

NSL リファレンスマニュアル

2011/04/15
オーバートーン株式会社

詳細目次

0.NSL 基本要項	5	7. 制御端子のアクション記述	41
●用語解説	5	●制御内部端子	41
●値	6	●制御入力端子	42
●信号線	6	●制御出力端子	42
●数値表記法	6	●戻り値	43
●演算時のビット幅推測	7	●seq ブロック	43
●モジュール名や信号名に利用可能な文字	7	●ラベル	44
●コメントアウト	7	●while ブロック	46
●一文の終了	8	●for ブロック	47
●クロック信号、リセット信号について	8	●カウント型 for ブロック	48
●NSL 処理系の処理順序	8	8. サブモジュールのアクション記述	51
●演算子	8	9. プロシージャのアクション記述	53
●整数を含む演算の優先順位	10	10. ステートのアクション記述	57
●整数および整数変数のみの演算の優先順位	10	11. メモリに対するアクション記述	59
2. 入出力構成要素の宣言	13	12. 構造体	61
●データ入力端子 / データ出力端子の宣言	13	13. インターフェース	63
●データ入出力端子の宣言	13	付録 1. ディレクティブ	65
●制御入力端子 / 制御出力端子の宣言	13	●include ディレクティブ	65
3. パラメータの宣言	15	●define/undef ディレクティブ	65
4. 内部構成要素の宣言	17	●ifdef / ifndef / else / endif ディレクティブ	66
●内部端子の宣言	17	付録 2. システムタスク	68
●レジスタの宣言	17	●_display と _monitor	68
●制御内部端子の宣言	17	●_time	69
●サブモジュールの宣言	18	●_finish	69
●プロシージャの宣言	20	●_readmemh, _readmemb.	70
●ステートの宣言	21	●_random	71
●メモリの宣言	22	付録 3. 予約語一覧	72
●構造体の宣言	22		
5. 単位アクション / ブロック	25		
●単位アクション	25		
●値の転送	25		
●レジスタのインクリメント及びデクリメント	25		
●基本的な演算記述例	26		
●条件演算	32		
●ブロック	33		
●並列動作ブロックの記述	33		
●alt ブロック	34		
●any ブロック	35		
●if ブロック	36		
6. 構造構文	37		
●構造構文 generate	37		
●構造構文 if	38		
●整数変数 Integer	38		
●一時端子 variable	38		

図表目次

記述例題 2-1. 入出力構成要素の宣言例.....	14	記述例題 12-1. 構造体の記述例.....	62
記述例題 3-1. パラメータ構文の使用例.....	15	記述例題 13-1. インターフェース記述例.....	64
記述例題 4-1. 内部端子の宣言 / レジスタ宣言の使用例.....	17	記述例題 O1-1. include 記述例.....	65
記述例題 4-2. 制御内部端子の宣言使用例.....	18	記述例題 O1-2. define 記述例.....	66
記述例題 4-4. サブモジュール宣言におけるパラメータ使用例.....	20	記述例題 O1-3. ifdef/ifndef/else/endif 記述例.....	67
記述例題 4-5. プロシージャの宣言例.....	21	記述例題 O2-1. システムタスク <code>_display</code> と <code>_monitor</code> および <code>_time</code> の例.....	69
記述例題 4-6. 状態変数ステートの宣言例.....	21	記述例題 O2-2. システムタスク <code>_finish</code> の例.....	70
記述例題 4-7. メモリの宣言例.....	22	記述例題 O2-3. システムタスク <code>_readmemh</code> の例.....	70
記述例題 4-8. 構造体の宣言例.....	23	記述例題 O2-4. システムタスク <code>_random</code> の例.....	71
記述例題 5-1. 基本的な単位アクションの記述例.....	26	表 0-1. 演算子.....	9
記述例題 5-2. ビット演算の記述例.....	26	表 0-2. 演算子優先順位.....	10
記述例題 5-3. 算術演算の記述例.....	27	表 1-1. NSL 基本構造.....	11
記述例題 5-4. シフト演算の記述例.....	27	表 2-1. 入出力構成要素表.....	13
記述例題 5-5. ビット連結の記述例.....	28	表 5-1. 転送の種類.....	25
記述例題 5-6. 左辺でのビット連結.....	28	表 5-2. どこでも使えるブロック.....	33
記述例題 5-7. リダクション演算の記述例.....	29	表 5-3. 使える条件が制限されるブロック.....	33
記述例題 5-8. 論理演算の記述例.....	29	表 7-1. seq ブロック内でのみ使える構文.....	44
記述例題 5-9. ビット切り出しの記述例.....	30	表 O2-1. NSL におけるシステムタスクの種類.....	68
記述例題 5-10. ビットの並び順反転例.....	31	表 O2-2. システムタスクの出力言語別対応表.....	68
記述例題 5-12. ビット幅拡張の記述例.....	32	表 O2-3. システムタスクのフォーマット指示子.....	69
記述例題 5-13. 条件演算の記述例.....	33		
記述例題 5-14. alt ブロック記述例.....	35		
記述例題 5-15. any ブロック記述例.....	36		
記述例題 5-16. if ブロック記述例.....	36		
記述例題 6-1. 構造構文 generate.....	37		
記述例題 6-2. 構造構文 if 記述例.....	38		
記述例題 6-3. 一時端子への部分代入の記述例.....	39		
記述例題 7-1. 制御内部端子の記述例.....	41		
記述例題 7-2. 制御入力端子の記述例.....	42		
記述例題 7-3. 制御出力端子の記述例.....	43		
記述例題 7-4. seq ブロック記述例.....	45		
記述例題 7-5. seq ブロック : ラベルの例.....	46		
記述例題 7-6. while ブロックの記述例.....	47		
記述例題 7-7. seq ブロックを使った等価回路.....	47		
記述例題 7-8. for ブロックの記述例.....	48		
記述例題 7-9. for ブロックの動作詳細.....	48		
記述例題 7-10. カウント型 for ブロック.....	49		
記述例題 8-1. サブモジュール構文の記述例.....	52		
記述例題 9-1. プロシージャの記述例.....	54		
記述例題 9-2. invoke ・ 遠隔 finish 記述例.....	56		
記述例題 10-1. ステート記述例.....	58		
記述例題 11-1. メモリ記述例.....	59		

第 0 章

0.NSL 基本要項

●用語解説

ハードウェア記述言語 (HDL:Hardware Description Language)

ASIC、FPGAなどを設計する際に用いるハードウェア設計用のコンピュータ言語のこと。

NSL (Next Synthesis Language)

処理記述主体のハードウェア記述言語

構文

NSL 言語の文法に則って記述された文のこと。

モジュール (module)

NSL 記述の最小単位のまとまり。

アクション (Action)

1 クロックで動作する部分のこと。

アクション記述

モジュール内部の、アクションを表している要素のこと。

共通アクション記述

アクション記述の内、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。

単位アクション (Atomic Action)

モジュールのアクション記述部分で記述される、アクションの最小単位のこと。

プロシージャ (proc : procedure)

何度も繰り返し使用するアクションを集約した処理手続きのこと。

ファンクション (func : function)

アクション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。

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

モジュールに入力する信号と、モジュールから出力する信号を合わせた要素のこと。

データ端子

数値を入力および出力させる信号線のこと

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。

制御端子

モジュール内部に存在するファンクションを起動させるための信号線のこと。

内部構成要素

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のこと。

ブロック (block)

アクション記述中で、{} 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。

階層構造

建物の階のように層状に重なった構造のこと。階層構造を実現していると大規模なシステムを構築することが容易になる。NSL では階層構造を実現するためにサブモジュール構文が使われる。

ブロックトップレベル

どのブロックにも入れ子になっていない、独立したブロックのこと。

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

NSL 言語では左端を 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。

●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)
- ・ レジスタとメモリ: "1", "0", "U"(Undriven)
- ・ 制御端子とタスク: "1", "0"

●信号線

信号線は、モジュール間でデータをやり取りするのに使用する線です。

NSL では1ビットの "単線信号" を用意した場合、"1"、"0"、"X" が伝達可能です。

また、より多くの情報を送りたい場合は単線を複数本まとめた "束線信号" を用意します。

結束信号のビットの桁数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 から1桁目として数えます。

例えば6ビットの信号線にデータ "101010" を出力した場合、1桁目は0、2桁目は1、3桁目は0、4桁目は1、5桁目は0、6桁目は1となります。

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

ですが、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許可されていません。これは、"可読性が著しく低下しやすい"、"動作不良の温床となりやすい"といった理由からです。

束線信号の宣言方法は第4章にて、束線信号のアクション記述については第5章にて解説しています。

●数値表記法

NSL での数値表記は、Verilog HDL 型と、C 言語型の2種類があります。

Verilog HDL 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進数の英語 binary の略です。

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 Verilog HDL 型では、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

●演算時のビット幅推測

NSL では端子、レジスタへの転送における信号同士の演算には、幅の確定した信号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、幅の確定した信号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ため、記述量を低減しながらあいまい性のない記述を実現しています。

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ため、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。
ここで、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。

●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 "__" は誤読防止のため禁止しています。

●コメントアウト

NSL のコメントアウトは 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せんが、エリアコメントのネストはできません。

●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；

単位アクション記述；

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も可能ですが、著しく可読性が落ちるため推奨しません。

●クロック信号、リセット信号について

Verilog HDL や VHDL はある信号によって挙動を変化させる信号主体の言語です。それに対して NSL はまずモジュールの挙動を記述して動きを決定する、処理主体の言語です。

NSL はモジュールのクロック信号を自動で用意して単一または多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。

●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）は構造展開という処理で値を決定します。

●演算子

NSL の演算子は基本的には Verilog HDL 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 0-1 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 0-1. 演算子

ビット演算	論理演算
& : ビット毎の論理積	! : 論理否定
: ビット毎の論理和	&& : 論理 AND
^ : ビット毎の排他的論理和	: 論理 OR
~ : ビット毎の論理否定	
算術演算	リダクション演算 (※※)
+ : 加算	& : AND
- : 減算	~& : NAND
* : 乗算	: OR
++ : インクリメント (※)	~ : NOR
-- : デクリメント (※)	^ : EX-OR
	~^ : EX-NOR
シフト演算	その他
>> : 右シフト	num#(sig) : 符号付きビット幅拡張
<< : 左シフト	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
関係演算	sig[numA:numB] : ビット切り出し
== : 等しい	numA{numB} : リピート演算
!= : 等しくない	{sigA,sigB,...,sigX} : ビット連結
> : 左辺が大	if(条件文) sigA else sigB : 条件演算
< : 左辺が小	
>= : 左辺が大または等しい	
<= : 左辺が小または等しい	

(※) 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 0-2 のとおりの優先順位が存在します。

一文の式の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 0-2. 演算子優先順位

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

●整数を含む演算の優先順位

前述の通り整数のビット幅は構造展開時に決定しますが、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

●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。

第 1 章

1. 基本構造

本章では、NSL の基本体系について解説します。

NSL の記述の基本構造は表 1-1 の体系で構成しています。

(以降、リファレンス内で <> で囲まれた構文は省略可能です。)

表 1-1. NSL 基本構造

```
declare モジュール名 < interface > {
  < 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 >
  < アクション記述 >
  - < 共通アクション記述部分 >
  - < ファンクション記述部分 >
  - < プロシージャ記述部分 >
}
```

NSL は declare 構文と module 構文の 2 つで構成されています。

declare 構文では " 入出力構成要素の宣言 " と " パラメータ宣言 " を行います。

" 入出力構成要素の宣言 " は第 2 章で、" パラメータ宣言 " は第 3 章で解説します。

module 構文では " 内部構成要素の宣言 " と " アクション記述 " を行います。

" 内部構成要素の宣言 " は第 4 章で、" アクション記述 " は第 5 ~ 10 章で解説します。

NSL ではこの declare 構文と module 構文を合わせて一つのモジュールとします。

declare 構文と module 構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 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。

第 2 章

2. 入出力構成要素の宣言

本章では、入出力構成要素の宣言について解説します。

入出力構成要素は、モジュールに入力または出力されるデータ端子や制御端子のことを指します。

入出力構成要素を、NSL モジュール内で使用するには宣言文が必要となります。

NSL における入出力構成要素を以下の表 2-1 に示します。

表 2-1. 入出力構成要素表

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

入出力構成要素において、一度の宣言で複数の信号を宣言する場合は、信号名を複数記述してカンマ","で区切ることで可能となります。

●データ入力端子 / データ出力端子の宣言

データ端子とは数値のデータを送受信する信号線のことを指します。

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" を設定することが可能です。ビット幅を省略した場合は、1 ビット幅の信号線となります。

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]

●データ入出力端子の宣言

データ入出力端子は入力と出力の両方が可能な線のことを指します。

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省略可能です。

データ入出力端子の宣言方法は以下のようになっています。

inout 入出力信号名 [ビット幅]

●制御入力端子 / 制御出力端子の宣言

NSL ではデータ端子の他に制御端子を記述することが可能です。

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させる線のことです。

また "ファンクション" とはある一定のアクション記述を集約したものです。

ファンクションの記述方法は第 6 章にて解説します。

制御入力端子は外部のモジュールからファンクションを起動するために使う制御端子で、制御出力端子は外部のファンクションを起動させるために使う制御端子です。制御端子にビット幅を設定することはできません。

制御入力端子 / 制御出力端子は一つの信号名に対して仮引数を付帯させることが可能です。仮引数に指定された要素にはアクション記述内で与えられた実引数のデータが転送されます。つまり、実引数が一時的に転送される引き受け先となります。

複数の仮引数を付帯させる場合は、仮引数をカンマ","で区切ることで実現可能です。

制御入力端子 / 制御出力端子の宣言はそれぞれ以下のように記述します。

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。

仮引数を記述しなくてもファンクションからデータ端子を操作することは可能ですが、可読性を向上させるために仮引数を書くことが推奨しています。

ファンクションは、戻り値を返すための端子を持つことができます。戻り値端子はデータ端子ですのでビット幅を持つことも可能です。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)

入出力構成要素の宣言例を示した記述例題 2-1 に示します。

記述例題 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 reto 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 の各信号を後述のアクション記述で使用することが可能になります。

第3章

3. パラメータの宣言

本章では、パラメータ構文の宣言方法について解説します。

NSL では複数モジュールを階層構造で構成して回路を設計することができます。下位となるモジュールには NSL で記述したものだけではなく、Verilog HDL/VHDL/SystemC で記述したものも利用することができます。Verilog HDL/VHDL/SystemC でパラメトリック構文で記述されているモジュールのインスタンスを生成を制御するためにパラメータ構文があります。(NSL で記述されたモジュールはパラメータ構文をサポートしません。この場合は define オプションでパラメータを与えます。define オプションについては付録を参照してください)

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)
```

下位モジュールを作成するには、後述の第4章、第7章のサブモジュール構文を参照して下さい。
記述例題 3-1 は、パラメータ構文の使用例です。

記述例題 3-1. パラメータ構文の使用例

Verilog HDL 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
);

    parameter NofA = 4;
    parameter NofB = 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
```

このモジュールを下位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下位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 // 下位モジュール向けパラメータ設定
#define Num_of_B 12 // 下位モジュール向けパラメータ設定
#define Num_of_Q (Num_of_A+Num_of_B) // 下位モジュール向けパラメータ設定

/* 'Parametalized Adder' */

declare parametalized_adder interface { // 下位モジュールを 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 で宣言された下位モジュールは、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、下位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 {

  // 下位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_adder.Add(add_a, add_b).q; // 下位モジュール実行
  }
}

```

第 4 章

4. 内部構成要素の宣言

本章では、内部構成要素の宣言方法について解説します。

内部構成要素は配線、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アクション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", " で区切ることで、一度に複数の内部構成要素を宣言することができます。

●内部端子の宣言

内部端子はデータを複数の端子に配置するための "配線" を提供する構文になります。

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]

wire に転送された値は同一クロックサイクル中有効です。

wire 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。

また、wire に何も入力がない間は "X"(不定) として扱われます。

●レジスタの宣言

レジスタはクロック信号の立ち上がりで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で値が記録されます。

またレジスタは任意の自然数 (1,2,3,...) をビット幅で用意することが可能で、NSL では宣言時に初期値を設定することも可能です。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題 4-1 に示します。

記述例題 4-1. 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
    // アクションの記述
}
```

記述例題 4-1 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、後述のアクション記述で wire, reg をそれぞれを使用することが可能になります。

なお、初期値設定は省略することも可能です。その場合リセット時には "X"(不定) が入ります。

●制御内部端子の宣言

制御内部端子はモジュール内部の制御端子の信号で、記述されたモジュール内でのみ呼び出しが可能です。

また制御内部端子はファンクションと関連づけを行うことが可能です。(ファンクションのアクション記述は第 6 章で解説しています。) 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、内部端子の宣言で定義された wire のみ使用可能です。

次の記述例題 4-2 が、制御内部端子の宣言例です。

記述例題 4-2. 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;
  // アクションの記述
}
```

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。

制御内部端子のアクション記述は第 7 章を参照してください。

●サブモジュールの宣言

NSL では階層構造を記述するためにサブモジュール構文があります。サブモジュール構文を使用する場合は、上位モジュールでサブモジュール宣言を行います。

サブモジュール構文を使用する場合には、サブモジュールになるべき下位モジュールを用意する必要があります。サブモジュール化する予定のモジュールのことを " テンプレート " と呼びます。

サブモジュール宣言は、上位モジュールの中でテンプレートを指定し名前をつけて実体化します。この実体化したテンプレートを " インスタンス " (実体) と呼びます。

インスタンスを作成することで、

- ・ データ入力端子への値の入力
- ・ 制御入力信号の呼び出し
- ・ 出てくるデータ出力端子の値の受け取り
- ・ 制御出力信号の受け取り

が可能になります。

サブモジュール構文は以下の方法で宣言します。

テンプレート名 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。

テンプレート名 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。

テンプレート名 インスタンス名 [3]

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。

記述例題 4-3. サブモジュールの宣言例

```
// テンプレートとなる下位モジュール "test_sub"
declare test_sub {
  input a ;
  output f ;
}
module test_sub {
  // アクションの記述
}

// 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

  // アクションの記述
}
```

記述例題 4-3 のように記述することで上位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下位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題 4-4 に示します。

記述例題 4-4. 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 {
  // アクションの記述
}

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

  // アクションの記述
}

```

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るわけです。サブモジュール構文のアクション記述に関しては第7章を参照してください。

●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、共通アクション記述以外でプロシージャ専用のアクション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。

記述例題 4-5. 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

  // アクション記述
}
```

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。
プロシージャのアクション記述は第 8 章を参照してください。

●ステートの宣言

NSL にはステートマシンを実現するためにステート (state) 構文があります。ステート構文を使うとアクションに状態変数を持たせることができます。

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。ステートマシンの詳細は第 10 章で解説します。

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>,...

ブロック内で宣言した各ステートに対応するアクションは、そのブロックのアクション記述で定義します。
ブロックが構成要素にステート名がある時、そのブロックには独自の状態変数が作成されます。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。

記述例題 4-6. 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 {

  // 共通アクション記述部分
  {
    // 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アクション記述方法は第 10 章を参照してください。

●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[4] 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題 4-6 に挙げます。

記述例題 4-7. 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
}
```

記述例題 4-7 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。

メモリを使用したアクション記述は第 10 章を参照してください。

●構造体の宣言

NSL では、複数のビット幅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。

構造体を使うことにより、さまざまなビット幅を持つ信号をまとめて扱うことができるようになります。

構造体は、まず宣言を行います。宣言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;
  構造体メンバ 2;
  構造体メンバ 3;
  :
  構造体メンバ X;
};
```

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 12 章で解説します。

以下の記述例題 4-8 に構造体の宣言例を挙げます。

記述例題 4-8. 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;
}
```


第 5 章

5. 単位アクション / ブロック

NSL はハードウェア記述言語であるため基本的にアクションは並列実行されます。その並列実行される一つ一つのアクションの最小単位を "単位アクション" と呼びます。

また、NSL ではアクションを "ブロック" という単位によって振る舞いを変えて記述します。

本章では単位アクション、ブロックについて解説していきます。

●単位アクション

NSL では一つのアクションを単位アクションと呼びます。

主な単位アクションの詳細は順を追って解説していきます。

●値の転送

NSL において単位アクションの基本となるものが "転送" です。

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 5-1 が転送の種類です。

表 5-1. 転送の種類

wire/output/inout 転送	=
reg/memory 転送	:=

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

●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アクションがあります。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アクションを使った例を記述例題 5-1 に示します。

記述例題 5-1. 基本的な単位アクションの記述例

```

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;

  // 共通アクション記述開始
  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
}

```

記述例題 5-1 では、共通アクション記述部分に単位アクションが 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)

●基本的な演算記述例

前節では NSL の単位アクションで重要となる "転送" について説明しました。

次はアクション記述の基本となる演算についての例題を提示します。まず HDL の基本であるビット演算の例題を記述例題 5-2 に示します。

記述例題 5-2. 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
}

```

記述例題 5-2 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。

記述例題 11 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記述例題 5-3 に示します。

記述例題 5-3. 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
}
```

記述例題 5-3 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。

記述例題 5-4. 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
}
```

記述例題 5-4 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、記述例題 5-5 を示します。

記述例題 5-5. 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
}
```

記述例題 5-5 はビット連結の記述例です。
 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。以下の記述例題 5-6 に例を示します。

記述例題 5-6. 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。

以下の記述例題 5-7 に示します。

記述例題 5-7. 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
}
```

記述例題 5-7 は、リダクション演算の記述例です。

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を記述例題 5-8 に示します。

記述例題 5-8. 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 があたら真、それ以外は偽が 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 があたら真、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。

以下の記述例題 5-9 にリピート演算の記述例を示します。

記述例題 5-9. 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。

記述例題 5-10 を示します。

記述例題 5-10. ビット切り出しの記述例

```

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
}

```

記述例題 5-10 はビット切り出しの記述例です。

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。

例を記述例題 5-11 に示します。

記述例題 5-11. 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題 5-12 を示します。

記述例題 5-12. 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題 5-13 を示します。

記述例題 5-13. 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。

●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題 5-14 に示します。

記述例題 5-14. 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;
}

```

●ブロック

アクション記述は、NSL 内で単位アクションの振る舞いを決定する領域のことを指します。ここではアクション記述のうち 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アクション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。

またアクション記述のうち応用編となる制御端子、サブモジュール、プロシージャ、ステート、メモリのアクション記述については、後述の第 6 ～ 10 章にてそれぞれ解説します。

以下の表 5-2、表 5-3 にブロックの種類を挙げます。表 5-2 にはどこでも使うことができるブロックを、表 5-3 には特別な条件の時に使うことができるブロックを挙げています。本章では表 5-2 に挙げるブロックについて解説します。(表 5-3 に挙げるブロックは第 7 章にて解説します。)

表 5-2. どこ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 5-3. 使える条件が制限されるブロック

seq ブロック	seq { }	※ファンクションアクション (func) 内のみ
while ブロック	while (式) { }	※ seq ブロック内のみ
for ブロック	for(ループ変数初期値 ; ループ条件式 ; ループ変数変化値) { } または for(ループ変数 := 初期値 , 終値) { }	※ seq ブロック内のみ

●並列動作ブロックの記述

並列動作ブロックは、ブロック内の単位アクション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。

並列動作ブロックは、ブロック内の単位アクションを全て並列に動作させるブロックです。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
  単位アクション 1
  単位アクション 2
  単位アクション 3
  ...
  単位アクション X
}
```

並列動作ブロックは、alt,any,if,seq など単位アクションを記述するブロック内に並列記述を書く際に使用します。

● alt ブロック

alt ブロックは alternative ブロックの略で、条件に合ったアクション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。条件に合ったアクション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アクションのみです。

また、全ての条件に合わない場合のアクション記述として "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: 単位アクション 1      // 優先順位 高
  条件 2: 単位アクション 2
  条件 3: 単位アクション 3
  ...
  条件 N: 単位アクション N    // 優先順位 低
  else : 単位アクション X
}
```

alt ブロック記述例を記述例題 5-15 に示します。

記述例題 5-15. 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];

  // 共通アクション記述開始
  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アクション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。

● any ブロック

any ブロックは条件付きブロックで、条件に合ったアクション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アクションに優先順位がなく条件に合ったアクションが全て起動します。

また、全ての条件に合わない場合のアクション記述として "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: 単位アクション 1
  条件 2: 単位アクション 2
  条件 3: 単位アクション 3
  ...
  条件 N: 単位アクション N
  else : 単位アクション X
}
```

記述例題 5-16 に any ブロック記述例を示します。

記述例題 5-16. 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];

  // 共通アクション記述開始
  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。

● 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 構文と同一のアクション、つまり条件が真の時にアクション記述で示されるアクションが起動されます。

また、条件に合わない場合のアクション記述として "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) 単位アクション 1
else      単位アクション 2

```

記述例題 5-17. 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、記述例題 5-16 と記述例題 5-17 は等価回路です。

第 6 章

6. 構造構文

NSL から RTL への合成では、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）と構成要素による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。構造展開を実現するために用意されているのが構造構文です。

●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アクションになる構文です。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 ) {
    単位アクション 1
    単位アクション 2
    ...
    単位アクション 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

以下の記述例題 6-1 に構造構文 generate の例を示します。

記述例題 6-1. 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;
```

構造構文では generate 文の中が 1 クロックのアクションとして展開されます。

つまり、例題 6-1 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。

●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) 単位アクション 1

else 単位アクション 2

整数変数条件が真の時に単位アクション 1 が、偽の時に単位アクション 2 が合成されます。
記述例題 6-2 に構造構文 if の例を示します。

記述例題 6-2. 構造構文 if 記述例

```
// Random 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;
  }
}
```

●整数変数 Integer

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。

●一時端子 variable

一時端子 variable の宣言方法は以下ようになります。

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。

記述例題 6-3 に部分代入の例を示します。

記述例題 6-3. 一時端子への部分代入の記述例

```
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;
}
```


第 7 章

7. 制御端子のアクション記述

第 2 章で制御入力端子、制御出力端子の宣言方法を、第 4 章で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、制御内部端子のアクション記述、およびファンクションアクション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記述する線です。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。

●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。アクション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 アクション記述

ファンクションのアクション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。

以下の記述例題 7-1 が制御内部端子の記述例です。

記述例題 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

  // 共通アクション記述
  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。

●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 アクション記述

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。

記述例題 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 ;
}
```

この記述例題 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。

●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、他の制御端子と同様に仮引数を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モジュールの外部にあるのでファンクションの記述はありません。

記述例題 7-3 に制御出力端子の記述例を挙げます。

記述例題 7-3. 制御出力端子の記述例

```
declare func_out_test{
  input a ;
  input b ;
  output f ;

  func_out func_do(f) ;
}
module func_out_test{
  if(a & b) func_do(1'b1) ;
}
```

このように記述することで、制御出力端子を外に向けて出力することが可能となります。ここでは制御出力端子に仮引数 f、実引数 1 を持たせて外部に出力しています。

これで仮引数 f を通して実引数 1 を外部に出力する動作が実現できます。

● 戻り値

ファンクションから結果を戻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、戻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。（制御内部端子：4 章、制御入力端子、制御出力端子：2 章）

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。

● seq ブロック

次に func(ファンクションアクション) 内でのみ使用可能な seq ブロックについて解説します。

seq ブロックは sequential ブロックの略で、このブロック内では記述した上から順に 1 文ずつ、1 クロックごとにアクション記述が起動します。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 ブロックがブロックトップレベルにあり、その seq ブロック中からプロシージャを呼び出した場合には、呼び出したプロシージャの終了を待ってから seq ブロックの次の単位アクションが起動します。

また、seq ブロック起動中に再度 seq ブロックが起動させられた場合、同時に両方の seq が起動するのでパイプラインを構成することができます。

seq ブロックの記述例を以下に示します。

```
seq {
  単位アクション 1
  単位アクション 2
  単位アクション 3
  ...
  単位アクション X
}
```

seq ブロック内でのみ使える構文を表 5-4 に挙げます。

表 7-1. 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

●ラベル

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

  単位アクション 1
  goto ラベル名 2
ラベル名 1:
  単位アクション 2
  単位アクション 3
ラベル名 2:
  単位アクション 4
  goto ラベル名 1
}
```

seq ブロックの記述例として記述例題 7-4 を示します。

記述例題 7-4. seq ブロック記述例

```
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。

この seq ブロックの動作は、exec_add を呼び出すと、まず 1 クロック目に seq ブロック 1 行目の実行が開始されます。そして 2 クロック目に 2 行目、3 クロック目に 3 行目の実行が開始されます。

以上のようなアクションの流れで順番に実行が開始されます。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。

また、ラベルを使用した例を以下の記述例題 7-5 に示します。

記述例題 7-5. 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;             // 次のクロック label2 から実行

    label1 :                  //label1 を提示
      r1++;

    label2 :                  //label2 を提示
      if(r1 == 10) f = r1;
      goto label1;           // 次のクロックに label1 から実行
  }
}

```

● 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。

順次アクションであることをはっきりさせるため、1 アクションのみでもブロックの括弧 {} が必須です。

while ブロックのアクション記述例は以下の通りです。

```

while(条件){
  単位アクション 1
  単位アクション 2
  ...
  単位アクション 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。

クロックの詳細は以下の記述例題 7-6 と 7-7 を参照して下さい。

まず、記述例題 7-6 に while ブロック記述例を、記述例題 7-7 に seq で記述した等価回路を示します。

記述例題 7-6. 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();
  }
}

```

記述例題 7-7. 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();
  }
}

```

記述例題 7-6 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題 7-7 です。

● 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アクションを 1 アクション 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 ブロックのアクション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 ){
  単位アクション 1
  単位アクション 2
  ...
  単位アクション 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 ブロック内のアクションが起動します。また条件式が偽である場合は for ブロック内のアクション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 つずつ単位アクションを起動していきます。for ブロック内の単位アクションが全て終了すると、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 ブロック内のアクションが起動し、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 と同じく順次アクションであることをはっきりさせるため、1 アクション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。

記述例題 7-8. 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() ;
  }
}
```

記述例題 7-9. 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() ;
  }
}
```

記述例題 7-8 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題 7-9 です。記述例題 7-8 と記述例題 7-9 は等価回路です。

●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 ) {
  単位アクション 1
  単位アクション 2
  ...
  単位アクション 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アクションを実行
- ・ 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アクションを実行
- ・ 初期値 = 終値 の時は一回だけブロック内アクションを実行

```
for (i:=0,5)
```

と記述した場合、i は 0,1,2,3,4,5 の値を取り、ブロック内のアクションを 6 回実行することに注意してください。

記述例題 7-8 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題 7-10 に示します。

記述例題 7-10. 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() ;
  }
}
```


第 8 章

8. サブモジュールのアクション記述

第 4 章でサブモジュールの宣言について解説しました。本章ではサブモジュールのアクション記述について解説します。

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下位モジュールの各端子を操作したり、下位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下位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即値 (1,2,3,...) と整数変数 (integer) のみです。

また、下位モジュールの制御端子を呼び出すことも可能です。下位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、下位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である記述例題 8-1 を示します。

記述例題 8-1. 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

  // 共通アクション記述
  SUB.calc1(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 := SUB.calc2(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。
}
```

記述例題 8-1 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 中のアクション記述ではインスタンスである SUB 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。

第 9 章

9. プロシージャのアクション記述

第 4 章にて、プロシージャの宣言方法を記述しました。本章ではプロシージャのアクション記述部分を解説をします。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。

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 ()

また、プロシージャ内部の動作を記述する場合は以下のように記述します。

```
proc プロシージャ名 {  
    単位アクション 1  
    単位アクション 2  
    単位アクション 3  
    ...  
    単位アクション X  
}
```

以下の記述例題 9-1 にプロシージャの例を示します。

記述例題 9-1. 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

  // 共通アクション記述
  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();
  }
}
```

記述例題 9-1 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。

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。

次に、invoke について解説します。

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 はプロシージャアクション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()

以下の記述例題 9-2 に finish と invoke の記述例を示します。

記述例題 9-2. 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;
    }
  }
}
```

第 10 章

10. ステートのアクション記述

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、ステートのアクション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アクション記述が可能です。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。

以下の記述例題 10-1 にステートの記述例を示します。

記述例題 10-1. 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。

第 11 章

11. メモリに対するアクション記述

第 4 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アクション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。

記述例題 11-1. 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。

第 12 章

12. 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、アクション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。

記述例題 12-1 に構造体の記述例題の例を挙げます。

記述例題 12-1. 構造体の記述例

```
struct strtest {
    test1[3];
    test2[4];
    test3;
};

declare st{
}

module st{
    reg r1[8],r2[3];
    strtest wire mmw ;
    strtest reg mmr ;

    r2 := mmw.test1;
    mmr := 8'h93; // mmr.test1 に 3b'100, mmr.test2 に 4b'1001, mmr.test3 に 1b'1 が転送される
    mmw.test2 = 0xa;
    r1 := mmw;
}
```

第 13 章

13. インターフェース

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名はリセット信号は `p_reset`、クロック信号は `m_clock` という名前でも自動合成されます。（信号名はコンパイルオプションで変更が可能です）

インターフェース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  
    // アクション記述部分  
}
```

インターフェース文を記述例題 13-1 にて解説します。

記述例題 13-1. インターフェース記述例

```

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 *****
    adder4.m_clock = sysclk; // 外部モジュールの m_clock 端子に sysclk を接続
    adder4.p_reset = sysrst; // 外部モジュールの p_reset 端子に sysrst を接続
    adder4.add_a = add_a;    // 外部モジュールの add_a 端子に add_a を接続
    adder4.add_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 = adder4.result_q; // result_q 端子を外部モジュールの result_q 端子に接続
  }
}

```

記述例題 13-1 はインターフェース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 修飾がついているので、

if_test_adder4 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_adder4 を直接制御することが可能です。

付録

付録 1. ディレクティブ

● 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題を以下に示します。

記述例題 O1-1. 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_test.nsl" というファイルを置いた場合。

```
#include "sub_test.nsl"  
// ↓ コンパイル時に、ここに "sub_test.nsl" 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  
    // アクション記述  
}
```

● define/undef ディレクティブ

NSL で記述したモジュールを下位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。（Verilog HDL/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%%で囲みます。

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題を以下に示します。

記述例題 O1-2. 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
}
```

● 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。

記述例題 O1-3 に ifdef/ifndef/else/endif ディレクティブの例を示します。

記述例題 O1-3. 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
}
```

付録 2. システムタスク

NSL は Verilog HDL 及び SystemC へのシンセサイズに限り、Verilog HDL/SystemC 互換のシステムタスクを使用することが可能です。

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 O2-1 が NSL で使用可能なシステムタスクの一覧です。

NSL の場合は、シンセサイズの関係で "\$" の代わりに "_" アンダーバーをつけます。

表 O2-1. 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	\$stop	sc_stop()	シミュレーションの停止
_time	\$time	sc_time_stamp()	シミュレーション時間を表す変数
_readmemb	\$readmemb	-	メモリファイル (2 進数) の読み出し
_readmemh	\$readmemh	-	メモリファイル (16 進数) の読み出し
_random	\$random	rand()	32bit 幅の乱数

以下に NSL でサポートするシステムタスクの言語別対応表を示します。

表 O2-2. システムタスクの出力言語別対応表

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

○ : 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 "_" アンダーバーをつけることで、後は Verilog HDL 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た場合でもシステムタスク部分は実回路に反映しません。

● _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後、自動的に改行します。

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。

● **_time**

_time はシミュレーション時間を表すシステム変数です。_display, _monitor, _finish の引数としてのみ使用可能です。
_display, _monitor のフォーマット指示子を以下の表 O2-3 に、およびシステム変数 _time を含めた記述例題を記述例題 O2-1 に示します。

表 O2-3. 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
% h	16 進数で出力
% e	実数を指数表記で出力
% f	実数を 10 進数で出力
% c	文字を出力
% s	文字列を出力

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。

記述例題 O2-1. システムタスク _display と _monitor および _time の例

```

declare test_task {
  input a[4], b[4];
  output f[4];
}
module test_task {
  reg trigger[4] = 0;
  reg r1[4] = 0;
  proc_name proc1, proc2;

  trigger := { trigger[3:1], 0b1 };
  if(trigger == 0b0111) proc1();

  proc proc1 {
    r1 := r1 + 0x1;

    if(r1 > 10) proc2();

    _display("a = % d, b = % d", a, b);
    _monitor("r1 = % d, T = % t", r1, _time);
  }
  proc proc2 {
    f = r1;
    finish();
  }
}

```

● **_finish**

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>")

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。

記述例題 O2-2. システムタスク _finish の例

```

declare test_finish {
  func_in exec_add ;
}
module test_finish {
  reg sum[8] = 0 ;
  reg cnt[4] = 0 ;

  func exec_add seq {
    for(cnt:=0; cnt<10; cnt++) {
      sum := sum + 0x01 ;
    }
    _finish() ;
  }
}

```

● _readmemb, _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表記して、このシステムタスクでファイルに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表記を書き込みます。数列表記が 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 使用法は _readmemh と変わりません。

_readmemb、_readmemh で読み込めるファイルの形式は、Verilog HDL 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。

記述例題 O2-3. 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 {
  mem memory[256][8] ;

  func write { memory[in_adr] := in_data ; }
  func read { outdata = memory[in_adr] ; }
  _readmemh("neko.txt", memory);
}

```

● `_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]
```

Verilog HDL では `$random` のビット切り出しは対応していないため、Verilog HDL 合成時はいったん中間端子に受けてから、転送するようにしています。

次にシステムタスク `_random` の記述例を提示します。

記述例題 O2-4. システムタスク `_random` の例

```
declare test {
  output f[8];
}

module test {
  f = _random[7:0];
}
```

付録 3. 予約語一覧

alt	_display
any	_finish
declare	_monitor
else	_random
finish	_readmemb
for	_readmemh
func	_time
func_in	
func_out	A:
func_self	B:
generate	C:
goto	D:
if	E:
inout	F:
input	G:
integer	H:
interface	I:
invoke	J:
label	K:
label_name	L:
mem	M:
module	N:
output	O:
param_int	P:
param_str	Q:
proc	R:
proc_name	S:
reg	T:
return	U:
seq	V:
simulation	W:
state	X:
state_name	Y:
variable	Z:
while	
wire	
#define	
#else	
#endif	
#ifdef	
#ifndef	
#include	
#undef	

詳細目次

0.NSL 基本要項	4	7. 制御端子のアクション記述	34
●用語解説	4	●制御内部端子	34
●値	5	●制御入力端子	35
●信号線	5	●制御出力端子	35
●数値表記法	5	●seq ブロック	36
●モジュール名や信号名に利用可能な文字	6	●ラベル	36
●コメントアウト	6	●while ブロック	38
●一文の終了	6	●for ブロック	39
●クロック信号、リセット信号について	6	8. サブモジュールのアクション記述	41
●演算子	7	9. プロシージャのアクション記述	43
●演算時のビット幅推測	8	10. ステートのアクション記述	47
●NSL 処理系の処理順序	8	11. メモリのアクション記述	49
●整数を含む演算の優先順位	9	12. インターフェース	50
●整数および整数変数のみの演算の優先順位	9	付録 1. ディレクティブ	52
2. 入出力構成要素の宣言	11	●include ディレクティブ	52
●データ入力端子 / データ出力端子の宣言	11	●define/undef ディレクティブ	52
●データ入出力端子の宣言	11	●ifdef / ifndef / else / endif ディレクティブ	53
●制御入力端子 / 制御出力端子の宣言	11	付録 2. システムタスク	55
3. パラメータの宣言	13	●display と monitor	55
4. 内部構成要素の宣言	15	●time	56
●内部端子の宣言	15	●finish	56
●レジスタの宣言	15	●readmemh, readmemb	57
●制御内部端子の宣言	15	●random	58
●サブモジュールの宣言	16	付録 3. 予約語一覧	59
●プロシージャの宣言	18		
●ステートの宣言	19		
●メモリの宣言	20		
5. 単位アクション / ブロック	21		
●単位アクション	21		
●値の転送	21		
●レジスタのインクリメント及びデクリメント	21		
●基本的な演算記述例	22		
●条件演算	26		
●ブロック	27		
●並列動作ブロックの記述	27		
●alt ブロック	28		
●any ブロック	29		
●if ブロック	30		
6. 構造構文	31		
●構造構文 generate	31		
●構造構文 if	32		
●整数変数 Integer	32		
●一時端子 variable	32		