Maia Reference Manual

Contents

1 INTRODUCTION

The purpose of a Maia program is to apply stimulus to an HDL module, to read data back from that module, and to determine whether or not that data has the expected value. The HDL module ('Device Under Test', or DUT) is not part of the Maia program, and is written in a language such as VHDL or Verilog. Maia generates a testbench for the DUT; the generated testbench must then be executed on an HDL simulator.

Maia requires a DUT definition in order to communicate with the HDL code. This definition is the *DUT Section*, which is described in chapter 8.

Maia communicates with the DUT by using drive statements, or 'test vectors', which are described in chapter 9, or by direct access to DUT signals. Drive statements automate the process of applying timed stimulus to the DUT, and checking the DUT outputs.

A drive statement evaluates a set of expressions which are used to drive the DUT inputs, and compares the DUT outputs against another set of expressions. Drive statements may be executed within various control flow constructs, to allow the creation of reactive testbenches. These facilities are provided by a simple imperative control language, which is described in chapters 2 through 7. These facilities are similar, and in many cases identical, to those provided by C and related languages.

1.1 Program structure

A program may be written in one of two forms. In the first, a single DUT definition is required, and the remainder of the program is made up of a list of drive statements, which are executed sequentially. No other statements are allowed. This form (*testvector-program*) is suitable only for simple tests.

In the second form (*procedural-program*), a program is composed of at least one function (the program entry point, which must be named `main`). There may optionally be a single DUT definition, and additional functions and declarations. In this form, the drive statements are executed as part of the normal program flow, inside a function.

The two examples below are complete examples of a testbench for a two-bit counter with a synchronous reset, coded in these two styles. The first is a test vector program, and tests the DUT by applying directives and constants to the DUT inputs, and comparing the DUT outputs against the expected values:

```
DUT {
     module counter(input CLK, RST; output [1:0] Q)
     create_clock CLK
     [CLK, RST] -> [Q]
}
[.C, 1] -> [0]               // sync reset
[.C, 0] -> [1]
[.C, 0] -> [2]
[.C, 0] -> [3]
[.C, 0] -> [0]               // roll-over
```

Example 1

The second example is the procedural equivalent of the test vector program. In this form, the DUT may be driven with and tested against arbitrary expressions, and drive statements can be enclosed in control and looping constructs:

```
DUT {
     module counter(input CLK, RST; output [1:0] Q)
     create_clock CLK
     [CLK, RST] -> [Q]
}

int main() {
   [.C, 1] -> [0];         // sync reset

   bit2 q = 1;             // q is a 2-bit integer, initialised to 1
   do
      [.C, 0] -> [q];      // count, with roll-over
   while(q++);
}
```

Example 2

Both programs execute 5 drive statements, and produce a log file entry stating that all 5 vectors have passed (assuming, of course, that the `counter` module has been correctly implemented):

```
(Log) (50 ns) 5 vectors executed (5 passes, 0 fails)
```

Syntax

```
maia-program :
   testvector-program
   procedural-program

testvector-program : tp-section-list

tp-section-list :
   tp-section
   tp-section-list tp-section

tp-section :
   DUT-definition
   label_opt vfile-drive-statement semicolon_opt

semicolon : ;

procedural-program : external-declaration-list

external-declaration-list:
   external-declaration
   external-declaration-list external-declaration

external-declaration :
   DUT-definition
   function-definition
   declaration
```

1.2 Syntax definition

The language grammar is presented in a simplified form throughout the text. The grammar is not exhaustive, and is not sufficient to construct a parser. Its purpose is merely to illustrate correct syntax, where the grammar is more concise or more complete than a textual description.

Terminals which are presented in a **`typewriter`** style should be entered literally. Note that terminals are not necessarily keywords (2.8); the ones that are not keywords may also appear as user-defined names, if they have the appropriate form for a name.

A number of the base operators (those shaded in Table 16) may optionally be signed and sized (4.2), or may have an alternative textual name (4.5.2). These alternatives are not listed in the grammar. In this production, for example:

```
shift-expression :
   ...
   shift-expression >> additive-expression
```

The **>>** operator is equivalent to the following set of right-shift operators:

```
a >>     b;             // 1: unsigned (logical) implicitly-sized Right Shift
a >>#    b;             // 2: signed(arithmetic) implicitly-sized RS
a >>$n   b;             // 3: unsigned n-bit RS
a >>#$n  b;             // 4: signed n-bit RS
a .SRL   b;             // 5: same as 1
a .SRA   b;             // 6: same as 2
a .SRL$n b;             // 7: same as 3
a .SRA$n b;             // 8: same as 4
```

Example 3

The `$n` form is used to represent any valid operator size. *n* must be greater than 0, and less than or equal to a compiler-determined maximum, which is at least 2^{24}.

bit_n and var_n represent a bit or var type mark which is optionally sized. If *n* is present, it must be greater than 0, and less than or equal to a compiler-determined maximum, which is at least 2^{24}.

1.2.1 Regular expressions

A number of productions are instead presented as regular expressions, for simplicity. In this case, the production name is followed by *::*, rather than *:*. The definition of a string, for example, is given as:

```
string :: "[^"\n]*"
```

A string is therefore composed of a double quote character, followed by zero or more characters which are not a double quote or a newline, followed by a second double-quote character.

A regular expression may also refer to a production, which is enclosed in braces { and }:

```
macro-name-lparen :: {pp-identifier}(
```

in this case, a *macro-name-lparen* is a *pp_identifier* which is immediately followed by a (character, with no intervening whitespace. The concept of "no intervening whitespace" cannot be represented in the regular (BNF-based) productions; the tokens in these productions may be separated by arbitrary line terminators and whitespace.

1.2.2 Line termination and whitespace

The language allows a number of UTF-8 code points to represent line terminators and whitespace. However, the preprocessor converts all line terminators to `\n` (LF, U+000A)[1], and all whitespace (with the exception of HT) to a space character (SP, U+0020)[2]. On completion of preprocessing, all line terminator and whitespace code points will appear as either LF (U+000A), SP (U+0020), or HT (U+0009). Any reference to `\n`, "newline", or "whitespace", outside the context of the preprocessor (in other words, any reference outside chapter 12), refers to the preprocessor output.

1.3 The preprocessor

The translation of a source file is carried out in two distinct stages. In the first, a *preprocessor* carries out a number of simple textual conversions on the source file. The preprocessor output is then used as input to the second stage of translation. This second stage is conventionally known as "compilation".

The preprocessor defines a *Macro Processing Language*, or MPL. The MPL provides a number of facilities, including the creation of macros with the `#define` directive (which allows an identifier to be replaced by an arbitrary sequence of characters), and the inclusion of a source file inside another source file, with the `#include` directive.

The operation of the preprocessor is logically distinct from the operation of the compiler, and is described in chapter 12. The preprocessor grammar is presented in the same form as the compiler grammar (1.2), but the two grammars are distinct. The grammars have a common definition of identifiers and constants, but do not otherwise reference each other. The use of the preprocessor is optional (12.1). However, the preprocessor provides a number of facilities beyond the MPL itself (including the checking of UTF-8 input), and the compiler is unlikely to be able to translate programs which have not been through the preprocessing stage.

[1] LF (Linefeed) is also variously known as "newline" and "NL". `\n` is an *escape sequence* which represents LF; see 2.6.1.

[2] See 12.2.3 and 12.2.4.

2 LEXICAL CONVENTIONS

2.1 File structure

A program must be compiled as a single unit. The name of the top-level source file is, by convention, given a `.tv` extension. Source files may use `#include` directives (12.3.2) to allow arbitrary file inclusion.

2.2 Character set

The source character set is UTF-8[1]. Source files may optionally be preceded by a 3-byte byte order mark (BOM[2]); the BOM is ignored if it is present.

All invalid byte sequences are rejected as errors, with the exception that the two-byte sequence 0xC0, 0x80 is treated as a null character (U+0000)[3]. In particular, CESU-8 encodings are not supported.

2.2.1 Line terminators and whitespace

The Unicode code points listed in 12.2.3 are recognised as line terminators; all line terminators are converted to a single LF character (U+000A) during preprocessing. The code points listed in 12.2.4 are recognised as whitespace. These code points, with the exception of HT (U+0009), are converted to a single SP character (U+0020) during preprocessing. On completion of preprocessing, all line terminator and whitespace characters in the source will appear as either LF, SP, or HT.

Maia is a "free-form" language, in the sense that line terminators and whitespace in the source are not generally significant, and are normally present simply for readability. A number of exceptions are listed below (the '`for all`' keyword, for example, cannot be spelt as '`forall`'). Another exception occurs when two adjacent textual tokens must be parsed as separate identifiers or keywords. In this case, they must be separated by line terminators or whitespace:

```
// function 'foo':
real2 foo(real2 x) { return x + π / 2; }

real2 foo1 ( real2
x ) {return x+π/2;}              // function 'foo1' is idential to 'foo'

real2foo2(real2 x){return x+π/2;}// error: 'real2foo2' is a single identifier
real2 foo3(real2 x){returnx+π/2;}// error: tokenised as (returnx)(+)(π)(/)(2)
```

Example 4

[1] UTF-8 is a Unicode multibyte character encoding. Characters which are not part of the ASCII subset (U+0000 through U+007F) are represented by a multi-byte sequence, with a maximum length of 4 bytes. UTF-8 is backwards-compatible with ASCII, and any source file which is valid ASCII is also valid UTF-8.

[2] Some Windows programs may add the three-byte sequence 0xEF, 0xBB, 0xBF to the start of any file saved as UTF-8. This is the UTF-8 encoding of the Unicode byte order mark, although byte order is not relevant to UTF-8.

[3] This exception is generally known as 'modified UTF-8'.

2.3 Comments

The `//` characters introduce a *line comment*. The compiler ignores everything after these characters, up to the end of the current line.

Comments may also be introduced by the `/*` characters, and terminated by the `*/` characters. This second form has the advantage that the comment may be spread over multiple lines, and is known as a *block comment*. These comments do not nest, and cannot be inserted within strings.

Some examples of comments are:

```
bit128 foo;      // this is a line comment
/* this is a
 * multi-line block comment */
```

2.4 Statement termination

Statements within functions are terminated by a semicolon.

Within a function, braces `{` and `}` are used to group declarations and statements into a compound statement, which is syntactically equivalent to a single statement. There is no terminating semicolon after the closing brace of a compound statement.

Statements within a DUT section may optionally be terminated by a semicolon, if desired. External drive statements in a *testvector-program* may similarly be terminated with a semicolon, if desired; the termination is not required in either case.

2.5 Identifiers

Identifiers may contain a set of *alphabetic* characters. The alphabetic characters are defined as `a` through `z`, `A` through `Z`, and all multibyte UTF-8 characters, with the exception of the multibyte line terminators (12.2.3), and the multibyte whitespace characters (12.2.4). The alphabetic characters are defined below as *ident-alpha*.

Legal identifiers consist of a combination of the alphabetic characters, the decimal digits `0` to `9`, and underscore (_, U+005F). The first character may not be a decimal digit.

User-defined identifiers must start with an alphabetic character, and should not be the same as a keyword. All identifiers which start with an underscore are reserved. Some of these reserved names may be legally used, and have predefined meanings, which are documented in 2.9 below. Identifiers may contain any number of characters up to a compiler-determined maximum, which is at least 2^{12}.

Syntax

```
identifier ::
   [{ident-alpha}_][{ident_alpha}_0-9]*
ident-alpha ::
   [U+0061-U+007A] | [U+0041-U+005A] |
   [U+0080-U+0084] | [U+0086-U+2027] | [U+202A-U+10FFFF]
```

2.6 Strings

Strings are arbitrary character sequences which are enclosed in double quotation marks ("). Strings cannot be continued onto a new line; it is an error if a single line of input contains an unterminated string. Within a function, adjacent strings are automatically concatenated, even across line boundaries. However, adjacent strings are never concatenated within a DUT section.

A string does not have a value, and may not be manipulated in an expression.

2.6.1 Escape sequences

A number of characters may be represented in a string using an *escape sequence*, consisting of a backslash (\, U+005C) followed by one or more characters. Escape sequences are replaced by the single character that they represent during parsing. An escape sequence may be a *simple-escape-sequence*, an *octal-escape-sequence*, or a *hexadecimal-escape-sequence*:

```
escape_sequence :
   simple-escape-sequence
   octal-escape-sequence
   hexadecimal-escape-sequence
octal-escape-sequence :
   \ octal-digit
   \ octal-digit octal-digit
   \ octal-digit octal-digit octal-digit
hexadecimal-escape-sequence :
   \x hexadecimal-digit
   hexadecimal-escape-sequence hexadecimal-digit
octal-digit :: [0-9]
hexadecimal-digit :: [0-9,a-f,A-F]
```

The simple escape sequences are listed in Table 1 below. An octal escape sequence is composed of a maximal-length sequence of octal digits (1, 2, or 3 digits), while a hexadecimal escape sequence is composed of a maximal-length sequence of hexadecimal digits (1 or 2 digits).

Newline	LF (NL)	\n	audible alert	BEL	\a
Horizontal tab	HT	\t	backslash	\	\\
vertical tab	VT	\v	question mark	?	\?
Backspace	BS	\b	single quote	'	\'
carriage return	CR	\r	double quote	"	\"
Formfeed	FF	\f			

Table 1: Simple escape sequences

Syntax

```
string ::
   "[^"\n]*"
```

2.7 Constants

Constants may represent integer, floating-point, or boolean values.

Integer constants can be specified as either a *Cinteger*, or a *Vinteger*. the Cinteger is based on C integer constants, while the Vinteger is based on Verilog integer constants. Cintegers are unsized 2-value integers (in other words, each bit can take on one of only 2 values; `0` or `1`). The Vinteger is an optionally-sized 4-value integer (each bit can take on one of the 4 values `0`, `1`, `X`, or `Z`).

Floating-point constants have the same form as C99 floating constants[1].

When used in an expression, a constant may be considered to be replaced by a temporary object, of a `bit`, `var`, or `bool` type, with the value of the constant. If an integer constant contains no metavalues, then this object is a `bit`; it is otherwise a `var`. For floating constants, this object is a `bit` (floating constants may not contain metavalues); for boolean constants, it is a `bool`.

A leading minus sign, if it is present, is not part of the constant; it is instead interpreted as a unary negation operator.

Syntax

```
constant :
   cinteger-constant
   vinteger-constant
   floating-constant
   boolean-constant
```

2.7.1 Cinteger

Cintegers are specified in a C-like form, with the addition that `0b` and `0B` prefixes may be used to specify binary data. A leading `0x` or `0X` specifies hexadecimal, while a leading zero otherwise specifies octal. Underscores may also be inserted arbitrarily into the constant to improve readability, although not as the first character, and not inside the base specifier.

Syntax

```
cinteger-binary ::          0[bB][01_]+
cinteger-octal ::           0[0-7_]*
cinteger-decimal ::         [1-9][0-9_]*
cinteger-hexadecimal ::     0[xX][a-fA-F0-9_]+

cinteger-constant :
   cinteger-binary
   cingteger-octal
   cinteger-decimal
   cinteger-hexadecimal
```

In other words, a Cinteger may be one of:

[1] ISO/IEC 9899:1999 (E), §6.4.4.2

1. A binary integer, which is prefixed by either `0b` or `0B`, and which is followed by one or more characters in the range `0` to `1`;
2. An octal integer, which starts with `0`, optionally followed by one or more characters in the range `0` to `7`;
3. A decimal integer, which starts with a character in the range `1` to `9`, optionally followed by one or more characters in the range `0` to `9`;
4. A hexadecimal integer, which is prefixed by either `0x` or `0X`, followed by one or more case-insensitive characters in the range `0` to `9`, or `A` to `F`.

These constants may not include metadata (unknown and high-impedance bits), and are scanned to the number of bits set by the `_DefaultWordSize` pragma (which defaults to 32). An overflow is reported if the constant cannot be represented in this many bits.

Some examples of Cinteger constants are:

```
bit64 i;                        // a 64-bit two-state variable
i = 0;                          // octal 0
i = 0b1010;                     // binary, equivalent to decimal 10
i = 012;                        // octal, equivalent to decimal 10
i = 10;                         // decimal 10
i = 0x000a;                     // hex, equivalent to decimal 10
i = 0x_ffff_ffff_ffff_ffff_;    // arbitrary underscores (_DefaultWordSize >= 64)
i = _0xffff_ffff;               // error:  leading '_'
i = 019;                        // error: decimal integers may not start with 0
```

Example 5

2.7.2 Vinteger

Vintegers are specified in a Verilog-like form. This allows metadata to be entered, and also allows constants of arbitrary width to be specified.

Syntax

```
size-prefix ::   [0-9]*['`]
vh-digit ::      [xXzZ?0-9a-fA-F]
vb-base ::       [bB]
vo-base ::       [oO]
vd-base ::       [dD]
vh-base ::       [hH]

vinteger-constant ::
 {size-prefix}({vb-base}|{vo-base}|{vd-base}|{vh-base}){vh-digit}({vh-digit}|_)*
```

In other words, a Vinteger constant is composed of:

- a mandatory size prefix, which is either (a) an unaccompanied *prefix character* for an unsized Vinteger, or (b) one or more decimal integers followed by a prefix character, for a sized Vinteger; followed by
- a mandatory base specifier, which must be one of binary (`b` or `B`), octal (`o` or `O`), decimal (`d` or `D`), or hex (`h` or `H`); followed by
- a single `vh_digit`; optionally followed by

- zero or more characters which are either a `vh_digit`, or an underscore character, which may be used arbitrarily to improve readability.

The prefix character may be either an *apostrophe* ' (U+0027), or a *grave accent* ` (U+0060)[1].

`vh_digit` is shown as including the full case-insensitive hex character set, for simplicity. The characters used must, however, be valid for the specified base. The `vh_digit` may also be specified (in any base) as `x` or `X` for an unknown value, or `z`, `Z`, or `?` for a high-impedance value; these 5 characters are the 'metavalues'.

A metavalue character specifies 1 bit for the binary base, 3 bits for octal, and 4 bits for hex. The integer `4'hx`, for example, is equivalent to `4'bxxxx`. Metavalues are illegal in decimal numbers, unless the entire integer is composed of a single metavalue. In this case, every bit of the integer is set to the metavalue. The integer `32'dx`, for example, contains 32 unknown bits.

Vintegers must not contain any whitespace; the entire constant is one token.

These integers are essentially identical to Verilog "based constants". There are, however, a number of differences between Vintegers and Verilog based constants:

- there must be no whitespace anywhere in the constant. "`5 'd 4`" is interpeted in Verilog as 0b00100, but is illegal in Maia; it should instead be specified as "`5'd4`"
- the 's' designator is illegal (as in Verilog-1995)
- if a size prefix is present and the constant cannot be represented in the specified size, an overflow error is reported (overflow is not an error in Verilog)
- if a size prefix is not present, the constant is scanned to the number of bits specified by `_DefaultWordSize`, and an overflow error is reported if it cannot be represented in this many bits
- if there is not enough data in the constant to fill the specified number of bits, then the data is always padded with `0` bits to the left; `x` and `z` padding is never used. The one exception is the case of the single-metavalue decimal Vinteger, as noted above.

Some examples of Vinteger constants are:

```
#pragma _DefaultWordSize 24    // scan unsized integer constants to 24 bits
var5 i;            // a 5-bit four-state variable
i =  'b1010;       // 24-bit decimal 10, truncated to 5 bits on assignment
i =  4'B00_1010;   // 4-bit decimal 10, no overflow
i =  4'B01_1010;   // cannot scan to 4 bits; overflow error
i =  'o12;         // 24-bit decimal 10, truncated on assignment
i =  'd10;         // 24-bit decimal 10, truncated on assignment
i =  4'D10;        // 4-bit decimal 10, 0-extended on assignment
i =# 4'D10;        // 4-bit decimal 10, sign-extended to 5'b11010 on assignment
i =  'dz;          // 24 z bits, truncated to 5 bits on assignment
i =  'h1x;         //  i == 5'b1xxxx
i =  4'h1x;        // overflow error
```

Example 6

[1] The apostrophe character is used for Verilog compatibility. However, this character may cause difficulties with tools for C-like languages (such as editors), and the grave accent (or 'back-tick') may be used as an alternative.

2.7.3 Floating constants

Maia floating constants are lexically identical to C99 floating constants. A float constant is composed of a *significand part*, followed by an optional *exponent part*, and an optional suffix. The suffix specifies the precision of the constant; it may be either **`f`** or **`F`** for single-precision, or **`l`** or **`L`** for extended double-precision. The constant is double-precision if the suffix is omitted.

The Verilog code generator supports only double-precision constants (14.7.1). However, any expressions which can be statically evaluated by the compiler may use any, or all, of the floating-point precisions.

The constant is hexadecimal if it is preceded by **`0x`** or **`0X`**; it is otherwise decimal. For a hexadecimal constant, the significand is interpreted as a hexadecimal number; for a decimal constant, the significand is interpreted as a decimal number.

For a hexadecimal constant, the exponent is interpreted as a decimal number, which specifies the power of two by which the significand is scaled. For a decimal constant, the exponent is interpreted as a decimal number, which specifies the power of ten by which the significand is scaled.

The components of the significand part may include a digit sequence representing the whole-number part, followed by a period (`.`), followed by a digit sequence representing the fraction part. At least one of the whole-number part and the fraction part must be present.

The components of the exponent part are an **`e`** or **`E`** (for a decimal constant), or **`p`** or **`P`** (for a hexadecimal constant), followed by an exponent consisting of an optionally signed decimal digit sequence.

For decimal floating constants, either the period or the exponent part has to be present. The exponent is always required for hexadecimal floating constants.

Syntax

```
floating-constant :
  dec-floating-constant
  hex-floating-constant

dec-floating-constant :
  dec-fractional-constant exponent10-part_opt floating-suffix_opt
  dec-digit-sequence      exponent10-part     floating-suffix_opt

hex-floating-constant :
  hex-prefix hex-fractional-constant exponent2-part floating-suffix_opt
  hex-prefix hex-digit-sequence      exponent2-part floating-suffix_opt

dec-fractional-constant :
  dec-digit-sequence_opt . dec-digit-sequence
  dec-digit-sequence   .

hex-fractional-constant :
  hex-digit-sequence_opt . hex-digit-sequence
  hex-digit-sequence   .

exponent10-part :
  e sign_opt dec-digit-sequence
```

```
    E sign_opt dec-digit-sequence

  exponent2-part :
    p sign_opt dec-digit-sequence
    P sign_opt dec-digit-sequence

  dec-digit-sequence :
    dec-digit
    dec-digit-sequence digit

  hex-digit-sequence :
    hex-digit
    hex-digit-sequence hex-digit

  sign :: + | -

  hex-prefix :: 0x | 0X

  floating-suffix :: f | F | l | L
```

Examples

Some examples of floating-point constants are given below; the comments include the report statement output. These are all double-precision constants. Single- and double-extended constants are specified identically, but with a trailing case-insensitive **F** or **L**, respectively.

```
report("'%5.2f'\n", 3.14159);       // ' 3.14'
report("'%5.2f'\n", 3.14159E0);     // ' 3.14'
report("'%5.2f'\n", 31.4159e-1);    // ' 3.14'
report("'%5.2f'\n", .314159e+1);    // ' 3.14'
report("'%5.2f'\n", 1.);            // ' 1.00'
report("'%5.2f'\n", 1e0);           // ' 1.00'
report("'%5.2f'\n", .1E1);          // ' 1.00'
report("'%5.2f'\n", .5);            // ' 0.50'
report("'%5.2f'\n", 0x0.8p0);       // ' 0.50' (0000.1000)
report("%5.2f'\n", 0x0.8p1);        // ' 1.00' (ie. 0000.1000 x 2^1)
report("%5.2f'\n", 0x0.4p2);        // ' 1.00' (ie. 0000.0100 x 2^2)
report("%5.2f'\n", 0x0.3p+4);       // ' 3.00' (ie. 0000.0011 x 2^4)
```

Example 7

2.7.4 Boolean constants

The literals `true` and `false` may be used wherever a value of a Boolean type is expected.

Note that care must be taken when displaying a boolean value with `report`. The `%d` and `%i` conversion specifications treat their argument as a signed integer quantity, and the displayed output may not be as expected[1]. Booleans should be printed with an unsigned conversion (`%b`, `%o`, `%u`, `%x`, or `%X`), or with the boolean specification (`%l`). `%l` produces the output `false` if the corresponding expression is false, and `true` otherwise.

[1] Most Verilog simulators will display a signed 1'b1 as '-1' , although at least one displays it as '0'.

Syntax

```
boolean-constant :: true | false
```

2.8 Keywords

The language keywords are listed in the tables below. These keywords may not be used as identifiers. The tables are separated for convenience; all the keywords are reserved in all parts of a program. Note that `var[0-9]*`, `kmap[0-9]*`, and `bit[0-9]*` are regular expressions, and not literal tokens. In other words, `var` itself is a keyword, and any token which is composed of `var` immediately followed by one or more decimal integers is also a keyword. The tokens of multi-word keywords (`when all` and `for all`) should be separated by whitespace.

Table 2 lists the DUT-related keywords. Note that `name`, `period`, `pipeline`, and `waveform` are listed, but have no significance unless they are immediately preceded by a hyphen character.

create_clock	create_enable	DUT	inout	input
macromodule	module	-name	negedge	output
-period	-pipeline	posedge	signal	timescale
-waveform				

Table 2: Keywords 1

The type-related keywords are listed in Table 3 below.

bit[0-9]*	bool	int	kmap[0-9]*	real1
real2	real3	stream	struct	ubit
uvar	var[0-9]*			

Table 3: Keywords 2

Table 4 lists the remaining keywords. `true` and `false` are listed as keywords for simplicity, although they are technically boolean literals.

and	assert	break	case	continue
default	do	else	exec	exit
false	for	for all	if	or
report	return	static	switch	trigger
true	void	wait	when	when all
while				

Table 4: Keywords 3

Table 5 list tokens which are reserved for future use as keywords; they may not be used as identifiers.

new	delete	enum	function	typedef
ref	in	out	goto	fork
join				

Table 5: Reserved words

Table 6 lists multi-character tokens which must not include whitespace. There are also a number of multi-character operator tokens (`*=`, for example) which may not include whitespace; these are listed in Table 16. The `meta`, `msb`, `size` and `offset` operators must be immediately preceded by either an apostrophe character, or a grave accent (backtick) character, with no intervening whitespace; the table lists only the apostrophe version, for simplicity.

```
@(              ::              ->              'meta           'msb
'size           'offset         'last
```

Table 6: Additional lexer tokens

2.9 Predefined identifiers

A number of identifiers are predefined, and are listed in Table 7 below.

Name	R/W	Scope	Function
`_errorCount`	RO	global	The global error counter; automatically incremented by assertion failures. The simulation will terminate when `_errorCount` reaches the value set by mtv's `-rte` switch (14.5). `_errorCount` may also be incremented by failures in runtime type checking (11). Note that `_errorCount` is unrelated to DUT failures (see `_failCount`).
`_assertCount`	RO	global	Total number of assertions executed; may be used with `_errorCount` for testing
`_version`	RO	global	The Maia version, as three integers packed into 32 bits. The top 12 bits are currently unused and read back as zero. The next 16 bits encode the release year, and the bottom 4 bits encode the release month.
`_timeNow`	RO	global	The current simulation time, as a bit64. The units are the timescale units defined in the DUT section, which default to nanoseconds.
`_vectorCount`	R/W	global	The total number of DUT vectors (drive statements) executed
`_passCount`	R/W	global	The DUT pass counter
`_failCount`	R/W	global	The DUT fail counter. `_vectorCount`, `_passCount`, and `_failCount` are automatically logged and displayed at the end of a simulation. Note that these variables are writeable, to allow for manual testing.
`result`	R/W	function	The return value from a function: see Section 7.

Table 7: Predefined variable names

3 CONCEPTS

3.1 Type checking

The syntactic and semantic correctness of a program is determined through a combination of *static* and *dynamic* error (or *type*) checking. Static checking is carried out during compilation; a failure at this stage is reported as a syntax error. However, various classes of error cannot be detected during compilation, and must be detected at runtime. Example 8 contains two static errors, and one dynamic error (in practice, however, errors which could be determined statically may instead be reported at runtime):

```
void test(void) {
  var12 x[3];         // an array of three 12-bit quantities
  x[3] = 0;           // static error: only x[0], x[1], and x[2] are valid
  x[2].(15:12) = 0;   // static error: x[2] only has 12 bits
  x[indx()] = 0;      // dynamic error: '3' is not a valid index
}

int indx(void) { return 3; }
```

Example 8

Dynamic error checking is described in (11) below.

The level of static checking which is carried out is determined by the user, by setting the `_StrictChecking` pragma. This has three possible values (0, 1, and 2), where the level corresponds to the level of 'strictness' of the checking. Level 0 defines a minimal level of 'weak' checking, which is generally associated with scripting languages. The default level is 1, which gives a level of checking which is approximately equivalent to that found in C and similar languages. A program which compiles without error at a given checking level is guaranteed to compile without error at any lower level.

It is generally possible to write more compact (and possibly more understandable) code with level 0. However, this disables a number of checks which might otherwise identify erroneous code, and should generally be considered to be suitable only for simple, and relatively short, tests.

The type checking level can be set only with the `_StrictChecking` and `_Implicits` pragmas. These pragmas are program-wide, and should appear once in the source code, before any functions are analysed. There are no corresponding compiler switches to set the level. The intention is to ensure that a given program will always compile with the same level of checking, irrespective of the manner in which it is compiled.

The level-specific checks and features are listed in Table 8 below, together with the corresponding `_StrictChecking` level:

Level:	0	1	2
Implicit variables (3.1.1)	Y	N	N
Default type for function formal (3.1.2)	Y	N	N
Default type for function return (3.1.3)	Y	Y	N
Port size checking (3.1.4)	N	Y	Y
Extended boolean checking (3.1.5)	N	N	Y
Separate boolean type (3.1.5)	N	N	Y

Table 8: level-specific checking

3.1.1 Implicit variables

Level 0 allows the use of implicitly-declared variables. These variables do not need to be declared, and are instead created automatically when an undeclared variable is first assigned to in the code. An error will be reported if an undeclared variable is read before it is written to. These variables are created with type `var`; in other words, they are 4-state data objects, with a default size[1]. The object must be scalar; implicit arrays will not be created.

```
#pragma _StrictChecking 0
main() {
  for(i=0; i<4; i++)
    report("i is %d\n");     // Ok
  report(j is %d\n", j);     // error; 'j' has not been written to
}
```

Example 9

Implicit variables can be enabled or disabled independently of the checking level by using the `_Implicits` pragma. If this pragma is used, the requested action takes precedence over the action implied from the `_StrictChecking` level. `#pragma _Implicits 0` disables the use of implicits, while `#pragma _Implicits 1` enables implicits.

Implicit variables which are created in a function have function scope, rather than block scope. The scope of the variable starts at the point at which it is first assigned to, and ends at the end of the associated function.

3.1.2 Function formal types

If a function formal parameter has no type specifier in level 0, a type of `uvar` (an unconstrained `var`) will be assumed. A syntax error is reported at any higher level.

[1] If implicits are enabled, creation of a `var` object creates an object with a size given by `_DefaultWordSize`. When implicits are not enabled, however, `var` is simply a synonym for `var1`, and an object declared as a `var` is a 1-bit object.

3.1.3 Function return types

If a function is declared with no type specifier in levels 0 and 1, a return type of `uvar` (an unconstrained `var`) will be assumed. A syntax error is reported at any higher level.

3.1.4 Port size checking

When reading from, or writing to, an object of an arithmetic type, the source expression will normally be extended or truncated as required, irrespective of the checking level. However, this is a potential source of errors when driving or reading DUT ports, and extension and truncation are therefore disabled for expressions which appear in drive statements, when the checking level is greater than 0. Note that:

1. When the checking level is 0, objects may be extended or truncated as required, with no restrictions;
2. When the checking level is greater than 0, extension and truncation are disabled and will result in a syntax error, with the exceptions noted in 3.1.4.1 below;
3. The exceptions of 3.1.4.1 apply *only* when using a drive statement. If the DUT ports are driven or read directly then they do not apply.

3.1.4.1 Port size checking exceptions

- If an unsized constant is used in a drive statement (on either the left-hand or the right-hand side) then that constant is not subject to port size checking, irrespective of the checking level.
- A bitslice of an object has the same size as that object, and this can complicate the use of bitslices to drive DUT ports. A port may therefore be driven directly if the slice has constant indexes, and the slice width is the same as the port width.

Various examples of valid and invalid drive statements are shown in Example 10 below.

```
DUT {
   module adder(input[15:0] A, B, output[15:0] C);
   [A,B] -> [C];
}

main() {
   bit32 in1 = 10;
   bit16 in2 = 6;
   int   x=30, y=0;

   [32`d10,           in2] -> [];    // error: sized const, not 16 bits
   [in1,              in2] -> [];    // error: in1 is 32-bit
   [in1.(14:0),       in2] -> [];    // error: 15 bit slice
   [in1.(15:0).(3:0), in2] -> [];    // error: 4 bit slice
   [in1.(15:0).(x:y), in2] -> [];    // error: not constant
   A = in1.(18:3);                   // error: 16 bit <- 32 bit
   [in1.(18:3),       in2] -> [7];   // but this is Ok
   [in1.(x:y).(17:2), in2] -> [8];   // Ok: 16 bits
   ['d10,             in2] -> [16];  // Ok: unsized const
}
```

Example 10

3.1.5 Boolean type

In levels 0 and 1, there is no separate boolean type, and `bool` is simply a synonym for `bit1` (a one-bit two-state integer). The boolean values `true` and `false` have the values `1'b1` and `1'b0`, respectively.

Level 2 introduces a separate boolean type. In level 2, an arithmetic object may still be used wherever a boolean is required by the syntax, and the object will be implicitly converted to a boolean. However, booleans undergo some additional type checking; two boolean objects may not be added by using the `+` operator, for example.

```
int foo() { return 1; }

while(true)             // Ok for all levels; true and false are not level-specific
   ...
if(foo())               // Ok for all levels
   ...
if(foo() != 0)          // Ok for all levels
   ...

bool x = false, y = true;
int z = x + y;          // Ok in levels 0,1; error in level 2
```

Example 11

3.2 Declaration order

In general, external declarations and function definitions may appear in the source code in any order. Local declarations, however, are treated conventionally, and must appear in the code before they are used. This flexibility is possible because a compiler pre-pass analyses external declarations, the DUT section, and all functions.

The specific rules are listed below; some of these are simply a reformulation of the scope requirements of (3.3).

1. There is no requirement for functions to be declared; a function's definition also serves as its declaration
2. There is no requirement that a function definition should precede any use of that function in the source code. The functions which make up a program may therefore appear in the source code in any order
3. The DUT definition is treated in the same way as a function, and may appear anywhere that a function may appear
4. There is no requirement that external variable declarations should precede any use of that variable in the source code
5. There is no requirement that external type declarations should precede any use of that type in the source code, with the exception noted in (6) below

6. If an external structure declaration contains a member which is of a user-defined type (a stream or another structure), then that type must have already been analysed

7. Automatic and static variables in a function must be declared before they are used. If, however, implicit variables are enabled, and the compiler encounters a write operation to an unknown variable, then it will implicitly declare it to be of type `var`

8. local type declarations (for structures and streams) in a function must precede any use of that type in the function

9. In a function, declarations may appear at any location; it is not necessary for declarations to appear at the beginning of the function.

This example program demonstrates all these requirements:

```
main() {
  b.a.x = 10;    // 4: external variable 'b' may be used before it is declared
  struct s2 c;   // 5: external type 'struct s2' may be used before it is declared
  foo();         // 2: 'foo' may be called before it is defined
}

foo() {          // 1: no declaration is required for 'foo'; this is the definition
  struct s3 {    // 8: 'struct s3' must be declared before it is used
    int z;
  } c;

  c.z = 20;      // 7: 'c' must be declared before it is used
  int d = 30;    // 9: declarations may occur anywhere inside a function
}

DUT {}           // 3: the DUT defn may appear anywhere where a function may appear

struct s1 { int x, y; };

struct s2 {
  struct s1 a;   // 6: the declaration of 's1' must appear before its use here
} b;             // 4: the declaration of 'b' may appear after its use in 'main'
```

Example 12

3.3 Scope

An identifier can denote an object, a function, a tag or a member of a structure or stream, a label name, a macro name, or a macro parameter. Macro names and parameters are expanded before translation, and so are not considered further here. The same identifier can denote different entities at different points in the source code.

An identifier that denotes a given entity is *visible* only within a specific region of the source code, known as its *scope*. The various entities denoted by an identifier have different scopes, or are in different namespaces. There are three kinds of scope: function, block, and global.

Any identifier which is a function name or is declared outside a function has *global scope*. These identifiers include DUT port and signal names, and drive declaration label names (8.3.7). These identifiers are visible throughout the source code that makes up the program, with one exception. This

exception occurs when a structure or stream tag is used in another structure (in other words, when the second structure includes an instance of the first structure or stream). In this case, the first tag is considered to have a scope that starts at its introduction for the purposes of inclusion in any other structure; it is visible throughout the entire source code for any other purposes (this is the exclusion listed in item (6) of section 3.2).

Every other identifier is introduced in a function parameter list, or within a function. The identifier has *function scope* if it introduces a new implicit variable (3.1.1), and otherwise has *block scope*. The scope of an implicit starts at the point at which it is first assigned to, and ends at the end of the associated function. The scope otherwise starts at the point of introduction and terminates at the end of the associated block.

An identifier at a given scope level (the *outer scope*) may be *hidden* by the same identifier in an enclosed scope (the *inner scope*). An entity with global scope may always be accessed by using the global scope operator[1], `::`. The example below shows a number of cases where an outer scope identifier is hidden, and prints '4321':

```
int i = 1;                    // global scope
main() {                      // block scope level 1
  int i = 2;
  do {                        // block scope level 2
    int i = 3;
    {                         // block scope level 3
      int i = 4;
      report("%d", i);        // 4
    }
    report("%d", i);          // 3
  } while(false);
  report("%d", i);            // 2
  report("%d\n", ::i);        // 1 (global scope operator)
}
```

Example 13

3.4 Namespaces

Under some circumstances, an identifier may potentially refer to more than one entity at a given point in the source code. This is possible where the surrounding syntactic context allows the use of the identifier to be disambiguated, and is formalised in the concept of a *namespace*. A namespace is therefore a context for an identifier. The possible namespaces are:

- drive statement label names, which are disambiguated by the syntax of their declaration and use;
- the tags of structures and streams, which are disambiguated by following the keywords `struct` or `stream`. The tags of structures and streams share the same namespace;
- the members of structures of streams, which are disambiguated by following the `.` operator. Each structure and stream creates its own namespace;
- all other identifiers.

[1] `::` is not, strictly speaking, an operator, although it may be considered to be a prefix operator in most circumstances.

3.5 Storage duration

An object has a *storage duration* that determines its *lifetime*. There are two storage durations: static, and automatic. The lifetime of an object is the portion of program execution during which storage is guaranteed to be reserved for it. An object exists, and retains its last-stored value, throughout its lifetime.

An external object, or one declared with the `static` modifier, has *static storage duration*. Its lifetime is the entire execution of the program and its stored value is initialized only once, prior to program startup. If no initialization is specified for the object, a default initialisation (3.6) is carried out.

Any other object has *automatic storage duration*. The lifetime of such an object extends from the point of its declaration, until the enclosing scope terminates. If an initialisation is specified for the object, it is performed each time the declaration is reached in the execution of the function; otherwise, a default initialisation (3.6) is carried out each time the declaration is reached.

3.6 Default initialisation

All data and boolean objects (of both static and automatic storage duration) which are not explicitly initialised are given a default initialisation. 4-state objects are initialised to all `X`, while 2-state objects are initialised to all 0. Booleans are initialised to `false`.

Stream objects are always automatically initialised; the "default" initialisation can be considered to be the initial value of the stream.

All objects are either data, boolean, or stream objects, or an aggregate of these basic objects. An aggregate is default-initialised by initialising all basic objects within the aggregate. If a basic object within an aggregate has no explicit initialiser, then that basic object is default-initialised.

3.7 Types

3.7.1 Introduction

Every expression has a *type* which is known at compile time. The type of the expression determines the operations that can be carried out on that expression, and the values which can be stored in, or read from, that expression.

At its simplest level, an expression is simply an identifier which has been declared as an object or as a function name. In this case, the type of the identifier determines the values which may be stored in or read from the object, or the values which can be returned from the function.

There are three *data* types (`int`, `bit`, and `var`, together with a number of specialisations), which are appropriate for representing 'data' items. The remaining types are a boolean type (`bool`); a stream type (`stream`); a structure type (`struct`); and an array type.

Some objects have no value, and so have no type; these objects are said to be of a `void` type. A function may be declared to be of type `void` when it is not required to return a value.

3.7.1.1 Data types

The `int`, `bit`, and `var` types are intended to store numeric 'data'. `int` and `bit` represent 2-state (binary) data, while `var` represents 4-state data. Objects of these three types are collectively known as *ivar* objects.

There are additionally a number of specialisations of the `bit` and `var` types. `kmap` is a specialisation of `var`, which simplifies the handling and manipulation of data representing Karnaugh maps. `kmap` and `var` support a different set of operators, and so are considered to be distinct types. The terms arithmetic type and data type are used to make this distinction; 'arithmetic' excludes `kmap`, while 'data' includes it.

`ubit` is an unconstrained `bit`, while `uvar` is an unconstrained `var`. These two specialisations are used to represent objects whose size is not known in advance. However, the underlying object is a `bit` or `var` object (albeit of an unknown size), so these specialisations do not introduce new types.

`real1`, `real2`, and `real3` are provided to simplify the handling of floating-point data. These are not additional types, but are simply synonyms for a `bit` which is correctly sized to hold IEC single, double, and extended double-precision data. On most systems, `real1` is identical to `bit32`; `real2` is identical to `bit64`; and `real3` is identical to either `bit80` or `bit128`.

`bool` is a synonym for `bit1` at levels 0 and 1 (3.1.5), and so can be considered to be a data type.

3.7.1.2 Non-data types

The structure type is a user-defined aggregate type; it is used to encapsulate a collection of objects which are potentially of different types (a *heterogeneous* collection). The array type is equivalent to the structure type, but encapsulates a collection of objects which are of the same type (a *homogeneous* collection).

The `stream` type represents files on the host operating system, and handles file input and output operations. The use of a dedicated stream type allows common vector file operations to be handled simply, without the use of an external I/O library.

`bool` is a distinct non-data type at Level 2 (3.1.5). There is no string type; the contents of a string cannot be manipulated.

3.7.1.3 Data object indexing

Data objects are always indexed in a descending fashion. The most significant bit of the object has an index value of one less than the size of the object, while the least significant bit has index value 0. The value of the most significant bit is returned by the `'msb` operator:

```
bit4 c = 4'b1000;
assert(c'size == 4);
assert(c'msb  == c.(c'size-1));
assert(c'msb  == c.(3:3));
assert(c'msb  == 1);
```

Example 14

3.7.1.4 int properties

The `int` type represents 'small' two's complement integers. `int` objects are signed and have a size which is given by the _DefaultWordSize pragma, which itself defaults to 32 bits. `int` is provided as a programmer convenience (for indexing and looping operations, for example), and is not intended for modelling hardware.

3.7.1.5 bit and var properties

`bit` and `var` objects have no properties apart from their size; they do not, for example, have "signed", "unsigned", "integer", or "floating-point" properties. A `bit` or `var` data object may contain any data pattern, including data which can be interpreted as a floating-point number.

When using `bit` and `var` objects, complexity is provided by *operators*, rather than by the type itself. There are, for example, different operators for signed and unsigned integer comparisons, and integer and floating-point addition. This behaviour reflects the structure of the electronic systems that Maia is intended to model and verify. In these systems, data is a secondary concern, and simply represents the contents of a storage location. Electronic systems are primarily concerned with the *transformation* of data, in function units. The same storage location may be connected to, for example, an unsigned integer comparator, a signed integer comparator, or a floating-point adder, at different times.

Maia is therefore fundamentally different from general-purpose object-oriented languages in which data is the primary concern, and in which complexity is provided by layering properties on top of the data (in, for example, classes).

3.7.2 Assignment compatibility

Objects *a* and *b* are assignment-compatible if the expression *a=b* is allowable. In most cases, assignment-compatibility is commutative; in other words, if *a=b* is a valid expression, then so is *b=a*. Any exceptions are listed below.

Arrays *a* and *b* are assignment-compatible if both arrays are of the same rank, each rank has the same bounds, and the array elements are both assignment-compatible and have the same size (3.7.12.4).

Scalars *a* and *b* are assignment-compatible in the following circumstances:

1. *a* and *b* are both `ivar` objects
2. *a* and *b* are both `bool` objects
3. *a* is an `ivar` object, and *b* is a `bool` object
4. *a* and *b* are `kmap` objects with the same number of variables
5. *a* and *b* are `struct` objects where both are instances of the same structure definition
6. *a* and *b* are `stream` objects where both are instances of the same stream definition
7. If *a* is a mode 1 stream, and *b* is an object of an `int` or `bit` type, then the assignment *a=b* (but not *b=a*) is allowable

In all other cases, *a* and *b* are not assignment-compatible.

3.7.3 ubit and uvar

A 2-state data object whose size is not known in advance should be declared as a `ubit`. Similarly, a 4-state data object whose size is not known in advance should be declared as a `uvar`. `ubit` and `uvar` may appear only as a function formal parameter, or as a function return type.

When a formal parameter is of an unconstrained type, its actual size may be retrieved with the `'size` attribute. Alternatively, a 'for all' loop will automatically loop over all values of the actual.

When a function returns an unconstrained object, that object will be sized to whatever was returned by the function. Different paths through the function may return a 10-bit or a 12-bit object, for example, on different calls[1].

This example returns true if the unconstrained input has odd parity, and false otherwise:

```
bool oddParity(ubit a) {
   result = false;
   for(int i = 0; i < a'size; i++)
      result ^= a.(i);
}
```

Example 15

This example byte-reverses an arbitrary byte-wide input:

```
ubit reverse(ubit data) {
   int nbytes = data'size/8;
   int src    = data'size-1;
   int dst    = 7;

   result = data;                  // set the return value size
   for(int i=0; i<nbytes; i++) {
      result.(dst:dst-7) = data.(src:src-7);
      dst += 8;
      src -= 8;
   }
}
```

Example 16

The compiler must statically determine the maximum possible size of an unconstrained formal or return value. There are some circumstances in which this may be difficult or impossible; this might happen, for example, in a circular chain of function calls in which there is a cycle of connected unconstrained return values and unconstrained actuals. In these circumstances it might be possible to complete sizing by increasing the number of sizing iterations performed by the compiler; see (14.4).

3.7.4 ivar operations

`int`, `bit` and `var` (*ivar*) objects have a number of semantic similarities, which are described in this subclause.

[1] The returned object is actually statically sized to the maximum size returned by any call of the function; this value may be found by applying the 'size attribute to the function call. In practice, the return size can almost always be considered to be dynamically set in the current function call.

3.7.4.1 Assignment compatibility

ivar objects are assignment-compatible. When a 2-state ivar object is assigned to a 4-state ivar object the destination bits will have the same value as the source bits. When a 4-state ivar object is assigned to a 2-state ivar object any metavalue bits in the source are converted to 1[1]. Under most circumstances[2], an ivar object may be assigned to a narrower ivar object, in which case the source data is truncated. Similarly, an ivar object may be assigned to a wider ivar object, in which case the source data is either zero-extended, or sign-extended, depending on which assignment operator is used:

```
bit3 a = 3'b101;
bit4 b, c;
var5 d = 5'b1xz01;

b =  a;   assert(b == 4`b0101);    // the '=' operator zero-extends
c =# a;   assert(c == 4`b1101);    // the '=#' operator sign-extends
b =  d;   assert(b == 4`b1101);    // truncate, and convert metavals to 1
```

Example 17

3.7.4.2 Using an ivar object as a boolean

An ivar object may be used in any context in which a boolean is expected. In this case, an all-zero value is equivalent to `false`, while any other value is equivalent to `true`[3].

3.7.4.3 Supported operators

The `bit` and `var` types support all the operators listed in Table 16, with the exception of `()` and `[]` (which are defined only for functions and arrays), and `'offset` (which is defined only for arrays, structures, and streams).

The `int` type supports the same operators, with the exception that the sized and signed versions of the operators may not be used where any of the operands are of type `int`. The 'plain' versions of the operators will correctly return a signed result.

The `'size` and `'meta` operators return a `bit` and a `bool`, respectively, for any ivar operand (0). The remaining unary operators have a return type which is the same as the type of the operand.

3.7.4.4 Binary operations

If the operands of a binary operator are both of an ivar type, then:

1. If both operands are of type `var`, then the operation is carried out using 4-state arithmetic or logic, and the result is of type `var`;
2. If one operand is of type `int` or `bit` and the other is of type `var`, then the `int` or `bit` is converted to a temporary `var` as if by assignment, the operation is carried out using 4-state arithmetic or logic, and the result is of type `var`;

[1] This is consistent with a definition of 'false' as 0, and 'true' as non-zero.

[2] There is an exception if DUT port size checking is enabled; see (3.1.4).

[3] This behaviour differs from Verilog. in Verilog, an expression is 'true' if it is non-zero *and* does not contain metavalues, and is 'false' otherwise. In Maia, an expression is false if it is zero, and is true otherwise. The expression `2'b1x` is therefore false in Verilog, and true in Maia.

3. If both operands are of type `int` or `bit`, the operation is carried out using conventional 2-state arithmetic or logic. If either operand is of type `bit`, the result is of type `bit`; otherwise, the result is of type `int`.

3.7.5 int

The `int` type is primarily intended for general 'software' operations which require a signed type, where the size of that type is not a specific concern. `int` is signed, and has a size given by the `_DefaultWordSize` pragma (12.7). `_DefaultWordSize` itself defaults to 32 bits if it has not been set.

3.7.6 bit

The `bit` type represents two-state data, where each bit can take on one of the values `0` or `1`. The size of a `bit` object must be set explicitly, as a decimal integer suffix which immediately follows the `bit` keyword, with no intervening whitespace. The suffix may be omitted for a single-bit object (in other words, `bit x` is equivalent to `bit1 x`).

```
bit a;      // 'a' is a one-bit two-state data object
bit18 b;    // 'b' is an 18-bit two-state data object (indexed as 17:0)
bit192 c;   // 'c' is a 192-bit two-state data object (indexed as 191:0)
```

Example 18

3.7.7 var

The var type represents four-state data, where each bit can take on one of the values `0`, `1`, `X`, or `Z`. `X` and `Z` are *metavalues*, and represent "unknown" and "tristate", respectively, in electronic systems. The `'meta` postfix operator may be used to determine whether or not an expression contains any metavalues. `expr'meta` will return `true` if `expr` contains any metavalue bits, and `false` otherwise.

`var` is essentially identical to `bit`, except that operations on `var` objects are carried out using 4-state arithmetic and logic, as defined in (3.7.7.3) through (3.7.7.6) below.

3.7.7.1 var declaration

var objects may be explicitly declared in exactly the same way as bit objects (3.7.6). var objects are additionally created in these circumstances (where 'level' is the `_StrictChecking` level):

1. Any ports or signals declared in a DUT section are implicitly declared as a correctly-sized external `var`
2. A rank-zero K-map (3.7.8) is a one-bit var (a `var1`)
3. At level 0, implicitly-declared variables are created as a default-sized var (undeclared object `x`, for example, is implicitly declared as `var`*`nn`* `x`, where *nn* is equal to _DefaultWordSize)
4. At level 0, a function formal parameter which does not have a type specifier is an unconstrained object of type var (a `uvar`)
5. At levels 0 and 1, a function which has no return type specifier returns an unconstrained object of type var (a `uvar`).

3.7.7.2 var operations

The 4-state arithmetic and logic operations are defined in (3.7.7.3) through (3.7.7.6) below. For the operators which are not listed in these subclauses, the 4-state behaviour is as follows:

1. the shift and rotate operators preserve any metavalues;
2. the logical AND and logical OR operators will convert 4-state operands into boolean values according to **Error! Reference source not found. Error! Reference source not found.**;
3. the conditional operator will convert its first expression into a boolean according to **Error! Reference source not found.**;
4. an assignment to a 4-state object preserves any metavalues;
5. the remaining operators have the same behaviour for `bit` and `var` operands.

3.7.7.3 4-state arithmetic

If a 4-state operand of an arithmetic operator (unary **+**, **-**, **++**, and **--**, and binary **+**, **-**, *****, **/**, and **%**) contains any metavalues, then the result of that operation will be all `X`[1]:

```
bit3 a = 1;
var3 b = 3'b011;
var3 c = 3'bx11;
assert(a + b == 3`b100);
assert(a + c == 3`bxxx);
```

Example 19

3.7.7.4 4-state comparisons

If a 4-state operand of a relational operator (**>**, **>=**, **<**, and **<=**) contains any metavalues, then the result of that operation will be `false`:

```
bit3 a = 4;
var3 b = 3`b011;
var3 c = 3`bx11;
assert(!(a < b) && !(a <= b) &&  (a > b) &&  (a>= b));
assert(!(a < c) && !(a <= c) && !(a > c) && !(a>= c));
```

Example 20

3.7.7.5 4-state equality

The 4-state equality operators (**==** and **!=**) include any metavalue bits in the comparison:

```
bit3 a = 4;
var3 b = 3`b100;
var3 c = 3`b10x;
assert(a == b && b != c);
```

Example 21

3.7.7.6 4-state logic

[1] This is same as the corresponding Verilog behaviour.

Table 9 below defines the results of the 4-state bitwise (**&**, **|**, **^**, and **~**) operators[1]. The corresponding tables for the 2-state operators can be found by simply ignoring the (shaded) `X` and `Z` rows and columns.

&	**0**	**1**	**X**	**Z**
0	0	0	0	0
1	0	1	X	X
X	0	X	X	X
Z	0	X	X	X

|	**0**	**1**	**X**	**Z**
0	0	1	X	X
1	1	1	1	1
X	X	1	X	X
Z	X	1	X	X

^	**0**	**1**	**X**	**Z**
0	0	1	X	X
1	1	0	X	X
X	X	X	X	X
Z	X	X	X	X

~	
0	1
1	0
X	X
Z	X

Table 9: 4-state logic operations

3.7.8 kmap

The `kmap` type is a specialisation of `var`, and is used to simplify the specification and testing of combinatorial functions of several variables. An *n*-variable `kmap` is essentially a multi-dimensional `var` array of rank *n*, in which the element addressing has been modified into a reflected-binary Gray-coded form. For example, Figure 1 below shows a 5-variable Karnaugh map:

DE \ ABC	000	001	011	010	110	111	101	100
00	1	1	1	0	1	0	1	1
01	0	1	0	1	0	1	0	1
11	1	0	1	0	1	0	1	0
10	0	1	1	1	1	1	0	1

Figure 1: 5-variable Karnaugh map

The diagram is shown in a standard form, and shows the required output of a logic function of 5 variables: `fn(A,B,C,D,E)`. These inputs are encoded in a 5-bit binary word, with `A` being the most significant input (with a weighting of 2^4) and `E` being the least significant bit (with a weighting of 2^0). This logic function is coded in Maia as follows:

```
// declare a 5-variable Karnaugh map
kmap fn =
      1 1 1 0   1 0 1 1
      0 1 0 1   0 1 0 1
      1 0 1 0   1 0 1 0
      0 1 1 1   1 1 0 1;

// examples of K-map usage, using Algol-style indexing:
assert(fn[0,0,0,0,0] == 1);   // top-left element:     ABCDE = 00000 =  0
assert(fn[1,0,0,0,0] == 1);   // top-right element:    ABCDE = 10000 = 16
assert(fn[0,0,0,1,0] == 0);   // bottom-left element:  ABCDE = 00010 =  2
assert(fn[1,0,0,1,0] == 1);   // bottom-right element: ABCDE = 10010 = 18
```

Example 22

When accessing individual elements of a `kmap` – in other words, the function output for a given set of inputs – it is important to understand the way in which the inputs are coded. To derive the coding for an *n*-variable K-map (which therefore contains 2^n elements), the map should be drawn as a square with *n* rows and *n* columns (if *n* is even), or a rectangle with *n*-1 rows and 2(*n*-1) columns (if *n* is odd). The example above shows a 5-variable function coded in 4 rows and 8 columns. The columns and rows should then be encoded using reflected-binary Gray addressing, as shown, starting at the top left.

The indexes into a `kmap` must be binary (in other words, an index variable may not contain a metavalue). The output, however, is a 4-state value, which must be coded as a one-bit literal constant (and not a constant expression). The metavalues `1'bx` and `1'bz` may alternatively be specified as a case-insensitive `X` or `Z` for simplicity[1].

K-map initialiser lists may optionally be enclosed in braces, in the same way that scalar initialisers may optionally be brace-enclosed. These two initialisers are identical:

```
kmap a = {
  0 X 1 Z
  1 0 0 1
};
kmap b = 0 X 1 Z 1 0 0 1;
```

Example 23

The precise format of the initialiser list is not important. Each element must be separated by whitespace, but newlines are not significant. The second initialiser contains a list of 8 elements, so must encode a 3-variable kmap, with 2 rows and 4 columns.

K-map objects may be declared either as a `kmap`, or as a `kmap`*n*, where *n* is the number of variables in the K-map. Objects declared as a `kmap` must be completely initialised in their declaration to allow the compiler to derive the number of variables. Objects declared as a `kmap`*n* may be left uninitialised if desired, in which case they are given a default value of all `X`:

```
kmap  a = 0 1 1 0;      // inferred as a 2-variable kmap
kmap2 b = 0 1 1 0;      // declared as a 2-variable kmap
kmap2 c;                // initialised to {X X X X}
c = b;                  // c now contains {0 1 1 0}
kmap  d;                // error: must be initialised
kmap  e = 0 1 1 0 1;    // error: initialisation must be complete
kmap3 f = 0 1 1 0 1;    // Ok: initialised to {0 1 1 0 1 X X X}
```

Example 24

A single element extracted from a K-map using an indexing operation is of type var (a `var1`). It is not possible to extract anything other than a single element (a row or column, for example) from a K-map.

K-map objects may be passed to and returned from functions in the normal way.

3.7.8.1 Assignment compatibility

K-maps are assignment-compatible only with other K-maps with the same number of variables. It is not possible to assign a K-map to, or from, a `var` array with the same underlying dimensionality.

[1] This is the only place in which the literals `X` and `Z` may be used as constants; these literals will result in a syntax error if used in any other context.

3.7.8.2 K-map operations

The operators which support K-map operands are listed in Table 10 below. The logic operators use 4-state logic (3.7.7.6).

Operator	result type	Operation
`x'size`	`bit`	Returns the number of elements in `x` (2^n, where *n* is the number of variables)
`x'meta`	`bool`	Return `true` if `x` contains any metavalues, and `false` otherwise
`~x`	`kmap`	Bitwise negation; inverts the entire K-map
`x = y`	`kmap`	Assign K-map `y` to K-map `x`
`x == y`	`bool`	Test `x` and `y` for equality
`x != y`	`bool`	Test `x` and `y` for inequality
`& | ^`	`kmap`	Bitwise and, or, and xor
`&= |= ^=`	`kmap`	Bitwise compound assignment
`(e1)?x:y`	`kmap`	Returns K-map `x` if expression `e1` evaluates true, and K-map `y` otherwise
`(..., x)`	`kmap`	The comma operator; returns K-map `x` if `x` is the last expression

Table 10: kmap operators

3.7.9 bool

An object which has been declared to be of type `bool` has only two potential values: *false*, and *true*. The specific behaviour of booleans depends on the level of the `_StrictChecking` pragma (12.7). At levels 0 and 1 the bool type has no special significance, and no boolean-related type checking is carried out.

The syntax requires a boolean as the controlling expression for the `if`, `do`, `while`, `for`, and `assert` statements, as an operand of the logical operators, and as the first operand of the conditional operator. An expression that evaluates to an `int`, `bit` or a `var` may instead be used in these contexts, and is implicitly converted to a boolean according to (3.7.4.2).

3.7.9.1 Levels 0, 1

`bool` is not a distinct type at levels 0 and 1; an object declared as a `bool` is simply a one-bit bit (a `bit1`). The `false` literal is defined as `1'b0`, while the `true` literal is defined as `1'b1`.

An ivar object may additionally be used anywhere where a boolean is required by the syntax. If the object has an all-zero value, then it is considered to have a value of false; it otherwise has the value true.

3.7.9.2 Level 2

`bool` is a distinct type at level 2, and supports only the operators listed in Table 11 below. A boolean object is assignment-compatible only with another boolean, or with an ivar object.

The binary logical and equality operators listed are defined if both operands are boolean, or if one is boolean and the other is an ivar:

Operator	result type	Operation
`x'size`	`bit`	Returns 1
`x'meta`	`bool`	Returns false
`!x`	`bool`	Logical negation
`=`	`bool`	Assignment
`== !=`	`bool`	Equality
`&& ||`	`bool`	Logical AND, OR
`(e1)?x:y`	`bool`	Conditional operator; `x` and `y` must both be boolean
`(..., x)`	`bool`	Comma operator; returns boolean `x` if `x` is the last expression

Table 11: boolean operators

As for levels 0 and 1, an ivar object may alternatively be used anywhere where a boolean is required by the syntax.

3.7.10 struct

A structure is a collection of one or more objects, possibly of different types, into a single named object. Structures are defined and declared conventionally, and may contain scalar or array objects of any type. Structure elements are accessed conventionally, using a dotted identifier notation.

Structures and streams are, in many respects, syntactically identical. The discussion of structure definition and declaration below is equally applicable to streams. Structures and stream tags occupy the same namespace (3.4); it is illegal to give a structure and a stream the same name when they are in the same scope.

Structures may be passed to and returned from functions in the normal way. Note that the keyword `struct` (or `stream`) must be used when declaring formal parameters and function return types[1]:

```
struct s1 {...};
// function 'foo' has a single structure parameter and returns a structure
s1 foo(s1 param) {...}                  // error; must be...
struct s1 foo(struct s1 param) {...}    // Ok
```

Example 25

3.7.10.1 Declaration and definition

A structure *declaration* creates a new structure *type*, which is either named or anonymous. A structure *definition* creates a scalar or array *object* of a given structure type. The code below shows various examples of the definition and declaration of structures:

[1] The keywords are required in C, but not in C++.

```
// create named structure type 's1'
struct s1 { int x,y; };                          // declaration of 'struct s1'

// create two objects: 'a' is of type 'struct s1', 'b' is of type 'array[4]
// of struct s1'
struct s1 a, b[4];                               // definition of a, b

// create objects 'c' and 'd'; both are of type 'array[3] of struct s1'
struct s1[3] c, d;                               // definition of c, d

// create an anonymous struct type, and objects 'e' and 'f' of that type
struct { int x,y; } e, f;                        // definition of e, f

// create named struct type s3, and objects 'g' and 'h' of that type
struct s3 { int x,y; } g;                        // definition of g
struct s3 h;                                     // definition of h
```

Example 26

Note that the declaration of both stream and structure types must be terminated with a '`;`' character, even when there is no trailing object list (as in the first example above). The semicolon is required because the object list is optional; without it, the parser would find it difficult to distinguish between a type declaration with a trailing object list, or a type declaration with no object list, immediately followed by a new statement.

Structures may contain declarations of other structures. The name of the nested structure is placed in the same scope as the structure in which it is nested. This code is therefore legal[1]:

```
struct S { struct T {...}; };
struct T x;
```

Example 27

3.7.10.2 Structure assignment compatibility

Structures are assignment compatible only if they are of the same type:

```
struct s1 { int x, y; } a;
struct s2 { int x, y; } b;
struct s1 c;
a = c;               // Ok; a and c are assignment-compatible
c = b;               // Error; c and b are not assignment-compatible
bool d = a == b;     // Error; a and b are not assignment-compatible
```

Example 28

3.7.10.3 Structure operations

The operators which support structure operands are listed in Table 12 below; `x` and `y` must be of the same type. Note that an object of type 'array of struct' is not a structure.

The `'offset` operator may also be applied to any member in a structure to find that member's offset within the structure, in bits.

[1] This again follows C practice; the code is illegal in C++.

Two assignment-compatible structures may be tested for equality and inequality; the structures are equal if every member contains the same data. For a member which is a stream, the stream identifiers (or 'handles') are tested for equality; the identifiers will compare equal if they refer to the same stream.

Operator	result type	Operation
`x'size`	`bit`	Return the size of structure `x`, in bits
`x'meta`	`bool`	Return `true` if `x` contains any metavalues, and `false` otherwise
`x.m`	any	Return member `m` in `x`
`x = y`	`struct`	Assign struct `y` to struct `x`
`x == y`	`bool`	Test `x` and `y` for equality
`x != y`	`bool`	Test `x` and `y` for inequality
`(e1)?x:y`	`struct`	Returns structure `x` if expression `e1` evaluates true, and structure `y` otherwise
`(..., x)`	`struct`	The comma operator; returns structure `x` if `x` is the last expression

Table 12: structure operators

3.7.10.4 Limitations

Structures may contain other structure objects, but not references to those objects; this means that linked lists of structures cannot be built.

3.7.11 stream

Stream objects handle file read and write operations. `stream` is not a single type; it is instead a family of types which are specialised for specific file operations. Two stream types are provided: mode 1 streams provide random read access into text data files, while mode 2 streams provide sequential write access to text data files. These two stream types do not provide generalised file I/O; they are instead specialised to allow the trivial creation and reading of data files which contain whitespace-separated data fields.

The syntax of stream definitions and declarations is essentially identical to the equivalent structure syntax (see 3.7.10.1). In a structure, members are explicitly declared with a type and a name. In a stream, by contrast, there are no explicit members; the compiler automatically creates members using the information found in the stream's *format* property. These members can then be accessed in exactly the same way as structure members, using a dotted notation. Mode 1 and 2 stream members correspond to data fields within text files.

Objects of a stream type may be viewed as either *handles* to the underlying stream or, alternatively, as references to that stream; both views are equivalent. The term handle is generally used here for clarity; this does not imply that the stream is represented by a small integer. There are no operators which provide access to the underlying implementation of a stream (a stream handle cannot, for example, be read into an integer)[1].

[1] However, the space occupied by a stream within a structure can be found with the `'offset` operator, and will be consistent with the size of a small integer.

The representation of a stream as a handle implies that streams are effectively passed to functions by reference. In other words, if a stream is passed to a function, that function sees exactly the same stream as the caller; if the function changes the state of the stream (by reading from it or writing to it) then the caller will see the new state of the stream when the function returns.

Mode 1 and 2 streams provide no operators for opening and closing the stream; these operations are carried out automatically. Global and static streams are opened at the start of program execution, and remain open throughout the lifetime of the program; local streams (those declared within a function) are opened when the definition is encountered, and are closed when the function returns. If it is necessary for a local stream to retain state between function calls then that stream should be declared as static.

3.7.11.1 Mode 1 streams

Mode 1 streams are read-only streams that may be accessed randomly. Each line of the corresponding text file contains a set of data fields which are described by a format specification. Each line of the file must therefore have the same format. The file should contain only 2-value data; mode 1 streams cannot be used to read data that contains `X` and `Z` metavalues. The stream is analysed and processed during compilation and an error is reported if any data fields are found to contain invalid data. The file may contain arbitrary comments and whitespace, and any data that can be parsed as a constant (see (2.7) and (2.7.3)), although the preceding size and base prefix can be omitted with an appropriate format string (3.7.11.1.4).

Mode 1 streams provide no operations to open or close the associated text file, or to explicitly read either entire lines or individual fields; these operations are handled automatically. The stream is positioned to a given line in the (processed) file by assigning an integer to the stream. This operation automatically reads the data fields in that line of the file. It is guaranteed that the stream will always point to the first line of the file when it is first used, and that the data fields will contain the corresponding data from the first line of the file.

3.7.11.1.1 Mode 1 pre-processing

The data file is located and processed during compilation. The compiler confirms that the file exists; it then removes all comments and superfluous whitespace from a temporary version of the file, and confirms that each line of the resulting file is appropriate for the format. An error is issued if any data contains metavalues, or if any significant data bits have to be truncated to match a format specification. A warning is issued if any sized data item must be extended to match a format specification.

The resulting temporary file contains exactly one line for each set of data inputs in the source file. The number of lines in this temporary file is defined as the 'size' of a mode 1 file. In the description below, a *line* refers to this processed set of data inputs, which contains one or more data *fields*; *offset* refers to the zero-based line number in the temporary file; and *size* refers to the number of lines in the temporary file. The original file is not modified by pre-processing.

3.7.11.1.2 Mode 1 file positioning

A mode 1 stream is set to a given offset simply by writing an integer to the stream; this integer is the required zero-based offset in the processed file. Note that the offset of a line in the original and the processed files will *not*, in general, be the same. A specific data item may occur on, for example, line 20 of the input file, but might appear on line 10 of the processed file after stripping comments and

whitespace. A mode 1 offset is then essentially a 'meta'-offset which refers only to the significant lines in the input file.

Any integer written to the stream is reduced modulo the size of the file; this means that it is not possible to have a file positioning error. A file offset therefore has the same behaviour as a 'small' `bit` or `var` object. The value of a 4-bit `bit`, for example, wraps around from 15 to 0 when incremented. The offset of a file which contains 100 lines wraps around from 99 to 0 when incremented, or from 0 to 99 when decremented, in exactly the same way.

The current offset in a file can be retrieved using the `'offset` operator. The addition and subtraction operators are overloaded when one of the operands is a mode 1 stream and the other is an `int` or `bit`. In this case, the operator reads the current file offset and returns the sum or difference of the offset and the second operand. This behaviour can be used to step arbitrarily through the file. Some examples of file positioning are given in the code below; this code assumes that the file contains at least 15 lines.

```
stream s1 a;
a = 0;           // file rewind (automatic when the stream is first used)
--a;             // set to the last line in the file
assert(a'offset == a'size-1);
a = 10;          // set to offset 10 (line 11)
a = 4 + a;       // set to offset 14 (line 15)
a -= 3;          // set to offset 11
a++;             // set to offset 12
```

Example 29

Any file positioning operation automatically reads the data fields at the new offset.

3.7.11.1.3 Mode 1 stream declaration

A mode 1 stream type must be declared with three properties: a *mode*, a *file*, and a *format*. The mode must be the integer `1`. All three properties must be present in the declaration, and may appear in any order. An example of a mode 1 stream declaration is:

```
stream s1 {
   mode   1;
   file   "vectors/testfile.dat";
   format "%8'i %64'h %f", f1, f2, f3;
} a;
```

Example 30

This declaration creates a new type of `stream s1`, and a new object `a` of this type. The corresponding text file is searched for in directories which are relative to the location of the source file containing the declaration. In this case, the compiler expects to find a directory named `vectors` in the same directory as this source file, and expects to find a file named `testfile.dat` in that directory. The file may be searched for in an absolute location by appropriately prefixing the filename (with '`/`' or '`C:\`', for example).

3.7.11.1.4 Mode 1 format property

The format property specifies the expected contents of each line of the input file. It is composed of a string containing text which should be matched in the input, and one or more *conversion specifications*, followed by one or more field names. Each conversion specification requires a corresponding field

name; the first conversion specification is associated with the first field name, and so on. There must be exactly the same number of conversion specifications as field names.

For the example above, a line is expected to start with an 8-bit integer, followed by whitespace, followed by a 64-bit hex integer, followed by whitespace, followed by a floating-point value. Any subsequent fields on the line are ignored. If it was necessary to read only the first 8-bit integer on this line, then the format string

```
format "%8'i", f1;
```

would be sufficient.

The format string must include exactly one name for each field. In the example above, the 8-bit integer field is named `f1`, the 64-bit integer field is named `f2`, and the float field is named `f3`. These are user-supplied names for automatically-created *read-only* members within the stream object, which contain the current value of the corresponding field.

The field data can be read using the same dotted notation used for structures. In this case, the current values of the 3 fields can be read as `a.f1`, `a.f2`, and `a.f3`. These values are automatically updated when the stream offset changes, to give the field values at the new offset. The fields may be considered to be read-only objects of a `bit` type, with a declared size given by the field width.

Any text in the format string which is not a conversion specification, and which is not whitespace, must be matched exactly.

3.7.11.1.5 Mode 1 conversions

A conversion specification is composed of the `%` character, optionally followed by a field width, followed by a conversion character. If the field width is present, it must be a decimal integer, which must be followed by an apostrophe or a grave accent (back-tick) character. A conversion specification may also be specified as `%%`, when it is necessary to match a single `%` character in the input. The conversion characters supported for mode 1 are listed below.

`f` Matches a floating-point constant in the format defined in (2.7.3). If a field width is present, it must be `0`, `1`, or `2`, for single, double, or extended double precision, respectively; the width defaults to `2` if it is not present. If the constant itself includes a precision suffix then that suffix must match the field width[1].

`i` Matches any integer constant in the format defined in (2.7). If the constant does not have a size specification, it is assumed to have the size specified by `_DefaultWordSize`. In normal use, no field width is specified, and the width is derived from the size of the input data.

`h d o b` Matches integer data in base 16, 10, 8, or 2, respectively. The input should contain only underscore characters and characters which are appropriate for the selected base; it must not contain any prefix characters. A field width is mandatory for these conversions.

[1] mtv's Verilog code generator supports only double-precision float data; an error will be reported if either the field width or the suffix do not have the appropriate values for double-precision.

For all conversions, the file data may optionally be preceded by a '–' character. For the integer conversions, the result is derived by taking the two's complement of the input data, for the appropriate field width.

For the **h**, **d**, **o**, and **b** conversions, a field width must be provided, and the data in the input file may not contain a size prefix. An error will be reported if any of the data items corresponding to this field cannot be represented in the specified width.

For the **i** conversion, a field width will not normally be provided, and the field width is found from the input data. In order for the compiler to derive the field width it is necessary for all the data items corresponding to this field to have the same size; an error is reported if this is not the case.

For the **i** conversion, a field width may be provided if necessary. In this case, an error will be reported if any data items cannot be represented in the field width, and a warning will be reported if any explicitly-sized data items must be extended to reach the field width.

3.7.11.1.6 Mode 1 stream example

Consider, for example, this input file:

```
/* comments
 */

vector 1 data 10  // comment
vector /* */ 2 data 20
vector 3 data 30
// comment

/* */ vector 4 /* */ data 42
```

Example 31

In this case, the processed file contains 4 lines, has a size of 4, and can be accessed with offsets of 0, 1, 2, and 3. This program is sufficient to read the entire file (assuming that the file is named 'test.dat', and is in the same directory as the program) and to display all eight values:

```
/* this program produces the output:
offset 0: vector is 1; data is 10
offset 1: vector is 2; data is 20
offset 2: vector is 3; data is 30
offset 3: vector is 4; data is 42
*/
stream s1 {
  mode   1;
  file   "test.dat";
  format "vector %i data %i", f1, f2;
} a;

main() {
  for(int i=0; i<a'size; i++) {
    a = i;            // set the file offset
    report("offset %d: vector is %d; data is %d\n", a'offset, a.f1, a.f2);
  }
}
```

Example 32

3.7.11.1.7 Mode 1 'for all' operation

A convenient way to iterate through all the lines in a mode 1 file is to use the `for all` statement. The example above can be more compactly coded as:

```
main() {
  for all a
    report("offset %d: vector is %d; data is %d\n", a'offset, a.f1, a.f2);
}
```

Example 33

The `for all` statement as used here is defined to be equivalent to:

```
main() {
  a = 0;                    // automatic rewind
  do {
    report("offset %d: vector is %d; data is %d\n", a'offset, a.f1, a.f2);
  } while((++a)'offset);    // relies on offset wrap-around
}
```

Example 34

3.7.11.1.8 Mode 1 stream assignment compatibility

Only mode 1 streams of the same type are assignment-compatible. This definition of assignment compatibility is the same as the corresponding one for structures; see 3.7.10.2.

3.7.11.1.9 Mode 1 stream operators

The operators which may be applied to mode 1 streams are listed in Table 13 below. When a stream is read in an expression the value returned is the stream, except in one specific case: the addition and subtraction operators are overloaded to read the current offset in the stream, rather than the stream itself.

The first column in the table shows the operator. In this column, `x`, `y`, and `z` are expressions which evaluate to the same stream type (and so are assignment-compatible), and `i` is an expression which evaluates to a `bit` type. `x`, `y`, and `z` may be any expression that evaluates to a scalar stream. Note that an object of type 'array of stream' is not a stream.

Operator	result type	Operation
x'size	bit	Return the number of lines in stream x
x'offset	bit	Return the current offset in stream x
x++	stream	Increment the offset in x, and return the old state of x
x--	stream	Decrement the offset in x, and return the old state of x
x.f	bit	Return the value of field f in x
++x	stream	Increment the offset in x, and return the new state of x
--x	stream	Decrement the offset in x, and return the new state of x
x+i, i+x	bit	Read the stream offset from x, carry out the addition, and return the bit result
x-i, i-x	bit	Read the stream offset from x, carry out the subtraction, and return the bit result
x = y	stream	Assign stream y to stream x, and return stream x
x = i	stream	Set the offset of stream x to i, and return stream x
x += i	stream	Identical to (x = x+i), and so returns stream x
x -= i	stream	Identical to (x = x-i), and so returns stream x
(e1)?y:z;	stream	The ternary operator; y and z must be assignment-compatible. Returns stream y if e1 evaluates true, and stream z otherwise
(..., x)	stream	The comma operator; returns stream x if x is the last expression

Table 13: mode 1 stream operators

3.7.11.2 Mode 2 streams

Mode 2 streams are write-only streams that must be accessed sequentially. Each line of the output text file contains a set of data fields which are described by a format specification; each line of the file must therefore have the same format. It is only possible to write 2-value data; a mode 2 stream cannot be used to write data that contains X and Z metavalues.

The stream is written by assigning data to any fields defined by its format specification (3.7.11.2.2), and then applying the pre-increment operator to the stream object. The increment operation writes the current line, and increases the stream size by one. The output file is created when the object is declared; if it already exists, it will be over-written. The first increment operation therefore writes the first line of the file.

If a data field is not assigned to before incrementing the stream, that field will retain its last value. The initial value of a field is undefined. An error is reported if a given field is never written to; however, no error is reported if a field is written on some occasions, but not others.

Mode 2 streams are associated with a single write-only output file, and it is therefore not possible for multiple stream objects to have their own private copy of this file. It therefore makes little sense to

declare multiple objects of the same mode 2 stream type. Where this does happen, the objects are defined to be synonyms to, or references for, each other:

```
// a and b are defined to be the same object, which writes to a single output file:
stream m2 {
  mode 2;
  ...
} a, b;

// but c and d are different objects, which read from their own private copies of
// an input file:
stream m1 {
  mode 1;
  ...
} c, d;
```

Example 35

3.7.11.2.1 Mode 2 stream declaration

A mode 2 stream is defined in the same way as a mode 1 stream (3.7.11.1.3), except that the mode must be the integer 2. The specified file is opened for writing; if the file already exists, it is over-written.

3.7.11.2.2 Mode 2 format property

The format property specifies the required form of each line in the output file. It has the same form as a mode 1 format (3.7.11.1.4), and is made up of a string containing whitespace[1], text which is copied to the output line, and conversion specifications; the string is followed by a list of field names.

3.7.11.2.3 Mode 2 conversions

The conversion specification is the same as the `report` statement conversion specification (6.13). The conversion specifiers supported for mode 2 are:

`f e E g G` Floating-point output

`x d o b` Hexadecimal, decimal, octal, and binary output, respectively

3.7.11.2.4 Mode 2 stream fields

Mode 2 stream fields are write-only objects with the properties of a `bit`, and can be assigned to from any object which is assignment-compatible with an `bit`. The size of the field is determined statically by the compiler, by examining the size of all objects which are assigned to the field[2]. It is an error if a given field is assigned to from multiple objects which do not have the same size.

[1] Whitespace in the format string is collapsed into a single space in the output line in 2019.9.

[2] For the report statement, the arguments are single expressions with a known size; for mode 2 format specifications, the arguments (fields) are simply names, and their size must be determined from assignments to the fields.

```
stream m2 {
  mode   2;
  file   "foo";
  format "%x %6.3f", field1, field2;
} a;
int4 b;
var5 c;
a.field1 = b;    // 'field1' sized at 4 bits
...
a.field1 = c;    // error: is 'field1' 4 or 5 bits?
```

Example 36

3.7.11.2.5 Mode 2 stream assignment compatibility

Only mode 2 streams of the same type are assignment-compatible. However, since all objects of a given mode 2 stream type are actually the same object, the concept of assignment compatibility is essentially redundant. The assignment and ternary operators are therefore also redundant with mode 2 stream operands, but are defined to allow these operators to be used with structure operands which contain mode 2 streams.

3.7.11.2.6 Mode 2 stream operators

The operators which may be applied to mode 2 streams are listed in Table 14 below. When a stream is read in an expression the value returned is the stream itself.

The first column in the table shows the operator. In this column, `x`, `y`, and `z` are expressions which evaluate to the same stream type (and so are assignment-compatible). Note that an object of type 'array of stream' is not a stream.

Operator	result type	Operation
`x'size`	`bit`	Return the number of lines in stream `x`; will be 0 before the first increment operation
`x.f`	`void`	Mode 2 stream field, write-only; see 3.7.11.2.1
`++x`	`stream`	Write the current line, increment the size of stream `x`, and return `x`
`x = y`	`stream`	Assign stream `y` to stream `x`, and return stream `x`
`(e1)?y:z`	`stream`	Conditional operator; `y` and `z` must be assignment-compatible. Returns stream `y` if `e1` evaluates true, and stream `z` otherwise
`(..., x)`	`stream`	Comma operator; returns stream `x` if `x` is the last expression

Table 14: mode 2 stream operators

3.7.12 array

Objects may be combined into *arrays*, which are aggregates of objects of the same type. Arrays are declared by listing the maximum size of each dimension in square brackets. Consider, for example, the array object defined by this declaration:

```
int [3][4][5] a;
```

Here `a` is a 3-dimensional array of 3 x 4 x 5 `int`s. The expression `a[i]` yields a two-dimensional array of 4 x 5 `int`s; the expression `a[i][j]` yields a one-dimensional array of 5 `int`s; and, finally, the expression `a[i][j][k]` yields a scalar object of type `int`. The *rank* of an expression or object is defined as its dimensionality. `a` is a 3-dimensional array and has rank 3. The expression `a[i]`, however, has rank 2; the expression `a[i][j]` has rank 1; and the expression `a[i][j][k]` has rank 0. Any scalar object has rank 0.

Arrays are stored in memory in row-major order; in other words, the last subscript varies fastest.

3.7.12.1 Array indexing

Zero-based indexes are used when accessing an array. For this example, the first index must evaluate to an integer in the range 0 to 2; the second must evaluate to an integer in the range 0 to 3; and the third must evaluate to an integer in the range 0 to 4. The index expressions must not contain any metavalues. A combination of static and dynamic checking is used to confirm that all array indexes are in range. An array index which is a constant expression is checked during compilation, and a syntax error is raised if it is out of range. A dynamic index is checked at run-time, and a run-time error is raised if it is out of range (see 11.1 below).

3.7.12.2 Subscript positioning

Arrays may be declared by listing the subscripts after the type name, or after the object name, or any combination of the two. These declarations, for example, define objects of type `int[3][4][5]`, and a function which returns an `int[3][4][5]`:

```
int [3][4][5] a;        // form 1
int [3][4] b[5];        // combined form 1 and 2
int [3] c[4][5];        // combined form 1 and 2
int d [3][4][5];        // form 2

int[3][4][5] foo {...}  // function returning array, form 1
```

Example 37

However, it should be noted that only form 1 can be used to define a function which returns an array object, and form 1 also makes it obvious that the object's dimensionality is part of the type of that object[1].

[1] Form 2 is provided for compatibility with C and related languages. More recent languages tend to use form 1; Maia's ability to use both forms, and a combination of the two, follows Java usage. Java, however, provides a (deprecated) feature to allow functions to return arrays using form 2; Maia requires the use of form 1. C does not allow functions to return arrays.

3.7.12.3 Comma-separated dimension lists

Arrays may be declared, or accessed, using an alternative syntax, in which the array dimensions are listed in a single pair of square brackets, with a comma-separated list of dimensions[1]. In this example, both `a` and `b` are of type `int[2][3]` (or, equivalently, `int[2,3]`):

```
int [2][3] a = {{0,1,2}, {3,4,5}};
int b[2,3]   = {{0,1,2}, {3,4,5}};

assert(a == b);
assert(a[0][1] == b[0,1]);
assert(a[0,1]  == b[0][1]);
```

Example 38

The comma-separated list is more compact when accessing multi-dimensional arrays. `kmap` objects, in particular, may have many dimensions and can be tedious to access using the fully-bracketed form.

When using this alternative syntax, array index expressions may not themselves be comma expressions, unless the comma expression is enclosed in parentheses[2].

3.7.12.4 Array assignment compatibility

Expressions that evaluate to arrays are assignment-compatible only if:

1. they have the same rank;
2. each dimension bound is identical;
3. the base type of each array is assignment-compatible;
4. the objects of that base type have the same size.

This code shows various examples of arrays which are, or are not, assignment-compatible:

```
int   [3,4,5] a;
int   [4,5]   b;
var32 [5]     c;
bit7  [5]     d;

a[0] = b;        // Ok
b = a[1];        // Ok
c = a[0,0];      // Ok (_DefaultWordSize assumed to be 32)
d = a[0,0];      // error: a is an array of ints; d is an array of bit7
```

Example 39

[1] The comma-separated list is used in Algol and derived languages; the fully-bracketed list is used in C and related languages.

[2] Function argument lists (argument-expression-list), which are also comma-separated, have the same restriction.

3.7.12.5 Array operations

The operators which support array operands are listed in Table 15 below. For the ternary operator (`?:`) `x` and `y` must be of the same type; for the remaining operators, they must simply be assignment-compatible.

The `'offset` operator may also be applied to any element in an array to find that element's offset within the array, in bits.

Two assignment-compatible arrays may be tested for equality and inequality. The arrays are equal if all corresponding elements contain the same data. For an array of streams, the stream identifiers (or 'handles') are tested for equality; the identifiers will compare equal if they refer to the same stream.

Operator	result type	Operation
`x'size`	`int`	Return the size of array `x`, in bits
`x'meta`	`bool`	Return `true` if `x` contains any metavalues, and `false` otherwise
`x[e1]`	any	Return element `e1` in `x`
`x = y`	`x`	Assign array `y` to array `x`
`x == y`	`bool`	Test `x` and `y` for equality
`x != y`	`bool`	Test `x` and `y` for inequality
`(e1)?x:y`	`x`	Returns array `x` if expression `e1` evaluates true, and array `y` otherwise
`(..., x)`	`x`	The comma operator; returns array `x` if `x` is the last expression

Table 15: array operators

4 OPERATORS AND EXPRESSIONS

4.1 Introduction

An *object* is a region of data storage which has an associated value. Every object is either a data object (2-state or 4-state), a boolean, a stream, or an aggregate containing a collection of data, boolean, and stream objects. Objects may be manipulated or combined using *operators*, in *expressions*.

The order in which the objects in an expression are combined is defined by the language's *precedence* and *associativity* rules (4.5.1). Each such combination (an addition or subtraction, for example) defines a *sub-expression*. Each sub-expression is evaluated in turn, and is replaced by a temporary object; this temporary object itself has a value, which may be combined with the values of the remaining sub-expressions.

An expression has a number of properties, which may be retrieved with the *attribute* operators (0). The most significant of these is its *size*. The meaning of the size attribute depends on the type of the object. However, in most cases, an object's size is the number of bits (2-state or 4-state) which are required to store that object. An object's size may range from 1 bit, up to a compiler-determined maximum, which is at least 2^{24} bits.

With the exception of `int` objects, Maia does not specify any interpretation of the data pattern within a data object. However, some operators (the signed comparisons, for example) may assume that the data is in 2's complement format, and that the data may be sign-extended by copying the value of the top bit. Other operators may assume that the data is in an IEC floating-point format. A non-`int` object itself has no property that specifies what format the data is in; the data interpretation is a higher-level concern, and is the responsibility of the programmer. In this respect, non-`int` data objects can be thought of as memory locations within a digital electronic system. The storage location itself has no properties, apart from its size; the control circuitry simply routes the contents of the storage location to a function unit, and then writes the transformed data to the same, or another, storage location. The function unit determines the operation to be carried out, and a given storage location may be connected to any function unit as required. In Maia, the function unit corresponds to an *operator*.

Under most circumstances, objects can be combined in expressions in a simple and intuitive way. This code, for example, carries out 4-state integer arithmetic operations on 24-bit variables:

```
var24 acc, b[10], c[10];
for(i=0; i<10; i++)
    acc += b[i] * c[i];
```

Example 40

For this example, the multiplication and addition are automatically selected as 24-bit integer operators, and the assignment to `acc` is selected as a 24-bit assignment. However, a number of potential complications may arise in these cases:

- if floating-point operations are required; or
- if the two operands of a binary operator have different sizes; or
- if an explicitly-sized operator is required.

Floating-point operations are described in (4.6). The complication in the remaining two cases is that at least one operand will have to be truncated or extended and, if it is extended, it may potentially require sign-extension. The sizing and extension rules that govern these cases are described (4.4). These rules define a *hardware-centric* view of arithmetic and logical operations, which is, in general, unlike the equivalent rules used in general-purpose programming languages, or in common HDLs.

4.2 Operator syntax

Many operators are available in both signed and unsigned versions, and in explicitly sized versions. The sign and size options must follow the basic operator symbol, with no intervening whitespace. The full operator syntax is `OP[#][$n]`, where the parts in brackets `[]` are optional. The `#` denotes a signed version of the operator, while the `$n` denotes an explicitly-sized (*n*-bit) operator.

All versions of an operator have the same precedence and associativity as the basic operator itself. An operator which is signed or sized may optionally be enclosed in parentheses for clarity.

The base operators are listed in Table 16, while the floating-point operators are listed in Table 18, Table 19, and Table 20. Some examples of operators are:

```
A = B - C;          // unsigned subtraction, implicitly sized
A = B -# C;         // signed subtraction, implicitly sized
A = B -$8 C;        // unsigned 8-bit subtraction
A = B -#$21 C;      // signed 21-bit subtraction
A = B (-#$21) C;    // operators may be bracketed for clarity

var16 d;
A =$21  d;          // unsigned (zero-extending) assignment (16 to 21 bits)
A =#$21 d;          // signed (sign-extending) assignment (16 to 21 bits)

A = (~#$21) d;      // invert operator: sign-extend d to 21 bits, invert
```

Example 41

4.3 Signed operators

Signed and unsigned operators are distinguished by the presence or absence, respectively, of a trailing `#` character. The `+` operator, for example, represents an unsigned addition, while the `+#` operator represents a signed addition. There are only two differences between the signed and unsigned versions of an operator:

1. If an input operand requires extension, then it will be sign-extended for a signed operator, and zero-extended for an unsigned operator.

2. For some operators the signed and unsigned versions of the operator may have different behaviour, and produce different results when given the same operands. The affected operators are the comparisons, right shift, division, and remainder (`<`, `<=`, `>`, `>=`, `>>`, `/`, `%`)[1].

The operators which may be signed are shaded in Table 16 below.

[1] The left shift operator has named signed and unsigned alternatives (`.SLA` for `<<#`, and `.SLL` for `<<`), but both have the same behaviour; the names are provided only for consistency with the right-shift versions (`.SRA` and `.SRL`).

4.4 Expression evaluation

When evaluating an expression, the current operator is first identified using the precedence and associativity summarised in Table 16. This operator, together with its operand(s), forms the current sub-expression. This sub-expression is evaluated according to (4.4.1), and is replaced with a temporary object of the same type and size as the sub-expression. This procedure is repeated until the complete expression has been evaluated.

Expressions may be arbitrarily parenthesised to specify the order in which operators should be evaluated.

The operands of an operator are always evaluated in a left-to-right order[1]:

```
int a = 4;
a = a++ + a;  assert(a == 9);      // a is guaranteed to be 9, and not 8
a = foo1() - foo2();               // foo1 is called before foo2
```

Example 42

4.4.1 sub-expression evaluation

The procedure for evaluating the current sub-expression is as follows:

1. The *operation size* is first determined as:
   i. If the operator is explicitly sized, then that size is the operation size;
   ii. Otherwise, if the operator is a shift or rotate, then the operation size is the size of the left operand;
   iii. Otherwise, the operation size is the size of the largest operand.
2. If any operands of the current operator have a size which is greater than the operation size, then those operands are truncated to the operation size
3. If any operands of the current operator have a size which is less than the operation size, then those operands are zero-extended if the operator is unsigned, and sign-extended if the operator is signed, to the operation size

Note that:

a) The assignment operator has a single operand (the right hand side)
b) The conditional or ternary operator is considered to have 2 operands for the purposes of deriving the operation size. In the expression `e1?e2:e3`, for example, only `e2` and `e3` participate in operation sizing
c) The compound assignments are expanded before applying these rules. The expression `a&=b`, for example, is treated as `a=a&b`
d) The floating-point operators are explicitly sized (as single, double, or extended-double precision), and the operands are required to have the same size as the operator[2]. There is therefore no potential ambiguity when evaluating floating-point sub-expressions.

[1] Left-to-right ordering is common in many languages (C# and Java, for example); the ordering is undefined in C.

[2] If an operand does not have the required size (a single-precision operand is required for a double-precision operator, for example) then it should be converted to the correct size using a cast operator (4.5.6).

4.5 Operators

4.5.1 Precedence and order of evaluation

Table 16 summarises the operator precedence and associativity rules. The operators listed in (4.5.4) onwards are also listed in order of precedence, with the highest first. Operators on the same line of Table 16 have the same precedence; rows are in order of decreasing precedence[1]. Where an operator in the table implies an arithmetic operation (`++`, `--`, `*`, `/`, `%`, `+`, `-`, `<`, `<=`, `>`, and `>=`), that operation is defined only for *integer* values, using 2-state or 4-state integer arithmetic. The floating-point operators are listed separately (on page 73), but have the same precedence and associativity as the integer version. Note that the equality operators (`==` and `!=`) are bitwise operators, and so are valid for both integer and floating-point use.

An operator's *associativity* determines the grouping of operators at the same precedence level. The addition and subtraction operators, for example, associate left-to-right, and the expression `A-B+C` is therefore evaluated as `(A-B)+C`, rather than `A-(B+C)`.

	Operators							**Associativity**
postfix:	()	[]	++	--	'size	'msb	'meta	left to right
	'offset	'last	.(x)	.(x:y)	.			
prefix:	!	~	++	--	+	-	(cast)	right to left
binary:	*	/	%					left to right
	+	-						left to right
	<<	>>	.R<<	.R>>				left to right
	<	<=	>	>=				left to right
	==	!=						left to right
	&							left to right
	^							left to right
	|							left to right
	&&							left to right
	||							left to right
ternary:	?:							right to left
assignment:	=	*=	/=	%=	+=	-=	&=	right to left
	^=	|=	<<=	>>=	.R<<=	.R>>=		
comma:	,							left to right

Table 16: precedence and associativity of operators

The operators which may optionally be signed and sized are shaded in the table.

[1] Apart from some deletions (`->`, `&`, `*`, and `sizeof`) and additions (`'size`, `'msb`, `'meta`, `'offset`, `'last`, `.(x)`, `.(x:y)`, `.R<<`, `.R>>`, `.R<<=`, and `.R>>=`), this table is otherwise identical to the corresponding table for C.

4.5.2 Operator equivalents

A number of operators have alternative names. These are listed in Table 17.

Operator	Form 1	Form 2
Rotate Left	.R<<	.ROL
Rotate Right	.R>>	.ROR
Shift Left Logical	<<	.SLL
Shift Left Arithmetic	<<#	.SLA
Shift Right Logical	>>	.SRL
Shift Right Arithmetic	>>#	.SRA
Unsigned Less Than	<	.ULT
Unsigned Greater Than	>	.UGT
Unsigned Less than or Equal	<=	.ULE
Unsigned Greater than or Equal	>=	.UGE
Signed Less Than	<#	.SLT
Signed Greater Than	>#	.SGT
Signed Less than or Equal	<=#	.SLE
Signed Greater than or Equal	>=#	.SGE
Equality	==	.EQ
Inequality	!=	.NE
Logical AND	&&	and
Logical OR	||	or

Table 17: Operator equivalents

The textual alternative names for the shifts and comparisons are already implicitly signed or unsigned, and so may not be followed by a # character. They may, however, be sized. Some examples of valid and invalid operators are shown below.

```
.SRL      // implicitly-sized right shift
.SRL#     // invalid; .SRL is implicitly unsigned
.SRL$12   // 12-bit right-shift
.SRA#     // invalid; .SRA is implicitly signed
>>#       // shift right arithmetic; equivalent to .SRA
```

Example 43

4.5.3 Primary expressions

Syntax

```
primary-expression:
    identifier
    constant
    ( expression )
```

4.5.4 Postfix operators

Syntax

```
postfix-expression :
   primary-expression
   postfix-expression [ expression ]
   postfix-expression ( argument-expression-list_opt )
   postfix-expression . identifier
   postfix-expression ++
   postfix-expression --
   postfix-expression . bitslice
   postfix-expression ` attribute-operator
   postfix-expression ' attribute-operator
```

4.5.4.1 Array subscripting

A postfix expression followed by one more expressions in square brackets designates an element of an array object (3.7.12).

4.5.4.2 Function calls

A postfix expression followed by parentheses `()` containing a possibly empty list of comma-separated expressions is a function call. The expressions within the parentheses are the actual parameters to that function; the expressions are evaluated left-to-right.

Syntax

```
argument-expression-list :
   assignment-expression
   argument-expression-list , assignment-expression
```

4.5.4.3 Structure and stream members

A postfix expression followed by the `.` operator and an identifier indicates a structure or stream access. The postfix expression must evaluate to a structure (3.7.10) or stream (3.7.11), while the identifier must be a member within that structure or stream.

4.5.4.4 Postfix increment and decrement operators

A postfix expression followed by `++` or `--` indicates an increment or decrement operation. In both cases, the result is the value of the operand. After the result is obtained, the value of the operand is incremented, for `++`, or decremented, for `--`. The operand must be an lvalue.

If the operand evaluates to a data object, then the increment or decrement operation is carried out by adding or subtracting 1 to or from the least significant bit of the object (in other words, it is an integer operation). If the operand evaluates to a mode 1 stream, then the increment or decrement operation is applied to the file offset within that stream. It is an error if the operand evaluates to anything else.

4.5.4.5 Bitslice operator

A postfix expression followed by the `.` operator and parentheses `()` indicates a bitslice, or a bitfield access, within the postfix expression. Bitslices may be applied both to lvalues and to rvalues. It is an error if the postfix expression does not evaluate to an arithmetic object.

Syntax

```
bitslice :
   expr1.(expr2)
   expr1.(expr-msb : expr-lsb)
expr1 :     expression
expr2 :     expression
expr-msb : expression
expr-lsb : expression
```

Semantics

Bitslices are addressed using descending indexes. If two indexes are specified, they may be equal to address a single bit. In this case, the bitslice may be expressed in a more compact form by specifying only one index.

The LSB of any object always has an index of `0`. The full set of requirements for the indexes can therefore be expressed as follows:

```
(expr2 < expr1'size) && (expr2 >= 0)
(expr-msb < expr1'size) && (expr-msb >= expr-lsb)
expr-lsb >= 0
```

Indexes are evaluated and checked at runtime, and a runtime error is raised if the equalities above are violated.

Some examples of bitslice usage are:

```
bit16 temp = 0xffff;
temp.(4:2) = 4;        // set bits [4,3,2] to [1,0,0]; others unchanged
temp.(7:5) = 0xb;      // set bits [4,3,2] to [0,1,1]
assert(temp == 0xff73);

bit16[3] R = {0, 0, 0xf000};
bit4 data;
R[2].(1) = 1;          // set bit 1 of R[2]
data = R[2].(15:12);   // set data to 15
assert(R[2] == 0xf002 && data == 15);

var16 test1 = 0xabcd;
var16 test2 = 0xbcde;
test1.(test1'size-1 : test1'size-4) = test2.(3:0);
assert(test1 == 0xebcd);
assert((test1 ^ test2).(15:8) == 0x57);
assert(test1.(0:0)'size == 16);
```

Example 44

The size of a bitslice expression is the size of the object being sliced (the postfix expression); it is not the size implied by the slice indexes, which may change at runtime. The slice can be considered to be a temporary object of the same size as the original object, with the required bits shifted to the bottom of the temporary.

Size checking for bitslice expressions may be relaxed when writing to a DUT port in a drive statement; see 3.1.4.

4.5.4.6 Attribute operators

A postfix expression followed by the ' (or `) operator and an attribute returns an attribute, or property, of the operand.

Syntax

```
attribute-operator: one of
   size offset msb meta last last(expression)
```

Semantics

size The size attribute returns the size of the operand. If the operand evaluates to a data object, a boolean, a structure, or an array, then the value of the attribute is the total size, in bits, of that data object, boolean, structure, or array. If the operand evaluates to a mode 1 or a mode 2 stream, then the value of the attribute is the number of lines in the corresponding text file. It is an error if the operand evaluates to anything else.

offset The offset attribute returns an offset within an object. If the operand evaluates to a member within a structure, or is an array indexing expression, then the value of the attribute is the offset of that member or element within the structure or array, measured in bits. If the operand evaluates to a mode 1 stream, then the value of the attribute is the current line number within that stream. It is an error if the operand evaluates to anything else.

The offset of the first object within its container always has the value 0.

msb The operand must evaluate to an ivar object. The return type of the msb attribute is `var1` for a var object, and `bit1` otherwise; its value is the value of the most significant bit of that operand.

meta The operand may evaluate to anything except a stream or a stream member. The meta attribute returns `true` if the operand is, or contains, a data object which has a metavalue (`X` or `Z`), and `false` otherwise. If the object is an aggregate which contains a stream, then that stream contributes a value of `false` to the overall determination.

last Returns a previous value of the operand.

The `'size` and `'offset` attributes return a `bit` object. The size of this value is compiler determined, but is at least 28 bits.

The `'last` attribute returns a previous value of a DUT input or IO, as it would have been sampled by the DUT[1]. The operand must be declared as a DUT input or IO, and must appear in a clocked drive declaration (8.3.2), to allow the sample clock to be identified. The expression `sig.last(n)` returns the `n`'th previous value of signal `sig`, as it would have been sampled by the relevant clock at the DUT.

The expression `sig.last(1)` returns the value of `sig` that would have been sampled on the previous clock edge, while `sig.last(2)` returns the value that would have been sampled on the preceding edge, and so on. The expression `sig.last` is equivalent to `sig.last(1)`. If the edge count (`n`) is

[1] The 'last attribute may be used to avoid race conditions where one thread generates a DUT input, while another thread reads the same input (either directly or from a DUT output which has a combinatorial path from the input). In this case, the writer and reader threads will execute in an arbitrary order, and the reader may read the value before it has been written. This condition can be avoided by instead reading the input as it would have been seen by the DUT at the previous clock edge.

supplied, it must be greater than or equal to 1. The maximum value of `n` is compiler-determined, but will be at least 4096. A run-time error will be raised if the edge count is out of range.

The value returned is maintained in a pipeline by the testbench itself, and is sampled on the relevant sample clock, with the supplied or default setup time for the relevant signal. If the edge count for a given signal is statically determinable then the compiler will generate the sample pipeline for that signal with the size given by the maximum edge count in the source code. If the edge count cannot be determined during compilation then the sample clock must instead be declared with a `pipeline` specification, which gives the pipeline size required. If, for example, a 10-cycle sample history is required for signal `D`, and the relevant clock is signal `C`, then `C` should be declared with '`create_clock C -pipeline 10`'. In this case, a run-time error will be raised if, during execution of the model, the edge count is found to be outside the range [1,10].

Examples

```
var24[2][3][4] x;
assert(
     (x`size == 576)       &&
     (x[0]`size == 288)    &&
     (x[0][0]`size == 96)  &&
     (x[0][0][0]`size == 24));
```

Example 45

```
DUT {
   module reg4                   // 4-bit reg with sync reset
     (input  C, R,
      input  [3:0] D,
      output [3:0] Q);
   create_clock C -pipeline 2;   // 2-level sample pipe
   [C, R, D] -> [Q];
}

main() {
   int level = 1;
   [.C, 1, 0] -> [0];            // reset
   [.C, 0, 1] -> [1];            // n+1 cycles required to prime n-cycle pipe
   for(bit4 i=2; i<10; i++) {
      [.C, 0, i] -> [i];
      assert((D'last(level) == i) && (D'last(level+1) == i-1));
   }
}
```

Example 46

4.5.5 Unary operators

Syntax

```
unary-expression:
   postfix-expression
   unary-operator unary-expression
   ( type-name )  unary-expression

unary-operator: one of
   ++ -- + - ~ ! float+ float-
```

4.5.5.1 Prefix increment and decrement

The expressions `++E` and `--E` are equivalent to `(E+=1)` and `(E-=1)`, respectively. `E` must be an lvalue.

If the operand evaluates to an `int`, `bit` or `var` object, then that object is incremented or decremented using 2-state or 4-state integer arithmetic, in the same way as the postfix `++` and `--` operators (4.5.4.4). The result is the new value of the operand after the increment or decrement has completed.

If the operand evaluates to a mode 1 stream, then the stream offset is incremented or decremented, and the new state of the stream is returned. If the operand evaluates to a mode 2 stream, then `++E` writes the current line to `E`, increments the size of `E`, and returns the new state of `E`.

It is an error if the operand evaluates to anything else, or if the `--` operator is applied to a mode 2 stream.

4.5.5.2 Unary arithmetic, bitwise, and logical operators

The operand of the unary `+` and unary `-` operators (and their floating-point equivalents) must evaluate to an ivar object. The result has the type and size of the operand.

The unary addition operators return their operand unless that operand is a `var` which contains one or more metavalues; in this case, the result has a value of all `X`, as if `+E` had been evaluated as `(0+E)`.

The plain `-` operator carries out integer subtraction from `0` (with the exception noted in (4.6.1.2)), while the floating-point equivalents carry out a floating-point subtraction from `0.0`. An integer subtraction from `0` is carried out using 2-state integer arithmetic if the operand is of type `int` or `bit`, or 4-state integer arithmetic if the operand is of type `var`. The `.F-`, `.F1-`, `.F2-`, and `.F3-` operators carry out a floating-point subtraction from `0.0`, and return the value of the result.

For the floating-point unary operators, the operand is required to have the same size as the operator (single, double, or extended-double precision).

The result of the complement operator **~** is the bitwise complement of its operand. The operand must evaluate to a data object; the result has the same type and size as the operand. The complement operation for 4-state objects is defined in (3.7.7.6).

The result of the logical negation operator `!` is of type `bool`. It has value `false` if the operand evaluates true, and value `true` otherwise.

4.5.6 Cast operators

The cast operators convert a data object to or from a floating-point representation. When converting a `var` to floating point, any metavalues bits are converted to `1`.

Syntax

```
type-name: one of
   real1 real2 real3 int bit_n var_n
```

Examples

```
var8  i = 255;          // i is 8`hff
real1 a = (real1)i;     // a is 255.0F
real2 b = (real2)i;     // a is 255.0 (64`h406f_e000_0000_0000)
real3 c = (real3)i;     // a is 255.0L
int   j = (int)b;       // j is an integer, with value 8`hff
```

Example 47

Floating-point data may also be converted to a sized integer, with unused high bits discarded:

```
real2 b;
int   d, e, f;
for(b = 254.0; b .F< 258.0; b = b .F+ 1.0) {
   d = (int) b;
   e = (bit8)b;
   f = (bit1)b;
   report "d: %d; e: %d; f: %d\n", d, e, f;
}
```

Example 48

This code produces the following output:

```
d: 254; e: 254; f: 0
d: 255; e: 255; f: 1
d: 256; e: 0; f: 0
d: 257; e: 1; f: 1
```

4.5.7 Multiplicative operators

The multiplicative operators implement multiplication, division, and remainder. Both operands are required to have arithmetic type.

When using the floating-point versions of the operators, both operands, and the operator itself, are required to have the same size (single, double, or extended-double precision), and the result has that size.

Syntax

```
multiplicative-expression:
   unary-expression
   multiplicative-expression * unary-expression
   multiplicative-expression / unary-expression
```

```
    multiplicative-expression % unary-expression
    multiplicative-expression float* unary-expression
    multiplicative-expression float/ unary-expression
```

Semantics

The `*`, `/`, and `%` operators, with no suffix, implement unsigned integer multiplication, rational division, and remainder, respectively. The `*#`, `/#`, and `%#` operators implement the signed versions of these operations. The unsigned and signed versions of the operators differ as follows:

1 If an operand requires extension, then the unsigned operators will zero-extend that operand, while the signed operators will sign-extend that operand;

2 The unsigned operators view their operands as positive binary integers, and carry out an unsigned operation; the signed operators view their operands as two's complement integers, and carry out a signed operation.

The `/` and `/#` operators return the rational result truncated towards 0; the `%` and `%#` operators return the remainder of the corresponding division (`/` or `/#`) operation. This is often referred to as "truncating division"[1]:

```
  7 /#  3  =  2 rem  1
 -7 /#  3  = -2 rem -1
  7 /# -3  = -2 rem  1
 -7 /# -3  =  2 rem -1
```

The relationship *dividend = quotient * divisor + remainder* holds for these operators.

4.5.8 Additive operators

The additive operators implement addition and subtraction. Both operands are required to have arithmetic type.

When using the floating-point versions of the operators, both operands, and the operator itself, are required to have the same size (single, double, or extended-double precision), and the result has that size.

Syntax

```
 additive-expression:
    multiplicative-expression
    additive-expression + multiplicative-expression
    additive-expression - multiplicative-expression
    additive-expression float+ multiplicative-expression
    additive-expression float- multiplicative-expression
```

Semantics

The `+` and `-` operators, with no suffix, implement unsigned integer addition and subtraction. The `+#` and `-#` operators implement the signed version of the operation. If an operand requires extension,

[1] `%` implements a remainder operation, and not a modulus operation. `/` and `%` have the same definition in Maia and C, although `%` is sometimes referred to as the 'modulus' operator in C. `%` is equivalent to the MOD function in Fortran 90, and the `rem` operator in Common Lisp, Ada, and VHDL.

then the unsigned operators will zero-extend that operand, while the signed operators will sign-extend that operand. The results of the unsigned and signed integer operations are otherwise identical.

4.5.9 Shift and rotate operators

The shift and rotate operators implement bitwise left and right shift, and bitwise left and right rotation. Both operands are required to have arithmetic type.

Syntax

```
shift-expression:
   additive-expression
   shift-expression <<  additive-expression
   shift-expression >>  additive-expression
   shift-expression .R<< additive-expression
   shift-expression .R>> additive-expression
```

Semantics

The `<<`, `>>`, `.R<<`, and `.R>>` operators, with no suffix, implement the unsigned versions of the operation. The same operators with a `#` suffix implement the signed versions of the operations. The unsigned and signed operations differ as follows:

1 If the left operand requires extension (4.4.1), then the unsigned operators will zero-extend that operand, while the signed operators will sign-extend that operand;

2 The `>>` operator carries out a logical shift (by shifting in `0`), while `>>#` carries out an arithmetic shift (by duplicating the sign bit). The remaining shift and rotate operators have the same behaviour, irrespective of whether or not they have a `#` suffix.

The result of **`E1 << E2`** `is` **`E1`** left-shifted **`E2`** bit positions; the vacated bits are filled with zeroes.

The result of **`E1 >> E2`** `is` **`E1`** right-shifted **`E2`** bit positions. The vacated bits are filled with zeroes for the unsigned operator, or with a copy of the top bit of **`E1`** for the signed operator.

The result of **`E1 .R<< E2`** `is` **`E1`** left-rotated **`E2`** bit positions. Rotation occurs within a word whose size is given by the size of the operator (4.4.1). If **`E1`** requires extension, then it will be zero-extended to the operator size if the operator is unsigned, or sign-extended to the operator size if the operator is signed, before the rotation is carried out.

The result of **`E1 .R>> E2`** `is` **`E1`** right-rotated **`E2`** bit positions. Rotation occurs within a word whose size is given by the size of the operator (4.4.1). If **`E1`** requires extension, then it will be zero-extended to the operator size if the operator is unsigned, or sign-extended to the operator size if the operator is signed, before the rotation is carried out.

4.5.10 Relational operators

The relational operators implement integer and floating-point comparisons. Both operands are required to have arithmetic type.

When using the floating-point versions of the operators, both operands, and the operator itself, are required to have the same size (single, double, or extended-double precision).

Syntax

```
relational-expression:
   shift-expression
   relational-expression  <  shift-expression
   relational-expression  >  shift-expression
   relational-expression  <=  shift-expression
   relational-expression  >=  shift-expression
   relational-expression float-compare shift-expression
```

Semantics

The operators return a result of type `bool`; the result is `true` if the specified relationship is true, and `false` otherwise.

The > (greater than), < (less than), >= (greater than or equal), and <= (less than or equal) operators, with no suffix, implement the unsigned integer comparisons. The >#, <#, >=#, and <=# operators implement the signed integer comparisons. The unsigned and signed versions of the operators differ as follows:

1 If an operand requires extension, then the unsigned operators will zero-extend that operand, while the signed operators will sign-extend that operand;

2 The unsigned operators assume that their operands are unsigned binary, while the signed operators assume that their operands are 2's complement. This affects the operator result, as shown in the examples below.

If the operands contain any metavalues, all four relationships will be false (3.7.7.4). Otherwise, at least one of the relationships will be true.

Examples

```
var4 r1, r2;
r1 = 0b1001;              // 9 or -7
r2 = 0b0011;              // 3
assert(r1 >  0);          //  9 > 0
assert(r1 <# 0);          // -7 < 0
assert(r1 >  r2);         //  9 > 3
assert(r1 <# r2);         // -7 < 3
```

Example 49

The size of the relational operators is defined by the normal operator sizing rules (4.4.1), and determines the number of bits of the operands which will be compared:

```
var4 r1 = 0b0110;
var4 r2 = 0b1000;
assert(r1 >$3 r2);        // 6 > 0
assert(r1 <$4 r2);        // 6 < 8
```

Example 50

4.5.11 Equality operators

The equality operators determine whether or not their operands have the same value. One of the following 3 conditions must hold:

1. both operands must be of the same type, where that type is `int`, `var`, `kmap`, or `bool`; or
2. both operands must be assignment-compatible structures (3.7.10.2); or
3. both operands must be assignment-compatible arrays (3.7.12.4).

The equality operators carry out a bitwise comparison, and so may be used for both integer and floating-point comparisons.

Syntax

```
equality-expression:
   relational-expression
   equality-expression == relational-expression
   equality-expression != relational-expression
```

Semantics

The equality operators are analogous to the relational operators, but have a lower precedence, and may be used to test K-maps, booleans, structures and arrays for equality. They return a result of type `bool`; the result is `true` if the specified relationship is true, and `false` otherwise.

If the operands contain any metavalues, then those metavalues are included in the test (3.7.7.5). For any pair of operands, exactly one of the relationships is true.

4.5.12 Bitwise AND operator

The `&` operator returns the bitwise AND of the two operands, as defined in (3.7.7.6). Both operands are required to have data type.

Syntax

```
AND-expression:
   equality-expression
   AND-expression & equality-expression
```

4.5.13 Bitwise exclusive OR operator

The `^` operator returns the bitwise exclusive-OR of the two operands, as defined in (3.7.7.6). Both operands are required to have data type.

Syntax

```
exclusive-OR-expression:
   AND-expression
   exclusive-OR-expression ^ AND-expression
```

4.5.14 Bitwise inclusive OR operator

The `|` operator returns the bitwise inclusive-OR of the two operands, as defined in (3.7.7.6). Both operands are required to have data type.

Syntax

```
inclusive-OR-expression:
   exclusive-OR-expression
```

```
    inclusive-OR-expression | exclusive-OR-expression
```

4.5.15 Logical AND operator

Both operands are required to be boolean. An operand of an arithmetic type is considered to be boolean; see (3.7.4.2).

Syntax

```
logical-AND-expression:
   inclusive-OR-expression
   logical-AND-expression && inclusive-OR-expression
```

Semantics

The `&&` operator return a result of type `bool`. **`E1 && E2`** yields `true` if both **`E1`** and **`E2`** are true, and `false` otherwise. **`E2`** is not evaluated if **`E1`** is false.

4.5.16 Logical OR operator

Both operands are required to be boolean. An operand of an arithmetic type is considered to be boolean; see (3.7.4.2).

Syntax

```
logical-OR-expression:
   logical-AND-expression
   logical-OR-expression || logical-AND-expression
```

Semantics

The `||` operator return a result of type `bool`. **`E1 || E2`** yields `true` if either **`E1`** or **`E2`** is true, and `false` otherwise. **`E2`** is not evaluated if **`E1`** is true.

4.5.17 Conditional operator

The first operand is required to be boolean. An operand of an arithmetic type is considered to be boolean; see (3.7.4.2). The second and third operands may be of any type, but must be assignment-compatible.

Syntax

```
conditional-expression:
   logical-OR-expression
   logical-OR-expression ? expression : conditional-expression
```

Semantics

For the expression **`E1?E2:E3`**, **`E1`** is evaluated first. **`E2`** is evaluated only if **`E1`** is true; **`E3`** is evaluated only if **`E1`** is false. The result has the value of whichever of **`E2`** or **`E3`** was evaluated.

If **`E2`** and **`E3`** are of an arithmetic type, and one or more of them is of type `var`, then the result is of type `var`; otherwise, the result is of type `int`. **`E2`** and **`E3`** must otherwise be of the same type, and the result is of that type.

4.5.18 Assignment operators

The assignment operators write a source location (an rvalue) to a destination location (an lvalue). The left operand must be an lvalue. The left and right operands must be assignment-compatible.

Syntax

```
constant-assignment-expression :
   assignment-expression

assignment-expression :
   conditional-expression
   unary-expression assignment_operator assignment-expression

assignment_operator: one of
   = *= /= %= += -= <<= >>= .R<<= .R>>= &= ^= |=
```

Semantics

An assignment expression has the value of the left operand after the assignment, but is not an lvalue. The type of the expression is the type of the left operand.

A `constant-assignment-expression` must have a known value during compilation.

An assignment is a compound assignment if it is of the form *op*=; it is otherwise a simple assignment. The compound assignment **E1 *op*= E2** is equivalent to **E1 = E1 *op* (E2).**

A simple assignment may optionally be sized or signed, or both. A size modifier specifies the number of bits which will be copied from the source location, while the absence or presence of a # modifier specifies whether these bits should be zero-extended or sign-extended, respectively. The compound assignments, however, may not be signed or sized, because of the potential confusion over whether the modifiers refer to the base operator, or to the assignment.

An assignment is evaluated by following this procedure:

1 the operation size (4.4.1) is first determined. For assignment, the operation size is the size of the assignment operator, if it is explicitly sized, and is otherwise the size of the source operand.

2 If the operation size is less than the source size, the source data is truncated to the operation size. Otherwise, if the operation size is greater than the source size, the source data is zero-extended to the operation size for an unsigned assignment (=), or sign-extended to the operation size for a signed assignment (=#).

3 The resized source data is then written to the destination. If the resized data is wider than the destination, it is truncated to the destination size; if it is narrower than the destination, it is zero-extended to the destination size for an unsigned assignment, or sign-extended to the destination size for a signed assignment. The entire destination is always over-written.

The bitslice operator (4.5.4.5) should be used if it is necessary to leave some bits of the destination unmodified.

The assignment operator is best viewed as a hardware logic unit, which contains, and controls writes to, a memory location. The unit has a fixed-size input bus, where the size of the bus is given by the operation size. The input to the logic unit is found by truncating, or extending, the operand to the size

of the input bus. The unit always overwrites the entire memory location from the input bus, truncating or extending the input bus as necessary.

This is illustrated in the diagram below, which shows a simple 2-input addition operation. In this example, a 5-bit adder adds a 4-bit and a 3-bit register, both of which are zero-extended. The 5-bit output is then sign-extended and written to a 6-bit register. The corresponding Maia code is:

```
var4 A;
var3 B;
var6 C;
C =# A +$5 B;
```

Example 51

The assignment in this example is unsized, but the operand is 5 bits, so the assignment operation size is also 5 bits. The 5-bit result of the addition operation is then sign-extended to 6 bits when written to the destination. The corresponding circuit is:

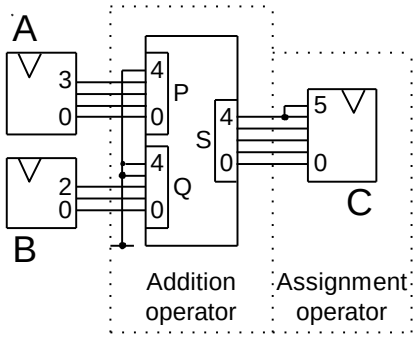


Figure 2: assignment input extension

Examples

```
var6 d;
var2 e = 2;

d =$4  6`h3f; assert(d == 6`h0f);
d =#$3 6`h0b; assert(d == 6`h03);
d =#$4 6`h0b; assert(d == 6`h3b);
d =$8  0xff;  assert(d == 6`h3f);  // note that no error is reported
d =    2`b10; assert(d == 6`h02);
d =#   2`b10; assert(d == 6`h3e);
d =    e;     assert(d == 6`h02);
d =#   e;     assert(d == 6`h3e);

int4 f = 4`b1001;
d =    f;     assert(d == 6`h09);
d =#   f;     assert(d == 6`h39);
d =$5  f;     assert(d == 6`h09);
d =#$5 f;     assert(d == 6`h39);
```

Example 52

4.5.19 Comma operator

Syntax

```
constant-expression :
   expression

expression :
   assignment-expression
   expression , assignment-expression
```

Semantics

The left operand of a comma operator is evaluated as a void expression. The right operand is then evaluated; the result has the type and value of the right operand.

A `constant-expression` must have a known value during compilation.

4.6 Floating-point operators and expressions

4.6.1 Introduction

Maia provides floating-point arithmetic, comparison, and cast operators. Each operator is preceded by `.F`, and has a different version for IEC 60559 single-precision, double-precision, and extended double-precision operands[1]. These three sizes are identified by the suffixes 1, 2, and 3, respectively. The three addition operators, for example, are `.F1+`, `.F2+`, and `.F3+`.

These operators are essentially equivalent to hardware floating-point units. The `.F1+` operator, for example, takes two single-precision operands, and returns a single-precision result. It is the programmer's responsibility to ensure that the operands are appropriate. There are no dedicated floating-point data types, and any data object may be used as an operand to a floating-point operator, as long as it is correctly sized. From a hardware perspective, this is analogous to allowing any 64-bit memory location to be connected to a 64-bit floating-point adder; the result returned by the adder will make little sense if the input memory locations do not actually contain floating-point data.

While this is the obvious way to handle hardware descriptions, it is not how most general-purpose programming languages (or HDLs) operate. Consider, for example, this C program[2]:

```
#include <stdio.h>
int main(void) {
   double a = 2;
   double b = 3 * a;
   printf("3a is %3.1f (%lx)\n", b, *(long *)&b);
   return 0;
}
```

Example 53

[1] Verilog supports only a 64-bit real type, which corresponds to double-precision on all supported systems. The Verilog code generator therefore does not support float and double-extended precisions; see (14.7.1).

[2] This code assumes that 'long' and 'double' both contain the same number of bits; in general, it will only work on a 64-bit machine. Note also that the actual bit pattern in a 'double' variable cannot simply be printed with an 'x' format; various casts are required.

The Maia equivalent[1] is:

```
int main(void) {
   int64 a = 2.0;          // or declare as real2 (4.6.2)
   int64 b = 3.0 .F* a;
   report("3a is %3.1f (%x)\n", b, b);
   return 0;
}
```

Example 54

both programs produce the same output:

```
3a is 6.0 (4018000000000000)
```

The C compiler has made a number of assumptions about the programmer's intentions. First, it has assumed that the constant `2` in the assignment `a=2` should actually be the bit pattern `0x4010000000000000`, rather than the bit pattern `0x2`. Secondly, for the assignment `b = 3*a`, the compiler has assumed that (since one of the operands is known to be a double) the constant `3` is actually the bit pattern `0x4008000000000000`, and that a double-precision multiply is required, rather than an integer multiply. These are useful assumptions for general-purpose programming problems, but are arguably not appropriate for hardware description and testing.

Maia makes no assumptions about the programmer's intentions. The constants `2.0` and `3.0` must therefore be explicitly entered as floating-point values, and not integer values; the multiplication operator must also be specified as `.F*`, rather than simply as a general-purpose `*`. `(3.0 * a)`, for example, carries out an integer multiplication, while `(3 .F* a)` multiplies the floating-point bit pattern in `a` by the integer `3`. In this context, only `(3.0 .F* a)` produces the expected answer of `6.0`.

The strict requirements that floating-point constants must contain a decimal (or hexadecimal) point, and that floating-point operators should be used for floating-point expressions, are relaxed in two specific cases (4.6.1.1 and 4.6.1.2). These relaxations simplify the handling of time values.

(15) contains an example floating-point program, which calculates π to 15 decimal digits. The program is not as concise as one written in a general-purpose language, but Maia is a domain-specific language, and will not normally be used for general floating-point arithmetic problems.

4.6.1.1 Decimal point exception

Any constant which represents a time value, and which is not part of a larger expression, is interpreted as floating-point, whether or not it contains a decimal point. This affects only `wait` statements and times specified in a DUT section.

```
DUT {
   D1 -> posedge C = ( 2:-0.1)    // tSU 2.0; tH 0.1
   ...
}

f() {
   wait 1;                        // waits 1.0 time units
   wait 1.0;                      // waits 1.0 time units
   wait 2.0 .F* 1.5;              // waits 3.0 time units
```

[1] This code assumes that a double contains 64 bits, which is true of all supported systems. 'real2' may be used rather than 'int64', to avoid this assumption.

```
   wait 2 .F* 1.5;               // ERROR: '2' is not floating-point in this context
}
```

Example 55

4.6.1.2 Floating operator exception

The integer unary plus and minus operators are interpreted as floating-point unary plus and minus when they precede a constant which represents a time value:

```
DUT {
   D1 -> posedge C = ( 2.1:-0.1)    // integer unary minus may be used instead of...
   D2 -> posedge C = ( 2.1:.F-0.1)  // verbose and potentially confusing
   D3 -> posedge C = ( 2.1:+0.1)    // integer unary plus may be used instead of...
   D4 -> posedge C = ( 2.1:.F+0.1)  // verbose and potentially confusing
   ...
}
```

Example 56

4.6.2 Declarations

Maia has no floating-point data types, but any objects which are intended to hold float data must be correctly sized for that data. On all currently-supported systems, single-precision data is 32-bit, and double-precision data is 64-bit. However, extended double-precision may be 64-bit, 80-bit, or 128-bit.

The `real1`, `real2`, and `real3` keywords are provided to avoid potential sizing problems; these are correctly sized by the compiler for the underlying types. When used in a declaration, these keywords are simply syntactic sugar for a correctly-sized variable:

```
real1 a;    // equivalent to 'int32 a' on most systems
real2 b;    // equivalent to 'int64 b' on most systems
real3 c;    // equivalent to 'int64 c', 'int80 c', or 'int128 c' on most systems
report("real2 is %d bits\n", b`size);
```

Example 57

Note that these declarations do *not* flag to the compiler that a floating-point value is stored in `a`, `b`, or `c`; the compiler has no interest in the contents of a data object. It is the programmer's responsibility to track the meaning of any bit pattern in an object.

4.6.3 Operators

Table 18, Table 19, and Table 20 below list the floating-point arithmetic operators. These operators have the same precedence and associativity as the corresponding integer arithmetic operators. The binary arithmetic operators take two operands of the same size, and return a result of that size; the comparison operators take two operands of the same size, and return a boolean result. An error will be reported if the operands of any of these operators are incorrectly sized.

The operators have alternative textual names, which are listed in the tables below. If the size numeral is omitted, it is assumed to be 2, for double-precision.

The compound assignment operators are not defined for floating-point data; the **+=** operator, for example, carries out an integer addition.

Floating-point data may be converted to a different precision, or to and from integer data using the cast operators; see (4.5.6).

Syntax

```
float+: one of
   .F1+  .F2+  .F3+  .F1ADD .F2ADD .F3ADD .F+  .FADD

float-: one of
   .F1-  .F2-  .F3-  .F1SUB .F2SUB .F3SUB .F-  .FSUB

float*: one of
   .F1*  .F2*  .F3*  .F1MUL .F2MUL .F3MUL .F*  .FMUL

float/: one of
   .F1/  .F2/  .F3/  .F1DIV .F2DIV .F3DIV .F/  .FDIV

float-compare: one of
   .F1<  .F2<  .F3<  .F1LT  .F2LT  .F3LT  .F<  .FLT
   .F1>  .F2>  .F3>  .F1GT  .F2GT  .F3GT  .F>  .FGT
   .F1<= .F2<= .F3<= .F1LE  .F2LE  .F3LE  .F<= .FLE
   .F1>= .F2>= .F3>= .F1GE  .F2GE  .F3GE  .F>= .FGE
```

Single precision	Form 1	Form 2		
Addition, unary +	.F1+	.F1ADD		
Subtraction, unary -	.F1-	.F1SUB		
Multiplication	.F1*	.F1MUL		
Division	.F1/	.F1DIV		
Less than	.F1<	.F1LT		
Greater than	.F1>	.F1GT		
Less than or equal	.F1<=	.F1LE		
Greater than or equal	.F1>=	.F1GE		

Table 18: single-precision real operators

Double precision	Form 1	Form 2	Form 3	Form 4
Addition, unary +	.F2+	.F2ADD	.F+	.FADD
Subtraction, unary -	.F2-	.F2SUB	.F-	.FSUB
Multiplication	.F2*	.F2MUL	.F*	.FMUL
Division	.F2/	.F2DIV	.F/	.FDIV
Less than	.F2<	.F2LT	.F<	.FLT
Greater than	.F2>	.F2GT	.F>	.FGT
Less than or equal	.F2<=	.F2LE	.F<=	.FLE
Greater than or equal	.F2>=	.F2GE	.F>=	.FGE

Table 19: double-precision real operators

Double extended	Form 1	Form 2		
Addition, unary +	.F3+	.F3ADD		
Subtraction, unary -	.F3-	.F3SUB		
Multiplication	.F3*	.F3MUL		
Division	.F3/	.F3DIV		
Less than	.F3<	.F3LT		
Greater than	.F3>	.F3GT		
Less than or equal	.F3<=	.F3LE		
Greater than or equal	.F3>=	.F3GE		

Table 20: double extended precision real operators

5 DECLARATIONS

5.1 Introduction

Syntax

```
declaration :
   ivb-declaration     ;
   struct-declaration  ;
   stream-declaration  ;
   kmap-declaration    ;
```

A declaration specifies the interpretation given to an identifier. A declaration that also reserves storage is a definition; a definition creates an object. A definition specifies the type of the object (3.7), and its storage duration and initialisation (3.5).

If the value of `StrictChecking` is greater than `0`, there must be exactly one declaration for every unique identifier[1] in a given scope (3.3) and namespace (3.4).

If the value of `StrictChecking` is 0, scalar variables inside a function do not require an explicit declaration (3.1.1). These objects are created when they are first written to, and are implicitly declared to be of type `var`, with automatic storage duration. No initialisation is defined for these objects, since they are created only when explicitly written to.

The general form of the declarations of all objects is the same. However, the declaration of a structure or stream may simply declare a new type, rather than an object, and these declarations are therefore listed separately in (5.5) and (5.6). The initialiser for a K-map has a unique form, and K-map declarations are therefore listed separately in (5.7).

5.2 Array dimensionality

Syntax

```
array-dimensions :
   array-dimensionsA
   array-dimensionsB

array-dimensionsA :
   dimensionA
   array-dimensionsA dimensionA

dimensionA :
   [ constant-assignment-expression(opt) ]

array-dimensionsB :
   [ comma(opt) constant-expression(opt) ]
```

When declaring an array (3.7.12), the dimensionality may be specified as part of the type, or following the object name, or both; see (3.7.12.2).

[1] The declaration for an identifier which is a function name occurs as part of the function definition.

The dimensionality may further be specified in two different forms. In the first form (array-dimensionA) the dimensionality is specified as a list, with each dimension in its own [] brackets. In the second form (array-dimensionB) (3.7.12.3), the dimensionality is specified as a comma-separated list in a single pair of [] brackets.

When declaring an array, the first dimension expression may be omitted if it can be found from an initialiser list:

```
int[]     c = {0, 1, 2, 3, 4, 5};        // c is int[6]
int[][2]  d = {{0,1}, {2,3}, {4,5}};     // d is int[3][2]
int[,2]   e = {{0,1}, {2,3}, {4,5}};     // e is int[3,2]
int[,3,4] f;                             // ERROR: no initialiser
```

Example 58

It is an error if any dimension expression other than the first is omitted.

5.3 Initialisation

Syntax

```
init-assignment :
   = initialiser

initialiser :
   assignment-expression
   { }
   { initialiser-list comma_opt }

initialiser-list :
   initialiser
   initialiser-list , initialiser

comma : ,
```

The initialisers for all objects (except K-maps; see (5.7)) have the same form, which is given by `init-assignment`.

An initialiser specifies the initial value of an object. If an object (or any sub-object within an aggregate) has no initialiser, then that object or sub-object is given a default initial value (3.6).

All the expressions in an initialiser for an object which has static storage duration (3.5) must be constant expressions.

The initialiser for a stream is always assigned automatically; if an explicit initialiser is given for a stream (or a stream in an aggregate) then that initialiser is ignored.

The initialiser for a scalar object must be a single expression, optionally enclosed in braces. The initialiser for a single K-map may also optionally be enclosed in braces.

The initialiser for a structure or array which has automatic storage duration must be a single expression that is an assignment-compatible aggregate (3.7.10.2 and 3.7.12.4), or an aggregate initialiser, as discussed below.

An aggregate initialiser is a brace-enclosed list of initialisers for the elements of the aggregate. The aggregate object is initialised in order, by assigning successive expressions from the initialiser list to successive element in the aggregate. Arrays are initialised in increasing subscript order (with the rightmost subscript cycling fastest), and structures are initialised in member declaration order.

If an aggregate contains sub-aggregates, then the initialiser list may omit initialisation of a sub-aggregate (by leaving the corresponding initialiser expression blank), or may specify a sub-aggregate initialiser in braces. Initialisation therefore occurs recursively down through any braces in the initialiser. An array of rank *n* therefore requires *n* brace levels for complete initialisation, if the array elements are scalar objects.

If the initialiser for an aggregate expires before the aggregate is completely initialised, then the remaining members of the aggregate are given default initialisations (3.6).

Examples

If aggregate sub-objects are to be initialised, the braces must be fully specified[1]:

```
1  struct x {
2     int a, b;
3  };
4
5  int main(void) {
6     int c[6] = {0, 1, 2, 3, 4, 5};                                    // Ok
7
8     struct x d[6] = {0,1, 2,3, 4,5, 6,7, 8,9, 10,11};                 // ERROR
9     struct x e[6] = {{0,1}, {2,3}, {4,5}, {6,7}, {8,9}, {10,11}};     // Ok
10
11    int f[2][3] = {0, 1, 2, 3, 4, 5};                                 // ERROR
12    int g[2][3] = {{0, 1, 2}, {3, 4, 5}};                             // Ok
13
14    struct x h[2][3] = {{0,1}, {2,3}, {4,5}, {6,7}, {8,9}, {10,11}};  // ERROR
15    struct x i[2][3] = {{0,1, 2,3, 4,5}, {6,7, 8,9, 10,11}};          // ERROR
16    struct x j[2][3] = {{{0,1}, {2,3}, {4,5}}, {{6,7}, {8,9}, {10,11}}}; // Ok
17    return 0;
18 }
```

Example 59

The initialisers for objects with static storage duration (external and static objects) must be constant expressions:

```
int foo(void) { return 41; }
bar() {
   static struct s1 {
      int x, y;
   } a = {foo(), 42};             // error: initialiser must be constant
   struct s1 b = {foo(), 42};     // Ok
}
```

Example 60

[1] Aggregate initialisation is, in practice, essentially identical to aggregate initialisation in C, except that conditions which are generally flagged as warnings in C compilers (such as misaligned braces) are reported as errors in Maia. For this example, gcc warns about missing braces on lines 8, 11, 14, and 15, while g++ warns about missing braces on lines 8, 11, and 15, and reports an error on line 14. Maia reports errors for all of lines 8, 11, 14, and 15.

5.4 int, bit, var, and bool

Syntax

```
ivb-declaration :
   storage-class_opt typespec-ivb object-list

storage-class :
   static

typespec-ivb :
   typemark-ivb array-dimensions_opt

typemark-ivb : one of
   int bit_n ubit var_n uvar bool

object-list :
   object-item
   object-list , object-item

object-item :
   identifier array-dimensions_opt init-assignment_opt
```

An ivb-declaration declares an object of a boolean or arithmetic type, or an array of these objects.

5.5 struct

Syntax

```
struct-declaration :
   struct-named-instance
   struct-tdecl
   struct-tdecl-with-objects

struct-named-instance :
   storage-class_opt typespec-struct object-list

typespec-struct :
   struct identifier array-dimensions_opt

struct-tdecl :
   struct { declaration-list_opt }
   struct identifier { declaration-list_opt }

declaration-list :
   declaration
   declaration-list declaration

struct-tdecl-with-objects :
   storage-class_opt struct-tdecl array-dimensions_opt object-list
```

A `struct-tdecl` declares a new structure (3.7.10) type. A `struct` is an aggregate type, and consists of a sequence of members, each of which may itself be of an arbitrary type. A `struct` may not, however, contain an instance of itself.

5.6 stream

Syntax

```
stream-declaration :
   stream-named-instance
   stream-tdecl
   stream-tdecl-with-objects

stream-named-instance :
   storage-class_opt typespec-stream object-list

typespec-stream :
   stream identifier array-dimensions_opt

stream-tdecl :
   stream { stream-defn-list_opt }
   stream identifier { stream-defn-list_opt }

stream-defn-list :
   stream-defn
   stream-defn-list stream-defn

stream-defn :
   mode   constant-expression  semicolon_opt
   file   string               semicolon_opt
   format string name-list_opt semicolon_opt

name-list :
   identifier
   name-list , identifier

stream-tdecl-with-objects :
   storage-class_opt stream-tdecl array-dimensions_opt object-list
```

A `stream-tdecl` declares a new stream (3.7.11) type. `struct` and `stream` declarations have the same form, except that a `stream` declaration contains the attributes of a named file, rather than a collection of members.

All three attributes (`mode`, `file`, and `format`) must be present in a stream declaration, and may occur in any order.

5.7 kmap

Syntax

```
kmap-declaration :
   storage-class_opt typespec-kmap kmap-object-list

typespec-kmap :
   kmap array-dimensions_opt

kmap-object-list :
   kmap-object-item
   kmap-object-list , kmap-object-item

kmap-object-item :
   identifier array-dimensions_opt kmap-initialiser_opt

kmap-initialiser :
   = kmap-init

kmap-init :
   kmap-const-list
   { }
   { kmap-init-list comma_opt }

kmap-const-list :
   kmap-constant
   kmap-const-list kmap-constant

kmap-constant
   constant
   kmap-const

kmap-const : one of
   x X z Z

kmap-init-list :
   kmap-init
   kmap-init-list , kmap-init
```

A `kmap-declaration` declares an object of type `kmap` (3.7.8), or an array of these objects.

A `kmap` initialiser is composed of a list of constants, rather than assignment expressions, and so has the form `kmap-initialiser` (rather than `init-assignment`). A `kmap` declaration otherwise has the same form as the declaration of an object of any other type.

For the purposes of initialisation, a `kmap` is regarded as a scalar object. The initialiser for this scalar is a `kmap-const-list`, which is a whitespace-separated list of constants or the characters `x`, `X`, `z`, or `Z`. Any constants in the list must have one of the values `0` or `1`.

6 STATEMENTS

6.1 Introduction

Syntax

```
statement :
   statementA
   statementB

statementA :
   compound-statement
   selection-statement
   iteration-statement
   jump-statement
   trigger-statement
   wait-statement
   exec-statement
   exit-statement
   assert-statement
   report-statement
   label : statementA

statementB :
   expression-statement
   drive-statement
   label : statementB

label : identifier
```

A statement specifies an action to be performed. Except where indicated otherwise, statements are executed in sequence.

Statements may optionally be preceded by an identifier and a ':', where the identifier *labels* the statement. The `default` label has special significance, and may not be used outside a `switch` statement. Labels may be used to disambiguate drive statements (6.8), but otherwise have no significance[1].

Statements are divided into two groups (`statementA` and `statementB`). This division has no significance, other than to define the statements which may follow an `if` statement or a `while` statement, when the controlling expression for that `if` or `while` is not enclosed in parentheses. If the parentheses are omitted, the following statement must be a `statementA`; if they are included, the following statement may be any `statement`.

Simulation time may be advanced only by executing a `wait` statement, or a `drive` statement; all other statements execute in zero time. Expression evaluation order is completely defined, and any expression which includes time-consuming function calls always has a defined result.

[1] C allows a label to be the destination of a `goto` statement.

6.2 Compound statement

Syntax

```
compound-statement :
      { block-item-list_opt }

block-item-list :
   block-item
   block-item-list block-item

block-item :
   declaration
   statement
```

A compound statement groups a set of declarations and statements together as a single syntactic unit.

6.3 Expression and null statements

Syntax

```
expression-statement :
   expression_opt ;
```

An expression statement is evaluated as a void expression for its side-effects.

A null statement (consisting of just a `;`) performs no operations.

6.4 Selection statements

The selection statements select between groups of statements depending on the value of a controlling expression.

Syntax

```
selection-statement :
   if   expression    statementA
   if   expression    statementA else statement
   if ( expression )  statement
   if ( expression )  statement  else statement
   switch expression { switch-body_opt }

switch-body :
   labelled-statement-switch
   switch-body labelled-statement-switch

labelled-statement-switch :
   case constant-expression : block-item-list
   default : block-item-list
```

Semantics

Parentheses may optionally be placed around the controlling expression if desired[1]. However, there is a potential parsing ambiguity for the `if` and `if-else` statements if the parentheses are omitted, and the associated statement is therefore required to be `statementA` (a subset of `statement`) in this case. In other words, if the parentheses are omitted, the associated statement may not be a single expression statement, or a single drive statement.

6.4.1 The if statement

The controlling expression must be of boolean type.

If the controlling expression evaluates true, the associated statement is executed; if it evaluates false, the associated statement is not executed.

6.4.2 The if-else statement

The controlling expression must be of boolean type.

If the controlling expression evaluates true, the first statement is executed; if it evaluates false, the second statement is instead executed.

The `else` clause is associated with the nearest lexically preceding `if`.

6.4.3 The switch statement

The controlling expression must be of arithmetic type.

There may be at most one `default` label in the switch statement. A `case` label expression must be a constant expression; the values of the `case` label expressions must all be unique within a given switch statement[2].

If there is no `default` label, and the value of the controlling expression does not match the value of any of the `case` labels, then no part of the switch body is executed.

The controlling expression and the `case` labels may be of different types if one is of type `var`, and the other is of type `int` or `bit`. In this case, the `case` label expression is converted to the type of the controlling expression, as if by assignment, before comparison with the value of the controlling expression.

[1] C requires parentheses here.

[2] If a `switch` statement itself includes one or more other `switch` statements, then those `switch` statements may themselves have `default` labels, and may have `case` label expressions which duplicate the expressions in the first `switch` statement.

6.5 Iteration statements

The `while`, `do`, and `for` iteration statements execute a loop under the control of a controlling expression. The `for all` statement executes a loop for all values of a control object.

Syntax

```
iteration-statement :
   while   expression   loop-bodyA
   while ( expression ) loop-body
   do loop-body while   expression ;
   for ( expression_opt ; expression_opt ; expression_opt ) loop-body
   for all identifier loop-body

loop-bodyA:
   statementA

loop-body:
   statement
```

Semantics

The controlling expression of the `while`, `do`, and `for` iteration statements must be of boolean type. These statements execute the loop body (*loop-bodyA* or *loop-body*) while the controlling expression is true; the iteration statement is terminated when the controlling expression evaluates false.

The `for all` iteration statement executes the loop body for all values of the control object, incrementing sequentially from `0`.

6.5.1 The while statement

The controlling expression is evaluated before each execution of the loop body.

Parentheses may optionally be placed around the controlling expression if desired[1]. However, there is a potential parsing ambiguity if the parentheses are omitted, and the loop body is therefore required to be `statementA` (a subset of `statement`) in this case. In other words, if the parentheses are omitted, the loop body may not be a single expression statement, or a single drive statement.

6.5.2 The do statement

The controlling expression is evaluated before each execution of the loop body. Parentheses may optionally be placed around the controlling expression if desired[2].

6.5.3 The for statement

The statement **`for(E1; E2; E3) loop-body`** is evaluated as follows:

[1] C requires parentheses here.

[2] C requires parentheses here.

1 `If` **`E1`** is present, it is evaluated once, as a void expression. **`E1`** is generally a loop initialisation operation.

2 **`E2`** is then evaluated; it is the controlling expression. If **`E2`** is omitted, it is given the value `true`. If **`E2`** evaluates false, execution continues at the statement after the `for` statement; if it evaluates true, the loop body is executed (step 3).

3 The loop body is executed; if **`E3`** is present, it is then evaluated as a void expression. Execution then resumes at step 2.

A `continue` statement in the loop body branches to a point just before **`E3`**; in other words, **`E3`** is always evaluated after a `continue`.

6.5.4 The for all statement

The statement **`for all identifier loop-body`** executes the associated loop body for all values of the identifier, starting at `0`. The identifier must name an arithmetic object, or a mode 1 stream. If the identifier names an arithmetic object, the `for all` statement is equivalent to:

```
identifier = 0;
do {
   loop-body
} while(++identifier != 0);
```

Example 61

The `for all` statement is primarily useful for iterating over all values of a 'small' variable[1], and avoids the complexity of handling the wrap-around of the variable, and the use of signed or unsigned comparisons in the equivalent loop control expression.

If the identifier names a mode 1 stream, the `for all` statement is equivalent to:

```
identifier = 0;
do {
   loop-body
} while((++identifier)'offset != 0);
```

Example 62

The `for all` statement may therefore be used to iterate over all lines of a mode 1 stream (3.7.11.1.7).

6.6 Jump statements

A jump statement causes an unconditional jump.

Syntax

```
jump_statement :
   continue constant-expression_opt ;
   break    constant-expression_opt ;
```

[1] This might be useful, for example, if it is necessary to apply all values of an 8-bit variable to a DUT. Care should be taken not to use a 'large' variable as the loop control variable; the number of loop iterations is $2^{identifier`size}$.

```
return   expression_opt ;
```

6.6.1 The continue statement

The `continue` statement causes a jump to the loop-continuation portion of an enclosing iteration statement. The `continue` has an associated *level*, which is given by the optional constant expression, and which specifies which enclosing iteration statement should be continued.

If the level expression is omitted, it defaults to 1. A one-level `continue` jumps to the end of the loop body of the closest enclosing iteration statement[1]. `continue 2` jumps to the end of the loop body of the next enclosing iteration statement, and so on. It is an error if the `continue` level is less than 1, or greater than the number of enclosing iteration statements[2].

Examples

The "loop-continuation portion" of an iteration statement is an implicit null statement at the end of the loop body. This null statement is jumped to by a `continue`:

```
while(a()) {
   if(b())
      continue;       // jumps to label1
   c();
   label1: ;
}

do {
   if(b())
      continue;       // jumps to label2
   c();
   label2: ;
} while(a());

for(;;) {
   if(b())
      continue;       // jumps to label3
   c();
   label3: ;
}

for all x {
   if(b())
      continue;       // jumps to label4
   c();
   label4: ;
}
```

Example 63

[1] A `continue` which has no level specified therefore has the same behaviour as C's `continue` statement.

[2] It is therefore an error if the `continue` statement does not appear inside an iteration statement.

6.6.2 The break statement

The break statement causes termination of an enclosing `switch` or iteration statement. The `break` has an associated level, which is given by the optional constant expression, and which specifies which enclosing `switch` or iteration statement should be terminated.

If the level expression is omitted, it defaults to 1. A one-level `break` terminates the closest enclosing `switch` or iteration statement[1]. `break 2` terminates the next enclosing `switch` or iteration statement, and so on. It is an error if the `break` level is less than 1, or greater than the number of enclosing switch and iteration statements[2].

Examples

This code shows an example of a multi-level `continue`, and a multi-level `break`:

```
for(;;) {
   while(true) {
      if(foo())
         continue;    // equivalent to continue 1; jumps to label jumpA
      if(foo())
         continue 2;  // jumps to label jumpB
      if(foo())
         break 2;     // breaks 2 levels; jumps to label jumpC
         jumpA: ;
   }
   if(foo())
      break;          // equivalent to break 1; jumps to label jumpC
      jumpB: ;
}
jumpC: ;
```

Example 64

6.6.3 The return statement

A `return` statement terminates execution of the current function and returns to the caller. Any number of `return` statements may appear in a function.

The optional return expression may be used to return a value to the caller. The expression is returned as if by assignment to a temporary object which has the declared type of the function; the value of the function is the value of this temporary object. It is an error if a function which has been declared to be of `void` type contains any return statements with an associated return expression.

If a function has a non-void type, and control is returned to the caller by reaching the terminating `}` or by executing a `return` statement with no associated return expression, then the value returned to the caller will be the current value of the predefined `result` variable. Every non-`void` function has an implicit `result` variable, which has the same type as the function itself. The `result` variable is default-initialised (3.6) when the function is entered.

[1] A `break` which has no level specified therefore has the same behaviour as C's `break` statement.

[2] It is therefore an error if the `break` statement does not appear inside a `switch` or iteration statement.

Examples

```
int f1(void) {
   result = 2;          // f1() returns 2
}
int f2(void) {
   result = 2;
   return 4;            // f2() returns 4
}
int f3(void) {
   struct s1 res = f4();
   assert(res.a == 4'b0000 && res.b == 4'bxxxx);
   return;              // f3 returns 0 (the default value of result, of type int)
}
struct s1 {
   int4 a;
   var4 b;
}
struct s1 f4(void) {}
```

Example 65

6.7 Trigger statement

Syntax

trigger-statement :
trigger *postfix-expression* **(** *argument-expression-list*$_{opt}$ **)** *trigger-condition*$_{opt}$ **;**

trigger-condition :
trigger-count expression

trigger-count : one of
when
when all

The trigger statement posts a trigger function (7.6) for later execution. The `postfix-expression` must denote a trigger function; the `()` parentheses contain a possibly empty comma-separated list of expressions. These expressions form the actual parameters to the trigger function; the number of arguments must agree with the number of formal parameters to the function. The actual parameters are sampled when the trigger statement is executed; the sampled values are stored, and are supplied to the formal parameters, as if by assignment, when the trigger function starts execution.

The trigger function starts execution when the expression supplied in `trigger-condition` is sampled true. This expression must be of boolean type. Sampling occurs on the edges of the sample clock defined by the triggered drive declaration for the function (9.2.3).

If `trigger-count` is specified as `when`, the trigger function executes only once, when the trigger condition is first sampled true.

If `trigger-count` is specified as `when all`, the trigger condition automatically re-arms when the trigger function completes execution; the condition is then checked on subsequent sampling clocks. A run-time error is reported if the trigger condition again becomes true while the function is running; it is not possible to run multiple instances of the same trigger function.

6.8 Drive statement

Syntax

```
vfile-drive-statement :
   base-drive-statement

drive-statement :
   base-drive-statement
   triggered-drive-statement

base-drive-statement :
   [ hdl-inputs ]
   [ hdl-inputs ] -> pipe-level_opt [ hdl-outputs ]

triggered-drive-statement :
   -> [ hdl-outputs ]

pipe-level :
   constant
   identifier
   ( expression )

hdl-inputs :
   hdl-expression-list

hdl-outputs :
   hdl-expression-list

hdl-expression-list :
   hdl-expression
   hdl-expression-list , hdl-expression

hdl-expression :
   assignment-expression
   drive-directive_opt

drive-directive : one of
   - .c .C .x .X .z .Z .r .R
```

The syntax of the `base-drive-statement` is shown only for procedural programs (`drive-statement`), and not for testvector programs (`vfile-drive-statement`), for simplicity (1.1). For a testvector program, an `hdl-expression` must be a `constant-assignment-expression` or a directive, rather than an `assignment-expression` or a directive.

The optional `pipe-level` specifies the expected number of pipeline levels (9.2.4) on the `hdl-outputs`; it defaults to `1` if it is omitted[1]. The level may be an arbitrary expression if required (and so may change at runtime). If the level is not a constant or an identifier, it must be enclosed in parentheses `()` to avoid parsing ambiguities.

A `drive-directive` (9.3) is a single character which specifies an action on an input, an output, or both. An empty directive is a don't-care condition, and is equivalent to '–'.

The drive statement is described in (9).

[1] In other words, `hdl-inputs` set up to a clock edge, and `hdl-outputs` are generated by the same clock edge.

6.9 Wait statement

Syntax

```
wait-statement :
   wait expression ;
```

The `wait` statement causes the currently-executing function to pause execution for the time given by the expression. The expression is interpreted as a floating-point number[1], in the timescale units specified in the DUT section (which default to nanoseconds).

`wait` statements may not be used in trigger functions.

6.10 Exec statement

Syntax

```
exec-statement :
   exec  function-name ( argument-expression-list ) ;

function-name : identifier
```

The `exec` statement creates a new thread (10.5), and initiates execution of the named function in that thread. The statement returns immediately (in zero simulation time), and the newly-created thread starts execution immediately.

argument-expression-list must contain at least one actual parameter. The first actual must be the name of an `int` object, which is passed by reference to the new thread function. The new Thread ID is returned to the object.

6.11 Exit statement

Syntax

```
exit-statement :
   exit expression_opt ;
```

The `exit` statement terminates program execution; the expression is an exit code. The exit code may, or may not, be returned to the operating system, depending on the code generator (14.7.4).

6.12 Assert statement

Syntax

```
assert-statement :
   assert expression ;
   assert expression report-statement
```

[1] Even if it has no decimal point; see (4.6.1.1)

The assert expression is required to be of boolean type. If the expression evaluates false, an error is reported on stdout and in the log file (if output is enabled); the error message includes the source file name and line number of the failing `assert` statement.

The `assert` expression may optionally be followed by a `report` statement. If the `report` statement is present, the `report` message is included as part of the assertion failure output.

The number of assertion failures which are required to terminate a program is set by a compiler switch; see (14.5).

6.13 Report statement

Syntax

```
report-statement :
   report   printf-varargs ;
   report ( printf-varargs ) ;

printf-varargs :
   string
   string , pv-list

pv-list :
   pv-element
   pv-list , pv-element

pv-element :
   assignment-expression
   string
```

The `report` statement provides formatted output to the console (stdout). The first argument is a *format* string, which specifies how subsequent arguments are converted for output. The number of supplied arguments must match the number expected for the format.

The format is a character sequence, which is composed of zero or more ordinary characters (not `%`), which are copied unmodified to stdout, and *conversion specifications*. The conversion specifications result in the fetching of zero or more arguments, which are written to stdout.

`report` is broadly compatible with C's `fprintf`, with the exceptions noted in (6.13.3). However, 2019.9 relies on the Verilog simulator to generate output, and different simulators have widely different support for formatted output. It is likely that there will be some deviation from this specification, depending on which simulator is used (14.7.2).

A conversion specification is introduced by the character `%`. After the `%`, the following appear, in sequence:

1. zero or more *flags*, in any order, which modify the meaning of the conversion specification. The **+**, **-**, **' '** (space), **#**, and **0** flags are recognised, but are not implemented in 2019.9; a warning is issued if these flags are detected.

2. An optional minimum *field width*. If the output has fewer characters than the field width, it is padded with spaces on the left. The field width must be a decimal integer.

3. An optional *precision* that gives the minimum number of digits to appear for the **d**, **i**, **b**, **o**, **u**, **x**, and **X** conversions, the number of digits to appear after the decimal point character for **a**, **A**, **e**, **E**, and **f** conversions, the maximum number of significant digits for the **g** and **G** conversions, or the maximum number of output characters for **s** conversion. The precision takes the form of a period (**.**) followed by an optional decimal integer; if the integer is omitted, it defaults to `0`.

4. An optional *length modifier* that specifies the size of the argument.

5. A *conversion specifier* character that specifies the form of the output.

6.13.1 Length modifiers

Length modifiers are required only for arguments which should be interpreted as floating-point numbers. The length modifiers and their meanings are (where a "real" conversion specifier is one of **f**, **e**, **E**, **g**, **G**, **a**, or **A**):

F Specifies that a following real conversion specifier applies to a single-precision float

D Specifies that a following real conversion specifier applies to a double-precision float

L Specifies that a following real conversion specifier applies to an extended double-precision float

If the length modifier is omitted, and the following conversion specifier is a real conversion specifier, then the length modifier defaults to **D**. The Verilog code generator does not support the **F** and **L** length modifiers, and an error is reported if they are used.

6.13.2 Conversion specifiers

The conversion specifiers are listed below. The argument corresponding to the specifier must be an arithmetic object, unless noted otherwise.

s The argument must be a string. If the precision is specified, no more than that many characters are output.

t The time coded in the argument is output as an integer; the current time may be output by using `_timeNow` as the argument.

T The time coded in the argument is output as a float; the current time may be output by using `_timeNow` as the argument.

l The argument must be boolean; it is output as the string `true`, or the string `false`.

d,i The argument is assumed to be in 2's complement form, and is output as signed decimal; a leading **-** sign is output if necessary. The precision specifies the minimum number of digits to output; if the argument can be specified in fewer digits, it is expanded with leading zeros. The default precision is 1.

b,o,u, The argument is output as unsigned binary (**b**), unsigned octal (**o**), unsigned decimal

x,X (**u**), or unsigned hexadecimal (**x** and **X**). The letters **abcdef** are used for **x** conversion, and **ABCDEF** for **X** conversion. The precision specifies the minimum number of digits to output; if the argument can be specified in fewer digits, it is padded with leading zeros. The default precision is 1.

f The argument is assumed to be floating-point, and is output in decimal in the style *[–]ddd*.*ddd*, where the number of digits after the decimal point is equal to the precision specification. The precision defaults to 6 if it not specified; if the precision is 0, no decimal point character appears. If a decimal point character is output, at least one digit will appear before it. The value is rounded to the appropriate number of digits.

An argument representing an infinity is output as either *[–]***inf** or *[–]***infinity**, depending on the simulator. An argument representing a NaN is output as *[–]***nan** or *[–]***nan(***n-char-sequence***)**, depending on the simulator.

e,E The argument is assumed to be floating-point, and is output in decimal in the style *[–]d*.*ddd* **e**±*dd*, where there is one digit (which is nonzero if the argument is nonzero) before the decimal-point character. the number of digits after the decimal point is equal to the precision specification. The precision defaults to 6 if it not specified; if the precision is 0, no decimal point character appears. The value is rounded to the appropriate number of digits.

The **E** conversion specifier outputs a number with **E** instead of **e** introducing the exponent. The exponent always contains at least two digits, and only as many more digits as necessary to represent the exponent. If the value is zero, the exponent is zero.

An argument representing an infinity or NaN is output in the same style as the **f** conversion specifier.

g,G The argument is assumed to be floating-point, and is output in style **f** or **e** (or in style **f** or **E** in the case of a **G** conversion specifier). The style used depends on the value converted; style **e** (or **E**) is used only if the corresponding exponent is less than −4 or greater than or equal to the precision.

The precision specifies the number of significant digits. If the precision is zero, it is taken as 1. Trailing zeros are removed from the fractional portion of the result; a decimal-point character appears only if it is followed by a digit.

An argument representing an infinity or NaN is output in the same style as the **f** conversion specifier.

a,A The argument is assumed to be floating-point, and is output in the style *[–]***0x***h*.*hhhh* **p**±*d*. There is one hexadecimal digit (which is nonzero if the argument is a normalized floating-point number and is otherwise unspecified) before the decimal-point character, and the number of hexadecimal digits after it is equal to the precision. If the precision is 0, no decimal point character appears.

The letters **x**, **p**, and **abcdef** are used for the **a** conversion, while the letters **X**, **P**, and **ABCDEF** are used for the **A** conversion.

The exponent always contains at least one digit, and only as many more digits as necessary to represent the decimal exponent. If the value is zero, the exponent is zero.

An argument representing an infinity or NaN is output in the same style as the **f** conversion specifier.

c The argument is output as a character.

% A % character is output; no argument is consumed.

6.13.3 fprintf compatibility

The conversion specifications are modelled on the conversion specifications defined for the C fprintf function[1]. However, report and fprintf currently differ in the following areas:

a) The flags are not implemented in 2019.9

b) Length modifiers are not required for arithmetic objects which are to be output as integers; the fprintf **hh**, **h**, **l**, **ll**, **j**, **z**, **t**, and **L** length modifiers are therefore unused. The **l** and **t** modifiers are reused as conversion specifiers; the remaining modifiers are reported as errors

c) The fprintf **p** and **n** conversion specifiers are not required

d) The **t**, **T**, **l**, and **b** conversion specifiers are added, to handle integer time, floating-point time, logical, and binary output, respectively

e) The fprintf ***** field width, and ***** precision, are not supported.

[1] ISO/IEC 9899:1999 (E), §7.19.6.1

7 FUNCTIONS

7.1 Introduction

Maia supports both *user functions* (7.5) and *trigger functions* (7.7). User functions are conventional functions which are executed as part of the normal user-initiated sequential program flow, while trigger functions are run automatically, in response to defined *trigger conditions*. Both user and trigger functions may execute in either zero or non-zero simulation time.

User function calls are primary expressions; they may appear anywhere where an expression may appear. User functions (of a non-`void` type) always have a value, whether or not they explicitly return data. A user function which returns nothing has the default value of the `result` variable (3.6).

Trigger functions are posted for later execution using *trigger statements* (6.7). A trigger statement specifies a set of conditions (normally DUT outputs) which are examined on every controlling clock edge, and the function, together with its actual parameters, which will be called when the condition is found to be true. A return statement in a trigger function simply terminates execution of that function; the function may not return data, and may not assign to `result`, and has no value.

User functions may be run in a new thread using *exec statements* (6.10). Any function initiated in this way is referred to as a *thread function* (7.6). Thread functions are simply user functions which have been initiated with an exec statement, but they have a number of restrictions which do not apply to general user functions, and so are documented separately here.

All function names are globally visible. If a program includes any functions, then one of these functions must be a user function named `main`. The `main` function is the program entry point; `main` should have no parameters in 2019.9. `main` may return a value, but this value may, or may not, be returned to the operating system (14.7.4). A program may be terminated only by returning from `main`, or by calling `exit`.

7.2 Syntax

```
function-definition :
   user-function-definition
   thread-function-defintion
   trigger-function-definition

user-function-definition :
   user-function-typespec_opt
      function-name ( formal-list_opt ) { sf-block-item-list_opt }

user-function-typespec :
   typespec-ivb
   typespec-struct
   typespec-stream
   typespec-kmap

thread-function-definition :
   void function-name ( formal-list ) { sf-block-item-list_opt }
```

```
trigger-function-definition :
   @ function-name ( formal-list_opt ) { tf-block-item-list_opt }

function-name : identifier

formal-list :
   void
   formal-item-list

formal-item-list :
   formal-item
   formal-item-list , formal-item

formal-item : formal-typespec &_opt identifier array-dimensions_opt

formal-typespec : user-function-typespec

sf-block-item-list : block-item-list

tf-block-item-list : block-item-list
```

The `user-function-typespec` may be omitted if the `_StrictChecking` level is 0 or 1 (3.1.3). In this case, the function return type is `uvar`. The `formal-typespec` may similarly be omitted if the `_StrictChecking` level is 0.

`sf-block-item-list` and `tf-block-item-list` are both shown as `block-item-list`, for simplicity. However, there are a number of differences between the statements which may be used in user and trigger functions; see (7.7).

7.3 Parameter passing semantics

The mechanism by which a parameter is passed to a function is determined by the presence or absence of an `&` (ampersand) character preceding the formal identifier. If no ampersand is present, the function receives a copy of the current value of the argument ('call by value'). If an ampersand is present, the function instead receives a reference to the argument ('call by reference'). This reference may be used to modify the value of the object in the calling function, as illustrated by the 'swap' functions in Example 66 below.

In 2019.9 parameters may only be passed by reference to a function if that function is not time-consuming (10.4); an error will be reported if an attempt is made to use call-by-reference with a time-consuming function.

When a stream object is passed to a function, the function receives a *handle* to that stream. Passing a stream by value therefore has exactly the same effect as passing that stream by reference, and any use of '`&`' is redundant.

```
main() {
   int i = 1;
   int j = 2;

   swap_val(i, j);
   report("i is %d; j is %d\n", i, j); // reports 'i is 1; j is 2'
```

```
   swap_ref(i, j);
   report("i is %d; j is %d\n", i, j); // reports 'i is 2; j is 1'
}

void swap_val(int a, int b) {
   int temp = b;
   b = a;
   a = temp;
}

void swap_ref(int& a, int& b) {
   int temp = b;
   b = a;
   a = temp;
}
```

Example 66

7.4 Function signatures

Function names may be overloaded, and are identified, or disambiguated, by the number of parameters to the function. A function signature is made up of the name of the function, together with the number of formal parameters, using the notation `name{nparams}`. All function signatures must be unique within a program.

```
main() {
     var a;
     ...
     test();                              // call function test{0}
     trigger test(a) when DATA_READY;     // post test{1} for later execution
}

test() {                                  // this is test{0} (0 formal parameters)
     report("test called\n");
}

@test(x) {                                // this is test{1} (1 formal parameter)
     report("DATA_READY active; 'x' is %u\n", x);
}
```

Example 67

7.5 User functions

A user function returns no value if it has a `void` type, or one value otherwise. It may have zero or more input parameters. A user function with a non-`void` return type returns a value either with a `return` statement, or by assigning to the predefined `result` variable. If a function returns by using a `return` statement with no expression, or by reaching the terminating `}`, then the value returned will be the last value assigned to `result`. `result` is default-initialised (3.6) when the function is entered, so all user functions have a defined default return value.

Functions do not need to be declared before use. The definition of a function also serves as its declaration.

Recursion is not supported in 2019.9 (14.7.5).

7.6 Thread functions

A Thread function is any user function which is named as the target of an `exec` statement. If a function name appears as the target of an `exec`, then it cannot be executed 'conventionally', by using its name as part of an expression; it can only be executed as an `exec` target.

While user and thread functions are syntactically and semantically identical, there are a number of usage restrictions for thread functions.

Thread functions do not return to the caller, and so must be declared with a `void` return type. There must be at least one formal parameter, and the first formal must be a reference to an integer. When the function starts execution, this parameter will contain the new thread ID. This thread ID is also returned to the caller:

```
void main() {
   int tid;
   exec f1(tid);
   report("%t: started thread %d\n", _timeNow, tid); // "1 ns: started thread 1"
}

void f1(int& tid) {
  report("%t: in thread %d\n", _timeNow, tid);        // "1 ns: in thread 1"
}
```

Example 68

7.7 Trigger functions

Trigger functions are essentially 'clocked' functions. They are automatically initiated when the run-time detects a defined trigger condition at a clock edge. The clock itself must be generated elsewhere (normally by a `drive` statement in a user function). Trigger functions use a special form of drive statement (triggered-drive-statement). These drive statements have no inputs, since they are responsible only for testing outputs in response to a clock edge:

```
 ->[out1, out2, ...outn];       // triggered drive statement: no inputs
```

Trigger functions are syntactically and semantically identical to user functions, apart from the following differences:

1. The name of a trigger function is preceded by an `@` character in its definition
2. Trigger functions may only be *posted* for later execution via a trigger statement (6.7); they cannot be 'called' in the conventional way. Trigger functions therefore have no value and cannot be used in expressions. The parameters to a trigger function are sampled when the function is posted, and not when the function eventually starts execution.
3. Trigger functions may not return a value; it is an error to use the `result` variable or to return an expression

4 Trigger functions may not execute `wait` statements, and may not post trigger functions; user functions can do both

5 User functions may use any sequential drive statement, but may not use any triggered drive statements. Trigger functions may only use one drive statement; this is the appropriate triggered drive statement declared in the DUT section (`triggered-drive-declaration`)

6 A user function may call any other user function. A trigger function may only call user functions that:

- execute in zero time (in other words, are not time-consuming), and
- do not execute trigger statements

A trigger function may advance time only by executing the drive statement associated with that trigger function. When a trigger function is initiated, it may execute zero or more of these drive statements before terminating.

7.8 Inter-function communication

Concurrent functions may communicate with each other through the use of external variables. A read-modify-write operation on an external variable is guaranteed to be atomic, as long as it includes no suspending statements (10.1).

8 DUT SECTION

8.1 Introduction

Maia communicates with an external HDL program which describes the DUT (Device Under Test). In order to carry out this communication, Maia requires some information about the DUT, and about any *test vectors* (drive statements) which will be used to test the DUT. This information is placed in the *DUT section*.

A DUT section (DUT-definition) is only required if there are drive statements in the program. There may be a maximum of one DUT section, which may appear anywhere where a function is permissible.

Syntax

```
DUT-definition :
   DUT { dut-declaration-list_opt }

dut-declaration-list :
   dut-declaration
   dut-declaration-list dut-declaration

dut-declaration :
   module-declaration              semicolon_opt
   sequential-drive-declaration    semicolon_opt
   triggered-drive-declaration     semicolon_opt
   dut-signal-declaration          semicolon_opt
   clock-declaration               semicolon_opt
   enable-declaration              semicolon_opt
   timescale-declaration           semicolon_opt
   timing-constraint               semicolon_opt
```

If the DUT section is present, and the program contains drive statements, then the DUT section must include:

1. one module declaration
2. one or more drive declarations (sequential or triggered)
3. zero or more internal DUT signal declarations
4. zero or more clock declarations
5. zero or more enable declarations
6. zero or one timescale declarations
7. zero or more timing constraint declarations

The DUT section must contain at least one clock declaration (8.5) if any clocked logic is to be tested. If it is only necessary to carry out delta-delay simulations on rising-edge clocked logic, then no timescale or timing constraint declarations are required, and the default clock waveform is sufficient. If it is necessary to carry out delta-delay simulations on falling-edge clocked logic, then the default waveform must be replaced with one that has a falling edge before a rising edge (8.5.3).

DUT section declarations may appear in any order; declarations may optionally be semicolon-terminated. DUT declarations have exactly the same lexical structure as the rest of a Maia program, with the following exceptions:

a) If a module declaration contains a list of parameter values, then the parameter text (modparam-text; everything between the `#(` and `)` terminators) is not analysed, and is duplicated exactly in the testbench output

b) Any Verilog-2001 attributes (attribute) are ignored and copied directly to the testbench output, without analysis

c) Adjacent strings are not concatenated in a DUT section; every occurrence of a sequence of characters between double-quote characters (") is a separate token.

The identifiers in a DUT section (with the exception of labels) must be valid identifiers for the target language (dotted-identifier), since they will be duplicated exactly in the testbench output. If a target identifier (a module or port name, for example) is not a valid Maia identifier, then it must be enclosed in double quotes in the DUT section; for example:

```
DUT {
     module "\VHDL.Extended.Ident\" (input A, B; output C);
     [A, B] -> [C];
}
```

Example 69

8.2 Module declaration

A *module declaration* provides the name of the DUT, and the names, sizes, and directions of its ports. The declaration may also be used to provide the values of any parameters required by the DUT.

If the DUT has a Verilog 2001-style module definition[1], then that definition may normally be cut-and-pasted directly into the Maia code. This Verilog definition of a FIFO module, for example, may be entered directly, with no changes, as a Maia module declaration:

```
DUT {
   module fifo                        // unmodified Verilog module definition; reused
      (input [7:0] in,                // as a Maia module declaration
      input clk, read, write, reset,
      output [7:0] out,
      output full, empty);
   ...
}
```

Example 70

Maia understands Verilog port declarations, and ignores the information that it doesn't require (the `signed`, `reg`, `integer` and `time` keywords, the various `net_type` keywords, attributes, and port initialisers). This code implicitly declares 8 external variables which may be used anywhere in the Maia code, as if the following explicit external declarations had been made:

[1] Often (incorrectly) known as an 'ANSI-C' style definition

```
var8 in, out;
var1 clk, read, write, reset, full, empty;
```

If no Verilog module definition is available (the code is VHDL, for example), or if the module definition is in a pre-2001 form, then it will be necessary to derive a 2001-style equivalent to place in the Maia code (8.2.3).

If the module definition is parameterised, or if it is necessary to pass parameter values into the instantiated module (to override default parameter values in that module), then the module definition will require some modification before being re-used as a Maia module declaration. No changes to the HDL source code are required.

8.2.1 Parameterised modules

Consider, for example, this Verilog module definition:

```
module generic_fifo
     #(parameter MSB=3, LSB=0, DEPTH=6)
      (input [MSB:LSB] in,
       input clk, read, write, reset,
       output [MSB:LSB] out,
       output full, empty);
```

Example 71

This is a generic FIFO module, which defaults to 4-bit input and output buses, and a depth of 6 words. A generic FIFO cannot be tested; a *specific instance* of that FIFO must be tested. The Maia declaration is essentially an *instantiation* of a specific instance. To create the instantiation, two things must be done:

1. any parameterised port sizes must be replaced with known sizes;
2. any parameters required in the module must be passed into the module.

If it is necessary to test `generic_fifo` with 16-bit ports, and an 8-word depth, then the following module declaration can be used:

```
DUT {
   module generic_fifo
     #(.MSB(15), .LSB(0), .DEPTH(8))
      (input [15:0] in,                  // must use '15:0', not 'MSB:LSB'
       input clk, read, write, reset,
       output [15:0] out,
       output full, empty);
   ...
}
```

Example 72

The mechanism used to assign parameters is identical to Verilog's "module instance parameter value assignment", which is the preferred way to assign values to module parameters in Verilog-2001.

An `@` (U+0040) character may be used to introduce the parameter list, rather than the `#` character, if preferred. This may be necessary if an external preprocessor is used[1]. If the `@(` syntax is used, then

[1] `#(` will generate an error in a C-compatible preprocessor.

there must be no whitespace between the two characters; if the **#(** syntax is used, then whitespace may be inserted between the two characters.

The entire parameter list is copied verbatim to the testbench output, with *no* analysis. The list may contain anything which is acceptable to the Verilog simulator. This example shows named association; ordered list assignment may be used if preferred.

8.2.2 Module declaration error checking

Maia does not analyse the DUT HDL source code, and so cannot confirm that there is no error in the module declaration. There are a number of potential errors which will only be caught by the Verilog simulator, which may produce a cryptic error message. This will happen in the following cases:

a) the module declaration contains an incorrect name, port size, direction, or parameter list

b) any signal declarations contain incorrect names, port sizes, or directions

Note, however, that port length mismatches may not be reported as errors by the Verilog simulator.

8.2.3 Module input, output, and inout declarations

A module port list is a list of `input`, `output`, and `inout` declarations. The list must be enclosed in parentheses, and individual items must be separated by either commas or semicolons[1].

This list (`list-of-port-declarations`) is also required for signal declarations (the 'ports' are actually internal signals for signal declarations, but the syntax is identical). The list is identical to Verilog's `list_of_port_declarations`[2], with the exceptions that the Verilog definition has been refactored to remove ambiguities, semicolon separators have been added (for compatibility with some Verilog tools), and the `@` syntax is added for parameter lists.

The full `list-of-port-declarations` is parsed and checked, but the items which are not required are ignored (the attributes, modifiers, initial assignments, and so on).

Some simple examples of module declarations are given below.

```
module test1(
       input [7:0] ina, inb,        // two 8-bit input ports
       input C,                     // a 1-bit input port
       output [32:1] Q,             // a 32-bit output port
       inout d, e, f, g)            // four 1-bit bidirectional ports
module test2(output Q; input D)    // 1-bit input, 1-bit output
module test2(output Q, input D);   // the same; terminating semicolons are optional
```

Example 73

[1] The Verilog LRM requires commas in port lists, but some tools also support semicolons, due to historical confusion in the LRM.

[2] IEEE Std 1364-2005, 12.3.4

Note that a comma-separated list of names shares the same direction and port size, which appears at the beginning of a sub-list; semicolons are illegal in this sub-list. Semicolons may only be used to separate different groups of declarations (where a new `input`, `output`, or `inout` keyword appears).

8.2.4 Syntax

```
module-declaration :
   attribute_opt module module-identifier module-params_opt list-of-port-declarations

module : one of
   module macromodule

module-identifier : videntifier

module-params :
   @( modparam-text )
   #( modparam-text )

list-of-port-declarations : ( port-declaration-list_opt )

port-declaration-list :
   port-list-first-item
   port-declaration-list port-list-next-item

port-list-first-item :
   attribute_opt inout  iodecl-modifiers_opt port-identifier
   attribute_opt input  iodecl-modifiers_opt port-identifier
   attribute_opt output iodecl-modifiers_opt port-identifier

port-list-next-item :
   , port-list-first-item
   ; port-list-first-item
   , port-identifier

port-identifier :
   videntifier
   videntifier = constant-expression

videntifier :
   dotted-identifier
   string

iodecl-modifiers :
   iodecl-modifier
   iodecl-modifiers iodecl-modifier

iodecl-modifier :
   range
   modifier

range: [ constant-expression : constant-expression ]

modifier : one of
   integer reg signed time supply0 supply1 tri tri0 tri1 triand trior uwire
   wand wire wor
```

```
attribute :          see below
dotted-identifier :  see below
modparam-text :      see below
```

The definitions of `attribute`, `dotted-identifier`, and `modparam-text` have been omitted for simplicity. `attribute` is an optional Verilog-2001 attribute, while `dotted-identifier` is a Verilog dotted identifier. `modparam-text` contains the entire contents of the parameter list, which is copied verbatim to the output testbench, without analysis.

8.3 Drive declaration

A drive declaration defines the format of any test vectors which will be used in the body of the code. The declaration simply lists the signals which are to be driven on the left-hand-side (LHS) of a test vector, and the signals which are to be tested on the right-hand-side (RHS) of a test vector. A 'signal', in this context, means either a DUT port, or an internal signal within the DUT.

This example (a complete testvector-program) shows a simple declaration, and some drive statements which use that declaration:

```
DUT {
     module test1(input D1, D2, CLK; output Q)
     create_clock CLK
     [D1, D2, CLK] -> [Q]    // the drive declaration
}
[0, 1, .C] -> [0]            // drive statement 1
[1, 0, .C] -> [1]            // drive statement 2
```

Example 74

The declaration is required to allow Maia to determine that, in the first clock cycle, signals `D1` and `D2` should be driven with `0` and `1` respectively, signal CLK should be driven with a default clock waveform, and signal Q should be tested against `0`.

Any signals used in a drive declaration must themselves be declared elsewhere as a DUT port (8.2.3), or as an internal DUT signal (8.4). Any signals on the LHS of a drive declaration must have an `input` or `inout` direction; any signals on the RHS must have an `output` or `inout` direction.

8.3.1 Syntax

```
sequential-drive-declaration :
   drive-declaration
   identifier : drive-declaration

triggered-drive-declaration :
   @ identifier drive-declaration
   @ identifier { constant-expression_opt } drive-declaration

drive-declaration :
    [ hdl-inputs-decl ]
    [ hdl-inputs-decl ] -> [ hdl-outputs-decl_opt ]

hdl-inputs-decl : videntifier-list

hdl-outputs-decl : videntifier-list
```

```
videntifier-list :
  videntifier
  videntifier-list , videntifier
```

8.3.2 Clocked and combinatorial drive declarations

A drive declaration is defined as a *clocked* drive declaration[1] if it includes a named signal on the LHS which is defined elsewhere as a clock (8.5); it is otherwise *combinatorial*.

8.3.3 Sequential and triggered drive declarations

The drive statements used in user functions (such as `main`) have a slightly different format to the drive statements used in trigger functions. If any drive statements are to be used in a user function, they must be declared as a *sequential* drive declaration (sequential-drive-declaration) in the DUT section. If any drive statements are to be used in a trigger function, they must be declared as a *triggered* drive declaration (triggered-drive-declaration) in the DUT section.

For the sequential form, the LHS must contain at least one signal. The entire RHS is optional; it may be omitted if the drive statement is expected to test nothing (if it used simply for internal DUT state preload, for example). Sequential drive declarations may be either clocked or combinatorial (8.3.2).

The triggered form must be preceded by the name of the corresponding trigger function, including the `@` character. If there is any ambiguity in the function name, a complete signature should be provided (with the number of parameters in `{}` braces). There must be exactly one signal on the LHS, and that signal must be declared as a clock (a triggered drive declaration is therefore also a clocked drive declaration).

Some examples of drive declarations and drive statements (6.8) are:

```
DUT {
     ...
     // sequential drive declarations:
     [A]                          // declaration 1: just drive A; no testing
     [A] -> [B]                   // declaration 2: drive A, test B
     [A] -> [B, C]                // declaration 3: drive A, test B and C

     // triggered drive declarations:
     create_clock D               // D is a clock for the triggered drives
     @trigfunc{0} [D] -> [E, F]   // declaration 4: test E and F
     @trigfunc{3} [D] -> [G, H]   // declaration 5: test G and H
}

main() {
   // drive statements which are matched to declaration 1:
   [x+y];                       // decl 1: drive A with x+y

   // drive statements which are matched to declaration 2:
   [z] -> [y];                  // drive A with z, test B against y
   [4] -> [];                   // drive A with 4, don't test B
   [4] -> [-];                  // drive A with 4, don't test B
```

[1] The term 'clocked' is used in preference to 'sequential' to avoid confusion with the software concept of sequential execution.

```
   // drive statements which are matched to declaration 3:
   [x] -> [y,z];              // B: test against y; C: test against z
   [x] -> [-,z];              // B: no test; C: test against z
   [x] -> [y,];               // B: test against y; C: no test
}

@trigfunc() {
   // any drive statements in this trigger function must be matched to declaration 4
   -> [x+y, 2];               // E: test against x+y; F: test against 2
}

@trigfunc(int i, int j, int k) {
   // any drive statements in this trigger function must be matched to declaration 5
   -> [foo1(), foo2()];       // G: test against foo1(); H: test against foo2()
}
```

Example 75

8.3.4 Clocked drives

A clocked drive declaration includes one clock signal on the LHS. If the DUT has multiple clocks, it will require multiple single-clock drive declarations.

Maia assumes that all the signals on the LHS of a clocked drive (apart from the clock itself) have setup and hold requirements to that clock, and that all the signals on the RHS are produced from that clock, and can be sampled at some time after the active clock edge (the active clock edge is determined from the clock declaration (8.5), the pipeline delay (9.2.4), and any timing declarations (8.7)). The required input setup and hold times, and the required output hold and delay times, are determined from any timing declarations; defaults are used if there are no timing declarations.

A sorted input event list is created for each drive declaration, for all the setup and hold events. The `.C` directive (9.3.1) is translated into two events; one for the clock rising edge, and one for the clock falling edge.

When a corresponding drive statement is encountered at runtime, it is 'executed' as follows:

1 Time is advanced to the first event in the input list

2 The input event is handled as follows:

i) If the event is a signal setup point, the corresponding expression on the LHS of the drive statement is evaluated, and the signal is driven with the value of that expression. If the expression was omitted, or specified as a '`-`' character, then the signal is not driven, and retains its previous value;

ii) If the event is a signal hold point, that signal is driven with `X`;

iii) If the event corresponds to the `.C` directive, the clock is driven to the required level;

iv) If the event corresponds to the `.R` directive, the corresponding internal DUT signal is released[1].

[1] There is no corresponding 'preload' directive or event; any drive of an internal signal is treated as a force, or preload.

3 Time is advanced to the next event in the input list, and step 2 is repeated until there are no more events in the input list;

4 When the input event list is empty, time is advanced to the end of the clock cycle, as defined by the `create_clock` declaration.

The period of time covered by input event processing is always a complete clock cycle, as defined by a `create_clock` declaration. Step 1 always occurs at time 0 in the waveform; step 4 advances to the last time slot in the waveform. The next drive statement will advance another clock cycle, although it will not necessarily be the same clock. An error is reported if a timing declaration requests input events which do not fit into the declared (or default) clock waveform. Note that time 0 in the waveform is the *Operating Point* (10.1); in other words, it is the time at which the statements before and after the drive statement are executed.

When the first clock edge is encountered during input event processing, it is also used to trigger a checker process for each RHS signal. The checker process confirms that the signal has the value of the corresponding RHS expression at the output delay point, and also confirms that the signal does not change at any time outside the window defined by the output hold and output delay times (the *stability window*). It is the checker process which is responsible for incrementing the internal test pass and fail counters (`passCount` and `failCount`), and for reporting any DUT failures. The checker process runs for one clock cycle from the first clock edge; it is an error if a timing declaration sets an output hold or delay time which does not fit into this period.

The checker processes for the various outputs run independently, and two or more failures at the same simulation time may be reported in different orders by different simulators, or during different runs with a single simulator. This may cause confusion in regression tests in which failures are expected. If this is the case, the output should be sorted before comparison[1].

8.3.5 Mixing clocked and combinatorial signals

The procedure described above cannot be used to test a selection of inputs which are both clocked and combinatorial, using a single drive declaration. Consider, for example, this declaration of a simple counter with an asynchronous reset:

```
DUT {
     module counter(
        input  ARST,              // async reset
               PLD,               // sync preload
               D,                 // input data
               CLK,
        output Q)
     create_clock CLK             // default clock declaration
     [ARST, PLD, D, CLK] -> [Q]
}
```

Example 76

[1] Error and warning messages from mtv have a simple format, with a file name in field 2, a line number in field 4, a time in field 5, and a signal name in field 7. This Unix command will sort the lines of a logfile according to simulation time, with identical times sorted according to signal name:

```
> sort -k 5,5n -k 7,7 -o mtv.log.sorted mtv.log
```

Given this declaration, the resulting testbench will treat ARST as a *synchronous* signal, which will be driven some time before the rising edge of CLK. This is likely to lead to a test failure when a test vector activates ARST. A failure will be reported if the change in ARST causes the DUT to change Q at any time outside the expected times (between Q's output hold and output delay times).

Maia has no knowledge of whether an input is 'clocked' or 'combinatorial'. The only information it has is the drive declaration which, in this case, requests that ARST be tested in the same way as PLD, D, and CLK. Since this drive declaration contains a declared clock, Maia treats it as a clocked drive. The correct way to test this DUT is to have two drive declarations:

```
DUT {
   module counter(input ARST, PLD, D, CLK, output Q)
   create_clock CLK              // default clock declaration
   [ARST] -> [Q]                 // async reset testing
   [PLD, D, CLK] -> [Q]          // synchronous operation testing
}
[0] -> [.X]                      // DUT output unknown
[1] -> [0]                       // reset the DUT
[0] -> [0]                       // turn off reset before sync testing
[1, 4, .C] -> [4]                // sync preload of data '4'
[0, -, .C] -> [5]                // count up
```

Example 77

8.3.6 Combinatorial drives

A combinatorial drive declaration is one which does not have a declared clock on the LHS.

Maia creates a 'cycle' time for combinatorial drives by using any relevant timing specifications, if there are any, or by using a default value otherwise. Each execution of a corresponding drive statement advances by this 'cycle' time.

There are essentially two choices for driving multiple combinatorial signals within this cycle. In the first, the inputs are all driven at the same time, and tested at their individual t_{OD} specifications. in the second, the input timing is adjusted such that the outputs should nominally all change at the same time, and the outputs are all tested together.

The second scheme has the advantage that a waveform display will quickly show what the spread of real, rather than specified, output delays is, and Maia currently uses this scheme. It should be noted that neither scheme is ideal where there are path dependencies (in other words, a single input affects multiple outputs, or multiple inputs affect a single output), and Maia may automatically relax specific timing constraints if a conflict is present; see the discussion on Constraint conflicts (8.9.7). A warning is always issued if a conflict is present.

8.3.6.1 Combinatorial cycle time

The combinatorial 'cycle time' is defined as twice the longest t_{OD} parameter (2 * t_{ODMAX}). The inputs are driven and the outputs are tested in the first t_{ODMAX} period, and an additional t_{ODMAX} delay is then added before starting the next cycle. This recovery period is added to simplify waveform displays.

Unconstrained input-to-output paths are given a default t_{OD} of 5ns. If no paths within a drive statement are constrained then all paths will be given the default timing, giving a 10ns cycle time. In this case,

the two alternative drive schemes become identical; the inputs are all driven at the same time, and are all sampled 5ns later, with a further delay of 5ns before the start of the next cycle.

8.3.7 Sequential declaration signature

A DUT declaration may contain any number of sequential drive declarations. Declarations are matched up to corresponding sequential drive statements using a signature, which is normally simply a count of the number of signals on the LHS and RHS of the declaration. This example is, again, a complete and valid program:

```
DUT {
     module test1(input D1, D2, CLK; output Q)
     create_clock CLK
     [D1, D2, CLK] -> [Q]       // signature (3:1)
     [D1, CLK] -> [Q]           // signature (2:1)
}
[0, 0, .C] -> [0]               // signature (3:1)
[0, 1, .C] -> [1]               // signature (3:1)
[1, .C] -> [1]                  // signature (2:1)
```

Example 78

However, it may be necessary to have more than one sequential drive declaration which has the same signal count. In this case, the declarations must be distinguished by adding a label (an identifier) to them. This label must be repeated in the drive statement:

```
DUT {
     module test1(input D1, D2, D3, D4, CLK; output Q)
     create_clock CLK
     v1: [D1, D2, CLK] -> [Q]  // label v1; signature (v1:3:1)
     v2: [D3, D4, CLK] -> [Q]  // label v2; signature (v2:3:1)
}
v1: [0, 0, 0, .C] -> [0]     // signature (v1:3:1)
v2: [0, 0, 0, .C] -> [1]     // signature (v2:3:1)
    [0, 1, 0, .C] -> [1]     // ERROR: unknown signature
```

Example 79

8.4 Signal declaration

A signal declaration declares one or more internal signals inside the DUT. The signal name is a `videntifier`, and may be anything that the back-end simulator recognises as an internal signal name. This will normally be a Verilog-style dotted identifier; it should be enclosed in double quotes if it is not a valid Maia identifier.

A signal declared in this way is treated identically to a port name declared in a module declaration (8.2). The signal is automatically declared as a global identifier, and so may be read or written directly, or it may be used in a drive statement.

When a signal is written to, it is automatically set to the required level inside the DUT using a 'force' mechanism. The force disables any other internal drivers on that signal, to allow it to take on the required value. The force must be explicitly released by using the `.R` directive (9.3.3).

An example of the driving and testing of an internal signal is given in (9.3.3).

8.4.1 Syntax

```
dut-signal-declaration:
   signal ( port-declaration-list )
```

8.5 Clock declaration

A clock declaration allows timing to be specified relative to a clock, and allows the `.C` directive to create a clock waveform to drive a DUT input, or to respond to a clock which is a DUT output.

`create_clock` identifies a DUT signal as a clock, and defines the required clock waveform. This declaration is required only if the `.C` directive is used in a drive statement, or if a virtual clock is required. A minimal `create_clock` declaration simply names a clock signal:

```
create_clock CLK   // minimal clock declaration, default waveform
```

The named signal (`CLK`, in this case) must be declared elsewhere as a single-bit DUT input or output port. If no waveform is specified, a default waveform is used; this is symmetrical, and has a period of 10ns. The waveform starts and ends at a low level (the 'default level'), and the rising edge occurs after a short delay.

A clock declaration must include exactly one clock name (`clock-name`). The period, waveform, and pipeline specifications are optional. If a waveform is specified, then a period must also be specified. There must be a maximum of one period, waveform, or pipeline specification.

If the clock name is preceded by `-name` then the clock is a *virtual clock*. A virtual clock is a clock which is not connected to the DUT; it is an error if the virtual clock name is also the name of a DUT port[1]. Conversely, a clock which is declared without `-name` is a physical clock which connects to a DUT port. In this case, it is an error if the physical clock name is *not* also the name of a DUT port.

Any constant values in a clock declaration are interpreted as floating-point numbers (4.6.1.1), in the units of the declared timescale. The timescale defaults to nanoseconds if it is not specified.

Lists of time values (`sdc-constant`) may be either space-separated, or comma-separated, for compatibility with other tools. A time may be specified as either a constant, or a constant expression. If an expression is used, it must be enclosed in parentheses, to avoid parsing ambiguities (since the expression may contain spaces).

Where `create_clock` is used to declare the waveform of a clock which is a DUT output, care should be taken to ensure that the declaration matches the physical clock. The generated testbench does not synchronise to the clock produced by the DUT, but instead assumes that the declared clock is correct.

8.5.1 Syntax

```
clock-declaration:
   create_clock clock-item-list

clock-item-list :
   clock-item
   clock-item-list clock-item
```

[1] Virtual clocks may be used to drive or sample DUT signals at times which are unrelated to the timing of physical clocks.

```
clock-item :
   clock-name
   clock-decl

clock-name :
  vname_opt videntifier
  vname_opt { videntifier }

vname : -name

clock-decl :
    period
    waveform
    pipeline

period : -period sdc-constant

waveform : -waveform { list-of-sdc-constants }

pipeline : -pipeline sdc-constant

list-of-sdc-constants :
   sdc-constant
   list-of-sdc-constants   sdc-constant
   list-of-sdc-constants , sdc-constant

sdc-constant :
   constant
( constant-expression )
```

8.5.2 Period declaration

The clock period is specified with `-period t`. If the period specification is omitted, it defaults to 10; if the timescale is also omitted, this is 10 ns.

The period must be greater than or equal to 2.

8.5.3 Waveform declaration

In normal usage, a waveform specification will include exactly two edges. In this case, the edge times may increase or decrease:

```
create_clock CLK1 -period 30.5 -waveform { 10.2 20.6 }
create_clock CLK2 -period 20   -waveform { 15 5 }
```

Example 80

The first entry (or, in general, the odd-numbered entries) correspond to rising edges (10.2ns for `CLK1`, and 15ns for `CLK2`), while even-numbered entries correspond to falling edges (20.6ns for `CLK1`, and 5ns for `CLK2`).

A waveform may be declared with any even number of edge times, for compatibility with other tools. If there are more than two edges, the edge times must increase monotonically. Any timing specifications will be relative to the earliest edge in this waveform.

A clocked drive statement defines the input and expected output signal behaviour over a single clock cycle, as defined by the waveform. In other words, the 'start' of the clocked drive statement occurs at the start of the clock waveform. This imposes two further restrictions on clock waveforms.

8.5.3.1 Input event timing restrictions

Any specified clock setup and hold times must fit inside the defined waveform. The example code below defines a 20ns clock. All events defined by the setup and hold specifications must therefore fit into time interval [0,20) ns[1]:

```
timescale ns
create_clock G period 20 waveform { 5 12 }
A -> posedge G = (5:-2)   // 5ns setup is valid
B -> posedge G = (6:-1)   // ERROR: 6ns setup is invalid
C -> negedge G = (12:1)   // 12ns setup is valid
D -> negedge G = (13:1)   // ERROR: 13ns setup is invalid
E -> negedge G = (1:7)    // 7ns hold is valid
F -> negedge G = (1:8)    // ERROR: 8ns hold is invalid
```

Example 81

An error will be reported during compilation if any setup and hold constraints cannot be met using the specified waveform. In this case, the waveform should simply be adjusted to accommodate the required setup or hold time.

8.5.3.2 Output event timing restrictions

The DUT output events do *not* have to fit into the time defined by the clock waveform. The reason is that Maia starts a checker process when the first edge in the clock waveform is encountered; this checker runs for one clock period from that edge. The output events must therefore fit into a clock period plus the delay to the first clock edge; in this case, for clock `G` declared above, this is the interval [5, 25) ns. An error will be reported during compilation if any output hold or output delay constraints cannot be met using the specified waveform. These constraints are unlikely to be violated, unless a drive statement tests outputs which are generated on both the clock rising and falling edges, and the delays are long compared to the clock period. If this is the case, the single drive declaration should be split into two declarations, one of which tests outputs generated from the clock rising edge, while the other tests outputs generated from the clock falling edge.

8.5.4 Pipeline declaration

The compiler must be able to statically determine the maximum pipeline level of any drive statement. This is always possible, unless a drive statement contains a variable pipeline level:

```
DUT {
     module test(input A, B, CLK; output C);
     create_clock CLK –pipeline 6;
     [A, B, CLK] -> [C];
}

main() {
     int plevel;
```

[1] Square brackets in intervals are inclusive; round brackets are exclusive. The interval [0, 20) therefore includes 0, and excludes 20.

```
    ...
    [expr1, expr2, .C] ->plevel [expr3];
}
```

Example 82

In this example, `plevel` can change at runtime, and the compiler cannot determine in advance the length of the checker pipeline associated with this drive statement. In this case, the maximum expected pipeline size must be specified as part of the clock declaration for the *clock* associated with the drive statement.

The maximum size is specified with `-pipeline`; in this case, it is declared to be 6 levels. A run-time error will be issued if `plevel` is found to be greater than 6 whenever this drive statement is executed. The compiler can always determine if a pipeline specification is required, and will report an error if it has not been supplied.

8.5.5 Examples

Some examples of clock declarations are:

```
timescale ns

// the clock signals must be declared as single-bit module inputs:
module test(input A,B,C,D; ...)

// A: symmetrical, default 10ns period, rising edge first
create_clock A

// B: 25ns period; will only be symmetrical if the resolution is <= 500ps
create_clock B -period 25

// C: 30.5ns period, with a rising edge at 10.2ns, and a falling edge at 20.6ns
create_clock "\top/C\" -period 30.5 -waveform { 10.2 20.6 }

// D: 20ns period, falling edge at 5ns, rising edge at 15ns
#define PERIOD 20
create_clock
  -period PERIOD
  -waveform { (PERIOD-5 /* = 15ns */) 5 } D
```

Example 83

8.6 Enable declaration

A bi-directional signal may be driven both by the Maia testbench, and by the DUT, and so is subject to possible contention. Contention can always be avoided by instructing the testbench to drive a `Z` to the DUT before enabling the DUT output drivers. However, this can be tedious and error-prone, and the process can be automated by declaring a DUT signal as an enable control, using `create_enable`.

8.6.1 Syntax

```
enable-declaration :
   create_enable enable-list
```

```
enable-list :
   enable-item
   enable-list , enable-item

enable-item :
   enable-port enable-port-slice_opt enable-control

enable-port : identifier

enable-port-slice :
   . ( constant-expression )
   . ( constant-expression : constant-expression )

enable-control :
   ( enable-level_opt control-sig )

enable-level : !

control-sig : identifier
```

enable-port must be a DUT output or inout; it may not be an internal DUT signal.

enable-port-slice has the same semantics as a bitslice (4.5.4.5). The indexes must be in range for *enable-port*, and must not overlap with any indexes specified in any other `create_enable` for this *enable-port*.

If *enable-level* is specified as `!`, then the enabling level is defined as `0`. If *enable-level* is omitted, then the enabling level is defined as any non-zero value. The testbench drives *enable-port* only when *control-sig* has the value of the enabling level. If *control-sig* has any other value, the testbench will tristate *enable-port*.

control-sig must be a DUT input or output, or an external variable.

8.6.2 Manual bidirectional control example

This code is part of a testbench for a 16-bit by 16-word RAM. The RAM has a bi-directional data bus (`D`), and an active-high data enable (`DEN`). The RAM tristates `D` when `DEN` is 0, and drives `D` when `DEN` is 1. Read operations are asynchronous.

```
DUT {
   module RAMB_1RW
       (inout  [15:0] D,
        input  [ 3:0] ADR,
        input  CLK, WE, DEN);

   ...
   create_clock CLK;       // default clock waveform, 10ns period
   [CLK, DEN, WE, ADR, D]; // clocked write; nothing to test
   [DEN, ADR, D] -> [D];   // combinatorial data bus read; D appears on both sides
}

main() {
   ...
   // 1: write data to the RAM
   [.C, 0, 1, e1, e2];           // write data e2 to address e1
```

```
   // 2: read data from the RAM
   [1, e3, .Z] -> [e4];          // read data from address e3, test against e4
}
```

Example 84

During the write operation, the testbench must set DEN to 0, to disable the RAM's output drivers.

During the read operation, the testbench must set DEN to 1, to enable the RAM's output drivers; it must also tristate it's own D output drivers, so that it can read back the data driven by the RAM. If the testbench accidentally enables both sources (by setting DEN to 1, and driving D with anything other than .Z), then it will read invalid data (probably X) from the DUT, and the test will fail.

8.6.3 Automatic bidirectional control example

The potential error of (8.6.2) can be avoided by declaring DEN as an enable signal, using create_enable:

```
DUT {
   ...
   create_enable D(!DEN);
}

main() {
   ...
   // 2: read data from the RAM
   [1, e3, -] -> [e4];         // read data from address e3, test against e4
}
```

Example 85

The declaration states that D is a tristate signal, and that the *testbench* may only drive D when DEN is 0. When DEN is non-zero, the testbench automatically drives D with Z, ignoring whatever value has been requested in the drive statement (in this case, a '–' was specified; this is a don't care condition).

8.7 Timescale declaration

The simulation timescale is specified as

```
timescale ts
```

where *ts* is one of seconds (s), milliseconds (ms), microseconds (us), nanoseconds (ns), picoseconds (ps), or femtoseconds (fs). If no timescale directive is provided, the timescale defaults to ns. There may be a maximum of one timescale declaration in the DUT section. The timing resolution is derived automatically, as described in (8.8).

Syntax

```
timescale-declaration:
   timescale timescale-units

timescale-units : one of
   s ms us ns ps fs
```

8.8 Time precision and representation

Times are represented as ordinary expressions. Time values are required primarily in the DUT section, but are also required for `wait` statements (6.9) in user functions. Times are not explicitly entered with a timescale unit (such as 'ns'); there is instead a timescale declaration which sets the required unit.

Times may be specified with an arbitrary precision. The compiler deduces the required precision by examining all constants in time expressions, and setting the precision to the maximum precision found.

```
DUT {                                   // a minimal DUT section with timing
     module test(input C, D; output Q)
     [C, D] -> [Q]                      // drive declaration
     timescale ns                       // timescale declaration
     create_clock C                     // clock declaration
     D -> posedge C = (0.1 : 2.340)     // tSU/tIH
     posedge C -> Q = (1 : 3.1)         // tOH/tOD
}
```

Example 86

In this example, the maximum precision is 2 decimal digits (note that trailing zeroes are ignored), which is equivalent to a precision of 10ps (assuming that the body of the code does not contain any `wait` statements with a higher timing precision).

In some circumstances it is possible to specify a timescale and precision which cannot be supported by the target simulator (a `ns` timescale, for example, with 7 decimal digits of precision, is equivalent to a timing precision of 0.1 fs, which is not in the range supported by Verilog simulators). The compiler will report an error in this case.

Hex floating-point values may not be used for time values, since the compiler needs to count decimal digits.

All time values in a DUT section must be constants or constant expressions (in other words, expressions which can be evaluated during compilation). The time delay in a `wait` statement must also be a constant or a constant expression; a dynamic wait can be achieved by putting a constant wait inside a loop.

A constant expression may include floating-point operations. The calculated precision is not affected by floating-point operators; the precision is derived solely from the constants in the code. If a floating-point operation is used, then care should be taken to ensure that the correct number formats and operators are used; `(2 * 0.1)`, for example, is *not* equal to `0.2`. In this case, `(2.0 .F* 0.1)` should be used; see (4.6.1.1) and (4.6.1.2).

8.8.1 Floating-point values in parameter lists

In some circumstances it may be necessary to pass floating-point values into a DUT as a parameter in a module declaration:

```
module test #(.tCO(0.2)) (input A, B; output C);
```

In this case, the intention is to pass the number '0.2' as the `tCO` parameter to the DUT. If the Maia timescale is `ns`, then this might be expected to set a clock-to-out delay of 0.2 ns. Recall, however, that module parameter lists are copied direct to the testbench code with no analysis or processing (8.2.1), and the number '0.2' will therefore appear verbatim in the DUT instantiation in the testbench code.

Furthermore, if the DUT is expecting a floating-point timing parameter, then it will almost certainly have its own timescale directive, which will over-ride the timescale directive in the Maia output. In summary, then, any floating-point values in a parameter list are ignored by Maia (along with the rest of the parameter list), and must be specified in a format which will be understood by the DUT itself.

8.9 Timing constraint declaration

A timing constraint declaration specifies either the required setup and hold times on the DUT's synchronous inputs, or the expected delays on the DUT outputs (both synchronous and combinatorial). Timing declarations are optional; they may be either omitted entirely, or applied to some, or all, of the DUT ports[1]. Timing declarations are only required when carrying out a timing simulation with a back-annotated netlist; the simulation will almost certainly fail without appropriate timing declarations. The declarations may be omitted when carrying out delta-delay simulations.

The purpose of the timing declarations is to ensure that any synthesis constraints were correctly specified, interpreted and applied, and that the resulting netlist correctly implements those constraints. The values in the Maia timing declarations should therefore be the same as any values that are specified in the synthesis constraints file. The synthesis constraints should therefore be translated directly into the equivalent Maia format. For this reason, Maia timing declarations are generally referred to as 'constraints' in this manual, although they are not of course synthesis constraints.

There are four constraints which can be applied to DUT signals, which are:

- A synchronous input setup time (`tSU`)
- A synchronous input hold time (`tIH`)
- A synchronous or combinatorial output hold time (`tOH`)
- A synchronous or combinatorial output delay (`tOD`). The symbol `tOD` is generally used in this manual for both cases, although `tCO` may also be used for synchronous (clocked) outputs.

Constraints may be applied either to DUT ports, or to internal DUT signals. `tSU` and `tIH` constraints may be applied to both inputs and inouts; `tOH` and `tOD` may be applied to outputs and inouts.

8.9.1 Syntax

```
timing-constraint :
     timing-constraint-LHS   = timing-constraint-RHS
   ( timing-constraint-LHS ) = timing-constraint-RHS

timing-constraint-LHS :
   tidentifier-list        timing-drive tidentifier-list
   timing-edge videntifier timing-drive tidentifier-list
   tidentifier-list        timing-drive timing-edge videntifier

tidentifier-list :
   *
   videntifier-list
```

[1] Care should be taken when constraining some paths to a combinatorial output, and not others; see (8.9.7).

```
videntifier-list :
   videntifier
   videntifier-list , videntifier

timing-drive : one of
   ->  to

timing-edge : one of
   posedge negedge

timing-constraint-RHS :
   ( constant-expression )
   ( tconstraint-RHS-val_opt : tconstraint-RHS-val_opt )

tconstraint-RHS-val :
   constant-expression
```

Constraints (timing-constraint-RHS) are normally specified as pairs of floating-point values, in the form `(t1:t2)`. `t1` and `t2` are time values in the simulation timescale (8.7). These pairs correspond to `(tSU:tIH)` for inputs, and `(tOH:tOD)` for outputs. The `tIH` and `tOH` values are optional, however, so constraints may also be specified as single floating-point values, in the form `(t)`, `(:t)`, or `(t:)`.

8.9.2 Input constraint definition

An example of an input constraint is:

```
D -> posedge C = (3.2 : 2.1)
```

Example 87

This constraint should be interpreted as follows, assuming a `ns` timescale:

- the new value of the `D` input must be valid by, at the latest, 3.2ns before the rising edge of `C`
- the new value of the `D` input must remain active for at least 2.1ns after the rising edge of `C`

These requirements are best represented in terms of an analog waveform:

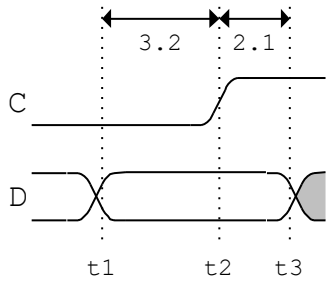


Figure 3: Input constraint definition

To match the required behaviour, the Maia testbench drives `D` in such a way as to make the following guarantees, which are independent of any race conditions:

- the DUT is guaranteed to see an invalid level on `D` (3.2+δ) ns before the clock edge, and a valid level on `D` 3.2ns before the clock edge, where δ is the minimum timing precision

- the DUT is guaranteed to see a valid level on `D` 2.1ns after the clock edge, and an invalid level on `D` (2.1+δ) ns after the clock edge.

The 'invalid' level is set by driving `X` to the DUT.

8.9.3 Output constraint definition

An example of an output constraint is:

```
posedge C -> Q = (2.5 : 5.3)   // clocked version
A -> B = (2.5 : 5.3)           // combinatorial version
```

Example 88

These two constraints are treated identically, except that the combinatorial constraint is tested from any change in `A`, while the clocked constraint is tested only from a rising edge of `C` (or a falling edge, if `negedge` is instead specified). For simplicity, the discussion below considers only the clocked case.

This constraint should be interpreted as follows, assuming a `ns` timescale:

- The old value of the `Q` output must hold until, at the earliest, 2.5ns after the rising edge of `C`
- The new value of the `Q` output must be valid by, at the latest, 5.3ns after the rising edge of `C`

These requirements are best represented in terms of an analog waveform:

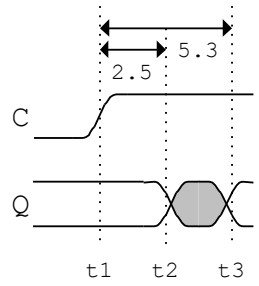


Figure 4: Output constraint definition

The Maia testbench samples `Q` in such a way as to guarantee that a pass will be reported if, and only if, the following three conditions are fulfilled, independently of any race conditions:

- Any sampling event external to the DUT at a time 2.5ns after the rising edge of `C` (at `t2`) will sample the old value of `Q`
- Any sampling event external to the DUT at a time 5.3ns after the rising edge of `C` (at `t3`) will sample the expected new value of `Q`
- Any sampling event external to the DUT after time `t3`, up to and including the next `t2`, will sample the new value of `Q`

8.9.4 Input setup and hold constraints

`tSU` and `tIH` may be either positive or negative. `tSU` is measured *before* the relevant clock edge: a `tSU` of 1, for example, means one time unit before the clock edge, while a `tSU` of –1 means one time

unit *after* the clock edge. `tIH` is measured *after* the relevant clock edge: a `tIH` of 1, for example, means one time unit after the clock edge, while a `tIH` of –1 means one time unit *before* the clock edge.

The time specified by the value of `tSU` must be before (or at the same time as) the time specified by the value of `tIH`.

If the `tSU` and `tIH` constraints are present, Maia will drive the relevant DUT input as described in (8.9.2). The examples below assume a `ns` timescale, and show the times over which the input is driven with the required value, as an inclusive range relative to the clock edge. The time interval [-2.1, -0.1], for example, means that the input is driven with the required value for a total time of 2.0ns, which starts 2.1ns before the clock edge, and ends 0.1ns before the clock edge.

```
D -> posedge C = ( 2.1: 0.1)   // valid 2.2ns: [-2.1, 0.1]
D -> posedge C = ( 2.1:-0.1)   // valid 2.0ns: [-2.1,-0.1]
D -> posedge C = (-0.5: 0.7)   // valid 0.2ns: [ 0.5, 0.7]
D -> posedge C = (-0.5:-0.1)   // ERROR: cannot both be <0

// D setup and hold relative to the falling edge of C
D -> negedge C = (-0.5: 0.7)   // valid 0.2ns: [ 0.5, 0.7]

// 'to' is alternative syntax for '->'
D to negedge C = (-0.5: 0.7)   // valid 0.2ns: [ 0.5, 0.7]

// the hold is optional, and defaults to 0: the setup
// must therefore be >= 0
D to posedge C = ( 0.5)        // valid 0.5ns: [-0.5, 0.0]
D to posedge C = (-0.5)        // ERROR: setup after hold
```

Example 89

If no constraints are applied to a synchronous DUT input, Maia will assume that the input is untimed, and will drive the input a short time before the first clock edge, and will maintain the driven value for a short time after the first clock edge. The specific times are arbitrary and are chosen simply to ensure a clear waveform display in a waveform viewer. The input is driven to `X`'s at all other times.

8.9.5 Output hold and delay constraints

`tOH` is the time for which a DUT output is guaranteed *not* to change after any change on a controlling input[1]. `tOH` must therefore be greater than or equal to zero. No hold time check is carried out if `tOH` is specified as zero, or omitted.

`tOD` is the time at which the DUT output is guaranteed to have its new value. `tOD` must therefore be greater than `tOH`; it is illegal for both to have the same value, unless both are zero (`tOH` = `tOD` = 0 is a special case, which corresponds to an untimed or delta-delay DUT).

Note that `tOH` is *not* equivalent to a minimum output delay. A `tOH` specification requires that the output has *not* changed at time `tOH`, while a minimum output delay specification requires that an

[1] A 'controlling input' is the relevant clock edge for a sequential drive statement, or the last input to change for a combinatorial drive statement.

output has, or may have, changed at that time. A minimum delay specification for a device output is of little or no use.

The `tOD` specification should normally be the same as the value specified in the synthesis constraints file, and so will normally be the expected maximum output delay. There is no mechanism to specify minimum, typical, and maximum output delays; it would make no more sense to specify these in Maia than it would in a synthesis constraints file.

If the `tOH` and `tOD` constraints are present for a given DUT output, Maia will test the output as described in (8.9.3).

Synchronous output constraints must name a declared clock on the LHS, and must have a `posedge` or `negedge` qualifier; for example:

```
posedge C -> Q = (2.0)          // tOD (tCO) from C to Q
```

Example 90

A constraint which does not have a `posedge` or `negedge` qualifier is a combinatorial constraint, and should be applied only to combinatorial outputs; for example:

```
A -> B = (2.0)                  // tOD from A to B
```

Example 91

Some example specifications are given below.

```
posedge C -> Q = (0.2 : 3.1)
posedge C -> Q = (-0.2 : 2.1)  // ERROR: tOH and tOD must be >= 0
posedge C to Q = (4.1 : 2.1)   // ERROR: tOH must be <= tOD
A -> B = (0.1 : 2.5)
B to C = (2.0)                 // hold optional; this is equivalent to (0:2.0)
```

Example 92

If no constraints are applied to a DUT output, Maia will assume that the output is untimed, and will sample it a short time after any controlling input has changed value. No output hold test is carried out; in other words, there is no test that the output retains its previous value after any controlling input changes value.

8.9.6 Wildcard constraints

Where the syntax allows a list of DUT signals (`tidentifier-list`) in a constraint declaration, that list may instead be specified as a `*` wildcard. Some examples of wildcard constraints are:

```
* -> posedge CLK1 = (0.8, 3.1)      // case 1: setup and hold to a clock
negedge CLK2 -> * = (9.6)           // case 2: output delay from a clock
* -> D = (2.1, 5.3)                 // case 3: combinatorial
E -> * = (4.2)                     // case 4: combinatorial
```

Example 93

The wildcard is a shorthand notation for all the signals on the LHS of a drive declaration (excluding the clock itself, for a clocked constraint), or all the signals on the RHS of a drive declaration. It is *not* a shorthand for all possible paths from an input to an output; Maia does not analyse the DUT to find these paths, but instead relies on any drive declarations.

For case (1), all the signals on the LHS of *any* drive declaration which is clocked from `CLK1` are assumed to have the same setup and hold to `CLK1` (in this case, 0.8 and 3.1).

For case (2), all the signals on the RHS of *any* drive declaration which is clocked from `CLK2` are assumed to have the same `tCO` specification from `CLK2` (in this case, 9.6).

Case (3) is applied to any combinatorial drive declaration which lists `D` as an output. In this case, all signals on the LHS of those drive declarations are given the same `tOH` and `tOD` specification to `D` (in this case, 2.1 and 5.3).

Case (4) is applied to any combinatorial drive declaration which lists `E` as an input. In this case, all signals on the RHS of those drive declarations are given the same `tOD` specification from `E` (in this case, 4.2).

Care should be taken when using a wildcard in a combinatorial drive, to avoid possible constraint conflicts; see (8.9.7).

8.9.7 Constraint conflicts

In general, there are two classes of conflict which are possible when testing combinatorial paths:

1. If a combinatorial output is derived from more than one input, then it may not be possible to test all the output hold requirements on that output (8.9.7.1);
2. If multiple outputs have dependent inputs, then it may not be possible to test all the output delay requirements on those outputs (8.9.7.2).

These conflicts apply only to combinatorial constraints. When a conflict is detected, one or more constraints must be relaxed; the compiler can always detect conflicts and will issue a warning identifying the relaxed constraints.

If it is necessary to fully test the timing of combinatorial outputs, then separate drive declarations should be created for each constrained input-to-output path, and each should be tested separately. Any attempt to test multiple combinatorial paths using single drive declarations will rapidly become unworkable; see (8.9.7.1) and (8.9.7.2).

8.9.7.1 Conflict case 1: multiple inputs

Consider these two timing constraints, for two inputs to a combinatorial circuit:

```
A -> D = (14 : 18)      // tOH = 14, tOD = 18
B -> D = (6 : 12)       // tOH = 6, tOD = 12
```

Example 94

In this case, Maia arranges the DUT timing such that `A` is driven at relative time 0, and `B` is driven at time 6. If `D` is to change, it should therefore take on its new value at, or before, time 18 (Figure 5). This method tests both `tOD` requirements, and guarantees that a `tOD` failure will be detected (although, if there is a failure, it cannot determine which of the two paths has failed). However, it is now impossible to test both `tOH` requirements.

If the DUT is correctly implemented, `D` will remain valid until at least time 12. However, if `D` does change in the time range (12, 14], Maia cannot tell whether the change is allowable (as a result of a

change in B), or a violation (as a result of a change in A, before A's hold time specification has expired). For these constraints, the compiler will report that the hold time requirement has been relaxed, and that the hold specification from A to D has been dropped.

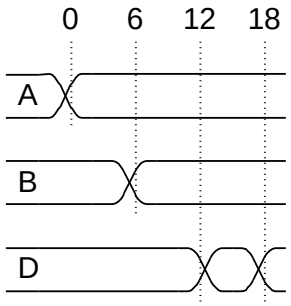


Figure 5: Multiple input constraint conflict

The situation is worse if there are any further inputs to D, but they have not been constrained. Consider this drive declaration:

```
    [A, B, C] -> [D]      // test a 3-in combinatorial circuit
```

Example 95

If only the two paths A-D and B-D have been constrained, Maia will set the unconstrained path from C to D as follows:

- the hold time is assumed to be 0;
- the output delay is set to a default value, which is not related to any specified output delays.

This will almost certainly not produce the required results. When constraining combinatorial circuits, *all* input to output paths should be constrained with a t_{OD} specification, and a single hold specification should be used for all paths by providing a wildcard. Maia will then only test the hold specification for the controlling input (the last input that changes). The required constraints are now:

```
* -> D = (6, 18)   // hold requirement from any input
A -> D = (18)      // tOD only
B -> D = (12)      // tOD only
```

Example 96

The tOD value in the wildcard constraint is arbitrary, since it will be overridden by the higher-priority individual specifications; however, it should be at least 6 to avoid a syntax error.

8.9.7.2 Conflict case 2: multiple outputs with dependent inputs

Consider these timing constraints, for two combinatorial outputs driven from two inputs:

```
DUT {
   module comb1 (input A,B; output D,E)
   [A,B] -> [D,E]
   A -> D = (14 : 18)
   B -> D = (6 : 12)
   A -> E = (6 : 9)
   B -> E = (5 : 12)
}
```

Example 97

If the D output is considered, and the same procedure is followed as in (8.9.7.1), then the times at which the testbench must drive A and B are again as shown in Figure 5. However, if the E output is now considered, it is apparent that A and B are driven at arbitrary times. The consequence of this is that the testbench now cannot detect a failure in the tOD specifications for the path A-E:

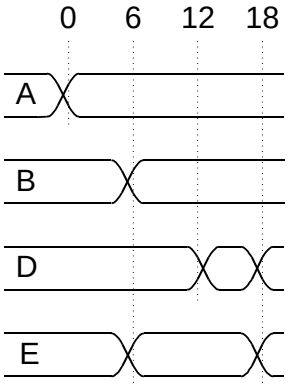


Figure 6: multiple output constraint conflict

For these constraints, the compiler will warn that the timing on input A has been relaxed (and that the output hold times on D and E have also been relaxed).

9 DRIVE STATEMENT

9.1 Introduction

A module declaration (8.2) creates a set of named external variables which correspond to the input, output, and bidirectional ports of the DUT, while a signal declaration (8.4) creates equivalent named variables which correspond to specified internal signals within the DUT. In principle, having direct access to these variables is sufficient to allow simple manual testing of the DUT, in a procedure such as:

- wait until an input port setup point, using a `wait` statement
- assign an expression to the input port
- wait, set the clock active
- wait until an output port is expected to be valid
- read the output port and check it; report the results with `report` or `assert`
- set the clock inactive, and wait until the start of the next cycle

However, this procedure is complex, and quickly becomes impractical when a number of ports have to be driven and tested, or when the outputs are pipelined, or when various inputs and outputs have different timing. Maia therefore automates this procedure using a *drive statement*. A drive statement has a number of advantages over the manual process described above:

- automated input, output, and bidirectional timing derived from constraints
- automated stability testing (glitch checking)
- automated pipelined output testing (outputs tested a known number of clock cycles after the inputs change)
- automated triggered output testing (outputs tested after a trigger condition is found)
- automated pass/fail counting, and error reporting
- internal DUT signals are treated identically to external ports, without the need for force/release semantics
- automated bus direction switching for bidirectional signals
- automated clock timing and driving
- static type checking on port and signal directions
- optional static type checking that confirms that module signals are driven by, or compared against, expressions of the correct bit size

9.2 Statement format

A drive-statement includes an optional list of input expressions on the left hand side, a separator, an optional pipeline expression, and an optional list of output expressions on the right hand side. Each input and output expression is matched to a port or a signal which is named in the corresponding drive declaration (8.3).

There are three different forms of drive statement, since the input and output lists are optional. Output-only drive statements are *triggered* drive statements. The other two forms (the form with inputs only, and the form with both inputs and outputs) are *sequential* drive statements. Sequential drive statements may additionally be *pipelined*.

Sequential drive statements may be used only in user functions (7.5); triggered drive statements may be used only in trigger functions (7.7).

9.2.1 Drive statements with both input and output expressions

A drive statement with both input and output expressions drives the inputs specified in the corresponding drive declaration, and tests the outputs specified in the same drive declaration. This program (an example of a complete testvector-program) includes two drive statements:

```
DUT {
     module test1(input D1, D2, CLK; output Q)
     create_clock CLK
     [D1, D2, CLK] -> [Q]    // sequential drive declaration (8.3.3)
}
[0, 1, .C] -> [0]            // drive statement 1
[1, 0, .C] -> [1]            // drive statement 2
```

Example 98

In the first clock cycle, D1 is driven to 0, and D2 is driven to 1, before driving the clock. The output is then tested against 0. In the second clock cycle, D1 is driven to 1, and D2 is driven to 0, before driving the clock. The output is then tested against 1.

9.2.2 Input-only drive statements

A drive statement, and the corresponding declaration, may omit the right hand side. In this case, the inputs are driven as specified, and no outputs are tested:

```
DUT {
     module test1(input D1, D2, CLK; output Q)
     create_clock CLK
     [D1, D2, CLK] -> [Q]    // sequential drive declaration 1 (8.3.3)
     [D1, CLK]               // sequential drive declaration 2
}
[0, 1, .C] -> [0]            // drive statement 1
[1, 0, .C] -> [1]            // drive statement 2
[0, .C]                      // clear D1, advance one clock cycle
```

Example 99

9.2.3 Output-only drive statements

An output-only drive statement is a *triggered* drive statement, and may be used only in a trigger function. The corresponding drive declaration must include a single input, which must be a declared clock. The trigger function is implicitly driven from this clock:

```
DUT {
     module test1(input D1, D2, CLK; output [15:0] Q);
     create clock CLK;
     @tfunc [CLK] -> [Q];    // triggered drive declaration (8.3.3)
}
main()
     int x;
     ...
     trigger tfunc(x) when Q == 7;
}
@tfunc(int y) {
     ->[y+2];                // Q should be 'x+2' the cycle after it is 7
     ->[8];                  // Q should be 8 two cycles after it is 7
}
```

Example 100

Output-only drive statements are unusual in that the drive declaration and the drive statement do not match. The declaration must include a single clock input, but the statement itself omits the clock input, since it is not responsible for driving the clock (in the example above, the clock might be driven from `main`, but the resulting output is tested in `tfunc`).

Triggered drive statements may not include a pipeline expression.

9.2.4 Pipelined drive statements

Sequential drive statements may optionally include a pipeline expression after the **->** separator; if the expression is omitted, the pipe level defaults to one (in other words, the outputs are tested immediately after the clock edge). Note that *combinatorial* drive statements may not be pipelined; a pipeline expression is illegal if there is not a declared clock on the LHS of the declaration.

The pipeline level may be specified as an integer constant, an identifier for a variable, or an expression (which must be enclosed in parentheses to avoid ambiguity). However, the pipeline level must be known before the drive statement is executed; the expression is sampled when the statement is reached. If the pipeline level cannot be determined in advance, then a triggered drive statement should be used, rather than a pipelined drive statement.

```
DUT {                // 4-stage pipelined 8x8 multiplier
     module test(
       input  [7:0] D1, D2;
       input  CLK;
       output [15:0] Q);
     create_clock CLK;
     [D1, D2, CLK] -> [Q];
}
main() {
     var8 i,j;
     for all i
       for all j
```

```
          [i, j, .C] ->4 [i *$16 j];
}
```

Example 101

9.2.4.1 The pipelined checker

A pipelined drive statement creates a *pipelined checker* for any signals on the RHS of the declaration. For the example above, the pipeline is advanced by `CLK`, and the expected output data is loaded into level 4 of the pipeline. The expected output data progresses down the pipeline, and is tested against `Q` when it emerges from level 1.

9.2.4.2 Checker flushing

Pipelined checkers are automatically flushed when the program exits (either by returning from `main`, or by calling `exit`). Any outstanding test operations are completed, and the results are recorded. If necessary, however, a pipelined checker can be manually flushed simply by issuing further operations with don't-care inputs. The 4-stage multiplier above may be manually flushed as follows:

```
main() {
     var8 i,j;
     for all i
        for all j
           [i, j, .C] ->4 [i *$16 j];
     [-,-,.C] -> [255 *$16 253];
     [-,-,.C] -> [255 *$16 254];
     [-,-,.C] -> [255 *$16 255];
}
```

Example 102

There is no simple way to test the pipeline output before the pipeline has filled (for the example above, `Q` is not tested until the fourth clock edge). If necessary, a separate trigger function may be used to check the outputs while the pipeline is filling.

When changing the required pipeline level, it is potentially possible to overwrite a given level in the checker. A runtime error will be reported if existing (untested) data in the checker is overwritten with new data. However, 'don't care' data may be written into the checker without affecting the previous data at that level. Uninitialised levels in the checker are treated as don't care data (in other words, no test is carried out when the uninitialised level emerges from the checker pipeline).

9.2.4.3 Determining the maximum checker pipeline size

The required maximum size of any checker pipeline must be statically determinable. The compiler automatically determines the required maximum size if all the drive statements referring to a given pipeline have statically determinable pipe levels (in other words, the level is specified as a constant or as a constant expression). If, however, the required maximum size cannot be determined during compilation, then it must be specified as part of the clock declaration, with a `-pipeline` specification (8.5.4).

9.3 Drive directives

Drive statements may contain directives, rather than expressions. Directives are case-insensitive, and are either shorthand for specific input and output conditions, or specify some action on or within the DUT.

9.3.1 .C

This directive may appear only on the left-hand side of a drive statement, in a position which corresponds to a single-bit port which has been declared as a clock (8.5). It may only appear once in a drive statement; there is no mechanism to clock more than one input simultaneously. This directive appears only in clocked drive statements; it cannot be used in combinatorial drive statements.

The clock directive instructs the simulator to advance one clock cycle, using the default clock waveform or the waveform defined in the clock declaration. The testbench automatically times inputs and samples outputs according to this clock waveform.

9.3.2 .X and .Z

When these directives appear on an input, the entire input is driven to `X` or `Z`; when they appear on an output, the entire output is tested against `X` or `Z`.

9.3.3 .R

This directive specifies an internal 'release' condition within the DUT. It may only appear on the left-hand side of a drive statement, in a position corresponding to an internal DUT signal (8.4); it may not be specified for a DUT port.

Internal DUT signals may be driven in the same way as external DUT ports, but this is handled internally by disabling the internal DUT driver that would otherwise have driven that signal. This internal driver is automatically disabled whenever a drive statement applies an expression to an internal signal; it remains disabled until the `.R` directive is issued.

This directive is applied with the same timing as any other expression applied to the specified internal signal. If, for example, a timing declaration specifies that the internal signal has a setup of 2.1ns to a declared external clock, then the `.R` directive will re-enable the internal driver 2.1ns before that clock.

The example below shows a 4-input 1-output clocked module. The `Q` output is driven from a D-type register, and the input to the D-type is named `Dint` in the HDL code. `Dint` is declared as a signal in the DUT declaration, and so may be driven from the testbench, to force the DUT output to a specific value. Left to its own devices, this DUT would produce the output sequence `01010101` over eight clock cycles; this code instead forces the output in cycles 3, 4, 5, and 6 to produce `01100001` instead.

```
DUT {
     module top(input A, B, C, CLK; output Q)
     signal (input top.mod1.Dint)   // internal signal
     create_clock CLK
     [CLK, top.mod1.Dint] -> [Q]    // force the output
}
// start with the internal driver enabled
[.C, -]  -> [0]    // the testbench is not driving Dint
```

```
[.C, -]  -> [1]
[.C, 1]  -> [1]    // output should be 0, is forced to 1
[.C, 0]  -> [0]    // output should be 1, is forced to 0
[.C, -]  -> [0]    // output should be 0, is forced to 0
[.C,]    -> [0]    // output should be 1, is forced to 0 ('-' is optional)
[.C, .R] -> [0]    // internal driver re-enabled, output takes on internal value
[.C, -]  -> [1]    // internal driver remains enabled
```

Example 103

9.3.4 Don't care conditions

A don't care condition is specified either with a '**-**' character, or by completely omitting the entry in the drive statement (both versions are shown in the example above). When applied to an input, the input is unchanged from its previous value. When applied to an output, the output is ignored for testing purposes.

9.4 Labelled drive statements

Drive statements must be labelled when two or more drive declarations would otherwise have the same signature (8.3.7). It is not possible for triggered drive statements to have the same signature, so triggered drive statements are never labelled.

10 SCHEDULING MODEL

10.1 Introduction

This section describes an idealised scheduling model which is independent of the back-end code generator, and whether or not the generator relies on an existing third-party simulator. There may potentially, however, be issues with specific Verilog simulators (14.7.6).

Maia uses a co-operative scheduling model, in which a given function remains in context until it executes a `wait`, `drive`, `trigger`, or `exec` statement. These statements are defined as 'suspending' statements, while all other statements are defined as non-suspending statements.

All maximally-sized blocks of non-suspending statements are guaranteed to execute atomically, in zero simulation time, without interference from any other function. When a suspending statement is executed the current function suspends and returns control to the scheduler, which may then schedule future activity as a result of that statement.

The scheduler then advances to the next time at which an activity has been scheduled. The corresponding function is then resumed, and it carries on execution until it executes a suspending statement.

There may potentially be more than one function which is scheduled to be resumed at a given time. If this is the case, the scheduler makes an arbitrary decision as to which of these functions to resume. User code should not assume any given order of function execution when statement blocks in different functions are scheduled to be executed at the same time; the statement blocks may have the desired order of execution in one program run, but have a different order in a second run.

10.2 Threads

Maia programs are multi-threaded. The `main` function is entered at or before time 1[1], and executes in thread 0 (the 'main' thread). New threads are created in one of two ways:

1. by execution of an *exec statement*. The `exec` statement returns immediately (in zero simulation time), and the newly-created thread starts execution immediately. A function which is entered by means of an exec statement is a 'Thread Function' (7.6), and may advance time as required (10.4). Every Thread Function has a unique thread identifier (a 'thread ID').

2. *trigger functions* (7.6) are automatically entered when the corresponding trigger condition is encountered. Trigger functions do not have a thread ID.

Statements which are scheduled to run at the same simulation time will execute in an arbitrary non-deterministic order. The two `report` statements in Example 68, for example, both run at the same time, in an unknown order.

[1] `main` may be entered at time 1, rather than time 0, in order to avoid start-up races in the generated Verilog code. 'Time 1' refers to the first step in the program's execution. If the time units are nanoseconds, and the precision is 100 ps, for example, then 'time 1' is 0.1 ns.

10.3 Program termination

A Maia program will continue execution until an `exit` statement is encountered, or until all threads have completed execution.

Execution of an `exit` statement (from any thread) will terminate the program cleanly, together with any threads which are currently active. Program execution will otherwise continue until all threads (including the main thread) have finished execution by "falling off the bottom". This second termination mechanism is equivalent to requiring all threads to re-join `main`, and then terminating `main`.

10.4 Advancing time

The only Maia statements which advance time are the wait statement (6.9) and the drive statement (9). A function which includes wait or drive statements is said to be *time-consuming*; all other functions execute in zero simulation time.

The wait statement suspends execution of the calling thread for the specified time. The thread resumes execution on completion of the wait.

Every drive statement has an associated drive declaration. The declaration itself has an associated cycle time, which may be explicit, or which may be created by the compiler. The *Operating Point*, or OP, is the time at the start of that cycle.

Consider, for example, the case where the user declares two clocks, where clock A has a cycle time of 6ns, and clock B has a cycle time of 8ns. Any drive statements which are associated with clock A will then have OPs at *6n* ns (0, 6, 12, and so on), while any drive statements associated with clock B will have OPs at *8n* ns (0, 8, 16, and so on).

When a drive statement is executed the calling thread is suspended, and resumes at a later time. The total time suspended depends on whether the calling thread is already at the relevant OP:

- If the calling thread is not already at the OP[1], it suspends until the second following OP is reached. If, for example, the drive statement is associated with clock A, and the current simulation time is 20ns, then the statement suspends for a total of 10ns (4 ns to synchronise with the correct OP, and then a further 6ns to advance a complete cycle time).
- If the thread is already at the OP, it suspends until the next OP. If, for example, the drive statement is associated with clock B, and the current simulation time is 24ns, then the statement suspends for 8ns, to advance for a complete cycle time.

In both cases, execution resumes at the relevant OP.

10.5 Thread Functions

When a function is entered by execution of an `exec` statement a new instance of that function is created, together with any local storage required by that function (including any static objects declared

[1] This will happen at (1) program start-up, or (2) when a wait statement has been executed, or (3) when a previous drive statement with a different cycle time declaration has been executed.

within the function). This local storage is referred to as 'Thread-local storage', or TLS, and the function itself is a 'Thread Function'. There may be multiple in-progress instances of any such Thread Function at a given time.

A function which is *not* a Thread Function exists as a single instance at runtime, irrespective of the time at which the function is called, or the thread from which it is called[1]. Example 104 calls `f3` from two threads:

```
void main() {
   int tid;
   exec f1(tid);
   wait 1.5;
   exec f2(tid);
   // f1(tid);  /* ILLEGAL: f1 is a Thread Function, and must be exec'ed */
}

void f1(int& tid) {
   f3(tid);                                          // direct call: no 'exec'
}

void f2(int& tid) {
   f3(tid);
}

void f3(int tid) {
   report("%T: f3, in thread %d\n", _timeNow, tid);  // upper-case 'T' for float
}
```

Example 104

This program reports:

```
0.1 ns: f3, in thread 1
1.6 ns: f3, in thread 2
```

The `f3` function that is called in threads 1 and 2 is the *same* instance of the `f3` function, and does not have any TLS. While this is not an error, it is likely to lead to unexpected results if `f3` advances time, and this should be avoided.

Note that, in this example, execution begins at 0.1 ns. This is 'time 1', because the default time units (ns) are used, and because the wait of 1.5 ns sets the precision to 100 ps (see 8.8).

10.6 HDL signal drivers

A thread may not be created if the new thread could potentially drive an HDL signal which is already driven in another thread. This determination is made statically, and an error is raised if necessary[2]. This procedure ensures that any HDL signal which is driven by the program has only a single driver.

[1] This is a limitation of the 2019.9 Verilog code generator. Recursive function calls are not supported in 2019.9 for the same reason.

[2] The compiler statically constructs a call graph to determine whether this is possible. mtv will output a dot-format call graph if the `-cg` option is specified.

If a given thread is responsible for driving a clock signal (by executing drive statements), then that thread will generate the clock waveform described by the relevant `create_clock` declaration. The waveform is 'anchored' at time 0, and not at the time at which the first drive statement is executed. This ensures that all clocks which are generated by the testbench have a known relationship to each other, which can be determined solely from the relevant `create_clock` declarations.

DUT-output clocks are generated by the DUT itself. The `create_clock` declaration describing a DUT-output clock must match the actual clock waveform generated by the HDL code, or signals generated by the testbench will not have the correct timing relative to the clock edges.

10.7 Operating point

All user statements are executed either at program start-up, or on completion of a wait statement, or at a time defined as an *operating point*, or 'OP'. Drive statements have an associated (potentially defaulted) cycle time definition, and advance time on a cycle-by-cycle basis. The OP is the point at the start of that cycle at which user statements are executed. This abstraction allows simple testbenches to be written in a cycle-synchronous way:

```
main() {
  // execute code at program start-up
  ...
  // this drive statement synchronises to the next OP, drives the A, B, C inputs,
  // advances one cycle to the next OP, and checks the outputs against D, E, F
  [.C, A, B, C] -> [D, E, F];

  for(int i=0; i<100; i++) {     // run for 100 clock cycles
    ...                          // now at an OP: set up inputs for next clock edge
    [.C, A, B, C] -> [D, E, F];  // drive inputs, advance one cycle, test outputs
  }
}
```

Example 105

10.7.1 Clocked drive statements

For a clocked drive statement, the OP is simply the time at the start of the relevant clock definition, and the drive statement will advance time by the cycle time given in the clock definition. In the absence of any timing declarations, a default waveform (8.9) is constructed, with a rising edge shortly after the start of the waveform. Waveform display tools will then show the DUT inputs being driven shortly before the clock edge.

If the DUT section contains setup constraints for DUT inputs which appear in the drive statement, then those inputs will be automatically driven at the specified time, and *not* at the OP. In this case, the OP simply defines the time at which user statements will be executed, rather than the time at which the DUT inputs will be driven. However, the OP must be at, or before, the declared setup point. The generated testbench will record the required values of the DUT inputs when the drive statement is executed, but will defer driving these values until the declared setup points are reached.

10.7.2 Combinatorial drive statements

For a combinatorial drive statement, the compiler determines an equivalent cycle time from the relevant combinatorial delays. This cycle time reflects the longest combinatorial path from the signals in the drive inputs, through to the drive outputs. The OP is the time at the start of this cycle. The compiler will again use a default cycle time if the relevant information is not supplied in the DUT declaration.

10.7.3 DUT output testing

The OP can be considered to be the time at which the user drives *inputs* to the DUT. If the output hold and delay times are appropriately constrained (8.9.5), the DUT *outputs* from the *previous* clock edge may also be 'manually' tested at this time, by reading and checking them. This is true when using the default clock waveform and timing constraints, but it is not generally the case (the outputs may not be valid until after the OP, for example). Manual DUT output testing is discussed in 10.11 below.

The runtime therefore ignores the OP when testing DUT outputs, and instead triggers a pipelined tester on the appropriate clock edge (see 8.5.3.2).

10.8 Sequential combinatorial drive statements

A combinatorial drive statement within a user function has no clock associated with it. The 'cycle time' is derived from any supplied t_{OD} timing parameters; see (8.3.6.1).

10.9 Sequential clocked drive statements

A clocked drive statement within a user function always has a defined clock signal.

A user function may use any defined combinatorial or clocked drives, which are all independent from the point of view of advancing time. The `main` function, in this code, is an example of a user function which uses multiple clocked and combinatorial drives:

```
DUT {
     module TEST(
        input CLK1, CLK2, CLK3, A, B, C, D, E, F;
        output Q1, Q2, Q3, G);

     create_clock CLK1 period 8;         // default timescale (ns)
     create_clock CLK2 period 13;
     create_clock CLK3 period 18;

     D1: [CLK1, A] -> [Q1];
     D2: [CLK2, B] -> [Q2];
     D3: [CLK3, C] -> [Q3];

     // D, E, F are combinatorial inputs, driving output G, with a maximum
     // tOD of 4.5ns
     D -> G = (0.2 : 2.5)
     E -> G = (0.4 : 3.5)
     F -> G = (1.0 : 4.5)
     [D, E, F] -> [G];
}
```

```
main() {
     var a, b, c, d, e, f;
     ...                  // these statements are executed at start-up
     D1: [.C, a] -> [d];  // advance one CLK1 waveform, CLK2/CLK3 unaffected
     ...                  // these statements are executed at 8ns
     D2: [.C, b] -> [e];  // advance one CLK2 waveform, CLK1/CLK3 unaffected
     ...                  // these statements are executed at 8+13 = 21ns
     D3: [.C, c] -> [f];  // advance one CLK3 waveform, CLK1/CLK2 unaffected
     ...                  // these statements are executed at 21+18 = 39ns
     [a, b, c] -> [d];    // advance 9ns, CLK1/CLK2/CLK3 unaffected
     ...                  // these statements are executed at 39+9 = 48ns
     wait 5;              // advance 5ns, CLK1/CLK2/CLK3 unaffected
     ...                  // these statements are executed at 48+5 = 53ns

     report("time is %3.1f\n", _timeNow);    // displays 'time is 53.0'
}
```

Example 106

10.10 Triggered drive statements

A trigger function always has exactly one clock associated with it, and may only use the single drive statement associated with the function. This code is an example of a trigger function which uses a triggered drive:

```
DUT {
     module TEST(
       input CLK, D1, D2;
       output [15:0] Q);
     create_clock CLK period 15;    // default timescale (ns)
     @tfunc [CLK] -> [Q];           // triggered drive declaration
}
...
@tfunc() {                          // assume time at entry is t1
     ...                            // operating point: t1
     ->[x];                         // advance one CLK waveform
     ...                            // operating point: t1 + 15ns
     ->[y];                         // advance one CLK waveform
     ...                            // operating point: t1 + 30ns
}
```

Example 107

10.11 Manual DUT testing at the operating point

The purpose of a drive statement is to automate the driving of DUT inputs in preparation for a test, and the sampling and testing of DUT outputs. However, under some circumstances, these processes may also be carried out manually, if required, at an operating point. Whether or not a manual test is possible depends on the specific timing parameters required, as described below. Note that manual testing is always possible when default timing is used.

The code below shows an example of a manual DUT test. This code uses a drive statement to generate a clock waveform for a D-type F/F, but *explicitly* tests the `Q` output, rather than testing it as part of a drive statement:

```
DUT {
     module DType(input D, CLK, output Q);
     [CLK];         // just drive the CLK input; don't test any outputs
     ...declare the timing parameters: CLK period and waveform, tSU, and tCO
}

main() {
     // D, CLK, and Q are external variables, and may be read or written normally
     D = 1;         // will be correctly driven to meet any setup spec
     [.C];          // advance one CLK period to the next operating point
     if(Q == 1)     // manual sample may or may not be correct: see (10.11.2)
       report("passed\n");
     else
       report("failed\n");
}
```

Example 108

10.11.1 Input driving

The DUT inputs may always be driven at an OP, and the inputs are guaranteed to meet any specified setup parameters. This follows from the definition of a clock waveform (8.5.3); the compiler will report an error if the clock waveform does not meet this requirement.

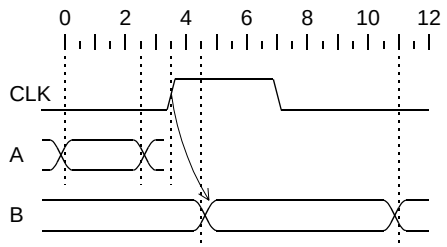
10.11.2 Output testing

Clocked DUT outputs are *not* guaranteed to be stable at an OP. For a clocked drive, if the setup times are relatively large compared to the clock period, and the output delay is also large compared to the clock period, then it is possible that the outputs will become valid *after* the start of the next operating point. Consider this example code, and the corresponding clock waveform:

```
DUT {
     module TEST(input CLK, A, output B)
     create_clock CLK period 10 waveform { 3.5 7 }  // default timescale (ns)
     A -> posedge CLK = (3.5 : -1.0)                // tSU is 3.5ns
     posedge CLK -> B = (1.0 : 7.5)                 // tCO is 7.5ns
}
```

Example 109

These are valid declarations, but the sum of the setup to the clock, and the output delay from the clock, is 11.0ns, while the total clock period is only 10.0 ns:



This is not a problem for a drive statement, since the drive statement pipelines the output test. However, when carrying out manual testing, the operating point occurs at 10.0 ns, which is *before* `B` is guaranteed to be valid.

If a manual test of the DUT outputs is necessary in these circumstances, then a wait statement can be executed at the operating point, to delay for 1 ns. However, this simply has the effect of stretching the clock low time by 1 ns, and has the same effect as changing the clock definition to:

```
create_clock CLK period 11 waveform { 3.5 7 }
```

10.11.3 Summary of manual testing requirements

The DUT outputs, and the `_vectorCount`, `_passCount`, and `_failCount` variables, may always be 'correctly' read at an operating point if this condition is satisfied:

- For all outputs which have synchronous output constraints, the specified t_{CO} occurs before the end of the defined clock waveform.

This condition is clearly not satisfied for the example in (10.11.2), since the output has a t_{CO} of 7.5ns, but the time available from the rising clock edge to the end of the waveform is only 6.5ns.

If this condition is not satisfied, then a drive statement will use a pipelined test to sample the output, and to update `_vectorCount`, `_passCount`, and `_failCount`, *after* the operating point. The user does not have the ability to do this by executing code *at* the operating point, and so will potentially sample incorrect DUT data.

11 RUN-TIME ERROR CHECKING

There are a number of program errors which cannot be detected until run-time, and which are described below. If a run-time error is detected, an error counter (errorCount) is incremented, and an error message is added to the logfile. The program will terminate when the run-time error count reaches the value specified as the `rte` parameter to either mtv or rtv (14.5). This parameter defaults to 1, so the default behaviour is to abort program execution when a run-time error is detected. Under some circumstances, it may be desirable and possible to continue execution; if this is the case, `rte` may be given a higher value.

Note that programmer-defined assertion errors are treated identically to run-time errors, and simply increase the error counter; the program will not abort until the `rte` limit has been reached.

Run-time errors should not be confused with DUT errors. A run-time error is the result of a programming error and is, as such, 'unexpected'; run-time errors should therefore normally result in a program abort. DUT errors, on the other hand, are (potentially) *expected* errors.

11.1 Array indexing errors

All array accesses are checked at runtime. Any access outside the declared range of the array is converted into an access to location 0, and an error is reported.

11.2 Bitslice indexing errors

Bitslice indexes are checked at runtime. Any access outside the declared range of the object is ignored, and an error is reported.

11.3 Checker Pipeline size errors

An error is issued if the maximum size of a checker pipeline was specified in a DUT declaration (8.5.4), and a drive statement subsequently attempts to access a pipeline level beyond this maximum size. If this happens, the first pipeline level is instead read or written.

11.4 Checker Pipeline over-write errors

A drive statement which specifies a pipeline level writes the expected data into that level of an internal checker pipeline. An error will be issued if a subsequent drive statement over-writes this expected data (9.2.4.2).

11.5 Trigger over-run

There may only ever be one executing instance of a given trigger function. An error will be reported if the conditions which led to the initiation of the trigger function again become true while the trigger function is already running. If this happens, the new start condition is ignored.

11.6 Last value pipeline errors

When the history of an object is read with the `'last` attribute (4.5.4.6) with a variable clock level, the clock level must evaluate to an integer which is in the inclusive range [1,*plevel*], where *plevel* is the declared pipeline level for that clock (8.5.4). If the clock level is outside this range an error will be reported, and an all-X value will be returned.

12 PREPROCESSOR

12.1 Introduction

The translation of a source file is carried out in two distinct stages. In the first, a *preprocessor* carries out a number of simple textual conversions on the source file. The preprocessor output is then used as input to the second stage of translation. This second stage is conventionally known as "compilation".

Translation is split into two stages for compatibility with other C-like languages (the Maia preprocessor is, for most intents and purposes, identical to the C preprocessor[1]). The primary purpose of preprocessing is to allow a *macro processor* to be run as a separate stage before compilation. This macro processor allows, among other things, the definition and expansion of *macros*, and the inclusion of additional source files. However, the preprocessor also includes other functionality which is not directly related to the macro processing functionality.

The macro processing language (MPL) has a syntax which is, with minor exceptions, unrelated to the syntax of the language being compiled (in this case, Maia). This chapter documents the functionality of the Maia preprocessor, and the MPL syntax.

The preprocessor is responsible for validating UTF-8 input, replacing trigraph and digraph character sequences, removing escaped LF characters ("line splicing"), comment removal, and carrying out macro operations in the MPL. The preprocessor stage is not required if all the following conditions are satisfied:

1. The source character set is ASCII (in other words, the source contains no multi-byte UTF-8 characters)
2. The source contains no trigraphs or digraphs
3. The source contains no escaped newlines
4. The source contains no operations in the MPL
5. The source contains no comments

The preprocessor has only minimal understanding of the lexical structure of a Maia program. It understands the form of strings, comments, constants, and identifiers, but does not otherwise carry out any tokenisation which is Maia-specific. It can therefore, in many situations, be used as a general-purpose textual preprocessor. However, the preprocessor emits *line directives* (12.3.3) in its output, to allow downstream tools to identify the current source file, and to keep track of the current line number in that file. These tools must therefore be capable of either processing, or ignoring, these directives.

[1] The primary differences are that the Maia preprocessor specifies UTF-8 as the input character set, and that there is no specific 'tokenisation' phase.

12.2 Preprocessor translation phases

The preprocessor carries out textual translation of the source file. This translation is split into a number of phases, which are carried out in the order defined by the paragraph headings below. The first 9 of these phases primarily carry out a number of simple character substitutions, UTF-8 validation, line concatenation, and comment removal.

The resulting source is then examined for operations in the MPL. These operations include *directive execution*, which is carried out in phase 10, and *macro expansion*, which is carried out in phase 11.

Conceptually, each of these phases is carried out separately, over the entire source file, before the next phase is started, with the single exception noted in step 2 of 12.3.1.1. However, the preprocessor may carry out the translation in any way that preserves the ordering defined by the paragraphs below. In particular, it is possible to carry out all preprocessing in a "line filter", operating only on the current line of input. When operating in this way, the preprocessor reads a single logical line of input (one or more physical lines separated by escaped newline characters), processes that line and then, if necessary, outputs that line.

Three of the initial transformation phases are optional, and are disabled by default. The preprocessor is, however, required to provide a mechanism to individually enable any, or all, of these three phases[1]. The optional phases are:

1. Trigraph replacement (12.2.1)
2. Digraph replacement (12.2.2)
3. Whitespace compression (12.2.9)

12.2.1 Trigraph replacement

The trigraphs are the 3-character sequences listed in Table 21. If trigraph processing is enabled, these sequences are replaced by their single-character equivalent[2].

Trigraph	Equivalent
??=	#
??(	[
??/	\
??)	]
??'	^
??<	{
??!	|
??>	}
??-	~

Table 21: trigraphs

[1] mtv 2019.9 carries out trigraph and digraph replacement, but does not compress whitespace. It does not currently provide a mechanism to disable trigraph or digraph replacement, or to enable whitespace compression.

[2] Trigraphs and digraphs may be required when, for example, a keyboard does not provide the equivalent character, or when a text editor reserves an equivalent character.

When replacement is enabled, all trigraph sequences are replaced, irrespective of context; in particular, trigraphs (and digraphs) within strings are also replaced. A trigraph within a string may be preserved by replacing a question mark with an escaped question mark:

```
report("(???)");      // produces '(?]'
report("(??\?)");     // produces '(???)'
```

12.2.2 Digraph replacement

The digraphs are two-character sequences which are a more compact equivalent of the most commonly used trigraphs, and are listed in Table 22. If digraph processing is enabled, these sequences are replaced by their single-character equivalent.

Digraph	Equivalent
<:	[
:>	]
<%	{
%>	}
%:	#
%:%:	##

Table 22: digraphs

12.2.3 Line terminator conversion

The code points and code point combinations listed in Table 23 are recognised as a single line terminator.

Code point	Name
U+000A	LF: line feed ("newline")
U+000C	FF: form feed
U+000D, U+000A	CR followed by LF
U+000D	CR: carriage return
U+0085	NEL: next line
U+2028	LS: Line separator
U+2029	PS: Paragraph separator

Table 23: line terminators

All these code points are converted into a `\n` character (LF, U+000A). The code point sequence U+000D, U+000A is tested before testing for a single U+000D; both are converted into a single LF character.

Any reference to a "line terminator" refers to the one or more code points which are used to terminate a user input line either during, or prior to, this phase. Any reference to a "newline" or to "LF" after this phase has completed refers to a single `\n` (LF, U+000A) character.

12.2.4 Whitespace conversion

The code points listed in Table 24 are recognised as whitespace.

One-byte code points
U+0009 U+000B U+0020

Two-byte code points			
U+00A0			
Three-byte code points			
U+1680	U+180E	U+2000	U+2001
U+2002	U+2003	U+2004	U+2005
U+2006	U+2007	U+2008	U+2009
U+200A	U+202F	U+205F	U+3000

Table 24: whitespace

All these code points, with the exception of HT ("horizontal tab", U+0009), are converted into a SP character (space, U+0020). Note that newline is not classified as whitespace. Any reference to "whitespace" after this phase has completed refers only to one or more consecutive HT or SP characters.

On completion of this phase, all line terminator and whitespace characters in the source will have been replaced with either LF (U+000A), SP (U+0020), or HT (U+0009).

12.2.5 UTF-8 validation

Characters are now checked for valid UTF-8 encoding. Any character with an invalid encoding[1] is rejected, with the exception that the two-byte sequence 0xC0, 0x80 is accepted as an overlong NUL[2].

12.2.6 Line continuation

Escaped newlines (a LF immediately preceded by a \ (U+005C) character) are stripped from the input, merging the current "physical" line with the next line to form a single "logical" line. A warning is issued if a \ character is the last non-whitespace character on the line, and is followed by one or more whitespace characters.

Line continuation is required only when it is necessary to split a preprocessor directive, or a string, over multiple physical lines[3].

12.2.7 String preservation

Strings are arbitrary character sequences which are enclosed in double quotation marks ("", U+0022). Strings are recognised during this phase and are preserved from further preprocessor transformations; they are passed unmodified to the output. The entire string must appear on a single logical line of input; an error is raised if the string has no closing quotation mark on the current logical line[4].

---

[1] Examples of invalid characters are characters which have an invalid byte count, or which have an overlong encoding, or which code more than 21 bits, or which have an invalid continuation byte.

[2] This exception is known as 'modified UTF-8'; it allows a NUL character to be placed into a string.

[3] The compiler itself is "free-form" and never requires input to be split over multiple lines using an escaped LF. "Line splicing" is relevant only to the preprocessor.

[4] Strings may be continued over multiple lines either by inserting an escaped LF within the string itself, or by placing adjacent strings on separate lines of input. In the former case, the preprocessor removes the escaped LF; in the latter, the compiler concatenates the adjacent strings.

Note that the filename argument to the `#include` directive may be specified as a string (12.3.2). This string is treated in the same way as any other string, and is protected from further transformation[1].

12.2.8 Comments

Both block and line comments are replaced with a single SP character.

12.2.9 Whitespace compression

If whitespace compression is enabled, the preprocessor replaces multiple consecutive SP characters with a single SP character.

12.2.10 Directive processing

Directives are instructions in the MPL, and are preceded by a `#` (U+0023) character. Directives are recognised and executed in this phase; see 12.3.

Phases 10 and 11 require partial tokenisation of the input in order to find identifiers, and to evaluate any arithmetic expressions which control conditional inclusion. However, this tokenisation is not required if there are no directives in the source, and is not treated as a separate phase.

12.2.11 Macro expansion

In the final phase, macros which have previously been defined by a `#define` directive are expanded; see 12.4. This phase differs from the previous phases in that it cannot be carried out in a single pass on the current line of input. Text which has been macro-expanded is rescanned until no more expansion is possible; this may require multiple passes over part or all of the current input line.

12.3 Preprocessor directives

If the first non-whitespace character on a line is `#` (U+0023), then the line is potentially a *preprocessor directive*. Any whitespace after the `#` character is ignored, and the remainder of the line is processed as a directive[2], unless one of the following three conditions is true:

- If the next character is a LF, this line is ignored and is not copied to the output (in other words, it is stripped from the input). This is a *null directive*;
- If the next character is `(` (U+0028), then the line is not considered to be a directive[3] (and is therefore subject to macro expansion in phase 11);

[1] For the C preprocessor, the string argument is treated as a special case and may later be macro-expanded.

[2] A directive must therefore appear on a single logical line of input. If it is necessary to split a directive over multiple physical lines, it can be continued either by a block comment which extends past the end of the line, or by escaping the line terminators with a `\` character (12.2.6).

[3] `#(` introduces a module parameter list in a module declaration; see 8.2.1.

- If the next 6 characters are `pragma`, followed by whitespace, then the entire line is protected from further transformation and is passed unmodified to the output[1]. This is a *pragma directive*; see 12.7.

The line is otherwise required to be a *pp-directive*.

Syntax

```
pp-directive :
  pp-cond-inclusion
  pp-control \n

pp-control :
  pp-include
  pp-line
  pp-warning
  pp-error
  pp-define
  pp-undefine
```

12.3.1 Conditional inclusion directives

A directive which has the form of one of the following

```
# ifdef  pp-identifier ...
# ifndef pp-identifier ...
# if     pp-condition  ...
```

introduces a *conditional inclusion* directive (a *pp-cond-inclusion*). The conditional inclusion directives allow a portion of the source file (a block, or *pp-cond-block*) to be conditionally included or excluded from preprocessing, according to the evaluation of a condition. The condition is evaluated as follows:

1. for the `#ifdef` directive, the condition evaluates to true if *pp-identifier* is currently defined as a macro name (in other words, a definition is currently in scope), and false otherwise. This condition is equivalent to `#if defined` *pp-identifier*.

2. for the `#ifndef` directive, the condition evaluates to true if *pp-identifier* is not currently defined as a macro name, and false otherwise. This condition is equivalent to `#if !defined` *pp-identifier*.

3. for the `#if` and `#elif` directives, *pp-condition* is evaluated as an arithmetic constant expression, using the procedure defined in 12.3.1.1. The condition evaluates to false if the expression evaluates to 0, and true otherwise.

Each directive in a conditional inclusion directive (a *pp-cond-inclusion*) is checked in order; only the block associated with the first condition that evaluates true is included. If none of the conditions evaluates to true, and there is a `#else` branch, the block associated with the `#else` branch is included. If there is no `#else` branch, none of the blocks associated with the *pp-cond-inclusion* is included.

[1] Macro expansion therefore does not occur inside a `#pragma` directive.

If *pp-cond-block* is excluded as a result of a condition evaluation, the preprocessor carries on analysing the text in the excluded *pp-cond-block* until it finds the matching *pp-elif-part*, *pp-else-part*, or *pp-endif-part*. The preprocessor is required to complete processing through to phase 10, and so will potentially report any errors detected in these phases, despite the fact that the block has been excluded. However, the preprocessor will not generate any output for these lines.

Syntax

```
pp-cond-inclusion :
  pp-if-part pp-elif-parts_opt pp-else-part_opt pp-endif-part

pp-if-part :
  # ifdef  pp-identifier \n pp-cond-block_opt
  # ifndef pp-identifier \n pp-cond-block_opt
  # if     pp-condition  \n pp-cond-block_opt

pp-elif-parts :
  pp-elif-parts pp-elif-part

pp-elif-part :
  # elif  pp-condition   \n pp-cond-block_opt

pp-else-part :
  # else                 \n pp-cond-block_opt

pp-endif-part :
  # endif \n

pp-cond-block :
  pp-cond-block pp-cond-block-part

pp-cond-block-part :
  pp-cond-inclusion
  pp-control \n
  text-line  \n
```

12.3.1.1 Condition evaluation

pp-condition is evaluated in four steps, in the order defined by the numbered items below.

1. The expression is examined for unary operators of the form

   `defined` *pp-identifier*

   or

   `defined (` *pp-identifier* `)`

   this operator evaluates to 1 if *pp-identifier* is currently defined as a macro name, and 0 otherwise.

2. Any macro invocations in *pp-condition* are expanded, using the procedure defined in 12.4 below. This replacement occurs before phase 11, and is the only violation of the evaluation-order rules defined in 12.2[1].

3. Any remaining *pp-identifier* tokens in *pp-condition* are replaced with 0.

4. The resulting expression should contain only whitespace, parentheses `(` and `)`, integer constants in the form of a Cinteger (2.7.1), and the operators defined in Table 25. The operators are listed in precedence order, with the highest precedence operators at the top of the table, and operators of equal precedence on the same row of the table. These operators are a subset of the full set of Maia operators, with the same precedence and associativity.

   The expression is evaluated using 64-bit precision, and an error is raised if any Cinteger constants cannot be represented in 64 bits.

   On completion, the *pp-condition* evaluates to false if the expression evaluates to zero, and true otherwise.

	Operator				Associativity
Unary	`!`	`~`	`+`	`-`	right to left
Multiplicative	`*`	`/`	`%`		left to right
Additive	`+`	`-`			left to right
Shift	`<<`	`>>`			left to right
Comparison	`<`	`<=`	`>`	`>=`	left to right
Equality	`==`	`!=`			left to right
Binary AND	`&`				left to right
Binary XOR	`^`				left to right
Binary OR	`|`				left to right
Logical AND	`&&`	`and`			left to right
Logical OR	`||`	`or`			left to right

Table 25: MPL operators

12.3.2 include directives

The include directive inserts the contents of the named file into the current source file, at the point at which the include directive appears. A line directive is also inserted prior to the first line of the included file, and after the last line of the included file, to allow the compiler to correctly track source file locations. There is no practical limit to the level at which include directives may be nested; the current source file is always closed before inserting the included file, and is re-opened when the included file has been processed and closed.

The specified filename may be either a rooted absolute filename, or a relative filename. When the `"filename"` syntax is used, relative filenames are searched for in a location relative to the current source file. If the required file is not found in this location, it is searched for in the same directories which are searched for the `<filename>` syntax.

[1] The C preprocessor also allows the argument of a `#include` directive to be macro-expanded before source file inclusion is carried out. This feature is not supported. Allowing this exception would not add any functionality that cannot easily be achieved while keeping strict phase ordering.

The `<filename>` syntax is used when searching for system files. No system file directories are specified for mtv 2019.9, and `<filename>` is currently treated identically to `"filename"`.

Syntax

```
pp-include :
  # include <filename>
  # include "filename"
```

12.3.3 Line directives

A line directive may be used to change the preprocessor's record of the current line number and, optionally, the current filename. This directive might be of use, for example, for external tools which themselves generate or process source code. The preprocessor also generates line directives in its own output; the compiler uses this information when generating warnings and errors.

The line number is supplied as *line-number*, which should be a decimal integer. The preprocessor will restart line numbering such that the next input line after this directive will be considered to have this line number.

The filename is optional; if it is not provided, the current filename remains unchanged.

Syntax

```
pp-line :
  # line line-number "filename"opt

line-number : pp-integer
```

12.3.4 Warning and error directives

The preprocessor issues a warning when it encounters a warning directive, and an error when it encounters an error directive. If *warning-text* or *error-text* is present, it is copied verbatim to the warning or error output, respectively.

Syntax

```
pp-warning :
  # warning warning-textopt

pp-error :
  # error error-textopt
```

12.3.5 define directives

12.3.5.1 Introduction

A macro definition associates the specified *replacement text* (or "macro body") with an identifier (the "macro name"). There is a single namespace for macro names[1]. The identifiers `defined`, `and`, and `or` may not be used as macro names.

[1] It is therefore not possible to define an object-like and a function-like macro with the same name, or to define multiple function-like macros with the same name, even if they have different numbers of arguments.

The scope of this association, or *macro definition*, lasts until a corresponding `#undef` directive for the same macro name, or until preprocessing completes if no `#undef` directive is found. The `#undef` directive need not occur in the same source file.

Within the scope of a macro definition, any valid invocation of the macro is replaced by the corresponding replacement text during phase 11 (see 12.4 below). Whitespace surrounding the replacement text is not significant, and is removed before the replacement occurs.

The replacement text may be empty; in this case, the macro invocation is simply removed from the output during the replacement phase.

Syntax

```
pp-define :
  object-like-macro-defn
  function-like-macro-defn

object-like-macro-defn :
  # define macro-name replacement-text

function-like-macro-defn :
  # define macro-name-lparen formal-param-list ) replacement-text

macro-name : pp-identifier

formal-param-list :
  formal-param
  formal-param-list , formal-param

formal-param : pp-identifier

macro-name-lparen :: {pp-identifier}(

replacement-text :: .*
```

12.3.5.2 Object-like macros

A directive of the form

```
# define macro-name replacement-text \n
```

defines an *object-like* macro with name *macro-name*, and with associated replacement text *replacement-text*. Subsequent occurrences of the macro name within the scope of the definition are replaced with the associated replacement text during phase 11; see 12.4 below.

12.3.5.3 Function-like macros

A directive of the form

```
# define macro-name-lparen formal-param-list ) replacement-text \n
```

defines a *function-like* macro. A function-like macro definition is syntactically similar to a function definition. The macro name is given by *macro-name-lparen*, which is the macro name, immediately followed by a `(` character, with no intervening whitespace. If there is whitespace between the macro name and the `(` character, the directive is instead interpreted as an object-like macro definition:

```
#define foo bar1 bar2 // object-like: 'foo' is replaced with 'bar1 bar2' in phase 11

// function-like: 'a(1,2)' is replaced with '((1)+(2))' in phase 11
#define a(x, y) ((x)+(y))

// object-like: 'b(1,2)' is replaced with '(x, y) ((x)+(y))(1,2)' in phase 11
#define b (x, y) ((x)+(y))
```

Example 110

Within the definition, an optional comma-separated list of identifiers (*formal-param-list*) names the "formal parameters" to the macro. The scope of a formal parameter lasts from its introduction in the parameter list to the end of the macro definition (in other words, to the newline which terminates the current logical line). The formal parameter names must be unique within the macro definition.

Subsequent occurrences of the macro name, when within the scope of the definition and when followed by a list of actual parameters enclosed in parentheses, are replaced with the associated replacement text during phase 11; see 12.4 below.

12.3.5.4 Macro redefinition

Within the scope of a macro definition, an object-like macro name may be redefined only if the replacement is also an object-like macro, and the replacement text (including any whitespace inside the replacement text) is identical. A function-like macro may be redefined only if the replacement is a function-like macro with the same number of parameters, and the parameters and replacement text (including any whitespace inside the replacement text) are identical.

12.3.6 undef directive

If *pp-identifier* is currently defined as a macro, `#undef pp-identifier` will remove that definition. The directive is ignored if *pp-identifier* is not currently defined as a macro.

Syntax

```
pp-undefine :
  # undef pp-identifier
```

12.4 Macro expansion

Macro expansion takes place in phase 11. The source is tokenised (12.5) to find any occurrences of a *pp-identifier* which is an in-scope macro name. The macro invocation is then replaced by the corresponding replacement text, subject to the constraints described in this section.

12.4.1 Self-referential macros

A macro may not, directly or indirectly, reference itself. If a macro invocation is found with the same name as a macro which is currently being replaced, then the invocation is ignored and is not expanded. In general, when the preprocessor encounters a *pp-identifier* which has previously been defined as a macro name, it may already be in the process of recursively expanding a stack of *pp-identifier*s. The current *pp-identifier* is not expanded if it appears anywhere in this stack. This is not treated as an error condition.

```
#define FOO     BAR
#define BAR     FOO
#define A(x, y) BAR

BAR               // 2-level expansion, but stops after 1 level; outputs FOO
A(1,2)            // 3-level expansion, but stops after 2 levels; outputs FOO
```

Example 111

12.4.2 Object-like macro expansion

A *pp-identifier* which is currently defined as an object-like macro is replaced with the corresponding replacement text[1], unless the preprocessor is currently replacing another instance of the macro named by *pp-identifier* (12.4.1).

When replacement of the *pp-identifier* has completed, the scan process restarts at the first character of the replacement text. The replacement text may therefore contain further complete or partial invocations of object- or function-like macros, and these invocations are themselves replaced, until no further replacements are possible.

```
#define FOO       BAR(1,  // 'FOO' expands to a partial invocation of 'BAR'
#define BAR(x, y) 2*x+y
FOO
    3)                    // eventually expands to '2*1+3'
```

Example 112

12.4.3 Function-like macro expansion

If the preprocessor identifies a *pp-identifier* which is currently defined as a function-like macro, it carries on to locate the next character which is not whitespace and which is not a newline. If this character is `(`, the identifier is treated as an invocation of a function-like macro; it is otherwise ignored, and copied to the output without modification.

The text between the outermost pair of matching parentheses following the macro name forms the macro argument list (the "actual parameters"). It is an error if the source does not contain a closing parenthesis after the argument list.

Individual parameters are separated by a comma character, unless that comma character is enclosed within a pair of parentheses which are not the outer-most pair of parentheses[2]. The number of arguments (including empty arguments) must match the number of formal parameters in the macro definition[3]. Whitespace before or after an argument is not significant, and is not substituted into the replacement text. If an argument is not present, or is composed entirely of whitespace, then it is considered to be an "empty" argument, and the corresponding formal parameter is omitted from the expanded replacement text.

[1] Strings and comments are processed in phases 7 and 8, respectively. Macro expansion therefore does not occur in either strings or comments.

[2] An argument may therefore contain matched pairs of parentheses, but not unmatched parentheses.

[3] 2019.9 does not support variable argument lists.

Within the text forming an invocation of a function-like macro, any newline characters following the macro name are treated as whitespace. An invocation of a function-like macro may therefore appear on more than one logical line of source.

```
#define A(x) x+y
A(1)   // expands to '1+y'
A      // not a macro invocation; the preprocessor outputs 'A'
A()    // an invocation of 'A' with one empty argument; preprocessor outputs '+y'
A(1,2) // an error: an invocation of 'A' requires exactly one argument

#define G(x,y) x+y
1G(,)2      // 2 empty arguments; preprocessor expands G to '+', and outputs '1+2'[1]
G(,2)       // preprocessor outputs '+2'
G(1,  )     // preprocessor outputs '1+'
G(1,
  2)        // preprocessor outputs '1+2'
G((1,2),3)  // preprocessor outputs '(1,2)+3'
```

Example 113

12.4.3.1 Argument substitution

The replacement text is scanned for occurrences of the macro's formal arguments. If a formal argument is found and is preceded by a `#` character, the corresponding actual argument is *stringified*; the result is then substituted into the replacement text in place of the formal argument and the preceding `#` character (12.4.3.2).

Otherwise, the actual argument is macro-expanded, and is then substituted into the replacement text in place of the formal argument (this is known as *argument prescan*). This substitution is carried out recursively, in the same way as for any other expansion; however, it is only the argument itself which is expanded (the substitution process will not attempt to read any text beyond the comma character or closing parenthesis which terminates the argument).

When argument substitution has completed, the entire replacement text is scanned for further replacements. This process continues until no more expansion is possible.

```
#define E      D
#define C(x,y) [x+y]
#define D      C(1,2)
#define F(a,b) a+#b

// the first 'E' in this invocation is macro-substituted; the second is stringified.
// the preprocessor eventually outputs '[1+2]+"E"'
F(E, E)
```

Example 114

12.4.3.2 The # operator

When a formal parameter in the replacement text is preceded by a `#` character the corresponding actual is not macro-expanded, and is instead enclosed in double-quote (", U+0022) characters before

[1] The C preprocessor parses `1G` as a "preprocessing number", and so does not recognise a macro in this case; it instead outputs `1G(,)2`

substitution into the replacement text. Leading and trailing whitespace around the actual argument is ignored.

The stringified actual is not subject to further macro replacement.

```
#define FOO  BAR
#define G(a) #a
G( FOO )         // preprocessor outputs '"FOO"', not '"BAR"'

#define TEST(expr)                          \
  do {                                      \
    if(!( expr ))                           \
      report("test " #expr " failed\n"); \
  } while(0)

// this invocation is replaced by:
//   do { if(!( a+b ) report("test " "a+b" " failed\n"); } while(0);
TEST
  (a+b);
```

Example 115

12.5 Tokenisation

The preprocessor has no specific tokenisation phase. However, tokenisation is required during phases 10 and 11, for the following reasons:

1. Macro names must be identifiers (a *pp-identifier*). A pp-identifier is therefore required as the operand of the `#ifdef`, `#ifndef`, and `#undef` directives, and as the operand of the `defined` operator. During phase 11, all occurrences of a *pp-identifier* are compared against macro names which are currently in scope.

2. Macro formal parameters must be a *pp-identifier*.

3. The controlling expression of the `#if` and `#elif` directives (12.3.1) must be evaluated as a constant expression.

In these contexts, the preprocessor classifies the remaining unprotected[1] input into *Vinteger* tokens, *Cinteger* tokens, *pp-identifier* tokens, MPL operator tokens, and punctuator tokens[2], where:

i. A Vinteger is a preprocessor Verilog-style integer, and is defined identically to a Maia Vinteger (2.7.2);

ii. A Cinteger is a preprocessor C-style integer, and is defined identically to a Maia Cinteger (2.7.1);

iii. A *pp-identifier* is a preprocessor identifier, and is defined identically to a Maia identifier (2.5);

[1] Protected input includes strings and pragma directives.

[2] The preprocessor does not identify floating constants (2.7.3) in 2019.9. Any part of a floating constant which has the same form as an identifier is therefore subject to macro expansion.

iv. The MPL operators are the operators defined in Table 25, together with parentheses `(` and `)`;

v. A punctuator is a minimal-length sequence of characters which cannot be classified as one of the other types above.

12.5.1 Preprocessor Identifiers

A *pp-identifier* is a preprocessor identifier, and is defined identically to a Maia identifier (2.5):

Syntax

```
pp-identifier ::
   [{ident-alpha}_][{ident_alpha}_0-9]*
ident-alpha ::
   [U+0061-U+007A] | [U+0041-U+005A] |
   [U+0080-U+0084] | [U+0086-U+2027] | [U+202A-U+10FFFF]
```

Preprocessor identifiers therefore consist of a combination of the alphabetic characters (*ident-alpha*), the decimal digits `0` to `9`, and underscore (_, U+005F). The first character may not be a decimal digit.

The "alphabetic characters" are defined as `a` through `z`, `A` through `Z`, and all multibyte UTF-8 characters, with the exception of the multibyte line terminators (12.2.3), and the multibyte whitespace characters (12.2.4).

12.5.2 constant expression evaluation

The controlling expression of the `#if` directive is evaluated as a constant expression (12.3.1.1). The expression must contain only whitespace, parentheses `(` and `)`, the `defined` operator, pp-identifiers, Cintegers, and the operators defined in Table 25.

12.6 Predefined macro names

The following macro names are predefined:

Name	Value
`__MTV__`	`1`
`__MAIA__`	`1`
`__MTV_VERSION__`	The mtv compiler version, as a 32-bit integer, in the same format as the predefined `_version` variable (p21). `__MTV_VERSION__` currently has the same value as `_version`, but this is not guaranteed.
`__MAIA_VERSION__`	The supported Maia version, as a 32-bit integer. `__MAIA_VERSION__` has the same value as `_version` (p21).
`__VHDL_TARGET__`	Will be set to 1 if the code generator is producing VHDL output, or undefined otherwise
`__VERILOG_TARGET__`	Will be set to 1 if the code generator is producing Verilog output, or undefined otherwise
`__MSWINDOWS__`	Will be set to `1` if running on Windows, or undefined otherwise

__UNIX__	Will be set to 1 if running on a Unix-like system, or undefined otherwise
__FILE__	The current source file name, as a string
__LINE__	The current source file line number, as a decimal integer
__DATE__	The compilation date, as a string in the format "mmm dd yyyy" (for example, "Apr 22 2019")
__TIME__	The compilation time, as a string in the format "hh:mm:ss" (for example, "17:20:56")

Table 26: predefined macro names

12.7 Pragma directives

Any directive of the form `#pragma` is ignored by the preprocessor, and is copied directly to the output. The Maia pragma directives are therefore handled by the compiler, and not by the preprocessor; they are, however, documented here for clarity. The supported pragmas are:

`#pragma _DefaultWordSize n`

Sets the size of implicit variables, and variables declared using the `int` keyword, to *n* bits. Unsized integer constants are also scanned to the number of bits specified by `_DefaultWordSize`. *n* may be any value from 1 up to a compiler determined maximum, which will be at least 2^{24}. `_DefaultWordSize` itself has a default value of 32.

`#pragma _Implicits n`

Enables (*n* = 1) or disables (*n* = 0) the use of implicit variables (variables which auto-declare themselves on first use). Implicits are disabled by default.

`#pragma _StrictChecking n`

Sets the level of static type checking which is carried out during compilation; see (3.1). Level 0 defines a level of weak checking; this is strengthened as *n* is increased. The default level is 1.

The `_Implicits`, `_StrictChecking` and `_DefaultWordSize` pragmas are program-wide, and should appear once in the source code, before any functions are analysed.

13 GLOSSARY

Aggregate object	A compound object which is a collection of scalar objects. If the scalar objects are all of the same type then the collection is homogeneous (an array); sotherwise, the collection is heteregenous (a structure).
Arithmetic object	Any object of an arithmetic type.
Arithmetic type	A type which supports arithmetic operations: `int`, `bit`, and `var`. If `_StrictChecking` is less than 2, `bool` is a synonym for `bit1`, and so is also an arithmetic type.
Assignment Compatible	Object *lhs* and *rhs* are assignment-compatible if the expression *lhs=rhs* is allowable.
Associativity	Operator associativity determines the order in which the sub-expressions in a full expression are evaluated, when the operators have the same precedence. In the expression `a=b*c/d`, for example, `*` and `/` have the same precedence, and associate left-to-right; the expression is therefore evaluated as `a=(b*c)/d`.
Bit	A unit of data storage sufficient to hold an `int1` or a `var1` object. An `int1` may take on one of the values `0` or `1`; a `var1` may take on one of the values `0`, `1`, `X`, or `Z`.
Constant	A lexical element which represents a numeric or boolean value. A constant is not an object. In some circumstances, however, the compiler can be considered to create a temporary object which is initialised with the value of the constant.
Constant expression	An arithmetic expression which can be evaluated during compilation; any combination of constants and operators. With few exceptions, a constant expression can be used wherever a constant is required in the syntax.
Data object	Any object of a data type.
Data type	A type which can be considered to hold 'data': the arithmetic types, and `kmap`.
Declaration	A declaration specifies the interpretation given to an identifier; a declaration that also reserves storage is a definition. A type (structure or stream) declaration does not create storage for a new object; it simply tells the compiler how much storage will be required, should an object of that type be declared.
Definition	A declaration which reserves storage and creates an object.
Field	See *member*
ivar object	Any object of an `int`, `bit`, or `var` type
LHS	Left hand side
lvalue	A writeable object; the left-hand-side, or destination, of an assignment

Member	An entity (*member*, or *field*) inside a structure or stream; see also *tag*
Object	A region of data storage, which may be readable, writeable, or both. If the object is readable, it yields a value when read. Every object has an associated type, which determines the interpretation of the value stored in the object, and the operations allowed on that object.
OP	Operating point
Precedence	Operator precedence determines the order in which the sub-expressions in a full expression are evaluated. In the expression `a=b+c*d`, for example, `*` has a higher precedence than `+`, and the expression is therefore evaluated as `a=b+(c*d)`. See also associativity.
Rank	The *rank* of an expression or object is defined as its dimensionality. If `a` is a 3-dimensional array, for example, it has rank 3. The expression `a[i]` has rank 2; the expression `a[i][j]` has rank 1; and the expression `a[i][j][k]` has rank 0. Any scalar object has rank 0.
RHS	Right hand side
rvalue	A readable object or expression; the right-hand-side of an assignment
Scope	For an object which has an identifier, the scope of that identifier is the region of the source code in which the identifier may be used to access that object
Tag	The name associated with a structure or stream declaration; for example, this declaration has the *tag* `a`, and has one *member*, `b`: `struct a { int b; }`
void expression	A void expression has no value (a call of a function which has been declared to be of type `void`, for example). If an expression of any other type is evaluated as a void expression, its result is discarded; in this case, the expression is evaluated solely for its side-effects.

14 MTV

This chapter documents features and issues which are specific to mtv, or the current implementation of mtv or a specific code generator, but which are not part of the language specification.

14.1 Preprocessor

A number of macro names are predefined, and are listed in Table 26 above. Macros may be defined or cancelled on the `mtv` (or `rtv`) command line, with these switches:

`-D NAME`	Predefine `NAME` as a macro, with definition '1'
`-D NAME=DEFINITION`	Predefine `NAME` as a macro, with definition `DEFINITION`
`-U NAME`	Cancel any previous definition of `NAME`, either built in or provided with a `-D` option

The `-D` and `-U` options are processed in the order in which they appear on the command line. The `MTV_CPPOPTIONS` environment variable may also be used to provide additional options to the preprocessor; the contents of this environment variable are processed before any additional `-D` or `-U` options.

The target language is set by mtv's `-target` option, or by the suffix of the output file; it is not overridden by the `__VHDL_TARGET__` and `__VERILOG_TARGET__` macros, which should not normally be changed.

`mtv` does not currently directly accept a **`-I`** switch to specify include file directories; these switches should instead be specified in the `CPP_OPTIONS` environment variable.

14.2 Environment variables

Table 27 lists the environment variables which are understood by `mtv`. The compiler may not function (or may appear not to function) if these variables are incorrectly set; they should be checked after installation.

Variable name	Default	Function
`MTV_PPENABLE`	1	Enable (1) or disable (0) the preprocessor stage
`MTV_KEEPCPP`	0	The preprocessor output may be retained by setting `MTV_KEEPCPP` to 1. The output will be in a temporary file with an 'mtv_' prefix, either in the current directory, or a system-defined temporary directory.
`MTV_CPPOPTIONS`	Unset	This string is appended to the cpp command line; it may be used to pass any macro definitions to the preprocessor.
`MTV_INPUT_FILE`	test.tv	The name of the top-level input file; this variable is ignored if mtv is invoked with the '-i' switch
`MTV_OUTPUT_FILE`	test.v	The name of the Verilog testbench output file; this variable is

		ignored if mtv is invoked with the '-o' switch
MTV_LOGENABLE	0	Set to 1 to increase the level of compiler diagnostic output. This name is deprecated, and will change.
MTV_COPY_STDOUT	1	If set, the compiler and testbench output is copied to 'stdout', as well as being written to the logfile. Set to 0 to disable.
MTV_LOGFILE	mtv.log	The name of the compiler and testbench logging file

Table 27: mtv environment variables

rtv, the compiler driver, also requires a number of environment variables. These variables have no default values; they must be set to valid values during installation. These variables are listed in Table 28.

Variable name	Default	Function
RTV_CONFIG		The full pathname of the `rtv` configuration file
RTV_SIMULATOR		The name of the required simulator. This name refers to an entry in the configuration file
MAIA_COMPILER		The `mtv` executable invoked by `rtv`

Table 28: rtv environment variables

14.3 Compiler logging

All output from mtv, and from a running testbench, is written to a logfile. The output may also optionally be displayed on stdout. Logging is controlled by a number of environment variables; see, in particular, `MTV_COPY_STDOUT`, and `MTV_LOGFILE`. These variables should only be changed if necessary; the compiler may appear not to function if logging is disabled.

14.4 Sizing iterations

mtv must determine the maximum possible size of any unconstrained formals or function return values. In general, this involves running a sizing pass which analyses chains of assignments to unconstrained objects. Almost all programs will complete this sizing within a single iteration, but in some complex circumstances multiple iterations will be required. mtv defaults to a maximum of 10 iterations before reporting an error (E191). The maximum number of iterations may alternatively be set with mtv's `-si` switch. '`-si 20`', for example, sets the maximum number of iterations to 20.

If sizing does not complete within the default number of iterations it is likely that the user code contains an erroneous loop involving a cycle of chained unconstrained objects.

14.5 Assertion and runtime failures

The `-rte n` switch sets the maximum run-time error count to `n`. The HDL code will terminate when this count is reached. Run-time errors occur for conditions such as out-of-range array accesses and

assertion failures, but do not include DUT failures. The default value of `n` is 1; in other words, a program will, by default, terminate when it encounters any assertion or run-time error.

The equivalent rtv switch is `--rte n`:

```
# terminate after 10 assertion or runtime failures:
> mtv -rte 10 test.tv
# is equivalent to:
> rtv --rte 10 test.tv
```

14.6 DUT failures

The `-fail n` switch sets the maximum DUT failure count to `n`. The HDL code will terminate when this count is reached. A DUT failure is defined as any failure of the DUT to match an output expected from a drive statement; it is not a program error. The default value of `n` is 20.

The equivalent rtv switch is `--fail n`:

```
# terminate after 10 DUT failures:
> mtv -fail 10 test.tv
# is equivalent to:
> rtv --fail 10 test.tv
```

14.7 Verilog code generator limitations

mtv's Verilog code generator[1] has a number of limitations, which are described below. The compiler will issue a warning or an error when it detects these conditions, unless noted otherwise.

14.7.1 Floating-point operations

All the floating-point operations are supported for expressions which can be statically evaluated. However, Verilog supports only a 64-bit floating type, while Maia supports three of the IEC 60559 types. Any Maia expression which requires runtime evaluation, and which cannot be fully evaluated using the Verilog 64-bit type, will be reported as an error during compilation.

14.7.2 report statements

Verilog simulators have widely differing support for width and precision specifications. Both are required to be supported for floating-point conversions, but the standard says nothing about the remaining conversions. '`%6d`', for example, correctly produces an integer in a 6-character field for two popular simulators, but produces garbled output on a third. Maia produces a warning rather than an error when it detects this condition; you will have to check whether the output produced by your simulator is acceptable, and modify the code if not.

Individual Verilog simulators also have widely differing support for the underlying `$write` system task, so report statements with complex formatting requirements are likely to display differently on different

[1] The code generator produces Verilog which conforms to IEEE1364-2005. This was the final LRM release for 'plain' Verilog. No SystemVerilog code is generated.

simulators, or possibly not at all. No error or warning messages are generated if the output does not conform to the `report` specification.

14.7.3 Mode 2 stream conversion specifications

Mode 2 stream conversion specifications (3.7.11.2.3) cannot be fully supported, because of the limitations of the underlying `$write` system task; see (14.7.2).

14.7.4 Exit code

A program terminates and returns an exit code either by executing an exit statement (6.11), or by returning a value from `main`. This code is not returned to the operating system, as Verilog has no support for this. However, the code is written to the logfile, if the logfile is enabled. If test runs are to be automated, the test script should search the logfile for the exit code, rather than using the value returned by the Verilog simulator.

14.7.5 Recursion

The Verilog code generator does not support recursive function calls[1] (in other words, a Maia function may not directly or indirectly call itself). mtv's '-cg' switch produces a call graph, which may be viewed with graphviz; the graph can be used to analyse illegal function call sequences.

14.7.6 Scheduling

Verilog's scheduling model is ambiguous, particularly with respect to the issue of the atomicity of different 'processes'. The LRM doesn't explicitly state that processes should de-schedule only at defined points, and leaves the option open for arbitrary process interleaving.

If the scheduler is implemented as defined, then it potentially has a number of undesirable effects (and no benefits). It is unlikely, for these reasons, that any vendor actually uses an interleaved scheduling model.

If a specific simulator does implement interleaved scheduling, then Maia is potentially affected if two or more concurrent functions attempt to modify a shared variable at the same time (in other words, the functions must have the same Operating Point). In this case, it is possible that the shared variable will take on an incorrect value.

[1] While Verilog-2005 does support 'auto' functions, this support is not sufficient to allow recursive function calls, except in simple cases.

15 FLOATING-POINT ARITHMETIC EXAMPLE

The example program below uses the BBP formula to calculate π to 15 decimal places, which is the best that can be represented in IEC 60559 64-bit precision (a `real2`). The program executes two report statements. The first simply outputs the 'π' variable (which is initialised from a constant which is correct to 16 decimal places), while the second outputs the result of the BBP calculation.

```
#pragma _strictChecking 0

main() {
   var64 π = 3.1415926535897932;
   real2 bbp[11];

   for(i=0; i<11; i++) {
      bbp[i] = term(i);
      if(i > 0)
         bbp[i] = bbp[i] .F+ bbp[i-1];
   }

   report("%19.16f\n", π);
   report("%19.16f\n", bbp[10]);
}  // main()

term(k) {
   real2 t1 = 1.0;
   for(i=0; i<k; i++)
      t1 = 16.0 .F* t1;
   t1 = 1.0 .F/ t1;
   return
      t1 .F* (
         (4.0 .F/ ((8.0 .F* (real2)k) .F+ 1.0)) .F-
         (2.0 .F/ ((8.0 .F* (real2)k) .F+ 4.0)) .F-
         (1.0 .F/ ((8.0 .F* (real2)k) .F+ 5.0)) .F-
         (1.0 .F/ ((8.0 .F* (real2)k) .F+ 6.0)));
}
```

Example 116

The program output is:

```
 3.1415926535897931
 3.1415926535897931
```