NEC

CyberWorkBench®

Pioneering C-based LSI Design

BDL Reference Manual
(Version 4.5.1)

BDL version 3.1

SCAD-CY-1300002

Contents

1. INTRODUCTION

1.1 What is BDL?

BDL (Behavioral design language) is a language based on C language with extensions for hardware description, developed to describe hardware at levels ranging from the algorithm level to the functional level. BDL is relatively easy to learn because it is based on a well-known programming language.

1.2 Features of BDL

Table 1 lists ways in which BDL extends C language in order to describe hardware.

Table 1. Features of BDL

Feature	Syntax	Description
Variable type (Physical type)	`ter/reg/var/mem/reset/clock, etc.`	Type declarations used to represent hardware (registers, memory, etc.)

Bit width	`(Start bit: bit width)`
	`(Start bit .. end bit)`
		Arbitrary bit specification

I/O type	`in/out/inout`	Input/output type declaration
External module specification	`outside/shared`	Specification of external module for memory/registers/ter/var
Process declaration	`process`	Specification of execution start function
Data transfer type	`::=`	Assignment operator for describing hardware structure
Operators	`::` (Concatenation operator)	Operator that concatenates (links) two variables
	`&>, |>, ^>, ~&>, ~|>, ~^>` (Reduction operator)	Operator that reduces the bit width of a variable to one bit
Timing descriptor	`$` (Cycle boundary)	Specification of a clock cycle
Control statement	`wait/watch` `dmux/nmux/allstates`	Special control statement used for hardware operations
Special constant	`HiZ`	Special constant for hardware

1. Variable type (Physical type)
   These types can be used in variable declarations for terminals, registers, etc., in order to represent hardware. They include ter (terminal type), reg (register type), var (variable type), and mem (memory type). For details, refer to the description of variable declarations in **Section 3**.

2. Bit width
   Any bit width can be specified for variables in BDL.
   There are two ways to specify bit width: (start bit : bit width) and (start bit..end bit)
   "(start bit : bit width)" specifies that bits are in ascending order and (start bit..end

bit) specifies that the bits are in descending order when the start bit's value is greater than the end bit's value, or in ascending order when the start bit's value is less than the end bit's value.

```
var(0:4)  a;    /* Ascending order:  start bit 0, bit width 4 */
reg(7..0) b;    /* Descending order: start bit 7, end bit 0   */
ter(0..3) c;    /* Ascending order:  start bit 0, end bit 3   */
```

For details, refer to the description of bit widths in **Section 4**.

3. I/O types

   These types are used in declarations of variables to specify circuit inputs and outputs. Variables are declared with in for input type port, out for output type port, or inout for bidirectional (I/O) type ports. For details, refer to the description of I/O variables in **Section 5**.

```
in    var(0:4)  a;  /* "in" variable of var type  */
out   reg(7..0) b;  /* "out" variable of reg type */
inout ter(0..3) c;  /* "I/O" variable of ter type */
```

4. External module specification

   Memory/registers/ter/var can be synthesized as external modules.

   Memory specified with outside, memory/registers /ter/var specified with “var array” or “shared” key words are considered as outside the circuit to be generated. The actual memory/register/ter/var is not generated in the circuit, instead, an I/O port which read/write the external memory/registers/ter/var is generated.

```
outside mem(0:8)  out_mem[256]; /* External memory    */
outside var(0:8)  out_var[128]; /* External var array */
shared  mem(0:8)  sh_mem[256];  /* Shared memory      */
shared  reg(0:8)  sh_reg;       /* Shared register    */
shared  ter(0:8)  sh_ter;       /* Shared ter         */
shared  var(0:8)  sh_var;       /* Shared var         */
```

5. Process declaration

   In C language, the function called "main" (hereafter referred to as the main function) is a special function, in that this main function is always called to start execution of a program. However, in BDL, program execution is started by a function declared by the process declaration (hereafter referred to as the process function). This means that each description must include a process function. It also means that a function other than "main" can be used in BDL description that is not intended to be compiled by a C compiler prior to simulating at a behavioral level.

```
in    ter(7..0) a, b;
out   ter(8..0) c;
process add()
{
    c = a + b;
}
```

6. Data transfer type

   In addition to the assignment operator (=) from C language, the following are provided in BDL to represent special connections for hardware.

```
o ::= o_r; /* continuous assignment */
a = b + 1; /* General transfer      */
```

   For details, refer to the description of data transfer types in **section 3.6**.

7. Operators
   - Concatenation operator

     This operator is used to concatenate (link) variables with other variables.

```
o(0:16) = a(0:8)::b(0:8);
```

     For details, refer to the description of concatenation operators in **section 13.3**.

   - Reduction operator

     This operator is used to reduce multi-bit data to single-bit data.

```
f(0:1) = ^>a(0:4);
```

     For details, refer to the description of reduction operator in **section 13.2**.

8. Timing descriptor

   The timing descriptor "$" is used to specify clock cycle boundaries.

```
/* state 1 */
o = 1;
$
/* state 2 */
o = 2;
$
```

   For details, refer to the description of timing descriptors in **Section 18**.

9. Control statements

   Control statements include nmux for representing a multiplexer, dmux for representing a multiplexer with a decoder, and allstates for representing a software reset.

```
o ::= nmux {
      when( c1 ): data1;
      when( c2 ): data2;
      default   : 0;
};
```

   For details, refer to the description of BDL special control statements in **Section 14**.

10. Special constant

    HiZ is a constant used to represent high impedance. For details, refer to the description of the special constant in **section 2.5**.

```
o(0:4) = HiZ;
```

1.3 Restrictions on Cyber

1.3.1 Non-supported C language syntax

BDL is an extension language of C language implemented in order to serve as a hardware description language. In the "Cyber" behavioral synthesis system, not all C language syntax is supported, since it assumes to describe hardware behavior. The following are principal functions in C language that are not supported by Cyber.

- Floating point constant, float/double type
- union
- Strings (A string is a sequence of characters enclosed by double quotes (“...”).)
- Enumeration variable array

```
enum abc {A=1, B=2, C=3} alpha[10];
```

- Restriction on functions
  – Recursive functions are not supported
  – Functions without processing (null functions) are not supported
- Restrictions on identifier length, bit width, number of array dimensions, etc.
  Restrictions are imposed on identifier length, bit width, number of array dimensions, etc.
- Restrictions on typedef
  – The typedef keyword for an array is not supported

```
typedef int ARRAY10[10];
ARRAY10 foo;
```

- Restrictions on malloc
  - Not supported in calloc or realloc
  - Not supported in malloc of in/out/inout/shared/outside type
  - Not supported in malloc which does not statically determine array size

```
in int ter a;
process main()
{
    char *p;
    p = malloc(a);
}
```

  - Not supported in malloc for pointer specifying multi-dimensional array

```
char (*p)[4];
p = malloc(12 * sizeof(char*));
```

  - In case malloc is used in a function besides process function, it cannot be used within goto converted function
    - ＊ When malloc exists in inline unrolled function, an error occurs when that function is called from goto converted function
    - ＊ However, in case malloc exists in functional unit converted function there will not be any error when that function is called from goto converted function

```
NG Example 1 :

/* Cyber func = goto */
void hoge(void)
{
    char *cp;
    /* malloc in the goto converted function is NG*/
    cp = malloc(10);
}

process bdlmain()
{
    hoge();
}
```

```
NG Example 2 :

/* Cyber func = inline */
void hoge(void)
{
    char *cp;
    cp = malloc(10);
}

/* Cyber func = goto */
void foo(void)
{
    /* NG as the function (hoge) having malloc in goto
       converted function (foo) is called */
    hoge();
}

process bdlmain()
{
    foo();
}
```

```
OK Example:

/* Cyber func = operator */
void hoge(void)
{
    char *cp;
    cp = malloc(10);
}

/* Cyber func = goto */
void foo(void)
{
    /* Function (hoge) having malloc is called in goto
       converted function (foo) and it is OK since hoge()
       is functional unit conversion */
    hoge();
}

process bdlmain()
{
    foo();
}
```

- C language's library functions
  Use of the following functions is supported even when stdio.h and stdlib.h files are not included.

- printf, fprintf, sprintf, scanf, fscanf, sscanf
  But, these are ignored when they are used in BDL source code.
  (Note)Since the actual arguments of these functions are ignored, we recommend that descriptions of expressions with side effects should not be included in the arguments.

- exit function
  This function is used to end a process.

1.4 Identifiers

Identifiers are character strings that represent variable or function names. These identifiers can include alphabet letters and numerals, the first character in the array must be a letter. The underscore character (_) is included in the set of letters that can be used. There is no limit to the maximum length of identifiers, but the behavioral synthesis system Cyber may limit their length.

Like C language, BDL is case-sensitive.
For example, the following declaration, which describes two variables, is possible.

```
int a, A;
```

However, there are some cases, such as in VHDL, where the output RTL language is not case-sensitive. In such cases, we recommend ignoring case sensitivity when coding in BDL. A function that automatically modifies variable names has been provided as an option in Cyber, in consideration of cases where variable name conflicts arise due to the lack of case sensitivity.

1.5 Keywords (reserved words)

BDL's keywords are listed below. These keywords cannot be used in function names, variable names, array names, etc.

```
Keywords shared with C
    auto break case char const continue default do double else
    enum extern float for goto if int long register return
    short signed sizeof static struct switch typedef union
    unsigned void volatile while CONFIDENTIAL

Keywords specific to BDL
    alias allstates bool bus caseof clock ctlvar defmod dmux
    exit fall fixed goto0 in inout input inside latch mem module
    nmux other out output outside par port procedure process
    reg reset rise sence shared template ter typename unl
    var wait watch when HiZ ZERO
    FX_RND FX_RND_CONV FX_RND_INF FX_RND_MIN_INF FX_RND_ZERO
    FX_SAT FX_SAT_SYM FX_SAT_ZERO FX_TRN FX_TRN_ZERO FX_WRAP
    FX_WRAP_SM
```

At present, there is no specification for the keywords bus, latch, module, procedure, sence, rise, fall, unl, and inside, which are intended for future use. Therefore, these keywords are not supported by the Cyber behavioral synthesis system. The keyword caseof has been changed to dmux but it remains a keyword. Further, ctlvar, other, par and port are no longer used, however, they are kept as keywords.

In addition to the BDL keywords listed above, there are some other keywords in the output language. Therefore, when coding in BDL, we recommend that these keywords in the output language should not be used in identifiers (refer to Appendix C).

1.6 Comments

BDL has two forms to introduce comments:

```
/* Comment */
// Comment
```

A block comment shall start with /* and end with */. A one-line comment shall start with the two characters // and end with a new line. Block comments shall not be nested.
Any comment that starts with the special character string "Cyber" is regarded as an attribute (for details, refer to **Section 19**).
Since one-line comments are not always supported by C language compilers, we recommend using only block comments when coding in BDL.

1.7 Preprocessor

Like C, BDL uses a preprocessor. The types of preprocessing control lines are listed below.

```
#include <filename>
#include "filename"

#define word replace_word

#undef word

#if OS == PC

#elif OS == Linux

#else

#endif

#ifdef BDL

#endif

#ifndef C

#endif

#line 10 "filename"
```

Further, unlike C Language, it is possible to substitute the contents of command by Macro substitution. Using this, it is possible to change the effect of attributes as explained in **Section 124**.

(Before Macro Substitution)	(After Macro Substitution)

`#define FOO BAR`
`/* FOO */`
	`/* BAR */`

2. CONSTANTS

Constant can be specified in decimal, hexadecimal, octal, or binary format. Special constants are provided to express the high impedance value. BDL's constants include integer constants, enumeration constants, and special constants. An example of an integer constant is shown below (for description of bit specifier, refer to **Section 4**).

```
Integer constant ::=
    {sign}[0-9]+ {suffix}{Bit specifier} /* Decimal number (Note 1)*/
          0[bB][0-1]+ {suffix}{Bit specifier}  /* Binary number */
          0[0-7]+ {suffix}{Bit specifier}      /* Octal number */
          0[oO][0-7]+ {suffix}{Bit specifier}  /* Octal number */
          0[xX][0-9a-fA-F]+ {suffix}{Bit specifier}/* Hexadecimal
                                                      number*/
Sign ::=
    (’+’ | ’−’)
Suffix ::=
    (’U’ | ’S’ | ’L’ | ’LL’| ’u’ | ’s’ | ’l’ | ’ll’)
```

(Note 1) The number strings that start with “0” and include two or more digits are not decimal numbers (such as 0129).

2.1 Decimal Constants

Decimal constants have sufficient bit width for expressing numerical values, and the width is not determined until they are referenced unless otherwise specified.

The bit width of a decimal constant can be specified explicitly by a bit specifier.

For reference of partial bit width, value of specified range can be referred if bit specification is in descending order but, if the bit specification is in ascending order it cannot be referred for constant value other than 0, -1. Value referred as partial bit width becomes unsigned value always through specification of partial bit (for description of partial bit width reference, refer to **section 4.3**).

A decimal constant becomes a 32 bit width constant when suffix 'L' or 'I' is added to it. If 'LL' or 'II' is added then it becomes a 64 bit width constant. It is not possible to specify 'L' or 'LL' and a bit specifier simultaneously.

When a decimal constant is positive, it is unsigned type and when it is negative it is signed type. When "U" suffix is appended, a decimal constant is unsigned regardless of its constant value, and when "S" suffix is appended, it is always signed type.

The Cyber behavioral synthesis system supports decimal constant values only up to 8192 digits.

```
    x(0:8) = 10;         /* "10" as 8-bit value  */
    x(0:6) = 10(0:6);    /* "10" as 6-bit value  */
    x(0:64) = 10LL;      /* "10" as 64-bit value */
<NG>x(0:64) = 10L(0:64); /* Cannot specify multiple bit width*/
    x(0:4) = 42(4..1);   /* "5" as 4 −bit value  */
<NG>x(0:4) = 42(1..4);   /* Cannot refer partial bit of ascending
                            order */
<NG>x(0:4) = 42(2:4);    /* Cannot refer partial bit of ascending
                            order */
    x(0:6) = 20S;        /* "20" as signed type   */
    x(0:8) = 30U;        /* "30" as unsigned type */
```

2.2 Binary, Octal and Hexadecimal Constants

Binary, octal, and hexadecimal constants have the bit width determined by the bit separators, unless otherwise specified. The bit width of these constants can be specified explicitly by a bit specifier or suffix such as 'L', 'LL'. However, partial bit field reference is not possible (for description of partial bit width reference, refer to **section 4.3**). These constants are unsigned unless a suffix is specified.

```
    x(0:4) = 0b1010;/* "10" as 4-bit value   */
    x(0:6) = 017;   /* "15" as 6-bit value   */
    x(0:12) = 0x0ff;/* "255" as 12-bit value */
    x = 0x0f0(6..3);/* "12" as 4-bit value   */
<NG>x = 0x0f0(3..6);/* Cannot refer partial bit of ascending
                       order */
<NG>x = 0x0f0(6:4); /* Cannot refer partial bit of ascending order */
```

2.3 Over 32-bit Constants

In case constants larger than 32 bits are specified in Cyber behavioral synthesis system then they are converted to expression with concatenation operator.

```
Before conversion:
    a(0:64) = 0xffff00000000ffff;

After conversion:
    a(0:64) = (unsigned ter(0:64))(-65536(0:32) :: 65535(0:32));
```

Due to certain restrictions over 32 bit constants cannot be used in following locations.

- Enumeration constant value
- Value of case label of switch & dmux statement
- Size, subscript of array.
- Bit specification of variable, constant etc.
- Bit field of structure

2.4 Enumeration Constants

Unless the number of an enumeration constant is given explicitly, its value is the same as in C language.

The bit width and type of enumeration constants are determined by the values of all enumeration constants that are declared by enum statements. The bit width of enumeration is defined as the sufficient and minimum bit width to represent the maximum value of the enumeration constants. If all enumeration constants are positive, the type of enumeration is defined as unsigned type. Otherwise it is defined as signed type.

However value that can be specified in enumeration constant ranges from -2147483648 to 2147483647.

```
enum optype { ADD = 1, SUB = 2, MUL = 3, DIV = 4 };

The enumeration constants ADD, SUB, MUL, and DIV have bit width
“3” and unsigned type to represent 4".
```

2.5 Special Constants

ZERO and HiZ are special constants. Their bit width can be specified by a bit specifier.

- ZERO has the same meaning as the constant "0" and has the bit width needed for the constant "0". Since it is currently possible to specify the bit width using a decimal value or a bit specifier, there is no need to use the special constant ZERO.

```
x = ZERO(0:17);
/*  ZERO(0:17) has the same meaning as 0b000000000000000000. */
```

- HiZ indicates high impedance.
- If the bit width is not specified by ZERO or HiZ, the assigned bit width will be used when referencing.

```
x(0:8) = HiZ; → x(0:8) = HiZ(0:8);
```

3. VARIABLE DECLARATION

Variables must be declared before they are used. As a declaration defines the nature of variables, it consists of a list of types and variable names. A declaration must have at least one declarator (physical type or logical type).

- Declaration type

Physical type	Variable, variable...;
Physical type (bit specifier)	Variable, variable...;
Physical type	Variable (bit specifier), variable (bit specifier)...
Physical type	Logical type Variable, variable...;
	Logical type Variable, variable...;

- Declaration example

```
reg                x, y, z;
ter(0:8)           x, y, z;
ter                x(0:8), y(0..4), z(7..0);
var      short     x, y, z;
         int       x, y, z;
```

3.1 Logical Type

The type that adds bool to int, char, short, etc., used in C language, is called a logical type. When signed or unsigned is not explicitly specified, types other than the bool type become the signed type. In case of bool type only, signed or unsigned specification is not permitted, and it becomes unsigned type.

In the case of the Cyber behavioral synthesis system, the various types indicate the following bit widths.

int	32 bits	char	8 bits
short	16 bits	long	32 bits
long long	64 bits	bool	1 bit
float	32 bits	double	64 bits
long double	64 bits		

3.2 Physical Type

The physical type is an original BDL type that is used to represent hardware. The physical type is indicated below.

```
         ter(0:8)    w;     /* ter type (terminal) */
         reg(7..0)   r;     /* reg type (register) */
  signed var(0..7)   v;     /* var type (variable) */
         mem int     M;     /* mem type (memory)   */
         reset       rst;   /* reset type (reset)  */
         clock       clk;   /* clock type (clock)  */
```

Any bit width can be explicitly specified using a bit specifier or logical type for each of the declarations. When no explicit bit is specified, the bit width is 1.

Moreover, it is possible to explicitly specify the signed type or unsigned type using the signed, unsigned, or the logical type. If there is no explicit signed or unsigned specification, nor logical type specification, the type is the unsigned type.

```
           reg         a; /* 1-bit unsigned type */
           reg(0:8)    b; /* 8-bit unsigned type */
           reg char    c; /* 8-bit signed type   */
signed     reg(0:8)    d; /* 8-bit signed type   */
unsigned   reg char    e; /* 8-bit unsigned type */
```

3.2.1 ter type (terminal)

This is a type for hardware terminals (pins), and does not hold values across the clock cycle boundaries. (The clock cycle indicates the rising edge or falling edge of the clock.)
If no value is assigned to the ter type variable, the value of that variable becomes 0. In other words, this means that in the next clock cycle, it is updated to 0.
However, in case synthesis is done in automatic scheduling mode of Cyber behavioral synthesis system, which does not have a clock cycle image then the ter type are synthesized as var type. However, as an exception, input/output variables and arrays are each synthesized to connote a terminal. (For details, refer to the Cyber Behavioral Synthesis System Reference Manual.)

3.2.2 reg type (register)

This is a type for hardware registers, and holds values across the clock cycle boundaries. However, values are not updated until the clock cycle ends. In other words, even if a value is assigned in a given state, updating of that value is not performed in that state, and the value is updated at the next clock cycle.
In case synthesis is done in automatic scheduling mode of Cyber behavioral synthesis system, which does not have a clock cycle image, then the specification of the reg type always implies a register from the viewpoint of synthesis. (For details, refer to the Cyber Behavioral Synthesis System Reference Manual.)

3.2.3 var type (variable)

This is a type with no corresponding hardware image, and is immediately updated to the assigned value and holds that value. In other words, variables are handled in the same way as in C language.

3.2.4 mem type (memory)

This is a type for hardware memory.
(For details, refer to "array/memory" in **Section 6**.)

3.2.5 reset type, clock type

This type is for hardware reset and the clock.

```
reset   rst;   /* reset */
clock   clk;   /* clock */
```

- Bit specification is not possible for clock/reset.
- Logical type and physical type specification are not possible.
- I/O types (in, out, inout) cannot be specified for clock/reset.

In the case of the Cyber behavioral synthesis system, even if reset/clock type variables are not declared, and if clock or reset is required because a register exists (etc.), a reset pin and clock pin are automatically generated. Moreover, declaring multiple clocks using clock is prohibited because it is then impossible to know which register belongs to which clock. To describe multiple clocks and resets, refer to "Specification of Clock Signal for Each Register and Specification of Reset Signal for Each Register" in the Cyber Behavioral Synthesis System Reference Manual.

3.3 Handling of Physical Type Variables in Cyber Behavioral Synthesis System

The Cyber behavioral synthesis system has two synthesis modes. For details, refer to the Cyber Behavioral Synthesis System Reference Manual.
Since the meaning of the physical type and restrictions, etc., differ according to the mode and variable type, so caution is required.

- Common items (for both manual and automatic scheduling mode)
  - The var type is recommended for the function parameters other than pointer variables. The physical significance of parameters other than the var type, such as the ter type and the reg type, are ambiguous, and currently parameters other than the var type are handled as the var type.
  - The var type is recommended as the physical type for the return values of functions. The physical significance of return values of functions other than the var type, such as the ter type and the reg type, are ambiguous.
  - Variable whose physical type has not been specified is treated as var type
  - The meaning when the physical type is specified for the in, out, and inout specification variables are described in **Section 5**.
- Automatic scheduling mode
  - Even if the ter type is specified for a variable, it is treated as the var type.
  - If the reg type is specified for a variable, it is treated as a variable that assigns value to register, and the reference to the assigned value is executed at the subsequent cycle.

3.4 Handling of Other Specifiers/Modifiers

1. const
   When const is attached to a variable, this means that the value of that variable cannot be changed.
2. static
   In the Cyber behavioral synthesis system, local variables declared as static are handled as external variables in order to assign a static storage class as in C language.
   Local variables for which static is declared within the function are changed to external variables with the name function name_variable name.
3. typedef
   typedef can be described when creating structural names, enumeration names, etc. It can also be used when creating multiple names.

```
typedef struct tnode TNode;
typedef enum abc ABC;

typedef struct tnode TNode, Tree;
typedef int Length;
```

4. signed, unsigned
   signed and unsigned specifier can be declared in the same manner as for normal C

language.
However, in the case of the Cyber behavioral synthesis system, the meaning may be changed by the synthesis options.
Normally, synthesis with signed and unsigned mixed together is performed, but a mode for synthesis of all variables as the unsigned type (synthesis option -Zunsigned) is also provided for backward compatibility with past specifications. Since support of synthesis option -Zunsigned is to be discontinued in the future, use of this option is not recommended.

5. auto, register, volatile
   auto, register, and volatile are meaningless as storage class specifiers in BDL. Currently, these specifiers are ignored if they are included in BDL input.

6. extern
   Refer to the description of extern specifications in BDL functions in **section 15.5.1**.

7. in/out/inout
   This specifier, which represents in, out, and inout, is unique to BDL. For details, refer to Section 5.

8. outside/shared
   These are storage class specifiers, also unique to BDL, which are used to represent hardware module configurations.

outside	Sets array/memory and registers as outside module (when moved to outside)
shared	Sets array/memory/registers/ter/var as outside module (when shared by multiple processes**)**

```
(Example)
outside mem(0:8) om[10] ;
shared  mem(0:8) sm[10] ;
shared  reg(0:8) sra[10] ;
shared  reg(0:8) sr ;
shared  ter(0:8) sta[10] ;
shared  ter(0:8) st ;
shared  var(0:8) sva[10] ;
shared  var(0:8) sv ;
```

3.5 Enumerated Type Variables

In BDL, the enum type is extended in the following ways compared to the C language specification.

- Bit width and unsigned/signed type of enumeration type
  An enum variable is signed if it has at least one enumerator with an assigned negative constant, and it is unsigned if it does not have any such enumerators.
  The bit width of the enum type will be the width needed for expressing all the constant values assigned to the various enumerators. However, any bit width that has been specified to an assigned constant is ignored.
  Enumerators has the bit width and signed, unsigned type similar to the enumerated type.
  Enumeration variable has bit width and unsigned, signed type similar to enumeration type. .Specification of bit specifier and signed, unsigned is not possible in enumeration variable.

```
enum months { Jan = 1, Feb, Mar, Apr, May, Jun,
                   Jul, Aug, Sep, Oct, Nov, Dec};
enum months x;     /* 4 bit unsigned */

typedef enum optype {ADD = -10, SUB = 0,
                     MUL = 10, DIV = 20} OP_TYPE;
OP_TYPE x;         /* 6 bit signed */
```

The enumerated type “months” is unsigned since positive constants are assigned to all enumerators. Since 12 is the maximum number of constants that can be assigned to enumerators, the bit width is 4 bits.
The enumeration type enum optype is signed since a negative constant is assigned to at least one enumerator. Since the constant 20 has been assigned to an enumerator, the bit width is 6 bits.

- Physical type can be specified for enumeration type variable:
  Physical type of ter, reg, var can be specified in enumeration type variable. However, bit specifier cannot be described.

```
enum boolean { NO, YES };

ter enum boolean x_t; /* ter type */
reg enum boolean x_r; /* reg type */

typedef enum boolean BOOL;
var BOOL x_v;         /* var type */

ter(0:8) enum boolean ng_t; /* Bit specification is an error */
```

3.6 Data Transfer Methods

BDL has its own specification for data transfer methods. The following two data methods indicate the way in which assignments are performed.

=	load	Generic transfer
::=	always connect	continuous assignment

- The continuous assignment (::=) method indicates a physical connection. In other words, it indicates that assignments can be made regardless of the condition or state.

However, in automatic scheduling mode of Cyber behavioral synthesis system, error is generated as the continuous assignment (::=) method is out of scope of support.

3.7 Initial value at the time of declaration

Variable is initialized during declaration in the below manner.

```
var(0:8) l = 4;
var(0:8) M[4] = {1,2,4,8};
struct ST {
       var(0:8) s;
       var(0:8) X[2];
};
struct ST t ={1,{0,1}};
```

The initial value is given a special treatment during the synthesis depending upon the type of variable. For example Initial value of global variable is treated as the value of reset state at the time of synthesis. For details, refer to Chapter "Handling at the time of process function start, stop" of Cyber Reference Manual.
Moreover, refer to **Section 6.3** for Initialization of array **and Section 8.2.4** for initialization of structure.
Variable wherein explicit initial value has not been written contains an indefinite value. In C language, the static variable and external variable is implicitly 0 initialized. However, since it is indefinite in BDL, so it is necessary to describe 0 initialization explicitly in case initial value is necessary.

3.8 Variable declaration in for statement

In BDL, variable can be declared in the initial setting part of for statement in the same way as C++ language.

```
for (int i = 0; i < 10; i++) { ... }
```

4. BIT SPECIFIERS

Bit specifiers are used to declare any bit width for variables and to extract partial bits value from variables.
Two methods used to specify the bit width in a declaration are:
(1) using the bit specifier immediately after the physical type
(2) using the bit specifier immediately after the variable name.
The bit specifier is described immediately after the variable name when referencing or assigning partial bit fields.

4.1 Bit Specifier Used in Declaration

Coding methods for bit specifiers used in declarations are listed below.

Syntax

Physical type (start bit: bit width) ascending order
Physical type (start bit ..end bit) ascending or descending order

Variable name (start bit: bit width) ascending order
Variable name (start bit ..end bit) ascending or descending order

Example

```
var(0:32)  x;         /* Ascending order MSB  x(0), LSB x(31)  */
var(0..7)  x;         /* Ascending order MSB  x(0), LSB x(7)   */
var(7..0)  x;         /* Descending order MSB  x(7), LSB x(0)*/
var        x(0:8);    /* Ascending order MSB  x(0), LSB x(7) not
                         recommended */
var        x(31..0); /* Descending order MSB x(31), LSB x(0)
                         not recommended */
```

Notes on bit field declaration

- The bit specifiers start bit, end bit, and bit width must be specified by constants (integer constants) in declarations.

- Variable and expression cannot be described
- Floating-point constants cannot be described

- Duplicated bits cannot be specified.
  A syntax error will occur even if the start bit and bit width are the same in both bit specifications.

- Bits are specified in physical type and variable name

```
<OK> ter                           a(0:8);
<OK> ter(0:8)                      a;
<NG> ter(0:8)                      a(0:8);
```

- Bit width cannot be specified at the time of declaration for logical variables (this would be a duplicate specification since the logical type includes a bit width specification).

```
<NG> int                           a(0:8);
```

#int indicates 32 bits
#Duplicated specification by int (32 bits) and bit specifier (0:8)

- When a declaration has no explicit bit width specification, the bit width is 1 bit.

```
reg b; /* 1 bit */
```

4.2 Bit Specifier for Bit Field Reference

The description of a bit specifier for bit field reference is shown below. When some of the bits of a variable are referenced, it is called partial bit reference.

Syntax

Variable name (start bit: bit width) Ascending order
Variable name (start bit ..end bit) Ascending order or descending order

Example

```
reg(0:32) a;

x = a(0:32);
x = a(2..9);
x = a(s..e);    /* Specification through variable is also possible */
x = a(24:8);
x = a(s:w);     /* Specification through variable is also possible */
x = a(2);       /* → Abbreviated form, same meaning as a(2:1) */
x = a(s);       /* → Abbreviated form, same meaning as a(s:1) */
```

Notes on bit field reference

- Variables that are declared in descending order can only be specified by bit specifiers that are in descending order. Similarly, variables that are declared in ascending order can only be specified by bit specifiers that are in ascending order.
- When referencing, the bit width or end bit can be abbreviated in the description. In such cases, the bit width is always 1 bit.
- Bit specifiers cannot be specified for a value returned from a function.

```
<NG> x(0:4) =      func(a)(24:4);
                   ~~~~~~~~~~~~~~
     var(0:32) func(var(0:8) a)
     {
         var (0:32) rtn;
              .
              .
         return       rtn;
     }
```

- Bit specifiers have higher priority than all other operations.
  For example, a+b(0:4) means adding a to b(0:4), not (a+b) (0:4).
- Values obtained through partial bit field reference are regarded as unsigned values regardless of the original variable's signed/unsigned state.
- In BDL description, the value obtained by partial bit referencing, where variable is used in bit specification, is always treated as unsigned type. When bit specification of the variable is used, the value will be unsigned type, even if bit specification had been expressing all bits.
- In partial bit referencing description part, where variable is used in bit specification, the behavior becomes undefined in case bit position and bit width exceeds the bit range specified at the time of variable declaration.

4.3 Bit Specifiers for Assignments

The coding of a bit specifier for assignments is described below. When values are assigned to some bits of a variable, it is called partial bit assignment.

Syntax

Variable name (start bit: bit width)	Ascending order
Variable name (start bit..end bit)	Ascending order or descending order

Example

```
reg(0:32) x;

x(0:32) = a;
x(2..9) = a;
x(s..e) = a;    /* Specification through variable is also possible */
x(24:8) = a;
x(s:w)  = a;    /* Specification through variable is also possible */
x(2) = a;       /* → Abbreviated form, same meaning as x(2:1) */
x(s) = a;       /* → Abbreviated form, same meaning as x(s:1) */
```

Notes on bit field assignment

- Variables that are declared in descending order can only be specified by bit specifiers that are in descending order. Similarly, variables that are declared in ascending order can only be specified by bit specifiers that are in ascending order.
- When assigning, the bit width or end bit can be abbreviated in the description. In such cases, the bit width is always 1 bit.
- Restrictions on partial bit field assignment under Cyber behavioral synthesis system

- Restrictions for manual scheduling mode

When a value is assigned to a partial bit field, if the type is reg or mem, the expression is converted to an intermediate expression that assigns the previous value to any bit other than the bits in the partial bit field assignment. The previous value remains after the partial bit assignment. If the type is ter, the expression is converted to an intermediate expression that assigns 0 to any bit other than the bits in the partial bit field assignment. Consequently, even if multiple descriptions specify partial bit field assignment in which the target partial fields are not overlapped, multiple assignments can still occur.

(Example)

```
          reg(0:8) a;
          ter(0:8) b;
          ter(0:8) t;
       a(0:4) = b(0:4);
       t(0:4) = b(0:4);
               ↓
   a(0:8) = b(0:4)::a(4:4);
   t(0:8) = b(0:4)::0(0:4);
```

- In partial bit assignment description part, where variable is used in bit specifier, the behavior becomes undefined in case bit position and bit width exceeds the bit range specified at declaration.

4.4 Structure Bit Field

In C language, the structure bit field is used to represent a set of contiguous bits in a single memory unit, but in BDL it simply indicates a bit width specification.

```
struct {
    unsigned int is_keyword : 4; -> unsigned ter(0:4) is_keyword;
    unsigned int is_extern  : 2; -> unsigned ter(0:2) is_extern;
    unsigned int is_static  : 2; -> unsigned ter(0:2) is_static;
} flags;
```

When the BDL's physical type is not specified, as in the above example, the variables are regarded as a ter type variable with specified bit-width.

```
struct {
    reg int is_keyword : 4; signed reg(0:4) is_keyword;
    var int is_extern  : 2; signed var(0:2) is_extern;
    ter int is_static  : 2; signed ter(0:2) is_static;
} flags;
```

When the BDL's physical type is specified in the description, the variables are regarded as having the bit width of the specified bit field.
Bit specifications cannot be made simultaneously via a structure bit field and a BDL bit specifier.

```
<NG>
    struct {
        reg(0:4) is_keyword : 2 ;
        var(0:2) is_extern  : 2 ;
         :
    } flags;
```

5. I/O SPECIFICATION

In BDL, I/O specifiers (in, out, and inout) are used to specify variables for logic inputs and outputs.

- in: input type
- out: output type
- inout: bidirectional (input/output) type

5.1 Notes on I/O Variables

- Input specifiers cannot be used with the mem type. Instead, outside and shared should be used to specify that memory is outside the module.
- In the Cyber behavioral synthesis system, in order to use I/O specifiers with variables in an array, all subscripts to be used when referencing or assigning the array must be constants (for details, refer to **section 5.5**).
- in, out, inout can be specified for global variables and pointer variables.

```
in      ter(0:8) a;
out     ter(0:8) b;
inout   ter(0:8) c;
in      var(0:8) *d;
out     var(0:8) *e;

process moduleA (){

     in ter (0:8) *p;
    out ter (0:8) *q;
          .
}
```

- in, out and inout can be specified for member variables of structure. However, they are restricted in structure and pointer variable when used as global variable. Further, in, out, inout, shared, outside, clock, reset and pointer variable cannot be specified for structure variable included as a member variable of structure.
- in, out can be specified in member variable of defmod.
- The arguments of process function that can be specified by in, out and inout are as follows:

- Can be specified in pointer variable and array variable

- Cannot be specified for variable that is not an array.

- However in Cyber5.0, it does not support pointer variables and structure variables. Further, specification is possible for variable that is not an array, however, as it is planned to be obsolete it is not recommended.

5.2 Physical Types That Can Be Specified for I/O Variables

In the Cyber behavioral synthesis system, physical types that can be specified for I/O variables vary according to the scheduling mode. List of types that can be specified for I/O variables in each scheduling mode has been shown in **Table 2**.

Table 2 List of Physical Types that Can Be Specified for I/O Variables

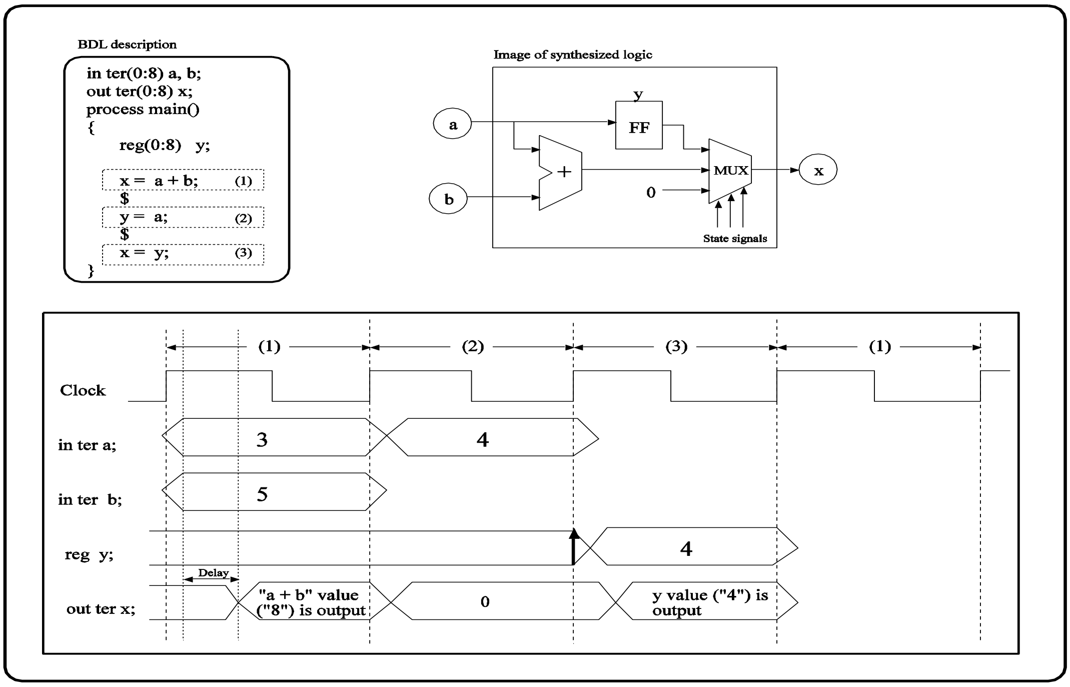
		Scheduling mode	
		Manual	Auto
in	ter	○	○
	reg	–	○
	var	○	○
out	ter	○	○
	reg	○	○
	var	○	○
inout	ter	○	○
	reg	–	–
	var	○	○

○ : Can be specified

– : Cannot be specified

5.3 Handling of I/O Variables in Manual Scheduling Mode

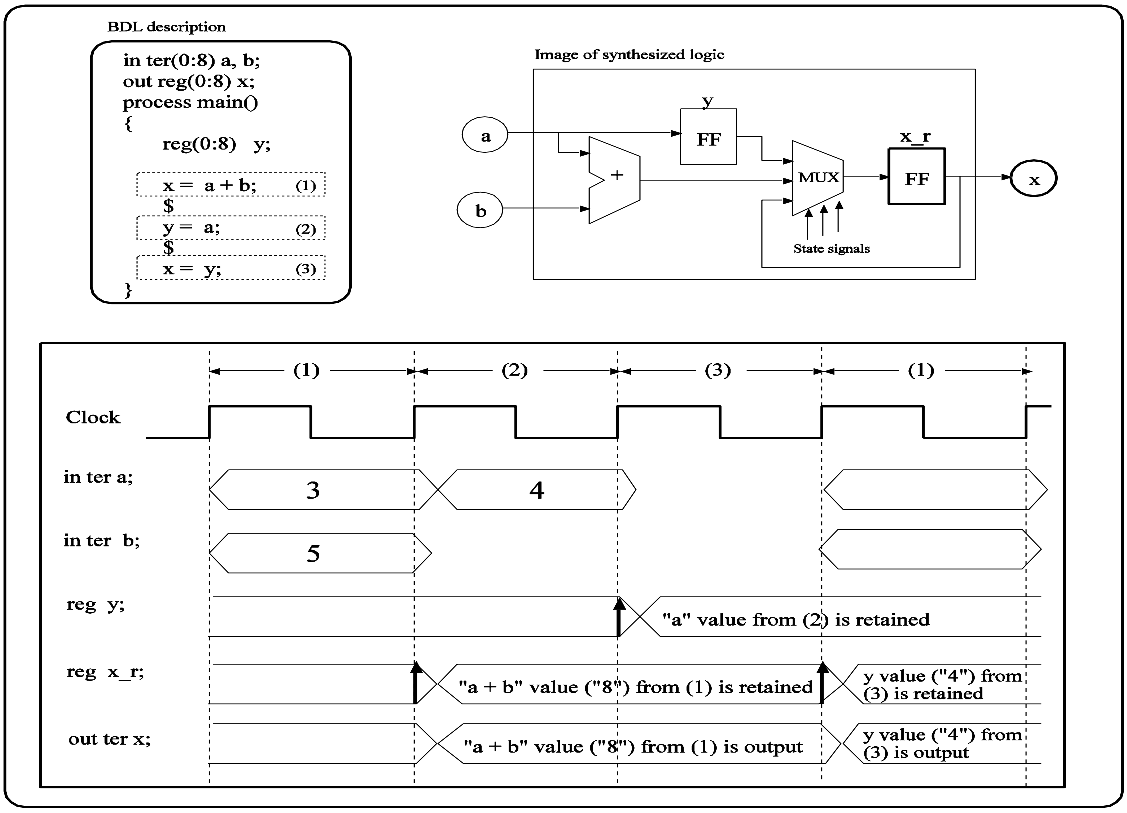
- in ter
  When reference occurs in a certain state, data which is input from an external source during that state is captured to this variable. The same values will be input if multiple references occur during the same state. Values cannot be assigned to this type of variables.
- out ter
  When assignment occurs in a certain state, data is output to an external destination during that state. Since this output is done without transfer via registers, it entails some risk of hazard. Also, caution is required concerning delays between the output and the registers outside the module. 0 is output during the state in which no value is assigned. Value of ter type output port cannot be referenced (read).
- inout ter
  When assignment occurs in a certain state, data is output during that state, and when reference occurs during a certain state, data which is input from an external source during that state is captured. The same values will be input if multiple references occur during the same state. If no values are referenced or assigned, high impedance output occurs. The behavior is undefined for the case where assignment and reference occur during the same state.



BDL description

```
in ter(0:8) a, b;
out ter(0:8) x;
process main()
{
     reg(0:8)   y;

   x =  a + b;       (1)
     $
   y =  a;           (2)
     $
   x =  y;           (3)
}
```

- in reg, inout reg
  These cannot be used with manual scheduling mode.
- out reg
  Data is output at the rising (or falling) edge of the next clock in the state where assignment occurred. In this configuration, registers are directly connected to external pins.
  Previous values are retained if there is no assignment. In other words, these values are retained until the next assignment occurs. Value of reg type output port can be referenced. (read).

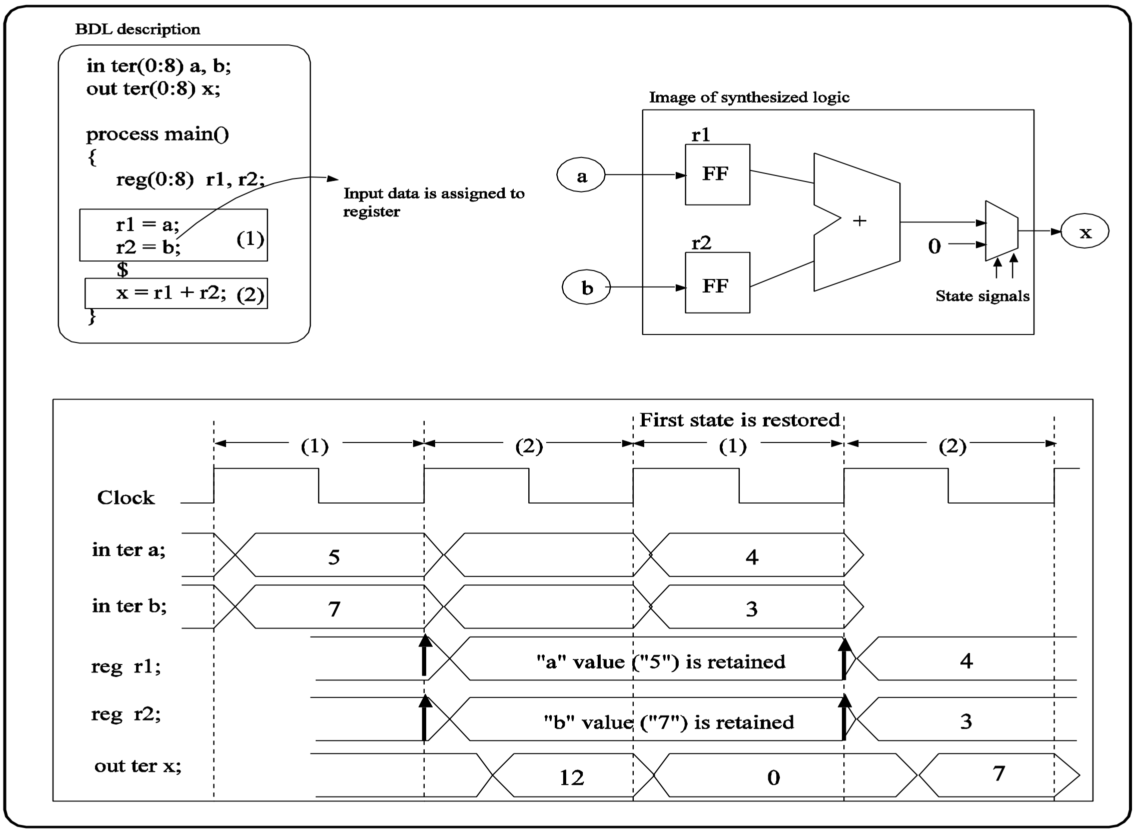


BDL description

```
in ter(0:8) a, b;
out reg(0:8) x;
process main()
{
    reg(0:8)   y;

    x =  a + b;        (1)
    $
    y =  a;            (2)
    $
    x =  y;            (3)
}
```

Image of synthesized logic
a
b
y
FF
+
MUX
x_r
FF
x
State signals
(1)
(2)
(3)
(1)
Clock
in ter a;
3
4
in ter b;
5
reg y;
"a" value from (2) is retained
reg x_r;
"a + b" value ("8") from (1) is retained
y value ("4") from (3) is retained
out ter x;
"a + b" value ("8") from (1) is output
y value ("4") from (3) is output

Use of the in reg variable is prohibited with manual scheduling mode. In order to retain input data in registers, the reg variable must be declared explicitly and the value of in ter variable should be assigned to this reg variable.



BDL description

```
in ter(0:8) a, b;
out ter(0:8) x;

process main()
{
    reg(0:8)  r1, r2;
    r1 = a;
    r2 = b;           (1)
    $
    x = r1 + r2;  (2)
}
```

5.3.1 Input Output variables of var type

In case of ter and reg type input/output variables, input data is taken from external source at the time of reference (input) and output data is driven at the time of assignment (write) without using input() and output() functions. Whereas var type input output variable explicitly uses input() function for data input from external source and output() function for driving output data externally.

- in var

  "in var" type variables take data input from external source explicitly by specifying input function.

  When it does not use input function for reference, it does not accept input from external source; however the value that has been read is retained. Also, value can be assigned.

```
in var (0:8)   a;
   var (0:8)   x;

input (a);
$
a = input(a);
$
x = input(a);
```

- In input(a) data is input from input variable a and the value is retained in variable "a."
- a = input (a) means that data is input from input variable "a" and value are retained in variable "a". It has same meaning as input(a).
- x = input (a) means that data is input from input variable 'a' and values are retained in variable 'a' while at the same time value are assigned variable x.
- when input function of same "in var a" exists multiple times in the same cycle then it results in an error.

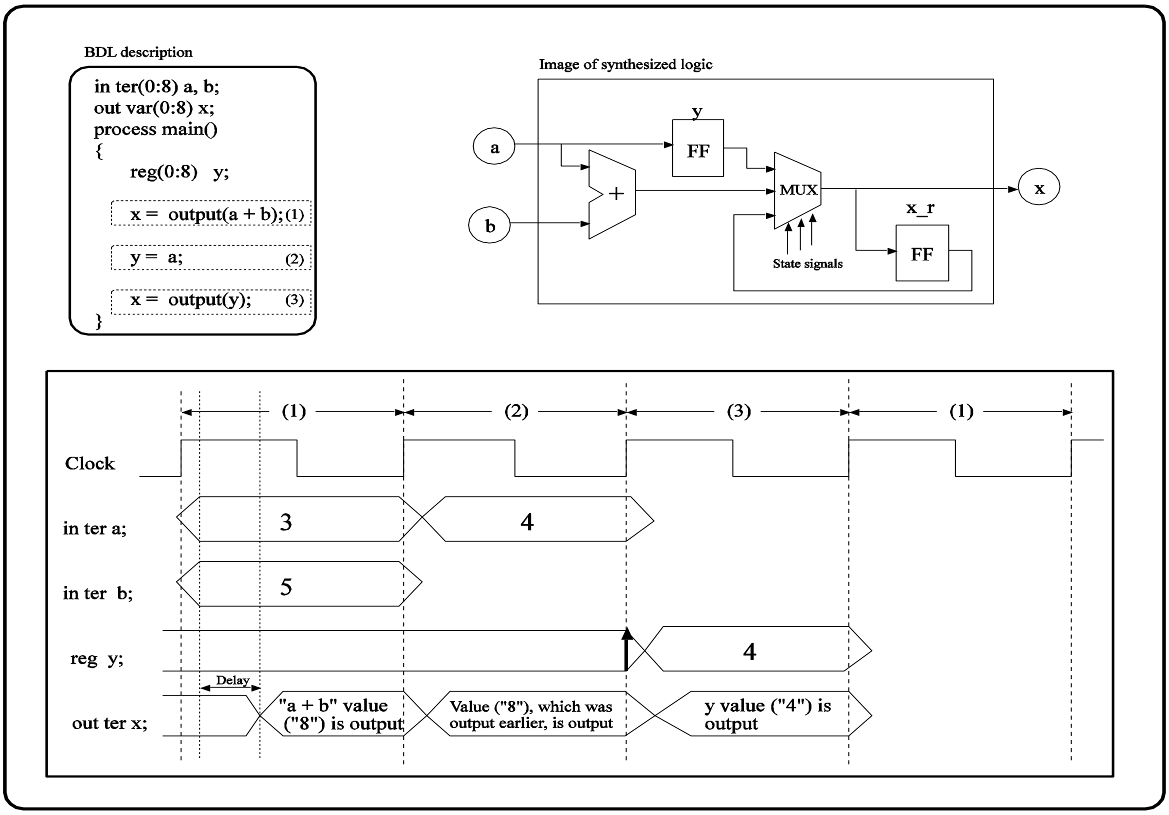
- out var

  An output function should be used to explicitly specify that data is output externally from the out variable of a var type. If an assignment occurs without using an output function, values will be retained but data will not be externally output. However, as an exception, the initial value on declaration with initialization would be output externally even if it has not used 'output'.

  An output function use example is shown below.

```
out var(0:8) b;
    var(0:8) x,y;

output(b);
$
b = output(b);
$
```

- In output (b), values retained by variable b are output to output variable b.
- b = output (b) means that values retained by variable b are output to output variable b, while at the same time assigned to b. It has same meaning as output (b).
- When output function is not executed, the previous value is output, but this entails some risk.
- The Cyber behavioral synthesis system provides a method of changing output attribute of the out var variables by specifying attribute, for example, "port_type" attribute can change the characteristic of the port.
- When output function of same out var b exists multiple times in the same cycle then it results in an error.



BDL description

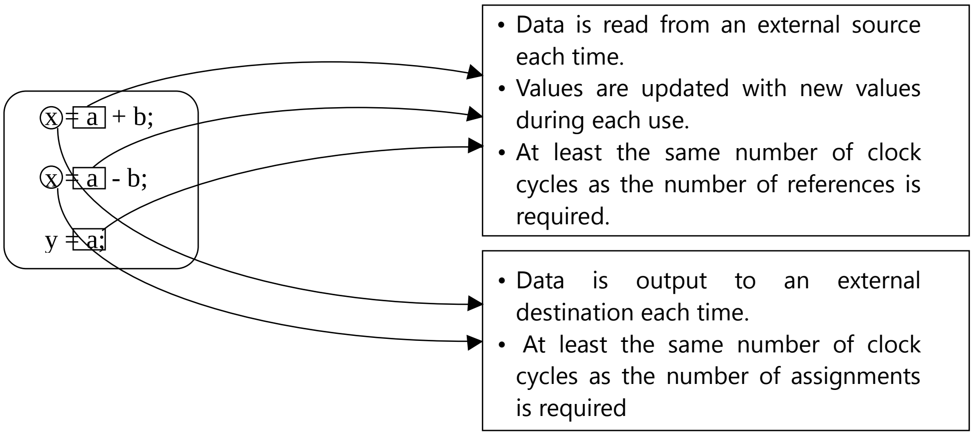
```
in ter(0:8) a, b;
out var(0:8) x;
process main()
{
     reg(0:8)   y;

     x =  output(a + b); (1)

     y =  a;             (2)

     x =  output(y);     (3)
}
```

- inout var
  The var type's inout variable is used with an output function when externally outputting data, and is used with an input function when inputting data from an external source. If an assignment is performed without using an output function, values will be retained but data will not be externally output. High impedance output occurs if no values are being output.

5.4 Handling of I/O Variables in Auto Scheduling Mode

With auto scheduling mode, data is input each time an in ter variable, in reg variable, or inout ter variable is referenced. Similarly, data is output to an external destination each time an out ter variable, out reg variable, or inout ter variable is assigned.

BDL description
- a and b are input variables
- x and y are output variables



- Data is read from an external source each time.
- Values are updated with new values during each use.
- At least the same number of clock cycles as the number of references is required.

- Data is output to an external destination each time.
- At least the same number of clock cycles as the number of assignments is required

The order of referencing or assigning to the same I/O variables is guaranteed, but the order of referencing or assigning to different I/O variables is not guaranteed. In the Cyber behavioral synthesis system, when the same input variable is referenced several times in one expression as shown in the following example, an error will occur because the order is uncertain between each reference. In such cases, an assignment to another variable must be made in order to set a certain order.

×

```
(Example)
    in ter(0:8) a;

    x = a + a;
```

○ Same values

```
in ter(0:8) a;
var(0:8) a_v;

a_v = a; /* Input1 */
x = a_v + a_v;
```

○ Different values

```
in ter(0:8) a;
var(0:8) a_v;

a_v = a;      /* Input1  */
x = a_v + a; /* Input 2 */
```

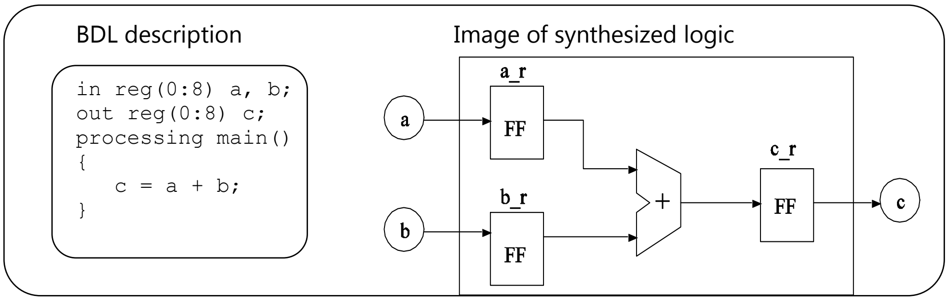
- in ter

Caution is required concerning delays from outside the module to the input, since the referenced data is used for other operations during the same state. It is not possible to assign values to ‘in ter’ type variables.

- out ter

Since operation results are externally output without transfer via registers, this entails some degree of risk. Also, caution is required concerning delays between the output and the registers outside the module. 0 is output if no values are being output. It is not possible to reference values of ‘out ter’ type variables.

- inout ter
  Assigned data is output externally during the same state without transfer via registers, and referenced data is used for operations during the same state. Either an assignment or a reference can be performed during a single state, but not both. High impedance output occurs if no values have been assigned.
- in reg
  Since referenced data is received via registers before being used internally, the circuit will be reliable in regards to delays. Value cannot be assigned to "in reg" type variables.
- out reg
  Since assigned data is received via registers before being externally output during the next assigned state, the circuit will be reliable in regards to delays. It is possible to reference values of 'out reg' type variables.
- inout reg
  This is currently not supported.



BDL description

```
in reg(0:8) a, b;
out reg(0:8) c;
processing main()
{
    c = a + b;
}
```

Image of synthesized logic

- in var, out var, inout var
  Handling of in var, out var, inout var in auto scheduling mode is same as manual scheduling mode (**section 5.3.1**). However, in auto scheduling mode input function and output function of same variable do not cause error and are always done in separate cycles.

5.5 I/O Array Variables

The following restrictions exist concerning the use of array variables for input or output under the Cyber behavioral synthesis system.

- All subscripts must be constants when referencing or assigning an array.

When all array subscripts are constants, references and assignments can be performed without using a decoder. However, the Cyber behavioral synthesis system does not support the case where input or output array subscripts include variables.

(Example)

```
in  ter in0[2];
out ter out0[2];

process main()
{
    reg x, y;

    x = in0[0];
    y = in0[1];
    $
    out0[0] = x + y;
    out0[1] = x - y;
    $
}
```

5.6 Structure I/O Variables

When structure variables are used for input or output, each member variable is regarded as a single I/O variable. Consequently, if structure variables are specified as input or output variables, I/O variables are created for each member variables, and the number of I/O pins becomes exactly the same number as the member variables.

[Example]

```
struct abc {
    ter(0:4) a;
    ter(0:8) b;
};
in  struct abc in0;
out ter(0:8) c;
process main()
{
    c = in0.a + in0.b;
}
```

The description example above is equivalent to the description example below.

```
in  ter(0:4) in0_a;
in  ter(0:8) in0_b;
out ter(0:8) c;
process main()
{
    c = in0_a + in0_b;
}
```

6. ARRAYS AND MEMORY

Register files and memory are described with array variables. When declaring an array variable, its size can be specified the same way as in C language, or its size can be specified via a start address and end address (for details of array synthesis, refer to the Cyber Behavioral Synthesis System Reference Manual.)

6.1 Declaration of Arrays and Memory

- Array declaration with its size , similar to C language (subscripts start from 0)

```
int       a[10]    /* (base 0, size 10)           */

reg(0:16) b[10];   /* Register array with decoder */
mem(0:16) c[10];   /* Memory array                */
mem(0:16) d[2][4]; /* Multi-dimensional array     */
```

- Array declaration with start and end address, specific in BDL

```
int       e[0..9];   /* (base 0, size 10) */
reg char  f[0..9];
mem (0:16)g[3..10];  /* (base 3, size 8)  */
mem(0:16) h[0..1][0..3];
```

[Caution points]

- Start and end addresses cannot be specified in descending order.

```
int i[3..0][4..0]; /* NG:  descending order */
int j[0..3][4..0]; /* NG:  descending order */
```

- As in C language, only integer constant expressions can be used when declaring array subscripts. Floating-point constants and variable expressions cannot be specified.

  – Example of integer constant expression

```
int a[10];
int a[10*2];
int a[0..10+5];
```

  – Example of other expressions

```
int a[0.23..0.3+10]; /* Floating-point constant */
int a[b];            /* Variable                */
```

[Restrictions on Cyber behavioral synthesis system]

- Multidimensional arrays are limited to 16 or fewer dimensions.
- Specification of start and end addresses for array elements is supported only for one-dimensional arrays (except for arrays synthesized as register files with decoders). The Specification is not supported for the Multi-dimensional array. However, when their base is 0, they are handled in the same way as element number specifications.

```
int        a[0..9];       /* O.K.          */
int        a[2..9];       /* Not supported */
reg char   a[4..9];       /* Not supported */
mem(0:16)  a[3..10];      /* O.K.          */
mem(0:16)  a[3..4][0..3]; /* Not supported */
```

6.2 Reference and Assignment of Arrays

A postfix expression enclosed by square brackets ("[" and "]") indicates reference or assignment of arrays or memory with subscripts.

- Bit specifiers can be used to specify partial bit fields for reference or assignment of arrays or memory. The bit specifiers are described after the subscript specification.

Example

```
b = a[c];       /* References all bits in a[c] */
b = a[c](2:3);  /* References partial bit field (2 start
                   bits and 3-bit width) in a[c] */
a[c] = d;       /* Assigns all bits in a[c] */
a[c](2:3) = e;  /* Assigns partial bit field (2 start bits
                   and 3-bit width) to a[c] */
```

6.3 Initialization of Arrays

As in C language, the array or memory size specification can be omitted when specifying initialization. The initialization expression for an array or memory is described in a list of initialization expressions that include braces for elements.

- Only the leftmost dimension can be omitted for a multidimensional array.

```
int a[] = {0,1,2,3};                          /* OK */
int b[][3] = {{0,1,2},{3,4,5}};               /* OK */
int c[2][] = {{0,1,2},{3,4,5}};               /* NG */
int d[][]  = {{0,1,2},{3,4,5}};               /* NG */
```

- When the array size is omitted, it is determined by the number of initial value expressions.

```
int e[][4] = {1, 2, 3, 4, 5, 6, 7, 8}; /* e[2][4] */
int f[] = {1, 2, 3, 4, 5, 6, 7, 8};    /* f[8]    */
```

- When the array size has been specified, the number of initial value expressions must not exceed the array size.
- When the array size has been specified and the number of initial value expressions is less than the array size, all remaining elements are initialized to zero.

```
int f[8] = {1, 2, 3, 4 };
            ↓
int f[8] = {1, 2, 3, 4, 0, 0, 0, 0};
```

7. POINTERS

7.1 Pointers in BDL

Pointers can be described in BDL in the same way as in C language. Examples of pointers that can be described in BDL are shown below.

- Passing of an array to a function
  Pointers can be used to pass arrays as arguments to functions.

```
in  ter(0:8) a, b;
out ter(0:8) x;
    reg(0:8) R1[128], R2[128];

process main()
{
    if (a) {
       func(R1); /* Becomes x = R1[b] */
    } else {
       func(R2); /* Becomes x = R2[b] */
    }
}
/* Cyber func=inline */
void func(
    reg(0:8) *r
)
{
    x = r[b];
}
```

- Passing of an array with offset to a function
  Similar to passing the base of the array as arguments, pointers can be used to pass arrays with offsets as arguments to functions.

```
in  ter(0:8) a, b;
out ter(0:8) x;
    reg(0:8) R[128];

process main()
{
    if (a) {
       func(&R[0]);   /* Becomes x = R[0 + b]   */
    } else {
       func(&R[100]); /* Becomes x = R[100 + b] */
    }
}

/* Cyber func=inline */
void func(
    reg(0:8) *r
)
{
    x = r[b];
}
```

- Passing an reference of a variable to a function
  Pointers can be used as references of variables as arguments in functions, so that the values of the variables can be changed in the functions.

```
in  int a_i, b_i;
out int o1, o2;

process main()
{
    reg int a, b;

    a = a_i; /* (1) */
    b = b_i; /* (2) */
    $
    swap(&a, &b);
    $
    o1 = a; /* Value of a becomes b_i value in (2) */
    o2 = b; /* Value of b becomes a_i value in (1) */
}

// Cyber func=inline
void swap(
    int reg *a,
    int reg *b
)
{
    int tmp;
    tmp = *a;
    *a = *b; /* Indirect assignment to pointer enables
                overwriting of values */
    *b = tmp;
}
```

- Dynamic allocation of array
  When array size is statically determined, array can be allocated by malloc

```
in  int ter a;
out int ter x;
#define MSIZE 10

process main()
{
    int i;
    int *p;

    p = malloc(MSIZE * sizeof(int)); /* partitioning of p[10]*/
    for (i = 0; i < MSIZE; i++) {
        p[i] = a;
        $
    }
    for (i = MSIZE-1; i >= 0; i--) {
        x = p[i];
        $
    }
}
```

7.2 Restrictions on Pointer Descriptions in BDL

In BDL there are several restrictions in describing pointers. The pointers that can be described in BDL differ from C language as follows.

- Assignments to pointer variables must match on both the sides in terms of pin specification (in, out, inout, shared, or outside), sign specification (signed or unsigned), physical type specification (ter, reg, var, mem), bit width, ascending or descending order and fixed point type.

```
          in signed ter(0:4) a;

          in signed ter(0:4) *p;
          /* Pointer to in pin, signed, ter type, 4-bit ascending
             variable */

          in        ter(0:4) *q;
          /* Pointer to in pin, unsigned, ter type, 4-bit ascending
             variable */

          signed    ter(0:4) *r;
          /* Pointer to internal variable, signed, ter type, 4-bit
             ascending variable */

          in signed reg(0:4) *s;
          /* Pointer to in pin, signed, reg type, 4-bit ascending
             variable */

     <OK> p = &a;
     <NG> q = &a; /* Sign differs */
     <NG> r = &a; /* Pin specification differs */
     <NG> s = &a; /* Physical type differs */
```

- Input/output through a pointer that indicates an input or output variable is performed when the variables which are indirectly referenced and assigned by pointers are read/written.

```
in signed ter(0:4) a;
in signed ter(0:4) *p;
    signed ter(0:4) x;

p = &a; /* Input is not read here */
$
x = *p; /* Input actually occurs here */
```

- Casts are allowed only for conversion of signs, bit order and fixed point type. Casts for changing the bit width or physical type cannot be described.

```
     signed   ter(0:4)  a;
     signed   ter(0:4) *p;
     unsigned ter(0:4) *q;
     signed   reg(0:4) *r;

     p = &a;
<OK> q = (unsigned *) p; /* Cast for converting sign */
<NG> r = (reg *) p;      /* Cast for converting physical type */
```

- Multi-level pointers (such as a pointer to another pointer) cannot be described.

```
signed var(0:4) **p; /* <NG> */
```

- NULL pointer and pointer to void cannot be described

- Subtraction of pointers cannot be described.

- Comparison operator of pointers cannot be described.

```
          signed ter(0:4) a[10];
          signed ter(0:4) *p, *q;

          p = &a[0];
          q = &a[10];

     <NG> if (p) {      /* Comparison between Pointer & 0 */
             p++;
          }
     <NG> if (p == q) { /* Equality/Inequality comparison of address*/
             p++;
          }
     <NG> if (p < q) {  /* Size comparison of address*/
             p++;
          }
```

- Pointers to functions and character strings cannot be described
- Functions that return pointers cannot be described
- Pointers cannot be described in allstates descriptions
- Pointers used in allstates description cannot be used except allstates

  In addition, the following restrictions exist for pointers that can be synthesized by the Cyber behavioral synthesis system.
- Some restrictions are there, in case function having pointers in argument is synthesized as functional unit.
- Only pointers that can be replaced by the object pointed by them during behavioral synthesis can be described.

  When a pointer is assigned during a branch or loop operation, an error occurs if the pointer's specification target is not uniquely determined before synthesis.

```
         bool flag;
         char a, b, c;
         char *p;

         if (flag) {
             p = &a;
         } else {
             p = &b;
         }
    <NG> c = *p; /* Not set except when &a or &b is executed */
```

In such cases, an intermediate variable is used to enable synthesis with replacements to ensure unique pointer definitions and references.

```
bool flag;
char a, b, c;
char tmp;
char *p;

if (flag) {
    p = &a;
    tmp = *p; /* This pointer (p) is determined when
                 synthesized with &a.*/
} else {
    p = &b;
    tmp = *p; /* This pointer (p) is determined when
                 synthesized with &b.*/
}
c = tmp;
```

Depending upon the condition, in case pointer points different elements of same array, synthesis is possible as the pointing destination is considered unique.

```
             bool flag;
             char ary[256], i, j;
             char *p;

             if (flag) {
                 p = &ary[i];
             } else {
                 p = &ary[j];
             }
     <OK>    c = *p; /* Synthesis is possible because both points to
                        element of array ary*/
```

- In case of goto conversion of function having pointer in argument, only the items that can be replaced actually at the time of behavioral synthesis can be specified at all the calling locations.

```
         bool flag;
         char a, b, c;
         if (flag) {
             c = func(&a);
         } else {
             c = func(&b);
         }
         /*....*/

         /* Cyber func = goto */
         char func(char *p) {
    <NG>     return *p; /* Either &a or &b will change depending upon
                           the calling location, this  cannot be known
                           until execution is done.*/
         }
```

Meanwhile, depending upon the condition if pointer points different elements of same array, synthesis is possible as the pointing destination is considered unique.

```
         bool flag;
         char c;
         char ary[256], i, j;
         if (flag) {
             c = func(&ary[i]);
         } else {
             c = func(&ary[j]);
         }
         /* .... */

         /* Cyber func = goto */
         char func(char *p) {
    <OK>     return *p; /* Synthesis is possible because both (of the
                           calling) points to element of array ary.*/
         }
```

- Since pointer variables behave like var type, the value of a pointer is kept across the clock boundary and the execution order is always descending.

```
ter(0:8) ary[256] a, b, x, y;
ter(0:8) *ptr;

ptr = &a;
x = *ptr;  /* This pointer (ptr) becomes &a */
ptr = &b;  /* Multiple definitions can be entered when they
              are within the same cycle*/
$
x = *ptr;  /* This ptr becomes &b */
ptr = &a;
y = *ptr;  /* This ptr becomes &a */
```

- Only when a pointer variable specifies an array, the pointer's add/subtract operations can be described.

```
        ter(0:8) ary[256], x, y;
        ter(0:8) *ptr;

        ptr = ary;
        x = *ptr;     /* This ptr becomes &a[0].*/
<OK>    ptr++;
        x = *ptr;     /* This ptr becomes &a[1] */
<OK>    ptr = ptr + 100;
        x = *ptr;     /* This ptr becomes &a[101] */
<OK>    ptr = &y;
        x = *ptr;     /* This ptr becomes &y */
<NG>    ptr++;        /* Error: did not indicate destination for
                         ptr */
```

- The behavior of pointer that points to multidimensional array is different from C language.

  In C Language, an element anywhere in the multidimensional array can be accessed using an offset with pointer to the base address. Here, when the offset for an element to be accessed is greater than the right most dimension the elements, the subsequent row is accessed, as shown in the example below.

```
          int ary[2][10], x,y;
          int *p;

          p = &ary[0][0];
 <OK>     x = p[5];  /* Points to ary[0][5] */
 <OK>     y = p[15]; /* Points to ary[1][5] */
```

On the other hand, in BDL, only elements in a row can be accessed using offset with pointer to the base address of the row. In case the offset is greater than the length of the row, it produces an error.

```
      int ary[2][10], x,y;
      int *p;

      p = &ary[0][0];
<OK>  x = p[5];         /* Points to ary[0][5] */
<NG>  y = p[15];        /* Points to area outside array
                           ary[0][15] */
```

In BDL, pointer to multi-dimensional array can be used to access memory location of a multi-dimensional array.

```
        int ary[2][10], x,y;
        int (*p)[10]; /* Pointer to 2 dimensional array with 10 rows */

        p = &ary[0];
<OK>    x = p[0][5]; /* Points to ary[0][5] */
<OK>    y = p[1][5]; /* Points to ary[1][5] */
```

8. STRUCTURES

8.1 Structure that can be described in BDL

Structure can be described in BDL similarly like C language.

- Multiple variables can be declared in one structure.

```
struct data {
     var(0:2) id;
     var(0:8) upper_data;
     var(0:8) lower_data;
};
struct data v0;
struct data v1;
```

- Assignment can be done.

```
struct data {
     var(0:2) id;
     var(0:8) upper_data;
     var(0:8) lower_data;
};

struct data v0;
struct data v1;

process struct_module1(){
     v1 = v0;
}
```

- The structure variable can be passed to function

```
struct data {
    var(0:2) id;
    var(0:8) upper_data;
    var(0:8) lower_data;
};
process struct_module2(){
    struct data v1,v2;
    v3 = func(v1,&v2);
}
struct data func(
    struct data i1,
    struct data *pi2;
) {
    struct data rv;
    ...
    return(rv);
}
```

8.1.1 BDL specific types and structure

Physical type, which is BDL specific type, and I/O type can be specified for member variable.

- Type that can be declared for structure member

```
<Types that can be declared>
   bool int char short long
   signed unsigned
   var ter reg mem
   in out inout
   shared outside
   reset clock

<Types that cannot be declared>
   static extern typedef register auto
   void
   const volatile
```

Physical type, which is BDL specific type, and I/O type can be specified for structure variable. However, there are restrictions for qualifiers that can be added to a structure variable depending upon the member included.

- Qualifiers that can be attached to structured variable

```
<Qualifiers that can be added>
      var ter reg mem
      in out inout
      shared outside
      static extern typedef register auto
      const volatile

<Qualifiers that cannot be added>
      bool int char short long
      signed unsigned
      reset clock
      void
```

- Restrictions of qualifiers that can be attached to structure variable
  - In the structure variable having member function (refer to 8.1.2), ter, reg, var, mem, in, out, inout, shared, outside cannot be added.
  - In case physical type var, ter, reg, mem are specified for structure variable, physical type is attached to members, excluding pointer variable, and physical type of member is ignored.
  - In case in, out, inout, shared, outside are specified for structure variable, attribute specified for member is attached.
    - * in, out, inout, shared, outside cannot be attached to structure variable that includes in, out, inout, shared, outside, reset, clock as member.
    - * in, out, inout, shared, outside cannot be attached to structure variable that includes pointer variable as member.
- For structure variables also, same restrictions are applicable as in case of other variable. For instance, inout reg variable cannot be declared; in, out, inout, shared, outside variable cannot be declared for local variable; and mem type can be specified only with array variable.

  In case of structure variable, it is possible to inherit the attribute(s) of structure variable and member variable and to change the attribute(s) of member variables by specifying the required attribute(s) to the structure variable. However it may be prohibited in cases where there is a restriction on either the attribute of structure variable or the member variable.

- In the following example, since only member variable in0 is inout reg type so even if ter type is specified in structure variable to make it inout ter type, there will be error due to restriction.

```
struct ST1 {
    inout reg in0; /* Member type is inout reg type, therefore,
                      error */
};
ter struct ST1 x;  /* Example, even on specifying ter type while
                      declaring structured variable */
```

- In the following example, structure variable y and z inherit the attributes of both structure variable and member variable and become array of mem type. There will be an error due to restriction since mem type is specified to member variable m1, which is not an array and mem type is specified to structure variable z, which is also not an array.

```
struct ST2 {
     mem(0:8) m1; /* Error because mem type cannot be specified
                     except in array.*/
};
struct ST2 y[10]; /* Even if structure variable is array.*/

struct ST3 {
     var(0:8) m2[10];/*Even if member variables are all array.*/
};
mem struct ST3 z;    /* Error because mem type cannot be specified
                        except in array.*/
```

8.1.2 Member function

In BDL, member function can be defined in structure just like C++ language.

- Member function defined in BDL will be converted into global function by naming rule called as "[Structure name] [Function name]" during input data analysis.
  In case member function is converted into user defined functional unit, the function name which is converted into global function will be specified in -Ztop option during synthesis. In case member function is converted into global function, since function name is generated by BP_I2001 message of data analysis log, it can be taken as reference.
- Specific example of member function is as follows

```
/*Definition example of member function*/
template <typename DATA_TYPE>
struct pipe_io {
    out ter rreq;
    out ter wreq;
    in ter ready;
    in DATA_TYPE data_out;
    out DATA_TYPE data_in;

    /*You can also specify attribute in member function*/
    /* Cyber func = inline */
    DATA_TYPE read() {
        /* You can also use template parameter of related structure in
           member function*/
        DATA_TYPE val_tmp;
        var is_read = 0;

        while (is_read == 0) {
            $
            /* You can use this pointer variable same as C++C++
               (You can omit also)*/
            this->rreq = 1;
            val_tmp = data_out;
            is_read = ready;
        }

        return val_tmp;
    }
    /* Cyber func = inline */
    var write(DATA_TYPE val) {
        var is_write = 0;

        while (is_write == 0) {
            $
            wreq = 1;
            data_in = val;
            is_write = ready;
        }
        return 1;
    }
};
template <typename DATA_TYPE>
struct pipe_body {
    in ter rreq;
    in ter wreq;
    out ter ready;
    out DATA_TYPE data_out;
    in DATA_TYPE data_in;
    void entry()
    {
        ter is_rreq;
        ter is_wreq;
        ter is_ready;
        DATA_TYPE tmp_data;
        /* Cyber scheduling_block */
        while (1) {
            is_rreq = rreq;
            is_wreq = wreq;
            tmp_data = data_in;
            data_out = tmp_data;
            if (is_rreq && is_wreq) {
                is_ready = 1;
            }
            ready = is_ready;
            $
        }
    }
};
```

```
/* Example of member function usage*/
#include "pipe.h"
struct pipe_io<ter(0:8)> chl_char1, chl_char2;
struct pipe_io<ter(0:16)> chl_wchar;
struct pipe_io<ter(0:32)> chl_out;

clock clk;
reset rst;

process concat_mod()
{
    unsigned char rdata1, rdata2;
    var(0:16) rdata3;
    var(0:32) wdata;
    /* Since the member function generates simultaneously
       with the structure instance hence, effort for defining
       function for each data type can be saved. */
    rdata1 = chl_char1.read();
    rdata2 = chl_char2.read();
    rdata3 = chl_wchar.read();

    wdata = rdata1 :: rdata3 :: rdata2;
    chl_out.write(wdata);
}
```

- When member function which carries out the prototype declaration is defined outside the structure, function can be defined after specifying the relevant structure using the scope resolution operator (::) (Refer to 13.4).

```
struct foo {
   int i1, i2;
   /* Prototype declaration of member function is possible */
   int total(void);
};

/* Function is defined as follows in case of prototype
   declaration of member function.*/
int foo::total(void) {
   int sum;

   sum = i1 + i2;
   return sum;
}
```

- Template can also be defined in member function when structure is not template. However, due to template restriction, prototype declaration cannot be described when template member function is defined.

- In case structure is a template, template function cannot be defined due to the restriction of BDL template but only those functions which are not template in member function can be defined.

```
template <typename T, int N>
struct foo {
    T ary[N];
    /* Parameter variable (T or N in this example) of
       template structure can be used in member function */
    void setdata(int idx, T v) {
        if (idx >= N) {
            idx = 0;
        }
        ary[idx] = v;
    }
    /* In case of member function which is not template
       prototype declaration is possible */
    T total();
};
/* In case prototype of member function of template structure
   is declared, function definition is shown below */
template <typename T, int N>
T foo<T,N>::total(void) {
    T sum = 0;
    for (int count=0; count < N; count++) {
        sum += ary[count];
    }
    return sum;
}

struct bar {
    int i;
    void seti(int v) {
        i = v;
    }
    /* If structure is not template, member function of
       template can be defined */
    template <typename T>
    T geti() {
        return i;
    }
};
```

- Following restrictions exist in member function of BDL
  - Since the variables used in the member functions have the dependency on the order of their definitions and their use location in BDL, member variables should be defined before their use.

    When the variable name is used before the member variable is defined, it results in an error undefined variable is used.

    However, in case the definition of global variable of same name exists before global variable is used instead of member variable, make sure it won't result in an error.
  - In the struct type defining the member function, the other types (ter/reg/var and in/out/inout/shared/outside etc) cannot be specified during the variable declaration.

```
int g;

struct foo {
    int getg() {
        /* Since definition of member variable g does not exist
           before, caution while using global variable g */
        return g;
    }
    int getv() {
        /* An error occurs when definition of variable v does
           not exist before */
        return v;
    }
    int v, g;
};

/* In the variable declaration of structured type defined by
   member function, ter/reg/var/in/out/inout/shared/outside
   cannot be specified */
ter struct foo rh;
in struct foo ih;
shared struct foo sh;
```

– Scope resolution operator can be used to describe the definition of member function only when prototype is declared.
  Otherwise, this operator (::) can be used as concatenation operator (Refer to 13.3) in BDL.
– Static member function cannot be defined
– It does not support member function which performs operations such as construct, destruct in C++.

8.2 Notes on usage of structure

8.2.1 Assignment of structured variable

In BDL assignment can be done by structure variables of same structure similarly like C language. However, assignment cannot be done by structured variables of different structures.
Therefore, in case assignment needs to be carried out using structure variable, physical type and I/O type are not described in member and are attached at the time of declaring structured variable.

```
struct data {
    var(0:2) id;
    var(0:8) upper_data;
    var(0:8) lower_data;
};
in  struct data in0;
    var struct data org_data;
out reg struct data out0;

process struct_module3(){
    input(in0);
    org_data = in0;
    ....
    out0 = org_data;
}
```

8.2.2 Structure that includes in, out in member

At the time of describing interface, if assignment or referencing is carried out by structure variable, when member has a mix of in, out such that, in variable will be formed in assignment destination and out variable will be formed at referencing source, and error will be generated. There is a need to specify the connection explicitly by using function or a macro.

```
struct interface1 {
    in ter(0:1) in_enable;
    in ter(0:8) in_data;
   out reg(0:1) out_enable
   out reg(0:8) out_data;
};
struct interface1 pre;
struct interface1 next;

process struct_module4(){
    pre = next; /* Error */
}
```

Example of using macro

```
struct interface1 {
     in ter(0:1) in_enable;
     in ter(0:8) in_data;
    out reg(0:1) out_enable
    out reg(0:8) out_data;
};

struct interface1 pre;
struct interface1 next;

#define interface1_connect(x,y) \
       (y).out_enable = (x).in_enable; \
       (y).out_data = (x).in_data; \
       (x).out_enable = (y).in_enable; \
       (x).out_data = (y).in_data;
process struct_module4(){
      interface1_connect(pre,next);
}
```

Example of using function

```
struct interface1 {
     in ter(0:1) in_enable;
     in ter(0:8) in_data;
    out reg(0:1) out_enable
    out reg(0:8) out_data;
};

struct interface1 pre;
struct interface1 next;

void interface1_connect(
      struct interface1 *x,
      struct interface1 *y
) {
   y->out_enable = x->in_enable;
   y->out_data = x->in_data;
   x->out_enable = y->in_enable;
   x->out_data = y->in_data;
}

process struct_module4(){
     data_connect(&pre,&next);
}
```

There are also methods to define the interface by declaring the structure that does not attach in, out in assignment unit, and defining the structure that considers the previously declared structure as member variable.

```
struct data {
    var(0:1) enable;
    var(0:8) data;
};
struct interface1 {
    in ter struct data in0;
   out reg struct data out0;
};

struct interface1 pre;
struct interface1 next;

process struct_module4(){
    pre.out0 = next.in0;
    next.out0 = pre.in0;
}
```

8.2.3 Structure that includes array in member

In case of assigning the structure variable that includes member array and transferring the function parameter or return value, value is copied similarly like C language, therefore, all the elements of arrays in that member will be copied. Therefore, if the numbers of elements are too many, it will become expensive. Improvement can be done by preventing the value from getting copied by converting parameters as pointers etc.

```
struct data {
       var(0:8) r[256];
};
process struct_module5(){
     struct data x,y;

     y = x;
     /* same as for(i=0;i<256;i++){
      *              y.r[i] = x.r[i];
      *          }
      */
     y = func(x);
     /* The assignment of member array is executed
      * both at the time of function call and
      * returning value respectively */}

struct data func(
       struct data v
) {
    .....
}
```

8.2.4 Initial value of structure

There is a restriction that initial value setup of structure variable should be handled as a normal assignment therefore structure having array implemented as combinatorial circuit (array = LOGIC attribute) and mem type as member variables are not supported. Generally, initialization for the variable of structure including mem type causes an error because it requires multiple cycles, that is "Number of Memory Word /Number of Memory Port" while the reset state must be single cycle.

There will be error in case of array is implemented by Combinatorial Circuit as combinatorial circuit as assignments in array is not permitted. In case initial value has to be described for these members, then it is possible to avoid the restriction by describing them separately out of the structure.

```
/*
 * In the following structure variables
 * it is not possible to describe initial value.
 */

/* In case array implemented by Combinatorial Circuit
   included */
struct logic_data {
    var(0:2) id;
    var(0:8) data[8] /* Cyber array = LOGIC */;
} ld_str;

/* In case members of mem type are included */
struct mem_data {
    var(0:2) id;
    mem(0:8) data[8];
} md_str;

/* In case mem type is added to structure */
struct var_data {
    var(0:8) data_v[4];
    var(0:8) data_h[4];
};
mem struct var_data vd_str[2];
```

9. FIXED POINT TYPE

9.1 Type Declaration

Fixed point type is declared by adding the reserved keyword *fixed* in type.

```
/* signed variable of integral part 5 bit, decimal part 3 bit */
signed var (0:8) fixed (5)                    a;

/* unsigned variable of integral part 5 bit, decimal part 3 bit */
/* quantization mode is FX_RND, overflow mode is FX_WRAP */
unsigned var (0:8) fixed (3, FX_RND, FX_WRAP)  b:
```

Following 4 actual arguments can be specified in *fixed*.

1st argument	Bit width of integer part (integer word length)	Cannot be omitted
2nd argument	Quantization mode (quantization mode)	Can be omitted
3rd argument	Overflow mode (overflow mode)	Can be omitted
4th argument	Overflow bit count (saturated bits parameter)	Can be omitted

Arguments from 2nd onwards can be omitted. In case omitted, the quantization mode is taken as "truncation(FX_TRN)", overflow mode is taken as "wrap(FX_WRAP)", and overflow mode bit count is taken as 0 bit. Bit width of the integer part of 1st argument cannot be omitted. Also, if 3rd argument is specified then 2nd argument cannot be omitted. Similarly, if 4th argument is specified then 2nd and 3rd argument cannot be omitted.

Following 7 types can be specified in quantization mode (Refer to **Section 9.7** for details)

FX_RND	Rounding to plus infinity
FX_RND_ZERO	Rounding to zero
FX_RND_MIN_INF	Rounding to minus infinity)
FX_RND_INF	Rounding to infinity
FX_RND_CONV	Convergent rounding
FX_TRN	Truncation (default)
FX_TRN_ZERO	Truncation to zero

Following 5 types can be specified in overflow mode (Refer to **Section 9.8** for details)

FX_SAT	Saturation
FX_SAT_ZERO	Saturation to zero
FX_SAT_SYM	Symmetrical saturation
FX_WRAP	Wrap-around (default)
FX_WRAP_SM	Sign magnitude wrap-around

Only the 2 modes "wrap(FX_WRAP)" and "sign magnitude warp-around(FX_WRAP_SM)" are valid for overflow bit count, others will be ignored.

9.2 Fixed point type operation

How much will be the total of Bit width of decimal part, bit width of integer part, bit width and sign of result of the fixed point type operation has been explained in this section.

However, in case of other than fixed point type, refer to **Section B**.
When carrying out the operation using the variable of fixed point type, if the bit width of decimal part is different, then a part of operation is removed and a correction with which the bit width of decimal part becomes the same takes place.

In correction of bit width of decimal part, 0 is added to the operand with lower width of decimal part so that it can match with the higher bit width of decimal part.

1. Arithmetic operator

1-1 Arithmetic addition (+), arithmetic subtraction (-), simple minus (-)
The bit width of integer part and operand sign is corrected in a way similar to other than fixed point type and bit width of decimal point is also corrected according to the higher one.
Bit width of integer part and operand sign is similar to other than fixed point type. Bit width of decimal part will be similar to the bit width of decimal part of operand.

1-2 Arithmetic multiplication (*)
The operand sign is corrected in a way similar to other than fixed point type, sign of return value and bit width of integer part is same as the sign of operand in a way similar to other than fixed point type and becomes the sum of bit width of integer part of operand.
The bit width of decimal part of return value is the sum of bit width of decimal part of operand.

1-3 Arithmetic division (/)
The sign of return value is corrected in a way similar to other than fixed point type. The bit width of the integer part of return value is the sum of bit width of the integer part of operand on left side and bit width of the decimal part of operand on the right side. The bit width of decimal part of return value will be similar to bit width of decimal part operand on left side. In case the result of division cannot be expressed in bit width of decimal part of return value then the "Truncation to zero(FX_TRN_ZERO)" will be carried out.
In the below example, since the sign of variable a and variable b is different therefore variable a is corrected to signed and treated as signed variable of 10 bit. The bit width of integer part of return value of division will be 8 bit. That is the sum of bit width 5 for integer part of variable a and bit width 3 for decimal part of variable b. The bit width of decimal part of return value of division will be bit width 5 for decimal part of variable a. The overall bit width of return value of division will be 13 bit.

```
       var(0:9)  fixed(4) a; /* Integer part 4 bit Decimal part 5 bit*/
signed var(0:5)  fixed(2) b; /* Integer part 2 bit Decimal part 3 bit*/
signed var(0:13) fixed(8) c;
       c = a / b;
```

1-4 Arithmetic surplus calculation (%, %=)
Since surplus calculation of fixed point type is not supported, it will be an error.

1-5 Increment calculation (++), decrement calculation (--)
Return value is similar to the type of argument. As regards the calculation, increment calculation will be same as the value which comes as a result of +1. It will be the value which comes after adding 1 in the least significant bit. Further, return value will return a value which comes after quantization mode or overflow mode is applied.

1-6 Compound assignment calculation (+=, -=, *=, /=)

The return value is similar to the type of assignment point. Further, return value will return a value which comes after quantization mode or overflow mode is applied.

2. Comparison operation (>, <, <=, >=, ==, !=)
   Similar to addition comparison is carried out after correcting bit width of integer part and sign, bit width of decimal part. Return value will be 1 bit of unsigned type.

3. Logical calculation (&,|,^)
   In logical if bit width of integer part or decimal part are different, then they are matched with the one with higher value.
   Return value is unsigned type and bit width of integer part and decimal part will be similar to operand.

4. Shift operation
   Since left shift operation (<<), right shift operation (>>)

5. Logical relation calculation (&&,||, !)
   Logical calculation is performed including decimal part also. Return value will be 1 bit of unsigned type.

6. Complementary calculation (~)
   Return value is of unsigned type, bit width of the integer part and decimal part will be similar to operand.

7. Conditional operation (? :)
   Correction similar to addition will be performed when bit width of decimal part, bit width of integer part and sign of 2nd, 3rd operand are different.
   Sign of return value, bit width of integer part, bit width of decimal part will be similar to value which comes after the correction of operand.

8. Cast operation
   In case the specified type is a non-pointer, type variable is similar to the thing which is assigned in intermediate variable of the specified type.

9. Redirection operation (|>, &>, ^>, ~|>, ~&>, ~^>)
   Redirection function is carried out including the decimal part also. Return value will be 1 bit of unsigned type.

10. Concatenation operation (::)
    Return value is of unsigned type, bit width of integer part is the sum total of bit width of operand. The bit width of decimal part of return value is 0 bit.

11. Bit specification calculation
    Return value is of unsigned type, Bit width of the integer part will be the bit specified in bit specification. The bit width of decimal part of return value is 0 bit.

9.3 Handling assignment to variable other than fixed point type

When a variable of fixed point type has been assigned to variable other than the fixed point type then quantization mode is treated as “Truncation to zero (FX_TRN_ZERO)”, over flow as "Wrap around (FX_WRAP)" and bit count for overflow mode is treated as 0 bit.

As shown below, when a negative value is assigned to a signed variable then the variable of fixed point type and variable of other than fixed point type will be different.

```
signed var(0:8)                        a; /* Not fixed point type */
signed var(0:8) fixed(8)               b; /* Fixed point type     */
signed var(0:8) fixed(8,FX_TRN)        c; /* Fixed point type     */
signed var(0:8) fixed(8,FX_TRN_ZERO)  d; /* Fixed point type     */
a = -1.25; /* a will become -1 */
b = -1.25; /* b will become -2* /
c = -1.25; /* c will become -2 */
d = -1.25; /* d will become -1 */
```

9.4 Constant value of decimal number

9.4.1 Decimal fraction

When constant value of decimal fraction is defined, it is treated as fixed point type and its bit width of fraction part is 32bit.

When the bit width of the fraction part of constant fixed point value is 33bit or more, it is truncated to zero (FX_TRN_ZERO).

The bit width of integer part will be the minimum bit width which can be expressed.

When decimal fraction is 1.1, the bit width of its fraction part is more than 33 bit.

So the above "truncated to zero" rule has been applied and the truncated value will be 1.09999999986030161380767822265625 (0x1.19999999 in hexadecimal).

The bit width of the fraction part of fixed point value can be adjusted by using cast operator or assigning to the certain bit width of variable.

```
var(0:8) fixed(3) foo;
var(0:4) in1

foo = 1.1; /* should be assigned to foo. Decimal number part is
              truncated to 5 bit.*/

/* Decimal part of 1.1 is reduced to 5 bit by using cast */
foo = ( var(0:6) fixed(1) )( 1.1 ) + in1;
```

Perhaps, the 2 below mentioned attributes have been provided for case when the accuracy of more than 33 bits of bit width of decimal number part is necessary or case when bit width of decimal number part of constant value of all the decimal numbers are changed. (Refer to Section 19 for details of attributes).

- Attribute which specifies the bit width of decimal number part of constant value of all the described decimal numbers.
  /* Cyber const_frac_bitw = # */ (Specifying in the definition of process function)
- Attribute which specifies the bit width of decimal number part of constant value of decimal numbers inside the function.
  /* Cyber func_const_frac_bitw = # */ (Specifying in the definition of process function)

In case both have been specified then func_const_frac_bitw attribute will be given priority

9.4.2 Binary fraction, Octal fraction and Hexadecimal fraction

As binary fraction, octal fraction and hexadecimal fraction is not supported, it will give an error. However, binary fraction and hexadecimal fraction has been used here for the sake of convenience.
Instead, the operator bvtofx() which is explained in **Section 9.5** has been used in description.

```
0b100.001 →   (var(0:6) fixed(3))(bvtofx(0b100001))
0xff.fff →    (var(0:20) fixed(8))(bvtofx(0xfffff))
```

9.5 Operator for not carrying out the implicit fixed point correction

When the value of fixed point type is assigned, when value is assigned to fixed point type, when operation is done with fixed point type, the position of fixed point is corrected and operation is carried out. This is known as implicit fixed point correction.

In the below example: the lower level 5 bit is assigned to 5 bit of integer part of x, and the value which comes as a result of adding constant 1 in 5 bit of integer part of x is assigned to z.

```
var(0:8) fixed(5)   x;
var(0:8)            y;
var(0:8)            z;

x = y;
z = x + 1;
```

Operator fxtobv () and operator bvtofx () have been provided such that assignment and operation can be made possible without carrying out this implicit fixed point correction.
Operator fxtobv() has been provided in order to take fixed point type value as a value which is not fixed point type (fx expresses fixed point type, b expresses bit vector). All bits are taken as integer part if a fixed point type value is passed on to this operator fxtobv().

Operator bvtofx() has been provided in order to assign a value which is not fixed type to fixed point type of assignment point as it is. When a value which is not fixed point type is passed on to this operator bvtofx and assignment is done in fixed point type then implicit fixed point correction does not take place and assignment is done as it is buy arranging the lowest bit together.

In the below example, the value of variable y is assigned to variable x of fixed point type as it is, the value of fixed point type variable x is treated as same as integer type of 8 bit, addition is done to that and it is assigned to variable z. As a result a value which comes by adding +1 in y as assigned to z.

```
var(0:8) fixed(5)   x;
var(0:8)            y;
var(0:8)            z;

x = bvtofx(y);
z = fxtobv(x) + 1;
```

9.6 Restriction and Precautions

- When quantization or overflow is executed in concatenation statements, inside *allstates*, *nmux* statement and inside *dmux* statement it is unsupported and error will take place.
  Specifically, in case the sign of assignment point type and assignment source type are same, in case bit width of decimal number part of assignment point and bit width of integer part are less than the assignment source type then quantization and overflow will be generated which is unsupported and error will take place.
- As regards the type specified in 1st argument and 2nd argument of *assign* statement, if same type is not specified including fixed point type also then error will take place.
- When fixed point type is used as the operand of the concatenation operator in assignment, it is not treated as fixed point type but treated as integer type.
  In the below example, integer part (6bit) of variable `a` is assigned in 8 bit variable where `v1` and `v2` are concatenated. So upper 2 bit of variable `a` (`a(8..7)`) will be assigned to `v1` and 4 bit of variable `a` (`a(6..3)`) is assigned to `v2`.

```
var(8..0) fixed(6) a;
var(3..0) fixed(2) v1,v2;

v1::v2 = a;
```

- A value bigger than bit width of all the types cannot be specified as the bit width of integer part.
  A negative number or 0 cannot be specified as the bit width of integer part of signed variable.
  Though a negative number cannot be specified as the bit width of integer part of unsigned variable but 0 can be specified.

```
  signed var(0:8) fixed(4) a; /* fixed(1) ～ fixed(8) OK */
unsigned var(0:8) fixed(4) b; /* fixed(0) ～ fixed(8) OK */
```

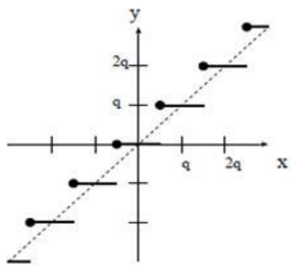
- Fixed point type should also match in type check at the time of substitution to pointer variable. Though it is possible to convert the fixed point type by using cast but the entire bit width and physical type cannot be converted.

```
var(0:8) fixed(4) a;
var(0:8) fixed(5) *p;
var(0:9) fixed(4) *q;

p = &a;                             /* NG */
p = (var(0:8) fixed(5) *)(&a);      /* OK */
q = (var(0:9) fixed(4) *)(&a);      /* NG */
```

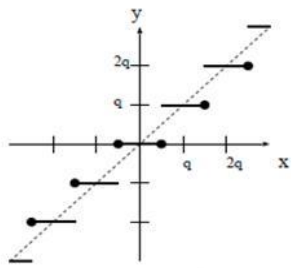
9.7 Description of quantization mode

The relation of value before the substitution (x axis) and value after the substitution (y axis) in each mode is shown in the below graph.
q in the graph expresses the maximum quantization unit of assignment point type. To be specific, the maximum quantization unit will be 0.25 in case the bit width of decimal number part of substitution point type is 2 bit.

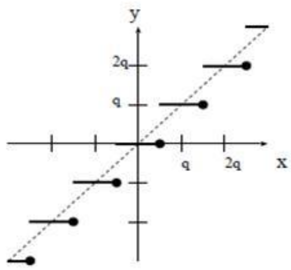
FX_RND : Rounding to plus infinity



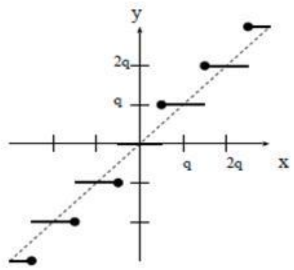
FX_RND_ZERO : Rounding to zero



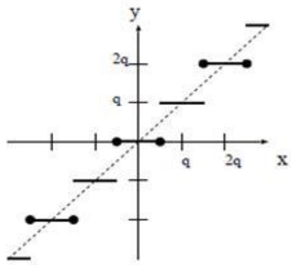
FX_RND_MIN_INF : Rounding to minus infinity



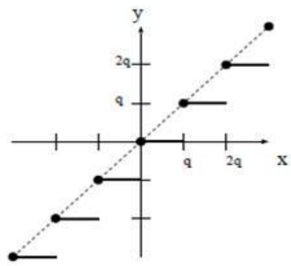
FX_RND_INF : Rounding to infinity



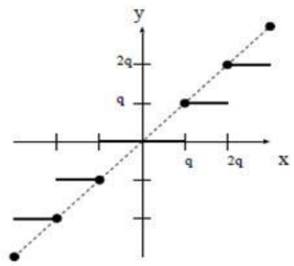
FX_RND_CONV : Convergent rounding



FX_TRN : Truncation (Default)



FX_TRN_ZERO : Truncation to zero



- When the value is positive then Truncation (FX_TRN) and truncation to zero (FX_TRN_ZERO) are same but they will differ when the value is negative.

```
signed var(0:4) fixed(2)               x;
signed var(0:3) fixed(2,FX_TRN)        y;
signed var(0:3) fixed(2,FX_TRN_ZERO)   z;

y = x;
z = x;
```

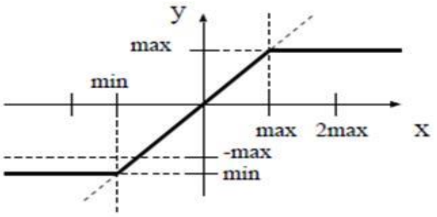
– When x is 1.75 (0b01.11), y and z will become 1.5 (0b01.1).
– When x is -1.75 (0b10.01), y will become -2.0 (0b10.0) and z will become -1.5 (0b10.1).

9.8 Description of overflow mode

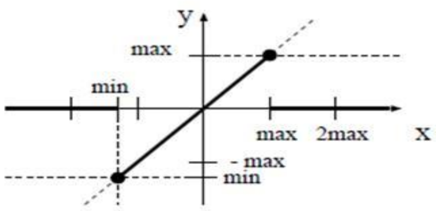
The relation of value before the substitution (x axis) and value after the substitution (y axis) in each mode is shown in the below graph. The overflow mode will differ in case type of assignment point is signed and in case it is unsigned.

- Overflow mode in case the type of assignment point is signed.
  The min, max in the graph shows the minimum value and the maximum value of assignment point type (y axis). q shows the minimum quantization unit of type of assignment source.
  Specifically, in case the type of assignment point is signed var(0:4) fixed(2) the minimum value will be -2 (0b10.00) and the maximum value will be 1.75 (0b01.11). In other words, the absolute value of minimum value and absolute value of maximum value will differ by only 1 minimum quantization unit.

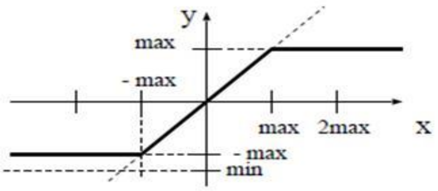
FX_SAT : Saturation



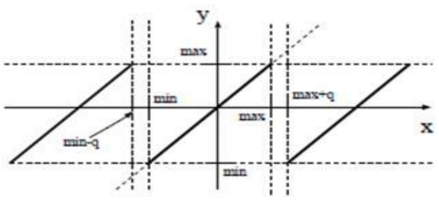
FX_SAT_ZERO : Saturation to zero



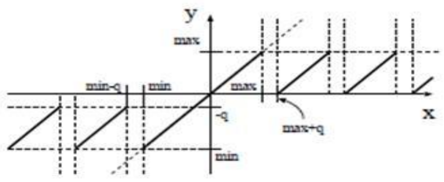
FX_SAT_SYM : Symmetrical saturation



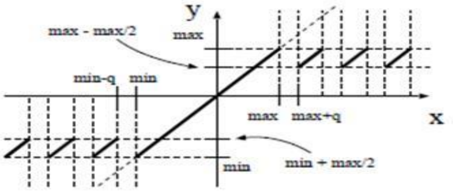
FX_WRAP : Wrap-around Bit count 0



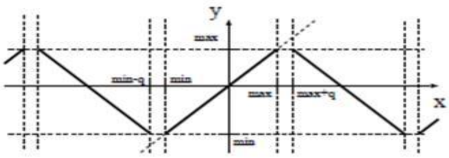
FX_WRAP : Wrap-around Bit count 1



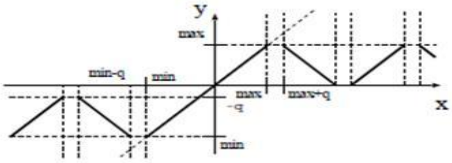
FX_WRAP : Wrap-around Bit count 2



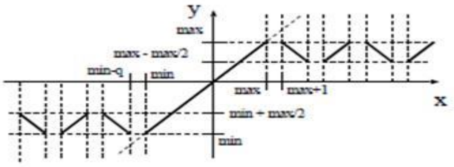
FX_WRAP_SM : Sign magnitude warp-around Bit count 0



FX_WRAP_SM : Sign magnitude warp-around Bit count 1



FX_WRAP_SM : Sign magnitude warp-around Bit count 2

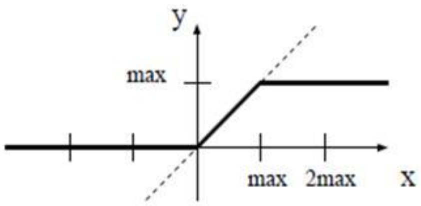


- Overflow mode in case the type of assignment point is unsigned.

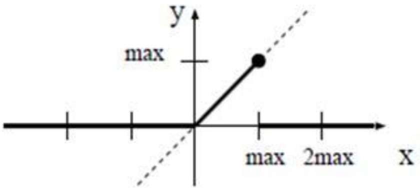
  The max in the graph shows the maximum value of assignment point type (y axis). q shows the minimum quantization unit of the type of assignment source.

  Specifically, in case the type of assignment point is unsigned var(0:4) fixed(2) the maximum value will be 3.75 (0b11.11). In case of unsigned type Saturation (FX_SAT) and Symmetrical saturation (FX_SAT_SYM) will become same. Since unsigned type is not defined in Sign magnitude wrap-around (FX_WRAP_SM) hence error will take place.

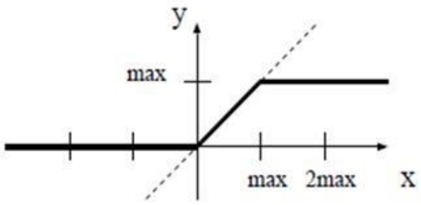
FX_SAT : Saturation

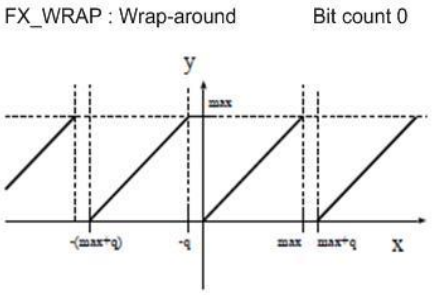


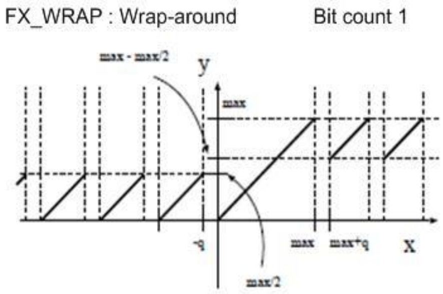
FX_SAT_ZERO : Saturation to zero

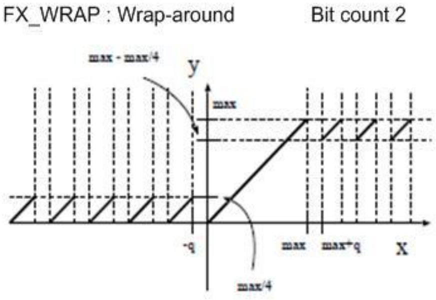


FX_SAT_SYM : Symmetrical saturation




FX_WRAP : Wrap-around
Bit count 0
y
max
-(max+q)
-q
max
max+q
x


FX_WRAP : Wrap-around
Bit count 1
max - max/2
y
max
-q
max
max+q
x
max/2


FX_WRAP : Wrap-around
Bit count 2
max - max/4
y
max
-q
max
max+q
x
max/4

10. FLOATING POINT TYPE

C/C++ arithmetic type specifiers `float` and `double` are supported.

10.1 Data type declaration

`float` or `double` type can be declared as floating point data type.
`float` corresponds to single precision floating point format (binary32) in IEEE 754 which occupies 32bits, with a 24-bit fraction and 8-bit exponent.
`double` corresponds to double precision floating-point format (binary64) in IEEE 754 which occupies 64bits, with a 53-bit fraction and 11-bit exponent.

10.2 Utility Functions

Utility functions below are available.

fltobv()	Conversion of single precision floating point(float) to 32-bit int
bvtofl()	Conversion of 32-bit int to single precision floating point(float)
dbtobv()	Conversion of double precision floating point(double) to 64-bit int
bvtodb()	Conversion of 64-bit int to double precision floating point(double)

In the below example, hexadecimal constant 0x3F000000 which is 0.5f in the decimal representation is assigned to float type variable f1.
Since the result of 0.5f + 0.5f equals to 1.0f, assigned value in float type variable f2 is 1.0f, with the result that assigned value in x is 0x3F800000.

```
float f1, f2;
unsigned int x;
f1 = bvtofl(0x3F000000);// f1 is 0.5f
f2 = f1 + 0.5f;         // f2 is 1.0f
x = fltobv(f2);         // x is 0x3F800000
```

10.3 Math.h Function

The synthesis in much of the standard C/C++ (math.h) math functions are supported.
Users must `#include<math.h>` in the use of these functions.
Below are the list of functions supported in each version of CyberWorkBench.

- FPGA version for Intel
  fabsf,fabs,fmaxf,fmax,fminf,fmin,hypotf,hypot,
  sinf,sin,cosf,cos,tanf,tan,asinf,asin,acosf,acos,atanf,atan,atan2f,atan2,
  logf,log,log2f,log2,log10f,log10,expf,exp,exp2f,exp2,exp10f,exp10,log1pf,log1p,powf,
  pow,sqrtf,sqrt,cbrtf,cbrt,roundf,round,floorf,floor,ceilf,ceil,truncf,trunc,copysignf,copysign

- FPGA version for Xilinx
  fabsf, fabs, expf, exp, logf, log,sqrtf,sqrt, roundf,round,floorf,floor,ceilf,ceil,truncf,trunc,
  copysignf,copysign,

10.4 Limitations

- Macro NAN (Not a Number) and INFINITY are not supported.
- Negative zero (-0.0 or -0.0f) is treated as positive zero (+0.0 or +0.0f).
- Fixed point type and floating point type cannot be used simultaneously in a module.
- `float` and `double` types are only supported which correspond to binary32 and binary64 in IEEE 754 floating point format, respectively. Arbitrary precision floating point types are not supported.

11. DEFMOD HIERARCHY DESCRIPTION

- defmod syntax is used to describe the hierarchical description explicitly. defmod syntax is BDL specific syntax, and is used to declare the interface of sub-hierarchy in the same syntax format as struct of C language.

11.1 defmod declaration

defmod syntax is declared similar to the syntax of struct in C language. "Module Name" of sub-hierarchy is described as structure name in struct. Module name is different from structure tab and cannot be abbreviated.
"Ports as member variable" that forms the interface with sub-hierarchy is declared as member variable in struct.
"defmod variable" that forms the instance name of sub-hierarchy is declared for structured variable name in struct.
The following is the example, where module name module1, member variable in1, in2, out1, defmod variable inst1, inst2, inst3, defmod array inst [2], and defmod type pointer variable ptr are declared.

```
defmod module1 {
     in    ter(0:8) in1;
     in    ter(0:8) in2;
    out    ter(0:8) out1;
} inst1,inst2;

defmod module1 inst3;
defmod module1 inst[2], *ptr;
```

in ter, out ter, shared reg, shared ter, shared var (any one of them including an array), shared mem array, reset, clock and other global variables not related to input output can be described in member variable of defmod.
In sub-hierarchical description, even if member variables of the module are described as in reg, in var, out var, out reg, the member variable of defmod declared in top-hierarchy must be declared as in ter, out ter.
Example of sub-hierarchy module 1

```
in  ter(0:8) in1;
in  var(0:8) in2;
out reg(0:8) out1;
process module1(){
    out1 = in1 + input(in2);
}
```

Array can also be described in member of defmod.

```
defmod module2 {
   in ter(0:8) in1[2][2];
  out ter(0:8) out1[2][2];
      int      x[3][4];
};
```

Structured variable can be described in member variable of defmod. However, it is restricted to the case when member-wise expansion is done and respective members become in ter, out ter, shared reg, shared ter, shared var, shared mem, reset, clock or if they are global variables unrelated to input-output.

```
struct ST1 {
   in ter(0:8) a;
  out reg(0:8) b;
      var(0:8) c;
      int      d;
};

defmod module3 {
   ter struct ST1 x;
};
```

11.2 Connection description through defmod conversion

"." Operator (Dot operator) is used for referencing the member of defmod variable similarly like structured variable.

Connection with sub-hierarchy is carried out by using assignment statement and continuous assignment statement in behavioral description. At the time of connection (port binding), the data flow determined by I/O characteristics of signals shall be from right to left.

```
in  ter(7..0) in1;
out reg(7..0) out1;

defmod module5 {
    in ter(7..0)  in1;
   out ter(7..0) out1;
};

defmod module5 mod1,mod2;

process top_module(){
   mod1.in1 ::= in1;
   mod2.in1 ::= mod1.out1;
   out1 ::= mod2.out1;
}
```

shared reg, shared ter, shared var (Any of them including an array), and shared mem array defines the connection by using assign() statement. assign() statement is described in allstates statement (Refer to section 14.5 for details of allstates statement).

In 1st argument of assign() statement, connecting entity is passed and in 2nd argument member of connecting defmod variable is passed. Sub hierarchy module can specify the resource of register and memory etc. which are used by using the assign() statement.

In the following example, register s_r is shared in the inst1 of module1 and inst2 of module2. On the other hand, memory instance is not shared between M1 of inst1 and M2 of inst2, which are declared as shared in module1 and module2 respectively.

```
in  ter(7..0) in1;
out reg(7..0) out1;

reg(0:8) s_r;
mem(0:8) M1[256];
mem(0:8) M2[256];

defmod module1 {
     in ter(0:8) in1;
    out ter(0:8) out1;
 shared reg(0:8) r1;
 shared mem(0:8) M1[256];
} inst1;

defmod module2 {
     in ter(0:8) in2;
    out ter(0:8) out2;
 shared reg(0:8) r2;
 shared mem(0:8) M2[256];
} inst2;

allstates {
    assign(s_r,inst1.r1);
    assign(s_r,inst2.r2);
    assign(M1,inst1.M1);
    assign(M2,inst2.M2);
}

process top_module(){
    inst1.in1 ::= in1;
    inst2.in2 ::= inst1.out1;
    out1 ::= inst2.out2;
}
```

Subscript can be specified in the array of argument specified in assign() statement.
In the below description, register r1, r2 used in module1 inst1 and register r3, r4 used in module2 inst2 are shared with register array s_r. Moreover, memory M1 used in module1 inst1 and memory M2 used in module2 inst2 is divided into 2 and the divided memory is allocated to M1 and M2.

```
 in ter(7..0) in1;
out reg(7..0) out1;

reg(0:8) s_r[2];
mem(0:8) M1[256],M2[256];

defmod module1 {
     in ter(0:8) in1;
    out ter(0:8) out1;
 shared reg(0:8) r1,r2;
 shared mem(0:8) M1[2][256];
} inst1;

defmod module2 {
     in ter(0:8) in2;
    out ter(0:8) out2;
 shared reg(0:8) r3,r4;
 shared mem(0:8) M2[2][256];
} inst2;

allstates {
    assign(s_r[0],inst1.r1);
    assign(s_r[0],inst2.r3);
    assign(s_r[1],inst1.r2);
    assign(s_r[1],inst2.r4);
    assign(M1,inst1.M1[0]);
    assign(M1,inst2.M2[0]);
    assign(M2,inst1.M1[1]);
    assign(M2,inst2.M2[1]);
}

process top_module(){
    inst1.in1 ::= in1;
    inst2.in2 ::= inst1.out1;
    out1 ::= inst2.out2;
}
```

Now, in case members of defmod are global variables unrelated to input-output then a description for their connection is not permitted. In the following description, since input output unrelated global variable x is referred in instA inside moduleA so it becomes an error termination.

```
in  ter(7..0) in1;
out reg(7..0) out1;
out reg(7..0) out2;

defmod moduleA {
    in ter(7..0) in1;
   out ter(7..0) out1;
       ter(7..0) x;
};

defmod moduleA instA;

process top_module(){
    instA.in1 ::= in1;
    out1 ::= instA.out1;
    out2 ::= instA.x;
}
```

11.3 Points to be noted at the time of behavioral synthesis

In Cyber behavioral synthesis system the hierarchy description using defmod statement is supported only in case of manual scheduling.
There is a need of explicit connection for reset and clock in hierarchy description.
In case shared mem is described for defmod member variable, there is a need to specify the constraints related to memory at the time of synthesis. Further, refer to "User manual Hierarchy design edition connection method multiple modules" for detailed synthesis specification method and restrictions at the time of using assign () statement.

11.4 Points to be noted at the time of defmod declaration

- typedef name can be declared by using typedef for defmod declaration.

```
typedef defmod module2 {
    in ter(7..0) in1;
   out ter(7..0) out1;
} M2;

M2 inst4;
```

- defmod declaration and defmod variable declaration can only be done as global variable.
- defmod variable and pointer variable cannot be declared as a member of defmod.

```
defmod module3 {
    defmod module2 inst5;
    in ter(0:8)    *p;
};
```

- inout variable is not supported for member of defmod.

```
defmod module4 {
    inout ter(0:8) io1;
};
```

- In one defmod, clock cannot be declared twice or more. From 2nd time onwards, it must be described as in ter(0:1).

```
defmod module5 {                       defmod module5 {
    clock        clk1;                     clock        clk1;
    clock        clk2; /* Error */         in ter(0:1)  clk2;
    in  ter(0:8) a;                        in ter(0:8)  a;
    out ter(0:8) b;                       out ter(0:8)  b;
};                                     };
```

12. alias

An alias can be used to declare a different name of a variable.

Syntax

alias name(bit) signal_name(bit)

Example

```
ter IR(0:6);

alias OPE(0:4) IR(0:4);
alias DST(0:2) IR(4:2);

process main()
{

    var(0:4) x;

    x = OPE;  /* Alias OPE is used   */
      :       /* Same as x = IR(0:4) */
}
```

- Use of aliases in expressions and arrays is not supported.

- Only variables in ascending order are supported; variables in descending order are not supported.

- An alias cannot be declared inside a function.

13. OPERATORS

13.1 Priority of Operators

Priority	Operator	Concatenation rule
1	(:) (. .) (Note 1)	Right to left
2	( ) [ ] . -> ++ -- (Note 6)	Left to right
3	! ~ ++ -- (Note 6) & (Note 2) * (Note 3) + - (Note 7) &> |> ^> ˜&> ˜|> ˜^> size of	Right to left
4	(type)	Right to left
5	::	Left to right
6	* (Note 4) / %	Left to right
7	+ - (Note 7)	Left to right
8	<< >>	Left to right
9	< <= > >=	Left to right
1	== !=	Left to right
11	& (Note 5)	Left to right
12	^	Left to right
13	|	Left to right
14	&&	Left to right
15	||	Left to right
16	?:	Right to left
17	= += -= *= /= %= &= ^= |= <<= >>=	Right to left
18	,	Left to right

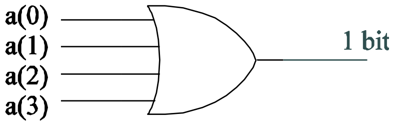
(Note 1) (:) and (. .) are bit specifications.
(Note 2) This & is an address operator.
(Note 3) This * is an indirect operator.
(Note 4) This * is an arithmetic operator that indicates multiplication.
(Note 5) This & is a logical operator that indicates an AND operation.
(Note 6) When ++ and -- are postfix type, their priority rank is 2 and when they are of prefix type their priority rank is 3.
(Note 7) When + and - are unary operators, their priority rank is 3 and when they are binary operators, their priority rank is 7.

13.2 Reduction Operator

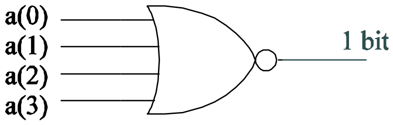
The reduction operator is a unary operator that reduces multi-bit width data to single-bit width data.

&>	Reduction AND	|>	Reduction OR	^>	Reduction XOR
~&>	Reduction NAND	~|>	Reduction NOR	~^>	Reduction XNOR

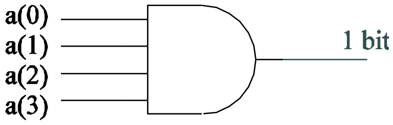
Reduction OR |>a(0:4)



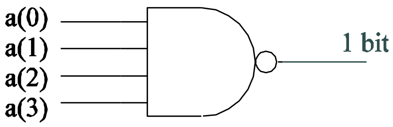
Reduction NOR ~|>a(0:4)



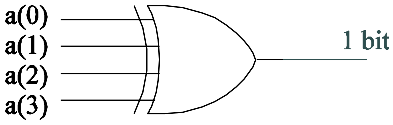
Reduction AND &>a(0:4)



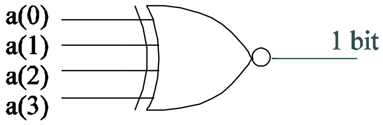
Reduction NAND ~&>a(0:4)



Reduction XOR ^>a(0:4)



Reduction XNOR ~^>a(0:4)



The reduction operators ^> and ~^> must use operands that have an established bit width. If the bit width has not been specified, or if the operation uses constants for which the bit width has not been specified, the reduction operator's operand will not be allowed.

In addition, we do not recommend using constants without a specified bit width or using operations involving constants without a specified bit width as the operand for the reduction operator &> or ~&>.

13.3 Concatenation Operator (::)

The concatenation operator is used to concatenate two variables.

This operator is similar to scope resolution operator (Refer to **13.4**), and it will change into scope resolution operator only when member function name is specified, but besides that it operates as a concatenation operator.

```
x(0:32) = a(0:16)::b(0:16);
```

The following shows a description that switches the upper and lower 16 bits of a variable defined by int in C language.

```
int a;
a = ((a << 16) & 0xFFFF0000) | ((a >> 16) & 0x0000FFFF);
```

In BDL, the concatenation operator can be used with partial bit field reference in a description such as the following.

```
int a;
a = a(16:16) :: a(0:16);
```

The concatenation operators can be used at the left side of an assignment. In the following example, the lower four bits of a are assigned to y and the upper four bits of a are assigned to x.

```
x(0:4) :: y(0:4) = a(0:8);
```

Note, however, that the Cyber behavioral synthesis system does not support using left-side values in expressions.

```
x = (y :: z = a + b); /* Not supported in expression */

if (a :: b = c)       /* Not supported in expression */
```

The concatenation operator's operand must have a specified bit width. A concatenation operator's operand if not allowed if it does not have a specified bit width or if it uses operations with constants that do not have a specified bit width. Even when the bit width is specified, a concatenation operator's operand is not allowed if it includes an operator with a carry operation (such as +, -, or *).

13.4 Scope Resolution Operator (::)

It is an operator which is used for specifying the relevant structure by that function when member variable which carried out prototype declaration is defined outside the structure.
This operator is similar to concatenation operator (Refer to 13.3) and it will change to scope resolution operator when it is used by member function for specifying the relevant structure, but besides that, it will operate as concatenation operator.

```
struct hoge {
    var(0:16) head, tail;
    var(0:32) get(); /* Prototype declaration of member function*/
};

template <typename T, int N>
struct foo {
    T c[N];
    T get(int v);   /* Prototype declaration of member  function*/
};
/* Definition of member function get() of structure hoge */
var(0:32) hoge::get() {
    return head::tail;      /* This :: is concatenation operator */
}

/* Definition of member function get() of template structure foo*/
template <typename T, int N>
T foo<T, N>::get(int v) {
    return c[v];
}
```

13.5 sizeof Operator

This operator is similar to sizeof() operator of C language and it calculates the byte count of type.
In the sizeof() argument, only type name or variable name can be described.

- Due to limitations, description combined with items besides variable name cannot be done even after including pointer operator (*) and address operator (&) when variable name is described.
- Not supported for defmod type
- Calculation of sizeof() is done as follows
- Bit count of type is divided by 8 and decimal number and rounded up value is assumed as byte count
- In case of pointer type, it is always 4 byte
- In case of array, value is the byte count of type multiplied by the array size

- In case of structure, byte count is calculated from the total value of bit count of all members.

  – In case the members of structure are array or structure, byte count of those members is calculated and the value multiplied with 8 is aggregated as the bit count.

  Specific example of byte count calculation of sizeof() is same as below:

```
out int x;
process main() {
    char *cp;
    struct hoge {
        ter(0:30) m1;
        var(0:10) m2[10];
    }st[5];
    x = sizeof(char); /* 1 */
    x = sizeof(cp); /* 4 */
    x = sizeof(struct hoge); /*(30 + ((10 / 8) * 10) * 8)/8 = 24 */
      x = sizeof(st); /* 24 * 5 = 120 */
}
```

13.6 exit Function

This function supports the C language's exit library function. It is described in the same way as a library function.

```
if (c) {
    exit(1);
}
```

13.7 Assignment Operator

This section describes how the assignment operator is handled.

13.7.1 Handling when several assignment operators exist in the same statement

```
int a;
int b;

a = b = x + y;
```

When a = b = x + y is described in C language, b = x + y is executed first, then an assignment to a is executed. In other words, in C language, examples 1 and 2 below are equivalent.

(Example 1)

```
int a;
int b;

a = b = x + y;
```

(Example 2)

```
int a;
int b;

b = x + y;
a = b;
```

Unlike C language, BDL supports the physical type in variables and therefore caution is required when using the Cyber behavioral synthesis system to synthesize in manual scheduling mode since values are updated differently according to the physical type. Specifically, Examples 3 and 4 below are not equivalent. In Example 4, an assignment is made to reg type variable b as a result of b = x + y, but the value is not updated until the next clock boundary, so the value assigned to variable a by a = b is not x + y but the value that existed before assignment of variable b's x + y.

(Example 3)

```
ter(0:8) a;
reg(0:8) b;

a = b = x + y;
```

(Example 4)

```
ter(0:8) a;
reg(0:8) b;

b = x + y;
a = b;
```

Example 5 shows a description that is equivalent to Example 3. In this case, variable b and the ter type variable that differs only in its physical type are initially assigned to tmp, then tmp is assigned to variable b and variable a.

(Example 5)

```
ter(0:8) a;
reg(0:8) b;
ter(0:8) tmp;

tmp = x + y;
b = tmp;
a = tmp;
```

14. CONTROL STATEMENTS

BDL provides not only the same control statements as in C language but also some other control statements that are convenient for hardware descriptions. This Section describes these BDL-specific control statements first, and then describes the goto statement (that also exists in C language) along with goto0 statement.

- BDL-specific control statements
  - wait statement and watch statement
  - dmux statement
  - nmux statement
  - allstates statement
  - goto0 statement

14.1 Wait Statement

Syntax

wait (expression) statement

The wait statement is a control statement that indicates a loop similar to a do-while loop; the statement is executed repeatedly, once per clock cycle, until its **expression** is evaluated as true.
The statement is executed, and then the expression is evaluated. If it is false, the statement is executed again. When the expression becomes true, the loop terminates. (Note that a wait is similar to a do-while, not to a while loop).
The wait statement is significant in the following two ways with regard to hardware.

- **Statement** and **expression** indicate completion of one cycle
  During manual scheduling mode under the Cyber behavioral synthesis system, the timing descriptor $ cannot be used in descriptions of wait processing.
  In the automatic scheduling mode of the Cyber behavioral synthesis system, a wait statement must be processed in one clock cycle, otherwise the synthesis will fail.

- Processing of the wait statement begins once all operations prior to the wait statement have been completed.
  Accordingly, if wait(1) is described, the operations after wait(1) will not begin until the operations before wait(1) have been completed.

(Example 1) Behaviorally equivalent description

```
Processing of A                       Processing of A
wait(c);                              L1:
Processing of B            →          if (!c){goto L1;}
                                      Processing of B
```

(Example 2) Behaviorally equivalent description

```
                                      L1:
wait(c);{Processing of A}             Processing of A
Processing of B            →          if (!c){goto L1;}
                                      Processing of B
```

About wait(1)

The description of wait(1)statement sets the behavior of evaluating the "statement" part only once, which is the same as when only the statement part is described.

However, in terms of hardware it means that the statement will be executed in the same clock cycle. In Example 3 below, x = a and y = b are executed in the same clock cycle. Once the operations before wait (here, d = 1 statement) have been executed, the operations in the statement (x = a and y = b) are executed, and then the operations that come after the wait (c = x + y and e = 1) are executed.

(Example 3)

```
    in  int  a,b;
    out int  c,d,e;

process main(){

    int x,y;

    d = 1;
    wait(1){
        x = a;
        y = b;
    }
    c = x + y;
    e = 1;
}
```

14.2 watch Statement

Syntax

watch (expression) statement

The watch statement is similar to the wait statement, except that, while the wait statement's loop is not executed until all operations described beforehand have been completed, the watch statement sets the order of execution based solely on data dependency, regardless of whether other operations are described before or after the watch statement. In other words, when there is no dependency among variables, the processing order before and after the watch statement is not preserved.

(Example 4)

```
a = b;     /* (1) */
watch(c); /* (2) */
x = c;     /* (3) */
y = z;     /* (4) */
```

In Example 4, x = c on line (3) is dependent on watch (c) in line (2), so line (3) is not executed until c is evaluated as true (c! = 0). However, y = z in line (4) is not dependent on watch (c), so it can be executed before watch (c). Likewise, a = b in line (1) is not dependent on watch (c), so it also can be executed after watch (c).

14.3 dmux Statement

Syntax

```
variable = dmux (expression) {
        case constant-expression:  expression;
        default:                   expression;
};
```

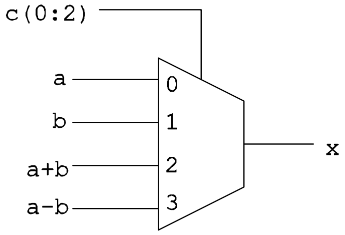
- * A semicolon (;) is required at the end of dmux () { ... }.
- * All case expressions must be of different values.
- * Over 32 bit constant cannot be described in the constant expression of case label.

A dmux statement is a control statement that represents a multiplexer with a decoder. If a case matches the expression value, the case expression is evaluated and evaluated value is assigned to the variable. If none of the other cases are matched the expression of the case labeled default is evaluated and evaluated value is assigned to the variable. (dmux is an abbreviation of "decoder multiplexer").

(Example)

```
x = dmux (c(0:2)) {
    case 0:  a;
    case 1:  b;
    case 2:  a+b;
    default: a-b;
  };
```

The description shown above generates the multiplexer shown below.



14.4 nmux Statement

Syntax

```
variable ::= nmux {
         when conditional-expression:  expression;
         default:              expression;
};
```

Caution: The operation of the synthesized circuit is not guaranteed unless each when conditional expression is exclusive.

The nmux statement is a control statement that represents a multiplexer without a decoder. When the value of the conditional expression in the when clause is true, the expression of the when clause is evaluated and evaluated value is assigned to the variable (nmux is an abbreviation of "no decoder multiplexer").
Under the Cyber behavioral synthesis system, the nmux statement is supported only when its transfer type is always-connected (::=), so this statement can be used only in allstates statements or during manual scheduling mode.

When the variable is a register

```
reg x;
x ::= nmux {
        when c1: a;
        when c2: b;
        default: x;
};
```

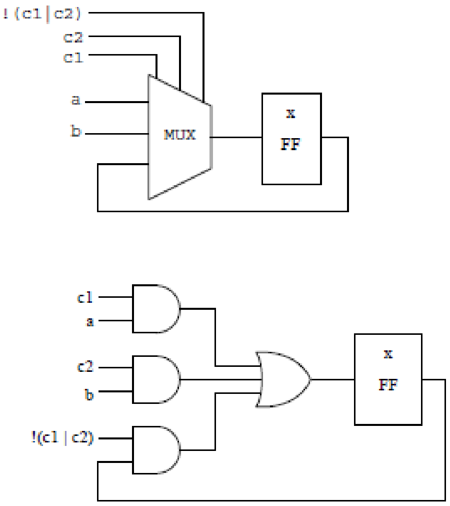
When the variable is a terminal

```
ter x;
x ::= nmux {
        when c1: a;
        when c2: b;
        default: 0;
};
```

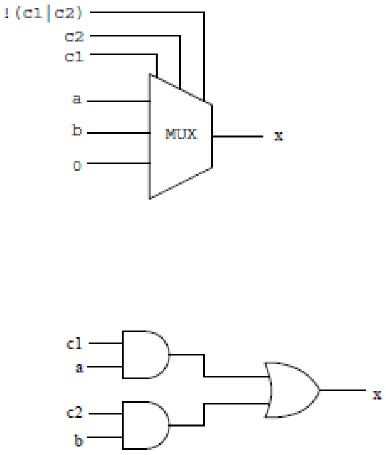
* "default: 0" can be omitted for ter type.

The description shown above generates the following logic.

Register



Terminal



14.5 allstates Statement

Syntax

allstates statement

The allstates statement is provided as a means of representing a software reset. The statement within the allstates statement is executed during all cycles (i.e. in all states).
In Example 1 below, the exit function is called when the value of the input variable ad is other than 0. In Example 2, operation of the process function is passed to label L1 when the value of the input variable flag is other than 0.

(Example 1)

```
in ter(0:8) ad;

allstates {
   if (ad != 0) {
      exit(1);
   }
}

process main() {

}
```

(Example 2)

```
in ter(0:8) flag;

process main() {

    allstates {
       if (flag != 0) {
          goto L1;
       }
    }

   L1:

}
```

- allstates statements can be described only outside of a function or at the start of a process function.

- The jump destination label of the goto statement within an allstates statement must be within a process function.

- In case function is called within the allstates statement, the content of that function is treated as allstates statement.

<Restriction in Cyber>

- There are several restriction on the use of an allstates statement. (For details, refer to the Cyber Behavioral Synthesis System Reference Manual.)

- allstates statements cannot be described in two or more places.

14.6 goto statement /goto0 statement

Syntax

```
goto0 label;
label:  statement;

goto label;
label:  statement;
```

goto statement and goto0 statement are control statements specifically for BDL, and can be used as unconditional jump statement, similar to the way they are used in C language. The label specified in a goto / goto0 statement must be a label for location present within the function.

The goto statement in BDL incorporates the functionality of clock boundary specifier. In other words, a goto statement can implicitly specify a clock boundary without having to explicitly specify it using $. On the other hand, in a goto0 statement, since the meaning of clock boundary specifier is not included hence, statements specified before and after goto0 statement are executed in same cycle.
Examples 3, 4 and 5 below are all same. In other words the goto statement of example 3 is equivalent to combination of $ and goto0 statement in example 4. Further it is equivalent to $ indicated by (a) in example 5.

(Example 3)

```
y = in1;
out1 = 1;
i = 0;
$
L1:
if( i < y ) {
    x[i] = in2;
    i++;
    goto L1;
}
$
out1 = 0;
```

(Example 4)

```
y = in1;
out1 = 1;
i = 0;
$
L1:
if( i < y ) {
    x[i] = in2;
    i++;
    $
    goto L1;
}
$
 out1 = 0;
```

(Example 5)

```
y = in1;
out1 = 1;
i = 0;
$

while( i < y ) {
    x[i] = in2;
    i++;
    $ /* (a) */
}
$
 out1 = 0;
```

15. FUNCTIONS

In BDL, functions can be used in the same way as in C language.

15.1 Return Values

A function returns either no value or only one value using a return statement.

15.1.1 Functions without return values

We recommend using the void type to declare functions that do not have return values. A return statement cannot be used inside a function whose return type is **void**.

```
void funcA( var(0:8) i ){


    return;
}
```

15.1.2 Specification of function's return value type

We recommend declaring a return value type for a function that will return a value in its definition. For the return value's physical type, we recommend specifying var type or ter type, or no specification at all (at present, specifications have not been established for cases where the reg type or mem type is specified as the return value type). The Cyber behavioral synthesis system does not support functions whose return value type is a pointer.

The return value is returned from each function to its caller by using a return statement. Any kind of expression can follow a return statement. Also, parentheses are not required in return statements. The expression shown below is converted to the function's type and is returned to the function’s caller.

```
var(0:8) funcB( var(0:8) i ){

    if( i == 255 ){
        return ( i );
    }
    return i + 1;
}
```

In cases where the function's type is not declared clearly, C language assumes the type to be int, but no such assumption has been established in BDL. Cyber behavioral synthesis system infers the type of the function whose type is not declared based on the expression returned by the return statement. Nevertheless this inference is not work for an expression consisting of a structure variable. Consequently, we do not recommend omitting declaration of the function's type, since the function is not guaranteed to suit the user's intention when declaration of the function's type is omitted.

```
funcC( var(0:8) i ){


    return i ;
}
```

15.1.3 Functions that return multiple values

Functions are able to return multiple values in the following three ways.

- The concatenation operator is used to combine multiple variables into one
- A structure is used to combine multiple variables into one
- Pointers to variables storing the return values is used in function's arguments

1. Using the concatenation operator to combine multiple variables into one
   When a function returns a value, it can use a concatenation operator to combine multiple variables into the one. The function's caller can divide it by using a concatenation operator to assign a return value to multiple variables.
   Note with caution that if the function's type and the total bit width of the variable used to return the value do not agree, a correct value cannot be returned.

```
process main(){
    var(0:8) a;
    var(0:4) x,y;

    x::y = funcA(a);
}

var(0:8) funcA(var(0:8) i){
    var(0:4) j,k;
        .
        .
    return  j::k ;
}
```

2. Using a structure to combine multiple variables into one
   When a function's type is declared as a structure, multiple variables can be returned as a single structure variable.
   A type check is performed whenever a structure variable is used, so this method is safer than the method that uses a concatenation operator. In addition, this method enables the structure's member variables to be revised at a later time.

```
struct stA {
    int m : 4;
    int n : 4;
};

process main(){
    var(0:8) a;
    var(0:4) x,y;
    struct stA t;

    t = funcA(a);
    x = t.m;
    y = t.n;
}

struct stA funcA(var(0:8) i){
    var(0:4) j,k;
    struct stA s;
    .
    .
    s.m = j;
    s.n = k;
    return s ;
}
```

3. Using pointers to variables storing the return values in function's arguments
   When pointers to variables to be changed are passed from the function's caller to a function using arguments, the function can store return values to the variables. If a structure is used as the variable to be changed, only one argument is needed to return multiple values.

```
process main(){
    var(0:8) a;
    var(0:4) x,y;

    funcA(a,&x,&y);
}

void funcA(var(0:8) i,
           var(0:4) *p,
           var(0:4) *q
          ){
    var(0:4) j,k;
        .
        .
    *p = j;
    *q = k;
}
```

```
struct stA {
    int m : 4;
    int n : 4;
};

process main(){
    var(0:8) a;
    var(0:4) x,y;
    struct stA t;

    funcA(a,&t);
    x = t.m;
    y = t.n;
}

void funcA(var(0:8) i,
           struct stA *s
          ){
    var(0:4) j,k;
        .
        .
    s->m = j;
    s->n = k;
}
```

15.2 Function parameters

The var type is recommended as the physical type of the function's parameters, except for pointer variables.

The specification has not yet been determined for usage of the ter, reg, or mem type for function's parameters. In the Cyber behavioral synthesis system, any parameters other than pointer variables are automatically converted to var type by the BDL parser (a warning is output when this conversion occurs).

```
var(0:8)
func( var(0:8) m,
      var(0:8) n ) {
    reg(0:8) x;

    x = n;
    $
    x += m;
    $
    return x ;
}
```

```
var(0:8)
func( var(0:8) m ) {
    reg(0:8) x;

    x = m;
    $
    m = x + 1; /* NG */
    $
    return m ;
}
```

15.3 Cautions Regarding Functions

1. Recursions are not supported by the Cyber behavioral synthesis system.

2. In BDL, as in C language, when there are several function calls in the same statement, the order in which the called functions are evaluated is not specified. In other words, in the example shown below we cannot tell whether func1() or func2() will be evaluated first.

   ```
   z = x + y + func1() + func2();
   ```

   In consideration of the operation faults that can occur depending on the evaluation order of func1() and func2(), we recommend splitting the statement as shown below in order to clarify the evaluation order.

   ```
   tmp1 = func1();
   tmp2 = func2();
   z = x + y + tmp1 + tmp2;
   ```

15.4 Synthesis of Functions by Cyber Behavioral Synthesis System

The Cyber behavioral synthesis system is able to synthesize functions in the following three ways.

Inline expansion of function

This method replaces function calls with the function's actual processing.
Expansion of the processing of all called functions can add to the time required for synthesis. However, this method is able to generate parallelism that extends beyond the functions, which tends to reduce the number of processing steps. On the other hand, this reduction in the number of processing steps comes at a cost of having more processing per step, which tends to generate a larger design.

goto conversion of function

This method replaces a function call with a goto statement that passes processing control to the function.
In cases where there are a lot of function calls, this tends to generate a smaller design than when synthesizing with inline expansion of functions, but since one cycle is needed for each function call, it also tends to require more processing steps than inline expansion does.

Synthesis of function

This method handles function calls as operators and assigns to the functional unit.
This method handles function calls as operators so that the function's processing is assigned to low-level operation modules. In addition, parallel control can be set up by specifying a number of low-level operation modules to be used. In other words, increasing the number of operation modules enlarges the design, but it also enables a reduction in the number of processing steps.
However, some functions cannot be synthesized with functional unit depending on the type of processing to be executed within the function. For further description of the method used to synthesize functions, refer to the Section on functions in the Cyber Behavioral Synthesis System Reference Manual.

15.4.1 Description of goto conversion of function in manual scheduling mode

When goto conversions are used to synthesize functions while in manual scheduling mode, the timing descriptor **$ must be described to represent the clock cycles before and after each function call**.
Since using goto conversions to synthesize a function means that the function's processing is executed in separate cycles, the timing descriptor $ must be used to divide the clock cycles before and after each function call, as is shown in Example 1 below. In other words, in cases where other processing occurs in the same cycle as a function call, the function cannot be synthesized via goto conversion.

(Example 1)

```
x = a;
y = b;
$
z = func(x, y);
$
```

(Example 2)

```
$
x = a;
y = b;
z = func(x, y);
$
```

15.5 Function Prototype Declarations

In C language, before a function can be called, the function type and its parameter type must be declared. This type of declaration is called a function prototype declaration. We recommend describing function prototype declarations in BDL as well.

```
/* Function prototype declaration */
int add3(int x, int y, int z);

in int i1,i2,i3;
out int o;

process main(){

    o = add3(i1,i2,i3);
}

/* Definition of function */
int add3(int x,int y,int z){

    return(x + Y + z);
}
```

If a function prototype declaration has not been described or if the function type or its parameters type has not been described in the function prototype declaration, an error will not occur in the present version to ensure compatibility with previous version, but such omissions are not recommended.

```
/* Function prototype declaration */
int add3();                /* Dummy argument type has not been described */
sub3(int x,int y,int z); /* Function type has not been described       */

process main(){

    o1 =  muladd(a,b,c); /* No prototype declaration */
    o2 =  add3(a,b,c);
    o3 =  sub3(a,b,c);
}

/* Function definitions */
int add3(int x, int y, int z){
       return(x + y + z);
}
int sub3(int x, int y, int z){
       return(x - y - z);
}
int muladd(int x, int y, int z){
       return(x * y + z);
}
```

15.5.1 Function with extern declaration

```
extern char func1(char in1);
```

Since the Cyber behavioral synthesis system does not support input of multiple source files, the extern specification has no significance. However, in case when extern has been specified for a function then that function is handled as an operator. Definition of a function with an extern declaration is not required (for details, refer to the Section on functions in the Cyber Behavioral Synthesis System Reference Manual).

16. PROCESS FUNCTION

In C language, a special function "main" is called to start any C program. In BDL, program execution starts with a "process function". Consequently, BDL description must contain this process function. The process function is given the special type “process” instead of a standard returned value type in the definition of the function.

```
in  ter(7..0) a, b;
out ter(8..0) c;

process add( ){

    c = a + b;

}
```

The Cyber behavioral synthesis system generates logic for repeated operation of process functions. How the Cyber behavioral synthesis system handles a value in a variable, while starting or ending of a process function operation such as via return or exit, depends on the option/attribute used at the time of synthesis. For details, refer to the Section on starting and ending process functions in the Cyber Behavioral Synthesis System Reference Manual.

- Restrictions on Cyber

– The Cyber behavioral synthesis system does not support coding of multiple process functions in one file. Therefore, in order to synthesize multiple process functions, each function must be set to a separate file which will be synthesized separately.

– Global variable, which is not initialized at the declaration, is treated as undefined. In case 0 is expected, 0 should be assigned explicitly by initialized declaration.

* However, at present, in case static is specified for local variable of function other than process function, 0 initialization is done implicitly.

17. TEMPLATE

In BDL, it is possible to use template of C++ language with limitations (Please refer to **Section 17.4** for details of limitations).
For example, in case same process is desired for variable with different data type, array size, bit width etc. like array or pointer of reg, shared reg, shared mem then they all can be described with 1 function by using the template function.

17.1 Template definition

- In BDL it is possible to define template for function and structure
- In continuation of description of 'template', list of parameters is specified within '<' '>' and then function or structure to be made into template is defined.
- Template parameters can be used in place of constant or type name inside its function or structure definition.
- Template parameter is declared with int type or typename type.
  - Declare with int type if you want to use parameter as constant value
  - Declare with typename type if you want to use parameter as type name

17.1.1 Template of Function

- In parameter template, it is possible to change the return value of function or parameter, type of local variable and constant value used in the function, to parameter.
- Below is the example of description of template function to copy the data between arrays of arbitrary type.

```
template <typename T1, typename T2, int N1, int N2>
void dataCopy(T1 *v1, T2 *v2) {
    int i;
    int t;
    for (t = 0, i = N1; i < N1 + N2; t++, i++) {
        v1[i] = v2[t];
    }
}

shared reg(0:32) sr[5];
outside mem(0:32) om[10];

process main()
{
    var(0:32) v[10];
    /* Copy the value from sr[0] - sr[4] to v[3] - v[7]    */
    dataCopy<var(0:32), shared reg(0:32), 3, 5> (v, sr);

    /* Copy the value from v[0] to v[9] to om[0] to om[9] */
    dataCopy<outside mem(0:32), var(0:32), 0, 10> (om, v);
}
```

- In particular template parameters can be used in the following locations in the template function.
  - In case of int type parameter

* Bit specification of function's return value

* Bit specification of local variable or parameter of function

* Array size specification of local variable or parameter of function

* Initial value of local variable or parameter of function

* Bit specification of 'type cast'

* case label of switch or dmux

* Specification of subscript of expression or array etc.

– In case of parameter of typename type

* Return value type of function

* Parameter type of function

* Type of local variable

* Cast type

- In case of specifying attribute in template function, specify in the name of 'template'

```
/* Attribute is specified here*/
template <int N>
void func() {
   ...
}
```

- Please take care that below limitations are there at the time of defining template function (Refer to 17.4 also)

– Prototype declaration of template function is not possible.

```
in int ter in1;
out int ter out1;

/* Not supported in the prototype declaration of template
   Function */
template <typename T>
T func(void);

process bdlmain()
{
    char reg r;

    r = func<char>();

    out1 = r;
}
/* It will be OK if prototype declaration is removed and the
   following definition is moved to the position where prototype
   declaration was there */
template <typename T>
T func(void)
{
    return in1;
}
```

- A name similar to function defined as template cannot be given to any other function or variable.

```
in int ter in1;
out int ter out1;

/* Define template function AAA */
template <typename T1>
T1 AAA(void) {
    return in1;
}

/* Function with a name similar to template function name cannot
   be defined*/
ter(0:8) AAA(int v1) {
    return v1 + 1;
}

process bdlmain()
{
    int AAA; /* Variable with a name similar to template name
                cannot be defined*/
}
```

- Template function which is being defined cannot be used in its own definition. And abbreviation of template argument is not possible.

```
/*Template function AAA */
template <typename T1>
T1 AAA(void) {
    T1 retval;
    /* Template function which is being defined cannot be used */
    retval = AAA<2>();
    /* Abbreviation of template argument is not possible */
    return retval + AAA();
}
```

17.1.2 Template of Structure

- Below is the example of description of structure type template.

```
template <typename T, int N>
struct foo {
    T *p1;
    mem ary[N];
};

shared reg r[4];
outside mem m[10];

in ter in1;
out ter out1;

/* Structure type having shared reg *p1 and the member
   of mem ary[10] */
struct foo <shared reg, 10> f1;

/* Structure type having outside mem *p1 and the member
   of mem ary[20]*/
struct foo <outside mem, 20> f2;

process main()
{
    f1.p1 = r;
    f2.p1 = m;
}
```

- In template definition of structure template parameter can be used in the following locations
  - In case of int type parameter
    - * Bit specification of member variable
    - * Bit field of member variable
    - * Array size specification of member variable
  - In case of typename type parameter
    - * Member type
- Please take care that below limitations are there in template structure (Refer 17.4 also)
  - It is not possible to declare structure tag of template structure

```
in int ter in1;

/* Structure tag declaration of template structure is not
   supported */
template <typename T> struct foo;

process bdlmain()
{
    struct foo<char> f;

    f.t = in1;
}
/* It will be OK if structure tag declaration is removed and
   the below definition is moved to the position of structure
   tag declaration */
template <typename T>
struct foo {
    int i;
    T t;
};
```

– Name similar to the structure defined as template cannot be given to a function or variable.

```
template <typename T1>
struct AAA {
    T1 t;
};

/* It is not possible to define a function with name similar to the
   template structure */
ter(0:8) AAA(int v1) {
    return v1 + 1;
}

process bdlmain()
{
    int AAA; /* It is not possible to define a variable with a name
                similar to the template structure */
}
```

– It is not possible to define other structure type in a template structure

```
struct bar {
    char c[2];
};

template <typename T1>
struct st1 {
    T1 t;
    /* Defining other structure in template structure is an error */
    struct foo {
        char c;
    }f;

    /* Since ‘struct bar’ has already been defined as global so it
       is not a problem */
    struct bar b;
};

process bdlmain()
{
    struct st1<int> s;
}
```

- Template structure type which is being defined cannot be used in its own definition. And abbreviation of template argument is also not possible.

```
template <typename T1>
struct st1 {
    /* Template structure type which is being defined cannot be
       used*/
    struct st1<T1> t;

    /* Abbreviation of template argument is not possible */
    /* It cannot be used also for return value of member function or
       argument type*/
    struct st1 func( struct st1 v ) {
        ...;
        return *this;
    }
};
```

17.2 Method to Use

- Specify actual argument within '<' '>' in continuation of structure & function name defined in template.
- Only the constant value of range (from -2147483648 to 2147483647) can be described with signed int in actual argument of int type parameter
  - In case constant value is defined with #define, then its name can also be specified
  - In case the result of operations turns out to be an constant then the formula can also be used
  - const variable having the initial variable can also be used (Error in case of array)
  - The concrete example of specifying actual argument is given below

```
in int ter in1;
out int ter out1;

#define OUT_IDX 3

const int WIDTH = 8;
const int TBL[10] = {1, 2, 4, 8, 16, 32, 64, 128, 255, 0};

template <int N>
struct foo {
    char v[N];
};


template <int N>
void func(char *v1) {
    out1 = v1[N] + in1;
}

process bdlmain()
{
    /* constant const and calculation between constants is OK */
    struct foo<WIDTH + 1> h = {{1, 2, 3, 4, 5, 6, 7, 8}};

    /* Character string defined in constant with #define is also OK
     */
    func<OUT_IDX> (h.v);

    /* Not supported in array even in case of constant const */
    func<TBL[2]> (h.v);
}
```

- Only typename can be specified in the actual argument of typename type parameter
  - typename defined with typedef also can be specified in actual argument of typename type
  - template structure type can also be specified
  - In case of typename which does not stand firm as 1 type with only specification of actual argument part, it ends up as an error due to limitation (Refer to **17.4** for details)
  - The concrete example of specifying actual argument is as given below

```
in int ter in1;
out int ter out1;

template <typename T>
struct foo {
    T v[10];
};

template <typename T1, typename T2>
T2 func(T1 *v1) {
    return v1->v[0] + in1;
}

shared reg r;
/* Type of template structure can also be defined as another name
   with typedef */
typedef struct foo <signed var(0:8)> FOO_V;

process bdlmain()
{
    FOO_V h = {{1, 2, 3, 4, 5, 6, 7, 8}};
    /* template structure type defined with typedef can also be
       specified as actual argument*/
    out1 = func<FOO_V, signed var(0:8)> (&h);

    /* It is an error caused by limitation in case of specification
       which are not of same type (such as shared or outside,
       const), cannot be created by actual argument only */
    r = func<signed var(0:8), shared> (h.v);
}
```

17.3 Regarding the entity of template

- In case template structure or template function is used its instantiation has been done automatically.
- The name of instantiated template consists of *"template name"_"the value of actual argument of template"*.
- In case template function is used as the user defined functional unit then its entity name will become the name of functional unit

– Name of int type parameter part will become the constant value of that actual argument.

– Name of typename type parameter part will become the name made by shortening and stringing the type information inside that type

- In case the name of entity to be generated is already being used as the name of other function or structure, then '_number' is added always after that

– '_number' starts with 1 and it will increase by 1 every time when there is a name conflict

- In case optional template is used multiple number of times in actual argument of same meaning then the all entity of those templates will be same
- Examples of template entity generation are given below.

– Example of BDL using the template

```
/* Template structure */
template <typename T>
struct foo {
    T m1;
};

/* Template function with attributes */
/* Cyber func = inline */
template <int N>
int func (void){
    return N;
}

/* Normal (not template) function */
int func_10(void) {
    return 0;
}
const int C = 10;

process main()
{
    struct foo<signed var(0:32)> f1;
    /* int type is same as signed var (0:32) */
    struct foo<int> f2;

    f1.m1 = func<C> (); /* C is a const variable
                           which has initial value
                           10 */
    f2 = f1;
}
```

– Analysis result of above mentioned BDL example

```
/* Entity of structure type of variable f1 and f2 */
struct foo_SgnVarS0W32 {
    signed var(0:32) m1;
};

signed const ter(0:32) C = 10;
signed var int func_10();

/* Prototype declaration of entity of template function
   func <C> '_1' is added to the entity name of template function
   in order to avoid function name conflict with the existing
   function 'func_10'*/
signed var int func_10_1();

process main()
{
    struct foo_SgnVarS0W32 f1;
    struct foo_SgnVarS0W32 f2;

    f1.m1(0:32) = func_10_1();
    f2 = f1;
}

signed var int
func_10()
{
    return (0);
}

/* Entity of func<C> */
/* Cyber func = inline */
signed var int
func_10_1()
{

       return (10);
}
```

17.4 Limitations

Following limitations are there in BDL template functionality

- In case of using template, it is necessary to define it.

```
process main()
{
    /* It is an error to use the template before defining
       the template*/
    func<10> ();
}

template <int N>
int func() {
    return N;
}
```

- Structure tag declaration or prototype declaration of template is not possible.

```
/* Prototype declaration of template function is an error */
template <int N> int func(void);

/* Structure tag declaration of template structure is an error */
template <typename T> struct foo;

template <int N>
int func() {
    return N;
}

template <typenameT>
struct foo {
    T m1;
};
```

- Things other than signed int type and typename type cannot be specified in type of template parameter

```
Example of type of parameter that cannot be used:
  o unsigned, unsigned int etc., unsigned int type
  o bool, char, long etc., logic type other than int
  o Constant value
```

- Default value cannot be specified. in template parameter

```
/*Specifying default value in parameter of template is an
error*/
template <int N = 10, typename T = char>
void func() {
    T c = N;
}
```

- Template definition cannot be nested

```
template <int N>
int func1() {
    /* Nesting the template definition is an error */
    template <int T>
    struct foo {
        var(0:T) m1;
    };
    return N;
}
```

- Template without the parameter is not supported (Specialization is not supported)

```
/* Template definition without parameter, specialization is an
   error */
template <>
int func<1>() {
    return 1;
}
```

- Recursive call of template function cannot be done

```
template <int N>
int func() {
    /* Recursive call of template is an error */
    return N + func<N-1>();
}
```

- Template of same name with different parameters, cannot be defined (Overloading is not supported).

```
template <int N>
int func() {
    return N;
}

/* Defining a template with name similar to already defined
   template name is an error */
template <int N, int M>
int func() {
    return N + M;
}
```

- enum type cannot be defined within template function.

```
template <int N>
int func() {
    /* Defining enum type inside template function is an error */
    enum { A = 1, B, C, D, E } e1;

    return N;
}
```

- It is not possible to define another new structure in template structure.

```
template <int N>
struct foo {
    /* Defining other structure in template structure is
       an error */
    struct bar {
        ter(0:N) i;
    } b;
};

/* It is OK to define ‘struct  bar’ member in the template
structure since ‘struct bar’ is defined earlier than
the template structure*/

struct bar {
    ter(0:N) i;
};

template <int N>
struct foo {
    struct bar b;
};
```

- signed constant less than 32 bit cannot be defined in actual argument of int parameter part of template.

```
template <int N>
int func() {
    return N;
}

enum {A = 1, B, C, D, E} e1;

process main()
{
    int i = 10;

    /* variable other than const constant is an error */
    func<i>();

    /* error if constant is defined as enum */
    func<A>();

    /* Value greater that the value expressible with
       signed 32 bit is an error */
    func<0x123456789>();
}
```

- Only the type which does stand firm as 1 type with only the actual argument part, can be specified in actual argument of typename type parameter of the template

```
template <typename T>
void func() {
       T *v;
}
process main()
{
      /* Below types cannot be specified in actual argument
         alone*/
       func<signed>();
       func<unsigned>();
       func<shared>();
       func<outside>();
       func<extern>();
       func<auto>();
       func<register>();
       func<static>();
       func<const>();
       func<volatile>();
}
```

- const array cannot be specified in the actual argument of int type parameter of the template

```
template <int N>
int func() {
       return N;
}

const int ary[10] = {1,2,3,4,5,6,7,8,9};
const int cval = 10;

process main()
{
    /* Error in case of array even it is const variable */
      func<ary[0]> ();

    /* scalar of const is OK */
    func<cval> ();
}
```

- In case physical type of ter, reg etc. or in, out etc. is specified in declaration of int type parameter of the template, it will be ignored

```
Example of modifier, type when specification to parameter is
ignored:
    o ter, var, reg, mem
    o shared, outside
    o in, out, inout
    o static, const, extern, etc.
```

- Array or pointer etc. cannot be specified in declaration of parameter of the template

```
/*Specifying array in template parameter is an error*/
template <typename T[]>
struct bar {
    T ary;
};

/*Specifying pointer in template parameter is an error*/
template <typename *T>
struct foo {
    T p;
};

process
bdlmain()
{
    struct bar <char [2]> b;
    struct foo<int> f;
}
```

18. TIMING DESCRIPTOR

In BDL, the dollar symbol "$" is used to explicitly describe the timing of operations. This symbol is called the clock cycle boundary specifier. It is also called as the timing descriptor, as it specifies timing.
In the Cyber behavioral synthesis system, timing descriptors are valid when synthesized in manual scheduling mode, but they are ignored when synthesized in automatic scheduling mode. (However, timing descriptors are not ignored when it is described in the block contents specified by scheduling_block attribute even in the automatic scheduling mode.) In manual scheduling mode, operations that occur between clock cycle boundary descriptors all occur during the same cycle.

18.1 Clock Cycle Boundary Specifier

The clock cycle boundary can be specified by the specifier "$". A clock cycle boundary is either the rising or falling edge of a clock signal. Operations such as updating of values in register variables and initialization of terminal variables occur at the clock cycle boundary.

(Example 1)

```
in ter int in1,in2;
out ter int out1;

process main(){
    reg int x,y;

    x = in1;
    y = in2;
    out1 = x + y;
    $
}
```

(Example 2)

```
in ter int in1,in2;
out ter int out1;

process main(){
    reg int x,y;
    x = in1;
    y = in2;
    $
    out1 = x + y;
    $
}
```

- Example 1
The values in register variables x and y are not updated until the next clock cycle boundary. Accordingly, the sum of the values retained in register variables x and y are assigned to output variable out1, not the sum of values in input variables in1 and in2.

- Example 2
The values in register variables x and y are updated to the values in in1 and in2 respectively at the clock cycle boundary. Consequently, the sum of the values in in1 and in2 are assigned to the output variable out1.

18.2 Timing Descriptor Examples

(Places marked with ★ can be used to describe timing descriptor)
(Places marked with ● cannot be used to describe timing descriptors)

- Timing descriptors can be described immediately after semicolons (;) and wavy brackets ({ }).

- Timing descriptors can be described immediately after the end parenthesis ")" in a conditional clause (if, for, while, or do-while), but they cannot be described immediately before the start parenthesis "(".

```
w = a + b;
★
while ● (…) ★ {
     ★
     x = a + b;
     ★
}
★
y = a + b;
★
z = a – b;
```

- Timing descriptors can be described immediately after else but not immediately before else.

```
if ● (expr) ★ {

} ● else ★ if ● (expr) ★ {

} ● else ★ {

}
```

- Timing descriptors cannot be described before and after a switch conditional clause.

```
switch ● (c) ● {
    :
}
```

- Timing descriptors cannot be described within conditional clauses (if, switch, for, while, or do-while) of control statements.

```
switch (● c ●){
  :
}
```

- When used with the initialization clause or reconfiguration clause of a for statement

–Immediately after the last expression
–Immediately after a comma operator
–In reconfiguration clause, immediately after a semicolon

```
for ( ● i = 0 ●, ★ j = 0 ★; i < MAX; ★ i++ ●,★ j++ ★) {

}
```

- Timing descriptors cannot be described in wait, watch, dmux, nmux, or allstates statements.

- When timing descriptors are specified without putting {} in the body of if statement, it is treated as following:

```
if ( c1)  $                              if ( c1)  $  {
    a=b;                   →                 a=b;
    $                                    }
                                         $
```

In case of following description, timing descriptor will be treated as outside of the ‘if’ statement, thus caution is required.

```
if ( c1)                                 if ( c1)  {
                           →                 a=b;
  a=b; $                                 }
                                         $
```

Also, presence of “;” changes the meaning, thus caution is required.

```
while ( f1) $                         while ( f1) $ {
    a=b;                                  a=b;
                                       }
while (f2)  $;          →              while (f2)  $ {
 c=d;                                  }
                                       c=d;
```

Note: when $ is present, it is recommended to put { }.
Even though the position of $ differs in the descriptions shown in Example 1 and Example 2 below, both are for operations that occur between a condition judgment and processing of a then clause, so their meaning is equivalent.

(Example 1)

```
if( c != 0 ) $ {

    x = a + b;
}
```

(Example 2)

```
if( c != 0 ) {

    $

    x = a + b;
}
```

However, when $ appears twice such as in Example 3 below, it means that there are two clock boundaries after the condition judgment is executed.

(Example 3)

```
if( c != 0 ) $ {
   $
   x = a + b;
}
```

When $ appears twice consecutively such as in Example 4 below, it means that the number of clock boundaries is exactly as specified.

(Example 4)

```
x = a + b;
$
$
y = a - b;
```

18.3 Variable Update Timing

- Values assigned to a ter type variable are valid only during the assigned state and are not retained during the next state. If no values are assigned, the value of the ter type variable is zero.

–Since "1" is assigned to the ter type variable ta in state 1 of **Figure 1**, the value of the ter type variable ta in state 1 is "1".

–Since no value is assigned to the ter type variable ta in state 2 of **Figure 1**, the value of the ter type variable ta in state 2 is "0".

- Values assigned to a reg type variable are valid only during the next state after the assigned state and are not updated during the assigned state. If no values are assigned, the value of the previous state is retained.

–"ta" is assigned to the reg type variable ra in state 1 of **Figure 1**, but this value is not updated until state 2.

–Since the reg type variable rb is not assigned during state 5 and state 6 of **Figure 1**, the previous value is retained in the reg type variable rb.

```
Process main(){

    ter(0:4) ta;
    reg(0:4) rb = 0;

    /* state 1 */
    ta = 1;
    rb = ta; // (a)
    $
    /* state 2 */
    rb = 2+ta; // (b)
    $
    /* state 3 */
    rb = 3;
    ta = rb; // (c)
    $
    /* state 4 */
    rb = ta; // (d)
    ta = 4;
    $
    /* state 5 */
    $
    /* state 6 */
    ta = rb; // (e)
    $
}
```

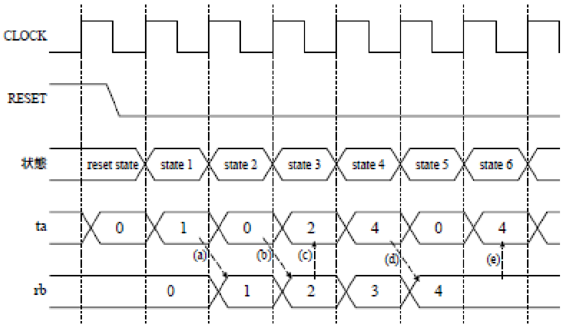


Figure 1 Example of Variable Update Timing

(Caution) During manual scheduling mode, all operations within each state are executed in parallel, and changing the order within the same state does not affect the behavior. However, we do not recommend using descriptions such as those shown in state 3 and state 4 of **Figure 1**, since the behavior differs between BDL and C language.

–We recommend assigning values to ter type variables in the same state before they are referenced.

–We recommend referencing values in the same state before they are assigned to reg type variables.

18.3.1 Multiple assignments

Since operations within the same state are executed in parallel in manual scheduling mode, multiple values cannot be assigned to the same variable during any single state. Assigning multiple values to the same variable in any single state is called “multi-assignment”. Description that contains multiple assignments is incorrect description because the values that are being assigned vary according to the implementation.
Figure 2 shows an example of multiple assignments. In the **Figure 2**, when the value of both flag1 and flag2 is 1, it cannot be determined whether the value of variable a, is a + 1 or 0. In this example, multiple assignments would not occur were it not for the fact that the value of both flag1 and flag2 is 1.

```
  in ter flag1;
  in ter flag2;

process main(){
  reg(0:8) a;
  a = 0;
  $
  if( flag1 ) {
    a = a + 1;
  }
  if( flag2 ) {
    a = 0;
  }$
}
```

Figure 2 Example of Multiple Assignments (1)

18.4 Timing Descriptions in Various Control Statements

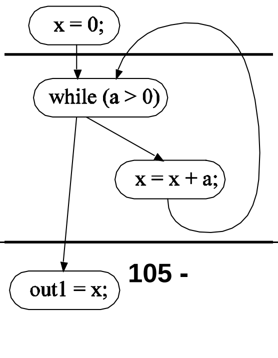
18.4.1 while statement

- Possible positions of $ description in while statement (indicated by □)

```
Processing A;
□
while ( c1 ) □ {
    □
    Processing B;
    □
}
□
Processing C;
```

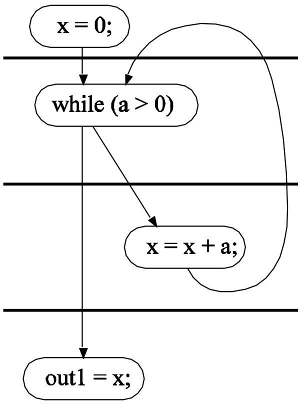
Other examples of timing descriptions in while statements are listed below.

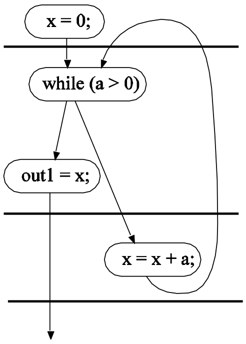
	while statement	Simplified expression (in steps)	Description

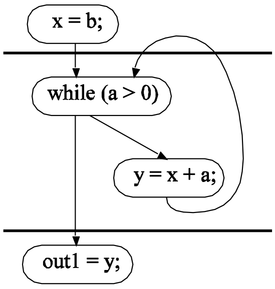
1	x = 0;
	$
	while (a>0)
	{
	  x = x +
	a;
			The $ before and after the while statement means that the processes before and after the loop processing are carried out in a different state. The $ at the end of the

	`while` statement	Simplified expression (in steps)	Description

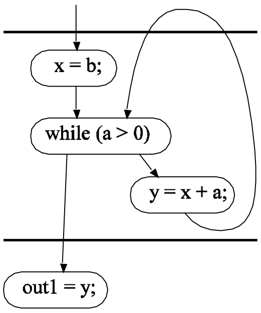
	`$`
	`}`
	`$`
	`out1 = x;`
			loop processing indicates the clock boundary for returning to the start of the loop.

2	`x = 0;`
	`$`
	`while (a>0) {`
	`$`
	`x = x + a;`
	`$`
	`}`
	`$`
	`out1 = x;`
			This means that the condition judgment and loop processing occur in different states.

3	`x = 0;`
	`$`
	`while (a>0) {`
	`$`
	`x = x + a;`
	`$`
	`}`
	`out1 = x;`
	`$`
			Since there is no $ between the while statement and out1 = x; the condition judgment is performed in the same state as out1 = x;.

4	`x = b;`
	`$`
	`while (a>0) {`
	`y = x + a;`
	`$`
	`}`
	`$`
	`out1 = y;`
			This specifies that "x = b" and "condition judgment and loop processing" occur in the different states.

	`while` statement	Simplified expression (in steps)	Description

5	`x = b;`
	`while (a>0) {`
	`  y = x + a;`
	`  $`
	`}`
	`$`
	`out1 = y;`
			This specifies that x = b and "condition judgment and loop processing" occur in the same state.
			x = b needs to be executed only once in the first cycle.

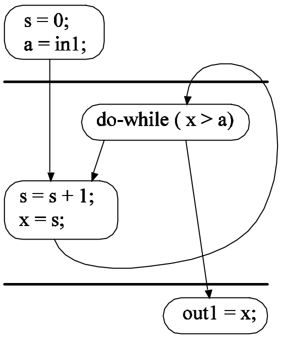
18.4.2 do-while statement

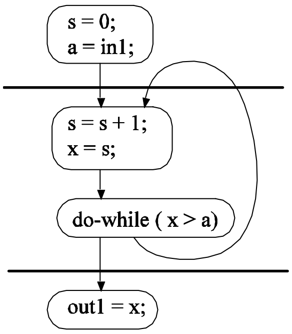
- Possible positions of $ description in do-while statement (indicated by □)

```
Processing A;
□
do {
    □
    Processing B
    □
}while ( c2 ) □ ;
□
Processing C;
```

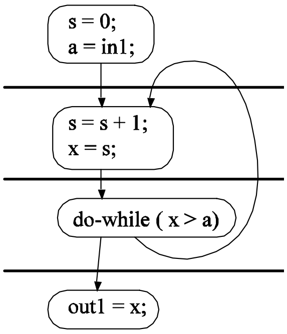
Other examples of timing descriptions in do-while statements are listed below.

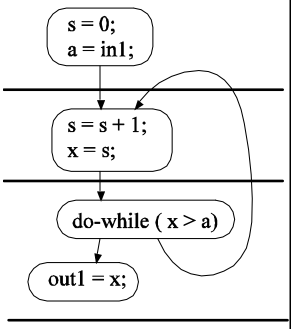
	do-while statement	Simplified expression (in steps)	Description

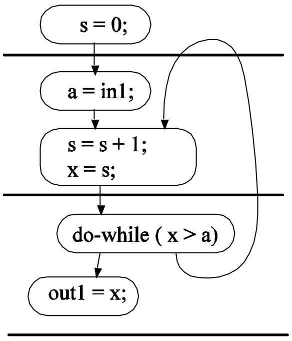
1	S = 0;
	a = inl;
	$
	do {
	    s = s + 1;
	    x = s;
	    $
	}while(x>a);
	$
	out1 = x;
		 s = 0;
		a = in1;
		do-while (x > a)
		s = s + 1;
		x = s;
		out1 = x;
			When $ appears immediately after loop processing:
			The first round of loop processing is executed regardless of the condition judgment. Afterward, a condition judgment is performed at the next state (during the next cycle).

2	S = 0;
	a = inl;
	$
	do {
	    s = s + 1;
	    x = s;
	}while(x>a) $;
	$
	out1 = x;
		 s = 0;
		a = in1;
		s = s + 1;
		x = s;
		do-while (x > a)
		out1 = x;
			When $ appears after a condition judgment:
			The first round of loop processing is executed regardless of the condition judgment. Afterward, a condition judgment is performed in the same state as loop processing.

	do-while statement	Simplified expression (in steps)	Description

3	S = 0;
	a = in1;
	$
	do {
	    s = s + 1;
	    x = s;
	  $
	}while(x>a) $;
	$
	out1 = x;
		 s = 0; a = in1;
		s = s + 1; x = s;
		do-while (x > a)
		out1 = x;
			When $ appears at the end of loop processing and after a condition judgment:
			The loop processing and condition judgment are executed in the separate states.

4	S = 0;
	a = in1;
	$
	do {
	    s = s + 1;
	    x = s;
	    $
	}while(x>a) $;
	out1 = x;
	$
		 s = 0; a = in1;
		s = s + 1; x = s;
		do-while (x > a)
		out1 = x;
			When there is no $ between a do-while statement and the next statement:
			out1 = x; is executed only once, in the same state as the condition judgment.

5	S = 0;
	$
	a = in1;
	do {
	    s = s + 1;
	    x = s;
	    $
	}while(x>a) $;
	out1 = x;
	$
		 s = 0;
		a = in1;
		s = s + 1; x = s;
		do-while (x > a)
		out1 = x;
			When there is no $ between a do-while statement and the previous statement:
			a = in1 is executed only once, at the start of the same state as loop processing.

18.4.3 for statement

- Possible positions of $ description in for statement (indicated by □)

```
Processing A;
□
for(i=0, □ j=0 □; i < x; □ i++, □ j++ □ ) □ {
    □
    Processing B;
    □
}
□
Processing C;
```

Description in which a for statement is replaced by a while statement:

```
e1 = 0;                              e1 = 0;
for(i=0;i<x;i++){                    i = 0;
    m[i] = 0;                        while(i<x){
}                         →              m[i] = 0;
e2 = 0;                                  i++;
                                     }
                                     e2 = 0;
```

The following describes the relationships when a for statement is replaced by a while statement.

	for statement	When replaced by while statement	Description

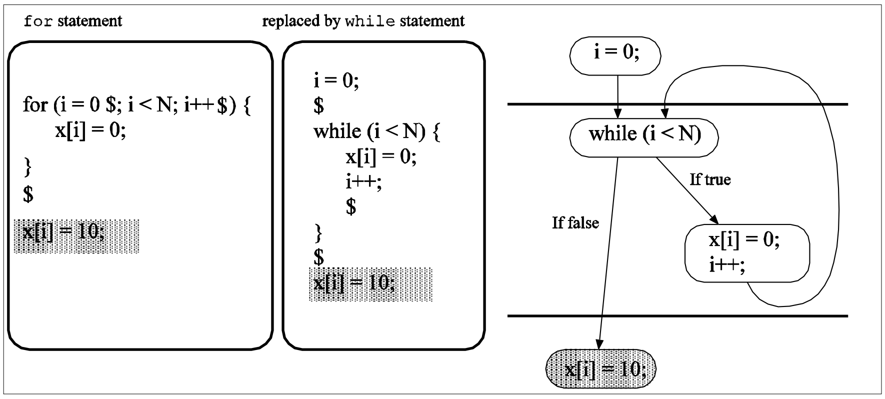
1	e1 = 0;
	$
	for(i=0 $;1<N;i++ $) {
	x[i]= 0;
	}
	$
	e2 = 0;
		el = 0;
		$
		i = 0
		$
		while(i<N)
		{
		x[i] = 0;
		i++;
		$
		}
		$
		e2 = 0;
			The $ before the for statement corresponds to the $ before the initialization that is executed before the while statement. The $ after the for statement corresponds to the $ after the while statement.

2	e1 = 0;
	for(i=0 $;1<N;i++ $) {
	x[i]= 0;
	}
	e2 = 0;
		el = 0;
		i = 0
		$
		while(i<N)
		{
		x[i] = 0;
		i++;
		$
		}
		e2 = 0;
			The $ at the end of the initialization statement corresponds to the $ after the initialization that is executed before the while statement.

	for statement	When replaced by while statement	Description

3	`e1 = 0;`
	`for(i = 0;1<N;  i++ $)`
	`{`
	`    x[i]= 0;`
	`    $`
	`}`
	`e2 = 0;`
		`el = 0;`
		`i = 0`
		`while(i<N)`
		`{`
		`  x[i] = 0;`
		`  $`
		`  i++;`
		`  $`
		`}`
		`e2 = 0;`
			The reconfiguration statement is executed after the loop processing in the while statement.

- Description in which $ appears immediately after the iteration statement of for statement x[i]=10 is executed in the next state of condition judgment.



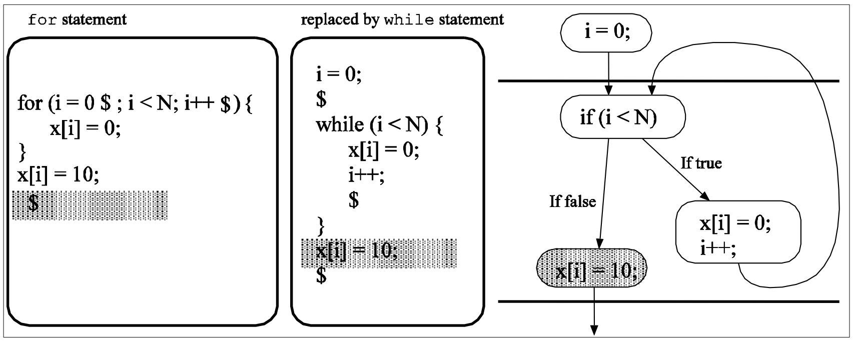
for statement

```
for (i = 0 $; i < N; i++$) {
    x[i] = 0;
}
$
x[i] = 10;
```

replaced by while statement

```
i = 0;
$
while (i < N) {
    x[i] = 0;
    i++;
    $
}
$
x[i] = 10;
```

- Description in which $ does not appear immediately after a for statement (x[i]=0 is executed in the same state as condition judgment)



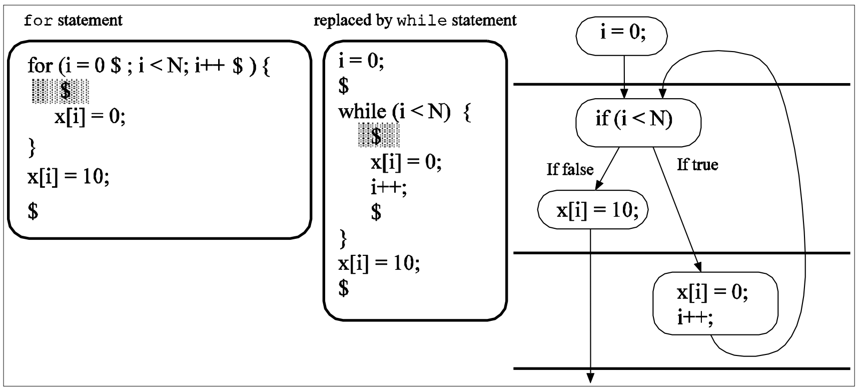
for statement

```
for (i = 0 $ ; i < N; i++ $) {
    x[i] = 0;
}
x[i] = 10;
$
```

replaced by while statement

```
i = 0;
$
while (i < N) {
    x[i] = 0;
    i++;
    $
}
x[i] = 10;
$
```

- Description in which $ appears at the start of loop processing (condition judgment and loop processing are executed in different states)



`for` statement

```
for (i = 0 $ ; i < N; i++ $ ) {
  $
    x[i] = 0;
}
x[i] = 10;
$
```

replaced by `while` statement

```
i = 0;
$
while (i < N)  {
    $
    x[i] = 0;
    i++;
    $
}
x[i] = 10;
$
```

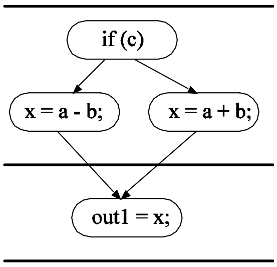
18.4.4 `if` statement

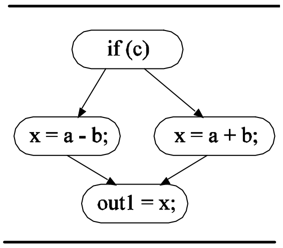
- Possible positions of $ description in if statement (indicated by □)

```
Processing A;
□
if( c1 ) □ {
   □
   Processing B;
   □
}
else □if ( c2 ) □ {
   □
   Processing C;
   □
}
else□ {
   □
   Processing D;
   □
}
□
Processing E;
```

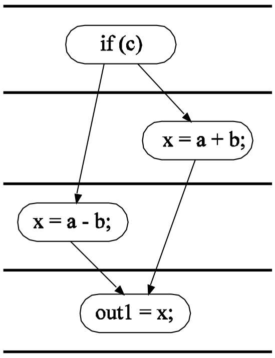
Description examples in which an if statement contains a timing description are shown below.

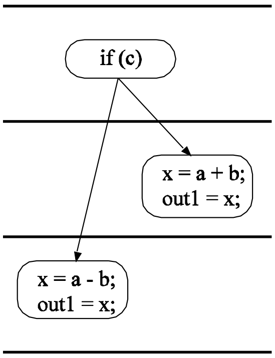
	`if` statement	Simplified expression (in steps)	Description

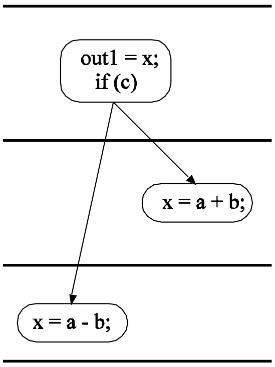
1	`if(c){`
	`    x = a + b;`
	`} else {`
	`    x = a - b;`
	`}`
	`$`
	`out1 = x;`
			If $ appears between an if statement and the next statement, all of the processing from the if statement is executed during the same state, and the next processing is executed at the next state.

2	`if(c){`
	`    x = a + b;`
	`} else {`
	`    x = a - b;`
	`}`
	
	`out1 = x;`
			If **there is no $** between an if statement and out1 = x, then out1 = x is executed during the same state as the processing for the then and else parts.

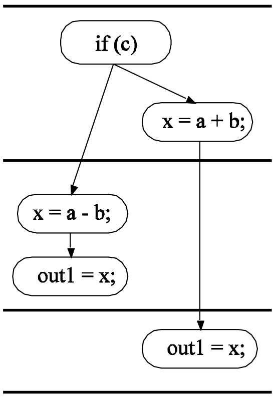
	`if` statement	Simplified expression (in steps)	Description

3	`if(c){`
	`$`
	`x = a + b;`
	`} else {`
	`$`
	`x = a - b;`
	`}`
	`$`
	`out1 = x;`
			This description means that each processing is executed in a different state.

4	`if(c){`
	`$`
	`x = a + b;`
	`} else {`
	`$`
	`x = a - b;`
	`}`
	`out1 = x;`
		 if (c)
		x = a + b;
		out1 = x;
		x = a - b;
		out1 = x;
			If $ does not appear between an if statement and out1 = x, then out1 = x is executed during the same state as the processing that is executed at the last state of each conditional branch.

5	`out1 = x;`
	`if(c){`
	`$`
	`x = a + b;`
	`} else {`
	`$`
	`x = a - b;`
	`}`
			out1 = x is described before the if statement so that the if statement's condition judgment and out1 = x will be executed in the same state.

	`if` statement	Simplified expression (in steps)	Description

6	`if(c){`
	`    x = a + b;`
	`    $`
	`} else {`
	`    $`
	`    x = a - b;`
	`}`
	`out1 =x`
		 if (c)
		x = a + b;
		x = a - b;
		out1 = x;
		out1 = x;
			If an **$** appears during an if statement's conditional branch but no other **$** appears after the if statement, the processing that follows the if statement will be executed in a different state after each branch.

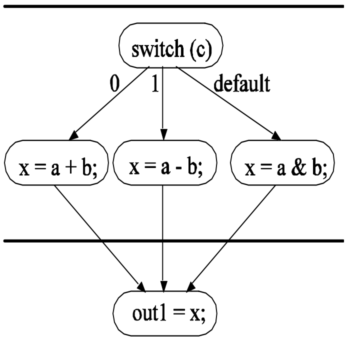
18.4.5 switch statement

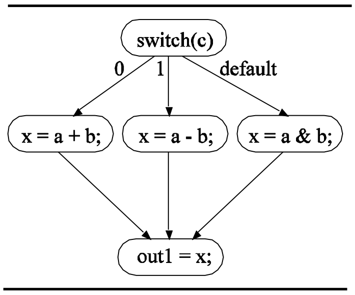
- Possible positions of $ description in switch statement (indicated by □)

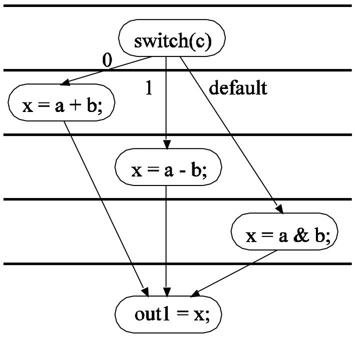
```
Processing A;
□
switch ( c ) {
  case 0:
      □
      Processing B;
      □
  case 1:
      □
      Processing C;
      □
  default:
      □
      Processing D;
      □
}
□
Processing E;
```

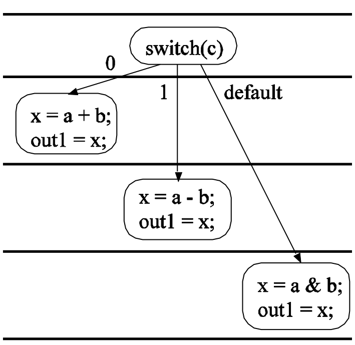
Description examples in which a switch statement contains a timing description are shown below.

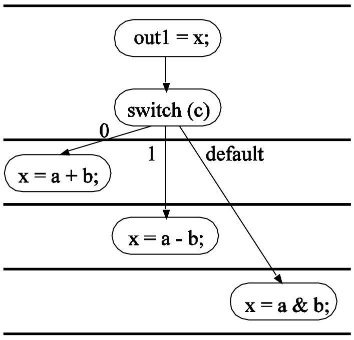
	switch statement	Simplified expression (in steps)	Description

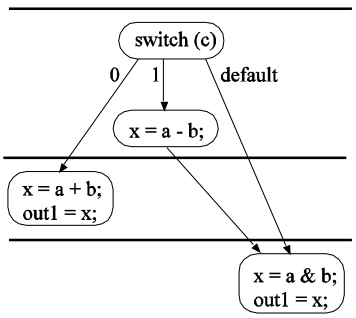
1	`Switch(c) {`
	`  case 0:`
	`    x = a + b;`
	`    break;`
	`  case 1:`
	`    x = a - b;`
	`    break;`
	`  default:`
	`    x = a & b;`
	`}`
	`$`
	`out1 = x;`
		 switch (c)
		0
		1
		default
		x = a + b;
		x = a - b;
		x = a & b;
		out1 = x;
			If $ appears between a switch statement and the next statement, the next processing will be executed in the state that follows the end of the switch statement's processing.

2	Switch(c) {
	case 0:
	x =a + b;
	break;
	case 1:
	x =a - b;
	break;
	default:
	x = a & b;
	}
	out1 = x;
		 switch(c)
		0
		1
		default
		x = a + b;
		x = a - b;
		x = a & b;
		out1 = x;
			If $ does not appear between a switch statement and the next statement, the next processing will be executed in the same state as the switch statement's processing.

3	Switch(c) {
	case 0:
	$
	x = a + b;
	break;
	case 1:
	$
	x = a - b;
	break;
	default:
	$
	x = a & b;
	}
	$
	out1 = x;
		 switch(c)
		0
		1
		default
		x = a + b;
		x = a - b;
		x = a & b;
		out1 = x;
			If $ is entered for each condition in the switch statement, each condition's processing will be executed in the separate state.

4	Switch(c) {
	case 0:
	$
	x = a + b;
	break;
	case 1:
	$
	x = a - b;
	break;
	default:
	$
	x = a & b;
	}
	out1 = x;
		 switch(c)
		0
		1
		default
		x = a + b;
		out1 = x;
		x = a - b;
		out1 = x;
		x = a & b;
		out1 = x;
			If $ does not appear between a switch statement and out1 = x, then out1 = x will be executed in the same state as the last processing of a branch.

5	out1 = x;
	Switch(c) {
	case 0:
	$
	x = a + b;
	break;
	case 1:
	$
	x = a - b;
	break;
	default:
	$
	x = a & b;
	}
		 out1 = x;
		switch (c)
		0
		1
		default
		x = a + b;
		x = a - b;
		x = a & b;
			If $ does not appear before a switch statement, a conditional branch will be executed in the same state as the processing prior to the switch statement.

6	Switch(c) {
	case 0:
	$
	x = a + b;
	break;
	case 1:
	x = a - b;
	break;
	default:
	$
	x = a & b;
	}
	out1 = x;
		 switch (c)
		0
		1
		default
		x = a - b;
		x = a + b;
		out1 = x;
		x = a & b;
		out1 = x;
			If $ appears somewhere in a switch statement's conditional branch but not after the switch statement, the processing after the switch statement will be executed in a separate state for each branch.

18.4.6 wait statement and watch statement

Since processing in a wait statement and watch statement is executed in only one state (cycle), $ cannot be described in these statements.

In Example 1 below, processing B is executed repeatedly until the wait condition c becomes true. Example 2 shows a case where the operation of the Example 1 is described using if and goto statements.

During manual scheduling mode, the execution of processing A occurs during the same state as the first execution of processing B, and processing C is executed during the same state as processing B once condition c becomes true.

The wait statement and watch statement have the same meaning in manual scheduling mode.

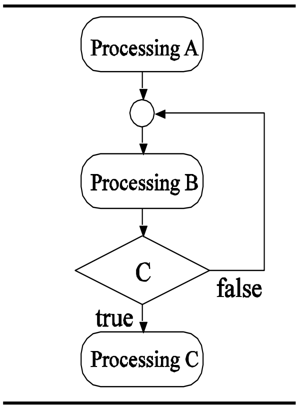
(Example 1)

```
Processing A
wait( c ) {
    Processing B
}
Processing C
$
```

→

(Example 2)

```
    Processing A
L1:
    Processing B
    if ( !c ) {
        goto L1;
    }
    Processing C
    $
```



18.4.7 goto statement, goto0 statement, label

When a goto statement or goto0 statement jumps to a label during a certain state, processing prior to that label is not executed.

A specific example is shown in **Figure 3**. Although a++ and b++ are executed during the same state, when a goto statement or goto0 statement causes a jump to LABEL1:, b++ is executed but a++ is not executed. In other words, in **Figure 3** a++ is executed only the first time.

```
    a = 0;
    b = 0;
    $
    a++;
LABEL1:
    b++;
    $
    goto  LABEL1;
```

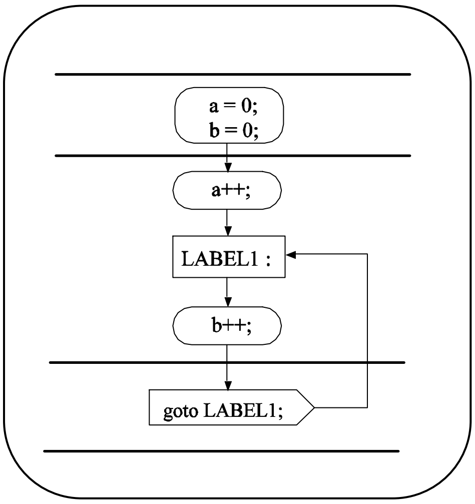


Figure 3 Description with Label During State

The goto statement itself implies the meaning of the clock boundary specifier $. Consequently, as is shown in **Figure 4**, when a goto statement appears during a state, the processing jumps to a different state. Specifically, in **Figure 4**, if the flag value is "true" a++ and c++ are executed during different states. If the flag value is "false" a++, b++, and c++ are executed during the same state.

```
    a = 0;
    b = 0;
    c = 0;
    $
    a++;
    if( flag ){
        goto LABEL2:
    }
    b++;
LABEL2:
    c++
    $
```

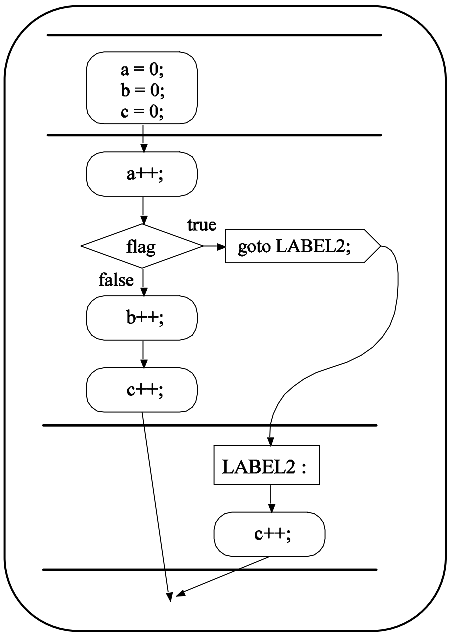


Figure 4 Description of goto Statement During State

Further, goto0 statement operate as goto statement without including the meaning of clock boundary specifier $. In **Figure 5**, when flag is true, a++ and c++ are carried out at same state. Further, when flag is false, a++, b++ and c++ are carried out at same state.

```
    a = 0;
    b = 0;
    c = 0;
    $
    a++;
    if( flag ){
        goto0 LABEL2:
    }
    b++;
LABEL2:
    c++
    $
```

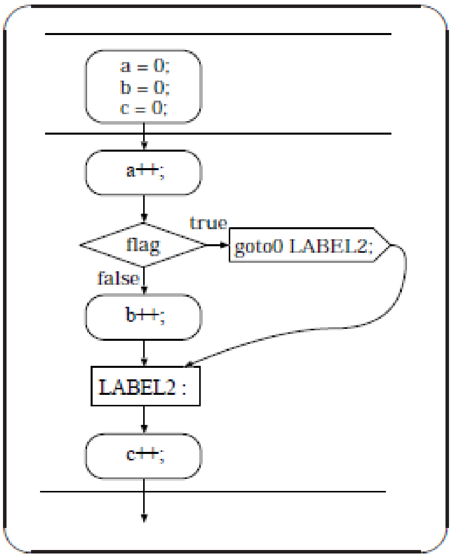


Figure 5: Description having goto0 statement in the middle of the state

18.4.8 break statement

In break statement used at the time of switch or breaking the loop, the meaning of clock boundary specifier $ is not included. Therefore, the operations before and after the execution of break are carried out in same state.

18.4.9 continue statement

The meaning of clock boundary specifier $ is not included in continue statement that suspends the loop in while, for, do while. Therefore, the operations before and after the execution of continue statement are carried out in same state.

18.4.10 Function call, return statement

The meaning of clock boundary is not retained when function is start by performing the function call and when return statement is used or function is completed and returned to calling source after reaching till the end of the function. In the other words, in case there is not even a single $ present inside the function, the processing of calling a function till its return is carried out in same state.
The two descriptions given below are different only in terms of using or not using the function, operationally they are same.

```
process foo1(){                        process foo2(){

    $                                    $
    Processing A                         Processing A
    a = function1(b);          =         a = b + 1;
    Processing B                         Processing B
    $                                    $

}                                      }
var(0:8)
function1(var(0:8) i){
   return(i + 1);
}
```

19. ATTRIBUTES

A group or line comment that starts with the special character string "Cyber" or “cyber” indicates an attribute. Application of attributes is one way to specify the synthesis method to be used by the behavioral synthesis system. For detailed descriptions of the various attributes, refer to the Cyber Behavioral Synthesis System Reference Manual. The description rules for attributes are described below.

19.1 Syntax of Attributes

Syntax

/* Cyber attribute name = attribute value */
/* Cyber attribute name */
// Cyber attribute name = attribute value
// Cyber attribute name

Description rules

1. Enter "Cyber" at the start of the comment.

- There is no distinction between upper case and lower case. For example, even if "cyber" is specified in the beginning then also it would be considered as an attribute.

- Spaces, tabs, and line feed (carriage return) characters can be described between "/*" or "//" and "Cyber".

- At least one space or tab character is required after "Cyber".

- If "Cyber" does not appear at the start of the comment, the comment is handled as an ordinary comment (not an attribute).

2. Multiple attributes can be described if they are separated by commas using group comment.

```
/* Cyber name1 = val1, name2 = val2 */
/* Cyber name3 = val3 */
```

3. Line feed characters can be described in attributes (but the line feed operation is ignored).

```
/* Cyber attri_name1 = val1,
         attri_name2 = val2 */
```

4. An attribute for a statement must appear immediately before the statement.

- Attaching an attribute to a function definition

```
/* Cyber func = inline */
var(0:8) func( var(0:8) i ){

}
```

- Attaching an attribute to a for statement

```
/*Cyber unroll_times = all */
for (i=0;i<5;i++){

}
```

5. An attribute for an item must appear immediately after the item.
   - Attaching an attribute to variable a

     ```
     int a /* Cyber share_name = REG01 */;
     ```

   - Attaching an attribute to variable b (an initial value)

     ```
     out reg(0:8) b /* Cyber valid_sig = b_i */ = 3;
     ```

   - Attaching an attribute to array variable r (an initial value)

     ```
     int r[4] /* Cyber array = REG */ = { 3,2,1,0 };
     ```

   - Attaching an attribute to a function declaration

     ```
     var(0:8) funcA(var(0:8) b )/* Cyber func = inline */;
     ```

   - Attaching an attribute to operator +

     ```
     y = a + /* Cyber ope2fu = add08 */b;
     ```

   - Attaching an attribute to array variable m[ ]

     ```
     m[i] /* Cyber mu_port = 1 */ = j;
     ```

   - Adding a separate member of structured variable

     Example: Specifying attribute only in s1.a

     ```
     struct ST1 {
         int a;
         int b;
     };
     ```

6. The attribute for a block is described immediately before the block.

   ```
   /* Cyber scheduling_block */
   {
       a = b;
       c = d;
   }
   ```

7. It is possible to change the effect of attribute by macro substitution of comment. In the below example value of array attribute of array variable m becomes RAM.

   ```
   #define ARRAY_ENTITY RAM
   int m[16] /* Cyber array = ARRAY_ENTITY */;
   ```

8. When a calculation expression is enclosed by $(), the specified range can be calculated and it can be replaced with constant value so that the calculation result can be treated as an attribute value after describing calculation expression of constant in attribute value.

   ```
   #define FOO 4

   out ter out1 /* Cyber latency_set = $(FOO + 1) */;
   out ter out2 /* Cyber latency_set = $(FOO + 2) */;
   ```

   ↓

   ```
   out ter out1 /* Cyber latency_set = 5 */;
   out ter out2 /* Cyber latency_set = 6 */;
   ```

9. It must my noted that rather than a comment attribute is treated as one statement, so if { } is not added then structure error will take place.

(Incorrect)

```
/*Cyber name1 = val1 */
if (c1)
    /* Cyber name2 = val2 */
    if (c2)
        a = b;
```

(Correct)

```
/*Cyber name1 = val1 */
if (c1) {
    /* Cyber name2 = val2 */
    if (c2) {
        a = b;
    }
}
```

* It is recommended to put { } even in 1 statement.

A. SOURCE FILES

A.1 Source File Name Extensions

Under the Cyber behavioral synthesis system, the filename extension used for input files that are described in BDL must be either ".bdl" or ".c".

A.2 Compiling Multiple Files

The Cyber behavioral synthesis system does not support reading of the multiple BDL source files. When descriptions are comprised of multiple source files, use #include to include all of the files into one source file for Cyber input.

B. EXPRESSION SIGNS AND BIT WIDTH DECISION RULES

B.1 Arithmetic Operation

When operands used in an arithmetic operation have different sign types and bit widths, they are automatically adjusted to the same sign type and bit width before the operations are executed (except for certain types of operations). These adjustments are made first for the sign type and then for the bit width, as described below.

1.When operands have different sign types, set the signed type for all of them. When an unsigned-type operand is converted to a signed-type operand, a "0" is set to the most significant bit. As a result, the bit width is increased by one bit.
2.After the sign has been adjusted, if the operands have different bit widths, set the largest bit width being used to all operands. When adjusting the bitwidth, sign extension (MSB extension) is executed for the signed type and zero expansion is executed for the unsigned type.
The sign type and bit width in the operation's results are defined for each operation based on the operands' adjusted sign type and bit width.
In the following tables, u(0:n) indicates an unsigned-type n-bit width expression, and "s(0:m)" indicates a signed-type m-bit width expression.

B.1.1 Arithmetic addition

The arithmetic addition (+) is applied from left to right. The operation occurs after the operands have been adjusted for sign type and bit width in that order. When both operands are of unsigned type, the sign of the result is of unsigned type. If the sign type of either is signed, the resulting sign type is signed. The bit width of the return value is determined by adding one bit as a carry bit to the bit width of the sign-type-adjusted.

OP1 + OP2	Type of return value	Bit width of return value
u(0:n) + u(0:m)	unsigned	*max*(*n*, *m*) + 1
s(0:n) + s(0:m)	signed	*max*(*n*, *m*) + 1
s(0:n) + u(0:m)	signed	*max*(*n*, *m* + 1) + 1

B.1.2 Arithmetic subtraction

The arithmetic subtraction (-) is applied from left to right. The operation occurs after the operands have been adjusted for sign type and bit width in that order. The sign type of the return value is always signed. The bit width of the return value is determined by adding one bit as a carry bit to the bit width of the sign type-adjusted operand.

OP1 - OP2	Type of return value	Bit width of return value
u(0:n) - u(0:m)	signed	*max*(*n*, *m*) + 1
s(0:n) - s(0:m)	signed	*max*(*n*, *m*) + 1
s(0:n) - u(0:m)	signed	*max*(*n*, *m* + 1) + 1

B.1.3 Unary minus

The result of a unary minus is a sign inverted value of the operand and its value is always signed. When the result of a unary minus operation is a signed value, its sign type is an inversion of the sign type of the operand. A negative 0 (-0) equals to 0. However, the sign type of the operand is left unchanged if the unary minus operation is executed an even number of times. If the unary minus operation is executed an odd number of times, the sign type of the operand is the same as when the unary minus operation is executed only once.

- OP	Type of return value	Bit width of return value
- u(0:n)	signed	*n* + 1
- s(0:n)	signed	*n* + 1

B.1.4 Arithmetic multiplication

The arithmetic multiplication (*) is applied from left to right. The operation occurs only after the operands have been adjusted for the sign type. When the sign types of both operands are unsigned, the resulting sign type is unsigned. If either sign type is signed, the resulting sign type is signed. The bit width of the return value is determined by taking the sum of the sign type-adjusted operands' bit widths.

OP1 * OP2	Type of return value	Bit width of return value
u(0:n) * u(0:m)	unsigned	*n* + *m*
s(0:n) * s(0:m)	signed	*n* + *m*
s(0:n) * u(0:m)	signed	*n* + (*m* + 1)

B.1.5 Arithmetic division

The arithmetic division (/) is applied from left to right. The operation occurs only after the operands have been adjusted for sign type. When the sign types of both operands are unsigned, the resulting sign type is unsigned. If either sign type is signed, the resulting sign type is signed. When the right operand is unsigned, the resulting bit width is the same as the left operand’s bit width. If the right operand is signed, the resulting bit width is decided by adding one bit to the left operand

OP1 / OP2	Type of return value	Bit width of return value
u(0:n) / u(0:m)	unsigned	*n*
s(0:n) / s(0:m)	signed	*n* + 1
s(0:n) / u(0:m)	signed	*n*
u(0:n) / s(0:m)	signed	*n* + 1

If both operands are nonnegative values, the result is nonnegative. Otherwise, the positive/negative decision depends on the specification of the divider used in RTL.

B.1.6 Arithmetic remainder

The arithmetic remainder (%) is applied from left to right. The operation occurs only after the operands have been adjusted for the sign type. When the sign types of both operands are unsigned, the resulting sign type is unsigned. If either sign type is signed, the resulting sign type is signed. The bit width of the return value is the bit width of the sign type-adjusted right operand.

OP1 % OP2	Type of return value	Bit width of return value
u(0:n) % u(0:m)	unsigned	*m*
s(0:n) % s(0:m)	signed	*m*
s(0:n) % u(0:m)	signed	*m* + 1
u(0:n) % s(0:m)	signed	*m*

If both operands are nonnegative value(s), the result is nonnegative. Otherwise, the positive/negative decision depends on the specification of the remainder used on RTL.

B.1.7 Increment operation

The sign type and bit width of the increment operation is the same as the sign type and bit width of the operand.

	Type of return value	Bit width of return value
++u(0:n)	unsigned	*n*
u(0:n)++	unsigned	*n*
++s(0:n)	signed	*n*
s(0:n)++	signed	*n*

B.1.8 Decrement operation

The sign type and bit width of the decrement operation is the same as the sign type and bit width of the operand.

	Type of return value	Bit width of return value
--u(0:n)	unsigned	*n*
u(0:n)--	unsigned	*n*
--s(0:n)	signed	*n*
s(0:n)--	signed	*n*

B.2 Comparison Operations

The compare operations include inequality operation (<, >, <=, and =>), not equal and equal operation (=, !=). Like arithmetic operations, if the signed type and bit width of the comparison operation's operands differ, the operation occurs after the operands have been adjusted for sign type and bit width in that order. The value returned from a compare operation has unsigned type and single bit width, regardless of the operands' type.

OP1 ⊙ P2	Type of return value	Input bit width of comparator
u(0:n) ⊙ u(0:m)	unsigned	*max(n, m)*
s(0:n) ⊙ s(0:m)	unsigned	*max(n, m)*
s(0:n) ⊙ u(0:m)	unsigned	*max*(*n*, *m* + 1)

B.3 Shift Operations

During shift operations, although the operands have different sign types and/or bit widths, they are not adjusted. The return value's sign type matches that of the left operand. The return value's bit width is calculated from both the operands’ bit widths.

B.3.1. Left shift operation

The result of the left shift operation is the left operand (treated as a bit string) that has been left-shifted by the number of bits specified by the right operand. The right side is then filled with zeros.

If the right operand is a signed negative value, the result value is undefined and depends on the behavior of the shifter being used in RTL.

OP1 << OP2	Type of return value	Bit width of return value
u(0:n) << u(0:m)	unsigned	$n + (2^m - 1)$

s(0:n) << s(0:m)	signed	$n + (2^{m-1} - 1)$
s(0:n) << u(0:m)	signed	$n + (2^{m} - 1)$
u(0:n) << s(0:m)	unsigned	$n + (2^{m-1} - 1)$

B.3.2. Right shift operation

In a right shift operation, if the left operand is signed an arithmetic shift is executed (i.e. the empty bits are filled with the MSB of the left operand), and if the left operand is unsigned a logical shift is executed (the empty bits are filled with zeros).
If the right operand is a signed negative value, the result value is undefined and depends on the behavior of the shifter being used in RTL.

OP1 >> OP2	Type of return value	Bit width of return value
u(0:n) >> u(0:m)	unsigned	*n*
s(0:n) >> s(0:m)	signed	*n*
s(0:n) >> u(0:m)	signed	*n*
u(0:n) >> s(0:m)	unsigned	*n*

B.4 Concatenation operation (::)

The sign of concatenation operation (::) is unsigned irrespective of operand sign. The bit width of return value corresponds to the sum of bit width of operands.

B.5 Logic Operations

Logic operations include logical AND (&), logical OR (|), and logical exclusive OR (^) operations. If the logical operation's operands have different sign types and/or bit widths, their bit widths are adjusted (except in some cases) before the operation is executed. Sign types are not adjusted.

- When the operands have different bit widths, the larger bit width is used. When adjusting the bit width, sign extension (MSB extension) is performed for signed type and zero extension is performed for unsigned type.

The operation result's sign type is unsigned and its bit width is the adjusted bit width taken from the operands.

OP1 ⊙ OP2	Type of return value	Bit width of return value
u(0:n) ⊙ u(0:m)	unsigned	*max*(*m*, *n*)
s(0:n) ⊙ s(0:m)	unsigned	*max*(*m*, *n*)
s(0:n) ⊙ u(0:m)	unsigned	*max*(*m*, *n*)
u(0:n) ⊙ s(0:m)	unsigned	*max*(*m*, *n*)

B.6 Logical Relational Operations

Logical relational operations include logical negation (!), logical AND (&&), and logical OR (||) operations. The return value is always unsigned type with one bit, regardless of the operands.

B.7 Complement Operation

This includes only the complement operation (~). All bits are inverted and returned, regardless of the operand types. The return value's sign type is unsigned. The return value's bit width is same as the operand's bit width. If the operand's bit width is undefined, the return value's bit width is also undefined.

B.8 Reduction Operation

The reduction operation's return value is always a single-bit unsigned value, regardless of the operand type.

B.9 Ternary Operations

If the 2nd and 3rd operands in a ternary operation have different sign types and/or bit widths, sign type and bit width of the return value is adjusted in this order. Sign type becomes unsigned when both 2nd and 3rd operand are unsigned, and becomes signed when sign types of any operand is signed. **The bit width of the return value is decided after adjusting 2nd and 3rd operands as specified below.**

OP1 ? OP2: OP3	Type of return value	Bit width of return value
x(0:p) ? u(0:n) : u(0:m)	unsigned	*max*(*n, m*)
x(0:p) ? s(0:n) : s(0:m)	signed	*max*(*n, m*)
x(0:p) ? s(0:n) : u(0:m)	signed	*max*(*n, m+1*)

Note: The specification has changed for a case where sign types of 2nd and 3rd operands differ (from version 3.6.1 of Cyber behavioral synthesis system.) In versions 3.6 and earlier versions, when sign types of 2nd and 3rd operands **differ,** then only bit width **is** adjusted and sign type of return value **is** always **unsigned**. Thus, caution is required if the sign type of 2nd and 3rd operands differ, because the sign type of the return value will be different from that of the most ANSI C language representations.

B.10 Type Conversion (Cast Operation)

Type conversions (casting) use a type name specified in parentheses before the elements of the expression to convert to the specified type. In case casting is non pointer (in case of value conversion), the result of casting is the same as that of assigning to intermediate variables having the specified type.

When var, ter or reg is used as the type name, either the start bit and bit width, or the LSB and MSB needs to be specified. However, be careful that only the bit width has significance and the start bit has no significance, even though any value other than zero can be specified as the start bit.

To avoid confusion, we recommend not using either ter or reg as the type specification. (Physical types are ignored.)

Refer to **section 7** for restrictions in case casting is a pointer.

B.11 Differences BDL vs. ANSI C language

BDL's specifications for determining the sign type and bit width differ in some ways from the ANSI C language specification. In the ANSI C language specification, types that are smaller than int (such as char) are first converted to int type before operations are executed. Since this is not done in BDL, bit precision differs somewhat between these two languages. The following are some examples of bit precision differences between ANSI C and BDL.

1. Bit extension of logic operation result

```
in char i_x;
int mid;

   mid = ~(i_x);
```

In the above example, a logic operation in BDL results in an unsigned return value, so the lower 8 bits are inverted and then zero-filled to extend to 32 bits. In ANSI C language, conversion to int type comes first, so all bits are inverted after 32-bit sign extension is performed. Therefore, while the mid MSB is always "0" in BDL, the mid MSB is the inverted value of the i_x MSB in ANSI C.

To avoid any ambiguity, it is recommended not to write a description which expects an automatic bit expansion for logical operation results.

C. LIST OF RESERVED WORDS

The reserved words in the input language (BDL) cannot be used as identifiers such as variable names, function names in input language of Cyber behavioral synthesis system. It is also recommend that reserved words in the output language for Cyber should not be used as identifiers. The reserved words of each input or output languages for Cyber are listed below.

C.1 Reserved Words in C Language

```
auto          else          long          typedef
              enum
break         extern        register      union
                            return        unsigned
case          float
char          for           short         void
const                       signed        volatile
continue      goto          sizeof
                            static        while
default       if            struct
do            int           switch
double
```

C.2 Reserved Words in BDL

All of the reserved words in C language are also reserved words in BDL. The following are reserved words that are specific to BDL and are not used as reserved words in C language.

```
alias         in            par           wait
allstates     inout         port          watch
              input         procedure     when
bool          inside        process
bus                                       HiZ
              latch
caseof                      reg
clock         mem           reset         ZERO
ctlvar        module        rise          FX_RND
                                          FX_RND_CONV
defmod        nmux          sence         FX_RND_INF
dmux                        shared        FX_RND_MIN_INF
              other                       FX_RND_ZERO
exit          out           template      FX_SAT
              output        ter           FX_SAT_SYM
fall          outside       typename      FX_SAT_ZERO
fixed                                     FX_TRN
                            unl           FX_TRN_ZERO
goto0                                     FX_WRAP
                            var           FX_WRAP_SM
                                          FX_RND
```

The following functions are ignored by the Cyber BDL input tool.

```
              printf fprintf sprintf scanf fscanf sscanf
```

C.3 Reserved Words in C++

All of the reserved words in C language are also reserved words in C++. The following are reserved words that are specific to C++ and are not used as reserved words in C language.

```
and                 false               reinterpret_cast
and_eq              friend
asm                                     static_cast
                    inline
bitand                                  template
bitor               mutable             this
bool                                    throw
                    namespace           true
catch               new                 try
class               not                 typeid
compl               not_eq              typename
const_cast
                    operator            using
delete              or
dynamic_cast        or_eq               virtual

explicit            private             wchar_t
export              protected
                    public              xor
                                        xor_eq
```

C.4 Reserved Words in VHDL

Note with caution that VHDL is not case-sensitive.

```
abs
access
after
alias
all
and
architecture
array
assert
attribute

begin
block
body
buffer
bus

case
component
configuration
constant

disconnect
downto

else
elsif
end
entity
exit

file
for
function

generate
generic
group
guarded
```

```
if
impure
in
inertial
inout
is

label
library
linkage
literal
loop

map
mod

nand
new
next
nor
not
null

of
on
open
or
others
out

package
port
postponed
procedural
procedure
process
protected
pure
```

```
range
record
reference
register
reject
rem
report
return
rol
ror
select
severity
signal
shared
sla
sll
sra
srl
subtype

then
to
transport
type

unaffected
units
until
use

variable

wait
when
while
with

xnor
xor
```

C.5 Reserved Words in Verilog-HDL

```
always          if                   reg
and             ifnone               release
assign          incdir               repeat
automatic       include              rnmos
                initial              rpmos
begin           inout                rtran
buf             input                rtranif0
bufif0          instance             rtranif1
bufif1          integer
                                     scalared
case            join                 showcancelled
casex                                signed
casez           large                small
cell            liblist              specify
cmos            library              specparam
config          localparam           strong0
                                     strong1
deassign        macromodule          supply0
default         medium               supply1
defparam        module
design                               table
disable         nand                 task
                negedge              time
edge            nmos                 tran
else            nor                  tranif0
end             noshowcancelled      tranif1
endcase         not                  tri
endconfig       notif0               tri0
endfunction     notif1               tri1
endgenerate                          triand
endmodule       or                   trior
endprimitive    output               trireg
endspecify
endtable        parameter            unsigned
endtask         pmos                 use
event           posedge
                primitive            vectored
for             pull0
force           pull1                wait
forever         pulldown             wand
fork            pullup               weak0
function        pulsestyle_onevent   weak1
                pulsestyle_ondetect  while
generate                             wire
genvar          rcmos                wor
                real
highz0          realtime             xnor
highz1                               xor
```

INDEX