以下の表 2-2 に示します。

表 2-2 入出力構成要素

input	データ入力端子
output	データ出力端子
inout	データ入出力端子
func_in	制御入力端子
func_out	制御出力端子

端子名をカンマ ″,″ で区切って複数記述することで、一度に複数の端子を宣言できます。

2.1.1 データ入力端子 /データ出力端子の宣言

データ入力端子/データ出力端子はそれぞれモジュールの入力方向/出力方向に向けられたデータ端子のことです。
データ入力端子、データ出力端子ともに、信号名の後ろに [ ]で囲んで ″ビット幅 ″を指定できます。ビット幅を省略した場合は、1 ビットの端子です。
データ入力端子/データ出力端子は以下のように宣言します。

input 入力信号名 [ビット幅]
output 出力信号名 [ビット幅]

2.1.2 データ入出力端子の宣言

データ入出力端子は、モジュールへのデータ入力と、モジュールからのデータ出力が可能な端子です。
データ入出力端子のビット幅もデータ入力端子/データ出力端子の宣言と同じようにビット幅を指定できます。
データ入出力端子は以下のように宣言します。

inout 入出力信号名 [ビット幅]

2.1.3 制御入力端子 /制御出力端子の宣言

制御端子とはモジュール内に存在する″ファンクション″をモジュール内部や外部から起動するための端子です。

制御端子には、モジュール外からモジュール内のファンクションを起動するために使う制御入力端子と、モジュール外のファンクションを起動させるために使う制御出力端子があります。制御端子にビット幅を設定することはできません。
制御端子を起動すると同時に、データ端子に信号を引き渡すための引数が指定できます。

制御端子の宣言で仮引数に指定されたデータ端子には、制御端子の起動の時、その動作記述で実引数として指定された式の信号が転送されます。

制御入力端子の宣言の仮引数にはデータ入力端子かデータ入出力端子を指定します。制御出力端子の宣言の仮引数にはデータ出力端子かデータ入出力端子を指定します。

仮引数はカンマ ″,″ で区切って複数個指定できます。

func_in　**制御入力信号名**　（<**仮引数** >, <**仮引数** >, <**仮引数** >, … ）
func_out　**制御出力信号名**　（<**仮引数** >, <**仮引数** >, <**仮引数** >, … ）

この時、制御入力信号に付帯する仮引数はデータ入力端子に限られ、制御出力信号に付帯する仮引数はデータ出力端子に限られます。
言語仕様上は、引数を経由せず、制御端子の起動と同時に行うデータの転送を直接記述しても構いませんが、可読性を上げるため引数を利用することを推奨します。

制御端子の宣言は、戻り値を返すための端子を持つことができます。制御入出力端子の場合、戻り値端子はデータ入出力端子になりますが、制御端子と戻り値端子の方向は逆になることに注意してください。
戻り値端子を持つファンクションの記述方法は以下の通りです。

func_in　**制御入力信号名**（<**仮引数** >, <**仮引数** >, <**仮引数**>, … ）: **戻り値出力端子**（**または入出力端子** ）
func_out　**制御出力信号名**（<**仮引数** >, <**仮引数** >, <**仮引数**>, … ）: **戻り値入力端子**（**または入出力端子** ）

制御入力端子、制御出力端子の動作記述は第5章を参照してください。

入出力構成要素の宣言例を示した記述例 2-1 に示します。

記述例 2-1　　入出力構成要素の宣言例

```
declare    test_inout  {
   input    a;                    //データ入力端子 a を 1 ビットで宣言
   output   b[4];                 //データ出力端子 b を 4 ビットで宣言
   inout    c[12];                //データ入出力端子 c を 12 ビットで宣言
   func_in  d;                    //制御入力端子 d を宣言
   func_in  e(a);                 //制御入力端子 e を、仮引数 a を設定して宣言
   func_out f(b);                 //制御出力端子 f を、仮引数 b を設定して宣言

   input    reti[8];              //戻り値端子用にデータ入力端子を 8 ビットで宣言
   output   reto[8];              //戻り値端子用にデータ出力端子を 8 ビットで宣言
   func_in  g:reto;               //制御入力端子 g を、戻り値 reto を設定して宣言
   func_out h(b):reti;            //制御出力端子 h を、仮引数 b と戻り値 reti を設定して宣言
}
module     test_inout{
           //内部構成要素を記述
           //動作を記述
}
```

このように入出力構成要素の宣言を行うことにより、a, b, c, d, e, f, g, h の各信号を後述の動作記述で使用することが可能になります。

2.2 内部構成要素の宣言

内部構成要素は端子、レジスタ、制御内部端子、状態変数、プロシージャ、メモリ等の構成要素を指したものです。
内部構成要素宣言を行うと、ソースリストの宣言以降の動作記述内で内部構成要素を使用することができます。(宣言より前では使えません)
内部構成要素名を複数記述してカンマ ″,″で区切って、一度に複数の内部構成要素を宣言することができます。

2.2.1 内部端子の宣言

内部端子は、モジュール内部のネット(配線)に名前をつけたものです。
内部端子の宣言は ″wire″で、以下のように定義します。

wire　　内部端子名　　[ビット幅] ;

内部端子に転送された値は、そのクロックサイクル中のみ有効です。内部端子への転送が行われないサイクルでは、値は不定です。
内部端子は自然数 (1,2,3,)をビット幅として設定可能で、ビット幅は省略することも可能です。

2.2.2 レジスタの宣言

レジスタはクロックに同期して直前の入力値を記憶する記憶素子です。レジスタに値を転送すると次クロック以後、値が保持されます。
レジスタは任意の自然数 (1,2,3,)のビット幅を指定できます。宣言時に初期値を設定することも可能です。指定が無い場合、初期値は ″X″(不定)です。レジスタの宣言は以下のような方法で行います。

reg　　レジスタ名　　[ビット幅] = <初期値 > ;

ここで、内部端子の宣言とレジスタ宣言を記述例 2-2 に示します。

記述例 2-2　　内部端子の宣言 /レジスタ宣言の使用例

```
declare  test_reg {
         // 入出力構成要素の記述
}
module   test_reg {
         wire     wire_a  [16] ;     // 内部端子wire_a を16 ビットで宣言
         reg      reg_b [4] ;        // レジスタreg_b を4 ビットで宣言
         reg      reg_c, reg_d, reg_e ; // レジスタreg_c,reg_d,reg_e を一度に宣言
         reg      reg_f [4] = 4'b1010 ; // レジスタreg_f を4 ビットで、初期値1010 で宣言

         // 動作の記述
}
```

記述例 2-2 のように、内部端子の宣言とレジスタ宣言を記述することで、後述の動作記述で wire,reg をそれぞれを使用することが可能になります。

2.2.3 制御内部端子の宣言

制御内部端子はモジュール内部のファンクションを起動する制御端子です。
制御内部端子には複数の仮引数と一つの戻り値端子を持たせることが可能です。
制御内部端子の宣言方法は以下の通りです。

func_self　制御内部端子名　;

また仮引数を持たせる場合の宣言方法は以下のように記述します。

func_self　制御内部端子　　(<仮引数 >,<仮引数 >,<仮引数 >, …) ;

制御内部端子の仮引数は内部端子の宣言で定義された wire のみ可能です。
可読性を向上させるため、仮引数を書くことを推奨します。(省略することもできます)
戻り値端子も宣言する場合は、以下のように記述します。

func_self　制御内部端子名　:戻り値端子名 ;
func_self　制御内部端子　　(<仮引数 >,<仮引数 >,<仮引数>, …) : 戻り値端子名 ;

戻り値端子は、内部端子で宣言されている wire のみ使用可能です。

次の記述例 2-3 が、制御内部端子の宣言例です。

記述例 2-3　　制御内部端子の宣言使用例

```
declare  test_func_self {
         // 入出力構成要素の記述
}
module   test_func_self {
         wire              a [4], b[2], r[8] ;
         func_self         funcC(a,b) : r ;
         // 動作の記述
}
```

このように制御内部端子の宣言を行うことで制御内部端子の動作を記述することができます。

2.2.4 サブモジュールの宣言

モジュール内に、別のモジュールを組込むことができます。
上位−下位という構造になることから、下位側のモジュールを「サブモジュール」と呼びます。

module 文中でサブモジュールを利用するには、サブモジュールの declare 文がソースファイル内のその場所より以前に必要です（定義は無くても構いません）。

サブモジュール構文は以下の方法で宣言します。

サブモジュール名　インスタンス名 ;

また、インスタンス名をカンマで区切って複数記述することにより一度に実体化させることも可能です。
以下に例を示します。

サブモジュール名　インスタンス名 1,　インスタンス名 2,　インスタンス名 3, …　;

さらに、インスタンス名に[ ]でインスタンス数を付けるという方法でもテンプレートを複数実体化する(多重度を持たせる)こともできます。

サブモジュール名　インスタンス名 [個数] ;
添え字の数がインスタンスの数になるので、[]内には自然数(1,2,3,)のみ設定することができます。

記述例 2-4　　サブモジュールの宣言例

```
//サブモジュール"test_sub"
declare   test_sub  {
          input     a ;
          output    f ;
}

module    test_sub  {
          // 動作の記述
}

// 上位モジュールとなる"test_module"
declare   test_module         {
          input     test_in   ;
          output    tset_out  ;
}

module    test_module         {
          test_sub            SUB         ;         // テンプレートのモジュール"test_sub" を
                                                    // test_module 内で"SUB" という名前で実体化して定義
          test_sub            SUB1,SUB2,SUB3;       // SUB1,SUB2,SUB3 の3 つのモジュールを一度に実体化
          test_sub            SUB_Array[3];         // SUB_Array[0],SUB_Array[1],SUB_Array[2] の3 つの
                                                    //モジュールを 一度に実体化
          // 動作の記述
}
```

記述例 2-4 のように記述することにより、モジュール内でサブモジュールを使用できます。
また上位モジュールからサブモジュールのパラメータを操作する場合、第3章で解説したパラメータ構文を使用します。
サブモジュールのパラメータ構文を使用した例を以下の記述例 2-5 に示します。

記述例 2-5　　サブモジュール宣言におけるパラメータ使用例

```
declare   test_sub {
          param_int           INT ;
          param_str           CHA ;
          input     a ;
          output    f ;
}
module    test_sub {
          // 動作の記述
}

declare   test_module {
          input     test_in ;
          output    tset_out ;
}
module    test_module {
          test_sub SUB1 ;                 // テンプレートのモジュール"test_sub" を
                                          // test_module 内で"SUB1" というインスタンスで定義
          test_sub SUB2(INT = 14) ;       // インスタンス"SUB2" のパラメータ"INT" に"14" を渡す
          test_sub SUB3(CHA = "NEKO") ;   // インスタンス"SUB3" のパラメータ"CHA" に
                                          //文字列"NEKO" を渡す
          // 動作の記述
}
```

このように記述することで各インスタンス "SUB1","SUB2","SUB3",に違うパラメータを渡すことが可能です。
つまり同じ構造を持ったインスタンスでも、生成時に様々なデータ、状態を持たせて開始することができます。
サブモジュール構文の動作記述に関しては第 4. 8章を参照してください。

2.2.5 プロシージャの宣言

プロシージャは状態遷移や、パイプライン、順序回路を用いた制御を提供する構文で、共通動作記述以外でプロシージャ専用の動作を記述する領域を持ちます。
プロシージャは一度起動すると他のプロシージャに遷移するか、プロシージャの終了を宣言するまで動作し続けます。

プロシージャを宣言するには、以下の様な記述方法を行います。宣言時に仮引数を付帯することも可能で、カンマで区切って複数の仮引数を与えることもできます。

proc_name　　プロシージャ名　　(<仮引数 >,<仮引数 >,<仮引数 >,　…);

プロシージャに付帯することができる仮引数は reg のみです。
以下にプロシージャ宣言の例題を挙げます。

記述例 2-6　　プロシージャの宣言例

```
declare   test_proc {
          // 入出力構成要素
}
module    test_proc {
          reg       r1, r2, r3 ;
          proc_name proc_A() ;          // プロシージャproc_A を宣言
          proc_name proc_B(r1) ;        // プロシージャproc_B を仮引数r1 を付帯させて宣言
          proc_name proc_C(r2, r3) ;    // プロシージャproc_C を仮引数r1,r2 を付帯させて宣言
          // 動作記述
}
```

このように宣言することでプロシージャをモジュール内で使用することが可能となります。
プロシージャの動作記述は第 6 章を参照してください。

2.2.6 ステートの宣言

モジュールやプロシージャ、par や any でないブロックはステートを持つことができます。
ステート名の宣言により、ステートを持つことを宣言します。ここではステートマシンの各状態であるステート名の宣言を解説します。

ステート名の宣言はモジュールやブロックの構成要素宣言部分で以下のように行います。

state_name　　ステート名 ,<ステート名 >,<ステート名 >, … ;

モジュール内で宣言した各ステートに対応する動作は、そのモジュールの動作記述で定義します。
ブロック内で宣言したステートは他のブロックから呼び出すことはできません。

モジュールをリセットすると、モジュール/ブロックの全状態変数はステート名の宣言の先頭に記述されたステートに初期化されます。

記述例 2-7　　状態変数ステートの宣言例

```
declare    test_state {
           // 入出力構成要素
}
module     test_state {
           // 共通動作記述部分
           {
           // ステートの宣言 先頭に記述したステートから開始する。
           state_name state1, state2, state3 ; // ステートstate1,state2,state3 を宣言
           }
}
```

ステート名の宣言を行うことでステートマシンを使えるようになります。ここで宣言した各ステートの動作記述方法は第4．7章を参照してください。

2.2.7 メモリの宣言

NSL では多量の情報を整理して保存するためにメモリを使うことができます。NSL のメモリは非同期読み出し、同期書き込み SRAM をモデル化しています。メモリは内部構成要素宣言部分で宣言して使用します。
メモリに書き込みを行った次のクロックで当該アドレスに値が反映されます。
メモリの宣言時にワード数とビット幅を指定します。ビット幅は省略することもでき、その場合ビット幅は 1 になります。

mem　　メモリ名　[ワード数] <[ビット幅]> ;

宣言時にメモリの初期化を行うことも可能です。メモリ初期化の方法は以下の通りです。

mem　　メモリ名　[ワード数] <[ビット幅]> = {0 番地のデータ , 1 番地のデータ , … ,X 番地のデータ }

ワード数より初期化データの数が少ない場合、初期化データがないアドレスのメモリは 0 に初期化されます。
また、初期値のビット幅がメモリのビット幅より広い場合、メモリのビット幅の部分だけが初期値として転送されます。例えば、

mem　　memsample [5] [2] = { 4, 3, 2, 1, 0 } ;

とした場合、memsample のビット幅は 2bit になるので、memsample[5][2]の初期値は { 0, 3, 2, 1, 0 }になります。
メモリの宣言例を記述例 2-8 に挙げます。

記述例 2-8　　メモリの宣言例

```
declare    test_mem {
}
module     test_mem {
           mem      memory1[1024][32] ;                          // 初期化なしメモリ宣言の例
           mem      memory2[4][8] = { 8'hFF, 8'hAA, 8'h12, 8'h32 } ;   // 初期化ありメモリ宣言の例
           mem      memory3[256] ;                               // ビット幅省略メモリ宣言の例
}
```

記述例 2-8 のように記述すると、ワード数 1024、ビット幅 32 ビットの memory1(初期化なし)と、ワード数 4、ビット幅 8 ビットの memory2(0xff,0xaa,0x12,0x32 で初期化)、ワード数 256、ビット幅 1 ビットの memory3 を宣言します。
メモリを使用した動作記述は第 4. 9章を参照してください。

2.2.8 構造体の宣言

複数の信号をまとめて扱えるよう、構造体を使うことができます。

構造体は、まず定義を行います。定義はモジュールの外部(declare 内でも module 内でもない部分)に記述します。
この時点では信号の型を指定しません。また、struct 宣言の最後に ";"が必要なことに注意してください。

```
struct 構造体名 {
        構造体メンバ 1<[ビット幅]>  ;
        構造体メンバ 2<[ビット幅]>  ;
        構造体メンバ 3<[ビット幅]>  ;
        :
        構造体メンバ X<[ビット幅]>  ;
};
```

構造体のメンバのうち先に宣言された方が構造体の上位側に配置されます。
この後、module 内で構造体のインスタンスを宣言します。インスタンス宣言の時に信号の種類が reg なのか wire なのかを指定します。構造体のインスタンス宣言についての詳細は第 8 章で解説します。

以下の記述例 2-9 に構造体の宣言例を挙げます。

記述例 2-9　　構造体の宣言例

```
struct config_addr {                    // config_addr 構造体を宣言
        p_enable;
        p_reserve[7];
        p_bus[8];
        p_device[5];
        p_func[3];
        p_regaddr[6];
        p_zero[2];
};                                      // ; が必須
declare {
        input p[32];
}
module test_mem {
        config_addr reg caddr_1 ; // config_addr 構造体のインスタンスcaddr_1 を宣言
        caddr_1 := p;
}
```

3 動作の記述　基本

NSL 動作記述の基本を説明します。

3.1 値の転送

転送はある端子やレジスタ等から他の端子やレジスタ等に値を入力することを指します。
以下の表 3-1 が転送の種類です。

表 3-1 **転送の種類**

wire/output/inout 転送	**4 =**
reg/memory 転送	**5 :=**

wire、output、inout に転送する場合は ″=″を使用します。
転送先 = 転送元

また、reg と memory に値を転送する場合は ″:=″を使用します。
転送先 := 転送元

5.1 レジスタのインクリメント及びデクリメント

変数の値を 1 足してその変数に書き戻すことをインクリメント、変数の値を 1 引いてその変数に書き戻すことをデクリメントと言いますが、 NSL ではレジスタに対してインクリメント、デクリメントできる単位動作があります。
前置 (プリ)、後置 (ポスト)の両方に対応しています。
どの場合でも変化した数値がレジスタに反映するのは次のクロックであることに注意してください。

レジスタの値をインクリメントするとき ″++″を、
デクリメントする場合は ″—″を用いて以下のように表記します。

++レジスタ名　　(プリインクリメント)
—レジスタ名　　(プリデクリメント)
レジスタ名 ++　　(ポストインクリメント)
レジスタ名 —　　(ポストデクリメント)

また、式の右辺でもインクリメントとデクリメントを使うことが可能です。
この場合でもインクリメント、デクリメントした値が反映するのは次のクロックになります。

たとえば、
i=r++
という処理は、まず r の値が i に転送され、次のクロックで r に r+1 が転送されます。
また、
i=++r という処理は、まず r+1 が i に転送され、次のクロックで r に r+1 が転送されます。

ここまでに出てきた基本的な単位動作を使った例を記述例 3-1 に示します。

記述例 3-1　　基本的な単位動作の記述例

```
declare test_par {
        input in_a[4] ;
        output out_b[4] ;
        output out_c[4] ;
        output out_d[4] ;
        output out_e[4] ;
}
module test_par {
        wire wire_i[4] ;
        reg r1[4], r2[4] = 4'd0, r3[4] = 4'd0 ;

        // 共通動作記述開始
        r1 := in_a ;            // in_a をr1 に転送
        out_b = 4'b1010 ;       // out_b に10 を転送
        out_c = r1 ;            // out_c にr1 を転送
        wire_i = 4'b1111 ;      // wire_i に15 を転送
        out_d = wire_i ;        // out_d にwire_i を転送
        out_e = wire_i ;        // out_e にwire_i を転送
        r2++ ;                  // r2 をインクリメント
        r3-- ;                  // r3 をデクリメント
}
```

記述例 3-1 では、共通動作記述部分に単位動作が 8 つ記述してありますが、これらは全て同時に実行されます。(ただし、r2、r3 にインクリメントやデクリメントの結果が反映するのは次のクロックです)

5.2 基本的な演算記述例

次は演算についての例題を提示します。

記述例 3-2　　ビット演算の記述例

```
declare test_bit_exec {
        input inA[8] ;
        input inB[8] ;
}
module test_bit_exec {
        reg r1[8], r2[8], r3[8], r4[8] ;

        r1 := inA | inB ;          //inA とinB の論理和をr1 に転送
        r2 := inA & inB ;          //inA とinB の論理積をr2 に転送
        r3 := ~inA ;               //inA の論理否定をr3 に転送
        r4 := ~( inA | ~inB ) ; //inA と"inB の論理否定" の論理和を否定したものをr4 に転送
}
```

記述例 3-2 はビット演算の論理和と論理積、そして論理否定の例です。
ビット演算は、各ビット毎の論理演算結果が出力される演算子です。

例えば、4ビットの信号 A と B がそれぞれ 1010と 1001だった場合、A&B は 1000となり、A|B は 1011となります。
このようにビット演算では各ビットの桁ごとに 1ビット対 1ビットで演算が行われます。ビット演算は演算対象同士のビット幅が同じである必要があります。

記述例 3-2 の場合には、
r1 には inA と inB 各ビット毎の論理和が転送されます。
r2 には inA と inB 各ビット毎の論理積が転送されます。
r3 には inA の各ビットの論理否定が転送されます。

次に算術演算の例をに示します。

記述例 3-3　　算術演算の記述例

```
declare test_math {
        input   inA[16] ;
        input   inB[16] ;
}
module test_math {

        reg r1[16], r2[16], r3[32] ;
        r1 := inA + inB ;       //inA とinB の和をr1 に転送
        r2 := inA - inB ;       //inA とinB の差をr2 に転送
        r3 := inA * inB ;       //inA とinB の積をr3 に転送
}
```

記述例 3-3 は算術演算の ″足し算″、″引き算″、″掛け算 ″の例です。
r1 には inA と inB の和が転送され、r2 には inA と inB の差が転送され、r3 には inA と inB の積が転送されます。
足し算と引き算は演算対象同士のビット幅が同じでないといけません。
掛け算の場合は ″演算対象のビット幅の和 ″が演算結果のビット幅となるので、演算結果の出力先のビット幅はあらかじめ的確な幅を確保しておかないといけません。

次にシフト演算の例を示します。
シフト演算は対象となる信号線やレジスタを、左右に任意のビットだけ数シフトする演算です。

記述例 3-4　　シフト演算の記述例

```
declare test_shift {
        input       inA[16] ;
}
module test_shift {
        reg r1[16], r2[16], r3[16] ;
        r1 := inA>>5 ; //inA を右に5 ビットシフトしたものをr1 へ転送
        r2 := inA<<6 ; //inA を左に6 ビットシフトしたものをr2 へ転送
}
```

記述例 3-4 はシフト演算の ″右シフト″、″左シフト ″の例です。
シフト演算で左右にシフトしてもビット幅はシフト前と同じです。
右シフト時に右側にはみ出したビットは破棄され、空いた左側は ″0″の値で埋められます。
左シフト時も同様に左側にはみ出したビットは破棄され、空いた右側は ″0″の値で埋められます。

次に、ビット連結の例を提示します。ビット連結は別々の信号を連結することができる演算です。
以下、記述例 3-5 を示します。

記述例 3-5　　ビット連結の記述例

```
declare    test_sig {
           input      inA[4] ;
           input      inB[4] ;
}
module     test_sig {
           reg        r1[8] ;
           r1 := { inA, inB } ;      //inA とinB を連結した8bit 信号をr1 に転送
}
```

r1 には inA と inB を連結した 8 ビット信号を転送します。
ビット連結は例題のように 2 本の信号線だけでなく、複数信号を連結させることも可能です。
転送先の信号と連結後の信号のビット幅は同じでないといけません。

また、左辺でビット連結を行うことにより、複数の信号にまとめて転送することができます。
その場合、連結を表す {}の前に を記述します。以下の記述例 3-6 に例を示します。

記述例 3-6　　左辺でのビット連結

```
declare    test_sig {
           input      inA[16];
           output     outW[4] ;
           output     outX[4] ;
           output     outY[4] ;
           output     outZ[4] ;
}
module     test_sig {
           reg        r1[4] ;
           reg        r2[3] ;
           reg        r3[5] ;
           reg        r4[2] ;
           {outW,outX,outY,outZ } = inA ;                  // 16bit 幅のinA を4 ビット幅の
                                                           // outW,outX,outY,outZ に分けて転送
                                                           // outW = inA[15:12] ; outX = inA[11:8] ;
                                                           // outY = inA[7:4] ; outZ = inA[3:0] ; と等しい
           {r1,r2,r3,r4} := 14'b0101_010_11100_11;         // レジスタr1,r2,r3,r4 に14bit 幅の値を転送
}
```

次に、リダクション演算の記述例について解説します。
以下の記述例 3-7 に示します。

記述例 3-7　　リダクション演算の記述例

```
declare     test_red {
}
module      test_red {
            reg         r1, r2, r3 ;
            wire        w1[4], w2[4] ;
            w1 = 4'b1010 ;
            w2 = 4'b0000 ;
            r1 := &w1 ;             //r1 にw1 のリダクション演算AND の演算結果が転送
            r2 := |w2 ;             //r2 にw2 のリダクション演算OR の演算結果が転送
            r3 := ^w1 ;             //r3 にw1 のリダクション演算EX-OR の演算結果が転送
}
```

リダクション演算は束線信号のビット桁毎の論理演算を行う演算子です。
例えば、 1010 という 2 進数の数値のリダクション演算子 AND は
1＆0＆1＆0

となり、答えは 1 ビットの偽となります。

次に論理演算の記述例 3-8 を記述例に示します。

記述例 3-8　　論理演算の記述例

```
declare     test_logic {
            input       inA[4] ;
            input       inB[4] ;
}
module      test_logic {
            reg r1, r2, r3 ;
            r1 := !inA ;            // 論理否定後、inA に1 つでも1 があったら真。
                                    // それ以外は偽がr1 に転送。
            r2 := inA && inB ;      // inA とinB の論理積後、演算結果に1 つでも1 があったら真を
                                    //それ以外は偽をr2 に転送。
            r3 := inA || inB ;      // inA とinB の論理和後、演算結果に1 つでも1 があったら真を
                                    // それ以外は偽がr3 に転送。
}
```

論理演算は論理否定、論理積、論理和の 3 種類があります。
論理演算は演算結果に 1 つでも 1 が存在したら真、それ以外は偽が出力される演算子です。
つまり論理演算の演算結果は真か偽、1 ビットの 1 か 0 のどちらかになります。

次にリピート演算について解説します。
リピート演算を使うことにより、任意のビット列を任意の回数繰り返して別のビット列を生成することができます。
以下の記述例 3-9 にリピート演算の記述例を示します。

記述例 3-9　　リピート演算の記述例

```
declare     test_repeat {
            input a[8];
            output rgb[24];
}
module      test_repeat {
            reg r1[4];
            reg r2[8];
            rgb = 3{a};      // 8bit の入力信号a を3 回繰り返して24bit にしたビット列をrgb に出力
            r2 := 2{r1};     // 4bit のレジスタr1 を2 回繰り返して8bit にしたビット列をr2 に転送
}
```

リピート演算子のリピート回数には integer とビット幅を持たない整数が、リピートされるビット列には reg,wire,variable の各信号と、ビット幅を持つ整数が使用可能です。

次にビット切り出しの記述例を提示します。
NSL では信号に []を付けてビットを指定することにより、任意のビットを読み出すことが可能です。
記述例 3-10 を示します。

記述例 3-10　　ビット切り出しの記述例

3

```
declare     test_bit_div {
            input inA[8] ;
            input inB[8] ;
}
module      test_bit_div {
            reg r1[4], r2[8], r3[14] ;
            r1 := inA[3:0] ;                    //inA の0 ~ 3 桁目をr1 に転送
            r2 := { inA[0], inB[6:0] } ;        //inA の0 桁目とinB の0 ~ 6 桁目を連結してr2 に転送
            r3 := { inA[7], inB, inA[4:0] } ;   //inA の7 桁目とinB とinA の0~4桁目を結合してr3 に転送
}
```

r1 には、inA の 0～ 3 桁目を転送しています。
r2 には、inA の 0 桁目と inB の 0～ 6 桁目をビット連結したものを転送しています。
r3 には、inA の 7 桁目と inB と inA の 0～ 4 桁目をビット連結したものを転送しています。
このように、任意のビットを切り出して読み出すことが可能です。

読み出し時に任意のビットを切り出すことは可能ですが、任意のビットに対して書き込むことは許可されていません。
任意のビットに対して書き込みたい場合は、第 6 章で解説する一時端子 variable を使用してください。

また Verilog HDL や SystemC で、ビット幅の広い信号からビット幅の狭い信号へビット切り出しを使わずそのまま転送すると上位ビットが切り捨てられて転送されます。
NSL でも同様の記述方法が使えますが「 VHDL ではエラーになる」「可読性が下がる」などの問題があるため、ビット幅変換の時はビット切り出しを使うようにしてください。

さらに、ビット切り出し時に []内の桁指定を、[大きい値 :小さい値]の順ではなく
[小さい値 :大きい値]の順に書くことにより、ビットの並び順を反転させることができます。
[]内の桁数指定は即値のみ使用可能です。
例を記述例 3-11 に示します。

記述例 3-11　　ビットの並び順反転例

```
declare     bit_field_reverse {
            input       a[8];
            output      b[8],       c[8];
}
module      bit_field_reverse {
            b = a[0:7];             // 全ビットの並びを反転
            c = {a[4:7],a[0:3]};    // 4 ビットずつ並びを反転したものを結合
}

このNSL コードは、次のようなVerilog HDL コードに合成されます。
module      bit_field_reverse ( p_reset , m_clock , a , b , c );
            input p_reset, m_clock;
            input [7:0] a;
            output [7:0] b;
            output [7:0] c;
            assign b = {{{{{{{a[0],a[1]},a[2]},a[3]},a[4]},a[5]},a[6]},a[7]};
            assign c = {{{{a[4],a[5]},a[6]},a[7]},{{{a[0],a[1]},a[2]},a[3]}};
endmodule
```

次にビット幅指定の記述例として以下に記述例 3-12 を示します。

記述例 3-12　　ビット幅指定の記述例

```
declare     test_bit_width_assign {
            input       inA[8] ;
            input       inB[8] ;
}
module      test_bit_width_assign {
            reg r1[16] ;
            reg r2[4] ;
            r1 := 16'(inA) ;          // inA を16 ビットに拡張してr1 に転送
            r2 := 4'(inB) ;           // inB を4 ビットに縮小してr2 に転送
}
```

転送元より転送先のビット幅が大きい場合（ビット拡張)、上位側を 0 で埋めて目的のビット幅の信号にします。
転送元より転送先のビット幅が小さい場合（ビット縮小)、 0 ビット目から目的とするビット幅分を切り出して転送します。
拡張、縮小とも変更後のビット幅を指定することに注意してください。

例えば 8'(4'b1010)は 8'b00001010 に、4'(8'b10100101)は 4'b0101 になります。

次にビット幅拡張の記述例として以下に記述例 3-13 を示します。

記述例 3-13　　ビット幅拡張の記述例

```
declare     test_bit_ext {
            input       inA[8] ;
            input       inB[8] ;
}
module      test_bit_ext {
            reg         r1[16] ;
            reg         r2[16] ;
            r1 := 16#(inA) ;        // inA を16 ビットに符号付きビット幅拡張してr1 に転送
            r2 := 16'(inB) ;        // inB を16 ビットに符号なしビット幅拡張してr2 に転送
}
```

符号付きビット幅拡張は信号の先頭ビットを符号ビットとみなして、符号を維持したまま任意のビット幅に信号を拡張する演算子です。
信号の先頭ビットが 0 だった場合、0 で拡張されます。
信号の先頭ビットが 1 だった場合、1 で拡張されます。

例えば
4'b0101 という数値を 8 ビットに符号付きビット幅拡張すると 8'b00000101 となり、
4'b1010 という数値を 8 ビットに符号付きビット幅拡張すると 8'b11111010 となります。

符号なしビット幅拡張は前述のビット幅指定演算子と同じもので、信号の先頭ビットと関係なく先頭に 0 を追加して任意のビット幅に信号を拡張します。

4'b0101 という数値を 8 ビットに符号なしビット幅拡張すると 8'b00000101 に、
4'b1010 という数値を 8 ビットに符号なしビット幅拡張すると 8'b00001010 となります。

どちらの場合も、拡張後のビット幅を指定してビット幅拡張することに注意してください。

5.3 条件演算

条件演算は転送の右辺で使用し、転送する信号や値を場合分けする演算です。
この演算は演算子 if と else を用いて以下の様に記述します。

if （条件式 ）<信号や値 > else <信号や値 >

if 直後の（ ）内に記述した条件式が真である場合、（条件式)直後の信号や値を演算に使用します。条件式が偽である場合、else 直後の値を演算に使用します。条件演算を使う上で注意すべき点は、else が必須ということです。

条件演算の記述例を以下の記述例 3-14 に示します。

記述例 3-14　　条件演算の記述例

```
declare     test_right_if {
            input       a[3], b[3] ;
            input       trigger ;
            output f[3], g[3] ;
}
module      test_right_if  {
            //trigger が真ならa を転送, 偽ならb を転送
            f = if  (trigger)  a  else  b ;
            //trigger が真ならa+b を転送, 偽ならa+1 を転送
            g = a + if (trigger)  b  else  0b001 ;
}
```

6 動作の記述　ブロック

6.1 ブロック

動作記述は、NSL 内で単位動作の振る舞いを決定する領域のことを指します。
ここでは動作記述のうち NSL 記述の基本となるブロックについて解説していきます。
ブロックは開始点と終了点を定めて、ブロック領域内での単位動作のふるまいを変化させる構文です。
NSL ではこのブロックを用いてシステムを構成していきます。

以下の表 6-1、表 6-2 にブロックの種類を挙げます。

表 6-1　**NSL 動作記述のどの部分でも使えるブロック**

並列動作ブロック	{ }
alt ブロック	alt { }
any ブロック	any { }
if ブロック	if (式) { } else { }

表 6-2　**使える条件が制限されるブロック**

seq ブロック	ファンクションおよびプロシージャの動作記述内でのみ記述可能
while ブロック	seq ブロック内でのみ記述可能
for ブロック	seq ブロック内でのみ記述可能

6.2 並列動作ブロックの記述

並列動作ブロックは、ブロック内の単位動作を全て並列に動作させるブロックです。
ブロックの先頭に内部構成要素の宣言を記述することができます。ブロック内で宣言された要素は、宣言後なら他のブロックからも参照できます。ただしステートマシンの状態変数のみ他のブロックから参照することはできません。

並列動作ブロックの記述方法を以下に示します。

```
{
        < 内部構成要素宣言 >
        単位動作 1
        単位動作 2
        単位動作 3
        …
        単位動作 X
}
```

並列動作ブロックは、alt,any,if,seq の中など動作を 1 個のみしか記述できない場所に並列記述を書く際に使用します。

6.3 alt ブロック

alt ブロックは alternative ブロックの略で、条件に合った動作が起動するブロックです。

alt ブロックは条件分岐なので、演算子の関係演算を使用します。
関係演算は左辺と右辺の関係を表し、左辺と右辺の関係が真であったら条件が成立します。左辺と右辺の関係が偽であったら条件は不成立となります。
条件式の記述方法は以下の通りです。

左辺式　関係演算子 右辺式

alt, any, if ブロックはこの関係演算を用いて条件式の判断を行います。
alt ブロックの動作には優先順位が存在します。条件に合った動作が複数ある場合でも、起動するのは記述順で一番上にある動作のみです。
また、全ての条件に合わない場合の動作記述として"else"を記述できます。"else"は省略可能です。
alt ブロックの記述方法は以下のようになっています。

```
alt {
        条件 1 : 単位動作   // 優先順位 高
        条件 2 : 単位動作 2
        条件 3 : 単位動作 3
        …
        条件 N : 単位動作 N          // 優先順位 低
        else : 単位動作 X
}
```

alt ブロック記述例を記述例 6-1 に示します。

記述例 6-1　　**alt ブロック記述例**

```
declare test_alt {
input in_a[4] ;
output out_b[4] ;
}
module test_alt {
   reg reg_c[4] ;
   // 共通動作記述開始
   alt{
      in_a[3] == 1'b1 : reg_c := 4'b1111 ; // 条件式が真ならばreg_c に1111 を転送
      in_a[2] == 1'b1 : reg_c := 4'b1010 ; // 条件式が真ならばreg_c に1010 を転送
      in_a[1] == 1'b1 : reg_c := 4'b0101 ; // 条件式が真ならばreg_c に0101 を転送
      // 条件分岐先での並列アクションブロック使用例
      in_a[0] == 1'b1 : {
         reg_c := 4'b0001 ;
         out_b = 4'b1111 ;
      }
   }
}
```

また alt ブロック内に並列動作ブロックを記述することで、条件の遷移後に複数の単位動作を記述することが可能です。これは他の条件文にも適用できます。

6.4 any ブロック

any ブロックは条件付きブロックで、条件に合った動作が起動するブロックです。
alt ブロックと違い、any ブロックでは条件付き動作に優先順位がなく条件に合った動作が全て起動します。
また、全ての条件に合わない場合の動作記述として"else"を記述することができます。"else"は省略可能です。
any ブロックの記述方法は以下のようになっています。

any {
条件 1 : 単位動作 1
条件 2 : 単位動作 2
条件 3 : 単位動作 3
…
条件 N : 単位動作 N
else : 単位動作 X
}

記述例 6-2 に any ブロック記述例を示します。

記述例 6-2 any ブロック記述例

```
declare test_any {
    input in_a[4] ;
}
module test_any {
    reg r1[4], r2[4], r3[4], r4[4], r5[4] ;
    // 共通動作記述開始
    any{
        in_a[3] == 1'b1 : r1 := 4'b1111 ; // 条件が真ならr1 に1111 を転送
        in_a[2] == 1'b1 : r2 := 4'b1010 ; // 条件が真ならr1 に1010 を転送
        in_a[1] == 1'b1 : r3 := 4'b0101 ; // 条件が真ならr1 に0101 を転送
        in_a[0] == 1'b1 : r4 := 4'b0001 ; // 条件が真ならr1 に0001 を転送
        else            : r5 := 4'b0000 ; // すべてが不成立ならr5 に0000 を転送
    }
}
```

このように記述することで、any ブロックを用いた条件判断回路が実現できます。

6.5 if ブロック

any ブロックの特別な型式に if 構文があります。
if 構文は条件が 1 個だけの any 構文と同一の動作、つまり条件が真の時に動作記述で示される動作が起動されます。
また、条件に合わない場合の動作記述として ″else″を記述します。″else″は省略可能です。
if ブロックの記述方法は以下のようになっています。

if （ 条件 ）　　単位動作 1
else　　単位動作 2

記述例 6-3　　if ブロック記述例

```
declare   test_if {
   input   in_a[4] ;
}
module   test_if {
   reg   r1[4],   r2[4],   r3[4],   r4[4],   r5[4]   ;
   if   (in_a[3] == 1'b1)   r1 := 4'b1111 ;
   if   (in_a[2] == 1'b1)   r2 := 4'b1010 ;
   if   (in_a[1] == 1'b1)   r3 := 4'b0101 ;
   if   (in_a[0] == 1'b1)   r4 := 4'b0001 ;
   else                     r5 := 4'b0000 ;
}
```

このように記述することで if ブロックを使用することが可能です。
また、記述例 6-2 と記述例 6-3 は等価です。

6.6 seq ブロック

seqブロック内では記述した上から順に 1 動作ずつ、1 クロックごとに動作記述が起動します。また第一動作は起動した時と同一クロックで実行されます。

seq ブロック中からプロシージャ(proc)を呼び出した場合、呼び出したプロシージャが終了するまで、seq ブロックの実行は停止します。そして、呼び出したプロシージャが終了(finish もしくは他のプロシージャを起動)するのと同じクロックで seq ブロックの実行が再開します。

seq ブロックの記述例を以下に示します。

```
seq {
        動作 1
        動作 2
        動作 3
        …
        動作 X
}
```

seq ブロック内でのみ使える構文を表 6-3 に挙げます。

表 6-3 **seq ブロック内でのみ使える構文**

label_name seq	ブロック内でラベルを定義します
goto	ラベルまで移動します
while	条件付きループブロック
for	ループ変数を使う条件付きループブロック

seqブロックの記述例は、5 章で説明するファンクションを使用しています。合わせて参照してください。

記述例 6-4　　**seq ブロックの記述例**

```
declare test_seq {
    input a[4], b[4] ;
    output f[4] ;
    func_in exec_add ;
}
module test_seq {
    reg opr1[4], opr2[4], result[4] ;
    func exec_add seq {
        { // 並列動作ブロック ※ 1 クロック目に起動
            opr1 := a ;
            opr2 := b ;
        }
        result := opr1 + opr2 ; // ※ 2 クロック目に起動
        f = result ; // ※ 3 クロック目に起動
    }
}
```

このように記述することで、seq ブロックを利用した順次実行回路を実現できます。
exec_add を呼び出すと、まず 1 クロック目に seq ブロック内先頭の並列動作ブロックが実行されます。そして 2 クロック目に result への転送、3 クロック目に f への転送が実行されます。
以上のに順番に実行されます。
seq ブロックの最終行を実行が開始されると、順次実行が終了します。
またモジュール test_seq の場合、exec_add を 1 度呼び出して seq ブロックが順次実行している最中に、再び exec_add を呼び出すとパイプライン処理になります。

6.6.1 ラベルと goto

seq ブロック内でラベルを定義して、ラベル位置まで goto で移動することが可能です。
ラベルは使用する seq ブロック内で宣言を行うことが必須です。ラベルの宣言方法は以下のとおりです。

label_name ラベル名

複数のラベル名を宣言する場合、ラベル名をカンマ","で区切って記述します。

label_name ラベル名 ,ラベル名 ,ラベル名

seq ブロック中でのラベルの定義は以下のように記述します。

ラベル名 :

そして、ラベルの位置に移動する場合は同一の seq ブロックの中で、以下のように記述します。

goto ラベル名

goto 文を用いた処理の遷移に 1 クロック使用します。ラベルの記述例は以下の通りです。

```
seq {
  label_name ラベル名 1, ラベル名 2

  動作 1
  goto ラベル名 2
ラベル名 1 :
  動作 2
  動作 3
ラベル名 2 :
  動作 4
  goto ラベル名 1
}
```

として記述例 6-5 を示します。

記述例 6-5　　seq ブロック :ラベルの例

```
declare test_label {
    output f[4] ;
    func_in exec_label ;
}
module test_label {
    reg r1[4]=0 ;
    func exec_label seq {
        label_name label1, label2 ; // ラベルlabel1,label2 を宣言
            goto label2 ; // 次クロックlabel2 から実行
        label1 : //label1 を提示
            r1++ ;
        label2 : //label2 を提示
            if(r1 == 10) f = r1 ;
            goto label1 ; // 次クロックにlabel1 から実行
    }
}
```

6.6.2 while ブロック

seq ブロック内限定の構文として while ブロックがあります。while ブロックは条件付きループ文として使用します。
動作開始前に条件が偽であれば、while ブロックは一度も起動せずに終了します。
動作開始前に条件が真であれば、while ブロックは上から記述した順番に起動されます。
while ブロック内の全ての動作が終了すると、もういちど条件が真であるかどうかを確認し、真ならば再び while ブロックの始めから動作が開始し、偽であれば while ブロックの動作が終了します。
1 動作のみでもブロックの括弧{}が必須です。
while ブロックの動作記述例は以下の通りです。

```
while( 条件 ){
        動作 1
        動作 2
        …
        動作 X
}
```

また、while ブロックは条件判定と動作遷移にそれぞれ 1 クロック使用します。
記述例 6-6 に while ブロック記述例を、

記述例 6-7 に seq で記述した等価回路を示します。

記述例 6-6　　while ブロックの記述例

```
declare test_while {
    input count_end_sig ;
    func_in exec_count ;
    func_out count_end_call ;
}
module test_while {
    reg cnt[8] = 0 ;
    func exec_count seq {
        //while 文のループ開始
        while (~count_end_sig) {
            cnt := cnt + 0x01 ;
        }
    count_end_call() ;
    }
}
```

記述例 6-7　　seq ブロックを使った等価回路

```
declare test_while {
    input count_end_sig ;
    func_in exec_count ;
    func_out count_end_call ;
}
module test_while {
    reg cnt[8] = 0 ;
    func exec_count seq {
        label_name label1, label2 ;
        label1 :
            if(count_end_sig) goto label2 ;
            {
                cnt := cnt + 0x01 ;
                goto label1 ;
            }
        label2 :
            count_end_call() ;
    }
}
```

記述例 6-6 では while ブロックの条件を満たす限り何度でもファンクション″exec″が呼び出されます。
seq ブロックでクロックの見えやすい形に書き換えたものが

記述例 6-7 です。

6.6.3 for ブロック

seq ブロック内限定の構文として for ブロックがあります。
for ブロックは while ブロックと同じく、条件付きループ文として使用します。C 言語の for 文のようにループ変数を変化させながらブロック内の動作を 1 動作 1 クロックで順次実行していきます。ループ変数にはレジスタ(reg)を使用します。他の型の信号をループ変数に使うと順次実行にならないので注意してください。
for ブロックの動作記述例は以下の通りです。

```
for(ループ変数初期値 ; ループ条件式 ; ループ変数変化値 ){
        動作 1
        動作 2
        …
        動作 X
}
```

for ブロックは、まずループ変数に初期値を設定します。そして条件式が真である場合に
for ブロック内の動作が起動します。また条件式が偽である場合は for ブロック内の動作は一度も起動せずに終了します。
for ブロックが起動すると、ブロック内を上から順に 1 クロック1つずつ単位動作を起動していきます。for ブロック内の単位動作が全て終了すると、ループ変数を更新してから条件式の比較を行います。そして条件式が真である場合は再び for ブロック内の動作が起動し、偽である場合はその場で
for ブロックは終了します。
for ブロックは条件判定と動作遷移にそれぞれ 1 クロック使用します。
また、while と同じく順次動作であることをはっきりさせるため、1 動作のみでもブロックの括弧 {}が必須です。

以下に for ブロック記述例と seq で記述した等価回路を示します。

記述例 6-8　　for ブロックの記述例

```
declare test_for {
    func_in exec_sum ;
    func_out exec_end_call ;
}
module test_for {
    reg i[4] = 0 ;
    reg sum[8] = 0 ;
    func exec_sum seq {
        //for 文 i をループ変数にしてループ開始。
        for(i:=0 ; i<10 ; i++) {
            sum := sum + { 0x0, i} ;
        }
        //exec の終わりをfunc_out でコールする。
        exec_end_call() ;
    }
}
```

記述例 6-9　　for ブロックの動作詳細

```
declare test_for {
    func_in exec_sum ;
    func_out exec_end_call ;
}
module test_for {
    reg i[4] = 0 ;
    reg sum[8] = 0 ;
    func exec_sum seq {
        label_name label1, label2 ;
            i := 0x0 ;
        label1 :
            if(i<10) goto label2 ;
            {
                sum := sum + { 0x0, i } ;
                goto label1 ;
            }
        label2 :
            exec_end_call() ;
    }
}
```

記述例 6-8 の for ブロックを seq ブロックを利用してクロックが見えやすいように書き換えたものが

記述例 6-9 です。記述例 6–8 と

記述例 6-9 は等価回路です。

6.6.4 カウント型 for ブロック

ループ変数の変化量が+1 または−1 でよいとき、カウント型 for が使えます。
ループ変数の変化量が+1 または−1 に固定されている以外は前述の for ブロックと変わりません。
カウント型 for ブロックは以下のように記述します。

```
for(ループ変数 := 初期値, 終値 ){
  動作 1
  動作 2
  …
  動作 X
}
```

初期値と終値の関係は以下の通りです。

・初期値<終値の時、ループ変数はアップカウントしながらブロック内動作を実行
・初期値>終値の時、ループ変数はダウンカウントしながらブロック内動作を実行
・初期値=終値の時は一回だけブロック内動作を実行

for(i:=0,5)
と記述した場合、i は 0,1,2,3,4,5 の値を取り、ブロック内の動作を 6 回実行することに注意してください。

記述例 6-8 と同じ回路をカウント型 for ブロックで記述した例を記述例 6-10 に示します。

記述例 6-10　　カウント型 for ブロック

```
declare test_for {
    func_in exec_sum ;
    func_out exec_end_call ;
}
module test_for {
    reg i[4] = 0 ;
    reg sum[8] = 0 ;
    func exec_sum seq {
        //for 文 i をループ変数にして0 から9 までループする。
        for(i:=0 ,9 ) {
            sum := sum + { 0x0, i} ;
        }
        //exec の終わりをfunc_out でコールする。
        exec_end_call() ;
    }
}
```

6.7　ステートの動作記述

第 4 章でステートの宣言方法について解説しました。本章では、ステートの動作記述について解説します。

ステートの内部動作を記述する場合は以下の方法を用います。

state ステート名 動作

まず、モジュール起動直後に最初に宣言されたステートが起動します。
ステートは一度起動すると別のステートに移動するまで動作を続けます。
ステートを複数宣言した場合、起動中のステートから別のステートを起動させる（遷移する）時には、goto 文を使用します。goto 文の使用方法を以下に示します。

goto ステート名

この"goto"により別のステートを起動することが可能となります。別のステートに移動すると、元のステートは停止します。
また起動中のステートは保持されるので、ステートの宣言先がプロシージャだった場合、起動しているプロシージャが遷移して、再び起動した場合は、最初に宣言したステートではなく、直前まで起動していたステートから開始します。

この"goto"により別のステートを起動できます。別のステートに移動すると元のステートは停止します。
起動中のステートは保持されます。プロシージャ内で宣言されたステートが起動中にそのプロシージャが遷移し、再度プロシージャが起動した時には起動していたステートから開始されます。

また、ステートは宣言を行った場所でのみ動作記述が可能です。ステートの宣言が可能な場所は

・並列動作ブロック内
・プロシージャの中

です。
以下の記述例 6-11 にステートの記述例を示します。

記述例 6-11　　ステート記述例

```
declare state_test {
    input a[4] ;
    input b[4] ;
    output f[4] ;
    func_in start() ;
}
module state_test {
    reg cnt_val [4] = 4'b0000 ;

    state_name idle, count, calc ; // ステートの宣言

    // ステート"idle" の動作
    state idle {
        if(start) goto count ;
    }

    // ステート"count" の動作
    state count {
    any{
        cnt_val == 4'b1111 : {
            cnt_val := 4'b0000 ;
            goto calc ;
        }
        else : {
            cnt_val := cnt_val + 4'b0001 ;
        }
        }
    }

    // ステート"calc" の動作
    state calc {
        f = a + b ;
        goto idle ;
    }
}
```

このように記述することで、ステートを実現できます。
共通動作部分やプロシージャ内でも使用できるのがプロシージャとの違いです。

また、プロシージャ内で使用した場合は、プロシージャの終了時のステートを記憶しているという部分で異なります。
プロシージャ内でステートを使用した場合、別のプロシージャに遷移して元のプロシージャに戻ったときもステートは元のプロシージャが遷移した時のステートの状態を記憶しているので、途中の状態から動作を開始することが可能です。

6.8　サブモジュールの動作記述

前述のとおり、サブモジュールは階層構造を実現する構文です。 NSL では上位モジュールからサブモジュールの各端子を操作したり、サブモジュールから上位モジュールにデータを渡したりすることが可能です。

サブモジュール宣言はサブモジュールのテンプレートを指定して、そのテンプレートの上位モジュール内で実体化する名前を記述します。テンプレートを上位モジュール内で実体化したものを″インスタンス″と呼びます。
サブモジュールのデータや制御端子など各端子を指定する場合は以下のように記述します。

インスタンス名 端子名

この記述を用いることでサブモジュールの値読み出しや転送を行うことが可能です。
インスタンス数付きでサブモジュール宣言した場合は、

インスタンス名 [インスタンス番号]端子名

と記述します。この際、インスタンス番号として使えるのは整数 (1,2,3,)と整数変数 (integer)のみです。

また、サブモジュールの制御端子を呼び出すことも可能です。サブモジュールの制御入力端子を呼び出す場合は以下のように記述します。

インスタンス名　　制御入力端子名 ()

また、サブモジュールの制御入力端子に実引数を持たせる場合は以下のように記述します。

インスタンス名　　制御入力端子名 (<実引数 >, <実引数 >, <実引数 >, …)

実引数を複数持たせる場合は、カンマ″,″で区切ります。

そして、制御入力端子を呼び出すと同時に出力信号を受け取る場合の記述方法は以下のように記述します。

インスタンス名　　制御入力端子名 (<実引数 >)出力端子名

サブモジュールに戻り値端子が設定されていて return で値を戻している場合は、出力端子名を省略することができます。

インスタンス名　　制御入力端子名 (<実引数 >)

以下にサブモジュール構文の記述例を示します。

記述例 6-12　　サブモジュール構文の記述例

```
declare sub_test {
    input inA[16] ;
    input inB[16] ;
    input inC[16] ;
    input inD[16] ;
    output outE[16] ;
    func_in calc1(inA, inB) ;
    func_in calc2(inC, inD) ;
}
module sub_test {
    reg reg1[16] ;
    func calc1 reg1 := inA & inB ;
    func calc2 outE = inC + inD ;
}
declare main_test {
    input in_val1[16] ;
    input in_val2[16] ;
}
module main_test {
    reg result[16] ;
    sub_test SUB ; // サブモジュールのテンプレート"sub_test" をSUB という名前で実体化

    // 共通動作記述
    SUBcalc1(in_val1, in_val2) ;      //sub_test のcalc1 をin_val1,in_val2 の引数を渡して呼び出し。
    result := SUBcalc2(in_val1, in_val2)outE ; //sub_test のcalc2 をin_val1,2 の引数を渡して、
                                        //outE から出力した値をresult に書込み。
}
```

記述例 6-12 のように記述することで、サブモジュール構文を実現することが可能です。
ここでは sub_test モジュールをテンプレートにして、main_test モジュールで SUB として宣言し、インスタンスとしています。
また、main_test 中の動作記述ではインスタンスである SUB の中のファンクションを呼び出して、SUB に仕事をさせて、返ってきた値を result というレジスタに格納しています。

6.9 メモリに対する動作記述

第2. 2. 7章でメモリの宣言方法に関して解説しました。本章ではメモリに対する動作記述について解説します。

メモリに対しての転送はレジスタと同じく、データを書き込む場合は″:=″の転送を使用し、メモリから読み出す場合は″=″の転送を使用します。これにより、クロックに同期した書き込みと非同期の読み出しが行われます。

例として read/write 可能な幅 4 ビット/256 ワードのメモリを示します。

記述例 6-13　　メモリ記述例

```
declare mem_test {
    input in_data[4] ;
    input in_addr[8] ;
    output out_data[4] ;
    func_in write() ;
    func_in read() ;
}
module mem_test {
    mem memory[256][4] = { 4'b1010, 4'b0101, 4'b0000, 4'b1100};

    func write memory[in_addr] := in_data ;
    func read out_data = memory[in_addr] ;
}
```

メモリ記述時は、ワード数がビット幅とは関係ないことに注意してください。ビット幅が 1bit であろうと 32bit であろうと、16 ワードのメモリのワード数は 16 です。

7 動作の記述 ファンクション

第 2 章で制御入力端子、制御出力端子制御内部端子の宣言方法を解説しました。
本章では制御入力端子、制御出力端子、制御内部端子の動作記述、およびファンクション動作内でのみ使うことができるブロックについて解説します。

NSL の言語仕様では、制御の流れ(path)とデータの流れを区別して扱います。つまり、input,output,inout などのデータの流れとは別に制御の流れを記述します。制御端子は制御の流れを示します。

制御端子の種類は 3 種類あります。
すなわち NSL モジュールに入ってくる制御信号である制御入力端子、NSL モジュールから外部へ出る制御信号である制御出力端子、NSL 内部の制御を記述する信号である制御内部端子です。

7.1 制御内部端子

制御内部端子はモジュール内部の制御を記述する制御端子なので、宣言されたモジュール内でしかファンクションを呼び出せません。動作記述中で制御内部端子を呼び出すときは以下のように記述します。

制御内部端子名 ()

ファンクションは、呼び出した時と同一クロックで起動します。
また、仮引数を持たせた制御内部端子には、モジュール内で呼び出す際に実引数を持たせることが可能です。実引数を持たせてファンクションを呼び出す場合は、制御内部端子名の後ろの()内に実引数を列挙します。

制御内部端子名 (実引数, 実引数, 実引数 , …)

制御内部端子のファンクションの記述方法は、以下のように記述します。

func 制御内部端子名 動作記述

ファンクションの動作記述は省略することも可能です。
また、宣言した制御内部端子はファンクションなしで呼び出すことも可能です。この時呼び出した制御内部端子は呼び出しと同一クロックで 1 になり、次クロックで 0 になります。また、呼び出さない時は常に 0 のままです。
以下の記述例 7-1 が制御内部端子の記述例です。

記述例 7-1　　制御内部端子の記述例

```
declare func_test{
    input a[4] ;
    input b[4] ;
    output f[4] ;
}
module func_test{
    func_self func_do ; //  制御内部端子の宣言

    //  共通動作記述
    func_do() ; //  制御内部端子の呼び出し
    //  制御内部端子の動作記述
    func func_do {
        f = a | b ;
    }
}
```

このように制御内部端子の宣言、ファンクション呼び出し、ファンクションの記述が揃って初めてファンクションが起動します。

7.2　制御入力端子

制御入力端子はモジュール外部から入ってくる制御端子の信号です。
モジュール外部から入ってくる制御端子を制御入力信号と呼び、モジュールの外からモジュール内部のファンクションを起動できます。ファンクションを作成する場合は宣言した制御入力端子名と同じ名前にする必要があります。
ファンクションの記述方法は以下のとおりです。

func　　制御入力端子名 動作記述

ファンクションは省略することも可能です。
また制御内部端子の時とは異なり、制御入力端子はモジュール外部からの制御端子の入力を待つため、制御入力端子を宣言した同一モジュールからは呼び出せません。

外部から呼び出された制御入力端子は呼出を受けたクロックで 1 になり、次クロックで 0 になります。また呼び出しを受けない時は常に 0 です。
この記述方法と第 2 章の宣言方法を踏まえて制御入力端子の記述例を挙げます。

記述例 7-2　　制御入力端子の記述例

```
declare func_in_test{
    input a ;
    input b ;
    output f ;
    func_in func_do ;
}
module func_in_test{
    func func_do f = a | b ;
}
```

この記述例 7-2 のように記述することで、制御入力端子 func_do が呼び出された時にファンクション func_do が起動します。

7.3　制御出力端子

制御出力端子は、モジュールの外へ出す制御端子の信号です。
これは外部のモジュールを制御するための制御端子で、他の制御端子と同ように仮引数を持もたせることが可能です。

制御出力端子をモジュールで呼び出す場合は以下のように記述します。

制御出力端子名 ()

制御端子は、呼び出した時と同一クロックで起動します。
また、仮引数を持たせた制御出力端子をモジュール内で呼び出す場合、実引数を持たせることが可能です。実引数を持たせて呼び出す場合は、

制御出力端子名 (実引数, 実引数, 実引数 ,)

となります。制御出力端子の動作は上位モジュールに記述します。
記述例 7-3 に制御出力端子の記述例を挙げます。

記述例 7-3　　制御出力端子の記述例

```
declare memory {
}
declare return_fo {
        input i_rddata[8] ;
        output o_rdadrs[4] ;
        func_out memory_read(o_rdadrs) ;
                        // "o_rdadrs"を仮引数に設定して宣言
}
module memory {

        mem rom[16][8]  ;

        return_fo i_return_fo ;

        function i_return_fo.memory_read {
                i_return_fo.i_rddata = rom[i_return_fo.o_rdadrs] ;
        }
}

module return_fo {
        reg trigger[5] = 0 ;
        reg rddata[8] ;

        trigger := { trigger[3:0], 0b1 } ;
        if  ( trigger == 5'b00011 )  rddata:=memory_read(4'b0101) ;
}
```

7.4 戻り値

ファンクションから結果を返すデータ端子は自由に設定できるようになっていますが、返す結果が一つしかない場合には記述の負担を減らすため、C 言語の関数のような書式で戻り値を設定できるようにしてあります。ファンクション内で戻り値を返すためには、以下のように記述します。

return 値

戻り値を使うためには、制御端子の宣言時に戻り値を使うよう宣言しておく必要があります。この記述は制御内部端子、制御入力端子、制御出力端子のどれでも同じように使うことができます。

ただし、戻り値の記述は、seq ブロックを使用して記述する制御端子の動作記述には使用できません。

記述例 7-4 制御出力端子の戻り値記述例

```
declare memory {
}
declare return_fo {
        input i_rddata[8] ;
        output o_rdadrs[4] ;
        func_out memory_read(o_rdadrs) : i_rddata ;
                        // "o_rdadrs"を仮引数に、
                        // "i_rddata"を返り値に設定して宣言
}
module memory {

        mem rom[16][8]  ;
        return_fo i_return_fo ;

        function i_return_fo.memory_read {
                return   rom[i_return_fo.o_rdadrs] ;
        }
}

module return_fo {
        reg trigger[5] = 0 ;
        reg rddata[8] ;

        trigger := { trigger[3:0], 0b1 } ;
        if  ( trigger == 5'b00011 )  rddata:=memory_read(4'b0101) ;
}
```

8 動作の記述　プロシージャ

プロシージャは前述したとおり、状態遷移やパイプライン、順序回路を用いた制御を提供する構文で、一度起動すると他のプロシージャに遷移するか終了を宣言するまで動作をおこない続ける構文です。

8.1 プロシージャの起動

プロシージャを起動、またはプロシージャ内で別のプロシージャに遷移する場合は以下のように記述します。
プロシージャは、起動を提示した次クロックから実際に起動を始めます。

プロシージャ名（ 引数 ）

引数つきでプロシージャの宣言を行っていれば、引数が有効です。

8.2 プロシージャの動作記述

プロシージャ内部の動作は以下のように記述します。

```
proc プロシージャ名 {
   動作 1
   動作 2
   動作 3
   …
   動作 X
}
```

以下の記述例 8-1 にプロシージャの例を示します。

記述例 8-1　　プロシージャの記述例

```
declare proc_test{
    input a[4] ;
    input b[4] ;
    output f[4] ;
    func_in start ;
}
module proc_test{
    reg r1 = 1'b0, r2 = 1'b0, r3 = 1'b0 ;
    reg result[4] = 4'b0000 ;
    proc_name idle() ; // プロシージャ"idle" の宣言
    proc_name calc() ; // プロシージャ"calc" の宣言
    proc_name out_data() ; // プロシージャ"out_data" の宣言
    // 共通動作記述
    r1 := r2 ;
    r2 := r3 ;
    r3 := 1'b1 ;
    if(^r1 & r2 & r3) idle() ;
    // プロシージャ"idle" の動作記述
    proc idle {
        if(start) calc() ;
    }
    // プロシージャ"calc" の動作記述
    proc calc {
        result := a + b ;
        out_data() ;
    }
    // プロシージャ"out_data" の動作記述
    proc out_data {
        f = result ;
        idle() ;
    }
}
```

記述例 8-1 のように記述することでプロシージャを実現することが可能です。この例ではプロシージャを状態変数として使用しています。
このプロシージャを続けて呼ぶことによりパイプラインが実現できます。

8.3 プロシージャの終了

プロシージャを終了する場合は
finish()を使います。finish()の使い方は以下の通りです。

プロシージャ名.finish()

また、プロシージャ中で、そのプロシージャ自身を終了させる場合に限り、プロシージャ名を省略することができます。
その場合、以下のように記述します。

finish()

プロシージャは"finish"を宣言した次クロックで、動作を停止します。

8.4 invoke

invoke はプロシージャを終了せずに別のプロシージャを起動する記述方法です。invoke はプロシージャ動作内でのみ記述できます。

invoke は以下のように記述します。

プロシージャ名.invoke (引数)

引数つきでプロシージャの宣言を行っていれば、引数が有効です。

以下の記述例 8-2 に finish と invoke の記述例を示します。

記述例 8-2　　**invoke・遠隔 finish 記述例**

```
declare proc_action_test {
    output f ;
}
module proc_action_test {
    reg r1[8] = 0 ;
    reg trigger[8]=0 ;
    reg cnt[3] = 3'b000 ;

    proc_name proc1(), proc2(), proc3() ;

    r1 := { r1[6:0], 0b1 } ;

    if(r1 == 0b00000011) {
        proc1() ;
        cnt := 0 ;
    }

    if(r1 == 0b01111111) proc2.finish() ; // 遠隔finish でproc2 を終了

    proc proc1 {
        trigger := { trigger[6:0], 0b1 } ;
        if(trigger == 0b00000011) proc2.invoke() ; //invoke でproc2 を起動
        if(trigger == 0b00111111) proc3.invoke() ; //invoke でproc3 を起動
    }
    proc proc2 {
        f = 0b1 ;
        cnt := cnt + 1 ;
    }
    proc proc3 {
        if ( cnt == 3'b110 ) {
            proc1.finish() ; // 遠隔finish でproc1 を終了
            finish() ;
        }
    }
}
```

8.5 プロシージャでの seq ブロック使用

プロシージャの動作の記述には seq ブロックを使用することができます。

```
proc プロシージャ名 seq {
        動作 1
        動作 2
        動作 3
        …
        動作 X
}
```

seq ブロックを使用するプロシージャは、起動されると seq ブロック内の動作を順番に一度だけ実行します。順序実行を最後まで行った後も、他のプロシージャを起動したり、finish や遠隔 finish をしたりするまで、プロシージャ自体は何の動作も行わないものの起動し続けたままです。
プロシージャを終了すると、再び同一のプロシージャを起動することができます。

記述例 8-3　　プロシージャでの seq ブロック記述例

```
declare proc_seq2 {
    func_in p1_finish() ;
    func_in p1_invoke() ;
}
module proc_seq2 {
    reg trigger[3]=0 ;
    reg cnt[5]=0;
    proc_name p1();
    func_self f_test1, f_test2, f_test3, end_call();

    trigger := {trigger[1:0], 0b1} ;
    if(trigger==3'b011) p1() ;

    proc p1 seq {
        {cnt++;f_test1() ;}
        {cnt++;f_test2() ;}
        {cnt++;f_test3() ;}
        end_call() ;
    }

    func p1_finish {
        p1.finish() ;
    }

    func p1_invoke {
        p1.invoke() ;
    }
}
```

記述例 8-3 において、プロシージャ p1 内のカウンタ cnt は、遠隔 finish でプロシージャ p1 を終了したのち invoke で再起動したさいに、終了前の値を保持しています。

8.6　プロシージャの引数

seq ブロック使用の有無にかかわらず、プロシージャに引数をつけて起動することにより、プロシージャ内部のレジスタに対して、プロシージャ起動ごとに任意の値を渡すことができます。

記述例 8-4　　プロシージャへの引数

```
declare proc_seq2 {
    func_in p1_finish() ;
    func_in p1_invoke() ;
}
module proc_seq2 {
    reg trigger[3]=0 ;
    reg cnt[5]=0;
    proc_name p1(cnt);    // 引数にレジスタ cnt を指定
    func_self f_test1, f_test2, f_test3, end_call();

    trigger := {trigger[1:0], 0b1} ;
    if(trigger==3'b011) p1(5'b0) ;  // p1 起動時に cnt をリセット

    proc p1 seq {
        {cnt++;f_test1() ;}
        {cnt++;f_test2() ;}
        {cnt++;f_test3() ;}
        end_call() ;
    }

    func p1_finish {
        p1.finish() ;
    }

    func p1_invoke {
        p1.invoke(5'b0) ;          // p1 起動時に cnt をリセット
    }
}
```

9 制御構文

NSL からの Verilog HDL や VHDL の合成では、プリプロセッサによるディレクティブの展開(付録 1 参照)と構成要素からの回路記述の合成の間に構造展開という処理が入ります。構造展開で展開される構文が構造構文です。構造展開は回路記述の要素と関係なく、記述された順番に展開されます。
構造展開を利用することにより、同じような回路を複数生成する際の記述量を大幅に減らすことができます。

9.1 構造構文 generate

構造構文 generate は seq ブロック内の for と異なり、下位言語へのコンパイル時に generate 文内を構造展開して同一クロックでの動作になる構文です。構造構文 generate は以下のように記述します。

```
generate (ループ変数初期値 ;ループ条件式 ;ループ変数変化値 ) {
        動作 1
        動作 2
        …
        動作 X
}
```

ループ変数には後述する整数変数 integer のみ使用可能です。

構造構文 generate の展開は以下の手順で行います。

1. ループ変数に初期値を設定する
2. 条件式を判定し真の場合3へ、偽の場合は終了
3. 動作記述を構造展開
4. 変化値を更新して2へ

以下の記述例 9-1 に構造構文 generate の例を示します。

記述例 9-1　　構造構文 generate

```
declare x {
    output f[8];
}
module x {
    integer i;
    variable v[8];

    generate(i=0;i<10;i++) {
        v=v+i;
    }
    f = v;
}
```

```
generate(i=0;i<10;i++){
    v=v+i;
}
↓
v1 = v0 + 0 ;
v2 = v1 + 1 ;
v3 = v2 + 2 ;
v4 = v3 + 3 ;
v5 = v4 + 4 ;
v6 = v5 + 5 ;
v7 = v6 + 6 ;
v8 = v7 + 7 ;
v9 = v8 + 8 ;
v = v9 + 9 ;
```

generate 構造構文の中が 1 クロックの動作として展開されます。
つまり、記述例 9-1 は構造展開後に右側のリストのように展開され、最終的に v には 45(8'b0100_0101)が転送されます。
この構造構文で、バレルシフタや乗算などの展開が容易になります。

9.2 構造構文 if

generate 中などで整数変数の状態によって使用する回路を変えたい時に使えるのが構造構文 if です。
条件が整数変数のみで構成される if ブロックは、構造構文として構造展開されます。

構造構文 if は以下のように記述します。

if （整数変数条件）動作 1
else 動作 2

整数変数条件が真の時に動作 1 が、偽の時に動作 2 が合成されます。
記述例 9-2 に構造構文 if の例を示します。

記述例 9-2 構造構文 if 記述例

```
// Randam Generator
declare glfsr {
    func_in next_rand;
    output q[16]; //  乱数出力
}
module glfsr {
    reg r[16] = 0x39a5; //  乱数の種
    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]; // i が13,12,10 の時こちらが選択される
            } else {
                v[i] = r[i+1]; // i が13,12,11 以外の時こちらが選択される
            }                   // 部分代入をしているので、variable 端子を使っている
        }
        v[15] = r[0];
        r:=v;
        q = r;
    }
}
```

9.3 整数変数

整数変数 integer は以下のように宣言します。

integer 整数変数名

integer は内部構成要素の宣言部分で宣言を行います。整数変数 integer は 32bit の符号付き整数が入力可能です。

9.4 一時端子

一時端子 variable の宣言方法は以下のようになります。

variable 一時端子名 [ビット幅]

variable は内部構成要素の宣言部分で宣言を行います。variable のビット幅は省略することも可能です。省略した場合は 1bit 幅になります。一時端子 variable は内部端子と異なり、同じ端子名を使い回すことが可能です。
また variable は初期化が必要なく、宣言した時点で初期値は 0 に設定されます。
一時端子への代入は、文法上のあいまいさがない場合、右辺に整数を使うことができます。
また、2 項演算でコンパイル時にビット数が確定できる場合は 2 項目に整数を許可します。

一時端子の特徴として、他の端子では許可されていない部分代入が可能です。
記述例 9-3 に部分代入の例を示します。

記述例 9-3 一時端子への部分代入

```
declare subrange interface {
    input a[8];
    output f[8];
}
module subrange {
    variable v[8];
    v[3:0] = a[7:4];
    v[7:4] = a[3:0];
    f=v;
}
```

10 構造体

構造体の宣言方法は 4 章で解説しました。本章では構造体のインスタンス宣言と、動作記述内での構造体メンバに関する記述方法を解説します。

モジュール外部で構造体の宣言をした後、module 内で構造体のインスタンス宣言を行います。
インスタンス宣言時に信号の型を明示します。指定できる型は
reg または wire です。
reg で宣言する場合、初期値を与えることもできます。

構造体名 reg インスタンス名 =<初期値 >
構造体名 wire インスタンス名

サブモジュールの宣言と同じように、宣言時にインスタンス名に[]で数を指定することでインスタンスに多重度を持たせることが可能です。例えば

someting reg anything[5];

と記述した場合、anything[0]から anything[4]の 5 つのインスタンスができます。

多重度を持つインスタンスの初期値設定は、以下のように記述します。

someting reg anything[5] = {0,2,4};

この場合、多重度 5 ですが初期値は 3 つしかないので、残りの 2 つ (anything[3],anything[4])には 0 が入ります。

インスタンスおよび各メンバに対して、独立に参照、転送が可能です。
インスタンスおよびメンバへの転送は

インスタンス名 := 転送元 (インスタンスが reg の場合)
インスタンス名 = 転送元 (インスタンスが wire の場合)

インスタンス名 メンバ := 転送元 (インスタンスが reg の場合)

インスタンス名 メンバ = 転送元（インスタンスが wire の場合 ）

と記述します。
インスタンスおよびメンバを参照するには

転送先 := インスタンス名（転送先が reg の場合 ）
転送先 = インスタンス名（転送先が wire の場合 ）

転送先 := インスタンス名 メンバ（転送先が reg の場合 ）
転送先 = インスタンス名 メンバ（転送先が wire の場合 ）

と記述します。

記述例 10-1 に構造体の記述例の例を挙げます。
記述例 10-1　　構造体記述例

```
struct strtest {
    test1[3];
    test2[4];
    test3;
};
declare st{
}
module st{
    reg r1[8],r2[3];
    strtest wire mmw ;
    strtest reg mmr ;

    r2 := mmwtest1;
    mmr := 8'h93;          // mmrtest1 に3b'100, mmrtest2 に4b'1001, mmrtest3 に1b'1 が転送される
    mmwtest2 = 0xa;
    r1 := mmw;
}
```

11 修飾子

11.1 インターフェース修飾子

NSL 処理系では通常、順序回路で用いるクロック信号とリセット信号は言語上で隠蔽し、下位言語の生成時にクロック入力端子、リセット入力端子を自動生成しています。このため、通常 NSL モジュールは単相クロックの回路となります。
しかし、
interface 修飾子を declare 構文につけることでクロック入力端子、リセット入力端子を自動生成しないようにできます。そのため、多相クロックを利用する場合などリセット・クロック信号を明示的に用いる必要がある回路でも、問題なく記述することができます。

注意 :interface 修飾子の有無にかかわらず、モジュール内で順序回路を記述した場合、生成する回路のリセット信号は p_reset、クロック信号は m_clock という名前で自動合成されます。(信号名は合成時オプションで変更が可能です)

インターフェース修飾子の記述方法は以下の通りです。

```
declare    モジュール名        interface {
  //入出力構成要素
}
module    モジュール名        {
  //内部構成要素
  //動作記述部分
}
```

記述例 11-1　　**インターフェース修飾子記述例**

```
declare if_test_adder4 interface {   // 外部モジュール
    input m_clock ;                  // Clock input
    input p_reset ;                  // Reset input
    input add_a[4] ;                 // Add value A
    input add_b[4] ;                 // Add value B
    output result_q[4] ;             // Result value Q
}
module if_test_adder4 {
    reg r1[4] = 0 ;
    r1 := add_a + add_b ;
    result_q = r1 ;
}
declare if_test {                    // メインモジュール宣言
    input sysclk ;                   // Clock input
    input sysrst ;                   // Reset input
    input add_a[4] ;                 // Add value A
    input add_b[4] ;                 // Add value B
    output result_q[4] ;             // Result value Q
}
module if_test {                     // メインモジュール定義
    if_test_adder4 adder4 ;
    {
        // ******** Input signals ********
        adder4m_clock = sysclk ;     // 外部モジュールのm_clock 端子にsysclk を接続
        adder4p_reset = sysrst ;     // 外部モジュールのp_reset 端子にsysrst 接続
        adder4add_a = add_a ;        // 外部モジュールのadd_a 端子にadd_a を接続
        adder4add_b = add_b ;        // 外部モジュールのadd_b 端子にadd_b を接続
        // ******** Output signals ********
        result_q = adder4result_q ;  // result_q 端子を外部モジュールのresult_q 端子に接続
    }
}
```

記述例 11-1 はインターフェース修飾子を使用した例題です。
例題では if_test がトップモジュール、if_test_adder4 がサブモジュールです。
if_test_adder4 には interface 修飾子がついているので、
if_test_addr4 のリセット信号"p_reset"とクロック信号"m_clock"の自動生成は行われません。
そのため、if_test_adder4 では、データ入力端子の宣言で p_reset と m_clock を宣言しています。
このインターフェース修飾子によって if_test_adder4 のリセット信号とクロック信号が直接の信号名として、

モジュール内で明示できました。
また、トップモジュール if_test から、sysclk,sysrst というクロック、リセット信号を直接渡すことで、サブモジュール if_test_addr4 を直接制御することが可能です。

11.2 simulation 修飾子

simulation 修飾子は、該当モジュールがシミュレーション専用であり、論理合成対象ではないことを NSL コンパイラに示します。

12 パラメータ

本章では、パラメータの宣言方法について解説します。
NSL で記述するモジュール自体はパラメータをサポートしません。

しかし、NSL は、サブモジュールとして、 Verilog HDL/VHDL/SystemC で記述されたモジュールも利用することができます。 それらの言語でパラメトリック構文を利用しているモジュールのために、NSL はパラメータを記述できます。

パラメータの宣言例

param_int	**パラメータ名**	**//整数値型パラメータ（符号付き 32bit 整数 ）**
param_str	**パラメータ名**	**//文字列型パラメータ（制限なし。処理環境のメモリに依存 ）**

記述例 12-1 は、パラメータの使用例です。

記述例 12-1　　パラメータの使用例

VerilogHDL で書かれた以下のようなモジュールがすでにあるとします。

```
module lower_mdl (
        a ,
        b ,
        q ,
        Add
);
        parameterNofA = 4 ;
        parameterNofB = 6 ;
        input    [NofA-1:0]        a ;
        input    [NofB-1:0]        b ;
        output   [(NofA+NofB-1):0]  q;
        input    Add;

        assign #1 q = ( Add == 1'b1 ) ? ( a + b ) : 0 ;

endmodule
```

このモジュールをサブモジュールとして使いたい場合、上位となるモジュールのサブモジュール宣言は以下のようになります。

```
/* include parameter table */
#define  Num_of_A     8                    // サブモジュール向けパラメータ設定
#define  Num_of_B    12                    // サブモジュール向けパラメータ設定
#define  Num_of_Q    (Num_of_A+Num_of_B)   // サブモジュール向けパラメータ設定

/* 'Parametalized Adder' */

declare  parametalized_adder  interface {   // サブモジュールをinterface で宣言
        param_int  NofA ;                   // パラメータ宣言
        param_int  NofB ;                   // パラメータ宣言

        input    a  [Num_of_A] ;
        input    b  [Num_of_B] ;
        output   q  [Num_of_Q] ;
        func_in  Add(a, b) ;
}
```

そして、declare で宣言されたサブモジュールは, 上位モジュールからは以下のように呼び出します。
このとき、実体化したインスタンス名にパラメータを付加することで、サブモジュールに整数値また文字列を受け渡すことが可能です。

```
/* Declare a 'TOP module" */ // 上位モジュール宣言
declare  TOP_module {
         input    add_a    [Num_of_A] ;
         input    add_b    [Num_of_B] ;
         output   result_q [Num_of_Q] ;
         func_in  Add(add_a, add_b) ;
}
/* Equation of 'TOP module' */
module   TOP_module {
         // サブモジュールのインスタンス生成とパラメータ渡し
         parametalized_adder  u_adder  ( NofA = Num_of_A, NofB = Num_of_B ) ;
         func Add {
                  result_q = u_adderAdd(add_a, add_b)q ; // サブモジュール実行
         }
}
```

13 付録

13.1 ディレクティブ

13.1.1 include ディレクティブ

NSL では C 言語と同じように include ディレクティブを使って外部ソースファイルを取り込むことができます。
include ディレクティブの記述方法は以下の通りです。

#include "ファイルパス名 "
#include <ファイルパス名 >

この include ディレクティブにより、モジュール単位で NSL ファイルを管理することが容易になります。
ファイルパス名を◇でくくった場合は環境変数 NSL_INCLUDE で指定された標準インクルードパスからファイルを取り込みます。

include ディレクティブの記述例を以下に示します。
記述例 13-1　　include 記述例

```
※ inc_test モジュールと同じディレクトリ( フォルダ) に"sub_testnsl" というファイルを置いた場合。

#include "sub_testnsl"
// ↓コンパイル時に、ここに"sub_testnsl" が展開される。

declare inc_test {
   // 入出力構成要素記述
}
module inc_test {
   // 内部構成要素記述
   // 動作記述
}
```

13.1.2 define/undef ディレクティブ

NSL で記述したモジュールをサブモジュールとして呼び出す場合、パラメータを与えるために define ディレクティブがあります。(VerilogHDL/VHDL/SystemC で記述されたモジュールにパラメータを与える場合はパラメータ構文を利用します)
define ディレクティブは C 言語と同じように、文字列や式を別の文字列などに置換するディレクティブです。
例えば、"0'b0"を"ZERO"と置き換えることが可能となります。ただし、NSL 予約語は置き換えられませ

ん。
記述法としては、
#define 定義する文字列 置き換えられる定数および式

となります。文字列は大文字小文字を区別します。
定義した文字列は NSL のソース中で使うことができます。定義した文字列をモジュール名などの識別子中で利用するには、文字列を%%で囲います。
また、定義した文字列に対して+/−で定数を加算、減算するように記述することが可能です。

また、undef ディレクティブを使うことにより、定義されている文字列を解除することができます。以下のように記述します。

#undef 定義済み文字列
define ディレクティブの記述例を以下に示します。
記述例 13-2　　define 記述例

```
#define N 8                      //N に8 を定義する
declare test_%N% {               // 識別子にN を利用
   input test_in[N];             // データ入力端子のビット幅としてN を利用
   output test_out[N-1];         // データ出力端子のビット幅としてN-1 を利用
}
module test_%N% {                // 識別子にN を利用
   test_out = test_in[N-2:0];    // データ入力端子のN-2 から0 ビット目をデータ出力端子に転送
}
```

この NSL コードは以下のように展開されます。

```
declare test_8 {                 // N を8 に置換
   input test_in[8];             // N を8 に置換
   output test_out[7];           // N-1 を7 に置換
}
module test_8 {                  // N を8 に置換
   test_out = test_in[6:0];      // N-2 を6 に置換
}
```

13.1.3　ifdef / ifndef / else / endif ディレクティブ

NSL では、C 言語と同じ ifdef や endif といったディレクティブを使うことができます。NSL の標準プリプロセッサでは以下のディレクティブがサポートされています。

・ifdef
・ifndef
・else
・endif
使い方は以下の通りです。

#ifdef 定義する文字列

シンボル名が定義されていた時に、else または endif ディレクティブまでが有効になります。

#ifndef 定義する文字列

シンボル名が定義されていなかった時に、else または endif ディレクティブまでが有効になります。

#else

ifdef/ifndef ディレクティブの条件が成立しなかった時、endif ディレクティブまでが有効になります。

#endif

ifdef/ifndef/else ディレクティブの効果範囲を終了させます。

また、C 言語のプリプロセッサを使うこともできます。
記述例 13-3 に ifdef/ifndef/else/endif ディレクティブの例を示します。

記述例 13-3　　ifdef/ifndef/else/endif 記述例

```
#define DEBUG                    // シンボル DEBUG を定義
declare test {
    input a[8];
    input b[8];
    #ifdef DEBUG                 // シンボルDEBUG が定義されていたら
        output d[8];             // この行をコンパイルする
    #else
        output q[8];             // 定義されていなかったら、この行をコンパイルする
    #endif
}
module test {
    #ifndef DEBUG                // シンボルDEBUG が定義されていなかったら
        q = a & b;               // この行をコンパイルする
    #else
        d = a & b;               // 定義されていたら、この行をコンパイルする
    #endif
}
```

13.2 システムタスク

NSL は Verilog HDL 及び SystemC を合成する際に限り、Verilog HDL/SystemC 互換のシステムタスクを使用できます。
システムタスクは主にデバッグの補助を行う構文で、シミュレーションに用います。以下の表 13-1 が NSL で使用可能なシステムタスクの一覧です。
VerilogHDL のシステムタスクは"$"で始まりますが、NSL の場合は"$"の代わりに"_"(アンダースコア)をつけます。

表 13-1 NSL におけるシステムタスクの種類

システムタスクコマンド	対応する Verilog HDL のシステムタスク	対応する SystemC の関数	意味
_display	$display	printf()	コマンドラインに値を表示
_monitor	$monitor	printf()	コマンドラインに値を表示
_finish	$finish	*sc_stop()*	シミュレーションの停止
_stop	$stop	sc_stop()	シミュレーションの一時停止
_time	_time(※1)	_time(※1)	シミュレーション時間を表す変数
_readmemb	$readmemb	_nsl_readmem(※2)	メモリファイル(2 進数)の読み出し
_readmemh	$readmemh	_nsl_readmem(※2)	メモリファイル(16 進数)の読み出し
_random	$random	rand()	32bit 幅の乱数
_init			初期化ブロックの記述
_delay			シミュレーションを一時遅延させる

(※1)　自動カウントアップのレジスタ_time を宣言し、利用します。
(※2)　SystemC のテンプレートとして _nsl_readmem を宣言し、利用します。

以下に NSL でサポートするシステムタスクの言語別対応を示します。

表 13-2 システムタスクの出力言語別対応

システムタスクコマンド	Verilog HDL	SystemC	VHDL
_display	○	○	×
_monitor	○	○	×
_finish	○	○	×
_stop	○	○	×
_time	○	○	×
_readmemb	○	○	×
_readmemh	○	○	×
_random	○	○	×
_init	○	○	×
_delay	○	○	×

○ :対応する　× :対応しない
システムタスクの使用方法は Verilog HDL と変わりません。
"$"の代わりに "_"(アンダースコア)をつけることで、後は VerilogHDL と同じようにシステムタスクを使用することが可能です。
また、これらの構文はシミュレーション用であるため、システムタスクを含むモジュールを論理合成してもシステムタスク部分は実回路に反映しません。

以下の表 13-3 にまとめたシステムタスクは、モジュールの宣言部 (declare)に simulation 修飾子をつける必要があります。これらは表 13-1 に挙げたシステムタスクの中でも特に合成対象回路の要素として使ってしまいやすいため、シミュレーション専用であることを設計者が明確に把握できるよう、simulation 修飾子を必須としています。

表 13-3 simulation 修飾子が必要なシステムタスク

_time	シミュレーション時間を表す 64bit 幅の変数
_random	32bit 幅の乱数
_init	リセット後自動的に実行される順序実行ブロック。モジュールの実行文より先に1つだけ記述可能。
_delay	_init もしくは seq の順序実行ブロック内で、指定する数値の遅延を行う。

13.2.1 _display と _monitor

_display の表記方法は以下のようになっています。

_display(″<フォーマット指示子・文字列 >″, <信号名 >, <信号名 >,)

_display は指定した信号の値とメッセージを標準出力に出力するシステムタスクです。
NSL 文中で_display を実行すると信号の値を出力した後、改行します。

また、_monitor の文法は以下のようになっています。

_monitor(″<フォーマット指示子・文字列 >″, <信号名 >, <信号名 >,)

_monitor は指定した信号の値とメッセージを標準出力に出力するシステムタスクです。
_monitor は_display と動作が異なり、NSL 文中で実行した場合、指定した信号の値が変化した時に限りメッセージを出力します。
_display、_monitor のフォーマット指示子を以下の表 13-4 **システムタスクのフォーマット指示子**に挙げます。

表 13-4 システムタスクのフォーマット指示子

%b	2 進数で出力
%o	8 進数で出力
%d	10 進数で出力
%x	16 進数で出力
%c	文字を出力

13.2.2 _time

_time はシミュレーション時間を表す 64bit 幅のシステム変数です。_display,_monitor,_finish の引数としてのみ使用可能です。_time はシミュレーション専用構文のため、利用する場合は declare に simulation 修飾を付ける必要があります。

_time を使う場合は、モジュールの宣言部(declare)に simulation 修飾子をつける必要があります。

システムタスクの使用例として_display と_monitor および_time を用いた例を以下に示します。

記述例 13-4　　システムタスク _displa y と _monitor および _time の例

```
declare test_task simulation {}

module test_task {

    reg trigger[4] = 0 ;
    reg r1[4] = 0 ;
    proc_name proc1, proc2 ;

    /* モジュールが起動すると同時にproc1 を起動する */
    trigger := { trigger[2:0], 0b1 } ;
    if(trigger == 0b0111) proc1() ;

    proc proc1 {
        /* r1 をカウントアップして10 になったらproc2 を起動する */
        r1++ ;
        if(r1 == 4'd10) proc2() ;

        _display("r1 = %d", r1) ; // 無条件でテキスト表示
        _monitor("T = %d", _time) ; // 値が変化したらテキスト表示
    }

    proc proc2 {
        /* proc2 が起動したらシミュレーション終了 */
        _finish() ;
    }
}
```

13.2.3 _finish, _stop

システムタスク

_finish はシミュレーションを停止するコマンドです。

_finish の表記方法は以下のようになっています。

_finish(”<文字列 >”)

文字列を出力する必要がない場合は、()を省略して

_finish

と記述することができます。

NSL 文中で

_finish を実行すると、標準出力に文字列を出力してシミュレーションが停止します。

以下にシステムタスク _finish の記述例を提示します。

記述例 13-5　　システムタスク _finish の例

```
declare test_finish {}
module test_finish {
    reg trigger[3] = 0 ;
    reg cnt[4] = 0 ;

    func_self exec_add() ;

    /* モジュールが起動すると同時にexec_add を起動する */
    trigger := { trigger[1:0], 0b1 } ;
    if(trigger == 3'b011) exec_add() ;

    func exec_add seq {
        /* cnt レジスタをカウントしつつコンソールにテキスト出力 */
        for(cnt:=0; cnt<10; cnt++) {
        _display("%d", cnt) ;
    }
    /* テキストを出力して終了 */
    _finish("neko") ;
    }
}
```

13.2.4 _readmemh,_readmemb

_readmemb と_readmemh は外部のファイルをメモリの初期値としてロードするシステムタスクです。
外部ファイルの中に数列を表記して、このシステムタスクでファイルを関連づけることで使用可能です。
外部ファイルは ASCII ファイルのテキストで数列を書き込みます。
数列が 2 進数の場合は_readmemb で読み込み、16 進数の場合は_readmemh で読み込みます。
_readmemb、_readmemh の表記方法は以下のようになっています。

_readmemb(″ファイル名″, メモリ名)
_readmemh(″ファイル名″, メモリ名)

NSL 文中の_readmemb の使用方法は_readmemh と変わりません。
_readmemb,_readmemh で読み込めるファイルの形式は、VerilogHDL の$readmemb,$readmemh に準拠します。

以下にシステムタスク_readmemh の記述例を提示します。

記述例 13-6　　システムタスク _readmemh の例

```
declare test_read {
    input in_adr[8], in_data[8] ;
    output outdata[8] ;
    func_in write(in_adr, in_data) ;
    func_in read(in_adr) ;
}
module test_read {
    /* 8 ビット幅、256 ワードのメモリを宣言 */
    mem memory[256][8] ;

    /* メモリにデータを書き込む関数 */
    func write {
        memory[in_adr] := in_data ;
    }

    /* メモリのデータを読み出す関数 */
    func read {
        outdata = memory[in_adr] ;
    }

    /* システムタスク、nekotxt の中に書かれた数値をmemory の初期値として宣言。 */
    _readmemh("nekotxt", memory);
}
```

13.2.5 _random

_random は乱数を返すシステムタスクです。32bit 幅の乱数が得られるので、必要なビット幅に切り出してください。
例えば 8bit 幅で切り出したい時は以下のように記述します。

f = _random[7:0]

VerilogHDL では $random のビット切り出しは対応していないため、VerilogHDL 合成時はいったん中間端子に受けた後、必要なビット幅に切り出して転送するような回路が合成されます。
また、乱数の種(seed)を渡す記述方法はサポートされていません。
_random を使う場合は、モジュールの宣言部(declare)に simulation 修飾子をつける必要があります。
次にシステムタスク_random の記述例を提示します。

記述例 13-7　　システムタスク _random の例

```
declare test_rand simulation { }

module test_rand {
    reg trigger[3] = 0 ;
    reg temp[8] = 0 ;
    reg cnt[8] = 0 ;
    func_self neko() ;

    /* モジュール起動とともに関数neko を起動 */
    trigger := { trigger[1:0], 0b1 } ;
    if(trigger == 3'b011) neko() ;

    func neko seq {
    /* ランダム値を20 回テキスト出力 */
    for(cnt:=0; cnt<20; cnt++) {
        _display( "random == %d", _random ) ;
    }

    /* シミュレーション終了 */
    _finish("End Simulation") ;
    }
}
```

13.2.6 _init と _delay

_init と _delay は、どちらもシミュレーション時にシミュレーションの実行を制御するシステムタスクです。
_init はリセット後自動的に実行される順序実行ブロックを表します。モジュールの実行文より先に1つだけ記述できます。

_delay は _init もしくは seq(ファンクションの順次実行ブロック)内でのみ使える、遅延を指定するシステムタスクです。
次にシステムタスク _init と _delay の記述例を提示します。

記述例 13-8　　システムタスク _init と _delay の例

```
declare init_test simulation { }
module init_test {
    _init {
        _display("Hello World: time = %d",_time);
        _delay(100);
        _finish("Hello World: time = %d",_time);
    }
}
```

このNSL から合成されるVerilog HDL をシミュレーションすると、以下のような出力結果が得られます。

```
Hello World: time = 2
Hello World: time = 103
```

13.3 予約語一覧

alt
any
declare
else
finish
for
func
func_in
func_out
func_self
generate
goto
if
inout
input
integer
interface
invoke
label
label_name
mem
module
output
param_int
param_str
proc
proc_name
reg
return
seq
simulation
state
state_name
variable
while
wire

#define
#else
#endif
#ifdef
#ifndef
#include
#undef

_display
_finish
_monitor
_random
_readmemb
_readmemh
_time