Bluespec™ SystemVerilog
Version 3.8
Reference Guide

Revision: 24 March 2006

Trademarks and copyrights

Verilog is a trademark of IEEE (the Institute of Electrical and Electronics Engineers). The Verilog standard is copyrighted, owned and maintained by IEEE.

VHDL is a trademark of IEEE (the Institute of Electrical and Electronics Engineers). The VHDL standard is copyrighted, owned and maintained by IEEE.

SystemVerilog is a trademark of Accellera, Inc. The SystemVerilog standard is owned and maintained by Accellera.

Bluespec is a trademark of Bluespec, Inc.

Contents

1 Introduction

Bluespec SystemVerilog (BSV) is aimed at hardware designers who are using or expect to use Verilog [IEE01], VHDL [IEE02], or SystemVerilog [Acc04] to design ASICs or FPGAs. BSV is based on a synthesizable subset of SystemVerilog, including SystemVerilog types, modules, module instantiation, interfaces, interface instantiation, parameterization, static elaboration, and "generate" elaboration. BSV can significantly improve the hardware designer's productivity with some key innovations:

- It expresses synthesizable behavior with *Rules* instead of synchronous `always` blocks. Rules are powerful concepts for achieving *correct* concurrency and eliminating race conditions. Each rule can be viewed as a declarative assertion expressing a potential *atomic* state transition. Although rules are expressed in a modular fashion, a rule may span multiple modules, i.e., it can test and affect the state in multiple modules. Rules need not be disjoint, i.e., two rules can read and write common state elements. The BSV compiler produces efficient RTL code that manages all the potential interactions between rules by inserting appropriate arbitration and scheduling logic, logic that would otherwise have to be designed and coded manually. The atomicity of rules gives a scalable way to avoid unwanted concurrency (races) in large designs.
- It enables more powerful generate-like elaboration. This is made possible because in BSV, actions, rules, modules, interfaces and functions are all first-class objects. BSV also has more general type parameterization (polymorphism). These enable the designer to "compute with design fragments," i.e., to reuse designs and to glue them together in much more flexible ways. This leads to much greater succinctness and correctness.
- It provides formal semantics, enabling formal verification and formal design-by-refinement. BSV rules are based on Term Rewriting Systems, a clean formalism supported by decades of theoretical research in the computer science community [Ter03]. This, together with a judicious choice of a design subset of SystemVerilog, makes programs in BSV amenable to formal reasoning.

This manual is meant to be a stand-alone reference for BSV, i.e., it fully describes the subset of Verilog and SystemVerilog used in BSV. It is not intended to be a tutorial for the beginner. A reader with a working knowledge of Verilog 1995 or Verilog 2001 should be able to read this manual easily. Prior knowledge of SystemVerilog is not required.

1.1 Meta notation

The grammar in this document is given using an extended BNF (Backus-Naur Form). Grammar alternatives are separated by a vertical bar ("|"). Items enclosed in square brackets ("[ ]") are optional. Items enclosed in curly braces ("{ }") can be repeated zero or more times.

Another BNF extension is parameterization. For example, a *moduleStmt* can be a *moduleIf*, and an *actionStmt* can be an *actionIf*. A *moduleIf* and an *actionIf* are almost identical; the only difference is that the former can contain (recursively) *moduleStmt*s whereas the latter can contain *actionStmt*s. Instead of tediously repeating the grammar for *moduleIf* and *actionIf*, we parameterize it by giving a single grammar for <*ctxt*>*If*, where <*ctxt*> is either *module* or *action*. In the productions for <*ctxt*>*If*, we call for <*ctxt*>*Stmt* which, therefore, either represents a *moduleStmt* or an *actionStmt*, depending on the context in which it is used.

2 Lexical elements

BSV has the same basic lexical elements as Verilog.

2.1 Whitespace and comments

Spaces, tabs, newlines, formfeeds, and carriage returns all constitute whitespace. They may be used freely between all lexical tokens.

A *comment* is treated as whitespace (it can only occur between, and never within, any lexical token). A one-line comment starts with `//` and ends with a newline. A block comment begins with `/*` and ends with `*/` and may span any number of lines.

Comments do not nest. In a one-line comment, the character sequences `//`, `/*` and `*/` have no special significance. In a block comment, the character sequences `//` and `/*` have no special significance.

2.2 Identifiers and keywords

An identifier in BSV consists of any sequence of letters, digits, dollar signs `$` and underscore characters (`_`). Identifiers are case-sensitive: `glurph`, `gluRph` and `Glurph` are three distinct identifiers. The first character cannot be a digit.

BSV currently requires a certain capitalization convention for the first letter in an identifier. Identifiers used for package names, type names, enumeration labels, union members and type classes must begin with a capital letter. In the syntax, we use the non-terminal *Identifier* to refer to these. Other identifiers (including names of variables, modules, interfaces, etc.) must begin with a lowercase letter and, in the syntax, we use the non-terminal *identifier* to refer to these.

As in Verilog, identifiers whose first character is `$` are reserved for so-called *system tasks and functions* (see Section 12.8).

There are a number of *keywords* that are essentially reserved identifiers, i.e., they cannot be used by the programmer as identifiers. Keywords generally do not use uppercase letters (the only exception is the keyword `valueOf`). BSV includes all keywords in SystemVerilog. All keywords are listed in Appendix A.

The types `Action` and `ActionValue` are special, and cannot be redefined.

2.3 Integer literals

Integer literals are written with the usual Verilog and C notations:

intLiteral	::=	`'0` | `'1` | *decLiteral* | *hexLiteral* | *octLiteral* | *binLiteral*
decLiteral	::=	*decDigits*
	|	[*bitWidth*] (`'d` | `'D`) *decDigits*
hexLiteral	::=	[*bitWidth*] (`'h` | `'H`) *hexDigits*
octLiteral	::=	[*bitWidth*] (`'o` | `'O`) *octDigits*
binLiteral	::=	[*bitWidth*] (`'b` | `'B`) *binDigits*
bitWidth	::=	*decDigits*
decDigits	::=	1 or more consecutive characters from the set `0`...`9`
hexDigits	::=	1 or more consecutive characters from the sets `0`...`9`, `a`...`f`, `A`...`F`
octDigits	::=	1 or more consecutive characters from the set `0`...`7`
binDigits	::=	1 or more consecutive characters from the set `0`...`1`

Note that there is no leading `+` or `-` in the syntax for integer literals. Instead, we provide unary prefix `+` or `-` operators that can be used in front of any integer expression, including literals (see Section 9).

Examples:


```
125
'h48454a
32'h48454a
8'o255
12'b101010
```

2.3.1 Type conversion of integer literals

Integer literals can be used to specify values for various integer types and even for user-defined types. BSV uses its systematic overloading resolution mechanism to perform these type conversions. Overloading resolution is described in more detail in Section 14.1.

In an integer literal, if a specific width w is given (e.g., `8'o255`), the literal is assumed to have type `bit [`$w-1$`:0]`. The compiler implicitly applies the overloaded function `unpack` to the literal to convert it to the type required by the context. Thus, sized literals can be used for any type on which the overloaded function `unpack` is defined, i.e., for any type in the `Bits` type class.

If a specific width is not given, the literal is assumed to have type `Integer`. The compiler implicitly applies the overloaded function `fromInteger` to the literal to convert it to the type required by the context. Thus, unsized literals can be used for any type on which the overloaded function `fromInteger` is defined.

The literal `'0` just stands for 0. The literal `'1` stands for a value in which all bits are 1 (the width depends on the context).

2.4 Real literals

Support for `real` (Verilog 2001) and `shortreal` (SystemVerilog) will be added to BSV in the future.

2.5 String literals

String literals are written enclosed in double quotes `"`$\cdots$`"` and must be contained on a single source line.

stringLiteral ::= `"` $\cdots$ string characters $\cdots$ `"`

Special characters may be inserted in string literals with the following backslash escape sequences:

`\n`	newline
`\t`	tab
`\\`	backslash
`\"`	double quote
`\v`	vertical tab
`\f`	form feed
`\a`	bell
`\`*OOO*	exactly 3 octal digits (8-bit character code)
`\x`*HH*	exactly 2 hexadecimal digits (8-bit character code)

Example - printing characters using form feed.

```
module mkPrinter (Empty);
   String display_value;

   display_value = "a\nb\nc";    //prints a
                                 //       b
```



```
                                    //       c   repeatedly
   rule every;
      $display(display_value);
   endrule
endmodule
```

2.6 Don't-care values

A lone question mark `?` is treated as a special don't-care value. For example, one may return `?` from an arm of a case statement that is known to be unreachable.

Example - Using ? as a don't care value

```
module mkExample (Empty);
   Reg#(Bit#(8)) r <- mkReg(?);     // don't care is used for the
   rule every;                      // reset value of the Reg
      $display("value is %h", r);   // the value of r is displayed
   endrule
endmodule
```

2.7 Compiler directives

The following compiler directives permit file inclusion, macro definition and substitution, and conditional compilation. They follow the specifications given in the Verilog 2001 LRM plus the extensions given in the SystemVerilog 3.1a LRM.

In general, these compiler directives can appear anywhere in the source text. In particular, they do not need to be on lines by themselves, and they need not begin in the first column. Of course, they should not be inside strings or comments, where the text remains uninterpreted.

2.7.1 File inclusion: ‘include and ‘line

compilerDirective ::= `‘include "`*filename*`"`
| `‘include <`*filename*`>`
| `‘include` *macroInvocation*

In an `‘include` directive, the contents of the named file are inserted in place of this line. The included files may themselves contain compiler directives. Currently there is no difference between the `"..."` and `<...>` forms. A *macroInvocation* should expand to one of the other two forms. The file name may be absolute, or relative to the current directory.

compilerDirective ::= `‘line` *lineNumber* `"`*filename*`"` *level*
lineNumber ::= *decLiteral*
level ::= `0` | `1` | `2`

A `‘line` directive is terminated by a newline, i.e., it cannot have any other source text after the *level*. The compiler automatically keeps track of the source file name and line number for every line of source text (including from included source files), so that error messages can be properly correlated to the source. This directive effectively overrides the compiler's internal tracking mechanism, forcing it to regard the next line onwards as coming from the given source file and line number. It is generally not necessary to use this directive explicitly; it is mainly intended to be generated by other preprocessors that may themselves need to alter the source files before passing them through the BSV compiler; this mechanism allows proper references to the original source.

The *level* specifier is either 0, 1 or 2:

- 1 indicates that an include file has just been entered
- 2 indicates that an include file has just been exited
- 0 is used in all other cases

2.7.2 Macro definition and substitution: ‘define and related directives

compilerDirective ::= `‘define` *macroName* [`(` *macroFormals* `)`] *macroText*

macroName ::= *identifier*

macroFormals ::= *identifier* { `,` *identifier* }

The `‘define` directive is terminated by a bare newline. A backslash (\) just before a newline continues the directive into the next line. When the macro text is substituted, each such continuation backslash-newline is replaced by a newline.

The *macroName* is an identifier and may be followed by formal arguments, which are a list of comma-separated identifiers in parentheses. For both the macro name and the formals, lower and upper case are acceptable (but case is distinguished). The *macroName* cannot be any of the compiler directives (such as `include`, `define`, ...).

The scope of the formal arguments extends to the end of the *macroText*.

The *macroText* represents almost arbitrary text that is to be substituted in place of invocations of this macro. The *macroText* can be empty.

One-line comments (i.e., beginning with `//`) may appear in the *macroText*; these are not considered part of the substitutable text and are removed during substitution. A one-line comment that is not on the last line of a `‘define` directive is terminated by a backslash-newline instead of a newline.

A block comment (`/*...*/`) is removed during substitution and replaced by a single space.

The *macroText* can also contain the following special escape sequences:

- `‘"` Indicates that a double-quote (") should be placed in the expanded text.
- `‘\‘"` Indicates that a backslash and a double-quote (\") should be placed in the expanded text.
- `‘‘` Indicates that there should be no whitespace between the preceding and following text. This allows construction of identifiers from the macro arguments.

A minimal amount of lexical analysis of *macroText* is done to identify comments, string literals, identifiers representing macro formals, and macro invocations. As described earlier, one-line comments are removed. The text inside string literals is not interpreted except for the usual string escape sequences described in Section 2.5.

There are two define macros in the define environment initially; `‘bluespec` and `‘BLUESPEC`.

Once defined, a macro can be invoked anywhere in the source text (including within other macro definitions) using the following syntax.

compilerDirective ::= *macroInvocation*

macroInvocation ::= `‘`*macroName* [`(` *macroActuals* `)`]

macroActuals ::= *substText* { `,` *substText* }

The *macroName* must refer to a macro definition available at expansion time. The *macroActuals*, if present, consist of substitution text *substText* that is arbitrary text, possibly spread over multiple

lines, excluding commas. A minimal amount of parsing of this substitution text is done, so that commas that are not at the top level are not interpreted as the commas separating *macroActuals*. Examples of such "inner" uninterpreted commas are those within strings and within comments.

compilerDirective ::= `‘undef` *macroName*
| `‘resetall`

The `‘undef` directive's effect is that the specified macro (with or without formal arguments) is no longer defined for the subsequent source text. Of course, it can be defined again with `‘define` in the subsequent text. The `‘resetall` directive has the effect of undefining all currently defined macros, i.e., there are no macros defined in the subsequent source text.

2.7.3 Conditional compilation: ‘ifdef and related directives

compilerDirective ::= `‘ifdef` *macroName*
| `‘ifndef` *macroName*
| `‘elsif` *macroName*
| `‘else`
| `‘endif`

These directives are used together in either an `‘ifdef-endif` sequence or an `ifndef-endif` sequence. In either case, the sequence can contain zero or more `elsif` directives followed by zero or one `else` directives. These sequences can be nested, i.e., each `‘ifdef` or `ifndef` introduces a new, nested sequence until a corresponding `endif`.

In an `‘ifdef` sequence, if the *macroName* is currently defined, the subsequent text is processed until the next corresponding `elsif`, `else` or `endif`. All text from that next corresponding `elsif` or `else` is ignored until the `endif`.

If the *macroName* is currently not defined, the subsequent text is ignored until the next corresponding `‘elsif`, `‘else` or `‘endif`. If the next corresponding directive is an `‘elsif`, it is treated just as if it were an `‘ifdef` at that point.

If the `‘ifdef` and all its corresponding `‘elsif`s fail (macros were not defined), and there is an `‘else` present, then the text between the `‘else` and `‘endif` is processed.

An `‘ifndef` sequence is just like an `‘ifdef` sequence, except that the sense of the first test is inverted, i.e., its following text is processes if the *macroName* is *not* defined, and its `‘elsif` and `‘else` arms are considered only if the macro *is* defined.

Example using ‘ifdef to determine the size of a register:

```
‘ifdef USE_16_BITS
   Reg#(Bit#(16)) a_reg <- mkReg(0);
‘else
   Reg#(Bit#(8)) a_reg <- mkReg(0);
‘endif
```

3 Packages and the outermost structure of a BSV design

A BSV program consists of one or more outermost constructs called packages. All BSV code is assumed to be inside a package. Further, the BSV compiler and other tools assume that there is one package per file, and they use the package name to derive the file name. For example, a package called `Foo` is assumed to be located in a file `Foo.bsv`.

A BSV package is purely a linguistic namespace-management mechanism and is particularly useful for programming in the large, so that the author of a package can choose identifiers for the package

components freely without worrying about choices made by authors of other packages. Package structure is usually uncorrelated with hardware structure, which is specified by the module construct.

A package contains a collection of top-level statements that include specifications of what it imports from other packages, what it exports to other packages, and its definitions of types, interfaces, functions, variables, and modules. BSV tools ensure that when a package is compiled, all the packages that it imports have already been compiled.

```
package     ::= package packageIde ;
                { exportDecl }
                { importDecl }
                { packageStmt }
                endpackage [ : packageIde ]

exportDecl  ::= export exportItem { , exportItem } ;
exportItem  ::= identifier [ (..) ]
            |   Identifier [ (..) ]

importDecl  ::= import importItem { , importItem } ;
importItem  ::= packageIde :: *

packageStmt ::= moduleDef
            |   interfaceDecl
            |   typeDef
            |   varDecl | varAssign
            |   functionDef
            |   typeclassDef
            |   typeclassInstanceDef
            |   externModuleImport

packageIde  ::= Identifier
```

The name of the package is the identifier following the `package` keyword. This name can optionally be repeated after the `endpackage` keyword (and a colon). We recommend using an uppercase first letter in package names. In fact, the `package` and `endpackage` lines are optional: if they are absent, BSV derives the assumed package name from the filename.

Each export item specifies an identifier defined elsewhere within this package, optionally followed by `(..)`. The identifier then becomes accessible outside this package. If there are any export statements, then only those items are exported. If there are no export statements, by default all identifiers defined in this package are exported.

If the exported identifier is the name of a struct (structure) or union type definition, then the members of that type will be visible only if `(..)` is used. By omitting the `(..)` suffix, only the type, but not its members, are visible outside the package. This is a way to define abstract data types, i.e., types whose internal structure is hidden.

Each import item specifies a package from which to import identifiers, i.e., to make them visible locally within this package. For each imported package, all identifiers exported from that package are made locally visible.

Example:

```
package Foo;
export x;
export y;

import Bar::*;

... top level definition ...
```


... top level definition ...
... top level definition ...

```
endpackage: Foo
```

Here, `Foo` is the name of this package. The identifiers `x` and `y`, which must be defined by the top-level definitions in this package are names exported from this package. From package `Bar` we import all its definitions.

3.1 Scopes, name clashes and qualified identifiers

BSV uses standard static scoping (also known as lexical scoping). Many constructs introduce new scopes nested inside their surrounding scopes. Identifiers can be declared inside nested scopes. Any use of an identifier refers to its declaration in the nearest textually surrounding scope. Thus, an identifier `x` declared in a nested scope "shadows", or hides, any declaration of `x` in surrounding scopes (however, we recommend that the programmer avoids such shadowing, because it often makes code more difficult to read.)

Packages form the the outermost scopes. Examples of nested scopes include modules, interfaces, functions, methods, rules, action and actionvalue blocks, begin-end statements and expressions, bodies of for and while loops, and seq and par blocks.

When used in any scope, an identifier must have an unambiguous meaning. If there is name clash for an identifier x because it is defined in the current package and/or it is available from one or more imported packages, then the ambiguity can be resolved by using a qualified name of the form $P :: x$ to refer to the version of x contained in package P.

3.2 The Standard Prelude package

The Standard Prelude is a predefined package that is imported implicitly into every BSV package, i.e., it does not need an explicit `import` statement. It contains a number of useful predefined entities (types, values, functions, modules, etc.). The Standard Prelude package is described in more detail in appendix B. Reusing the name of Prelude entity when defining other entities, which would require the entity's name to be qualified with the package name, is strongly discouraged.

4 Types

Every variable and every expression in BSV has a *type*. Almost all variables must be declared with their type.

The syntax of types (type expressions) is given below:

type	::=	*typePrimary*	
	|	*typePrimary* `(` *type* { `,` *type* } `)`	Function type
typePrimary	::=	*typeIde* [`#` `(` *type* { `,` *type* } `)`]	
	|	*typeNat*	
	|	`bit [` *typeNat* `:` *typeNat* `]`	
typeIde	::=	*Identifier*	
typeNat	::=	*decDigits*	

Examples of simple types:


```
Integer                 // Unbounded signed integers, for static elaboration only
int                     // 32-bit signed integers
Bool
String
Action
```

Type expressions of the form X`#(`t_1`,`$\cdots$`,`t_N`)` are called *parameterized* types. X is called a *type constructor* and the types t_1,$\cdots$,t_N are the parameters of X. Examples:

```
Tuple2#(int,Bool)           // pair of items, an int and a Bool
Tuple3#(int,Bool,String)    // triple of items, an int, a Bool and a String
List#(Bool)                 // list containing booleans
List#(List#(Bool))          // list containing lists of booleans
RegFile#(Integer, String)   // a register file (array) indexed by integers, containing strings
```

Type parameters can be natural numbers (also known as *size types*). These usually indicate some aspect of the size of the type, such as a bit-width or a table capacity. Examples:

```
Bit#(16)             // 16-bit wide bit-vector
bit [15:0]           // synonym for Bit#(16)
UInt#(32)            // unsigned integers, 32 bits wide
Int#(29)             // signed integers, 29 bits wide
Vector#(16,Int#(29)  // Vector of size 16 containing Int#(29)'s
```

Currently the second index n in a `bit[`m`:`n`]` type must be 0. The type `bit[`m`:0]` represents the type of bit vectors, with bits indexed from m (msb/left) down through 0 (lsb/right), for $m \geq 0$.

4.1 Polymorphism

A type can be *polymorphic*. This is indicated by using type variables as parameters. Examples:

```
List#(a)                // lists containing items of some type a
List#(List#(b))         // lists containing lists of items of some type a
RegFile#(i, List#(x))   // arrays indexed by some type i, containing
                        //        lists that contain items of some type x
```

The type variables represent unknown (but specific) types. In other words, `List#(a)` represents the type of a list containing items all of which have some type `a`. It does not mean that different elements of a list can have different types.

4.2 Provisos (brief intro)

Provisos are described in detail in Section 14.1.1, and the general facility of type classes (overloading groups), of which provisos form a part, is described in Section 14.1. Here we provide a brief description, which is adequate for most uses and for continuity in a serial reading of this manual.

A proviso is a static condition attached to certain constructs, to impose certain restrictions on the types involved in the construct. The restrictions are of two kinds:

- Require instance of a type class (overloading group): this kind of proviso states that certain types must be instances of certain type classes, i.e., that certain overloaded functions are defined on this type.

- Require size relationships: this kind of proviso expresses certain constraints between the sizes of certain types.

The most common overloading provisos are:

```
Bits#(t,n)      // Type class (overloading group) Bits
                // Meaning: overloaded operators pack/unpack are defined
                //     on type t to convert to/from Bit#(n)

Eq#(t)          // Type class (overloading group) Eq
                // Meaning: overloaded operators == and != are defined on type t

Literal#(t)     // Type class (overloading group) Literal
                // Meaning: Overloaded function fromInteger() defined on type t
                //     to convert an integer literal to type t

Ord#(t)         // Type class (overloading group) Ord
                // Meaning: Overloaded order-comparison operators <, <=,
                //     > and >= are defined on type t

Bounded#(t)     // Type class (overloading group) Bounded
                // Meaning: Overloaded identifiers minBound and maxBound
                //     are defined for type t

Bitwise#(t)     // Type class (overloading group) Bitwise
                // Meaning: Overloaded operators &, |, ^, ~^, ^~, ~, << and >>
                //     and overloaded function invert are defined on type t

BitReduction#(t)// Type class (overloading group) BitReduction
                // Meaning: Overloaded prefix operators &, |, ^,
                //     ~&, ~|, ~^, and ^~ are defined on type t

BitExtend#(t)   // Type class (overloading group) BitExtend
                // Meaning: Overloaded functions zeroExtend, signExtend
                //     and truncate are defined on type t

Arith#(t)       // Type class (overloading group) Arith
                // Meaning: Overloaded operators +, -, and *, and overloaded
                //     prefix operator - (same as function negate), and
                //     overloaded function negate are defined on type t
```

The size relationship provisos are:

```
Add#(n1,n2,n3)    // Meaning: assert n1 + n2 = n3

Mul#(n1,n2,n3)    // Meaning: assert n1 * n2 = n3

Div#(n1,n2,n3)    // Meaning: assert n1 / n2 = n3

Max#(n1,n2,n3)    // Meaning: assert max(n1,n2) = n3

Log#(n1,n2)       // Meaning: assert ceiling(log(n1)) = n2
                  // The logarithm is base 2
```


Example:

```
module mkExample (ProvideCurrent#(a))
   provisos(Bits#(a, sa), Arith#(a));

   Reg#(a) value_reg <- mkReg(?); // requires that type "a" be in the Bits typeclass.
   rule every;
      value_reg <= value_reg + 1; // requires that type "a" be in the Arith typeclass.
   endrule
```

Example:

```
function Bit#(m) pad0101 (Bit#(n) x)
   provisos (Add#(n,4,m));   // m is 4 bits longer than n
   pad0101 = { x, 0b0101 };
endfunction: pad0101
```

This defines a function `pad0101` that takes a bit vector `x` and pads it to the right with the four bits "0101" using the standard bit-concatenation notation. The types and proviso express the idea that the function takes a bit vector of length n and returns a bit vector of length m, where $n + 4 = m$. These provisos permit the BSV compiler to statically verify that entities (values, variables, registers, memories, FIFOs, and so on) have the correct bit-width.

4.2.1 The pseudo-function valueof

To get the value that corresponds to a size type, there is a special pseudo-function, `valueof` that takes a size type and gives the corresponding `Integer` value.

exprPrimary ::= `valueof (` *type* `)`

In other words, it converts from a numeric type expression into an ordinary value. These mechanisms can be used to do arithmetic to derive dependent sizes. Example:

```
function ... foo (Vector#(n,int) xs) provisos (Log#(n,k));
   Int#(k) index;
   index = valueof(n) - 1;
   ...
endfunction
```

This function takes a vector of length `n` as an argument. The proviso fixes `k` to be the (ceiling of the) logarithm of `n`. The variable `index` has bit-width `k`, which will be adequate to hold an index into the list. The variable is initialized to the maximum index.

Note that the function `foo` may be invoked in multiple contexts, each with a different vector length. The compiler will statically verify that each use is correct (e.g., the index has the correct width).

The pseudo-function `valueof`, which converts a numeric type to a value, should not be confused with the pseudo-function `SizeOf`, described in Section 14.1.5, which converts a type to a numeric type.

4.3 A brief introduction to `deriving` clauses

The `deriving` clause is a part of the general facility of type classes (overloading groups), which is described in detail in Section 14.1. Here we provide a brief description, which is adequate for most uses and for continuity in a serial reading of this manual.

It is possible to attach a `deriving` clause to a type definition (Section 7), thereby directing the compiler to define automatically certain overloaded functions for that type. The most common forms of these clauses are:

```
deriving(Eq)       // Meaning: automatically define == and !=
                   // for equality and inequality comparisons

deriving(Bits)     // Meaning: automatically define pack and unpack
                   // for converting to/from bits

deriving(Bounded)  // Meaning: automatically define minBound and maxBound
```

Example:

```
typedef enum {LOW, NORMAL, URGENT} Severity deriving(Eq, Bits);
// == and != are defined for variables of type Severity
// pack and unpack are defined for variables of type Severity

module mkSeverityProcessor (SeverityProcessor);
   method Action process(Severity value);
      // value is a variable of type Severity
      if (value == URGENT) $display("WARNING: Urgent severity encountered.");
      // Since value is of the type Severity, == is defined
   endmethod
endmodule
```

5 Modules and interfaces, and their instances

Modules and interfaces form the heart of BSV. Modules and interfaces turn into actual hardware. An interface for a module m mediates between m and other, external modules that use the facilities of m. We often refer to these other modules as *clients* of m.

In SystemVerilog and BSV we separate the declaration of an interface from module definitions. There was no such separation in Verilog 1995 and Verilog 2001, where a module's interface was represented by its port list, which was part of the module definition itself. By separating the interface declaration, we can express the idea of a common interface that may be offered by several modules, without having to repeat that declaration in each of the implementation modules.

As in Verilog and SystemVerilog, it is important to distinguish between a module *definition* and a module *instantiation*. A module definition can be regarded as specifying a scheme that can be instantiated multiple times. For example, we may have a single module definition for a FIFO, and a particular design may instantiate it multiple times for all the FIFOs it contains.

Similarly, we also distinguish interface declarations and instances, i.e., a design will contain interface declarations, and each of these may have multiple instances. For example an interface declaration I may have one instance i_1 for communication between module instances a_1 and b_1, and another instance i_2 for communication between module instances a_2 and b_2.

Module instances form a pure hierarchy. Inside a module definition mkM, one can specify instantiations of other modules. When mkM is used to instantiate a module m, it creates the specified inner module instances. Thus, every module instance other than the top of the hierarchy unambiguously has a single parent module instance. We refer to the top of the hierarchy as the root module. Every module instance has a unique set, possibly empty, of child module instances. If there are no children, we refer to it as a leaf module.

A module consists of three things: state, rules that operate on that state, and the module's interface to the outside world (surrounding hierarchy). The state conceptually consists of all state in the sub-hierarchy headed by this module; ultimately, it consists of all the lower leaf module instances (see next section on state and module instantiation). Rules are the fundamental means to express behavior in BSV (instead of the `always` blocks used in traditional Verilog). In BSV, an interface consists of *methods* that encapsulate the possible transactions that clients can perform, i.e., the micro-protocols with which clients interact with the module. When compiled into RTL, an interface becomes a collection of wires.

5.1 Explicit state via module instantiation, not variables

In Verilog and SystemVerilog RTL, one simply declares variables, and a synthesis tool "infers" how these variables actually map into state elements in hardware using, for example, their lifetimes relative to events. A variable may map into a bus, a latch, a flip-flop, or even nothing at all. This ambiguity is acknowledged in the Verilog 2001 and SystemVerilog LRMs.[1]

BSV removes this ambiguity and places control over state instantiation explicitly in the hands of the designer. From the smallest state elements (such as registers) to the largest (such as memories), all state instances are specified explicitly using module instantiation.

Conversely, an ordinary declared variable in BSV *never* implies state, i.e., it never holds a value over time. Ordinary declared variables are always just convenient names for intermediate values in a computation. Ordinary declared variables include variables declared in blocks, formal parameters, pattern variables, loop iterators, and so on. Another way to think about this is that ordinary variables play a role only in static elaboration, not in the dynamic semantics. This is one of the aspects of BSV style that may initially appear unusual to the Verilog or SystemVerilog programmer.

Example:

```
module mkExample (Empty);
   // Hardware registers are created here
   Reg#(Bit#(8)) value_reg <- mkReg(0);

   FIFO#(Bit#(8)) fifo <- mkFIFO;

   rule pop;
      let value = fifo.first();  // value is a ordinary declared variable
                                 // no state is implied or created
      value_reg <= fifo.first(); // value_reg is state variable
      fifo.deq();
    endrule
endmodule
```

5.2 Interface declaration

In BSV an interface contains members that are called *methods* (an interface may also contain subinterfaces, which are described in Section 5.2.1). To first order, a method can be regarded exactly like a function, i.e., it is a procedure that takes zero or more arguments and returns a result. Thus, method declarations inside interface declarations look just like function prototypes, the only difference being the use of the keyword `method` instead of the keyword `function`. Each method represents

[1] In the Verilog 2001 LRM, Section 3.2.2, Variable declarations, says: "A *variable* is an abstraction of a data storage element.···NOTE In previous versions of the Verilog standard, the term *register* was used to encompass both the **reg**, **integer**, **time**, **real** and **realtime** types; but that term is no longer used as a Verilog data type."

In the SystemVerilog LRM, Section 5.1 says: "Since the keyword `reg` no longer describes the user's intent in many cases,···Verilog-2001 has already deprecated the use of the term *register* in favor of *variable*."

one kind of transaction between a module and its clients. When translated into RTL, each method becomes a bundle of wires.

The fundamental difference between a method and a function is that a method also carries with it a so-called implicit condition. These will be described later along with method definitions and rules.

An interface declaration also looks similar to a struct declaration. One can think of an interface declaration as declaring a new type similar to a struct type (Section 7), where the members all happen to be method prototypes. A method prototype is essentially the header of a method definition (Section 5.5).

interfaceDecl ::= [*interfaceAttribute*]
`interface` *typeDefType* `;`
{ *interfaceMemberDecl* }
`endinterface` [`:` *typeIde*]

typeDefType ::= *typeIde* [*typeFormals*]

typeFormals ::= `#(` *typeFormal* { `,` *typeFormal* }`)`

typeFormal ::= [`numeric`] `type` *typeIde*

interfaceMemberDecl ::= *methodProto* | *subinterfaceDecl*

methodProto ::= [*interfaceAttribute*]
`method` *type identifier* `(` [*methodProtoFormals*] `)` `;`

methodProtoFormals ::= *methodProtoFormal* { `,` *methodProtoFormal* }

methodProtoFormal ::= [*interfaceAttribute*] *type identifier*

Example: a stack of integers:

```
interface IntStack;
    method  Action  push  (int x);
    method  Action  pop;
    method  int     top;
endinterface: IntStack
```

This describes an interface to a circuit that implements a stack (LIFO) of integers. The `push` method takes an `int` argument, the item to be pushed onto the stack. Its output type is `Action`, namely it returns an *enable* wire which, when asserted, will carry out the pushing action.[2] The `pop` method takes no arguments, and simply returns an enable wire which, when asserted, will discard the element from the top of the stack. The `top` method takes no arguments, and returns a value of type `int`, i.e., the element at the top of the stack.

What if the stack is empty? In that state, it should be illegal to use the `pop` and `top` methods. This is exactly where the difference between methods and functions arises. Each method has an implicit *ready* wire, which governs when it is legal to use it, and these wires for the `pop` and `top` methods will presumably be de-asserted if the stack is empty. Exactly how this is accomplished is an internal detail of the module, and is therefore not visible as part of the interface declaration. (We can similarly discuss the case where the stack has a fixed, finite depth; in this situation, it should be illegal to use the `push` method when the stack is full.)

One of the major advantages of BSV is that the compiler automatically generates all the control circuitry needed to ensure that a method (transaction) is only used when it is legal to use it.

Interface types can be polymorphic, i.e., parameterized by other types. For example, the following declaration describes an interface for a stack containing an arbitrary but fixed type:

[2] The type `Action` is discussed in more detail in Section 9.6.


```
interface Stack#(type a);
    method  Action  push  (a x);
    method  Action  pop;
    method  a       top;
endinterface: Stack
```

We have replaced the previous specific type `int` with a type variable `a`. By "arbitrary but fixed" we mean that a particular stack will specify a particular type for `a`, and all items in that stack will have that type. It does not mean that a particular stack can contain items of different types.

For example, using this more general definition, we can also define the `IntStack` type as follows:

```
typedef  Stack#(int)  IntStack;
```

i.e., we simply specialize the more general type with the particular type `int`. All items in a stack of this type will have the `int` type.

Usually there is information within the interface declaration which indicates whether a polymorphic interface type is numeric or nonnumeric. The optional `numeric` is required before the type when the interface type is polymorphic and must be numeric but there is no information in the interface declaration which would indicate that the type is numeric.

For example, in the following polymorphic interface, `count_size` must be numeric because it is defined as a parameter to `Bit#()`.

```
interface Counter#(type count_size);
   method Action increment();
   method Bit#(count_size) read();
endinterface
```

From this use, it can be deduced that `Counter's` parameter `count_size` must be numeric. However, sometimes you might want to encode a size in an interface type which isn't visible in the methods, but is used by the module implementing the interface. For instance:

```
interface SizedBuffer#(numeric type buffer_size, type element_type);
   method Action enq(element_type e);
   method ActionValue#(element_type) deq();
endinterface
```

In this interface, the depth of the buffer is encoded in the type. For instance, `SizedBuffer#(8, Bool)` would be a buffer of depth 8 with elements of type `Bool`. The depth is not visible in the interface, but is used by the module to know how much storage to instantiate.

Because the parameter is not mentioned anywhere else in the interface, there is no information to determine whether the parameter is a numeric type or a non-numeric type. In this situation, the default is to assume that the paramater is non-numeric. The user can override this default by specifying `numeric` in the interface declaration.

5.2.1 Subinterfaces

Note: this is an advanced topic that may be skipped on first reading.

Interfaces can also be declared hierarchically, using subinterfaces.

subinterfaceDecl ::= [*interfaceAttribute*]
`interface` *type identifier*

where *type* is another interface type available in the current scope. Example:


```
interface ILookup;
    interface  Server#( RequestType, ResponseType )  mif;
    interface  RAMclient#( AddrType, DataType )      ram;
    method     Bool  initialized;
endinterface: ILookup
```

This declares an interface `ILookup` module that consists of three members: a `Server` subinterface called `mif`, a `RAMClient` subinterface called `ram`, and a boolean method called `initialized` (the `Server` and `RAMClient` interface types are defined in the libraries, see Appendix C). Methods of subinterfaces are accessed using dot notation to select the desired component, e.g.,

```
ilookup.mif.request.put(...);
```

5.3 Module definition

A module definition begins with a module header containing the `module` keyword, the module name, parameters, arguments, interface type and provisos. The header is followed by zero or more module statements. Finally we have the closing `endmodule` keyword, optionally labelled again with the module name.

moduleDef ::= [*modgenAttribute*] [*docAttribute*]
`module` [`[` *type* `]`] *identifier*
[*moduleFormalParams*] `(` [*moduleFormalArgs*] `)` [*provisos*];
{ *moduleStmt* }
`endmodule` [`:` *identifier*]

moduleFormalParams ::= `#` (*moduleFormalParam* { `,` *moduleFormalParam* })

moduleFormalParam ::= [`parameter`] *type identifier*

moduleFormalArgs ::= *type*
| *type identifier* { `,` *type identifier* }

As a stylistic convention, many BSV examples use module names like `mkFoo`, i.e., beginning with the letters `mk`, suggesting the word *make*. This serves as a reminder that a module definition is not a module instance. When the module is instantiated, one invokes `mkFoo` to actually create a module instance.

The optional *moduleFormalParams* are exactly as in Verilog and SystemVerilog, i.e., they represent module parameters that must be supplied at each instantiation of this module, and are resolved at elaboration time.

The optional *moduleFormalArgs* represent the interfaces *used by* the module, such as clocks or wires. The final argument is a single interface *provided by* the module instead of Verilog's port list. The interpretation is that this module will define and offer an interface of that type to its clients. If the only argument is the interface, only the interface type is required. If there are other arguments, both a *type* and an *identifier* must be specified for consistency, but the final interface name will not be used in the body. Omitting the interface type completely is equivalent to using the pre-defined `Empty` interface type, which is a trivial interface containing no methods.

The arguments and parameters may be enclosed in a single set of parentheses, in which case the `#` would be omitted.

Provisos, which are optional, come next. These are part of an advanced feature called type classes (overloading groups), and are discussed in more detail in Section 14.1.

Examples

A module with parameters and an interface.


```
module mkFifo#(Int#(8) a) (Fifo);
...
endmodule
```

A module with arguments and an interface, but no parameters

```
module mkSyncPulse (Clock sClkIn, Reset sRstIn,
                    Clock dClkIn,
                    SyncPulseIfc ifc);
...
endmodule
```

A module definition with parameters, arguments, and provisos

```
module mkSyncReg#(a_type initValue)
                 (Clock sClkIn, Reset sRstIn,
                  Clock dClkIn,
                  Reg#(a_type) ifc)
       provisos (Bits#(a_type_sa));
...
endmodule
```

The above module definition may also be written with the arguments and parameters combined in a single set of parentheses.

```
module mkSyncReg (a_type initValue,
                  Clock sClkIn, Reset sRstIn,
                  Clock dClkIn,
                  Reg#(a_type) ifc)
       provisos (Bits#(a_type_sa));
...
endmodule
```

The body of the module consists of a sequence of *moduleStmt*s:

moduleStmt	::=	*moduleInst*
	|	*methodDef*
	|	*subinterfaceDef*
	|	*rule*
	|	<*module*>*If* | <*module*>*Case*
	|	<*module*>*BeginEndStmt*
	|	*varDecl* | *varAssign*
	|	*varDo* | *varDeclDo*
	|	*functionDef*
	|	*functionCall*
	|	*systemTaskCall*
	|	(*expression*)
	|	*returnStmt*

Most of these are discussed elsewhere since they can also occur in other contexts (e.g., in packages, function bodies, and method bodies). Below, we focus solely on those statements that are found only in module bodies or are treated specially in module bodies.

5.4 Module and interface instantiation

Module instances form a hierarchy. A module definition can contain specifications for instantiating other modules, and in the process, instantiating their interfaces. A single module definition be instantiated multiple times within a module.

5.4.1 Short form instantiation

There is a one-line shorthand for instantiating a module and its interfaces.

moduleInst ::= *type identifier* `<-` *moduleApp* `;`

moduleApp ::= *identifier* `(` [*moduleActualParam* { `,` *moduleActualParam* }] [*moduleActualArg* { `,` *moduleActualArg* }] `)`

moduleActualParam ::= *expression*

moduleActualArg ::= *expression*
| `clocked_by` *expression*
| `reset_by` *expression*

The statement first declares an identifier with an interface type. After the `<-` symbol, we have a module application, consisting of a module *identifier* optionally followed by a parameter list and an argument list, if the module had been defined to have parameters and arguments. Note that the parameters and the arguments are within a single set of parentheses and there is no `#` before the actual parameter list.

Each module has an implicit clock and reset. These defaults can be changed by explicitly specifying a `clocked_by` or `reset_by` argument in the module instantiation.

The following skeleton illustrates the structure and relationships between interface and module definition and instantiation.

```
interface ArithIO#(type a);             //interface type called ArithIO
    method  Action  input (a x, a y);   //parameterized by type a
    method  a       output;             //contains 2 methods, input and output
endinterface: ArithIO

module mkGCD#(int N) (ArithIO#(bit [31:0]));
    ...                                 //module definition for mkGCD
    ...                                 //one parameter, an integer N
endmodule: mkGCD                        //presents interface of type ArithIO#(bit{31:0])

//declare the interface instance gcdIFC, instantiate the module mkGCD, set N=5
module mkTest ();
    ...
    ArithIO#(bit [31:0]) gcdIfc <- mkGCD (5, clocked_by dClkIn);
    ...
 endmodule: mkTest
```

The following example shows an module instantiation using a clocked_by statement.

```
interface Design_IFC;
    method Action start(Bit#(3) in_data1, Bit#(3) in_data2, Bool select);
    interface Clock clk_out;
    method Bit#(4) out_data();
endinterface : Design_IFC
```



```
module mkDesign(Clock prim_clk, Clock sec_clk, Design_IFC ifc);
    ...
    RWire#(Bool) select <- mkRWire (select, clocked_by sec_clk);
    ...
endmodule:mkDesign
```

5.4.2 Long form instantiation

A module instantiation can also be written in its full form on two consecutive lines, as typical in SystemVerilog. The full form specifies names for both the interface instance and the module instance. In the shorthand described above, there is no name provided for the module instance and the compiler infers one based on the interface name. This is often acceptable because module instance names are used occasionally in debugging and in hierarchical names.

moduleInst	::=	*type identifier* `( ) ;` *moduleApp2 identifier* `(` [*moduleActualArgs*] `) ;`
moduleApp2	::=	*identifier* [`#` `(` *moduleActualParam* { `,` *moduleActualParam* } `)`]
moduleActualParam	::=	*expression*
moduleActualArgs	::=	*moduleActualArg* { `,` *moduleActualArg* }
moduleActualArg	::=	*expression* | `clocked_by` *expression* | `reset_by` *expression*

The first line declares an identifier with an interface type. The second line actually instantiates the module and defines the interface. The *moduleApp2* is the module (definition) identifier, and it must be applied to actual parameters (in `#(..)`) if it had been defined to have parameters. After the *moduleApp*, the first *identifier* names the new module instance. This may be followed by one or more *moduleActualArg* which define the arguments being used by the module. The last *identifier* (in parentheses) of the *moduleActualArg* must be the same as the interface identifier declared immediately above. It may be followed by a `clocked_by` or `reset_by` statement.

The following examples show the complete form of the module instantiations of the examples shown above.

```
module mkTest ();                       //declares a module mkTest
   ...                                  //
   ArithIO#(bit [31:0]) gcdIfc();       //declares the interface instance
   mkGCD#(5) a_GCD (gcdIfc);            //instantiates module mkGCD
   ...                                  //sets N=5, names module instance a_GCD
endmodule: mkTest                       //and interface instance gcdIfc
```

```
module mkDesign(Clock prim_clk, Clock sec_clk, Design_IFC ifc);
   ...
   RWire#(Bool)     select();
   mkRWire          t_select(select, clocked_by sec_clk);
   ...
endmodule:mkDesign
```


5.5 Interface definition (definition of methods)

A module definition contains a definition of its interface. Typically this takes the form of a collection of definitions, one for each method in its interface. Each method definition begins with the keyword `method`, followed optionally by the return-type of the method, then the method name, its formal parameters, and an optional implicit condition. After this comes the method body which is exactly like a function body. It ends with the keyword `endmethod`, optionally labelled again with the method name.

moduleStmt ::= *methodDef*

methodDef ::= `method` [*type*] *identifier* `(` *methodFormals* `)` [*implicitCond*] `;`
functionBody
`endmethod` [`:` *identifier*]

methodFormals ::= *methodFormal* { `,` *methodFormal* }

methodFormal ::= [*type*] *identifier*

implicitCond ::= `if` `(` *condPredicate* `)`
condPredicate ::= *exprOrCondPattern* { `&&&` *exprOrCondPattern* }
exprOrCondPattern ::= *expression*
| *expression* `matches` *pattern*

The method name must be one of the methods in the interface whose type is specified in the module header. Each of the module's interface methods must be defined exactly once in the module body.

The compiler will issue a warning if a method is not defined within the body of the module.

The return type of the method and the types of its formal arguments are optional, and are present for readability and documentation purposes only. The compiler knows these types from the method prototypes in the interface declaration. If specified here, they must exactly match the corresponding types in the method prototype.

The implicit condition, if present, may be a boolean expression, or it may be a pattern-match (pattern matching is described in Section 10). Expressions in the implicit condition can use any of the variables in scope surrounding the method definition, i.e., visible in the module body, but they cannot use the formal parameters of the method itself. If the implicit condition is a pattern-match, any variables bound in the pattern are available in the method body. Omitting the implicit condition is equivalent to saying `if (True)`. The semantics of implicit conditions are discussed in Section 9.13, on rules.

Every method is ultimately invoked from a rule (a method m_1 may be invoked from another method m_2 which, in turn, may be invoked from another method m_3, and so on, but if you follow the chain, it will end in a method invocation inside a rule). A method's implicit condition controls whether the invoking rule is enabled. Using implicit conditions, it is possible to write client code that is not cluttered with conditionals that test whether the method is applicable. For example, a client of a FIFO module can just call the `enqueue` or the `dequeue` method without having explicitly to test whether the FIFO is full or empty, respectively; those predicates are usually specified as implicit conditions attached to the FIFO methods.

Please note carefully that the implicit condition precedes the semicolon that terminates the method definition header. There is a very big semantic difference between the following:

```
method ... foo (...) if (expr);
    ...
endmethod
```

and


```
method ... foo (...); if (expr)
    ...
endmethod
```

The only syntactic difference is the position of the semicolon. In the first case, `if (expr)` is an implicit condition on the method. In the second case the method has no implicit condition, and `if (expr)` starts a conditional statement inside the method. In the first case, if the expression is false, any rule that invokes this method cannot fire, i.e., no action in the rule or the rest of this method is performed. In the second case, the method does not prevent an invoking rule from firing, and if the rule does fire, the conditional statement is not executed but other actions in the rule and the method may be performed.

The method body is exactly like a function body, which is discussed in Section 8.8 on function definitions.

See also Section 9.12 for the more general concepts of interface expressions and expressions as first-class objects.

Example:

```
interface GrabAndGive;                  // interface is declared
    method Action grab(Bit#(8) value); // method grab is declared
    method Bit#(8) give();             // method give is declared
 endinterface

 module mkExample (GrabAndGive);
    Reg#(Bit#(8)) value_reg <- mkReg(?);
    Reg#(Bool) not_yet <- mkReg(True);

    // method grap is defined
    method Action grab(Bit#(8) value) if (not_yet);
       value_reg <= value;
       not_yet <= False;
    endmethod

    //method give is defined
    method Bit#(8) give() if (!not_yet);
       return value_reg;
    endmethod
 endmodule
```

5.5.1 Shorthands for Action and ActionValue method definitions

If a method has type `Action`, then the following shorthand syntax may be used. Section 9.6 describes action blocks in more detail.

methodDef ::= `method Action` *identifier* `(` *methodFormals* `)` [*implicitCond*] `;`
{ *actionStmt* }
`endmethod` [`:` *identifier*]

i.e., if the type `Action` is used after the `method` keyword, then the method body can directly contain a sequence of *actionStmt*s without the enclosing `action` and `endaction` keywords.

Similarly, if a method has type `ActionValue(`*t*`)` (Section 9.7), the following shorthand syntax may be used:

methodDef ::= `method ActionValue #(` *type* `)` *identifier* `(` *methodFormals* `)`
[*implicitCond* `;`]

{ *actionValueStmt* }
`endmethod` [`:` *identifier*]

i.e., if the type `ActionValue(`t`)` is used after the `method` keyword, then the method body can directly contain a sequence of *actionStmt*s without the enclosing `actionvalue` and `endactionvalue` keywords.

Example: The long form definition of an `Action` method:

```
method grab(Bit#(8) value);
   action
       last_value <= value;
   endaction
endmethod
```

can be replaced by the following shorthand definition:

```
method Action grab(Bit#(8) value);
   last_value <= value;
endmethod
```

5.5.2 Definition of subinterfaces

Note: this is an advanced topic and can be skipped on first reading.

Declaration of subinterfaces (hierarchical interfaces) was described in Section 5.2.1. A subinterface member of an interface can be defined using the following syntax.

moduleStmt ::= *subinterfaceDef*

subinterfaceDef ::= `interface` *Identifier identifier* `;`
{ *subinterfaceDefStmt* }
`endinterface` [`:` *identifier*]

subinterfaceDefStmt ::= *methodDef* | *subinterfaceDef*

The subinterface member is defined within `interface-endinterface` brackets. The first *Identifier* must be the name of the subinterface member's type (an interface type), without any parameters. The second *identifier* (and the optional *identifier* following the `endinterface` must be the subinterface member name. The *subinterfaceDefStmt*s then define the methods or further nested subinterfaces of this member. Example (please refer to the `ILookup` interface defined in Section 5.2.1):

```
module ...
   ...
   ...
   interface Server mif;

       interface Put request;
           method put(...);
               ...
           endmethod: put
       endinterface: request

       interface Get response;
           method get();
               ...
           endmethod: get
```



```
        endinterface: response

    endinterface: mif
    ...
endmodule
```

5.5.3 Definition of methods and subinterfaces by assignment

Note: this is an advanced topic and can be skipped on first reading.

A method can also be defined using the following syntax.

methodDef ::= `method` [*type*] *identifier* `(` *methodFormals* `)` [*implicitCond*]
`=` *expression* `;`

The part up to and including the *implicitCond* is the same as the standard syntax shown in Section 5.5. Then, instead of a semicolon, we have an assignment to an expression that represents the method body. The expression can of course use the method's formal arguments, and it must have the same type as the return type of the method. See Sections 9.6 and 9.7 for how to construct expressions of `Action` type and `ActionValue` type, respectively.

A subinterface member can also be defined using the following syntax.

subinterfaceDef ::= `interface` *identifier* `=` *expression* `;`

The *identifier* is just the subinterface member name. The *expression* is an interface expression (described in Section 9.12) of the appropriate interface type.

For example, in the following module the subinterface `Put` is defined by assignment.

```
//in this module, there is an instanciated FIFO, and the Put interface
//of the "mkSameInterface" module is the same interface as the fifo's:

interface IFC1 ;
   interface Put#(int) in0 ;
endinterface

(*synthesize*)
module mkSameInterface (IFC1);
   FIFO#(int) myFifo <- mkFIFO;
   interface Put in0 = fifoToPut(myFifo);
endmodule
```

5.6 Rules in module definitions

The internal behavior of a module is described using zero or more rules.

moduleStmt ::= *rule*

rule ::= [*ruleAttribute*] [*docAttribute*]
`rule` *identifier* [*ruleCond*] `;`
ruleBody
`endrule` [`:` *identifier*]

ruleCond ::= `(` *condPredicate* `)`

condPredicate ::= *exprOrCondPattern* { `&&&` *exprOrCondPattern* }

exprOrCondPattern ::= *expression*
| *expression* `matches` *pattern*

ruleBody ::= { *actionStmt* }

A rule is optionally preceded by a *ruleAttribute*; these are described in Section 13.3. Every rule must have a name (the *identifier*). If the closing `endrule` is labelled with an identifier, it must be the same name. Rule names need not be unique, since they do not have any semantic significance and are only used for debugging; however, it is good style (and helps in debugging) to use unique names.

The *ruleCond*, if present, may be a boolean expression, or it may be a pattern-match (pattern matching is described in Section 10). It can use any identifiers from the scope surrounding the rule, i.e., visible in the module body. If it is a pattern-match, any variables bound in the pattern are available in the rule body.

The *ruleBody* must be of type `Action`, using a sequence of zero or more *actionStmt*s. We discuss *actionStmt*s in Section 9.6, but here we make a key observation. Actions include updates to state elements (including register writes). There are *no restrictions* on different rules updating the same state elements. The BSV compiler will generate all the control logic necessary for such shared update, including multiplexing, arbitration, and resource control. The generated control logic will ensure rule atomicity, discussed briefly in the next paragraphs.

A more detailed discussion of rule semantics is given in Section 6.2, Dynamic Semantics, but we outline the key point briefly here. The *ruleCond* is called the *explicit condition* of the rule. Within the *ruleCond* and *ruleBody*, there may be calls to various methods of various interfaces. Each such method call has an associated implicit condition. The rule is *enabled* when its explicit condition and all its implicit conditions are true. A rule can *fire*, i.e., execute the actions in its *ruleBody*, when the rule is enabled and when the actions cannot "interfere" with the actions in the bodies of other rules. Non-interference is described more precisely in Section 6.2 but, roughly speaking, it means that the rule execution can be viewed as an *atomic* state transition, i.e., there cannot be any race conditions between this rule and other rules.

This atomicity and the automatic generation of control logic to guarantee atomicity is a key benefit of BSV. Note that because of method calls in the rule and, transitively, method calls in those methods, a rule can touch (read/write) state that is distributed in several modules. Thus, a rule can express a major state change in the design. The fact that it has atomic semantics guarantees the absence of a whole class of race conditions that might otherwise bedevil the designer. Further, changes in the design, whether in this module or in other modules, cannot introduce races, because the compiler will verify atomicity.

See also Section 9.13 for a discussion of the more general concepts of rule expressions and rules as first-class objects.

5.7 Examples

A register is primitive module with the following predefined interface:

```
interface Reg#(type a);
    method  Action  _write (a x1);
    method  a       _read  ();
endinterface: Reg
```

It is polymorphic, i.e., it can contain values of any type `a`. It has two methods. The `_write()` method takes an argument `x1` of type `a` and returns an `Action`, i.e., an enable-wire that, when asserted, will deposit the value into the register. The `_read()` method takes no arguments and returns the value that is in the register.

The principal predefined module definition for a register has the following header:

```
// takes an initial value for the register
module mkReg#(a v) (Reg#(a)) provisos (Bits#(a, sa));
```


The module parameter `v` of type `a` is specified when instantiating the module (creating the register), and represents the initial value of the register. The module defines an interface of type `Reg #(a)`. The proviso specifies that the type `a` must be convertible into an `sa`-bit value. Provisos are discussed in more detail in Sections 4.2 and 14.1.

Here is a module to compute the GCD (greatest common divisor) of two numbers using Euclid's algorithm.

```
interface ArithIO#(type a);
    method  Action  start (a x, a y);
    method  a       result;
endinterface: ArithIO

module mkGCD(ArithIO#(Bit#(size_t)));

    Reg#(Bit#(size_t)) x(); // x is the interface to the register
    mkRegU reg_1(x);        // reg_1 is the register instance

    Reg #(Bit#(size_t)) y(); // y is the interface to the register
    mkRegU reg_2(y);         // reg_2 is the register instance

    rule flip (x > y && y != 0);
        x <= y;
        y <= x;
    endrule

    rule sub (x <= y && y != 0);
        y <= y - x;
    endrule

    method Action start(Bit#(size_t) num1, Bit#(size_t) num2) if (y == 0);
        action
            x <= num1;
            y <= num2;
        endaction
    endmethod: start

    method Bit#(size_t) result() if (y == 0);
        result = x;
    endmethod: result

endmodule: mkGCD
```

The interface type is called `ArithIO` because it expresses the interactions of modules that do any kind of two-input, one-output arithmetic. Computing the GCD is just one example of such arithmetic. We could define other modules with the same interface that do other kinds of arithmetic.

The module contains two rules, `flip` and `sub`, which implement Euclid's algorithm. In other words, assuming the registers `x` and `y` have been initialized with the input values, the rules repeatedly update the registers with transformed values, terminating when the register `y` contains zero. At that point, the rules stop firing, and the GCD result is in register `x`. Rule `flip` uses standard Verilog non-blocking assignments to express an exchange of values between the two registers. As in Verilog, the symbol `<=` is used both for non-blocking assignment as well as for the less-than-or-equal operator (e.g., in rule `sub`'s explicit condition), and as usual these are disambiguated by context.

The `start` method takes two arguments `num1` and `num2` representing the numbers whose GCD is sought, and loads them into the registers `x` and `y`, respectively. The `result` method returns the

result value from the `x` register. Both methods have an implicit condition (`y == 0`) that prevents them from being used while the module is busy computing a GCD result.

A test bench for this module might look like this:

```
module mkTest ();
   ArithIO#(Bit#(32)) gcd;     // declare ArithIO interface gcd
   mkGCD the_gcd (gcd);   // instantiate gcd module the_gcd

   rule getInputs;
       ... read next num1 and num2 from file ...
       the_gcd.start (num1, num2);    // start the GCD computation
   endrule

   rule putOutput;
       $display("Output is %d", the_gcd.result());    // print result
   endrule
endmodule: mkTest
```

The first two lines instantiate a GCD module. The `getInputs` rule gets the next two inputs from a file, and then initiates the GCD computation by calling the `start` method. The `putOutput` rule prints the result. Note that because of the semantics of implicit conditions and enabling of rules, the `getInputs` rule will not fire until the GCD module is ready to accept input. Similarly, the `putOutput` rule will not fire until the `output` method is ready to deliver a result.[3]

The `mkGCD` module is trivial in that the rule conditions (`(x > y)` and `(x <= y)`) are mutually exclusive, so they can never fire together. Nevertheless, since they both write to register `y`, the compiler will insert the appropriate multiplexers and multiplexer control logic.

Similarly, the rule `getInputs`, which calls the `start` method, can never fire together with the `mkGCD` rules because the implicit condition of `getInputs`, i.e., `(y == 0)` is mutually exclusive with the explicit condition `(y != 0)` in `flip` and `sub`. Nevertheless, since `getInputs` writes into `the_gcd`'s registers via the `start` method, the compiler will insert the appropriate multiplexers and multiplexer control logic.

In general, many rules may be enabled simultaneously, and subsets of rules that are simultaneously enabled may both read and write common state. The BSV compiler will insert appropriate scheduling, datapath multiplexing, and control to ensure that when rules fire in parallel, the net state change is consistent with the atomic semantics of rules.

5.8 Synthesizing Modules

In order to generate code for a BSV design (for either Verilog or Bluesim), it is necessary to indicate to the complier which module(s) are to be synthesized. A BSV module that is marked for code generation is said to be a *synthesized* module. A module can be marked for synthesis in one of two ways.

1. A module can be annotated with the `synthesize` attribute (see section 13.1.1). The appropriate syntax is show below.

[3] The astute reader will recognize that in this small example, since the `result` method is initially ready, the test bench will first output a result of `0` before initiating the first computation. Let us overlook this by imagining that Euclid is clearing his throat before launching into his discourse.


```
(* synthesize *)
module mkFoo (FooIfc);
...
endmodule
```

2. Alternatively, the `-g` compiler flag can be used on the bsc command line to indicate which module is to be synthesized. In order to have the same effect as the attribute syntax shown above, the flag would be used with the format `-g mkFoo` (the appropriate module name follows the `-g` flag).

Note that multiple modules may be selected for code generation (by using multiple `synthesize` attributes, multiple `-g` compiler flags, or a combination of the two).

5.8.1 Type Polymorphism

As discussed in section 4.1, BSV supports polymorphic types, including interfaces (which are themselves types). Thus, a single BSV module definition, which provides a polymorphic interface, in effect defines a family of different modules with different characteristics based on the specific parameter(s) of the polymorphic interface. Consider the module definition presented in section 5.7.

```
module mkGCD (ArithIO#(Bit#(size_t)));
...
endmodule
```

Based on the specific type parameter given to the `ArithIO` interface, the code required to implement `mkGCD` will differ. Since the Bluespec compiler does not create "parameterized" Verilog, in order for a module to be synthesizable, the associated interface must be fully specified (i.e not polymorphic). If the `mkGCD` module is annotated for code generation *as is*

```
(* synthesize *)
module mkGCD (ArithIO#(Bit#(size_t)));
...
endmodule
```

and we then run the compiler, we get the following error message.

```
Error: "GCD.bsv", line 7, column 8: (T0043)
  Bad top level type: polymorphic type:
  Prelude::Module#(GCD::ArithIO#(Prelude::Bit#(size_t)))
```

The compiler is telling us that the top level type of a synthesized module cannot be polymorphic. If however we instead re-write the definition of `mkGCD` such that all the references to the type parameter `size_t` are replaced by a specific value, in other words if we write something like,

```
(* synthesize *)
module mkGCD32 (ArithIO#(Bit#(32)));

   Reg#(Bit#(32)) x(); // x is the interface to the register
   mkRegU reg_1(x);    // reg_1 is the register instance

   ...

endmodule
```


then the compiler will complete successfully and provide code for a 32-bit version of the module (called mkGCD32). Equivalently, we can leave the code for mkGCD unchanged and instantiate it inside another synthesized module which fully specifies the provided interface.

```
(* synthesize *)
module mkGCD32(ArithIO#(Bit#(32)));
   let ifc();
   mkGCD _temp(ifc);
   return (ifc);
endmodule
```

6 Static and dynamic semantics

What is a legal BSV source text, and what are its legal behaviors? These questions are addressed by the static and dynamic semantics of BSV. The BSV compiler checks that the design is legal according to the static semantics, and produces RTL hardware that exhibits legal behaviors according to the dynamic semantics.

Conceptually, there are three phases in processing a BSV design, just like in Verilog and SystemVerilog:

- *Static checking:* this includes syntactic correctness, type checking and proviso checking.
- *Static elaboration:* actual instantiation of the design and propagation of parameters, producing the module instance hierarchy.
- *Execution:* execution of the design, either in a simulator or as real hardware.

We refer to the first two as the static phase (i.e., pre-execution), and to the third as the dynamic phase. Dynamic semantics are about the temporal behavior of the statically elaborated design, that is, they describe the dynamic execution of rules and methods and their mapping into clocked synchronous hardware.

A BSV program can also contain assertions; assertion checking can occur in all three phases, depending on the kind of assertion.

6.1 Static semantics

The static semantics of BSV are about syntactic correctness, type checking, proviso checking, static elaboration and static assertion checking. Syntactic correctness of a BSV design is checked by the parser in the BSV compiler, according to the grammar described throughout this document.

6.1.1 Type checking

BSV is statically typed, just like Verilog, SystemVerilog, C, C++, and Java. This means the usual things: every variable and every expression has a type; variables must be assigned values that have compatible types; actual and formal parameters/arguments must have compatible types, etc. All this checking is done on the original source code, before any elaboration or execution.

BSV uses SystemVerilog's new tagged union mechanism instead of the older ordinary unions, thereby closing off a certain kind of type loophole. BSV also allows more type parameterization (polymorphism), without compromising full static type checking.

6.1.2 Proviso checking and bit-width constraints

In BSV, overloading constraints and bit-width constraints are expressed using provisos (Sections 4.2 and 14.1.1). Overloading constraints provide an extensible mechanism for overloading.

BSV is stricter about bit-width constraints than Verilog and SystemVerilog in that it avoids implicit zero-extension, sign-extension and truncation of bit-vectors. These operations must be performed consciously by the designer, using library functions, thereby avoiding another source of potential errors.

6.1.3 Static elaboration

As in Verilog and SystemVerilog, static elaboration is the phase in which the design is instantiated, starting with a top-level module instance, instantiating its immediate children, instantiating their children, and so on to produce the complete instance hierarchy.

BSV has powerful generate-like facilities for succinctly expressing regular structures in designs. For example, the structure of a linear pipeline may be expressed using a loop, and the structure of a tree-structured reduction circuit may be expressed using a recursive function. All these are also unfolded and instantiated during static elaboration. In fact, the BSV compiler unfolds all structural loops and functions during static elaboration.

A fully elaborated BSV design consists of no more than the following components:

- A module instance hierarchy. There is a single top-level module instance, and each module instance contains zero or more module instances as children.
- An interface instance. Each module instance presents an interface to its clients, and may itself be a client of zero or more interfaces of other module instances.
- Method definitions. Each interface instance consists of zero or more method definitions.

  A method's body may contain zero or more invocations of methods in other interfaces.

  Every method has an implicit condition, which can be regarded as a single output wire that is asserted only when the method is ready to be invoked. The implicit condition may directly test state internal to its module, and may indirectly test state of other modules by invoking their interface methods.
- Rules. Each module instance contains zero or more rules, each of which contains a condition and an action. The condition is a boolean expression. Both the condition and the action may contain invocations of interface methods of other modules. Since those interface methods can themselves contain invocations of other interface methods, the conditions and actions of a rule may span many modules.

6.2 Dynamic semantics

The dynamic semantics of BSV specify the temporal behavior of rules and methods and their mapping into clocked synchronous hardware.

Every rule has a syntactically explicit condition and action. Both of these may contain invocations of interface methods, each of which has an implicit condition. A rule's *composite condition* consists of its syntactically explicit condition ANDed with the implicit conditions of all the methods invoked in the rule. A rule is said to be *enabled* if its composite condition is true.

6.2.1 Reference semantics

The simplest way to understand the dynamic semantics is through a reference semantics, which is completely sequential. However, please do not equate this with slow execution; the execution steps described below are not the same as clocks; we will see in the next section that many steps can be mapped into each clock. The execution of any BSV program can be understood using the following very simple procedure:

Repeat forever:
Step: Pick any *one* enabled rule, and perform its action.
(We say that the rule is *fired* or *executed*.)

Note that after each step, a different set of rules may be enabled, since the current rule's action will typically update some state elements in the system which, in turn, may change the value of rule conditions and implicit conditions.

Also note that this sequential, reference semantics does not specify how to choose which rule to execute at each step. Thus, it specifies a *set* of legal behaviors, not just a single unique behavior. The principles that determine which rules in a BSV program will be chosen to fire (and, hence, more precisely constrain its behavior) are described in section 6.2.3.

Nevertheless, this simple reference semantics makes it very easy for the designer to reason about invariants (correctness conditions). Since only one rule is executed in each step, we only have to look at the actions of each rule in isolation to check how it maintains or transforms invariants. In particular, we do not have to consider interactions with other rules executing simultaneously.

Another way of saying this is: each rule execution can be viewed as an *atomic state transition*.[4] Race conditions, the bane of the hardware designer, can generally be explained as an atomicity violation; BSV's rules are a powerful way to avoid most races.

The reference semantics is based on Term Rewriting Systems (TRSs), a formalism supported by decades of research in the computer science community [Ter03]. For this reason, we also refer to the reference semantics as "the TRS semantics of BSV."

6.2.2 Mapping into efficient parallel clocked synchronous hardware

A BSV design is mapped by the BSV compiler into efficient parallel clocked synchronous hardware. In particular, the mapping permits multiple rules to be executed in each clock cycle. This is done in a manner that is consistent with the reference TRS semantics, so that any correctness properties ascertained using the TRS semantics continue to hold in the hardware.

Standard clocked synchronous hardware imposes the following restrictions:

- Persistent state is updated only once per clock cycle, at a clock edge. During a clock cycle, values read from persistent state elements are the ones that were registered in the last cycle.
- Clock-speed requirements place a limit on the amount of combinational computation that can be performed between state elements, because of propagation delay.

The composite condition of each rule is mapped into a combinational circuit whose inputs, possibly many, sense the current state and whose 1-bit output specifies whether this rule is enabled or not.

The action of each rule is mapped into a combinational circuit that represents the state transition function of the action. It can have multiple inputs and multiple outputs, the latter being the computed next-state values.

[4] We use the term *atomic* as it is used in concurrency theory (and in operating systems and databases), i.e., to mean *indivisible*.

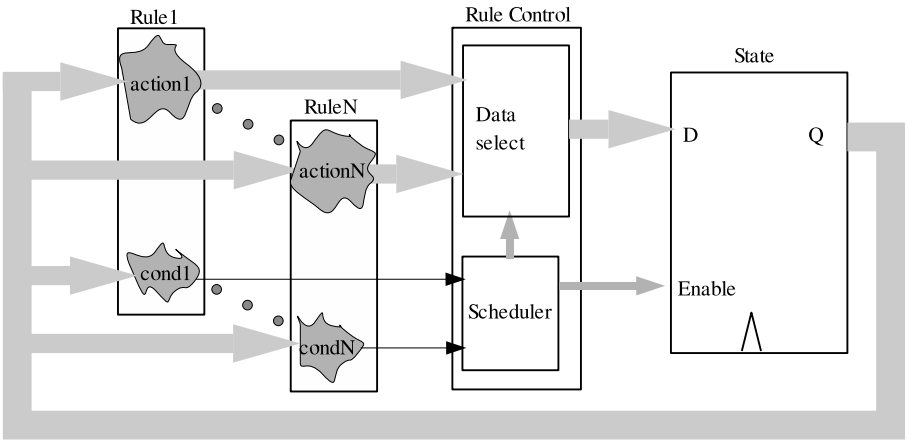

Figure 1: A general scheme for mapping an N-rule system into clocked synchronous hardware.

Figure 1 illustrates a general scheme to compose rule components when mapping the design to clocked synchronous hardware. The State box lumps together all the state elements in the BSV design (as described earlier, state elements are explicitly specified in BSV). The BSV compiler produces a rule-control circuit which conceptually takes all the enable (cond) signals and all the data (action) outputs and controls which of the data outputs are actually captured at the next clock in the state elements. The enable signals feed a *scheduler* circuit that decides which of the rules will actually fire. The scheduler, in turn, controls data multiplexers that select which data outputs reach the data inputs of state elements, and controls which state elements are enabled to capture the new data values. Firing a rule simply means that the scheduler selects its data output and clocks it into the next state.

At each clock, the scheduler selects a subset of rules to fire. Not all subsets are legal. A subset is legal if and only if the rules in the subset can be ordered with the following properties:

- A hypothetical sequential execution of the ordered subset of rules is legal at this point, according to the TRS semantics. In particular, the first rule in the ordered subset is currently enabled, and each subsequent rule would indeed be enabled when execution reaches it in the hypothetical sequence.

  A special case is where all rules in the subset are already currently enabled, and no rule would be disabled by execution of prior rules in the order.

- The hardware execution produces the same net effect on the state as the hypothetical sequential execution, even though the hardware execution performs reads and writes in a different order from the hypothetical sequential execution.

The BSV compiler performs a very sophisticated analysis of the rules in a design and synthesizes an efficient hardware scheduler that controls execution in this manner.

Note that the scheme in Figure 1 is for illustrative purposes only. First, it lumps together all the state, shows a single rule-control box, etc., whereas in the real hardware generated by the BSV compiler these are distributed, localized and modular. Second, it is not the only way to map the design into clocked synchronous hardware. For example, any two enabled rules can also be executed in a single clock by feeding the action outputs of the first rule into the action inputs of the second rule, or by synthesizing hardware for a composite circuit that computes the same function as the composition of the two actions, and so on. In general, these alternative schemes may be more complex to analyze, or may increase total propagation delay, but the compiler may use them in special circumstances.

In summary, the BSV compiler performs a detailed and sophisticated analysis of rules and their interactions, and maps the design into very efficient, highly parallel, clocked synchronous hardware including a dynamic scheduler that allows many rules to fire in parallel in each clock, but always in a manner that is consistent with the reference TRS semantics. The designer can use the simple reference semantics to reason about correctness properties and be confident that the synthesized parallel hardware will preserve those properties. (See Section 13.3 for the "scheduling attributes" mechanism using which the designer can guide the compiler in implementing the mapping.)

When coding in other HDLs, the designer must maintain atomicity manually. He must recognize potential race conditions, and design the appropriate data paths, control and synchronization to avoid them. Reasoning about race conditions can cross module boundaries, and can be introduced late in the design cycle as the problem specification evolves. The BSV compiler automates all of this and, further, is capable of producing RTL that is competitive with hand-coded RTL.

6.2.3 How rules are chosen to fire

The previous section described how an efficient circuit can be built whose behavior will be consistent with sequential TRS semantics of BSV. However, as noted previously, the sequential reference semantics can be consistent with a range of different behaviors. There are two rule scheduling principles that guide the BSV compiler in choosing which rules to schedule in a clock cycle (and help a designer build circuits with predictable behavior). Except when overridden by an explicit user command or annotation, the BSV compiler schedules rules according to the following two principles:

1. Every rule enabled during a clock cycle will either be fired as part of that clock cycle or a warning will be issued during compilation.
2. A rule will fire at most one time during a particular clock cycle.

The first principle comes into play when two (or more) rules conflict - either because they are competing for a limited resource or because the result of their simultaneous execution is not consistent with any sequential rule execution. In the absence of a user annotation, the compiler will arbitrarily choose [5] which rule to prioritize, but *must* also issue a warning. This guarantees the designer is aware of the ambiguity in the design and can correct it. It might be corrected by changing the rules themselves (rearranging their predicates so they are never simultaneously applicable, for example) or by adding an urgency annotation which tells the compiler which rule to prefer (see section 13.3.3). When there are no scheduling warnings, it is guaranteed that the compiler is making no arbitrary choices about which rules to execute.

The second principle ensures that continuously enabled rules (like a counter increment rule) will not executed an unpredictable number of time during a clock cycle. According to the first rule scheduling principle, a rule that is always enabled will be executed at least once during a clock cycle. However, since the rule remains enabled it theoretically could execute multiple times in a clock cycle (since that behavior would be consistent with a sequential semantics). Since rules (even simple things like a counter increment) consume limited resources (like register write ports) it is pragmatically useful to restrict them to executing only once in a cycle (in the absence of specific user instructions to the contrary). Executing a continuously enabled rule only once in a cycle is also the more straightforward and intuitive behavior.

Together, these two principles allow a designer to completely determine the rules that will be chosen to fire by the schedule (and, hence, the behavior of the resulting circuit).

[5] The compiler's choice, while arbitrary, is deterministic. Given the same source and compiler version, the same schedule (and, hence, the same hardware) will be produced. However, because it is an arbitrary choice, it can be sensitive to otherwise irrelevant details of the program and is not guaranteed to remain the same if the source or compiler version changes.

7 User-defined types (type definitions)

User-defined types may appear at the top level of packages.

typeDef	::=	*typedefSynonym*
	|	*typedefEnum*
	|	*typedefStruct*
	|	*typedefTaggedUnion*

As a matter of style, BSV requires that all enumerations, structs and unions be declared only via `typedef`, i.e., it is not possible directly to declare a variable, formal parameter or formal argument as an enum, struct or union without first giving that type a name using a typedef.

Each typedef of an enum, struct or union introduces a new type that is different from all other types. For example, even if two typedefs give names to struct types with exactly the same corresponding member names and types, they define two distinct types.

Other typedefs, i.e., not involving an enum, struct or union, merely introduce type synonyms for existing types.

7.1 Type synonyms

Type synonyms are just for convenience and readability, allowing one to define shorter or more meaningful names for existing types. The new type and the original type can be used interchangeably anywhere.

typedefSynonym	::=	`typedef` *type typeDefType* `;`
typeDefType	::=	*typeIde* [*typeFormals*]
typeFormals	::=	`#(` *typeFormal* { `,` *typeFormal* }`)`
typeFormal	::=	[`numeric`] `type` *typeIde*

Examples. Defining names for bit vectors of certain lengths:

```
typedef bit [7:0]   Byte;
typedef bit [31:0]  Word;
typedef bit [63:0]  LongWord;
```

Examples. Defining names for polymorphic data types.

```
typedef Tuple#3(a, a, a) Triple#(type a);

typdef Int#(n) MyInt#(type n);
```

The above example could also be written as:

```
typedef Int#(n) MyInt#(numeric type n);
```

The `numeric` is not required because the parameter to `Int` will always be numeric. `numeric` is only required when the compiler can't determine whether the parameter is a numeric or non-numeric type. It will then default to assuming it is non-numeric. The user can override this default by specifying `numeric` in the `typedef` statement.

A `typedef` statement can be used to define a synonym for an already defined synonym. Example:

```
typedef Triple#(Longword) TLW;
```

Since an Interface is a type, we can have nested types:

```
typedef Reg#(Vector#(8, UInt#(8))) ListReg;
typedef List#(List#(Bit#(4)))      ArrayOf4Bits;
```


7.2 Enumerations

typedefEnum ::= `enum {` *typedefEnumElement* { `,` *typedefEnumElement* } `}` *Identifier* [*derives*] `;`

typedefEnumElement ::= *Identifier* [`=` *intLiteral*]
| *Identifier* `[`*intLiteral*`]` [`=` *intLiteral*]
| *Identifier* `[`*intLiteral* `:` *intLiteral*`]` [`=` *intLiteral*]

Enumerations (enums) provide a way to define a set of unique symbolic constants, also called *labels* or *member names*. Each enum definition creates a new type different from all other types. The newly defined labels must be unique (within a package), i.e., two different enums cannot use common labels. Enumeration labels must begin with an uppercase letter.

The optional *derives* clause is discussed in more detail in Sections 4.3 and 14.1. One common form is `deriving (Bits)`, which tells the compiler to generate a bit-representation for this enum. Another common form of the clause is `deriving (Eq)`, which tells the compiler to pick a default equality operation for these labels, so they can also be tested for equality and inequality. A third common form is `deriving (Bounded)`, which tells the compiler to define constants `minBound` and `maxBound` for this type, equal in value to the first and last labels in the enumeration. These specifications can be combined, e.g., `deriving (Bits, Eq, Bounded)`. All these default choices for representation, equality and bounds can be overridden (see Section 14.1).

The declaration may specify the encoding used by `deriving(Bits)` by assigning numbers to tags. When an assignment is omitted, the tag receives an encoding of the previous tag incremented by one; when the encoding for the initial tag is omitted, it defaults to zero. Specifying the same encoding for more than one tag results in an error.

Multiple tags may be declared by using the index (*Tag*`[`*ntags*`]`) or range (*Tag*`[`*start*`:`*end*`]`) notation. In the former case, *ntags* tags will be generated, from `Tag0` to `Tag`*n-1*; in the latter case, $|end - start| + 1$ tags, from `Tag`*start* to `Tag`*end*.

Example. The boolean type can be defined in the language itself:

```
typedef enum { False, True } Bool deriving (Bits, Eq);
```

The compiler will pick a one-bit representation, with 1'b0 and 1'b1 as the representations for `False` and `True`, respectively. It will define the `==` and `!=` operators to also work on `Bool` values.

Example. Excerpts from the specification of a processor:

```
typedef enum { R0, R1, ..., R31 }  RegName deriving (Bits);
typedef RegName  Rdest;
typedef RegName  Rsrc;
```

The first line defines an enum type with 32 register names. The second and third lines define type synonyms for `RegName` that may be more informative in certain contexts ("destination" and "source" registers). Because of the `deriving` clause, the compiler will pick a five-bit representation, with values 5'h00 through 5'h1F for `R0` through `R31`.

Example. Tag encoding when `deriving(Bits)` can be specified manually:

```
typedef enum {
  Add = 5,
  Sub = 0,
  Not,
  Xor = 3,
  ...
} OpCode deriving (Bits);
```


The `Add` tag will be encoded to five, `Sub` to zero, `Not` to one, and `Xor` to three.

Example. A range of tags may be declared in a single clause:

```
typedef enum {
  Foo[2],
  Bar[5:7],
  Quux[3:2]
} Glurph;
```

This is equivalent to the declaration

```
typedef enum {
  Foo0,
  Foo1,
  Bar5,
  Bar6,
  Bar7,
  Quux3,
  Quux2
} Glurph;
```

7.3 Structs and tagged unions

A struct definition introduces a new record type.

SystemVerilog has ordinary unions as well as tagged unions, but in BSV we only use tagged unions, for several reasons. The principal benefit is safety (verification). Ordinary unions open a serious type-checking loophole, whereas tagged unions are completely type-safe. Other reasons are that, in conjunction with pattern matching (Section 10), tagged unions yield much more succinct and readable code, which also improves correctness. In the text below, we may simply say "union" for brevity, but it always means "tagged union."

typedefStruct ::= `typedef struct {`
{ *structMember* }
`}` *typeDefType* [*derives*] ;

typedefTaggedUnion ::= `typedef union tagged {`
{ *unionMember* }
`}` *typeDefType* [*derives*] ;

structMember ::= *type identifier* ;
| *subStruct identifier* ;
| *subUnion identifier* ;

unionMember ::= *type Identifier* ;
| *subStruct Identifier* ;
| *subUnion Identifier* ;
| `void` *Identifier* ;

subStruct ::= `struct {`
{ *structMember* }
`}`

subUnion ::= `union tagged {`
{ *unionMember* }
`}`

typeDefType ::= *typeIde* [*typeFormals*]

typeFormals ::= `#` `(` *typeFormal* { `,` *typeFormal* }`)`

typeFormal ::= [`numeric`] `type` *typeIde*

All types can of course be mutually nested if mediated by typedefs, but structs and unions can also be mutually nested directly, as described in the syntax above. Structs and unions contain *members*. A union member (but not a struct member) can have the special `void` type (see the types `MaybeInt` and `Maybe` in the examples below for uses of `void`). All the member names in a particular struct or union must be unique, but the same names can be used in other structs and members; the compiler will try to disambiguate based on type.

A struct value contains the first member *and* the second member *and* the third member, and so on. A union value contains just the first member *or* just the second member *or* just the third member, and so on. Struct member names must begin with a lowercase letter, whereas union member names must begin with an uppercase letter.

In a tagged union, the member names are also called *tags*. Tags play a very important safety role. Suppose we had the following:

```
typedef union tagged { int Tagi; OneHot Tagoh; } U deriving (Bits);
U  x;
```

The variable `x` not only contains the bits corresponding to one of its member types `int` or `OneHot`, but also some extra bits (in this case just one bit) that remember the tag, 0 for `Tagi` and 1 for `Tagoh`. When the tag is `Tagi`, it is impossible to read it as a `OneHot` member, and when the tag is `Tagoh` it is impossible to read it as an `int` member, i.e., the syntax and type checking ensure this. Thus, it is impossible accidentally to misread what is in a union value.

The optional *derives* clause is discussed in more detail in Section 14.1. One common form is `deriving (Bits)`, which tells the compiler to pick a default bit-representation for the struct or union. For structs it is simply a concatenation of the representations of the members. For unions, the representation consists of $t + m$ bits, where t is the minimum number of bits to code for the tags in this union and m is the number of bits for the largest member. Every union value has a code in the t-bit field that identifies the tag, concatenated with the bits of the corresponding member, right-justified in the m-bit field. If the member needs fewer than m bits, the remaining bits (between the tag and the member bits) are undefined.

Struct and union typedefs can define new, polymorphic types, signalled by the presence of type parameters in `#(...)`. Polymorphic types are discussed in section 4.1.

Section 9.11 on struct and union expressions describes how to construct struct and union values and to access and update members. Section 10 on pattern-matching describes a more high-level way to access members from structs and unions and to test union tags.

Example. Ordinary, traditional record structures:

```
typedef struct { int x; int y; } Coord;
typedef struct { Addr pc; RegFile rf; Memory mem; }  Proc;
```

Example. Encoding instruction operands in a processor:

```
typedef union tagged {
    bit  [4:0] Register;
    bit [21:0] Literal;
    struct {
        bit  [4:0] RegAddr;
        bit  [4:0] RegIndex;
    } Indexed;
} InstrOperand;
```


An instruction operand is either a 5-bit register specifier, a 22-bit literal value, or an indexed memory specifier, consisting of two 5-bit register specifiers.

Example. Encoding instructions in a processor:

```
typedef union tagged {
   struct {
       Op op; Reg rs; CPUReg rt; UInt16 imm;
   } Immediate;

   struct {
       Op op; UInt26 target;
   } Jump;
} Instruction
  deriving (Bits);
```

An `Instruction` is either an `Immediate` or a `Jump`. In the former case, it contains a field, `op`, containing a value of type `Op`; a field, `rs`, containing a value of type `Reg`; a field, `rt`, containing a value of type `CPUReg`; and a field, `imm`, containing a value of type `UInt16`. In the latter case, it contains a field, `op`, containing a value of type `Op`, and a field, `target`, containing a value of type `UInt26`.

Example. Optional integers (an integer together with a valid bit):

```
typedef union tagged {
    void    Invalid;
    int     Valid;
} MaybeInt
  deriving (Bits);
```

A `MaybeInt` is either invalid, or it contains an integer (Valid tag). The representation of this type will be 33 bits— one bit to represent `Invalid` or `Valid` tag, plus 32 bits for an `int`. When it carries an invalid value, the remaining 32 bits are undefined. It will be impossible to read/interpret those 32 bits when the tag bit says it is `Invalid`.

This `MaybeInt` type is very useful, and not just for integers. We generalize it to a polymorphic type:

```
typedef union tagged {
    void  Invalid;
    a     Valid;
} Maybe#(type a)
  deriving (Bits);
```

This `Maybe` type can be used with any type `a`. Consider a function that, given a key, looks up a table and returns some value associated with that key. Such a function can return either an invalid result (`Invalid`), if the table does not contain an entry for the given key, or a valid result `Valid` v if v is associated with the key in the table. The type is polymorphic (type parameter `a`) because it may be used with lookup functions for integer tables, string tables, IP address tables, etc. In other words, we do not over-specify the type of the value v at which it may be used.

See Section 12.4 for an important, predefined set of struct types called *Tuples* for adhoc structs of between two and seven members.

8 Variable declarations and statements

Statements can occur in various contexts: in packages, modules, function bodies, rule bodies, action blocks and actionvalue blocks. Some kinds of statements have been described earlier because they were specific to certain contexts: module definitions (*moduleDef*) and instantiation (*moduleInst*), interface declarations (*interfaceDecl*), type definitions (*typeDef*), method definitions (*methodDef*) inside modules, rules (*rule*) inside modules, and action blocks (*actionBlock*) inside modules.

Here we describe variable declarations, register assignments, variable assignments, loops, and function definitions. These can be used in all statement contexts.

8.1 Variable and array declaration and initialization

Variables in BSV are used to name intermediate values. Unlike Verilog and SystemVerilog, variables never represent state, i.e., they do not hold values over time. Every variable's type must be declared, after which it can be bound to a value one or more times.

One or more variables can be declared by giving the type followed by a comma-separated list of identifiers with optional initializations:

varDecl	::=	*type varInit* { , *varInit* } ;
varInit	::=	*identifier* [*arrayDims*] [= *expression*]
arrayDims	::=	[*expression*] { [*expression*] }

The declared identifier can be an array (when *arrayDims* is present). The *expression*s in *arrayDims* represent the array dimensions, and must be constant expressions (i.e., computable during static elaboration). The array can be multidimensional.

Note that array variables are distinct from the `RegFile` data type, which is described in Appendix C. RegFile variables are just a structuring mechanism for values, whereas the `RegFile` type represents a particular hardware module, like a register file, with a limited number of read and write ports. In many programs, array variables are used purely for static elaboration, e.g., an array of registers is just a convenient way to refer to a collection of registers with a numeric index.

Each declared variable can optionally have an initialization.

Example. Declare two `integer` variables and initialize them:

```
Integer x = 16, y = 32;
```

Example. Declare two array identifiers `a` and `b` containing `int` values at each index:

```
int  a[20], b[40];
```

Example. Declare an array of 3 `Int#(5)` values and initialize them:

```
Int#(5) xs[3] = {14, 12, 9};
```

Example. Declare an array of 3 arrays of 4 `Int#(5)` values and initialize them:

```
Int#(5) xs[3][4] = {{1,2,3,4},
                    {5,6,7,8},
                    {9,10,11,12}};
```

Example. The array values can be polymorphic, but they must defined during elaboration:

```
Get #(a) gs[3] = {g0,g2, g2};
```


8.2 Variable assignment

A variable can be bound to a value using assignment:

varAssign	::=	*lValue* `=` *expression* `;`
lValue	::=	*identifier*
	|	*lValue* `.` *identifier*
	|	*lValue* `[` *expression* `]`
	|	*lValue* `[` *expression* `:` *expression* `]`

The left-hand side (*lValue*) in its simplest form is a simple variable (*identifier*).

Example. Declare a variable `wordSize` to have type `Integer` and assign it the value 16:

```
Integer wordSize;
wordSize = 16;
```

Multiple assignments to the same variable are just a shorthand for a cascaded computation. Example:

```
int x;
x = 23;
// Here, x represents the value 23
x = ifc.meth (34);
// Here, x represents the value returned by the method call
x = x + 1;
// Here, x represents the value returned by the method call, plus 1
```

Note that these assignments are ordinary, zero-time assignments, i.e., they never represent a dynamic assignment of a value to a register. These assignments only represent the convenient naming of an intermediate value in some zero-time computation. Dynamic assignments are always written using the non-blocking assignment operator `<=`, and are described in Section 8.4.

In general, the left-hand side (*lValue*) in an assignment statement can be a series of index- and field-selections from an identifier representing a nesting of arrays, structs and unions. The array-indexing expressions must be computable during static elaboration.

For bit vectors, the left-hand side (*lValue*) may also be a range between two indices. The indices must be computable during static elaboration, and, if the indices are not literal constants, the right-hand side of the assignment must have a defined bit width.

Example. Update an array variable `b`:

```
b[15] = foo.bar(x);
```

Example. Update bits 15 to 8 (inclusive) of a bit vector `b`:

```
b[15:8] = foo.bar(x);
```

Example. Update a struct variable (using the processor example from Section 7.3):

```
cpu.pc = cpu.pc + 4;
```

Semantically, this can be seen as an abbreviation for:

```
cpu = Proc { pc: cpu.pc + 4, rf: cpu.rf, mem: cpu.mem };
```


i.e., it reassigns the struct variable to contain a new struct value in which all members other than the updated member have their old values. The right-hand side is a struct expression; these are described in Section 9.11.

Update of tagged union variables is done using normal assignment notation, i.e., one replaces the current value in a tagged union variable by an entirely new tagged union value. In a struct it makes sense to update a single member and leave the others unchanged, but in a union, one member replaces another. Example (extending the previous processor example):

```
typedef union tagged {
    bit  [4:0] Register;
    bit [21:0] Literal;
    struct {
        bit  [4:0] regAddr;
        bit  [4:0] regIndex;
    } Indexed;
} InstrOperand;
...
InstrOperand  orand;
...
orand = tagged Indexed { regAddr:3, regIndex:4 };
...
orand = tagged Register 23;
```

The right-hand sides of the assignments are tagged union expressions; these are described in Section 9.11.

8.3 Implicit declaration and initialization

The `let` statement is a shorthand way to declare and initialize a variable in a single statement. A variable which has not been declared can be assigned an initial value and the compiler will infer the type of the variable from the expression on the right hand side of the statement:

varDecl ::= `let` *identifier* `=` *expression* `;`

Example:

```
let n = valueof(BuffSize);
```

The pseudo-function `valueof` returns an `Integer` value, which will be assigned to `n` at compile time. Thus the variable `n` is assumed to have the type of Integer.

If the expression is the value returned by an actionvalue method, the notation will be:

varAssign ::= `let` *identifier* `<-` *expression* `;`

Note the difference between this statement:

```
let m1 = m1displayfifo.first;
```

and this statement:

```
let z1 <- rndm.get;
```

In the first example, `m1displayfifo.first` is a value method; `m1` is assigned the value and type returned by the value method. In the latter, `rndm.get` is an actionvalue method; `z1` is assigned the value and type returned by the actionvalue method.

8.4 Register reads and writes

Register writes occur primarily inside rules and methods.

regWrite ::= *identifier* `<=` *expression*
| `(` *expression* `)` `<=` *expression*

The left-hand side must have a register interface type, i.e., `Reg#(`t`)` for some type t that has a representation in bits. It is either an identifier or a parenthesized expression (e.g., the register interface could be selected from an array of register interfaces). The right-hand side must have the same type t, i.e., it is an expression that evaluates to a value of type t. BSV allows only the so-called *non-blocking assignments* of Verilog, i.e., the statement specifies that the register gets the new value at the end of the current cycle, and is only available in the next cycle.

Following BSV's principle that all state elements (including registers) are module instances, and all interaction with a module happens through its interface, a register assignment r`<=`e is just a convenient alternative notation for a method call:

r._write (e)

Similarly, if r is an expression of type `Reg#(`t`)`, then mentioning r in an expression is just a convenient alternative notation for a method call:

r._read ()

Example. Instantiating a register interface and a register, and using it:

```
Reg#(int) r();          // create a register interface
mkReg#(0) the_r (r);    // create a register the_r with interface r
...
...
rule ...
    r <= r + 1;          // Convenient notation for: r._write (r._read() + 1)
endrule
```

Example. Updating a register in an array of registers:

```
RegFile#(Integer, Reg#(int)) regs;
...
(reg[3]) <= regs[3] + 1;    // increment the fourth register
```

The register assignment notation can be used for partial register updates, when the register contains an array of elements of some type t (in a particular case, this could be an array of bits).

regWrite ::= *identifier arrayIndexes* `<=` *expression*

arrayIndexes ::= `[` *expression* `]` { `[` *expression* `]` }

The *expressions* in *arrayIndexes* must be static expressions. This notation is just a shorthand for a whole register update where only the selected element is updated. In other words,

```
x[j] <= v;
```

is a shorthand for:

```
x <= replace (x, j, v);
```


where `replace` is a pure function that takes the whole value from register `x` and produces a whole new value with the `j`'th element replaced by `v`. The statement then assigns this new value to the register `x`.

It is important to understand the distinction between the following:

```
x[3]   <= e;
(y[3]) <= e;
```

In the former case, `x` must be a register containing an array of items, and the statement updates item "3" in the register. In the latter case, signalled by the parentheses, `y` is an array of registers, and register "3" is updated. Currently, it is not possible to mix these notations, i.e., one cannot write a single statement to perform a partial update of a register in an array of registers; this can always be expressed with no overhead by splitting it into two statements, the first one binding a variable to the particular register, and the second statement performing the partial update. Example:

```
RegFile#(Integer,Reg#(RegFile#(bit[3:0],int))) ys;
Reg#(RegFile#(bit[3:0],int)) y4;
...
y4 = ys[4];        // Let y4 refer to the fourth register
y4[3] <= e;        // Update y4's third element
```

8.5 Begin-end statements

A begin-end statement is a block that allows one to collect multiple statements into a single statement, which can then be used in any context where a statement is required.

<ctxt>BeginEndStmt ::= `begin` [`:` *identifier*]
{ *<ctxt>Stmt* }
`end` [`:` *identifier*]

The optional identifier labels are currently used for documentation purposes only; in the future they may be used for hierarchical references. The statements contained in the block can contain local variable declarations and all the other kinds of statements. Example:

```
module mkBeginEnd#(Bit#(2) sel) ();
   Reg#(Bit#(4)) a     <- mkReg(0);
   Reg#(Bool)    done  <- mkReg(False);

   rule decode (!done);
     case (sel)
        2'b00: a <= 0;
        2'b01: a <= 1;
        2'b10: a <= 2;
        2'b11: begin
           a    <= 3;        //in the 2'b11 case we don't want more than
           done <= True;     //one action done, therefore we add begin/end
        end
     endcase
   endrule
endmodule
```


8.6 Conditional statements

Conditional statements include `if` statements and `case` statements. An `if` statement contains a predicate, a statement representing the true arm and, optionally, the keyword `else` followed by a statement representing the false arm.

```
<ctxt>If              ::= if ( condPredicate )
                              <ctxt>Stmt
                          [ else
                              <ctxt>Stmt ]

condPredicate         ::= exprOrCondPattern { &&& exprOrCondPattern }
exprOrCondPattern     ::= expression
                      |   expression matches pattern
```

If-statements have the usual semantics— the predicate is evaluated, and if true, the true arm is executed, otherwise the false arm (if present) is executed. The predicate can be any boolean expression. More generally, the predicate can include pattern matching, and this is described in Section 10, on pattern matching.

There are two kinds of case statements: ordinary case statements and pattern-matching case statements. Ordinary case statements have the following grammar:

```
<ctxt>Case            ::= case ( expression )
                              { <ctxt>CaseItem }
                              [ <ctxt>DefaultItem ]
                          endcase

<ctxt>CaseItem        ::= expression [ , expression ] : <ctxt>Stmt

<ctxt>DefaultItem     ::= default [ : ] <ctxt>Stmt
```

Each case item contains a left-hand side and a right-hand side, separated by a colon. The left-hand side contains a series of expressions, separated by commas. The case items may optionally be followed, finally, by a default item (the colon after the `default` keyword is optional).

Case statements are equivalent to an expansion into a series of nested if-then-else statements. For example:

```
case (e1)
  e2, e3    : s2;
  e4        : s4;
  e5, e6, e7: s5;
  default   : s6;
endcase
```

is equivalent to:

```
x1 = e1;    // where x1 is a new variable:
if        (x1 == e2)  s2;
else if   (x1 == e3)  s2;
else if   (x1 == e4)  s4;
else if   (x1 == e5)  s5;
else if   (x1 == e6)  s5;
else if   (x1 == e7)  s5;
else                  s6;
```

The case expression (`e1`) is evaluated once, and tested for equality in sequence against the value of each of the left-hand side expressions. If any test succeeds, then the corresponding right-hand

side statement is executed. If no test succeeds, and there is a default item, then the default item's right-hand side is executed. If no test succeeds, and there is no default item, then no right-hand side is executed.

Example:

```
module mkConditional#(Bit#(2) sel) ();
   Reg#(Bit#(4)) a     <- mkReg(0);
   Reg#(Bool)    done  <- mkReg(False);

   rule decode ;
     case (sel)
        2'b00: a <= 0;
        2'b01: a <= 1;
        2'b10: a <= 2;
        2'b11: a <= 3;
     endcase
   endrule

   rule finish ;
     if (a == 3)
        done <= True;
     else
        done <= False;
   endrule
endmodule
```

Pattern-matching case statements are described in Section 10.

8.7 Loop statements

BSV has `for` loops and `while` loops.

It is important to note that this use of loops does not express time-based behavior. Instead, they are used purely as a means to express zero-time iterative computations, i.e., they are statically unrolled and express the concatenation of multiple instances of the loop body statements. In particular, the loop condition must be evaluable during static elaboration. For example, the loop condition can never depend on a value in a register, or a value returned in a method call, which are only known during execution and not during static elaboration.

See Section 11 on FSMs for an alternative use of loops to express time-based (temporal) behavior.

8.7.1 While loops

<ctxt>While ::= `while (` *expression* `)`
<ctxt>Stmt

While loops have the usual semantics. The predicate *expression* is evaluated and, if true, the loop body statement is executed, and then the while loop is repeated. Note that if the predicate initially evaluates false, the loop body is not executed at all.

Example. Sum the values in an array:

```
int a[32];
int x = 0;
```



```
int j = 0;
...
while (j < 32)
    x = x + a[j];
```

8.7.2 For loops

<ctxt>For	::=	`for (` *forInit* `;` *forTest* `;` *forIncr* `)` *<ctxt>Stmt*
forInit	::=	*forOldInit* | *forNewInit*
forOldInit	::=	*simpleVarAssign* [`,` *simpleVarAssign*]
simpleVarAssign	::=	*identifier* `=` *expression*
forNewInit	::=	*type identifier* `=` *expression* [`,` *simpleVarDeclAssign*]
simpleVarDeclAssign	::=	[*type*] *identifier* `=` *expression*
forTest	::=	*expression*
forIncr	::=	*varIncr* [`,` *varIncr*]
varIncr	::=	*identifier* `=` *expression*

The *forInit* phrase can either initialize previously declared variables (*forOldInit*), or it can declare and initialize new variables whose scope is just this loop (*forNewInit*). They differ in whether or not the first thing after the open parenthesis is a type.

In *forOldInit*, the initializer is just a comma-separated list of variable assignments.

In *forNewInit*, the initializer is a comma-separated list of variable declarations and initializations. After the first one, not every initializer in the list needs a *type*; if missing, the type is the nearest *type* earlier in the list. The scope of each variable declared extends to subsequent initializers, the rest of the for-loop header, and the loop body statement.

Example. Copy values from one array to another:

```
int a[32], b[32];
...
...
for (int i = 0, j = i+offset; i < 32-offset; i = i+1, j = j+1)
    a[i] = b[j];
```

8.8 Function definitions

A function definition is introduced by the `function` keyword. This is followed by the type of the function return-value, the name of the function being defined, the formal arguments, and optional provisos (provisos are discussed in more detail in Section 14.1). After this is the function body and, finally, the `endfunction` keyword that is optionally labelled again with the function name. Each formal argument declares an identifier and its type.

functionDef	::=	[*modgenAttribute*] *functionProto* *functionBody* `endfunction` [`:` *identifier*]
functionProto	::=	`function` *type identifier* `(` [*functionFormals*] `)` [*provisos*] `;`
functionFormals	::=	*functionFormal* { `,` *functionFormal* }
functionFormal	::=	*type identifier*

The function body can contain the usual repertoire of statements:

functionBody	::=	*actionBlock*
	|	*actionValueBlock*
	|	{ *functionBodyStmt* }
functionBodyStmt	::=	*<functionBody>If* | *<functionBody>Case*
	|	*<functionBody>BeginEndStmt*
	|	*varDecl* | *varAssign*
	|	*varDo* | *varDeclDo*
	|	*functionDef*
	|	*functionCall*
	|	*systemTaskCall*
	|	(*expression*)
	|	*returnStmt*
returnStmt	::=	`return` *expression* ;

A value can be returned from a function in two ways, as in SystemVerilog. The first method is to assign a value to the function name used as an ordinary variable. This "variable" can be assigned multiple times in the function body, including in different arms of conditionals, in loop bodies, and so on. The function body is viewed as a traditional sequential program, and value in the special variable at the end of the body is the value returned. However, the "variable" cannot be used in an expression (e.g., on the right-hand side of an assignment) because of ambiguity with recursive function calls.

Alternatively, one can use a `return` statement anywhere in the function body to return a value immediately without any further computation. If the value is not explicitly returned nor bound, the returned value is undefined.

Example. The boolean negation function:

```
function Bool notFn (Bool x);
    if (x) notFn = False;
    else   notFn = True;
endfunction: notFn
```

Example. The boolean negation function, but using `return` instead:

```
function Bool notFn (Bool x);
    if (x) return False;
    else   return True;
endfunction: notFn
```

Example. The factorial function, using a loop:

```
function int factorial (int n);
   int f = 1, j = 0;
   while (j < n)
     begin
       f = f * j;
       j = j + 1;
     end
   factorial = f;
endfunction: factorial
```

Example. The factorial function, using recursion:


```
function int factorial (int n);
   if (n <= 1) return (1);
   else return (n * factorial (n - 1));
endfunction: factorial
```

9 Expressions

Expressions occur on the right-hand sides of variable assignments, on the left-hand and right-hand side of register assignments, as actual parameters and arguments in module instantiation, function calls, method calls, array indexing, and so on.

There are many kinds of primary expressions. Complex expressions are built using the conditional expressions and unary and binary operators.

expression	::=	*condExpr*
	|	*operatorExpr*
	|	*exprPrimary*
exprPrimary	::=	*identifier*
	|	*intLiteral*
	|	*stringLiteral*
	|	*systemFunctionCall*
	|	`(` *expression* `)`
	|	··· see other productions ···

9.1 Don't-care expressions

When the value of an expression does not matter, a *don't-care* expression can be used. It is written with just a question mark and can be used at any type. The compiler will pick a suitable value.

exprPrimary	::=	`?`

A don't-care expression is similar, but not identical to, the `x` value in Verilog, which represents an unknown value. A don't-care expression is unknown to the programmer, but represents a particular fixed value chosen statically by the compiler.

The programmer is encouraged to use don't-care values where possible, both because it is useful documentation and because the compiler can often choose values that lead to better circuits.

Example:

```
module mkDontCare ();

// instantiating registers where the initial value is "Dontcare"
   Reg#(Bit#(4)) a     <- mkReg(?);
   Reg#(Bit#(4)) b     <- mkReg(?);

   Bool   done  = (a==b);
// defining a Variable with an initial value of "Dontcare"
   Bool   mybool  = ?;
endmodule
```


9.2 Conditional expressions

Conditional expressions include the conditional operator and case expressions. The conditional operator has the usual syntax:

condExpr	::=	*condPredicate* `?` *expression* `:` *expression*
condPredicate	::=	*exprOrCondPattern* { `&&&` *exprOrCondPattern* }
exprOrCondPattern	::=	*expression*
	|	*expression* `matches` *pattern*

Conditional expressions have the usual semantics. In an expression e_1`:`e_2`:`e_3, e_1 can be a boolean expression. If it evaluates to `True`, then the value of e_2 is returned; otherwise the value of e_3 is returned. More generally, e_1 can include pattern matching, and this is described in Section 10, on pattern matching

Example.

```
module mkCondExp ();

// instantiating registers
   Reg#(Bit#(4)) a     <- mkReg(0);
   Reg#(Bit#(4)) b     <- mkReg(0);

   rule dostuff;
      a <= (b>4) ? 2 : 10;
   endrule
endmodule
```

Case expressions are described in Section 10, on pattern matching.

9.3 Unary and binary operators

operatorExpr	::=	*unop expression*
	|	*expression binop expression*

Binary operator expressions are built using the *unop* and *binop* operators listed in the following table, which are a subset of the operators in SystemVerilog. The operators are listed here in order of decreasing precedence.

Operator	Associativity	Comments
+ - ! ~	n/a	Unary: plus, minus, logical not, bitwise invert
&	n/a	Unary: and reduction
~&	n/a	Unary: nand reduction
|	n/a	Unary: or reduction
~|	n/a	Unary: nor reduction
^	n/a	Unary: xor reduction
^~ ~^	n/a	Unary: xnor reduction
* / %	Left	multiplication, division, modulus
+ -	Left	addition, subtraction
<< >>	Left	left and right logical shift
<= >= < >	Left	comparison ops
== !=	Left	equality, inequality
&	Left	bitwise and
^	Left	bitwise xor
^~ ~^	Left	bitwise equivalence
|	Left	bitwise or
&&	Left	logical and
||	Left	logical or

Constructs that do not have any closing token, such as conditional statements and expressions, have lowest precedence so that, for example,

```
e1 ? e2 : e3 + e4
```

is parsed as follows:

```
e1 ? e2 : (e3 + e4)
```

and not as follows:

```
(e1 ? e2 : e3) + e4
```

9.4 Bit concatenation and selection

Bit concatenation and selection are expressed in the usual Verilog notation:

exprPrimary ::= *bitConcat* | *bitSelect*

bitConcat ::= `{` *expression* { `,` *expression* } `}`

bitSelect ::= *exprPrimary* `[` *expression* [`:` *expression*] `]`

In a bit concatenation, each component must have the type `bit[`m`:0]` ($m \geq 0$, width $m+1$). The result has type `bit[`n`:0]` where $n+1$ is the sum of the individual bit-widths ($n \geq 0$).

In a bit or part selection, the *exprPrimary* must have type `bit[`m`:0]` ($m \geq 0$), and the index *expressions* must have type `bit[31:0]`. With a single index (`[`e`]`), a single bit is selected, and the output is of type `bit[1:0]`. With two indexes (`[`e_1`:`e_2`]`), e_1 must be $\geq e_2$, and the indexes are inclusive, i.e., the bits selected go from the low index to the high index, inclusively. The selection has type `bit[`k`:0]` where $k+1$ is the width of the selection. Since the index expressions can in general be dynamic values (e.g., read out of a register), the type-checker may not be able to figure out this type, in which case it may be necessary to use a type assertion to tell the compiler the desired result type (see Section 9.10). The type specified by the type assertion need not agree with width specified by the indexes— the system will truncate from the left (most-significant bits) or pad with zeros to the left as necessary.

Example:


```
module mkBitConcatSelect ();

   Bit#(3) a = 3'b010;          //a = 010
   Bit#(7) b = 7'h5e;           //b = 1011110

   Bit#(10) abconcat = {a,b};  //  = 0101011110
   Bit#(4)  bselect  = b[6:3]; //  = 1011
endmodule
```

In BSV programs one will sometimes encounter the `Bit#(0)` type. One common idiomatic example is the type `Maybe#(Bit#(0))` (see the `Maybe#()` type in Section 7.3). Here, the type `Bit#(0)` is just used as a place holder, when all the information is being carried by the `Maybe` structure.

9.5 Begin-end expressions

A begin-end expression is like an "inline" function, i.e., it allows one to express a computation using local variables and multiple variable assignments and then finally to return a value. A begin-end expression is analogous to a "let block" commonly found in functional programming languages. It can be used in any context where an expression is required.

exprPrimary ::= *beginEndExpr*

beginEndExpr ::= `begin` [`:` *identifier*]
{ *beginEndExprStmt* }
expression
`end` [`:` *identifier*]

Optional identifier labels are currently used for documentation purposes only. The statements contained in the block can contain local variable declarations and all the other kinds of statements.

beginEndExprStmt ::= *varDecl* | *varAssign*
| *functionDef*
| *functionCall*
| *systemTaskCall*
| `(` *expression* `)`

Example:

```
int z;
z = (begin
       int x2 = x * x;    // x2 is local, x from surrounding scope
       int y2 = y * y;    // y2 is local, y from surrounding scope
       (x2 + y2);         // returned value (sum of squares)
     end);
```

9.6 Actions and action blocks

Any expression that is intended to act on the state of the circuit (at circuit execution time) is called an *action* and has type `Action`. The type `Action` is special, and cannot be redefined.

Primitive actions are provided as methods in interfaces to predefined objects (such as registers or arrays). For example, the predefined interface for registers includes a `._write()` method of type `Action`:


```
interface Reg#(type a);
    method Action _write (a x);
    method a      _read ();
endinterface: Reg
```

Section 8.4 describes special syntax for register reads and writes using non-blocking assignment so that most of the time one never needs to mention these methods explicitly.

The programmer can create new actions only by building on these primitives, or by using Verilog modules. Actions are combined by using action blocks:

exprPrimary ::= *actionBlock*

actionBlock ::= `action` [`:` *identifier*]
{ *actionStmt* }
`endaction` [`:` *identifier*]

actionStmt ::= *<action>If* | *<action>Case*
| *<action>BeginEndStmt*
| *regWrite*
| *varDecl* | *varAssign*
| *varDo* | *varDeclDo*
| *functionCall*
| *systemTaskCall*
| `(` *expression* `)`
| *actionBlock*

The action block can be labelled with an identifier, and the `endaction` keyword can optionally be labelled again with this identifier. Currently this is just for documentation purposes.

Example:

```
Action a;
a = (action
        x <= x+1;
        y <= z;
     endaction);
```

The Standard Prelude package defines the trivial action that does nothing:

```
Action noAction;
```

which is equivalent to the expression:

```
action
endaction
```

The `Action` type is actually a special case of the more general type `ActionValue`, described in the next section:

```
typedef ActionValue#(void) Action;
```


9.7 Actionvalue blocks

Note: this is an advanced topic and can be skipped on first reading.

Actionvalue blocks express the concept of performing an action and simultaneously returning a value. For example, the `pop()` method of a stack interface may both pop a value from a stack (the action) and return what was at the top of the stack (the value). `ActionValue` is a predefined abstract type:

```
ActionValue#(a)
```

The type parameter `a` represents the type of the returned value. The type `ActionValue` is special, and cannot be redefined.

Actionvalues are created using actionvalue blocks. The statements in the block contain the actions to be performed, and a `return` statement specifies the value to be returned.

exprPrimary ::= *actionValueBlock*

actionValueBlock ::= `actionvalue` [`:` *identifier*]
{ *actionValueStmt* }
`endactionvalue` [`:` *identifier*]

actionValueStmt ::= *<actionValue>If* | *<actionValue>Case*
| *<actionValue>BeginEndStmt*
| *regWrite*
| *varDecl* | *varAssign*
| *varDo* | *varDeclDo*
| *functionCall*
| *systemTaskCall*
| `(` *expression* `)`
| *returnStmt*

Given an actionvalue *av*, we use a special notation to perform the action and yield the value:

varDeclDo ::= *type identifier* `<-` *expression* `;`

varDo ::= *identifier* `<-` *expression* `;`

The first rule above declares the identifier, performs the actionvalue represented by the expression, and assigns the returned value to the identifier. The second rule is similar and just assumes the identifier has previously been declared.

Example. A stack:

```
interface IntStack;
    method Action             push (int x);
    method ActionValue#(int)  pop();
endinterface: IntStack




...
    IntStack s1;
...
    IntStack s2;
...
    action
        int x <- s1.pop;        -- A
        s2.push (x+1);          -- B
    endaction
```


In line A, we perform a pop action on stack `s1`, and the returned value is bound to `x`. If we wanted to discard the returned value, we could have omitted the "`x <-`" part. In line B, we perform a `push` action on `s2`.

Note the difference between this statement:

```
x <- s1.pop;
```

and this statement:

```
z =  s1.pop;
```

In the former, `x` must be of type `int`; the statement performs the pop action and `x` is bound to the returned value. In the latter, `z` must be of type `Method#(ActionValue#(int))` and `z` is simply bound to the method `s1.pop`. Later, we could say:

```
x <- z;
```

to perform the action and assign the returned value to `x`. Thus, the `=` notation simply assigns the left-hand side to the right-hand side. The `<-` notation, which is only used with actionvalue right-hand sides, performs the action and assigns the returned value to the left-hand side.

9.8 Function calls

Function calls are expressed in the usual notation, i.e., a function applied to its arguments, listed in parentheses:

exprPrimary ::= *functionCall*

functionCall ::= *exprPrimary* `(` *expression* { `,` *expression* } `)`

Note that the function position is specified as *exprPrimary*, of which *identifier* is just one special case. This is because in BSV functions are first-class objects, and so the function position can be an expression that evaluates to a function value. Function values and higher-order functions are described in Section 14.2.

Example:

```
module mkFunctionCalls ();

   function Bit#(4) everyOtherBit(Bit#(8) a);
      let result = {a[7], a[5], a[3], a[1]};
      return result;
   endfunction

   function Bool isEven(Bit#(8) b);
      return (b[0] == 0);
   endfunction

   Reg#(Bit#(8)) a     <- mkReg(0);
   Reg#(Bit#(4)) b     <- mkReg(0);

   rule doSomething (isEven(a)); // calling "isEven" in predicate: fire if a is an even number
      b <= everyOtherBit(a);     // calling a function in the rule body
   endrule
endmodule
```


9.9 Method calls

Method calls are expressed by selecting a method from an interface using dot notation, and then applying it to arguments, listed in parentheses:

exprPrimary ::= *methodCall*

methodCall ::= *exprPrimary* `.` *identifier* `(` *expression* { `,` *expression* } `)`

The *exprPrimary* is any expression that represents an interface, of which *identifier* is just one special case. This is because in BSV interfaces are first-class objects. The *identifier* must be a method in the supplied interface. Example:

```
// consider the following stack interface

interface StackIFC #(type data_t);
   method Action push(data_t data);   // an Action method with an argument
   method ActionValue#(data_t) pop(); // an actionvalue method
   method data_t first;               // a value method
endinterface

// when instantiated in a top module
module mkTop ();
  StackIFC#(int) stack   <- mkStack; // instantiating a stack module
  Reg#(int)      counter <- mkReg(0);// a counter register
  Reg#(int)      result  <- mkReg(0);// a result register

  rule pushdata;
     stack.push(counter); // calling an Action method
  endrule

  rule popdata;
     let x  <- stack.pop;  // calling an ActionValue method
     result <= x;
  endrule

  rule readValue;
     let temp_val = stack.first; // calling a value method
  endrule

  rule inc_counter;
     counter <= counter +1;
  endrule

endmodule
```

9.10 Static type assertions

We can assert that an expression must have a given type by using Verilog's "type cast" notation:

exprPrimary ::= *typeAssertion*

typeAssertion ::= *type* `'` *bitConcat*
| *type* `'` `(` *expression* `)`

bitConcat ::= `{` *expression* { `,` *expression* } `}`

In most cases type assertions are used optionally just for documentation purposes. Type assertions are necessary in a few places where the compiler cannot work out the type of the expression (an example is a bit-selection with run-time indexes).

In BSV although type assertions use Verilog's type cast notation, they are never used to change an expression's type. They are used either to supply a type that the compiler is unable to determine by itself, or for documentation (to make the type of an expression apparent to the reader of the source code).

9.11 Struct and union expressions

Section 7.3 describes how to define struct and union types. Section 8.1 describes how to declare variables of such types. Section 8.2 describes how to update variables of such types.

9.11.1 Struct expressions

To create a struct value, e.g., to assign it to a struct variable or to pass it an actual argument for a struct formal argument, we use the following notation:

exprPrimary ::= *structExpr*

structExpr ::= *Identifier* `{` *memberBind* { `,` *memberBind* } `}`

memberBind ::= *identifier* `:` *expression*

The leading *Identifier* is the type name to which the struct type was typedefed. Each *memberBind* specifies a member name (*identifier*) and the value (*expression*) it should be bound to. The members need not be listed in the same order as in the original typedef. If any member name is missing, that member's value is undefined.

Semantically, a *structExpr* creates a struct value, which can then be bound to a variable, passed as an argument, stored in a register, etc.

Example (using the processor example from Section 7.3):

```
typedef struct { Addr pc; RegFile rf; Memory mem; }  Proc;
...
Proc cpu;

cpu = Proc { pc : 0, rf : ... };
```

In this example, the `mem` field is undefined since it is omitted from the struct expression.

9.11.2 Struct member selection

A member of a struct value can be selected with dot notation.

exprPrimary ::= *exprPrimary* `.` *identifier*

Example (using the processor example from Section 7.3):

```
cpu.pc
```

Since the same member name can occur in multiple types, the compiler uses type information to resolve which member name you mean when you do a member selection. Occasionally, you may need to add a type assertion to help the compiler resolve this.

Update of struct variables is described in Section 8.2.

9.11.3 Tagged union expressions

To create a tagged union value, e.g., to assign it to a tagged union variable or to pass it an actual argument for a tagged union formal argument, we use the following notation:

exprPrimary	::=	*taggedUnionExpr*
taggedUnionExpr	::=	`tagged` *Identifier* `{` *memberBind* `{ , ` *memberBind* ` }` `}`
	|	`tagged` *Identifier exprPrimary*
memberBind	::=	*identifier* `:` *expression*

The leading *Identifier* is a member name of a union type, i.e., it specifies which variant of the union is being constructed.

The first form of *taggedUnionExpr* can be used when the corresponding member type is a struct. In this case, one directly lists the struct member bindings, enclosed in braces. Each *memberBind* specifies a member name (*identifier*) and the value (*expression*) it should be bound to. The members do not need to be listed in the same order as in the original struct definition. If any member name is missing, that member's value is undefined.

Otherwise, one can use the second form of *taggedUnionExpr*, which is the more general notation, where *exprPrimary* is directly an expression of the required member type.

Semantically, a *taggedUnionExpr* creates a tagged union value, which can then be bound to a variable, passed as an argument, stored in a register, etc.

Example (extending the previous one-hot example):

```
typedef union tagged { int Tagi; OneHot Tagoh; } U deriving (Bits);
...
  U  x; // these lines are (e.g.) in a module body.
  x = tagged Tagi 23;
  ...
  x = tagged Tagoh (encodeOneHot (23));
```

Example (extending the previous processor example):

```
typedef union tagged {
    bit  [4:0] Register;
    bit [21:0] Literal;
    struct {
        bit  [4:0] regAddr;
        bit  [4:0] regIndex;
    } Indexed;
} InstrOperand;
...
InstrOperand  orand;
...
orand = tagged Indexed { regAddr:3, regIndex:4 };
```

9.11.4 Tagged union member selection

A tagged union member can be selected with the usual dot notation. The selection is type safe, i.e., if the tagged union value does not have the tag corresponding to the member selection, an error is raised. Example:


```
InstrOperand  orand;
...
... orand.Indexed.regAddr ...
```

In this expression, if `orand` does not currently have the `Indexed` tag, an error is raised. Otherwise, the `regAddr` field of the contained struct is returned.

Selection of tagged union members is more often done with pattern matching, which is discussed in Section 10.

Update of tagged union variables is described in Section 8.2.

9.12 Interface expressions

Note: this is an advanced topic that may be skipped on first reading.

Section 5.2 described top-level interface declarations. Section 5.5 described definition of the interface offered by a module, by defining each of the methods in the interface, using *methodDef*s. That is the most common way of defining interfaces, but it is actually just a convenient alternative notation for the more general mechanism described in this section. In particular, method definitions in a module are a convenient alternative notation for a `return` statement that returns an interface value specified by an interface expression.

moduleStmt	::=	*returnStmt*
returnStmt	::=	`return` *expression* `;`
expression	::=	··· see other productions ···
	|	*exprPrimary*
exprPrimary	::=	*interfaceExpr*
interfaceExpr	::=	`interface` *Identifier* `;` { *interfaceStmt* } `endinterface` [`:` *Identifier*]
interfaceStmt	::=	*varDecl* | *varAssign*
	|	*methodDef*

An interface expression defines a value of an interface type. The *Identifier* must be an interface type in an existing interface type definition.

Example. Defining the interface for a stack of depth one (using a register for storage):

```
module mkStack#(type a) (Stack#(a));
  Reg#(Maybe#(a)) r;
  ...
  Stack#(a) stkIfc;
  stkIfc = interface Stack;
              method push (x) if (r matches tagged Invalid);
                  r <= tagged Valid x;
              endmethod: push

              method pop if (r matches tagged Valid .*);
                  r <= tagged Invalid
              endmethod: pop

              method top if (r matches tagged Valid .v);
                  return v
```



```
            endmethod: top
         endinterface: Stack
   return stkIfc;
endmodule: mkStack
```

The `Maybe` type is described in Section 7.3. Note that an interface expression looks similar to an interface declaration (Section 5.2) except that it does not list type parameters and it contains method definitions instead of method prototypes.

Interface values are first-class objects. For example, this makes it possible to write interface *transformers* that convert one form of interface into another. Example:

```
interface FIFO#(type a);           // define interface type FIFO
    method Action enq (a x);
    method Action deq;
    method a      first;
endinterface: FIFO

interface Get#(type a);            // define interface type Get
    ActionValue#(a) get;
endinterface: Get

// Function to transform a FIFO interface into a Get interface

function Get#(a) fifoToGet (FIFO#(a) f);
   return (interface Get
             method get();
                 actionValue
                     f.deq();
                     return f.first();
                 endactionValue
             endmethod: get
          endinterface);
endfunction: fifoToGet
```

9.12.1 Differences between interfaces and structs

Interfaces are similar to structs in the sense that both contain a set of named items—members in structs, methods in interfaces. Both are first-class values—structs are created with struct expressions, and interfaces are created with interface expressions. A named item is selected from both using the same notation—*struct.member* or *interface.method*.

However, they are different in the following ways:

- Structs cannot contain methods; interfaces can contain nothing but methods (and subinterfaces).
- Struct members can be updated; interface methods cannot.
- Struct members can be selected; interface methods cannot be selected, they can only be invoked (inside rules or other interface methods).
- Structs can be used in pattern matching; interfaces cannot.

9.13 Rule expressions

Note: This is an advanced topic that may be skipped on first reading.

Section 5.6 described definition of rules in a module. That is the most common way to define rules, but it is actually just a convenient alternative notation for the more general mechanism described in this section. In particular, rule definitions in a module are a convenient alternative notation for a call to the built-in `addRules()` function passing it an argument value of type `Rules`. Such a value is in general created using a rule expression. A rule expression has type `Rules` and consists of a collection of individual rule constructs.

exprPrimary ::= *rulesExpr*

rulesExpr ::= [*ruleAttribute*] [*docAttribute*]
`rules` [`:` *identifier*]
rulesStmt
`endrules` [`:` *identifier*]

rulesStmt ::= *varDecl* | *varAssign*
| *rule*

A rule expression is optionally preceded by a *ruleAttribute*; these are described in Section 13.3. A rule expression is a block, bracketed by `rules` and `endrules` keywords, and optionally labelled with an identifier. Currently the identifier is used only for documentation. The individual rule construct is described in Section 5.6.

Example. Executing a processor instruction:

```
rules
  Word instr = mem[pc];

  rule instrExec;
      case (instr) matches
         tagged Add { .r1, .r2, .r3 } : action
                                           pc <= pc+1;
                                           rf[r1] <= rf[r2] + rf[r3];
                                        endaction;
         tagged Jz {.r1, .r2}         : if (r1 == 0)
                                           action
                                             pc <= r2;
                                           endaction
                                        else
                                           noAction;
      endcase
  endrule
endrules
```

Example. Defining a counter:

```
// IfcCounter with read method
interface IfcCounter#(type t);
  method t      readCounter;
endinterface


// Definition of CounterType
typedef Bit#(16) CounterType;
```



```
// The next function returns the rule addOne
function Rules incReg(Reg#(CounterType) a);
  return( rules
    rule addOne;
       a <= a + 1;
    endrule
  endrules);
endfunction


// Module counter using IfcCounter interface
(* synthesize,
   RST_N = "reset_b",
   CLK   = "counter_clk",
   always_ready, always_enabled *)
module counter (IfcCounter#(CounterType));

   // Reg counter gets reset to 1 asynchronously with the RST signal
   Reg#(CounterType)   counter   <-  mkRegA(1);

   // Add incReg rule to increment the counter
   addRules(incReg(asReg(counter)));

   // Next rule resets the counter to 1 when it reaches its limit
   rule resetCounter (counter == '1);
   action
     counter <= 0;
   endaction
   endrule

   // Output the counters value
   method CounterType readCounter;
     return counter;
   endmethod

endmodule
```

10 Pattern matching

Pattern matching provides a visual and succinct notation to compare a value against structs, tagged unions and constants, and to access members of structs and tagged unions. Pattern matching can be used in `case` statements, `case` expressions, `if` statements, conditional expressions, rule conditions, and method conditions.

pattern	::=	. *identifier*	Pattern variable
	|	`.*`	Wildcard
	|	*constantPattern*	Constant
	|	*taggedUnionPattern*	Tagged union
	|	*structPattern*	Struct
	|	*tuplePattern*	Tuple
constantPattern	::=	*intLiteral*	
	|	*Identifier*	Enum label

taggedUnionPattern ::= `tagged` *Identifier* [*pattern*]

structPattern ::= `tagged` *Identifier* `{` *identifier* `:` *pattern* { `,` *identifier* `:` *pattern* } `}`

tuplePattern ::= `{` *pattern* { `,` *pattern* } `}`

A pattern is a nesting of tagged union and struct patterns with the leaves consisting of pattern variables, constant expressions, and the wildcard pattern `.*`.

In a pattern `.`x, the variable x is declared at that point as a pattern variable, and is bound to the corresponding component of the value being matched.

A constant pattern is an integer literal, or an enumeration label (such as `True` or `False`).

A tagged union pattern consists of the `tagged` keyword followed by an identifier which is a union member name. If that union member is not a `void` member, it must be followed by a pattern for that member.

In a struct pattern, the *Identifier* following the `tagged` keyword is the type name of the struct as given in its typedef declaration. Within the braces are listed, recursively, the member name and a pattern for each member of the struct. The members can be listed in any order, and members can be omitted.

A tuple pattern is enclosed in braces and lists, recursively, a pattern for each member of the tuple (tuples are described in Section 12.4).

A pattern always occurs in a context of known type because it is matched against an expression of known type. Recursively, its nested patterns also have known type. Thus a pattern can always be statically type-checked.

Each pattern introduces a new scope; the extent of this scope is described separately for each of the contexts in which pattern matching may be used. Each pattern variable is implicitly declared as a new variable within the pattern's scope. Its type is uniquely determined by its position in the pattern. Pattern variables must be unique in the pattern, i.e., the same pattern variable cannot be used in more than one position in a single pattern.

In pattern matching, the value V of an expression is matched against a pattern. Note that static type checking ensures that V and the pattern have the same type. The result of a pattern match is:

- A boolean value, `True`, if the pattern match succeeds, or `False`, if the pattern match fails.
- If the match succeeds, the pattern variables are bound to the corresponding members from V, using ordinary assignment.

Each pattern is matched using the following simple recursive rule:

- A pattern variable always succeeds (matches any value), and the variable is bound to that value (using ordinary procedural assignment).
- The wildcard pattern `.*` always succeeds.
- A constant pattern succeeds if V is equal to the value of the constant.
- A tagged union pattern succeeds if the value has the same tag and, recursively, if the nested pattern matches the member value of the tagged union.
- A struct or tuple pattern succeeds if, recursively, each of the nested member patterns matches the corresponding member values in V. In struct patterns with named members, the textual order of members does not matter, and members may be omitted. Omitted members are ignored.

Conceptually, if the value V is seen as a flattened vector of bits, the pattern specifies the following: which bits to match, what values they should be matched with and, if the match is successful, which bits to extract and bind to the pattern identifiers.

10.1 Case statements with pattern matching

Case statements can occur in various contexts, such as in modules, function bodies, action and actionValue blocks, and so on. Ordinary case statements are described in Section 8.6. Here we describe pattern-matching case statements.

```
<ctxt>Case          ::= case ( expression ) matches
                            { <ctxt>CasePatItem }
                            [ <ctxt>DefaultItem ]
                        endcase

<ctxt>CasePatItem   ::= pattern [ &&& expression ] : <ctxt>Stmt

<ctxt>DefaultItem   ::= default [ : ] <ctxt>Stmt
```

The keyword `matches` after the main *expression* (following the `case` keyword) signals that this is a pattern-matching case statement instead of an ordinary case statement.

Each case item contains a left-hand side and a right-hand side, separated by a colon. The left-hand side contains a pattern and an optional filter (`&&&` followed by a boolean expression). The right-hand side is a statement. The pattern variables in a pattern may be used in the corresponding filter and right-hand side. The case items may optionally be followed, finally, by a default item (the colon after the `default` keyword is optional).

The value of the main *expression* (following the `case` keyword) is matched against each case item, in the order given, until an item is selected. A case item is selected if and only if the value matches the pattern and the filter (if present) evaluates to `True`. Note that there is a left-to-right sequentiality in each item— the filter is evaluated only if the pattern match succeeds. This is because the filter expression may use pattern variables that are meaningful only if the pattern match succeeds. If none of the case items matches, and a default item is present, then the default item is selected.

If a case item (or the default item) is selected, the right-hand side statement is executed. Note that the right-hand side statement may use pattern variables bound on the left hand side. If none of the case items succeed, and there is no default item, no statement is executed.

Example (uses the `Maybe` type definition of Section 7.3):

```
case (f(a)) matches
    tagged Valid .x : return x;
    tagged Invalid : return 0;
endcase
```

First, the expression `f(a)` is evaluated. In the first arm, the value is checked to see if it has the form `tagged Valid .x`, in which case the pattern variable `x` is assigned the component value. If so, then the case arm succeeds and we execute `return x`. Otherwise, we fall through to the second case arm, which must match since it is the only other possibility, and we return 0.

Example:

```
typedef union tagged {
    bit  [4:0] Register;
    bit [21:0] Literal;
    struct {
        bit  [4:0] regAddr;
        bit  [4:0] regIndex;
    } Indexed;
} InstrOperand;
...
```



```
InstrOperand  orand;
...
    case (orand) matches
       tagged Register .r          : x  = rf [r];
       tagged Literal  .n          : x  = n;
       tagged Indexed { .ra, .ri } : x = mem[ra+ri];
    endcase
```

10.2 Case expressions with pattern matching

caseExpr	::=	`case (` *expression* `) matches` { *caseExprItem* } `endcase`
caseExprItem	::=	*pattern* [`&&&` *expression*] `:` *expression*
	|	`default` [`:`] *expression*

Case expressions with pattern matching are similar to case statements with pattern matching. In fact, the process of selecting a case item is identical, i.e., the main expression is evaluated and matched against each case item in sequence until one is selected. Case expressions can occur in any expression context, and the right-hand side of each case item is an expression. The whole case expression returns a value, which is the value of the right-hand side expression of the selected item. It is an error if no case item is selected and there is no default item.

In contrast, case statements can only occur in statement contexts, and the right-hand side of each case arm is a statement that is executed for side effect. The difference between case statements and case expressions is analogous to the difference between if statements and conditional expressions.

Example. Rules and rule composition for Pipeline FIFO using case statements with pattern matching.

```
package PipelineFIFO;

import RWire::*;
import FIFO::*;

module mkPipelineFIFO (FIFO#(a))
   provisos (Bits#(a,sa));

   // STATE ----------------

   Reg#(Maybe#(a))   taggedReg <- mkReg (tagged Invalid); // the FIFO
   RWire#(a)         rw_enq    <- mkRWire;                // enq method signal
   RWire#(Bit#(0))   rw_deq    <- mkRWire;                // deq method signal

   // RULES and RULE COMPOSITION ----------------

   Maybe#(a) taggedReg_post_deq = case (rw_deq.wget) matches
                                    tagged Invalid  : return taggedReg;
                                    tagged Valid .x : return tagged Invalid;
                                  endcase;

   Maybe#(a) taggedReg_post_enq = case (rw_enq.wget) matches
                                    tagged Invalid  : return taggedReg_post_deq;
                                    tagged Valid .v : return tagged Valid v;
```



```
                                endcase;

rule update_final (isValid(rw_enq.wget) || isValid(rw_deq.wget));
    taggedReg <= taggedReg_post_enq;
endrule
```

10.3 Pattern matching in if statements and other contexts

If statements are described in Section 8.6. As the grammar shows, the predicate (*condPredicate*) can be a series of pattern matches and expressions, separated by `&&&`. Example:

```
if ( e1 matches p1   &&&   e2   &&&   e3 matches p3 )
  stmt1
else
  stmt2
```

Here, the value of e_1 is matched against the pattern p_1; if it succeeds, the expression e_2 is evaluated; if it is true, the value of e_3 is matched against the pattern p_3; if it succeeds, *stmt1* is executed, otherwise *stmt2* is executed. The sequential order is important, because e_2 and e_3 may use pattern variables bound in p_1, and *stmt1* may use pattern variables bound in p_1 and p_3, and pattern variables are only meaningful if the pattern matches. Of course, *stmt2* cannot use any of the pattern variables, because none of them may be meaningful when it is executed.

In general the *condPredicate* can be a series of terms, where each term is either a pattern match or a filter expression (they do not have to alternate). These are executed sequentially from left to right, and the *condPredicate* succeeds only if all of them do. In each pattern match e `matches` p, the value of the expression e is matched against the pattern p and, if successful, the pattern variables are bound appropriately and are available for the remaining terms. Filter expressions must be boolean expressions, and succeed if they evaluate to `True`. If the whole *condPredicate* succeeds, the bound pattern variables are available in the corresponding "consequent" arm of the construct.

The following contexts also permit a *condPredicate cp* with pattern matching:

- Conditional expressions (Section 9.2):

  cp `?` e_2 `:` e_3

  The pattern variables from *cp* are available in e_2 but not in e_3.

- Conditions of rules (Sections 5.6 and 9.13):

  ```
  rule r (cp);
    ... rule body ...
  endrule
  ```

  The pattern variables from *cp* are available in the rule body.

- Conditions of methods (Sections 5.5 and 9.12):

  ```
  method t f (...) if (cp);
    ... method body ...
  endmethod
  ```

  The pattern variables from *cp* are available in the method body.

Example. Continuing the Pipeline FIFO example from the previous section (10.2).


```
 // INTERFACE ----------------

   method Action enq(v) if (taggedReg_post_deq  matches  tagged Invalid);
      rw_enq.wset(v);
   endmethod

   method Action deq() if (taggedReg matches tagged Valid .v);
      rw_deq.wset(?);
   endmethod

   method first() if (taggedReg matches tagged Valid .v);
      return v;
   endmethod

   method Action clear();
      taggedReg <= tagged Invalid;
   endmethod

endmodule: mkPipelineFIFO

endpackage:  PipelineFIFO
```

10.4 Pattern matching assignment statements

Pattern matching can be used in variable assignments for convenient access to the components of a struct or union value.

varAssign ::= `match` *pattern* `=` *expression* `;`

The pattern variables in the left-hand side pattern are declared at this point and their scope extends to subsequent statements in the same statement sequence. The types of the pattern variables are determined by their position in the pattern.

The left-hand side pattern is matched against the value of the right-hand side expression. On a successful match, the pattern variables are assigned the corresponding components in the value. It is an error if the pattern does not match (this can only occur if there are constants in the pattern that do not match, or if the pattern and value contain tagged unions with different tags.

Example:

```
// Type Frame
typedef struct{ Bit#(32)  payload;
              } Frame deriving(Bits);

...
   // Get the data from Rx1 and assign it to variable a
   match tagged Frame {payload: .a}  = rx1.getData;

   // Get the data from Rx2 and assign it to variable b
   match tagged Frame {payload: .b}  = rx2.getData;
```

11 Finite state machines

BSV contains a powerful and convenient notation for expressing finite state machines (FSMs). FSMs are essentially well-structured processes involving sequencing, parallelism, conditions and loops, with

a precise compositional model of time. In principle, FSMs can be coded with rules, which are strictly more powerful, but the FSM sublanguage herein provides a succinct notation for FSM structures and automates all the generation and management of the actual FSM state. In fact, the BSV compiler translates all the constructs described here internally into rules. In particular, the primitive statements in these FSMs are standard actions (Section 9.6), obeying all the scheduling semantics of actions (Section 6.2).

First, one uses the `Stmt` sublanguage, described in Section C.3.1 to compose the actions of an FSM using sequential, parallel, conditional and looping structures. This sublanguage is within the *expression* syntactic category, i.e., a term in the sublanguage is an expression whose value is of type `Stmt`. This value can be bound to identifiers, passed as arguments and results of functions, held in static data structures, etc., like any other value. Finally, the FSM can be instantiated into hardware, multiple times if desired, by passing the `Stmt` value to the module constructor `mkFSM`. The resulting module interface has type `FSM`, which has methods to start the FSM and to wait until it completes.

In order to use this sublanguage, it is necessary to import the `StmtFSM` package, which is described in more detail in Section C.3.1.

12 Important primitives

These primitives are available via the Standard Prelude package and other standard libraries. See also Appendix C more useful libraries.

12.1 The types `bit` and `Bit`

The type `bit[`m`:0]` and its synonym `Bit#(Mplus1)` represents bit-vectors of width $m+1$, provided the type `Mplus1` has been suitably defined. The lower (lsb) index must be zero. Example:

```
bit [15:0] zero;
zero = 0

typedef bit [50:0] BurroughsWord;
```

Syntax for bit concatenation and selection is described in Section 9.4.

There is also a useful function, `split`, to split a bit-vector into two sub-vectors:

```
function Tuple2#(Bit#(m), Bit#(n)) split (Bit#(mn) xy)
   provisos (Add#(m,n,mn));
```

It takes a bit-vector of size mn and returns a 2-tuple (a pair, see Section 12.4) of bit-vectors of size m and n, respectively. The proviso expresses the size constraints using the built-in `Add` type class.

The function `split` is polymorphic, i.e, m and n may be different in different applications of the function, but each use is fully type-checked statically, i.e., the compiler verifies the proviso, performing any calculations necessary to do so.

12.1.1 Bit-width compatibility

BSV is currently very strict about bit-width compatibility compared to Verilog and SystemVerilog, in order to reduce the possibility of unintentional errors. In BSV, the types `bit[`m`:0]` and `bit[`n`:0]` are compatible only if $m = n$. For example, an attempt to assign from one type to the other, when $m \neq n$, will be reported by the compiler as a type-checking error—there is no automatic padding or

truncation. The Standard Prelude package (see Section B) contains functions such as `zeroExtend()` and `truncate()`, which may be used explicitly to extend or truncate to a required bit-width. These functions, being overloaded over all bit widths, are convenient to use, i.e., you do not have to constantly calculate the amount by which to extend or truncate; the type checker will do it for you.

12.2 UInt, Int, int and Integer

The types `UInt#(`n`)` and `Int#(`n`)`, respectively, represent unsigned and signed integer data types of width n bits. These types have all the operations from the type classes (overloading groups) `Bits`, `Literal`, `Eq`, `Arith`, `Ord`, `Bounded`, `Bitwise`, `BitReduction`, and `BitExtend`. (See Appendix B for the specifications of these type classes and their associated operations.)

Note that the types `UInt` and `Int` are not really primitive; they are defined completely in BSV.

The type `int` is just a synonym for `Bit#(32)` (i.e., it is typedefed thus).

The type `Integer` represents unbounded integers. Because they are unbounded, they are only used to represent static values used during static elaboration. The overloaded function `fromInteger` allows conversion from an `Integer` to various other types.

12.3 String

The type `String` is defined in the Standard Prelude package. Strings are mostly used in system tasks (such as `$display`). Strings can be concatenated using the `strConcat` function, and they can be tested for equality and inequality using the `==` and `!=` operators. String literals, written in double-quotes, are described in Section 2.5.

12.4 Tuples

It is frequently necessary to group a small number of values together, e.g., when returning multiple results from a function. Of course, one could define a special struct type for this purpose, but BSV predefines a number of structs called *tuples* that are convenient:

```
typedef struct {a _1; b _2;} Tuple2#(type a, type b) deriving (Bits,Eq,Bounded);
typedef      ...             Tuple3#(type a, type b, type c) ...;
typedef      ...             ...                ...;
typedef      ...             Tuple7#(type a, ..., type g) ...;
```

Values of these types can be created by applying a predefined family of constructor functions:

```
tuple2 (e1, e2)
tuple3 (e1, e2, e3)
...
tuple7 (e1, e2, e3, ..., e7)
```

where the expressions `e`J evaluate to the component values of the tuples.

Components of tuples can be extracted using a predefined family of selector functions:

```
tpl_1 (e)
tpl_2 (e)
...
tpl_7 (e)
```


where the expression `e` evaluates to tuple value. Of course, only the first two are applicable to `Tuple2` types, only the first three are applicable to `Tuple3` types, and so on.

In using a tuple component selector, it is sometimes necessary to use a static type assertion to help the compiler work out the type of the result. Example:

```
UInt#(6)'(tpl_2 (e))
```

Tuple components are more conveniently selected using pattern matching. Example:

```
Tuple2#(int, Bool)  xy;
...
   case (xy) matches
       { .x, .y } : ... use x and y ...
   endcase
```

12.5 Registers

The most elementary module available in BSV is the register, which has a `Reg` interface. Registers are instantiated using the `mkReg` module, whose single parameter is the initial value of the register. Registers can also be instantiated using the `mkRegU` module, which takes no parameters (don't-care initial value). The `Reg` interface type and the module types are shown below.

```
interface Reg#(type a);
    Action  _write (a x);
    a       _read;
endinterface: Reg

module mkReg#(a initVal) (Reg#(a))
   provisos
       (Bits#(a, sa));

module mkRegU (Reg#(a))
   provisos
       (Bits#(a, sa));
```

Registers are polymorphic, i.e., in principle they can hold a value of any type but, of course, ultimately registers store bits. Thus, the provisos on the modules indicate that the type must be in the `Bits` type class (overloading group), i.e., the operations `pack()` and `unpack()` must be defined on this type to convert into to bits and back.

Section 8.4 describes special notation whereby one rarely uses the `_write()` and `_read` methods explicitly. Instead, one more commonly uses the traditional non-blocking assignment notation for writes and, for reads, one just mentions the register interface in an expression.

Since mentioning the register interface in an expression is shorthand for applying the `_read` method, BSV also provides a notation for overriding this implicit read, producing an expression representing the register interface itself:

```
asReg (r)
```


12.6 FIFOs

Package `FIFO` defines several useful interfaces and modules for FIFOs:

```
interface FIFO#(type a);
    Action  enq (a x);
    Action  deq;
    a       first;
    Action  clear;
endinterface: FIFO

module mkFIFO (FIFO#(a))
    provisos (Bits#(a, as));

module mkSizedFIFO#(Integer depth) (FIFO#(a))
    provisos (Bits#(a, as));
```

The `FIFO` interface type is polymorphic, i.e., the FIFO contents can be of any type a. However, since FIFOs ultimately store bits, the content type a must be in the `Bits` type class (overloading group); this is specified in the provisos for the modules.

The module `mkFIFO` leaves the capacity of the FIFO unspecified (the number of entries in the FIFO before it becomes full).

The module `mkSizedFIFO` takes the desired capacity of the FIFO explicitly as a parameter.

Of course, when compiled, `mkFIFO` will pick a particular capacity, but for formal verification purposes it is useful to leave this undetermined. It is often useful to be able to prove the correctness of a design without relying on the capacity of the FIFO. Then the choice of FIFO depth can only affect circuit performance (speed, area) and cannot affect functional correctness, so it enables one to separate the questions of correctness and "performance tuning." Thus, it is good design practice initially to use `mkFIFO` and address all functional correctness questions. Then, if performance tuning is necessary, it can be replaced with `mkSizedFIFO`.

12.7 FIFOFs

Package `FIFOF` defines several useful interfaces and modules for FIFOs. The `FIFOF` interface is like `FIFO`, but it also has methods to test whether the FIFO is full or empty:

```
interface FIFOF a =
    Action  enq (a x);
    Action  deq;
    a       first;
    Action  clear;
    Bool    notFull;
    Bool    notEmpty;
endinterface

module mkFIFOF (FIFOF#(a))
   provisos (Bits#(a, as));

module mkSizedFIFOF#(Integer depth) (FIFOF#(a))
   provisos (Bits#(a, as));
```

The module `mkFIFOF` leaves the capacity of the FIFO unspecified (the number of entries in the FIFO before it becomes full). The module `mkSizedFIFOF` takes the desired capacity of the FIFO as an argument.

12.8 System tasks and functions

BSV supports a number of Verilog's system tasks and functions.

12.8.1 System tasks for displaying information

systemTaskCall ::= *displayTaskName* `(` [*expression* [`,` *expression*]] `)`

displayTaskName ::= `$display` | `$displayb` | `$displayo` | `$displayh`
| `$write` | `$writeb` | `$writeo` | `$writeh`

These system task calls are conceptually function calls of type `Action`, and can be used in any context where an action is expected.

The only difference between the `$display` family and the `$write` family is that members of the former always output a newline after displaying the arguments, whereas members of the latter do not.

The only difference between the ordinary, `b`, `o` and `h` variants of each family is the format in which numeric expressions are displayed if there is no explicit format specifier. The ordinary `$display` and `$write` will output, by default, in decimal format, whereas the `b`, `o` and `h` variants will output in binary, octal and hexadecimal formats, respectively.

There can be any number of argument expressions between the parentheses. The arguments are displayed in the order given. If there are no arguments, `$display` just outputs a newline, whereas `$write` outputs nothing.

The argument expressions can be of type `String`, `Bit#(n)` (i.e., of type `bit[n-1:0]`), `Integer`, or any type that is a member of the overloading group `Bits`. Members of `Bits` will display their packed representation. The output will be interpreted as a signed number for the types `Integer` and `Int#(n)`. Arguments can also be literals. `Integer`s and literals are limited to 32 bits.

Arguments of type `String` are interpreted as they are displayed. The characters in the string are output literally, except for certain special character sequences beginning with a `%` character, which are interpreted as format-specifiers for subsequent arguments. The following format specifiers are supported[6]:

`%d`	Output a number in decimal format
`%b`	Output a number in binary format
`%o`	Output a number in octal format
`%h`	Output a number in hexadecimal format
`%c`	Output a character with given ASCII code
`%s`	Output a string (argument must be a string)
`%t`	Output a number in time format

Actionvalues (see Section 9.7) whose returned type is displayable can also be directly displayed. This is done by performing the associated action (as part of the action invoking `$display`) and displaying the returned value.

[6] Displayed strings are passed through the compiler unchanged, so other format specifiers may be supported by your Verilog simulator. Only the format specifiers above are supported by Bluespec's C-based simulator.

12.8.2 System tasks for stopping simulation

systemTaskCall ::= `$finish` [`(` *expression* `)`]
| `$stop` [`(` *expression* `)`]

These system task calls are conceptually function calls of type `Action`, and can be used in any context where an action is expected.

The `$finish` task causes simulation to stop immediately and exit back to the operating system. The `$stop` task causes simulation to suspend immediately and enter an interactive mode. The optional argument expressions can be 0, 1 or 2, and control the verbosity of the diagnostic messages printed by the simulator. the default (if there is no argument expression) is 1.

12.8.3 System tasks for VCD dumping

systemTaskCall ::= `$dumpvars` | `$dumpon` | `$dumpoff`

These system task calls are conceptually function calls of type `Action`, and can be used in any context where an action is expected.

A call to `$dumpvars` starts dumping the changes of all the state elements in the design to the VCD file. BSV's `$dumpvars` does not currently support arguments that control the specific module instances or levels of hierarchy that are dumped.

Subsequently, a call to `$dumpoff` stops dumping, and a call to `$dumpon` resumes dumping.

12.8.4 System functions returning the current time

systemFunctionCall ::= `$time` | `$stime`

These system function calls are conceptually of `ActionValue` type (see Section 9.7), and can be used anywhere an `ActionValue` is expected. The time returned is the time when the associated action was performed.

The `$time` function returns a 64-bit integer (specifically, of type `Bit#(64)`) representing time, scaled to the timescale unit of the module that invoked it.

The `$stime` function returns a 32-bit integer (specifically, of type `Bit#(32)`) representing time, scaled to the timescale unit of the module that invoked it. If the actual simulation time does not fit in 32 bits, the lower-order 32 bits are returned.

12.8.5 System functions for testing command line input

Information for use in simulation can be provided on the command line. This information is specified via optional arguments in the command used to invoke the simulator. These arguments are distinguished from other simulator arguments by starting with a plus (`+`) character and are therefore known as *plusargs*. Following the plus is a string which can be examined during simulation via system functions.

systemTaskCall ::= `$test$plusargs` `(` *expression* `)`

The `$test$plusargs` system function call is conceptually of `ActionValue` type (see Section 9.7), and can be used anywhere an `ActionValue` is expected. An argument of type `String` is expected and a boolean value is returned indicating whether the provided string matches the beginning of any plusarg from the command line.

13 Guiding the compiler with attributes

This section describes how to guide the compiler in some of its decisions using attributes, which are expressed with Verilog's *attribute* syntax.

attributeInstance ::= `(*` *attrSpec* { `,` *attrSpec* } `*)`

attrSpec ::= *attrName* [`=` *expression*]

attrName ::= *identifier* |*Identifier*

Multiple attributes can be written together on a single line

```
(* synthesize, always_ready = "read, subifc.enq" *)
```

Or attributes can be written on multiple lines

```
(* synthesize *)
(* always_ready = "read, subifc.enq" *)
```

Attributes can be associated with a number of different language constructs such as module, interface, and function definitions. A given attribute declaration is applied to the first attribute construct that follows the declaration. The following table summarizes the available attributes and the language constructs to which they may be applied. Each attribute is described in more detail in the referenced section.

Attribute Name	Section	Modules	Methods	Interfaces	Functions	Rules
synthesize	13.1.1	√				
noinline	13.1.2				√	
RST_N=	13.1.3	√				
CLK=	13.1.3	√				
always_ready	13.1.4, 13.2.4	√	√	√		
always_enabled	13.1.4, 13.2.4	√	√	√		
ready=	13.2.1		√			
enable=	13.2.1		√			
result=	13.2.2		√			
prefix=	13.2.3		√			
port=	13.2.3		√			
fire_when_enabled	13.3.1					√
no_implicit_conditions	13.3.2					√
descending_urgency	13.3.3	√				√
preempts	13.3.4	√				√
doc=	13.4	√				√

13.1 Verilog module generation attributes

In addition to compiler flags on the command line, it is possible to guide the compiler with attributes that are included in the BSV source code.

modgenAttribute ::= *attributeInstance*

The attributes `synthesize` and `noinline` control code generation for modules and functions, respectively. The remaining attributes, `RST_N=`, `CLK=`, `always_enabled` and `always_ready` are dependent on the `synthesize` attribute. If the `synthesize` attribute is not specified for the module, they are ignored.

13.1.1 Module generation attribute `synthesize`

When the compiler is directed to generate Verilog or Bluesim code for a BSV module, by default it tries to integrate all definitions into one big module. The `synthesize` attribute marks a module for code generation and ensures that, when generated, instantiations of the module are not flattened but instead remain as references to a separate module definition. Modules that are annotated with the `synthesize` attribute are said to be *synthesized* modules. The BSV hierarchy boundaries associated with synthesized modules are maintained during code generation. Not all BSV modules can be synthesized modules (*i.e.*,can maintain a module boundary during code generation). Section 5.8 describes in more detail which modules are synthesizable.

13.1.2 Module generation attribute `noinline`

The `noinline` attribute is applied to functions. It tells the compiler not to inline the function, but to generate a separate module for it, whose single instantiation is shared among all the callers of the function. The function has the same type restrictions as interface methods that are involved in code generation. The `noinline` attribute can only be applied to functions that are defined at the top level. Example:

```
(* noinline *)
function Bit#(LogK) popCK(Bit#(K) x);
  return (popCountTable(x));
endfunction: popCK
```

13.1.3 Generated port renaming attributes `RST_N=` and `CLK=`

The generated port renaming attributes allow renaming of the default ports for the low reset and clock signals. The attributes are associated with a module and are only applied when the `synthesize` attribute is specified for the module.

By default, the BSV compiler names the asserted low reset signal associated with each module `RST_N`. The `RST_N=` attribute is used to specify an alternate name for the reset signal.

By default, the BSV compiler names the clock signal associated with each module `CLK`. The `CLK=` attribute is used to specify an alternate name for the clock signal.

Example

```
( * synthesize, CLK = "clock", RST_N = "reset" * )
```

13.1.4 Port protocol attributes `always_enabled` and `always_ready`

The port protocol attributes `always_enabled` and `always_ready` allow the removal of unnecessary ports. In both cases the compiler will verify that the attribute is correct.

`always_enabled` specifies that there should be no enable signal generated for the associated interface methods. The methods will be executed on every clock cycle, and the compiler verifies that the caller does this.

`always_ready` specifies that no ready signals should be generated. The compiler verifies that the associated interface methods are permanently ready. `always_enabled` implies `always_ready`.

These attributes can be applied either to an entire interface or to an individual interface method. If applied to the entire interface, the attributes affect all the methods included in that interface.

The attributes are applied when the interface is implemented within a module, not at the declaration of the interface. Example:


```
interface ILookup;                      //the definition of the interface
    interface Fifo#(int) subifc;
    method Action read ();
endinterface: ILookup

(* always_ready = "read, subifc.enq" * )//the attribute is applied when the
module mkServer (ILookup);              //interface is implemented within
   ...                                  //a module.
endmodule: mkServer
```

In the above example, note that only the subifc.enq method of the subifc interface is `always_ready`. Other methods of the interface, such as subifc.deq, are not `always_ready`.

If every method of the interface is `always_ready` or `always_enabled`, individual methods don't have to be specified:

```
(* always_enabled *)
module mkServer (ILookup);
```

13.2 Interface attributes

Interface attributes are used to express protocol and naming requirements for generated Verilog interfaces.

interfaceAttribute ::= *attributeInstance*

Interface attributes are not considered until the generation of the Verilog module which uses the interface.

There is a direct correlation between interfaces in Bluespec and ports in the generated Verilog. Several attributes can be annotated in interfaces to directly guide the naming and the protocols of the generated Verilog ports.

Bluespec uses a simple Ready-Enable micro-protocol for each method within the module's interface. Each method contains both a output Ready (RDY) signal and an input Enable (EN) signal in addition to any needed directional data lines. When a method can be safely called it asserts its RDY signal. When an external caller sees the RDY signal it may then call (in the same cycle) the method by asserting the method's EN signal and providing any required data.

Generated Verilog ports names are based the method name and argument names, with some standard prefixes. This ActionValue method generates the following ports

```
method ActionValue#( type_out )  method1 ( type_in data_in ) ;
```

`RDY_method1`	Output ready signal of the protocol
`EN_method1`	Input signal for Action and Action Value methods
`method1`	Output signal of ActionValue and Value methods
`method1_data_in`	Input signal for method arguments

Interface attributes allow control over the naming and protocols of individual methods or entire interfaces.

13.2.1 Interface attributes `ready=` and `enable=`

Ready and enable ports use `RDY_` and `EN_` as their default prefix to the method names. Using attributes `ready = "string"` and `enable = "string"` will stop the prefix annotation and use the string as listed. These attributes may be associated with method declarations (*methodProto*) only, see Section 5.2.

In the above example, the following attribute would replace the `RDY_method1` with `avMethodIsReady` and `EN_method1` with `GO`.

```
(* ready = "avMethodIsReady", enable = "GO" *)
```

Note that the `ready=` attribute is ignored if the method or module is annotated as `always_ready` or `always_enabled`, while the `enable=` attribute is ignored for value methods, and those annotated as `always_enabled`.

13.2.2 Interface attribute `result=`

By default the output port for value methods and `ActionValue` methods use the method name. Using the attribute `result = "string"` causes the output to be renamed to `"string"`. This is useful when the desired port names must begin a upper case letter, which is not valid for a method name. These attributes may be associated with method declarations (*methodProto*) only, see Section 5.2.

In the above example, the following attribute would replace the `method1` port with `OUT`.

```
(* result = "OUT" *)
```

Note that the `result=` attribute is ignored if the method is an `Action` method which does not return a result.

13.2.3 Interface attributes `prefix=` and `port=`

By default input ports for methods are named as `methodName_argumentName` Using the combination of attributes `prefix = "string"` and `port = "string"` can cause any strings to be generated for the Verilog ports. The `prefix=` attribute replaces the `methodName` prefix for the generated and name. The prefix string may be empty, in which case the joining underscore is not added. The `port=` attribute is associated with each formal port declaration of the method, and replaces `argumentName` in the generated Verilog port name.

The `prefix=` attribute may be associated with method declarations (*methodProto*) or sub-interface declarations (*subinterfaceDecl*). The `port=` attribute may be associate with each method prototype argument in the interface declaration (*methodProtoFormal*), see Section 5.2.

In the above example, the following attribute would replace the `method1_data_in` port with `IN_DATA`.

```
(* prefix = "" *)
   method ActionValue#( type_out )
        method1( (* port="IN_DATA" *) type_in data_in ) ;
```

Note that the `prefix=` attribute is ignored if the method does not have any arguments.

The `prefix=` attribute may also be used on sub-interface declarations to aid the renaming of interface hierarchies. By default, interface hierarchies are named by prefixing the sub-interface name to names of the methods within that interface. (See Section 5.2.1.) Using the `prefix` attribute, will replace the sub-interface name. See the example later in this section.

13.2.4 Interface attributes `always_ready` and `always_enabled`

The port protocol attributes `always_enabled` and `always_ready` allow the removal of unnecessary ports. In all cases the compiler verifies that the attribute and design are correctly applied.

`always_enabled` specifies that there should be no enable signal generated for the associated interface methods. The methods will be executed on every clock cycle, and the compiler verifies that the caller does this.

`always_ready` specifies that no ready signals should be generated. The compiler verifies that the associated interface methods are permanently ready. `always_enabled` implies `always_ready`.

The `always_ready` and `always_enabled` attributes can be associate with the method declarations (*methodProto*), the sub-interface declarations (*subinterfaceDecl*), or the interface declaration (*interfaceDecl*) itself.

13.2.5 Interface attributes example

```
(* always_ready *)                  // all methods in this and all sub-interface
                                    // have this property
                                    // always_enabled is also allowed here
interface ILookup;
    (* prefix = "" *)               // subifc_ will not be used in naming
                                    // always_enabled and always_ready are allowed.
    interface Fifo#(int) subifc;

    (* enable = "GOread" *)         // EN_read becomes GOread
    method Action read ();


    (* always_enabled *)            // always_enabled and always_ready
                                    // are allowed on any individual method
    (* result = "CHECKOK" *)       // output checkData becomes CHECKOK
    (* prefix = "" *)               // checkData_datain1 becomes DIN1
                                    // checkData_datain2 becomes DIN2
    method ActionValue#(Bool)  checkData ( (* port= "DIN1" *) int datain1
                                           (* port= "DIN2" *) int datain2 ) ;

endinterface: ILookup
```

13.3 Scheduling attributes

Scheduling attributes are used to express certain performance requirements. When the compiler maps rules into clocks, as described in Section 6.2.2, scheduling attributes guide or constrain its choices, in order to produce a schedule that will meet performance goals.

Scheduling attributes are most often attached to rules or to rule expressions.

ruleAttribute ::= *attributeInstance*

Scheduling attributes are not considered until the generation of Verilog or C code for the module that includes them.

13.3.1 Scheduling attribute `fire_when_enabled`

The `fire_when_enabled` scheduling attribute immediately precedes a rule (just before the `rule` keyword) and governs the rule.

It asserts that this rule must fire whenever its predicate and its implicit conditions are true, *i.e.,* when they are true, there are no scheduling conflicts with other rules that will prevent it from firing. This is statically verified by the compiler, and it will report an error if necessary.

Example. Using `fire_when_enabled` to ensure the counter is reset.

```
// IfcCounter with read method
interface IfcCounter#(type t);
  method t      readCounter;
endinterface

// Definition of CounterType
typedef Bit#(16) CounterType;

// Module counter using IfcCounter interface. It never contains 0.

(* synthesize,
   RST_N = "reset_b",
   CLK = "counter_clk",
   always_ready = "readCounter",
   always_enabled= "readCounter" *)

module counter (IfcCounter#(CounterType));
   // Reg counter gets reset to 1 asynchronously with the RST signal
   Reg#(CounterType)   counter   <-  mkRegA(1);

   //    Next rule resets the counter to 1 when it reaches its limit.
   //    The attribute fire_when_enabled will check that this rule will fire
   //    if counter == '1
   (* fire_when_enabled *)
   rule resetCounter (counter == '1);
   action
     counter <= 1;
   endaction
   endrule

   //  Next rule updates the counter.
   rule updateCounter;
   action
     counter <= counter + 1;
   endaction
   endrule

   // Method to output the counter's value
   method CounterType readCounter;
     return counter;
   endmethod
endmodule
```

Rule `resetCounter` conflicts with rule `updateCounter` because both try to modify the counter register when it contains all its bits set to one. The rule `updateCounter` rule will obtain more

urgency, meaning that if the predicate of `resetCounter` is met, only the rule `updateCounter` will fire. The assertion `fire_when_enabled` will be violated and the compiler will produce an error message. Please note that without the assertion `fire_when_enabled` the compilation process will be correct and the designer will be only warned about the scheduling of the conflicting rules.

13.3.2 Scheduling attribute `no_implicit_conditions`

The `no_implicit_conditions` scheduling attribute immediately precedes a rule (just before the `rule` keyword) and governs the rule.

It asserts that the implicit conditions of all interface methods called within the rule must always be true, and therefore do not control its enabling. Only the explicit rule predicate controls whether it is enabled or not. This is statically verified by the compiler, and it will report an error if necessary.

Example.

```
// Import the FIFO package
import FIFO :: *;

// IfcCounter with read method
interface IfcCounter#(type t);
  method t      readCounter;
  method Action setReset(t a);
endinterface

// Definition of CounterType
typedef Bit#(16) CounterType;

// Module counter using IfcCounter interface
(* synthesize,
   RST_N = "reset_b",
   CLK   = "counter_clk",
   always_ready = "readCounter",
   always_enabled = "readCounter" *)
module counter (IfcCounter#(CounterType));

   // Reg counter gets reset to 1 asynchronously with the RST signal
   Reg#(CounterType)   counter   <-  mkRegA(1);

   // The 4 depth valueFifo contains a list of reset values
   FIFO#(CounterType)  valueFifo <-  mkSizedFIFO(4);

   /*   Next rule increases the counter with each counter_clk rising edge
        if the maximum has not been reached                              */
   (* no_implicit_conditions *)
   rule updateCounter;
   action
     if (counter != '1)
       counter <= counter + 1;
   endaction
   endrule

   // Next rule resets the counter to a value stored in the valueFifo
   (* no_implicit_conditions *)
```



```
  rule resetCounter (counter == '1);
  action
    counter <= valueFifo.first();
    valueFifo.deq();
  endaction
  endrule

  // Output the counters value
  method CounterType readCounter;
    return counter;
  endmethod

  // Update the valueFifo
  method Action setReset(CounterType a);
  action
    valueFifo.enq(a);
  endaction
  endmethod
endmodule
```

Assertion `no_implicit_conditions` will not be met for the rule `resetCounter` resulting in a compilation error. This rule has the implicit condition in the FIFO module due to the fact that the `deq` method cannot be invoked if the fifo `valueFifo` is empty. Please note that without the assertion no error will be produced and that the condition `if (counter != '1)` is not considered an implicit one.

13.3.3 Scheduling attribute `descending_urgency`

The compiler maps rules into clocks, as described in Section 6.2.2. In each clock, amongst all the rules enabled in that clock, the system picks a subset of rules that do not conflict with each other, so that their parallel execution is consistent with the reference TRS semantics. The order in which rules are considered for selection can affect the subset chosen. For example, suppose rules r1 and r2 conflict, and both are enabled. If r1 is considered first and selected, it may disqualify r2 from consideration, and vice versa. Note that the urgency ordering is independent of the TRS ordering of the rules, i.e., the TRS ordering may be r1-before-r2, but either one could be considered first by the compiler.

The designer can specify that one rule is more *urgent* than another, so that it is always considered for scheduling before the other. The relationship is transitive, i.e., if rule r1 is more urgent than rule r2, and rule r2 is more urgent than rule r3, then r1 is considered more urgent than r3.

Urgency is specified with the `descending_urgency` attribute. Its argument is a string containing a comma-separated list of rule names (see Section 5.6 for rule syntax, including rule names). Example:

```
(* descending_urgency = "r1, r2, r3" *)
```

This example specifies that `r1` is more urgent than `r2` which, in turn, is more urgent than `r3`.

If urgency attributes are contradictory, i.e., they specify both that one rule is more urgent than another and its converse, the compiler will report an error. Note that such a contradiction may be a consequence of a collection of urgency attributes, because of transitivity. One attribute may specify `r1` more urgent than `r2`, another attribute may specify `r2` more urgent than `r3`, and another attribute may specify `r3` more urgent than `r1`, leading to a cycle, which is a contradiction.

The `descending_urgency` attribute can be placed in any of three syntactic positions:

- It can be placed just before the `rules` keyword in a `rules-endrules` expression (Section 9.13), in which case it can refer directly to any of the rules in the expression.
- It can be placed just before the `rule` keyword in a `rule-endrule` construct, (Section 5.6) in which case it can refer directly to the rule or any other rules at the same level.
- It can be placed just before the `module` keyword in a `module-endmodule` construct (Section 5.3), in which case it can refer directly to any of the rules inside the module.

In addition, an urgency attribute can refer to any rule in the module hierarchy at or below the current module, using a hierarchical name. For example, suppose we have:

```
module mkFoo ...;

    mkBar  the_bar  (barInterface);

    (* descending_urgency = "r1, the_bar.r2" *)
    rule r1 ...
        ...
    endrule

endmodule: mkFoo
```

The hierarchical name `the_bar.r2` refers to a rule named `r2` inside the module instance `the_bar`. This can be several levels deep, i.e., the scheduling attribute can refer to a rule deep in the module hierarchy, not just the sub-module immediately below. In general a hierarchical rule name is a sequence of module instance names and finally a rule name, separated by periods.

A reference to a rule in a sub-module cannot cross synthesis boundaries. This is because synthesis boundaries are also scheduler boundaries. Each separately synthesized part of the module hierarchy contains its own scheduler, and cannot directly affect other schedulers. Urgency can only apply to rules considered within the same scheduler.

If rule urgency is not specified, and it impacts the choice of schedule, the compiler will print a notification to this effect during compilation.

Example. Using `descending_urgency` to control the scheduling of conflicting rules.

```
// IfcCounter with read method
interface IfcCounter#(type t);
  method t      readCounter;
endinterface

// Definition of CounterType
typedef Bit#(16) CounterType;

// Module counter using IfcCounter interface. It never contains 0.
(* synthesize,
   RST_N = "reset_b",
   CLK = "counter_clk",
   always_ready = "readCounter",
   always_enabled= "readCounter" *)
module counter (IfcCounter#(CounterType));

   // Reg counter gets reset to 1 asynchronously with the RST signal
```



```
  Reg#(CounterType)   counter   <-  mkRegA(1);

  /*    The descending_urgency attribute will indicate the scheduling
        order for the indicated rules.                                  */
  (* descending_urgency = "resetCounter, updateCounter" *)

  // Next rule resets the counter to 1 when it reaches its limit.
  rule resetCounter (counter == '1);
  action
    counter <= 1;
  endaction
  endrule

  //  Next rule updates the counter.
  rule updateCounter;
  action
    counter <= counter + 1;
  endaction
  endrule

  // Method to output the counter's value
  method CounterType readCounter;
    return counter;
  endmethod

endmodule
```

Rule `resetCounter` conflicts with rule `updateCounter` because both try to modify the counter register when it contains all its bits set to one. Without any `descending_urgency` attribute, the `updateCounter` rule will obtain more urgency, meaning that if the predicate of `resetCounter` is met, only the rule `updateCounter` will fire. By setting the `descending_urgency` attribute the designer can control the scheduling in the case of conflicting rules.

13.3.4 Scheduling attribute `preempts`

The designer can also prevent a rule from firing whenever another rule (or set of rules) fires. The `preempts` attribute accepts two elements as arguments. Each element may be either a rule name or a list of rule names. A list of rule names must be separated by commas and enclosed in parentheses. In each cycle, if any of the rule names specified in the first element can be executed and are scheduled to fire, then none of the rules specified in the second element will be allowed to fire.

The `preempts` attribute is similar to the `descending_urgency` attribute (section 13.3.3), and may occur in the same syntactic positions. The `preempts` attribute is equivalent to forcing a conflict and adding `descending_urgency`. With `descending_urgency`, if two rules do not conflict, then both would be allowed to fire even if an urgency order had been specified; with `preempts`, if one rule preempts the other, they can never fire together. If rule1 preempts rule2, then the compiler forces a conflict and gives rule1 priority. If rule1 is able to fire, but is not scheduled to, then rule2 can still fire.

Examples

```
(* preempts = "r1, r2" *)
```

If r1 will fire, r2 will not.


```
(* preempts = "(r1, r2), r3" *)
```

If either r1 or r2 (or both) will fire, r3 will not.

```
(* preempts = "(the_bar.r1, (r2, r3)" *)
```

If the rule r1 in the submodule the_bar will fire, then neither r2 nor r3 will fire.

13.4 Documentation attributes

A BSV design can specify comments to be included in the generated Verilog by use of the `doc` attribute.

docAttribute ::= *attributeInstance*

where

attrSpec ::= *attrName* [`=` *expression*]

and *expression* is a text string.

Example:

```
(* doc = "This is a user-provided comment" *)
```

To provide a multi-line comment, either include a `\n` character:

```
(* doc = "This is one line\nAnd this is another" *)
```

Or provide several instances of the `doc` attribute:

```
(* doc = "This is one line" *)
(* doc = "And this is another" *)
```

Or:

```
(* doc = "This is one line",
   doc = "And this is another" *)
```

Multiple `doc` attributes will appear together in the order that they are given. `doc` attributes can be added to modules, module instantiations, and rules, as described in the following sections.

13.4.1 Documentation attribute - modules

The Verilog file that is generated for a synthesized BSV module contains a header comment prior to the Verilog module definition. A designer can include additional comments between this header and the module by attaching a `doc` attribute to the module being synthesized. This attribute is like the *modgenAttributes* described in Section 13.1, in that it is dependent on the `synthesize` attribute. If the `synthesize` attribute is not specified for the module, the `doc` attributes are ignored.

Example:

```
(* synthesize *)
(* doc = "This is important information about the following module" *)
module mkMod (IFC);
  ...
endmodule
```


13.4.2 Documentation attribute - module instantiation

In generated Verilog, a designer might want to include a comment on submodule instantiations, to document something about that submodule. This can be achieved with a `doc` attribute on the corresponding BSV module. There are three ways to express instantiation in BSV syntax, and the `doc` attribute can be attached to all three.

```
(* doc = "This submodule does something" *)
FIFO#(Bool) f();
mkFIFO the_f(f);

(* doc = "This submodule does something else" *)
Server srv <- mkServer;

Client c;
...
(* doc = "This submodule does a third thing" *)
c <- mkClient;
```

The syntax also works if the type of the module interface is given with `let`, a variable, or the current `module` type. Example:

```
(* doc = "This submodule does something else" *)
let srv <- mkServer;
```

If the submodule being instantiated is a separately synthesized module or primitive, then its corresponding Verilog instantiation will be preceded by the comments. Example:

```
// submodule the_f
// This submodule does something
wire the_f$CLR, the_f$DEQ, the_f$ENQ;
FIFO2 #(.width(1)) the_f(...);
```

If the submodule is not separately synthesized, then there is no place in the Verilog module to attach the comment. Instead, the comment is included in the header at the beginning of the module. For example, assume that the module `the_sub` was instantiated inside `mkTop` with a user-provided comment but was not separately synthesized. The generated Verilog would include these lines:

```
// ...
// Comments on the inlined module ‘the_sub’:
//   This is the submodule
//
module mkTop(...);
```

The `doc` attribute can be attached to submodule instantiations inside functions and for-loops.

If several submodules are inlined and their comments carry to the top-module's header comment, all of their comments are printed. To save space, if the comments on several modules are the same, the comment is only displayed once. This can occur, for instance, with `doc` attributes on instantiations inside for-loops. For example:

```
// Comments on the inlined modules ‘the_sub_1‘, ‘the_sub_2‘,
// ‘the_sub_3‘:
//   ...
```


If the doc attribute is attached to a register instantiation, the Verilog comment is included with the declaration of the register signals. Example:

```
// register the_r
// This is a register
reg the_r;
wire the_r$D_IN, the_r$EN;
```

If the doc attribute is attached to an RWire instantiation, and the wire instantiation is inlined (as is the default), then the comment is carried to the top-module's header comment.

If the doc attribute is attached to a probe instantiation, the comment appears in the Verilog above the declaration of the probe signals. Since the probe signals are declared as a group, the comments are listed at the start of the group. Example:

```
// probes
//
// Comments for probe 'the_r':
//   This is a probe
//
wire the_s$PROBE;
wire the_r$PROBE;
...
```

13.4.3 Documentation attribute - rules

In generated Verilog, a designer might want to include a comment on rule scheduling signals (such as CAN_FIRE_ and WILL_FIRE_ signals), to say something about the actions that are performed when that rule is scheduled. This can be achieved with a doc attribute attached to a BSV rule declaration.

The doc attribute can be attached to any rule..endrule statement. Example:

```
(* doc = "This rule is important" *)
rule do_something (b);
   x <= !x;
endrule
```

If any scheduling signals for the rule are explicit in the Verilog output, their definition will be preceeded by the comment. Example:

```
// rule RL_do_something
//   This rule is important
assign CAN_FIRE_RL_do_something = b ;
assign WILL_FIRE_RL_do_something = CAN_FIRE_RL_do_something ;
```

If the signals have been inlined or otherwise optimized away and thus do not appear in the Verilog, then there is no place to attach the comments. In that case, the comments are carried to the top module's header. Example:

```
// ...
// Comments on the inlined rule 'RL_do_something':
//   This rule is important
//
module mkTop(...);
```


The designer can ensure that the signals will exist in the Verilog by using an appropriate compiler flag, the `-keep-fires` flag which is documented in the Bluespec SystemVerilog User Guide.

The `doc` attribute can be attached to any `rule..endrule` expression, such as inside a function or inside a for-loop.

As with comments on submodules, if the comments on several rules are the same, and those comments are carried to the top-level module header, the comment is only displayed once.

```
// ...
// Comments on the inlined rules 'RL_do_something_2', 'RL_do_something_1',
// 'RL_do_something':
//   This rule is important
//
module mkTop(...);
```

14 Advanced topics

This section can be skipped on first reading.

14.1 Type classes (overloading groups) and provisos

Note that for most BSV programming, one just needs to know about a few predefined type classes such as `Bits` and `Eq`, about provisos, and about the automatic mechanism for defining the overloaded functions in those type classes using a `deriving` clause. The brief introduction in Sections 4.2 and 4.3 should suffice.

This section is intended for the advanced programmer who may wish to define new type classes (using a `typeclass` declaration), or explicitly to define overloaded functions using an `instance` declaration.

In programming languages, the term *overloading* refers to the use of a common function name or operator symbol to represent some number (usually finite) of functions with distinct types. For example, it is common to overload the operator symbol `+` to represent integer addition, floating point addition, complex number addition, matrix addition, and so on.

Note that overloading is distinct from *polymorphism*, which is used to describe a single function or operator that can operate at an infinity of types. For example, in many languages, a single polymorphic function `arraySize()` may be used to determine the number of elements in any array, no matter what the type of the contents of the array.

A *type class* (or *overloading group*) further recognizes that overloading is often performed with related groups of function names or operators, giving the group of related functions and operators a name. For example, the type class `Ord` contains the overloaded operators for order-comparison: `<`, `<=`, `>` and `>=`.

If we specify the functions represented by these operator symbols for the types `int`, `Bool`, `bit[`m`:0]` and so on, we say that those types are *instances* of the `Ord` type class.

A *proviso* is a (static) condition attached to some constructs. A proviso requires that certain types involved in the construct must be instances of certain type classes. For example, a generic `sort` function for sorting lists of type `List#(`t`)` will have a proviso (condition) that t must be an instance of the `Ord` type class, because the generic function uses an overloaded comparison operator from that type class, such as the operator `<` or `>`.

Type classes are created explicitly using a `typeclass` declaration (Section 14.1.2). Further, a type class is explicitly populated with a new instance type t, using an `instance` declaration (Section 14.1.3), in which the programmer provides the specifications for the overloaded functions for the type t.

14.1.1 Provisos

Consider the following function prototype:

```
function List#(t) sort (List#(t) xs)
    provisos (Ord#(t));
```

This prototype expresses the idea that the sorting function takes an input list `xs` of items of type `t` (presumably unsorted), and produces an output list of type `t` (presumably sorted). In order to perform its function it needs to compare elements of the list against each other using an overloaded comparison operator such as `<`. This, in turn, requires that the overloaded operator be defined on objects of type `t`. This is exactly what is expressed in the proviso, i.e., that `t` must be an instance of the type class (overloading group) `Ord`, which contains the overloaded operator `<`.

Thus, it is permissible to apply `sort` to lists of `Integer`s or lists of `Bool`s, because those types are instances of `Ord`, but it is not permissible to apply `sort` to a list of, say, some interface type `Ifc` (assuming `Ifc` is not an instance of the `Ord` type class).

The syntax of provisos is the following:

provisos ::= `provisos (` *proviso* { `,` *proviso* } `)`

proviso ::= *Identifier* `#(`*type* { `,` *type* } `)`

In each *proviso*, the *Identifier* is the name of type class (overloading group). In most provisos, the type class name T is followed by a single type t, and can be read as a simple assertion that t is an instance of T, i.e., that the overloaded functions of type class T are defined for the type t. In some provisos the type class name T may be followed by more than one type t_1, ..., t_n and these express more general relationships. For example, a proviso like this:

```
provisos (Bits#(macAddress, 48))
```

can be read literally as saying that the types `macAddress` and `48` are in the `Bits` type class, or can be read more generally as saying that values of type `macAddress` can be converted to and from values of the type `bit[47:0]` using the `pack` and `unpack` overloaded functions of type class `Bits`.

We sometimes also refer to provisos as *contexts*, meaning that they constrain the types that may be used within the construct to which the provisos are attached.

Occasionally, if the context is too weak, the compiler may be unable to figure out how to resolve an overloading. Usually the compiler's error message will be a strong hint about what information is missing. In these situations it may be necessary for the programmer to guide the compiler by adding more type information to the program, in either or both of the following ways:

- Add a static type assertion (Section 9.10) to some expression that narrows down its type.
- Add a proviso to the surrounding construct.

14.1.2 Type class declarations

A new class is declared using the following syntax:

typeclassDef ::= `typeclass` *typeclassIde typeFormals* [*provisos*]
[*typedepends*] `;`
{ *overloadedDef* }
`endtypeclass` [`:` *typeclassIde*]

typeclassIde ::= *Identifier*

typeFormals	::=	`#` `(` *typeFormal* { `,` *typeFormal* })
typeFormal	::=	[`numeric`] `type` *typeIde*
typedepends	::=	`dependencies` `(` *typedepend* { `,` *typedepend* } `)`
typedepend	::=	*type* `determines` *type*
overloadedDef	::=	*functionProto*
	|	*varDecl*

The *typeclassIde* is the newly declared class name. The *typeFormals* represent the types that will be instances of this class. These *typeFormals* may themselves be constrained by *provisos*, in which case the classes named in *provisos* are called the "super type classes" of this type class. Type dependencies (*typedepends*) are relevant only if there are two or more *type* parameters; the *typedepends* comes after the typeclass's provisos (if any) and before the semicolon. The *overloadedDef*s declare the overloaded variables or function names, and their types.

Example (from the Standard Prelude package):

```
typeclass Literal#(type a);
    function a  fromInteger (Integer x);
endtypeclass: Literal
```

This defines the type class `Literal`. Any type `a` that is an instance of `Literal` must have an overloaded function called `fromInteger` that converts an `Integer` value into the type `a`. In fact, this is the mechanism that BSV uses to interpret integer literal constants, e.g., to resolve whether a literal like `6847` is to be interpreted as a signed integer, an unsigned integer, a floating point number, a bit value of 10 bits, a bit value of 8 bits, etc. (See Section 2.3.1 for a more detailed description.)

Example (from a predefined type class in BSV):

```
typeclass Bounded#(type a);
    a minBound;
    a maxBound;
endtypeclass
```

This defines the type class `Bounded`. Any type `a` that is an instance of `Bounded` will have two values called `minBound` and `maxBound` that, respectively, represent the minimum and maximum of all values of this type.

Example (from a predefined type class in BSV):[7]

```
typeclass Arith#(type a) provisos (Literal#(a));
    function a \+     (a x1, a x2);
    function a \-     (a x1, a x2);
    function a negate (a x1);           // available as prefix "-"
    function a \*     (a x1, a x2);
endtypeclass
```

This defines the type class `Arith` with super type class `Literal`, i.e., the proviso states that in order for a type `a` to be an instance of `Arith` it must also be an instance of the type class `Literal`. Further, it has four overloaded functions with the given names and types. Said another way, a type that is an instance of the `Arith` type class must have a way to convert integer literals into that type, and it must have addition, subtraction, negation and multiplication defined on it.

[7] We are using Verilog's notation for *escaped identifiers* to treat operator symbols as ordinary identifiers. The notation allows an identifier to be constructed from arbitrary characters beginning with a backslash and ending with a whitespace (the backslash and whitespace are not part of the identifier.)

The semantics of a dependency says that once the types on the left of the colon are fixed, the types on the right are also uniquely determined. For example, for any type `t` we know that `Get#(t)` and `Put#(t)` are connectable (because there's an instance delcaration in the `GetPut` package that says so); then the dependency specification above syas that if you know that `a` is `Get#(t)`, the *only* possibility for `b` is `Put#(t)`.

An example of a typeclass definition specifying type dependencies:

```
typeclass Connectable #(type a, type b)
    dependencies (a determines b, b determines a);
       module mkConnections#(a x1, b x2) (Empty);
endtypeclass
```

An example of a case where the dependencies are not commutative:

```
typeclass Bita#(type a, type sa)
   dependencies (a determines sa);
      function Bit#(sa) pack(a x);
      function a unpack (Bit#(sa) x);
endtypeclass
```

In the above example, if `a` were `UInt#(16)` the dependency would requrie that `b` had to be 16; but that fact that something occupies 16 bits by no means implies that it has to be a `UInt`.

14.1.3 Instance declarations

A type can be declared to be an instance of a class in two ways, with a general mechanism or with a convenient shorthand. The general mechanism of `instance` declarations is the following:

typeclassInstanceDef ::= `instance` *typeclassIde* `#` `(` *type* { `,` *type* } `)` [*provisos*] `;`
{ *varAssign* `;` | *functionDef* | *BSVmoduleHead* }
`endinstance` [`:` *typeclassIde*]

This says that the *type*s are an instance of type class *typeclassIde* with the given provisos. The *varAssign*s, *functionDef*s and *BSVmoduleHead*s specify the implementation of the overloaded identifiers of the type class.

Example, declaring a type as an instance of the `Eq` typeclass:

```
typedef enum { Red, Blue, Green } Color;

instance Eq#(Color);
  function Bool \== (Color x, Color y); //must use \== with a trailing
    return True;                        //space to define custom instances
  endfunction                           //of the Eq typeclass
endinstance
```

The shorthand mechanism is to attach a `deriving` clause to a typedef of an enum, struct or tagged union and let the compiler do the work. In this case the compiler chooses the "obvious" implementation of the overloaded functions (details in the following sections). The only type classes for which `deriving` can be used for general types are `Bits`, `Eq` and `Bounded`. Furthermore, `deriving` can be used for any class if the type is a data type that is isomorphic to a type that has an instance for the derived class.

derives ::= `deriving` `(` *typeclassIde* { `,` *typeclassIde* } `)`

Example:

```
typedef enum { Red, Blue, Green } Color deriving (Eq);
```


14.1.4 The `Bits` type class (overloading group)

The type class `Bits` contains the types that are convertible to bit strings of a certain size. Many constructs have membership in the `Bits` class as a proviso, such as putting a value into a register, array, or FIFO.

Example. The `Bits` type class definition (which is actually predefined in BSV) looks something like this:

```
typeclass Bits#(type a, type n);
    function  Bit#(n)  pack   (a x);
    function  a        unpack (Bit#(n) y);
endtypeclass
```

Here, `a` represents the type that can be converted to/from bits, and `n` is always instantiated by a size type (Section 4) representing the number of bits needed to represent it. Implementations of modules such as registers and FIFOs use these functions to convert between values of other types and the bit representations that are really stored in those elements.

Example. The most trivial instance declaration states that a bit-vector can be converted to a bit vector, by defining both the `pack` and `unpack` functions to be identity functions:

```
instance Bits#(Bit#(k), k);
    function Bit#(k) pack (Bit#(k) x);
        return x;
    endfunction: pack

    function Bit#(k) unpack (Bit#(k) x);
        return x;
    endfunction: unpack
endinstance
```

Example:

```
typedef enum { Red, Green, Blue } Color deriving (Eq);

instance Bits#(Color, 2);
    function Bits#(2) pack (Color c);
        if      (c == Red)   return 3;
        else if (c == Green) return 2;
        else                 return 1;   // (c == Blue)
    endfunction: pack

    function Color unpack (Bits#(2) x);
        if      (x == 3) return Red;
        else if (x == 2) return Green;
        else if (x == 1) return Blue;
        else $error("Illegal code 0 for unpacking a Color'');
    endfunction: unpack
endinstance
```

Note that the `deriving (Eq)` phrase permits us to use the equality operator `==` on `Color` types in the `pack` function. `Red`, `Green` and `Blue` are coded as 3, 2 and 1, respectively. If we had used the `deriving(Bits)` shorthand in the `Color` typedef, they would have been coded as 0, 1 and 2, respectively (Section 14.1.6).

14.1.5 The `SizeOf` pseudo-function

The pseudo-function `SizeOf(`t`)` can be applied to a type t to get the numeric type representing its bit size. The type t must be in the `Bits` class, i.e., it must already be an instance of `Bits#(`t`,`n`)`, either through a `deriving` clause or through an explicit instance declaration. The `SizeOf` function then returns the corresponding bit size n. Note that `SizeOf` returns a numeric type, not a numeric value, i.e., the output of `SizeOf` can be used in a type expression, and not in a value expression.

`SizeOf`, which converts a type to a (numeric) type, should not be confused with the pseudo-function `valueof`, described in Section 4.2.1, which converts a numeric type to a numeric value.

14.1.6 Deriving `Bits`

When attaching a `deriving(Bits)` clause to a user-defined type, the instance derived for the `Bits` type class can be described as follows:

- For an enum type it is simply an integer code, starting with zero for the first enum constant and incrementing by one for each subsequent enum constant. The number of bits used is the minimum number of bits needed to represent distinct codes for all the enum constants.
- For a struct type it is simply the concatenation of the bits for all the members. The first member is in the leftmost bits (most significant) and the last member is in the rightmost bits (least significant).
- For a tagged union type, all values of the type occupy the same number of bits, regardless of which member it belongs to. The bit representation consists of two parts—a tag on the left (most significant) and a member value on the right (least significant).

  The tag part uses the minimum number of bits needed to code for all the member names. The first member name is given code zero, the next member name is given code one, and so on.

  The size of the member value part is always the size of the largest member. The member value is stored in this field, right-justified (i.e., flush with the least-significant end). If the member value requires fewer bits than the size of the field, the intermediate bits are don't-care bits.

Example. Symbolic names for colors:

```
typedef enum { Red, Green, Blue } Color deriving (Eq, Bits);
```

This is the same type as in Section 14.1.4 except that `Red`, `Green` and `Blue` are now coded as 0, 1 and 2, instead of 3, 2, and 1, respectively, because the canonical choice made by the compiler is to code consecutive labels incrementing from 0.

Example. The boolean type can be defined in the language itself:

```
typedef enum { False, True} Bool deriving (Bits);
```

The type `Bool` is represented with one bit. `False` is represented by 0 and `True` by 1.

Example. A struct type:

```
typedef struct { Bit#(8) foo; Bit#(16) bar } Glurph deriving (Bits);
```

The type `Glurph` is represented in 24 bits, with `foo` in the upper 8 bits and `bar` in the lower 16 bits.

Example. Another struct type:


```
typedef struct{ int x; int y } Coord deriving (Bits);
```

The type `Coord` is represented in 64 bits, with `x` in the upper 32 bits and `y` in the lower 32 bits.

Example. The `Maybe` type from Section 7.3:

```
typedef union tagged {
    void  Invalid;
    a     Valid;
} Maybe#(type a)
  deriving (Bits);
```

is represented in $1 + n$ bits, where n bits are needed to represent values of type `a`. If the leftmost bit is 0 (for `Invalid`) the remaining n bits are unspecified (don't-care). If the leftmost bit is 1 (for `Valid`) then the remaining n bits will contain a value of type `a`.

14.1.7 Deriving `Eq`

The `Eq` type class contains the overloaded operators `==` (logical equality) and `!=` (logical inequality):

```
typeclass Eq#(type a);
    function Bool \== (a x1, a x2);
    function Bool \!= (a x1, a x2);
endtypeclass: Eq
```

When `deriving(Eq)` is present on a a user-defined type definition t, the compiler defines these equality/inequality operators for values of type t. It is the natural recursive definition of these operators, i.e.,

- If t is an enum type, two values of type t are equal if they represent the same enum constant.
- If t is a struct type, two values of type t are equal if the corresponding members are pairwise equal.
- If t is a tagged union type, two values of type t are equal if they have the same tag (member name) and the two corresponding member values are equal.

14.1.8 Deriving `Bounded`

The predefined type class `Bounded` contains two overloaded identifiers `minBound` and `maxBound` representing the minimum and maximum values of a type `a`:

```
typeclass Bounded#(type a);
    a minBound;
    a maxBound;
endtypeclass
```

The clause `deriving(Bounded)` can be attached to any user-defined enum definition t, and the compiler will define the values `minBound` and `maxBound` for values of type t as the first and last enum constants, respectively.

The clause `deriving(Bounded)` can be attached to any user-defined struct definition t with the proviso that the type of each member is also an instance of `Bounded`. The compiler-defined `minBound` (or `maxBound`) will be the struct with each member having its respective `minBound` (respectively, `maxBound`).

14.1.9 Deriving type class instances for isomorphic types

Generally speaking, the `deriving(...)` clause can only be used for the predefined type classes `Bits`, `Eq` and `Bounded`. However there is a special case where it can be used for any type class. When a user-defined type t is *isomorphic* to an existing type t', then all the functions on t' automatically work on t, and so the compiler can trivially derive a function for t by just using the corresponding function for t'.

There are two situations where a newly defined type is isomorphic to an old type: a struct or tagged union with precisely one member. For example:

```
typedef struct { t' x; } t deriving (anyClass);
typedef union tagged { t' X; } t deriving (anyClass);
```

One sometimes defines such a type precisely for type-safety reasons because the new type is distinct from the old type although isomorphic to it, so that it is impossible to accidentally use a t value in a t' context and vice versa. Example:

```
typedef struct { UInt#(32) x; } Apples deriving (Literal, Arith);
...
Apples five;
...
five = 5;    // ok, since RHS applies 'fromInteger()' from Literal
             // class to Integer 5 to create an Apples value

function Apples eatApple (Apples n);
    return n - 1;        // '1' is converted to Apples by fromInteger()
                         // '-' is available on Apples from Arith class
endfunction: eatApple
```

The typedef could also have been written with a singleton tagged union instead of a singleton struct:

```
typedef union tagged { UInt#(32) X; } Apples deriving (Literal, Arith);
```

14.2 Higher-order functions

In BSV it is possible to write an expression whose value is a *function value*. These function values can be passed as arguments to other functions, returned as results from functions, and even carried in data structures. Example:

```
function Arr#(n,b) mapArr (function  b  f (a x),
                           Arr#(n,a)    xs);
    Arr#(n,b) ys;

    for (Bit#(n) j = 0; j < n; j=j+1)
        ys [j] = f (xs [j]);

    return ys;
endfunction: mapArr

function int sqr (int x);
   return x * x;
```



```
endfunction: sqr

Arr#(100,int) as = ...; // initialize array as

Arr#(100,int) bs = mapArr (sqr, as);
```

The function `mapArr` is polymorphic, i.e., is defined for any size type `n` and value types `a` and `b`. It takes two arguments:

- A function `f` with input of type `a` and output of type `b`.
- An array `xs` of size `n` containing values of type `a`.

Its result is a new array `ys` that is also of size `n` and containing values of type `b`, such that `ys[j]=f(xs[j])`. In the last line of the example, we call `mapArr` passing it the `sqr` function and the array `as` to produce an array `bs` that contains the squared versions of all the elements of array `as`.

Observe that in the last line, the expression `sqr` is a function-valued expression, representing the squaring function. It is not an invocation of the `sqr` function. Similarly, inside `mapArr`, the identifier `f` is a function-valued identifier, and the expression `f (xs [j])` invokes the function.

The function `mapArr` could be called with a variety of arguments:

```
// shift all elements of as left by 2
Arr#(100,int) bs = mapArr (shiftLeft2, as);
```

or

```
// test all elements of as for even-ness
Arr#(100,Bool) bs = mapArr (isEven, as);
```

In other words, `mapArr` captures, in one definition, the generic idea of applying some function to all elements of an array and returning all the results in another array. This is a very powerful idea enabled by treating functions as first-class values. Here is another example, which may be useful in many hardware designs:

```
interface SearchableFIFO#(type a);
    ... usual enq() and deq() methods ...

    method Bool search (a y);

endinterface: SearchableFIFO

module mkSearchableFIFO#(function Bool f(a x, a y))
                        (SearchableFIFO#(a));
    ...
    method search (a y);
        ... apply f(x, y) to each element of the FIFO, ...
        ... return OR of all results ...
    endmethod: search
endmodule: mkSearchableFIFO
```


The `SearchableFIFO` interface is like a normal FIFO interface (contains usual `enq()` and `deq()` methods), but it has an additional bit of functionality. It has a `search()` method to which you can pass a search key `y`, and it searches the FIFO using that key, returning `True` if the search succeeds.

Inside the `mkSearchableFIFO` module, the method applies some element test `f` to the search key and each element of the FIFO and ORs all the results. The particular element-test function `f` to be used is passed in as a parameter to `mkSearchableFIFO`. In one instantiation of `mkSearchableFIFO` we might pass in the equality function for this parameter ("search this FIFO for this particular element"). In another instantiation of `mkSearchableFIFO` we might pass in the "greater-than" function ("search this FIFO for any element greater than the search key"). Thus, a single FIFO definition captures the general idea of being able to search a FIFO, and can be customized for different applications by passing in different search functions to the module constructor.

A final important point is that all this is perfectly *synthesizable* in `BSV`, i.e., the compiler can produce RTL hardware for such descriptions.

14.3 Calling foreign functions

This section describes how to encapsulate a Verilog function inside a BSV wrapper.

TBD: This section needs to be updated.

15 Interfacing to Verilog

This section describes two related mechanisms:

- How to embed a Verilog module in a BSV module, i.e., how to encapsulate a Verilog module inside a BSV shim. This is the method to utilize existing Verilog components, or Verilog components generated by other tools or how to define a custom set of primitives to be used in multiple designs. One example is the BSV primitives (registers, FIFOs, etc.) which are implemented this way.
- How to embed a BSV module in a Verilog module, i.e., how a surrounding Verilog module can use a BSV module as a client.

15.1 Embedding Verilog in a BSV design

This is used when there are existing Verilog components which the designer would like to include in a BSV module.

externModuleImport ::= `import` "BVI" [*identifier*] `=` *BSVmoduleHead*
{ *moduleStmt* }
{ *importBVIStmt* }
`endmodule` [`:` *identifier*]

The body consists of a sequence of *importBVIStmts*:

importBVIStmt ::= `parameter` ...
| `port` ...
| `default_clock` ...
| `input_clock` ...
| `output_clock` ...
| `no_reset` ...
| `default_reset` ...


```
|    input_reset ...
|    ouput_reset ...
|    ancestor ...
|    same_family ...
|    method ...
|    schedule ...
|    path ...
```

A general example of embedding Verilog in BSV follows; the meaning of each line is explained in the following sections. This example is unusually complex in order to show many statements.

```
import "BVI" VerilogModule =
    module mkModule#(Integer start) (Clock sClkIn, Reset sRstIn,
                Clock dClkIn, Bool oscillator,
                SyncBitIfc#(a_type) ifc )
                provisos( Bits#(a_type, awidth)) ;
    default_clock clk_(CLK200);
    input_clock dstclk(CLK400) = dCLKIn;
    method D_OUT bsv_name(D_IN) enable(EN) ready(RDY) clocked_by(dstclk);
    parameter VPARAM = valueOf(awidth);
    port VPORT = oscillator;
    schedule (first_, notEmpty_, notFull_) CF (first_, notEmpty_, notFull_);
    schedule first_ SB (clear_, deq_);
    path (ENQ, D_OUT);
endmodule: mkModule
```

15.1.1 Header

The header takes the form

externModuleImport ::= `import` "BVI" [*identifier* =] *BSVmoduleHead*

BSVmoduleHead ::= `module` *identifier*
[*moduleFormalParams*] `(` [*moduleFormalArgs*] `)` [*provisos*];

The *identifier* is the name of the Verilog module to be imported. This will usually be found in a file called *identifier*.v, normally in the home directory or the Verilog sub-directory of the BSV system directory. If the *identifier* is excluded, it is assumed that the Verilog module name is the same as the BSV name of the module.

The *BSVmoduleHead* is the usual first line in the module definition as explained in 5.3.

The BSV module may be of any module type. Note that the BSV module's parameters have no inherent relationship to the Verilog module's parameters.

Example:

```
import "BVI" SyncBit15 =
module SyncBit15 #(Integer start)
                  (Clock sClkIn, Reset sRstIn, Clock dClkIn,
                  SyncBitIfc#(a_type) ifc )
         provisos( Bits#(a_type, awidth)) ;
```

Since the Verilog module's name matches the BSV name, the header could be also written as:


```
import "BVI"
module SyncBit15 #(Integer start)
                  (Clock sClkIn, Reset sRstIn, Clock dClkIn,
                  SyncBitIfc#(a_type) ifc )
         provisos( Bits#(a_type, awidth)) ;
```

15.1.2 Body

The module body may contain both *moduleStmt*s and *importBVIStmt*s. Typically, when including a Verilog module, the only module statements would be a few local definitions. However, all module statements, except for method and sub-interface definitions and return statements are valid, though most are rarely used in this instance. Only the statements specific to *importBVIStmt* bodies are described below.

The *importBVIStmt*s must occur at the end of the body, after the *moduleStmt*s. They may be written in any order. If any contain expressions with side effects, these side effects will occur in the order given.

15.1.3 `parameter`

The form is

`parameter` *name* `=` *expression* `;`

The value of the expression is supplied to the Verilog module as the parameter named *name*. The *expression* must be a compile-time constant. If the compiler cannot infer its type, it will default to Integer, but the value will later be converted to `Bit#(32)`. Note that the String type is also supported.

Example:

```
import "BVI" ClockGen =
module vAbsoluteClock#(Integer start,
                       Integer period) ( ClockGenIfc );
    let halfPeriod = div( period,  2) ;

    parameter initDelay  = start;              //the parameters start,
    parameter v1Width = halfPeriod ;           //halfPeriod and period
    parameter v2Width = period - halfPeriod ;  //must be compile-time constants
endmodule
```

15.1.4 `port`

The form is

`port` *name* `=` *expression* `;`

The value of the *expression* is supplied to the Verilog port named *name*. The operator `<-` may be used as an alternative to `=`, where appropriate. The type of the *expression* must be in the Bits typeclass. It may even be dynamic (e.g. the `_read` method of a register instantiated elsewhere in the module body). If the width of the value is not the same as that of the port, the normal Verilog linking convention will be followed; the value will be truncated or zero-extended to fit.

Example:


```
import "BVI" MakeClock =
module vClock( Bit#(1) oscillator, Bool gate, ClockGenIfc ifc);
   default_clock no_clock ;
   no_reset;
   port OSSC_IN = oscillator;
   port GATE_IN = gate;
   output_clock gen_clk(CLK, CLKGATE);
endmodule
```

15.1.5 clock statements

The following sections describe statements which define the clock signals in the module. A clock consists of two signals, or ports. The first is an oscillator (clock) and the second is a gating signal (gate). In general these are implemented as two wires. In many statements, the second port may be optional. If there is only one port specified, it is the oscillator.

This example shows an *importBVI* statement defining clocks.

```
import "BVI" ClockSelect =  module vClockSelect( Integer stages,
                            Clock aClk, Clock bClk,
                            SelectClkIfc ifcout ) ;

   default_clock xclk(CLK) ;
   default_reset xrst(RST_N) ;

   input_clock  aClk(A_CLK, A_CLKGATE) = aClk ;
   input_clock  bClk(B_CLK, B_CLKGATE) = bClk ;

   // Generate the clock output interface
   output_clock clock_out( OUT_CLK, OUT_CLKGATE) ;

   output_reset reset_out( OUT_RST_N ) clocked_by(clock_out);

endmodule
```

15.1.6 default_clock

Each module has a default_clock, which is the clock signal which will be passed to any interior instantiations of the module, unless another clock is explicitly specified. Each clock consists of two ports, the oscillator and the gate. If there is only one port specified, it is the oscillator.

The form is

`default_clock` *name* `(` *portname*`,` *portname* `) =` *expression* `;`

This is precisely equivalent to

`default_clock` *name* `;`

`input_clock` *name* `(` *portname*`,` *portname* `) =` *expression* `;`

The *name* of the `default_clock` specifies the clock which is to be associated with any method which is not given an explicit `clocked_by` statement.

The defining operator `=` or `<-` may be used. `<-` indicates that the *expression* is a procedure with side effects, that must be evaluated to produce the value of the clock.

If either the parentheses and their contents or the = *expression* is omitted (but not both), a default value is used instead. In this case the name also may be omitted. (If everything except the name is omitted, no default values are supplied, and there is no implicit `input_clock` definition as in the equivalent form given above. It is an error to omit all three items—the intended effect is probably the same as omitting the default_clock statement altogether.)

The = *expression* in the `default_clock` statement defaults to `<- exposeCurrentClock`. The parentheses and their contents default to `CLK, CLK_GATE`.

If the *name* is omitted, and if the *expression* is an identifier and the defining operator is =, that identifier is used as the name. Otherwise a new unique name is invented. Example

```
default_clock (OSC, GATE) = clk;
```

is equivalent to

```
default_clock clk (OSC, GATE) = clk;
```

Omitting the entire statement is equivalent to

```
default_clock (CLK, CLK_GATE) <- exposeCurrentClock;
```

specifying that the current clock is to be associated with all methods which do not specify otherwise.

There is one exceptional case: the statement

```
default_clock no_clock;
```

is valid, and one may assume that the clock `no_clock` is implicitly defined.

Example:

```
import "BVI" GatedClock =
module vGatedClock2( Clock clk_in, Bool gate, ClockGenIfc ifc );
    default_clock clk_in(CLK, CLK_GATE) = clk_in;
    no_reset;
    port COND = gate ;
    output_clock gen_clk(CLK_OUT, CLK_GATE_OUT);
    ancestor(gen_clk, clk_in);
endmodule
```

15.1.7 `input_clock`

The input clock is defined as

`input_clock` *name* `(` *portname*`,` *portname* `)` `=` *expression* `;`

where the = may be replaced by <-. The form specifies that the clock given by *expression* is to be connected to the two Verilog ports specified in the parentheses (the first being the oscillator and the second the gate). The *name* may be used to associate the clock with methods using it in a `clocked_by` argument.

The = *expression* may not be omitted.

Either or both of the *portnames* may be omitted, indicating that the wire concerned is not connected to any Verilog port. It is the designer's responsibility to ensure that this does not lead to incorrect

behavior. For example, if the Verilog module is purely combinational, there is no requirement to connect a clock, though there may still be a need to associate its methods with a clock, to ensure that they are in the correct clock domain. Similarly, if a Verilog module has no internal transitions and responds only to method calls, it might be unnecessary to connect the gating signal, as the implicit condition mechanism will ensure that no method is invoked if its clock is off. Note that if only one *portname* is specified it is assumed to be the clock, not the gate. If both *portnames* are given, but the *expression* specifies a clock which is ungated, the gate port is tied to logical 1. The parentheses themselves may not be omitted.

The *name* associated with the `input_clock` name may be omitted. In this case, if the *expression* is actually an identifier and the defining operator is `=`, that identifier is used instead as the name (and may be used for the `clocked_by` argument of methods). Otherwise, no name is associated with the clock and, accordingly, no method may be associated with it either. It is an error if both *portnames* and the *name* are omitted, as the clock is then unusable.

Example:

```
import "BVI" JoinClock =
module vJoinedClock(Clock c, ClockGenIfc ifc );
    default_clock clk(CLK, CLK_GATE);
    no_reset;
    input_clock c(JOIN_CLK, JOIN_CLK_GATE) = c;
    output_clock gen_clk(NEW_CLK, NEW_CLK_GATE);
    same_family(clk, c);

    ancestor(gen_clk, clk);
    ancestor(gen_clk, c);
endmodule
```

15.1.8 `output_clock`

The output clock is the named instance of a clock being provided by the BSV module.

The form is

`output_clock` *name* `(` *portname*`,` *portname* `);`

The second *portname* may be omitted, indicating that the clock being provided is ungated. It is an error for the same *name* to be declared by more than one `output_clock` statement.

Example:

```
import "BVI" ClockMux =
module vClockMux( Bool ab, Clock aClk, Clock bClk, ClockGenIfc ifc ) ;
    default_clock clk() ;
    default_reset rst() ;

    input_clock  aClkX(A_CLK, A_CLKGATE) = aClk ;
    input_clock  bClkX(B_CLK, B_CLKGATE) = bClk ;
    port  SELECT = ab ;

    // Generate the clock output interface
    output_clock gen_clk( CLK, CLKGATE) ;

endmodule
```


15.1.9 no_reset

The form is

```
no_reset ;
```

`no_reset` specifies that the reset signal is not connected to any port.

Example:

```
import "BVI" ClockGen =
module vAbsoluteClock#(Integer start,
                       Integer period) ( ClockGenIfc );
    no_reset;
 endmodule
```

15.1.10 default_reset

Each module has a default_reset, which is the reset signal which will be passed to any interior instantiations of the module, unless another reset signal is explicitly specified or no_reset is specified.

The form is

```
default_reset name ( portname ) = expression ;
```

The *name* of the `default_reset` specifies the reset which is to be associated with any method which is not given an explicit `reset_by` statement.

The defining operator `=` or `<-` may be used. `<-` indicates that the *expression* is a procedure with side effects, that must be evaluated to produce the value of the reset.

The `=` *expression* in the `default_reset` statement may be omitted and defaults to `<- exposeCurrentReset`.

Example:

```
import "BVI" GatedClock =
module vGatedClock ( ClockTickIfc );
    default_clock clk_in(CLK, CLK_GATE);
    default_reset rst() ;
endmodule
```

15.1.11 input_reset

The input reset is defined as

```
input_reset name ( portname ) = expression ;
```

where the `=` may be replaced by `<-`. The form specifies that the reset given by *expression* is to be connected to the Verilog port specified in the parentheses. The *name* may be used to associate the reset with methods using it in a `reset_by` argument.

The `=` *expression* may not be omitted.

Example:

```
import "BVI" SyncBit =
module vSyncBit( Clock sClkIn, Reset sRstIn,
                 Clock dClkIn,
                 SyncBitIfc#(a_type) ifc )
```



```
        provisos( Bits#(a_type, awidth)) ;

    default_clock clk() ;
    default_reset rst() ;

    parameter init =  0;

    input_clock clk_src( sCLK ) = sClkIn ;
    input_clock clk_dst( dCLK ) = dClkIn ;

    input_reset (sRST_N)= sRstIn ;

endmodule
```

15.1.12 output_reset

The output reset is the named instance of a reset signal being provided by the BSV module.

The form is

`output_reset` *name* `(` *portname* `);`

Example:

```
import "BVI" SyncResetA =
module vResetAB#(Integer stages ) ( Bool rstIn, ResetGenIfc rstOut ) ;
    default_clock clk(CLK) ;
    no_reset ;
    port IN_RST_N = rstIn ;

    output_reset gen_rst(OUT_RST_N) clocked_by(clk) ;

endmodule
```

15.1.13 ancestor, same family

The statement

`ancestor` *name name*`;`

indicates that the first named clock is an ancestor of the second named clock.

The statement

`same_family` *name name*`;`

indicates that they are in the same family. Note that `ancestor` implies `same_family`, which then need not be explicitly stated.

Example:

```
import "BVI" JoinClock =
module vJoinedClock(Clock c, ClockGenIfc ifc );
    default_clock clk(CLK, CLK_GATE);
    no_reset;
    input_clock c(JOIN_CLK, JOIN_CLK_GATE) = c;
    output_clock gen_clk(NEW_CLK, NEW_CLK_GATE);
```



```
    same_family(clk, c);

    ancestor(gen_clk, clk);
    ancestor(gen_clk, c);
endmodule
```

15.1.14 `method`

The method statement is used to connect methods in a Bluespec interface to the appropriate Verilog wires. The syntax imitates a function prototype, and is as follows:

```
method portname name ( portname , ..., portname )
  enable (portname) ready ( portname) clocked_by (name) reset_by ( name);
```

The first *portname* is the output port for the method, and is optional; the next *name* is the method's name according to the BSV interface definition. The parenthesized list is the input port names corresponding to the method's arguments. There may follow up to four optional annotations (in any order): `enable` (for the enable input port), `ready` (for the ready output port), `clocked_by` (to indicate the clock of the method, otherwise the default clock will be assumed) and `reset_by` (for the reset signal). If the input port list is empty, its parentheses may optionally be omitted, except that they may not be omitted (that is, the empty list `()` must be shown) if any of the optional annotations are present. The BSV types of all the method's arguments and its result (if any) must all be in the `Bits` typeclass.

Example:

```
import "BVI" ClockInverter =
module vClockInverter ( ClockDivider_internal ifc ) ;

    default_clock clk(IN_CLK) ;
    default_reset rst() = noReset ;

    output_clock slowClock(CLK) ;

    //the empty list () is required
    method PREEDGE clockReady()  clocked_by( clk ) reset_by(rst) ;

endmodule
```

Any of the port names may have an attribute attached to them. The allowable attributes are `const`, `reg`, `inhigh`, and `unused`. The attributes are translated into port descriptions.

Example:

```
import "BVI" odd_div_duty50 =
module v_div (I_div);

    method load(flopA, flopB) enable((*inhigh*)EN);
    method clk_out clk_out();
    method reset_out reset_out();

endmodule
```


Note that only action or actionvalue methods can have an `enable` signal. If an `enable` signal has the `inhigh` attribute, the method is assumed to be always enabled, and the compiler will check that this is the case. If no output `ready` signal is specified, the method is assumed to be always ready (and the Verilog code must ensure that this is the case, as the compiler cannot check).

Note that output ports may be shared across methods (and ready signals). This may be used, for example, to give FIFO methods different scheduling annotations (so `FIFOF` can have CF methods for the implicit condition tests, while having SB `notFull` and `notEmpty` methods).

15.1.15 `schedule`

The form is

`schedule (` *name*`,` *name*`, ...)  op (` *name*`,` *name*`, ...);`

The parenthesized lists are sets of names, and the order is unimportant; the parentheses may be omitted if there is only one name in the set. The acceptable list of operators is `CF`, `SB`, `SBR`, `C`, all spelled like that. These operators relate two sets of methods; the specified relation is understood to hold between each element of the first set and each element of the second set.

The meanings of the operators are

`CF` conflict-free

`SB` sequences before

`SBR` sequences before, with range conflict (that is, not composable in parallel)

`C` conflicts

It is an error to specify a scheduling annotation other than `CF` for methods clocked by unrelated clocks. For such methods, `CF` is the default; for methods clocked by related clocks the default is `C`.

Example:

```
import "BVI" FIFO2 =
module vFIFOF2_MC                 (  Clock sClkIn, Reset sRstIn,
                                     Clock dClkIn, Reset dRstIn,
                                     Clock realClock,  Reset realReset,
                                     FIFOF_MC#(a) ifc )
                                  provisos (Bits#(a,sa));

    method          enq( D_IN ) enable(ENQ) clocked_by( clk_src ) reset_by( srst ) ;
    method FULL_N   notFull                 clocked_by( clk_src ) reset_by( srst ) ;
    method FULL_N   i_notFull               clocked_by( clk_src ) reset_by( srst ) ;

    method          deq()       enable(DEQ) clocked_by( clk_dst ) reset_by( drst ) ;
    method D_OUT    first                   clocked_by( clk_dst ) reset_by( drst ) ;
    method EMPTY_N  notEmpty                clocked_by( clk_dst ) reset_by( drst ) ;
    method EMPTY_N  i_notEmpty              clocked_by( clk_dst ) reset_by( drst ) ;

        schedule (enq, notFull, i_notFull) CF (deq, first, notEmpty, i_notEmpty ) ;

        schedule deq CF (i_notEmpty) ;
        schedule enq CF (i_notFull) ;
        schedule (first, notEmpty) CF
                 (first, i_notEmpty, notEmpty) ;
```



```
        schedule first SB deq ;
        schedule (notEmpty) SB (deq) ;
        schedule (notFull) SB (enq) ;

endmodule
```

15.1.16 `path`

The form is

`path` *portname portname* ;

and indicates that there is a combinational path from the first port to the second port. It is an error to specify a path between ports that are connected to methods clocked by unrelated clocks. This would be, by definition, an unsafe clock domain crossing. Note that the compiler assumes that there will be a path between a value or actionvalue method's input parameters and its result, so this need not be explicitly specified.

Example:

```
import "BVI" EnabletoReady =
module mkEn2Rdy3v (En2Rdy3Interv);
       method start() enable(En_start);
       method Rdy_start rdy_start();
       method Result result();
       method Rdy_result rdy_result();
       path (En_start, Rdy_start);
   endmodule
```

References

[Acc04] Accellera. SystemVerilog 3.1a Language Reference Manual: Accellera's Extensions to Verilog (R), 2004. See: www.accelera.org, www.systemverilog.org.

[IEE01] IEEE. IEEE Standard Verilog (R) Hardware Description Language, March 2001. IEEE Std 1364-2001.

[IEE02] IEEE. IEEE Standard VHDL Language Reference Manual, IEEE Std 1076-1993, 2002.

[Ter03] Terese. *Term Rewriting Systems*. Cambridge University Press, 2003.

A Keywords

In general, keywords do not use uppercase letters (the only exception is the keyword `valueOf`). The following are the keywords in BSV (and so they cannot be used as identifiers).

```
Action
ActionValue
action                          endaction
actionvalue                     endactionvalue
ancestor
begin
bit
case                            endcase
clocked_by
default
default_clock
default_reset
dependencies
deriving
determines
else
end
enum
export
for
function                        endfunction
if
import
input_clock
input_reset
instance                        endinstance
interface                       endinterface
let
match
matches
method                          endmethod
module                          endmodule
no_reset
numeric
ouput_reset
output_clock
package                         endpackage
parameter
path
port
provisos
reset_by
return
rule                            endrule
rules                           endrules
same_family
schedule
struct
tagged
```



```
type
typeclass                       endtypeclass
typedef
union
valueof
void
while
```


The following are keywords in SystemVerilog (which includes all the keywords in Verilog). Although most of them are not used in BSV, for compatibility reasons they are not allowed as identifiers in BSV either.

```
alias                          expect                         negedge
always                         export                         new
always_comb                    extends                        nmos
always_ff                      extern                         nor
always_latch                   final                          noshowcancelled
and                            first_match                    not
assert                         for                            notif0
assert_strobe                  force                          notif1
assign                         foreach                        null
assume                         forever                        or
automatic                      fork                           output
before                         forkjoin                       package      endpackage
begin    end                   function    endfunction        packed
bind                           generate    endgenerate        parameter
bins                           genvar                         pmos
binsof                         highz0                         posedge
bit                            highz1                         primitive    endprimitive
break                          if                             priority
buf                            iff                            program      endprogram
bufif0                         ifnone                         property     endproperty
bufif1                         ignore_bins                    protected
byte                           illegal_bins                   pull0
case    endcase                import                         pull1
casex                          incdir                         pulldown
casez                          include                        pullup
cell                           initial                        pulsestyle_onevent
chandle                        inout                          pulsestyle_ondetect
class       endclass           input                          pure
clocking    endclocking        inside                         rand
cmos                           instance                       randc
config      endconfig          int                            randcase
const                          integer                        randsequence
constraint                     interface   endinterface       rcmos
context                        intersect                      real
continue                       join                           realtime
cover                          join_any                       ref
covergroup  endgroup           join_none                      reg
coverpoint                     large                          release
cross                          liblist                        repeat
deassign                       library                        return
default                        local                          rnmos
defparam                       localparam                     rpmos
design                         logic                          rtran
disable                        longint                        rtranif0
dist                           macromodule                    rtranif1
do                             matches                        scalared
edge                           medium                         sequence     endsequence
else                           modport                        shortint
enum                           module       endmodule         shortreal
event                          nand                           showcancelled
```



```
signed                  time                    var
small                   timeprecision           vectored
solve                   timeunit                virtual
specify     endspecify  tran                    void
specparam               tranif0                 wait
static                  tranif1                 wait_order
string                  tri                     wand
strong0                 tri0                    weak0
strong1                 tri1                    weak1
struct                  triand                  while
super                   trior                   wildcard
supply0                 trireg                  wire
supply1                 type                    with
table       endtable    typedef                 within
tagged                  union                   wor
task        endtask     unique                  xnor
this                    unsigned                xor
throughout              use
```


B The Standard Prelude package

This sections describes the type classes, data types, interfaces and functions which are provided by the Standard Prelude package, and therefore always available to the programmer.

The Standard Prelude package is automatically included in all packages, i.e., the programmer does not need to take any special action to use any of the features described here. Please see also Section C for a number of useful libraries that must be explicitly imported into a package in order to use them.

B.1 Type classes

A type class groups related functions and operators and allows for instances across the various datatypes which are members of the typeclass. Hence the function names within a type class are *overloaded* across the various type class members.

A `typeclass` declaration creates a type class. An `instance` declaration defines a datatype as belonging to a type class. A datatype may belong to zero or many type classes. A brief introduction to types and type classes can be found in Section 4, with a more detailed explaination of type classes in Section 14.1.

The Prelude package declares the following type classes:

Prelude Type Classes	
`Bits`	Types that can be converted to bit vectors and back.
`Eq`	Types on which equality is defined.
`Literal`	Types which can be created from integer literals.
`Arith`	Types on which arithmetic operations are defined.
`Ord`	Types on which comparison operations are defined.
`Bounded`	Types with a finite range.
`Bitwise`	Types on which bitwise operations are defined.
`BitReduction`	Types on which bitwise operations on a single operand to produce a single bit result are defined.
`BitExtend`	Types on which extend operations are defined.

B.1.1 Bits

`Bits` defines the class of types that can be converted to bit vectors and back. Membership in this class is required for a data type to be stored in a state, such as a Register or a FIFO, or to be used at a synthesized module boundary. Often instance of this class can be automatically derived using the `deriving` statement (Section 4.3).

```
typeclass Bits #(type a, numeric type n)
    function Bit#(n) pack(a x);
    function a unpack(Bit#(n) x);
endtypeclass
```

Note: the numeric keyword is not required

The functions `pack` and `unpack` are provided to convert elements to `Bit#()` and to convert `Bit#()` elements to another datatype.

Bits Functions	
pack	Converts element a of datatype data_t to a element of datatype Bit#() of size_a.
	function Bit#(size_a) pack(data_t a);

unpack	Converts an element a of datatype Bit#() and size_a into an element with of element type data_t.
	function data_t unpack(Bit#(size_a) a);

B.1.2 Eq

Eq defines the class of types whose values can be compared for equality. Instances of the Eq class are often automatically derived using the deriving statement (Section 4.3).

```
typeclass Eq #(type data_t);
    function Bool \== (data_t x, data_t y);
    function Bool \/= (data_t x, data_t y);
endtypeclass
```

The equality functions == and != are Boolean functions which return a value of True if the equality condition is met. When defining an instance of an Eq typeclass, the \== and \/= notations must be used. If using or referring to the functions, the standard Verilog operators == and != may be used.

Eq Functions	
==	Returns True if x is equal to y.
	function Bool \== (data_t x, data_t y,);

!=	Returns True if x is not equal to y.
	function Bool \/= (data_t x, data_t y,);

B.1.3 Literal

Literal defines the class of types which can be created from integer literals.

```
typeclass Literal #(type data_t);
    function data_t fromInteger(Integer x);
endtypeclass
```

The fromInteger function converts an Integer into an element of datatype data_t. Whenever your write an integer literal in BSV(such as "0" or "1"), there is an implied fromInteger applied to it, which turns the literal inot the type you are using it as (such as Int, UInt, Bit, etc.). By defining an instance of Literal for your own datatypes, you can create vlaues from literals just as for these predefined types.

`Literal` Functions	
`fromInteger`	Converts an element `x` of datatype `Integer` into an element of data type `data_t`
	`function data_t fromInteger(Integer x);`

B.1.4 Arith

`Arith` defines the class of types on which arithmetic operations are defined.

```
typeclass Arith #(type data_t)
  provisos (Literal#(data_t));
    function data_t \+ (data_t x, data_t y);
    function data_t \- (data_t x, data_t y);
    function data_t negate (data_t x);
    function data_t \* (data_t x, data_t y);
endtypeclass
```

The `Arith` functions provide arithmetic operations. When defining an instance of an `Arith` typeclass, the escaped operator names must be used (and the `negate` name for negation). If using or referring to the functions, the standard (non-escaped) Verilog operators can be used.

`Arith` Functions	
`+`	Element `x` is added to element `y`.
	`function data_t \+ (data_t x, data_t y);`

`-`	Element `y` is subtracted from element `x`.
	`function data_t \- (data_t x, data_t y);`

`negate`
 	
`-`
	Change the sign of the number. When using the function the Verilog negate operator, `-`, may be used.

	`function data_t negate (data_t x);`

`*`	Element `x` is multiplied by `y`.
	`function data_t \* (data_t x, data_t y);`

B.1.5 Ord

`Ord` defines the class of types for which an *order* is defined, which allows comparison operations.

```
typpeclass Ord #(type data_t);
    function Bool \<  (data_t x, data_t y);
    function Bool \<= (data_t x, data_t y);
    function Bool \>  (data_t x, data_t y);
    function Bool \>= (data_t x, data_t y);
endtypeclass
```


The `Ord` functions are Boolean functions which return a value of `True` if the comparison condition is met.

`Ord` Functions	
`<`	Returns `True` if `x` is less than `y`.
	`function Bool \<  (data_t x, data_t y);`

`<=`	Returns `True` if `x` is less than or equal to `y`.
	`function Bool \<=  (data_t x, data_t y);`

`>`	Returns `True` if `x` is greater than `y`.
	`function Bool \>  (data_t x, data_t y);`

`>=`	Returns `True` if `x` is greater than or equal to `y`.
	`function Bool \>=  (data_t x, data_t y);`

B.1.6 Bounded

`Bounded` defines the class of types with a finite range and provides functions to define the range.

```
typeclass Bounded #(type data_t);
    data_t minBound;
    data_t maxBound;
endtypeclass
```

The `Bounded` functions `minBound` and `maxBound` define the minimum and maximum values for the type `data_t`.

`Bounded` Functions	
`minBound`	The minimum value the type `data_t` can have.
	`data_t minBound;`

`maxBound`	The maximum value the type `data_t` can have.
	`data_t maxBound;`

B.1.7 Bitwise

`Bitwise` defines the class of types on which bitwise operations are defined.

```
typeclass Bitwise #(type data_t);
    function data_t \& (data_t x1, data_t x2);
    function data_t \| (data_t x1, data_t x2);
    function data_t \^ (data_t x1, data_t x2);
```



```
    function data_t \~^ (data_t x1, data_t x2);
    function data_t \^~ (data_t x1, data_t x2);
    function data_t invert (data_t x1);
    function data_t \<< (data_t x1, Nat x2);
    function data_t \>> (data_t x1, Nat x2);
endtypeclass
```

The `Bitwise` functions compare two operands bit by bit to calculate a result. That is, the bit in the first operand is compared to its equivalent bit in the second operand to calculate a single bit for the result.

`Bitwise` Functions	
`&`	Performs an *and* operation on each bit in `x1` and `x2` to calculate the result.
	`function data_t \& (data_t x1, data_t x2);`
`|`	Performs an *or* operation on each bit in `x1` and `x2` to calculate the result.
	`function data_t | (data_t x1, data_t x2);`
`^`	Performs an *exclusive or* operation on each bit in `x1` and `x2` to calculate the result.
	`function data_t \^ (data_t x1, data_t x2);`
`~^` `^~`	Performs an *exclusive nor* operation on each bit in `x1` and `x2` to calculate the result.
	`function data_t \~^ (data_t x1, data_t x2);` `function data_t \^~ (data_t x1, data_t x2);`
`~` `invert`	Performs a *unary negation* operation on each bit in `x1`. When using this function, the corresponding Verilog operator, `~`, may be used.
	`function data_t invert (data_t x1);`
`<<`	Performs a *left shift* operation of `x1` by the number of bit positions given by `x2`.
	`function data_t \<< (data_t x1, Nat x2);`
`>>`	Performs a *right shift* operation of `x1` by the number of bit positions given by `x2`.
	`function data_t \>> (data_t x1, Nat x2);`

B.1.8 BitReduction

`BitReduction` defines the class of types on which the Verilog bit reduction operations are defined.

```
typeclass BitReduction #(type x, numeric type n)
    function x#(1) reduceAnd (x#(n) d);
    function x#(1) reduceOr (x#(n) d);
    function x#(1) reduceXor (x#(n) d);
    function x#(1) reduceNand (x#(n) d);
    function x#(1) reduceNor (x#(n) d);
    function x#(1) reduceXnor (x#(n) d);
endtypeclass
```

Note: the numeric keyword is not required

The `BitReduction` functions take a sized type and reduce it to one element. The most common example is to operate on a `Bit#()` to produce a single bit result. The first step of the operation applies the operator between the first bit of the operand and the second bit of the operand to produce a result. The function then applies the operator between the result and the next bit of the operand, until the final bit is processed.

Typically the bit reduction operators will be accessed through their Verilog operators. When defining a new instance of the `BitReduction` type class the BSV names must be used. The table below lists both values. For example, the BSV bit reduction *and* operator is `reduceAnd` and the corresponding Verilog operator is `&`.

`BitReduction` Functions	

`reduceAnd`
 	
`&`
	Performs an *and* bit reduction operation between the elements of `d` to calculate the result.

	`function x#(1) reduceAnd (x#(n) d);`

`reduceOr`
 	
`|`
	Performs an *or* bit reduction operation between the elements of `d` to calculate the result.

	`function x#(1) reduceOr (x#(n) d);`

`reduceXor`
 	
`^`
	Performs an *xor* bit reduction operation between the elements of `d` to calculate the result.

	`function x#(1) reduceXor (x#(n) d);`

`reduceNand`
 	
`~&`
	Performs an *nand* bit reduction operation between the elements of `d` to calculate the result.

	`function x#(1) reduceNand (x#(n) d);`

reduceNor ~|	Performs an *nor* bit reduction operation between the elements of `d` to calculate the result.
	`function x#(1) reduceNor (x#(n) d);`

reduceXnor ~^ ^~	Performs an *xnor* bit reduction operation between the elements of `d` to calculate the result.
	`function x#(1) reduceXnor (x#(n) d);`

B.1.9 BitExtend

`BitExtend` defines types on which bit extension operations are defined.

```
typeclass BitExtend #(type x, numeric type n, numeric type m);  // n > m
    function x#(n) zeroExtend (x#(m) d);
    function x#(n) signExtend (x#(m) d);
    function x#(m) truncate (x#(n) d);
endtypeclass
```

The `BitExtend` operations take as input a datatype of size `n` and changes it to a datatype of size `m`.

`BitExtend` Functions	
zeroExtend	Adds extra zero bits to the MSB of argument `d` of size `m` to make the datatype size `n`.
	`function x#(n) zeroExtend (x#(m) d)` `  provisos (Add#(k, n, m));`

signExtend	Adds extra zero bits to the MSB of argument `d` of size `m` to make the datatype size `n` by replicating the sign bit.
	`function x#(n) signExtend (x#(m) d)` `  provisos (Add#(k, n, m));`

truncate	Removes bits from the MSB of argument `d` of size `m` to make the datatype size `n`.
	`function x#(m) truncate (x#(n) d)` `  provisos (Add#(k, m, n));`

B.2 Data Types

Every variable and every expression in BSV has a *type*. Prelude defines the data types which are always available. Each data type may belong to one or more type classes; all functions, modules,

and operators declared for the type class are then defined for the data type. A data type does not have to belong to any type classes.

Data type identifiers must always begin with a capital letter. There are two exceptions; `bit` and `int`, which are predefined for backwards compatibility.

An `instance` declaration defines a data type as belonging to a type class. The following table summarizes which type classes each data type in Prelude belongs to.

Type Classes by Data Types

	Bits	Eq	Literal	Arith	Ord	Bounded	Bitwise	Bit Reduction	Bit Extend
Bit	√	√	√	√	√	√	√	√	√
Int	√	√	√	√	√	√	√	√	√
UInt	√	√	√	√	√	√	√	√	√
Integer		√	√	√	√				
Bool	√	√							
String		√	√	√					
Maybe	√	√							
Action									
Rules									

B.2.1 Bit

To define a value of type Bit:

```
Bit#(type n);
```

Type Classes for `Bit`

	Bits	Eq	Literal	Arith	Ord	Bounded	Bitwise	Bit Reduction	Bit Extend
Bit	√	√	√	√	√	√	√	√	√

`Bit` type aliases

`bit`	The data type `bit` is defined as a single bit. This is a special case of `Bit`.
	`typedef Bit#(1) bit;`

`Nat`	The data type `Nat` is defined as a 32 bit wide bit-vector. This is a special case of `Bit`.
	`typedef Bit#(32) Nat;`

The `Bit` data type provides functions to concatenate and split bit-vectors.

Bit Functions	
{x,y}	Concatenate two bit vectors, x of size n and y of size m returning a bit vector of size k. The Verilog operator { } is used.
	function Bit#(k) bitconcat(Bit#(n) x, Bit#(m) y) provisos (Add#(n, m, k));

split	Split a bit vector into two bit vectors (higher-order bits (n), lower-order bits (m)).
	function Tuple2 #(Bit#(n), Bit#(m)) split(Bit#(k) x) provisos (Add#(n, m, k));

B.2.2 UInt

The UInt type is an unsigned fixed width representation of an integer value.

Type Classes for UInt									
	Bits	Eq	Literal	Arith	Ord	Bounded	Bitwise	Bit Reduction	Bit Extend
UInt	√	√	√	√	√	√	√	√	√

B.2.3 Int

The Int type is a signed fixed width representation of an integer value.

Type Classes for Int									
	Bits	Eq	Literal	Arith	Ord	Bounded	Bitwise	Bit Reduction	Bit Extend
Int	√	√	√	√	√	√	√	√	√

Int type aliases	
int	The data type int is defined as a 32-bit signed integer. This is a special case of Int.
	typedef Int#(32) int;

B.2.4 Integer

The Integer type is a data type used for integer values and functions. Because Integer is not part of the Bits typeclass, the Integer type is used for static elaboration only; all values must be resolved at compile time.

Type Classes for Integer									
	Bits	Eq	Literal	Arith	Ord	Bounded	Bitwise	Bit Reduction	Bit Extend
Integer		√	√	√	√				

Integer Functions	
div	Element x is divided by element y. Any fractional remiander is truncated.
	function Integer div(Integer x, Integer y);

mod	Element x is divided by element y and the remainder is returned as an Integer value.
	function Integer mod(Integer x, Integer y);

exp	The element *base* is raised to the *pwr* power and the exponential value is returned as type Integer, *exp*.
	function Integer exp(Integer base, Integer pwr);

log2	Takes the base 2 logarithm of the Integer x and returns the closest higher Integer value, if the result itself is not an Integer.
	function Integer log2(Integer x) ;

B.2.5 Bool

The Bool type is defined to have two values, True and False.

```
typedef enum {False, True} Bool;
```

Type Classes for Bool									
	Bits	Eq	Literal	Arith	Ord	Bounded	Bitwise	Bit Reduction	Bit Extend
Bool	√	√							

The Bool functions return either a value of True or False.

Bool Functions	
not !	Returns True if x is false, returns False if x is true.
	function Bool not (Bool x);

&&	Returns True if x *and* y are true, else it returns False.
	function Bool \&& (Bool x, Bool y);

||	Returns True if x *or* y is true, else it returns False.
	function Bool \\|| (Bool x, Bool y);

B.2.6 String

Strings are mostly used in system tasks (such as `$display`). The `String` type belongs to the `Eq` type class; strings can be tested for equality and inequality using the `==` and `!=` operators. The `String` type is also part of the `Arith` class, but only the addition (`+`) operator is defined. All other `Arith` operators will produce an error message.

Type Classes for `String`									
	`Bits`	`Eq`	`Literal`	`Arith`	`Ord`	`Bounded`	`Bitwise`	`Bit Reduction`	`Bit Extend`
`String`		√	√	√					

The `strConcat` function is provided for combining `String` values.

`String` Functions	
`strConcat` `+`	Concatenates two elements of type `String`, `x` and `y`.
	`function String strConcat(String x, String y);`

B.2.7 Maybe

The `Maybe` type is used for tagging values as either *Valid* or *Invalid*. If the value is *Valid*, the value contains a datatype `data_t`.

```
typedef union tagged {
    void    Invalid;
    data_t  Valid;
} Maybe #(type data_t) deriving (Eq, Bits);
```

Type Classes for `Maybe`									
	`Bits`	`Eq`	`Literal`	`Arith`	`Ord`	`Bounded`	`Bitwise`	`Bit Reduction`	`Bit Extend`
`Maybe`	√	√							

The `Maybe` data type provides functions to check if the value is *Valid* and to extract the valid value.

`Maybe` Functions	
`fromMaybe`	Extracts the `Valid` value out of a `Maybe` argument. If the tag is `Invalid` the default value, `defaultval`, is returned.
	`function data_t fromMaybe( data_t defaultval,` `Maybe#(data_t) val ) ;`

`isValid`	Returns a value of `True` if the `Maybe` argument is `Valid`.
	`function Bool isValid( Maybe#(data_t) val ) ;`

B.2.8 Action/ActionValue

Any expression that is intended to act on the state of the circuit (at circuit execution time) is called an *action* and has type `Action` or `ActionValue#(a)`. The type parameter `a` represents the type of the returned value.

Type Classes for `Action/ActionValue`									
	Bits	Eq	Literal	Arith	Ord	Bounded	Bitwise	Bit Reduction	Bit Extend
Action									

The types `Action` and `ActionValue` are special keywords, and therefore cannot be redefined. Section 9.6 describes `Action` and `ActionValue` expressions in more detail.

`typedef` ··· *abstract* ··· `struct ActionValue#(type a);`

`ActionValue` type aliases	
`Action`	The `Action` type is a special case of the more general type `ActionValue` where nothing is returned. That is, the returns type is `(void)`.
	`typedef ActionValue#(void) Action;`

`Action` Functions	
`noAction`	An empty `Action`, this is an `Action` that does nothing.
	`function Action noAction();`

B.2.9 Rules

A rule expression has type `Rules` and consists of a collection of individual rule constructs. Rules are first class objects, hence variables of type `Rules` may be created and manipulated. `Rules` values must eventually be added to a module in order to appear in synthesized hardware.

Type Classes for `Rules`									
	Bits	Eq	Literal	Arith	Ord	Bounded	Bitwise	Bit Reduction	Bit Extend
Rules									

The `Rules` data type provides functions to create, manipulate, and combine values of the type `Rules`.

`Rules` Functions	
`emptyRules`	An empty rules variable.
	`function Rules emptyRules();`

`addRules`	Takes rules `r` and adds them into a module. This function may only be called from within a module. The return type `void` indicates that the instantiation does not return anything.
	`function module addRules#(Rules r) (void);`

rJoin	Symmetric union of two sets of rules. A symmetric union means that neither set is implied to have any relation to the other: not more urgent, not execute before, etc.
	`function Rules rJoin(Rules x, Rules y);`

rJoinPreempts	Union of two sets of rules, with rules on the left getting scheduling precedence and blocking the rules on the right. That is, if a rule in set `x` fires, then all rules in set `y` are prevented from firing. This is the same as specifying `descending_urgency` plus a forced conflict.
	`function Rules rJoinPreempts(Rules x, Rules y);`

rJoinDescendingUrgency	
	Union of two sets of rule, with rules in the left having higher urgency. That is, if some rules compete for resources, then scheduling will select rules in set `x` set before set `y`. If the rules do not conflict, no conflict is added; the rules can fire together.
	`function Rules rJoinDescendingUrgency(Rules x, Rules y);`

B.3 Operations on Numeric Types

B.3.1 Size Relationship/Provisos

These classes are used in provisos to express constraints between the sizes of types.

Class	Proviso	Description
`Add`	`Add#(n1,n2,n3)`	Assert $n1 + n2 = n3$
`Max`	`Max#(n1,n2,n3)`	Assert $\max(n1, n2) = n3$
`Log`	`Log#(n1,n2)`	Assert ceiling $\log_2(n1) = n2$.
`Mul`	`Mul#(n1,n2,n3)`	Assert $n1 * n2 = n3$
`Div`	`Div#(n1,n2,n3)`	Assert $n1/n2 = n3$ Only integer results are provided, any fractional remainder is truncated.

Examples of Provisos using size relationships:

```
instance Bits #( Vector#(vsize, element_type), tsize)
   provisos (Bits#(element_type, sizea),
             Mul#(vsize, sizea, tsize));       // vsize * sizea = tsize

function Vector#(vsize1, element_type)
     cons (element_type elem, Vector#(vsize, element_type) vect)
  provisos (Add#(1, vsize, vsize1));           // 1 + vsize = vsize1

function Vector#(mvsize,element_type)
      concat(Vector#(m,Vector#(n,element_type)) xss)
  provisos (Mul#(m,n,mvsize));                 // m * n = mvsize
```


B.3.2 Size Relationship Type Functions

These type functions are used when "defining" size relationships between data types, where the defined value need not (or cannot) be named in a proviso. They may be used in datatype definition statements when the size of the datatype may be calculated from other parameters.

Type Function	Size Relationship	Description
`TAdd`	`TAdd#(n1,n2)`	Calculate $n1 + n2$
`TSub`	`TSub#(n1,n2)`	Calculate $n1 - n2$
`TLog`	`TLog#(n1)`	Calculate ceiling $\log_2(n1)$
`TExp`	`TExp#(n1)`	Calculate 2^{n1}
`TMul`	`TMul#(n1,n2)`	Calculate $n1 * n2$
`TDiv`	`TDiv#(n1,n2)`	Calculate $n1/n2$

Examples using other arithmetic functions:

```
Int#(TAdd#(5,n));                          // defines a signed integer n+5 bits wide
                                           // n must be in scope somewhere

typedef bigsize TAdd#(vsize, 8);           // defines a new type bigsize which
                                           // is 8 bits wider than vsize

typedef Bit#(TLog#(n)) CBToken#(type n); // defines a new parameterized type,
                                           // CBToken, which is log(n) bits wide.

typedef 8 wordsize;                        // blocksize is based on wordsize
typedef TAdd#(wordsize, 1) blocksize;
```

B.4 Registers and Wires

Prelude provides the following interfaces and wires: `Reg`, `RWire`, `Wire`, `BypassWire`, and `PulseWire`.

Interfaces	
Name	Description
`Reg`	Register interface
`RWire`	Similar to a register with output wrapped in a `Maybe` type to indicate validity
`Wire`	Interchangeable with a `Reg` interface, validity of the data is implicit
`BypassWire`	Implementation of the Wire interface where the _write method is always_enabled.
`PulseWire`	`RWire` without any data

B.4.1 `Reg`

The most elementary module available in BSV is the register, which has a `Reg` interface. Registers are polymorphic, i.e., in principle they can hold a value of any type but, of course, ultimately registers store bits. Thus, the provisos on register modules indicate that the type of the value stored in the register must be in the `Bits` type class, i.e., the operations `pack` and `unpack` are defined on the type to convert into bits and back.

Note that all Bluespec registers are considered atomic units, which means that even if one bit is updated (written), then all the bits are considered updated. This prevents multiple rules from updating register fields in an inconsistent manner.

Interfaces and Methods

The Reg interface contains two methods, _write and _read.

```
interface Reg #(type a_type);
    method Action _write(a_type x1);
    method a_type _read();
endinterface: Reg
```

The _write and _read methods are rarely used. Instead, for writes, one uses the non-blocking assignment notation and, for reads, one just mentions the register interface in an expression. Both these notations are described in more detail in Section 8.4.

Reg Interface				
Method			Arguments	
Name	Type	Description	Name	Description
_write	Action	writes a value x1	x1	data to be written
_read	a_type	returns the value of the register		

Modules

Prelude provides three modules to create a register: mkReg creates a register with a given reset value, mkRegU creates a register without any reset, and mkRegA creates a register with a given reset value and with asynchronous reset logic.

mkReg	Make a register with a given reset value. Reset logic is synchronous.
	`module mkReg#(a_type resetval)(Reg#(a_type))` `  provisos (Bits#(a_type, sizea));`

mkRegU	Make a register without any reset; initial simulation value is alternating 01 bits.
	`module mkRegU(Reg#(a_type))` `  provisos (Bits#(a_type, sizea));`

mkRegA	Make a register with a given reset value. Reset logic is asynchronous.
	`module mkRegA#(a_type resetval)(Reg#(a_type))` `  provisos (Bits#(a_type, sizea));`

Functions

Three functions are provided for using registers: asReg returns the register interface instead of the value of the register; readReg reads the value of a register, useful when managing arrays or lists of registers; and writeReg to write a value into a register, also useful when managing arrays or lists of registers.

asReg	Treat a register as a register, i.e., suppress the normal behavior where the interface name implicitly represents the value that the register contains (the _read value). This function returns the register interface, not the value of the register.
	`function Reg#(a_type) asReg(Reg#(a_type) regIfc);`

readReg	Read the value out of a register. Useful for giving as the argument to higher-order array and list functions.
	`function a_type readReg(Reg#(a_type) regIfc);`

writeReg	Write a value into a register. Useful for giving as the argument to higher-order array and list functions.
	`function Action writeReg(Reg#(a_atype) regIfc, a_type din);`

B.4.2 RWire

An `RWire` is a primitive stateless module whose purpose is to allow data transfer between methods and rules without the cycle latency of a register. That is, a `RWire` may be written in a cycle and that value can be read out in the same cycle; values are not stored across clock cycles.

Interfaces and Methods

The `RWire` interface is conceptually similar to a register's interface, but the output value is wrapped in a `Maybe` type. The `wset` method places a value on the wire and sets the valid signal. The read-like method, `wget`, returns the value and a valid signal in a `Maybe` type. The output is only `Valid` if a write has a occurred in the same clock cycle, otherwise the output is `Invalid`.

`RWire` Interface				
Method			Arguments	
Name	Type	Description	Name	Description
`wset`	`Action`	writes a value and sets the valid signal	`datain`	data to be sent on the wire
`wget`	`Maybe`	returns the value and the valid signal		

```
interface RWire#(type element_type) ;
   method Action wset(element_type datain) ;
   method Maybe#(element_type) wget() ;
endinterface: RWire
```

Modules

The `mkRWire` module is proivded to create an `RWire`.

mkRWire	Creates an `RWire`. Output is only valid if a write has occurred in the same clock cycle.
	`module mkRWire(RWire#(element_type))` `  provisos (Bits#(element_type, element_width)) ;`

B.4.3 Wire

The `Wire` interface and module are simular to `RWire`, but the valid bit is hidden from the user and the validity of the read is considered an implicit condition. The `Wire` interface works like the `Reg` interface, so mentioning the name of the wire gets (reads) its contents whenever they're valid, and using `<=` writes the wire. `Wire` is an `RWire` that is designed to be interchangeable with `Reg`. You can replace a `Reg` with an `Wire` without changing the syntax.


```
typedef Reg#(element_type) Wire#(type element_type);
```

Modules

The `mkWire` module is provided to create a `Wire`.

`mkWire`	Creates a `Wire`. Validity of the output is automatically checked as an implicit condition of the read method.
	`module mkWire(Wire#(element_type))` `  provisos (Bits#(element_type, element_width));`

B.4.4 BypassWire

BypassWire is an implementation of the Wire interface where the `_write` method is an `always_enabled` method. The compiler will issue a warning if the method does not appear to be called every clock cycle. The advantage of this tradeoff is that the `_read` method of this interface does not carry any implicit condition (so it can satisfy a `no_implicit_conditions` assertion or an `always_ready` method). See sections 13.1.4 and 13.3.2 for more discussion on the `always_ready`, `always_enabled`, and `no_implicit_conditions` attributes.

`mkBypassWire`	Creates a `BypassWire`. The write method is always_enabled.
	`module mkBypassWire(Wire#(element_type))` `  provisos (Bits#(element_type, element_width));`

B.4.5 PulseWire

Interfaces and Methods

The `PulseWire` interface is an `RWire` without any data. It is useful within rules and action methods to signal other methods or rules in the same clock cycle. Note that because the read method is called `_read`, the register shorthand can be used to get its value without mentioning the method `_read` (it is implicitly added).

`PulseWire` Interface		
Name	Type	Description
`send`	Action	sends a signal down the wire
`_read`	Bool	returns the valid signal

```
interface PulseWire;
  method Action send();
  method Bool _read();
endinterface
```

Modules

The `mkPulsewire` module is provided to create a `PulseWire`.

`mkPulseWire`	The writing to this type of wire is used in rules and action methods to send a single bit to signal other methods or rules in the same clock cycle.
	`module mkPulseWire(PulseWire);`

Counter Example - Using `Reg` and `PulseWire`

```
interface Counter#(type size_t);
    method Bit#(size_t) read();
    method Action load(Bit#(size_t) newval);
    method Action increment();
    method Action decrement();
endinterface

module mkCounter(Counter#(size_t));
    Reg#(Bit#(size_t)) value <- mkReg(0);          // define a Reg

    PulseWire increment_called <- mkPulseWire();   // define the PulseWires used
    PulseWire decrement_called <- mkPulseWire();   // to signal other methods or rules

    // whether rules fire is based on values of PulseWires
    rule do_increment(increment_called && !decrement_called);
        value <= value + 1;
    endrule

    rule do_decrement(!increment_called && decrement_called);
        value <= value - 1;
    endrule

    method Bit#(size_t) read();                 // read the register
        return value;
    endmethod

    method Action load(Bit#(size_t) newval);    // load the register
        value <= newval;                        // with a new value
    endmethod

    method Action increment();                  // sends the signal on the
        increment_called.send();                // PulseWire increment_called
    endmethod

    method Action decrement();                  /  sends the signal on the
        decrement_called.send();                // PulseWire decrement_called
    endmethod
endmodule
```

B.5 Miscellaneous Functions

Compile-time Messages

`error`	Generate a compile-time error message, `s`, and halt compilation.
	`function a_type error(String s);`

`warning`	When applied to a value `v` of type `a`, generate a compile-time warning message, `s`, and continue compilation, returning `v`.
	`function a_type warning(String s, a_type v);`

message	When applied to a value v of type a, generate a compile-time informative message, s, and continue compilation, returning v.
	`function a_type message(String s, a_type v);`

Arithmetic Functions

max	Returns the maximum of two values, x and y.
	`function a_type max(a_type x, a_type y)` `  provisos (Ord#(a_type));`

min	Returns the minimum of two values, x and y.
	`function a_type min(a_type x, a_type y)` `  provisos (Ord#(a_type));`

abs	Returns the absolute value of x.
	`function a_type abs(a_type x)` `  provisos (Arith#(a_type), Ord#(a_type));`

Operations on Functions

These are useful with higher-order list and array functions.

compose	Creates a new function, c, made up of functions, f and g. c(a) = f(g(a))

	`function (function c_type (a_type x0))`
	`  compose(function c_type f(b_type x1),`
	`  function b_type g(a_type x2));`

id	Identity function, returns x when given x. This function is useful when the argument requrires a function which doesn't do anything.
	`function a_type id(a_type x);`

constFn	Constant function, returns x.
	`function a_type constFn(a_type x, b_type y);`

flip	Flips the arguments x and y.
	`function c_type flip (a_type x, b_type y);`

Example - using function `constFn` to set the initial values of the registers in a list:

```
List#(Reg#(Resource)) items <- mapM( constFn(mkReg(initRes)), upto(1,numAdd) );
```

Control Flow Function

`while`	Repeat a function while a predicate holds
	`function a_type while(function Bool pred(a_type x1), function a_type f(a_type x1), a_type x);`

B.6 Environment Values

The `Environment` section of the Prelude contains some value definitions that remain static within a compilation, but may vary between compilations.

Test whether the compiler is generating C.

`genC`	Returns `True` if the compiler is generating C.
	`function Bool genC();`

Test whether the compiler is generating Verilog.

`genVerilog`	Returns `True` if the compiler is generating Verilog.
	`function Bool genVerilog();`

Return the version of the compiler.

`compilerVersion`	Returns a `String` containing the compiler version. This si the same string used with the `-v` flag.
	`String compilerVersion;`

Example:

```
the statement:
     $display("compilerversion = %d", compilerVersion);
produces this output:
     Bluespec Compiler, version 3.8.56 (build 7084, 2005-07-22)
```

Get the current date and time.

`date`	Returns a `String` containing the date.
	`String date;`

Example:

```
the statement:
     $display("date = %s", date);
produces this output:
     "Mon Feb 6 08:39:59 EST 2006"
```


C Libraries

Note: this section is currently under revision to improve the documentation.

Section 12 defined some important primitives. Section B defined the Standard Prelude package, which is automatically imported into every package. This section describes BSV's large and continuously growing collection of libraries that provide common and useful programming idioms and hardware idioms.

To use any of these libraries in a package, the programmer must explicitly import it into the package using an `import` clause.

C.1 Data Structures and Containers

C.1.1 Register File

Package Name

import RegFile :: * ;

Description

This package provides 5-read-port 1-write-port register array modules.

Note: In a design that uses RegFiles, some of the read ports may remain unused. This may generate a warning in various downstream tool. Downstream tools should be instructed to optimize away the unused ports.

Interfaces and Methods

The `RegFile` package defines one interface, `RegFile`. The `RegFile` interface provides two methods, `upd` and `sub`. The `upd` method is an `Action` method used to modify (or update) the value of an element in the register file. The `sub` method (from "sub"script) is a `Value` method which reads and returns the value of an element in the register file. The value returned is of a datatype `data_t`.

Interface Name	Parameter name	Parameter Description	Restrictions
`RegFile`	*index_type*	datatype of the index	must be in the `Bits` class
	data_t	datatype of the element values	must be in the `Bits` class

```
interface RegFile #(type index_t, type data_t);
    method Action upd(index_t addr, data_t d);
    method data_t sub(index_t addr);
endinterface: RegFile
```

Method			Arguments	
Name	Type	Description	Name	Description
`upd`	`Action`	Change or update an element within the register file.	`addr`	index of the element to be changed, with a datatype of `index_t`
			`d`	new value to be stored, with a datatype of `data_t`
`sub`	*data_t*	Read an element from the register file and return it.	`addr`	index of the element, with a datatype of `index_t`

Modules

The `RegFile` package provides three modules: `mkRegFile` creates a RegFile with registers allocated from the `lo_index` to the `hi_index`; `mkRegFileFull` creates a RegFile from the minimum index to the maximum index; and `mkRegFileWCF` creates a RegFile from `lo_index` to `hi_index` for which the reads and the write are scheduled conflict-free. There is a second set of these modules, the `RegFileLoad` variants, which take as an argument a file containing the initial contents of the array.

`mkRegFile`	Create a RegFile with registers allocated from `lo_index` to `hi_index`. `lo_index` and `hi_index` are of the `index_t` datatype and the elements are of the `data_t` datatype.

	`module mkRegFile#( index_t lo_index, index_t hi_index )`
	`  ( RegFile#(index_t, data_t) )`
	`  provisos (Bits#(index_t, size_index),`
	`  Bits#(data_t, size_data));`

`mkRegFileFull`	Create a RegFile from min to max index where the index is of a datatype `index_t` and the elements are of datatype `data_t`. The min and max are specified by the `Bounded` typeclass instance (0 to N-1 for N-bit numbers).

	`module mkRegFileFull#( RegFile#(index_t, data_t) )`
	`  provisos (Bits#(index_t, size_index),`
	`  Bits#(data_t, size_data)`
	`  Bounded#(index_t) );`

`mkRegFileWCF`	Create a RegFile from `lo_index` to `hi_index` for which the reads and the write are scheduled conflict-free. For the implications of this scheduling, see the documentation for `ConfigReg` (Section C.1.5).

	`module mkRegFileWCF#( index_t lo_index, index_t hi_index )`
	`  ( RegFile#(index_t, data_t) )`
	`  provisos (Bits#(index_t, size_index),`
	`  Bits#(data_t, size_data));`

The `RegFileLoad` variants provide the same functionality as `RegFile`, but each constructor function takes an additional file name argument. The file contains the initial contents of the array using the Verilog hex memory file syntax.

`mkRegFileLoad`	Create a RegFile using the file to provide the initial contents of the array.

	`module mkRegFileLoad#`
	`  ( String file, index_t lo_index, index_t hi_index)`
	`  ( RegFile#(index_t, data_t) )`
	`  provisos (Bits#(index_t, size_index),`
	`  Bits#(data_t, size_data));`

`mkRegFileFullLoad`	Create a RegFile from min to max index using the file to provide the initial contents of the array. The min and max are specified by the `Bounded` typeclass instance (0 to N-1 for N-bit numbers).

	`module mkRegFileFullLoad#( String file)`
	`( RegFile#(index_t, data_t))`
	`provisos (Bits#(index_t, size_index),`
	`Bits#(data_t, size_data),`
	`Bounded#(index_t) );`

`mkRegFileWCFLoad`	Create a RegFile from `lo_index` to `hi_index` for which the reads and the write are scheduled conflict-free (see Section C.1.5), using the file to provide the initial contents of the array.

	`module mkRegFileWCFLoad#`
	`( String file, index_t lo_index, index_t hi_index)`
	`( RegFile#(index_t, data_t) )`
	`provisos (Bits#(index_t, size_index),`
	`Bits#(data_t, size_data));`

Examples

Use `mkRegFileLoad` to create Register files and then read the values.

```
Reg#(Cntr) count <- mkReg(0);

// Create Register files to use as inputs in a testbench
RegFile#(Cntr, Fp64) vecA  <- mkRegFileLoad("vec.a.txt", 0, 9);
RegFile#(Cntr, Fp64) vecB  <- mkRegFileLoad("vec.b.txt", 0, 9);

//read the values from the Register files
rule drivein (count < 10);
      Fp64 a = vecA.sub(count);
      Fp64 b = vecB.sub(count);
      uut.start(a, b);
      count <= count + 1;
endrule
```

Verilog Modules

`RegFile` modules correspond to the following Verilog modules, which are found in the Bluespec Verilog library, `$BLUESPECDIR/Verilog/`.

BSV Module Name	Verilog Module Name	Defined in File

`mkRegFile`
`mkRegFileFull`
`mkRegFileWCF`
	RegFile	RegFile.v

`mkRegFileLoad`
`mkRegFileFullLoad`
`mkRegFileWCFLoad`
	RegFileLoad	RegFileLoad.v

C.1.2 FIFO Overview

There are three FIFO packages, `FIFO`, `FIFOF`, and `LevelFIFO`. The following table shows when to use each FIFO, and which methods are in implemented in each FIFO. All FIFOs include the methods `enq`, `deq`, `first`, `clear`. These are referred to as the common methods in the table.

Package Name	Description	Methods

All FIFO packages	common methods in all FIFOs	`enq`
		`deq`
		`first`
		`clear`

FIFO	Implicit full and empty signals	common methods

FIFOF	Explicit full and empty signals	common methods
		`notFull`
		`notEmpty`

LevelFIFO	Indicates the level or current number of items stored in the FIFO	common methods
		`notFull`
		`notEmpty`
		`isLessThan`
		`isGreaterThan`
		`fifoDepth`

Common Methods

The following four methods are provided in all FIFO packages.

Method			Argument	
Name	Type	Description	Name	Description
`enq`	Action	adds an entry to the `FIFO`	x1	variable to be added to the `FIFO` must be of type *element_type*
`deq`	Action	removes first entry from the `FIFO`		
`first`	*element_type*	returns first entry		the entry returned is of *element_type*
`clear`	Action	clears all entries from the `FIFO`		

C.1.3 FIFO and FIFOF packages

Package Name

```
import FIFO :: * ;
import FIFOF :: * ;
```

Description

The FIFO package defines the `FIFO` interface and four module constructors. The `FIFO` package is for fifos with implicit full and empty signals.

The `FIFOF` package defines fifos with explicit full and empty signals.

The standard version of `FIFOF` has fifos with the enq, deq and first methods guarded by the appropriate (notFull or notEmpty) implicit condition for safety and improved scheduling. Unguarded (UG) versions of `FIFOF` are available for the rare cases when implicit conditions are not desired.

Interfaces and methods

Interface Name	Parameter name	Parameter Description	Restrictions
`FIFO`	*element_type*	type of the elements stored in the `FIFO`	must be in `Bits` class
`FIFOF`	*element_type*	type of the elements stored in the `FIFO`	must be in `Bits` class

The four common methods, `enq`, `deq`, `first` and `clear` are provided by the `FIFO` and `FIFOF` interfaces.

Method			Argument	
Name	Type	Description	Name	Description
`enq`	Action	adds an entry to the `FIFO`	x1	variable to be added to the `FIFO` must be of type *element_type*
`deq`	Action	removes first entry from the `FIFO`		
`first`	*element_type*	returns first entry		the entry returned is of *element_type*
`clear`	Action	clears all entries from the `FIFO`		

```
interface FIFO #(type element_type);
    method Action enq(element_type x1);
    method Action deq();
    method element_type first();
    method Action clear();
endinterface: FIFO
```

`FIFOF` provides two additional methods, `notFull` and `notEmpty`.

Method			Argument	
Name	Type	Description	Name	Description
`notFull`	Bool	returns a True value if there is space, you can enqueue an entry into the fifo		
`notEmpty`	Bool	returns a True value if there are elements in the fifo, you can dequeue from the fifo		

```
interface FIFOF #(type element_type);
    method Action enq(element_type x1);
    method Action deq();
    method element_type first();
    method Bool notFull();
    method Bool notEmpty();
    method Action clear();
endinterface: FIFOF
```

Modules

The `FIFO` and `FIFOF` interface types are provided by the module constructors: `mkFIFO`, `mkFIFO1`, `mkSizedFIFO`, and `mkLFIFO`. Each `FIFO` is safe with implicit conditions; it does not allow an `enq` when the `FIFO` is full and does not allow a `deq` or `first` when the `FIFO` is empty. Except for `mkLFIFO`, when the `FIFO` is full it does not allow simultaneous enqueue and dequeue operations.

For creating a FIFOF interface use the "F" version of the module, such as mkFIFO.

Unguarded (UG) versions of FIFOF are available for the rare cases when implicit conditions are not desired. During rule and method processing the implicit conditions for correct fifo operations are NOT considered. That is, with an unguarded fifo, it is possible to enqueue when full, and to dequeue when empty. The Unguarded versions of the FIFOF modules provide the FIFOF interface.

Module Name	BSV Module Declaration *For all modules, **width_any** may be 0*	Description
mkFIFO mkFIFOF mkUGFIFOF	module mkFIFO# (FIFO#(element_type)) provisos (Bits#(element_type, width_any));	FIFO of depth 2.
mkFIFO1 mkFIFOF1 mkUGFIFOF1	module mkFIFO1# (FIFO#(element_type)) provisos (Bits#(element_type, width_any));	FIFO of depth 1
mkSizedFIFO mkSizedFIFOF mkUGSizedFIFOF	module mkSizedFIFO# (Integer n)(FIFO#(element_type)) provisos (Bits#(element_type, width_any));	FIFO of given depth n
mkLFIFO mkLFIFOF mkUGLFIFOF	module mkLFIFO# (FIFO#(element_type)) provisos (Bits#(element_type, width_any));	FIFO of depth 1. deq and enq can be simultaneously applied in the same clock cycle when the FIFO is full.

Example using the FIFO package

This example creates 2 input FIFOs and moves data from the input FIFOs to the output FIFOs.

```
import FIFO::*;


typedef Bit#(24) DataT;

// define a single interface into our example block
interface BlockIFC;
   method Action push1 (DataT a);
   method Action push2 (DataT a);
   method ActionValue#(DataT) get();
```



```
endinterface


module mkBlock1( BlockIFC  );
   Integer fifo_depth = 16;

   // create the first inbound FIFO instance
   FIFO#(DataT) inbound1 <- mkSizedFIFO(fifo_depth);

   // create the second inbound FIFO instance
   FIFO#(DataT) inbound2 <- mkSizedFIFO(fifo_depth);

   // create the outbound instance
   FIFO#(DataT) outbound <- mkSizedFIFO(fifo_depth);

   // rule for enqueue of outbound from inbound1
   // implicit conditions ensure correct behavior
   rule enq1 (True);
      DataT in_data = inbound1.first;
      DataT out_data = in_data;
      outbound.enq(out_data);
      inbound1.deq;
   endrule: enq1

   // rule for enqueue of outbound from inbound2
   // implicit conditions ensure correct behavior
   rule enq2 (True);
      DataT in_data = inbound2.first;
      DataT out_data = in_data;
      outbound.enq(out_data);
      inbound2.deq;
   endrule: enq2

   //Add an entry  to the inbound1 FIFO
   method Action push1 (DataT a);
         inbound1.enq(a);
   endmethod

   //Add an entry  to the inbound2 FIFO
   method Action push2 (DataT a);
         inbound2.enq(a);
   endmethod

   //Remove first value from outbound and return it
   method ActionValue#(DataT) get();
         outbound.deq();
         return outbound.first();
   endmethod
endmodule
```

Verilog Modules

`FIFO` and `FIFOF` modules correspond to the following Verilog modules, which are found in the Bluespec Verilog library, `$BLUESPECDIR/Verilog/`.

BSV Module Name	Verilog Module Names	
mkFIFO mkFIFOF mkUGFIFOF	FIFO2.v	FIFO20.v
mkFIFO1 mkFIFOF1 mkUGFIFOF1	FIFO1.v	FIFO10.v
mkSizedFIFO mkSizedFIFOF mkUGSizedFIFOF	SizedFIFO.v	SizedFIFO0.v
mkLFIFO mkLFIFOF mkUGLFIFOF	FIFOL1.v	FIFOL10.v

C.1.4 Level FIFO

Package Name

import LevelFIFO :: * ;

Description

The BSV `LevelFIFO` library provides enhanced FIFO interfaces and modules which include methods to indicate the level or the current number of items stored in the FIFO. Two versions are included in this package; `FIFOLevelIfc` for a single clock, and `SyncFIFOLevelIfc` with dual clocks, that is separate clocks for the enqueue side and dequeue side.

Interfaces and methods

Interface Name	Parameter name	Parameter Description	Restrictions
`FIFOLevelIfc`	*element_type*	type of the elements stored in the `FIFO`	must be in `Bits` class
	fifoDepth	the depth of the `FIFO`	must be `numeric` type
`SyncFIFOLevelIfc`	*element_type*	type of the elements stored in the `FIFO`	must be in `Bits` class
	fifoDepth	the depth of the `FIFO`	must be `numeric` type

- `FIFOLevelIfc`

  In addition to common FIFO methods, the `FIFOLevelIfc` interface defines methods to compare the current level to `Integer` constants. See Section C.1.2 for details on `enq`, `deq`, `first`, `clear`, `notFull`, and `notEmpty`. Note that `FIFOLevelIfc` interface has a type parameter for the fifoDepth. This numeric type parameter is needed, since the width of the counter is dependant on the FIFO depth.

FIFOLevelIfc				
Method			Argument	
Name	Type	Description	Name	Description
`isLessThan`	Bool	Returns `True` if the depth of the `FIFO` is less than the `Integer` constant, `c1`.	c1	an `Integer` constant
`isGreaterThan`	Bool	Returns `True` if the depth of the `FIFO` is greater than the `Integer` constant, `c1`.	c1	an `Integer` constant

```
interface FIFOLevelIfc#( type element_type, parameter type fifoDepth ) ;
   method Action enq( element_type x1 );
   method Action deq();
   method element_type first();
   method Action clear();

   method Bool notFull ;
   method Bool notEmpty ;

   method Bool isLessThan   ( Integer c1 ) ;
   method Bool isGreaterThan( Integer c1 ) ;

   method UInt#(TLog#(fifoDepth)) maxDepth ;
endinterface
```

- `SyncFIFOLevelIfc`

  In addition to common FIFO methods (Section C.1.2), the `SyncFIFOLevelIfc` interface defines methods to compare the current level to `Integer` constants. Methods are provided for both the source (enqueue side) and destination (dequeue side) clock domains. Note that `SyncFIFOLevelIfc` interface has a type parameter for the fifoDepth. This numeric type parameter is needed, since the width of the counter is dependant on the FIFO depth.

SyncFIFOLevelIfc				
Method			Argument	
Name	Type	Description	Name	Description
`sNotFull`	Bool	Returns `True` if the `FIFO` appears as not full from the source side clock.		
`sNotEmpty`	Bool	Returns `True` if the `FIFO` appears as not empty from the source side clock.		
`dNotFull`	Bool	Returns `True` if the `FIFO` appears as not full from the destination side clock.		
`dNotEmpty`	Bool	Returns `True` if the `FIFO` appears as not empty from the destination side clock.		

`sIsLessThan`	Bool	Returns `True` if the depth of the `FIFO`, as appears on the source side clock, is less than the `Integer` constant, `c1`.	c1	an `Integer` compile-time constant

`sIsGreaterThan`	Bool	Returns `True` if the depth of the `FIFO`, as appears on the source side clock, is greater than the `Integer` constant, `c1`.	c1	an `Integer` compile-time constant.

`dIsLessThan`	Bool	Returns `True` if the depth of the `FIFO`, as appears on the destination side clock, is less than the `Integer` constant, `c1`.	c1	an `Integer` compile-time constant

`dIsGreaterThan`	Bool	Returns `True` if the depth of the `FIFO`, as appears on the destination side clock, is greater than the `Integer` constant, `c1`.	c1	an `Integer` compile-time constant.

```
interface SyncFIFOLevelIfc#( type element_type, parameter type fifoDepth ) ;
   method Action enq ( element_type sendData ) ;
   method Action deq () ;
   method element_type first () ;

   method Bool sNotFull ;
   method Bool sNotEmpty ;
   method Bool dNotFull ;
   method Bool dNotEmpty ;

   // Note that for the following methods, the Integer argument,
   // c1, must be a compile-time constant.
   method Bool sIsLessThan   ( Integer c1 ) ;
   method Bool sIsGreaterThan( Integer c1 ) ;
   method Bool dIsLessThan   ( Integer c1 ) ;
   method Bool dIsGreaterThan( Integer c1 ) ;

   method UInt#(TLog#(fifoDepth)) maxDepth ;
endinterface
```

Modules

- `mkFIFOLevel`

  The `FIFOLevelIfc` interface type is provided by the module constructor `mkFIFOLevel`. Note that the implementation allows any number of `isLessThan` and `isGreaterThan` method calls. Each call with a unique argument adds an additional comparator to the design.

Module Name	BSV Module Declaration `width_any` may be 0

`mkFIFOLevel`	`module mkFIFOLevel (`
	`FIFOLevelIfc#(element_type, fifoDepth) )`
	`provisos( Bits#(element_type, width_element )) ;`

- `mkSyncFIFOLevel`

  The module `mkSyncFIFOLevel` is dual clock FIFO, where enqueue and dequeue methods are in separate clocks domains – `sClkIn` and `dClkIn` respectively. Because of the synchronization latency, the flag indicators will not necessarily be identical between the source and the destination clocks. Note however, that the `sNotFull` and `dNotEmpty` flags always give proper (pessimistic) indications for the safe use of `enq` and `deq` methods; these are automatically included as implicit condition in the `enq` and `deq` (and `first`) methods.

Module Name	BSV Module Declaration `width_any` may be 0

`mkSyncFIFOLevel`	`module mkSyncFIFOLevel (`
	`Clock sClkIn, Reset sRstIn,`
	`Clock dClkIn,`
	`SyncFIFOLevelIfc#(element_type, fifoDepth) ifc )`
	`provisos( Bits#(element_type, width_element) ) ;`

Example

The following example shows the use of `SyncLevelFIFO` as a way to collect data into a FIFO, and then send it out in a burst mode. The portion of the design shown, waits until the FIFO is almost full, and then sets a register, `burstOut` which indicates that the FIFO should dequeue. When the FIFO is almost empty, the flag is cleared, and FIFO fills again.

```
. . .
// Define a fifo of Int(#23) with 128 entries
SyncFIFOLevelIfc#(Int#(23),128) fifo <- mkSyncFIFOLevel(sclk, rst, dclk ) ;

// Define some constants
let sFifoAlmostFull = fifo.sIsGreaterThan( 120 ) ;
let dFifoAlmostFull = fifo.dIsGreaterThan( 120 ) ;
let dFifoAlmostEmpty = fifo.dIsLessThan( 12 ) ;

// a register to indicate a burst mode
Reg#(Bool)  burstOut <- mkReg( False, clocked_by (dclk)) ;

. . .
// Set and clear the burst mode depending on fifo status
rule timeToDeque( dFifoAlmostFull && ! burstOut ) ;
   burstOut <= True ;
endrule

rule timeToStop ( dFifoAlmostEmpty && burstOut );
   burstOut <= False ;
```



```
endrule

rule moveData ( burstOut ) ;
   let dataToSend = fifo.first ;
   fifo.deq ;
   ...
   bursting.send( dataToSend ) ;
endrule
```

C.1.5 ConfigReg

Package Name

import ConfigReg :: * ;

Description

The `ConfigReg` package provides a way to create registers where each update clobbers the current value, but the precise timing of updates is not important. These registers are identical to the `mkReg` registers except that their scheduling annotations allows reads and writes to occur in either order during rule execution.

Rules which fire during the clock cycle where the register is written read a stale value (that is the value from the beginning of the clock cycle) regardless of firing order and writes which have occurred during the clock cycle. Thus if rule `r1` writes to a ConfigReg `cr` and rule `r2` reads `cr` later in the same cycle, the old or stale value of `cr` is read, not the value written in `r1`. If a standard register is used instead, rule `r2`'s execution will be blocked by `r1`'s execution or the scheduler may create a different rule execution order.

The hardware implementation is identical for the more common registers (`mkReg`, `mkRegU` and `mkRegA`), and the module constructors parallel these as well.

Interfaces

The `ConfigReg` interface is an alias of the `Reg` interface (sections 12.5 and B.4.1).

```
typedef Reg#(a_type) ConfigReg #(type a_type);
```

Modules

The `ConfigReg` package provides three modules; `mkConfigReg` creates a register with a given reset value and synchronous reset logic, `mkConfigRegU` creates a register without any reset, and `mkConfigRegA` creates a register with a given reset value and asynchronous reset logic.

`mkConfigReg`	Make a register with a given reset value. Reset logic is synchronous
	`module mkConfigReg#(a_type resetval)(Reg#(a_type))` `  provisos (Bits#(a_type, sizea));`

`mkConfigRegU`	Make a register without any reset; initial simulation value is alternating 01 bits.
	`module mkConfigRegU(Reg#(a_type))` `  provisos (Bits#(a_type, sizea));`

mkConfigRegA	Make a register with a given reset value. Reset logic is asynchronous.
	`module mkConfigRegA#(a_type, resetval)(Reg#(a_type))` `  provisos (Bits#(a_type, sizea));`

C.1.6 List

Package Name

import List :: * ;

Description

The `List` package defines a data type and functions which create and operate on this data type. Lists are similar to Vectors, but are used when the number of items on the list may vary at compile-time or need not be strictly enforced by the type system. All elements of a list must be of the same type. The list type is defined as a tagged union as follows.

```
typedef union tagged {
    void Nil;
    struct {
        a         head;
        List #(a) tail;
    } Cons;
} List #(type a);
```

A list is tagged `Nil` if it has no elements, otherwise it is tagged `Cons`. `Cons` is a structure of a single element and the rest of the list.

Lists are most often used during static elaboration (compile-time) to manipulate collections of objects. Since `List#(element_type)` is not in the `Bits` typeclass, lists cannot be stored in registers or other dynamic elements. However, one can have a list of registers or variables corresponding to hardware functions.

Functions for Creating and Generating Lists

cons	Adds an element to a list. The new element will be at the 0th position.
	`function List#(element_type)` `  cons (element_type x, List#(element_type) xs);`

upto	Create a list of Integers counting up over a range of numbers, from m to n. If m > n, an empty list (`Nil`) will be returned.
	`List#(Integer) upto(Integer m, Integer n);`

replicate	Generate a list of n elements by replicating the given argument, `elem`.
	`function List#(element_type)` `  replicate(Integer n, element_type elem);`

append	Append two lists, returning the combined list. The elements of both lists must be the same datatype, `element_type`. The combined list will contain all the elements of `xs` followed in order by all the elements of `ys`.
	`function List#(element_type)` `  append(List#(element_type) xs, List#(element_type) ys);`

concat	Append (*con*catenate) many lists, that is a list of lists, into one list.
	`function List# (element_type)` `  concat (List#(List#(element_type)) xss);`

Examples - Creating and Generating Lists

Create a new list, `my_list`, of elements of datatytpe `Int#(32)` which are undefined

```
List #(Int#(32)) my_list;
```

Create a list, `my_list`, of five 1's

```
List #(Int #(32)) my_list = replicate (5,32'd1);

//my_list = {1,1,1,1,1}
```

Create a new list using the `upto` function

```
List #(Integer) my_list2 = upto (1, 5);

//my_list2 = {1,2,3,4,5}
```

Functions for Extracting Elements and Sub-Lists

[i]	The square-bracket notation is available to extract an element from a list. Extracts the ith element, where the first element is [0]. Index i must be an indexable type; `Integer`, `Bit#(n)`, `Int` or `UInt`.
	`anyList[i]`

select	The select function is another form of the subscript notation ([i]). It may be necessary when the compiler can't determine the type of the subscript i.
	`function element_type` `  select(List#(element_type) alist, idx_type index);`

update	Update an element in a list returning a new list.

	`function List#(element_type)`
	`  update(List#(element_type) alist,`
	`  idx_type index,`
	`  element_type newElem)`
	`  provisos(Eq#(idx_type), Literal#(idx_type));`

oneHotSelect	Select a list element with a Boolean list. The Boolean list should have exactly one element that is `True`, otherwise the result is undefined. The returned element is the one in the corresponding position to the `True` element in the Boolean list.

	`function element_type`
	`  oneHotSelect (List#(Bool) bool_list,`
	`  List#(element_type) alist);`

head	Extract the first element of a list. The input list must have at least 1 element, or an error will be returned.
	`function element_type head (List#(element_type) listIn);`

last	Extract the last element of a list. The input list must have at least 1 element, or an error will be returned.
	`function element_type last (List#(element_type) alist);`

tail	Remove the head element of a list leaving the remaining elements in a smaller list. The input list must have at least 1 element, or an error will be returned.
	`function List#(element_type) tail (List#(element_type) alist);`

init	Remove the last element of a list the remaining elements in a smaller list. The input list must have at least one element, or an error will be returned.
	`function List#(element_type) init (List#(element_type) alist);`

`take`	Take a number of elements from a list starting from index 0. The number to take is specified by the argument `n`. If the argument is greater than the number of elements on the list, the function stops taking at the end of the list and returns the entire input list.
	`function List#(element_type)` `  take (Integer n, List#(element_type) alist);`

`drop`	Drop a number of elements from a list starting from index 0. The number to drop is specified by the argument `n`. If the argument is greater than the number of elements on the list, the entire input list is dropped, returning an empty list.
	`function List#(element_type)` `  drop (Integer n, List#(element_type) alist);`

`filter`	Create a new list from a given list where the new list has only the elements which satisfy the predicate function.

	`function List#(element_type)`
	`  filter (function Bool pred(element_type),`
	`  List#(element_type) alist);`

`takeWhile`	Returns the first set of elements of a list which satisfy the predicate function.

	`function List#(element_type)`
	`  takeWhile (function Bool pred(element_type x),`
	`  List#(element_type) alist);`

`takeWhileRev`	Returns the last set of elements on a list which satisfy the predicate function.

	`function List#(element_type)`
	`  takeWhileRev (function Bool pred(element_type x),`
	`  List#(element_type) alist);`

`dropWhile`	Removes the first set of elements on a list which satisfy the predicate function, returning a list with the remaining elements.

	`function List#(element_type)`
	`  dropWhile (function Bool pred(element_type x),`
	`  List#(element_type) alist);`

dropWhileRev	Removes the last set of elements on a list which satisfy the predicate function, returning a list with the remaining elements.
	function List#(element_type) dropWhileRev (function Bool pred(element_type x), List#(element_type) alist);

Examples - Extracting Elements and Sub-Lists

Extract the element from a list, my_list, at the position of index.

```
//my_list = {1,2,3,4,5}, index = 3

newvalue = select (my_list, index);

//newvalue = 4
```

Extract the zeroth element of the list my_list.

```
//my_list = {1,2,3,4,5}

newvalue = head(my_list);

//newvalue = 1
```

Create a list, my_list2, of size 4 by removing the head (zeroth) element of the list my_list1.

```
//my_list1 is a list with 5 elements, {0,1,2,3,4}

List #(Int #(32)) my_list2 = tail (my_list1);
List #(Int #(32)) my_list3 = tail(tail(tail(tail(tail(my_list1);

//my_list2 = {1,2,3,4}
//my_list3 = Nil
```

Create a 2 element list, my_list2, by taking the first two elements of the list my_list1.

```
//my_list1 is list with 5 elements, {0,1,2,3,4}
List #(Int #(4)) my_list2 = take (2,my_list1);

//my_list2 = {0,1}
```

The number of elements specified to take in take can be greater than the number of elements on the list, in which case the entire input list will be returned.

```
//my_list1 is list with 5 elements, {0,1,2,3,4}
List #(Int #(4)) my_list2 = take (7,my_list1);

//my_list2 = {0,1,2,3,4}
```

Select an element based on a boolean list.

```
//my_list1 is a list of unsigned integers, {1,2,3,4,5}
//my_list2 is a list of Booleans, only one value in my_list2 can be True.
//my_list2 = {False, False, True, False,False, False, False}.

result = oneHotSelect (my_list2, my_list1));

//result = 3
```


Create a list by removing the initial segment of a list that meets a predicate.

```
//the predicate function is a < 2

function Bool lessthan2 (Int #(4) a);
    return (a < 2);
endfunction

//my_list1 = {0,1,2,0,1,7,8}

 List #(Int #(4)) my_result = (dropWhile(lessthan2, my_list1));

//my_result = {2,0,1,7,8}
```

Tests on Lists

`==` `!=`	Lists can be compared for equality if the elements in the list can be compared.
	`instance Eq #( List#(element_type) )` `  provisos( Eq#( element_type ) ) ;`

`elem`	Check if a value is an element in a list.
	`function Bool elem (element_type x, List#(element_type) alist )` `  proviso (Eq#(element_type));`

`isNull`	Check if a list is empty. Returns `True` if the list is empty, that is if there are zero elements.
	`function Bool isNull (element_type x, List#(element_type) alist );`

`length`	Determine the length of a list. Can be done at elaboration time only.
	`function Integer length (List#(element_type) alist );`

`any`	Test if a predicate holds for any element of a list.
	`function Bool any(function Bool pred(element_type x1),` `  List#(element_type) alist );`

`all`	Test if a predicate holds for all elements of a list.
	`function Bool all(function Bool pred(element_type x1), List#(element_type) alist );`

`or`	Combine all elements in a Boolean list with a logical or.
	`function Bool or (List# (Bool) bool_list);`

`and`	Combine all elements in a Boolean list with a logical and.
	`function Bool and (List# (Bool) bool_list);`

Examples - Tests on Lists

Test that all elements of the list `my_list1` are positive integers

```
function Bool isPositive (Int #(32) a);
     return (a > 0)
endfunction

// function isPositive checks that "a" is a  positive integer
// if my_list1 has n elements, n instances of the predicate
// function isPositive will be generated.

if (all(isPositive, my_list1))
   $display ("List contains all negative values");
```

Test if any elements in the list are positive integers.

```
// function isPositive checks that "a" is a  positive integer
// if my_list1 has n elements, n instances of the predicate
// function isPositive will be generated.

if (any(pos, my_list1))
   $display ("List contains some negative values");
```

Check if the integer 5 is in `my_list`

```
// if my_list contains n elements, elem will generate n copies
// of the eq test
if (elem(5,my_list))
   $display ("List contains the integer 5");
```

Combining Lists with Zip Functions

The family of `zip` functions takes two or more lists and combines them into one list of `Tuples`. Several variations are provided for different resulting `Tuples`. All variants can handle input lists of different sizes. The resulting lists will be the size of the smallest list. Tuples are described in Section 12.4.

`zip`	Combine two lists into a list of Tuples.

	`function List#(Tuple2 #(a_type, b_type))`
	`zip( List#(a_type) lista,`
	`List#(b_type) listb);`

`zip3`	Combine 3 lists into a list of Tuple3.

	`function List#(Tuple3 #(a_type, b_type, c_type))`
	`zip3( List#(a_type) lista,`
	`List#(b_type) listb,`
	`List#(c_type) listc);`

`zip4`	Combine 4 lists into a list of Tuple4.

	`function List#(Tuple4 #(a_type, b_type, c_type, d_type))`
	`zip4( List#(a_type) lista,`
	`List#(b_type) listb,`
	`List#(c_type) listc,`
	`List#(d_type) listd);`

`unzip`	Separate a list of pairs (i.e. a `Tuple2#(a,b)`) into a pair of two lists.
	`function Tuple2#(List#(a_type), List#(b_type))` `unzip(List#(Tuple2 #(a_type, b_type)) listab);`

Examples - Combining Lists with Zip

Combine two lists into a list of Tuples

```
//my_list1 is a list of elements {0,1,2,3,4,5,6,7}
//my_list2 is a list of elements {True,False,True,True,False}

my_list3 = zip(my_list1, my_list2);

//my_list3 is a list of Tuples {(0,True),(1,False),(2,True),(3,True),(4,False)}
```

Separate a list of pairs into a Tuple of two lists

```
//my_list is a list of pairs {(0,5),(1,6),(2,7),(3,8),(4,9)}

Tuple2#(List#(Int#(5)),List#(Int#(5))) my_list2 = unzip(my_list);

//my_list2 is ({0,1,2,3,4},{5,6,7,8,9})
```

Mapping Functions over Lists

A function can be applied to all elements of a list, using high-order functions such as `map`. These functions take as an argument a function, which is applied to the elements of the list.

`map`	Map a function over a list, returning a new list of results.
	`function List#(b_type) map (function b_type func(a_type), List#(a_type) alist);`

Example - Mapping Functions over Lists

Consider the following code example which applies the `zeroExtend` function to each element of `alist` creating a new list, `resultlist`.

```
List#(Bit#(5))   alist;
List#(Bit#(10))  resultlist;
...
resultlist = map( zeroExtend, alist ) ;
```

This is equivalent to saying:

```
for (Integer i=0; i<13; i=i+1)
    resultlist[i] = zeroExtend(alist[i]);
```

Map a negate function over a list

```
//my_list1 is a list of 5 elements {0,1,2,3,4}
//negate is a function which makes each element negative

List #(Int #(32)) my_list2 = map (negate, my_list1);

//my_list2 is a list of 5 elements {0,-1,-2,-3,-4}
```

ZipWith Functions

The zipWith functions combine two or more lists with a function and generate a new list. These functions combine features of map and zip functions.

`zipWith`	Combine two lists with a function. The lists do not have to have the same number of elements.
	`function List#(c_type) zipWith (function c_type func(a_type x, b_type y), List#(a_type) listx, List#(b_type) listy );`

`zipWith3`	Combine three lists with a function. The lists do not have to have the same number of elements.
	`function List#(d_type) zipWith3(function d_type func(a_type x, b_type y, c_type z), List#(a_type) listx, List#(b_type) listy, List#(c_type) listz );`

zipWith4	Combine four lists with a function. The lists do not have to have the same number of elements.
	function List#(e_type) zipWith4 (function e_type func(a_type x, b_type y, c_type z, d_type w), List#(a_type) listx, List#(b_type) listy, List#(c_type) listz List#(d_type) listw);

Examples - ZipWith

Create a list by applying a function over the elements of 3 lists.

```
//the function add3 adds 3 values
function Int#(8) add3 (Int #(8) a,Int #(8) b,Int #(8) c);
    Int#(8) d = a + b +c ;
    return(d);
endfunction

//Create the list my_list4 by adding the ith element of each of
//3 lists (my_list1, my_list2, my_list3) to generate the ith
//element of my_list4.

//my_list1 = {0,1,2,3,4}
//my_list2 = {5,6,7,8,9}
//my_list3 = {10,11,12,13,14}

List #(Int #(8)) my_list4 = zipWith3(add3, my_list1, my_list2, my_list3);

//my_list4 = {15,18,21,24,27}

// This is equivalent to saying:
   for (Integer i=0; i<5; i=i+1)
      my_list4[i] = my_list1[i] + my_list2[i] + my_list3[i];
```

Fold Functions

The `fold` family of functions reduces a list to a single result by applying a function over all its elements. That is, given a list of `element_type`, $L_0, L_1, L_2, ..., L_{n-1}$, a seed of type `b_type`, and a function `func`, the reduction for `foldr` is given by

$$func(L_0, func(L_1, ..., func(L_{n-2}, func(L_{n-1}, seed))));$$

Note that `foldr` start processing from the highest index position to the lowest, while foldl starts from the lowest index (zero), i.e.,

$$func(...(func(func(seed, L_0), L_1), ...)L_{n-1})$$

`foldr`	Reduce a list by applying a function over all its elements. Start processing from the highest index to the lowest.

	`function b_type foldr(b_type function func(a_type x, b_type y),`
	`b_type seed,`
	`List#(a_type) alist);`

`foldl`	Reduce a list by applying a function over all its elements. Start processing from the lowest index (zero).

	`function b_type foldl (b_type function func(b_type y, a_type x),`
	`b_type seed,`
	`List#(a_type) alist);`

The functions `foldr1` and `foldl1` use the first element as the seed. This means they only work on lists of at least one element. Since the result type will be the same as the element type, there is no `b_type` as there is in the `foldr` and `foldl` functions.

`foldr1`	`foldr` function for a non-zero sized list. Uses element L_{n-1} as the seed. List must have at least 1 element.

	`function element_type foldr1`
	`(element_type function func(element_type x, element_type y),`
	`List#(element_type) alist);`

`foldl1`	`foldl` function for a non-zero sized list. Uses element L_0 as the seed. List must have at least 1 element.

	`function element_type foldl1`
	`(element_type function func(element_type y, element_type x),`
	`List#(element_type) alist);`

The `fold` function also operates over a non-empty list, but processing is accomplished in a binary tree-like structure. Hence the depth or delay through the resulting function will be $O(log_2(lsize)$ rather than $O(lsize)$.

`fold`	Reduce a list by applying a function over all its elements, using a binary tree-like structure. The function returns the same type as the arguments.

	`function element_type fold`
	`(element_type function func(element_type y, element_type x),`
	`List#(element_type) alist );`

Example - Folds

```
// my_list1 is a list of five integers {1,2,3,4,5}
```



```
// \+ is a function which returns the sum of the elements

my_sum =  foldr (\+ , 0, my_list1));

// my_sum = 15
```

Use fold to find the element with the maximum value

```
// my_list1 is a list of five integers  {2,45,5,8,32}

my_max = fold (max, my_list1);

// my_max = 45
```

Scan Functions

The `scan` family of functions applies a function over a list, creating a new List result. The `scan` function is similar to `fold`, but the intermediate results are saved and returned in a list, instead of returning just the last result. The result of a `scan` function is a list. That is, given a list of `element_type`, $L_0, L_1, ..., L_{n-1}$, an initial value `initb` of type `b_type`, and a function `func`, application of the `scanr` functions creates a new list W, where

$$\begin{aligned} W_n &= init; \\ W_{n-1} &= func(L_{n-1}, W_n); \\ W_{n-2} &= func(L_{n-2}, W_{n-1}); \\ &... \\ W_1 &= func(L_1, W_2); \\ W_0 &= func(L_0, W_1); \end{aligned}$$

`scanr`	Apply a function over a list, creating a new list result. Processes elements from the highest index position to the lowest, and fills the resulting list in the same way. The result list is one element longer than the input list.

	`function List#(b_type)`
	`  scanr(function b_type func(a_type x1, b_type x2),`
	`  b_type initb,`
	`  List#(a_type) alist);`

`sscanr`	Apply a function over a list, creating a new list result. The elements are processed from the highest index position to the lowest. Drops the W_n element from the result. Input and output lists are the same size.

	`function List#(b_type)`
	`  sscanr(function b_type func(a_type x1, b_type x2),`
	`  b_type initb,`
	`  List#(a_type) alist );`

The `scanl` function creates the resulting list in a similar way as `scanr` except that the processing happens from the zeroth element up to the nth element.

$$\begin{aligned} W_0 &= init; \\ W_1 &= func(W_0, L_0); \\ W_2 &= func(W_1, L_1); \\ &... \\ W_{n-1} &= func(W_{n-2}, L_{n-2}); \\ W_n &= func(W_{n-1}, L_{n-1}); \end{aligned}$$

The `sscanl` function drops the first result, *init*, shifting the result index by one.

`scanl`	Apply a function over a list, creating a new list result. Processes elements from the zeroth element up to the nth element. The result list is 1 element longer than the input list.

	`function List#(a_type)`
	`  scanl(function a_type func(a_type x1, b_type x2),`
	`  a_type inita,`
	`  List#(b_type) alist);`

`sscanl`	Apply a function over a list, creating a new list result. Processes elements from the zeroth element up to the nth element. Drop the first result, *init*, shifting the result index by one. The length of the input and output lists are the same.

	`function List#(a_type)`
	`  sscanl(function a_type func(a_type x1, b_type x2),`
	`  a_type inita,`
	`  List#(b) alist );`

Examples - Scan

Create a list of factorials

```
//the function my_mult multiplies element a by element b
function Bit #(16) my_mult (Bit #(16) b, Bit #(8) a);
   return (zeroExtend (a) * b);
endfunction

// Create a list of factorials by multiplying each input list element
// by the previous product (the output list element), to generate
// the next product.  The seed is a Bit#(16) with a value of 1.
// The elements are processed from the zeroth element up to the nth element.
//my_list1 = {1,2,3,4,5,6,7}

List #(Bit #(16)) my_list2 = scanl (my_mult, 16'd1, my_list1);

//my_list2 = {1,1,2,6,24,120,720,5040}
```

List to List Functions

rotate	Move the first element to the last and shift each element to the left.
	`function List#(element_type) rotate (List#(element_type) alist);`

rotateR	Move last element to the beginning and shift each element to the right.
	`function List#(element_type) rotateR (List#(element_type) alist);`

reverse	Reverse element order
	`function List#(element_type) reverse(List#(element_type) alist);`

transpose	Matrix transposition of a list of lists.
	`function List#(List#(element_type))` `transpose ( List#(List#(element_type)) matrix );`

Examples - List to List Functions

Create a list by moving the last element to the first, then shifting each element to the right.

```
//my_list1 is a List of elements with values {1,2,3,4,5}

my_list2 = rotateR (my_list1);

//my_list2 is a List of elements with values {5,1,2,3,4}
```

Create a list which is the reverse of the input List

```
//my_list1 is a List of elements {1,2,3,4,5}

my_list2 = reverse (my_list1);

//my_list2 is a List of elements {5,4,3,2,1}
```

Use transpose to create a new list

```
//my_list1 has the values:
//{{0,1,2,3,4},{5,6,7,8,9},{10,11,12,13,14}}

my_list2 = transpose(my_list1);

//my_list2 has the values:
//{{0,5,10},{1,6,11},{2,7,12},{3,8,13},{4,9,14}}
```

Monadic Operations

Within Bluespec, there are some functions which can only be invoked in certain contexts. Two common examples are: `ActionValue`, and module instantiation. ActionValues can only be invoked

within an `Action` context, such as a rule block or an Action method, and can be considered as two parts - the action and the value. Module instantiation can similarly be considered, modules can only be instantiated in the module context, while the two parts are the module instantiation (the action performed) and the interface (the result returned). These situations are considered *monadic*.

When a monadic function is to be applied over a list using map-like functions such as `map, zipWith`, or `replicate`, the monadic versions of these functions must be used. Moreover, the context requirements of the applied function must hold.

`mapM`	Takes a monadic function and a list, and applies the function to all list elements returning the list of corresponding results.

	`function m#(List#(b_type))`
	`  mapM ( function m#(b_type) func(a_type x),`
	`  List#(a_type) alist )`
	`  provisos (Monad#(m));`

`mapM_`	Takes a monadic function and a list, applies the function to all list elements, and throws away the resulting list leaving the action in its context.
	`function m#(List#(b_type) mapM_(m#(b_type) c_type)` `  provisos (Monad#(m));`

`zipWithM`	Take a monadic function (which takes two arguments) and two lists; the function applied to the corresponding element from each list would return an action and result. Perform all those actions and return the list of corresponding results.

	`function m#(List#(c_type))`
	`  zipWithM( function m#(c_type) func(a_type x, b_type y),`
	`  List#(a_type) alist,`
	`  List#(b_type) blist )`
	`  provisos (Monad#(m));`

`zipWith3M`	Same as `zipWithM` but combines three lists with a function. The function is applied to the corresponding element from each list and returns an action and the list of corresponding results.

	`function m#(List#(d_type))`
	`  zipWith3M( function m#(d_type)`
	`  func(a_type x, b_type y, c_type z),`
	`  List#(a_type) alist ,`
	`  List#(b_type) blist,`
	`  List#(c_type) clist )`
	`  provisos (Monad#(m));`

replicateM	Generate a list of elements by using the given monadic value repeatedly.

	`function m#(List#(element_type))`
	`  replicateM( Integer n, m#(element_type) c)`
	`  provisos (Monad#(m));`

Miscellaneous Functions on Lists

joinActions	Join a number of actions together.
	`function Action joinActions (List#(Action) list_actions);`

joinRules	Join a number of rules together.
	`function Rules joinRules (List#(Rules) list_rules);`

mapAccumL	Map a function, but pass an accumulator from head to tail.

	`function Tuple2 #(a_type, List#(c_type))`
	`  mapAccumL (function Tuple2 #(a_type, c_type)`
	`  func(a_type x, b_type y),a_type x0,`
	`  List#(b_type) alist );`

mapAccumR	Map a function, but pass an accumulator from tail to head.

	`function Tuple2 #(a_type, List#(c_type))`
	`  mapAccumR(function Tuple2 #(a_type, c_type)`
	`  func(a_type x, b_type y),a_type x0,`
	`  List#(b_type) alist );`

mapPairs	Map a function over a list consuming two elements at a time. Any straggling element is processed by the second function.

	`function List#(b_type)`
	`  mapPairs (`
	`  function b_type func1(a_type x, a_type y),`
	`  function b_type func2(a_type x),`
	`  List#(a_type) alist );`

Examples - Miscellaneous Functions on Lists

Create a new list using `mapPairs`. The function `sum` is applied to each pair of elements (the first and second, the third and fourth, etc.). If there is an uneven number of elements, the function `pass` is applied to the remaining element.


```
//sum is defined as c = a+b
function Int#(4) sum (Int #(4) a,Int #(4) b);
     Int#(4) c = a + b;
       return(c);
endfunction

//pass is defined as a
function Int#(4) pass (Int #(4) a);
       return(a);
endfunction

//my_list1 has the elements {0,1,2,3,4}

my_list2 = mapPairs(sum,pass,my_list1);

//my_list2 has the elements {1,5,4}
//my_list2[0] = 0 + 1
//my_list2[1] = 2 + 3
//my_list2[3] = 4
```

C.1.7 Vector

Package Name

import Vector :: * ;

Description

The `Vector` package defines an abstract data type which is a container of a specific length, holding elements of one type. Functions which create and operate on this type are also defined within this package. Because it is abstract, there are no constructors available for this type (like `Cons` and `Nil` for the `List` type).

```
typedef struct Vector#(type numeric vsize, type element_type);
```

Here, the type variable `element_type` represents the type of the contents of the elements while the numeric type variable `vsize` represents the length of the vector.

If the elements are in the `Bits` class, then the vector is as well. Thus a vector of these elements can be stored into Registers or FIFOs; for example a Register holding a vector of type `int`. Note that a vector can also store abstract types, such as a vector of `Rules` or a vector of `Reg` interfaces. These are useful during static elaboration although they have no hardware implementation.

Typeclasses

Type Classes for `Vector`									
	`Bits`	`Eq`	`Literal`	`Arith`	`Ord`	`Bounded`	`Bitwise`	`Bit` Reduction	`Bit` Extend
`Vector`	√	√				√			

A vector can be turned into bits if the individual elements can be turned into bits. When packed and unpacked, the zeroth element of the vector is stored in the least significant bits. The size of the resulting bits is given by $tsize = vsize * \texttt{sizeOf}(element\_type)$ which is specified in the provisos.

```
instance Bits #( Vector#(vsize, element_type), tsize)
   provisos (Bits#(element_type, sizea),
             Mul#(vsize, sizea, tsize));
```


Vectors are zero-indexed; the first element of a vector `v`, is `v[0]`. When vectors are packed, they are packed in order from the LSB to the MSB.

Example. Vector#(5, Bit#(7)):

From the type, you can see that this will back into a 35-bit vector (5 elements, each with 7 bits).

MSB	34 bit positions 0					LSB
	V1[4]	V1[3]	V1[2]	V1[1]	V1[0]	

Example. A vector with a structure:

```
typedef struct { Bool a, UInt#(5) b} Newstruct deriving (Bits);
Vector#(3, NewStruct) v2;
```

The structure, `Newstruct` packs into 6 bits. Therefore `v2` will pack into an 18-bit vector. And its structure would look as follows:

	17	16 - 12	11	10 - 6	5	0	
MSB	v2[2].a	v2[2].b	v2[1].a	v2[1].b	v2[0].a	v2[0].b	LSB
	v2[2]		v2[1]		v2[0]		

Vectors can be compared for equality if the elements can. That is, the operators `==` and `!=` are defined.

Vectors are bounded if the elements are.

Functions for Creating and Generating vectors

The following functions are used to create new vectors, with and without defined elements. There are no Bluespec SystemVerilog constructors available for this abstract type (and hence no pattern-matching is available for this type) but the following functions may be used to construct values of the `Vector` type.

`newVector`	Generate a vector with undefined elements, typically used when vectors are declared.
	`function Vector#(vsize, element_type) newvector();`

`genVector`	Generate a vector containing integers 0 through N-1, vector[0] will have value 0.
	`function Vector#(vsize, Integer) genVector();`

`replicate`	Generate a vector of elements by replicating the given argument (c).
	`function Vector#(vsize, element_type) replicate(element_type c);`

`genWith`	Generate a vector of elements by applying the given function to 0 through N-1. The argument to the function is another function which has one argument of type `Integer` and returns an `element_type`.
	`function Vector#(vsize, element_type)` `  genWith(function element_type func(Integer x1));`

`cons`	Adds an element to a vector creating a vector one element larger. The new element will be at the 0th position.

	`function Vector#(vsize1, element_type)`
	`  cons (element_type elem, Vector#(vsize, element_type) vect)`
	`  provisos (Add#(1, vsize, vsize1));`

`nil`	Defines a zero-sized vector.
	`function Vector#(0, element_type) nil;`

`append`	Append two vectors containing elements of the same type, returning the combined vector. The resulting vector will contain all the elements of `vecta` followed by all the elements of `vectb`.

	`function Vector#( vsize, element_type )`
	`  append( Vector#(v0size,element_type) vecta`
	`  Vector#(v1size,element_type) vectb`
	`  provisos (Add#(v0size, v1size, vsize)); //vsize = vsize0 + v1size`

`concat`	Append (*con*catenate) many vectors, that is a vector of vectors into one vector.

	`function Vector#(mvsize,element_type)`
	`  concat(Vector#(m,Vector#(n,element_type)) xss)`
	`  provisos (Mul#(m,n,mvsize));`

Examples - Creating and Generating Vectors

Create a new vector, `my_vector`, of 5 elements of datatytpe `Int#(32)`, with elements which are undefined.

```
Vector #(5, Int#(32)) my_vector;
```

Create a vector, my_vector, of five 1's

```
Vector #(5,Int #(32)) my_vector = replicate (1);

// my_vector is a 5 element vector {1,1,1,1,1}
```

Create a vector, `my_vector`, by applying the given function `add2` to 0 through N-1.

```
function Integer add2 (Integer a);
    Integer c = a + 2;
return(c);
endfunction

Vector #(5,Integer) my_vector = genWith(add2);

// a is the index of the vector, 0 to N-1
// my_vector = {2,3,4,5,6,}
```


Functions for Extracting Elements and Sub-Vectors

These functions are used to select elements or vectors from existing vectors, while retaining the input vector.

`[i]`	The square-bracket notation is available to extract an element from a vector. Extracts the ith element, where the first element is [0]. Index i must be one of the following types; `Integer`, `Bit#(n)`, `Int` or `UInt`.
	`anyVector[i]`

`select`	The select function is another form of the subscript notation ([i]). It is necessary when the compiler can't determine the type of the subscript i.

	`function element_type`
	`  select(Vector#(vsize,element_type) vect, idx_type index)`
	`  provisos (Eq#(idx_type), Literal#(idx_type));`

`update`	Update an element in a vector and return a new vector which is the given vector with one element changed/updated. This function does not change the given vector.

	`function Vector#(vsize, element_type)`
	`  update(Vector#(vsize, element_type) vectIn,`
	`  idx_type index,`
	`  element_type newElem)`
	`  provisos(Eq#(idx_type), Literal#(idx_type));`

`head`	Extract the zeroth (head) element of a vector. The vector must have at least one element.

	`function element_type`
	`  head (Vector#(vsize, element_type) vect)`
	`  provisos(Add#(1,xxx,vxize)); // vsize >= 1`

`last`	Extract the last (tail) element of a vector. The vector must have at least one element.

	`function element_type`
	`  last (Vector#(vsize, element_type) vect)`
	`  provisos(Add#(1,xxx,vxize)); // vsize >= 1`

`tail`	Remove the head element of a vector leaving its tail in a smaller vector.

	`function Vector#(vsize,element_type)`
	`  tail (Vector#(vsize1, element_type) xs)`
	`  provisos (Add#(1, vsize, vsize1));`

`init`	Remove the last element of a vector leaving its initial part in a smaller vector.

	`function Vector#(vsize,element_type)`
	`  init (Vector#(vsize1, element_type) xs)`
	`  provisos (Add#(1, vsize, vsize1));`

`take`	Take a number of elements from a vector starting from index 0. The number of elements to take is indicated by the type of the context where this is called, and is not specified as an argument to the function.

	`function Vector#(vxize2,element_type)`
	`  take (Vector#(vsize,element_type) vect)`
	`  provisos (Add#(vsize2,xxx,vsize)); // vsize2 <= vsize.`

`takeTail`	Take a number of elements from the tail by dropping elements at the 0th position. The elements in the result vector will be in the same order as the input vector.

	`function Vector#(vxize2,element_type)`
	`  takeTail (Vector#(vsize,element_type) vect)`
	`  provisos (Add#(vsize2,xxx,vsize)); // vsize2 <= vsize.`

`takeAt`	Take a number of elements starting at `startPos`. `startPos` must be a compile-time constant. If the `startPos` and vector size cause the function to go past the end of the vector, an error will be returned.

	`function Vector#(vsize2,element_type)`
	`  takeAt (Integer startPos, Vector#(vsize,element_type) vect)`
	`  provisos (Add#(vsize2,xxx,vsize)); // vsize2 <= vsize`

Examples - Extracting Elements and Sub-Vectors

Extract the element from a vector, `my_vector`, at the position of index

```
// my_vector is a vector of elements {6,7,8,9,10,11}
// index = 3
// select will generate a MUX

newvalue = select (my_vector, index);
```



```
// newvalue = 9
```

Extract the zeroth element of the vector `my_vector`

```
// my_vector is a vector of elements {6,7,8,9,10,11}

newvalue = head(my_vector);

// newvalue = 6
```

Create a vector, `my_vector2`, of size 4 by removing the head (zeroth) element of the vector `my_vector1`

```
// my_vector1 is a vector with 5 elements {0,1,2,3,4}

Vector #(4, Int#(32)) my_vector2 = tail (my_vector1);

// my_vector2 is a vector of 4 elements {1,2,3,4}
```

Create a 2 element vector, `my_vector2`, by taking the first two elements of the vector `my_vector1`

```
// my_vector1 is vector with 5 elements {0,1,2,3,4}

Vector #(2, Int#(4)) my_vector2 = take (my_vector1);

// my_vector2 is a 2 element vector {0,1}
```

Create a 3 element vector, `my_vector2`, by taking the last 3 elements of vector, `my_vector1` using `takeTail`

```
// my_vector1 is Vector with 5 elements {0,1,2,3,4}

Vector #(3,Int #(4)) my_vector2 = takeTail (my_vector1);

// my_vector2 is a 3 element vector {2,3,4}
```

Create a 3 element vector, `my_vector2`, by taking the 1st - 3rd elements of vector, `my_vector1` using `takeAt`

```
// my_vector1 is Vector with 5 elements {0,1,2,3,4}

Vector #(3,Int #(4)) my_vector2 = takeAt (1, my_vector1);

// my_vector2 is a 3 element vector {1,2,3}
```

Functions for Combining Vectors with Zip

The family of `zip` functions takes two or more vectors and combines them into one vector of `Tuples`. Several variations are provided for different resulting `Tuples`, as well as support for mis-matched vector sizes. Tuples are described in Section 12.4.

`zip`	Combine two vectors into a vector of Tuples.

	`function Vector#(vsize,Tuple2 #(a_type, b_type))`
	`  zip( Vector#(vsize, a_type) vecta,`
	`  Vector#(vsize, b_type) vectb);`

zip3	Combine three vectors into a vector of Tuple3.

	`function Vector#(vsize,Tuple3 #(a_type, b_type, c_type))`
	`  zip3( Vector#(vsize, a_type) vecta,`
	`  Vector#(vsize, b_type) vectb,`
	`  Vector#(vsize, c_type) vectc);`

zip4	Combine four vectors into a vector of Tuple4.

	`function Vector#(vsize,Tuple4 #(a_type, b_type, c_type, d_type))`
	`  zip4( Vector#(vsize, a_type) vecta,`
	`  Vector#(vsize, b_type) vectb,`
	`  Vector#(vsize, c_type) vectc,`
	`  Vector#(vsize, d_type) vectd);`

zipAny	Combine two vectors into one vector of pairs (2-tuples); result is as long as the smaller vector.

	`function Vector#(vsize,Tuple2 #(a_type, b_type))`
	`  zipAny(Vector#(m,a_type) vect1,`
	`  Vector#(n,b_type) vect2);`
	`  provisos (Max#(m,vsize,m), Max#(n, vsize, n));`

unzip	Separate a vector of pairs (i.e. a `Tuple2#(a,b)`) into a pair of two vectors.
	`function Tuple2#(Vector#(vsize,a_type), Vector#(vsize, b_type))` `  unzip(Vector#(vsize,Tuple2 #(a_type, b_type)) vectab);`

Examples - Combining Vectors with Zip

Combine two vectors into a vector of Tuples

```
// my_vector1 is a vector of elements {0,1,2,3,4}
// my_vector2 is a vector of elements {5,6,7,8,9}

my_vector3 = zip(my_vector1, my_vector2);

// my_vector3 is a vector of Tuples {(0,5),(1,6),(2,7),(3,8),(4,9)}
```

Separate a vector of pairs into a Tuple of two vectors

```
// my_vector3 is a vector of pairs {(0,5),(1,6),(2,7),(3,8),(4,9)}
Tuple2#(Vector #(5,Int #(5)),Vector #(5,Int #(5))) my_vector4 =
                                                  unzip(my_vector3);

// my_vector4 is ({0,1,2,3,4},{5,6,7,8,9})
```


Mapping Functions over Vectors

A function can be applied to all elements of a vector, using high-order functions such as `map`. These functions take as an argument a function, which is applied to the elements of the vector.

`map`	Map a function over a vector, returning a new vector of results.

	`function Vector#(vsize,b_type)`
	`  map (function b_type func(a_type x),`
	`  Vector#(vsize, a_type) vect);`

Example - Mapping Functions over Vectors

Consider the following code example which applies the `zeroExtend` function to each element of `avector` into a new vector, `resultvector`.

```
Vector#(13,Bit#(5))   avector;
Vector#(13,Bit#(10))  resultvector;
...
resultvector = map( zeroExtend, avector ) ;
```

This is equivalent to saying:

```
for (Integer i=0; i<13; i=i+1)
    resultvector[i] = zeroExtend(avector[i]);
```

Map a negate function over a Vector

```
// my_vector1 is a vector of 5 elements {0,1,2,3,4}
// negate is a function which makes each element negative

Vector #(5,Int #(32)) my_vector2 = map (negate, my_vector1);

// my_vector2 is a vector of 5 elements {0,-1,-2,-3,-4}
```

ZipWith Functions

The zipWith functions combine two or more vectors with a function and generate a new vector. These functions combine features of map and zip functions.

`zipWith`	Combine two vectors with a function.

	`function Vector#(vsize,c_type)`
	`  zipWith (function c_type func(a_type x, b_type y),`
	`  Vector#(vsize,a_type) vecta,`
	`  Vector#(vsize,b_type) vectb );`

`zipWithAny`	Combine two vectors with a function; result is as long as the smaller vector.

	`function Vector#(vsize,c_type)`
	`  zipWithAny (function c_type func(a_type x, b_type y),`
	`  Vector#(m,a_type) vecta,`
	`  Vector#(n,b_type) vectb )`
	`  provisos (Max#(n, vsize, n), Max#(m, vsize, m));`

zipWith3	Combine three vectors with a function.

	`function Vector#(vsize,d_type)`
	`  zipWith3(function d_type func(a_type x, b_type y, c_type z),`
	`  Vector#(vsize,a_type) vecta,`
	`  Vector#(vsize,b_type) vectb,`
	`  Vector#(vsize,c_type) vectc );`

zipWithAny3	Combine three vectors with a function; result is as long as the smallest vector.

	`function Vector#(vsize,c_type)`
	`  zipWithAny3(function d_type func(a_type x, b_type y, c_type z),`
	`  Vector#(m,a_type) vecta,`
	`  Vector#(n,b_type) vectb,`
	`  Vector#(o,c_type) vectc )`
	`provisos (Max#(n, vsize, n), Max#(m, vsize, m), Max#(o, vsize, o));`

Examples - ZipWith

Create a vector by applying a function over the elements of 3 vectors.

```
// the function add3 adds 3 values
function Int#(n) add3 (Int #(n) a,Int #(n) b,Int #(n) c);
    Int#(n) d = a + b +c ;
    return d;
endfunction

// Create the vector my_vector4 by adding the ith element of each of
// 3 vectors (my_vector1, my_vector2, my_vector3) to generate the ith
// element of my_vector4.

// my_vector1 = {0,1,2,3,4}
// my_vector2 = {5,6,7,8,9}
// my_vector3 = {10,11,12,13,14}

Vector #(5,Int #(8)) my_vector4 = zipWith3(add3, my_vector1, my_vector2, my_vector3);
// creates 5 instances of the add3 function in hardware.
// my_vector4 = {15,18,21,24,27}

// This is equivalent to saying:
   for (Integer i=0; i<5; i=i+1)
      my_vector4[i] = my_vector1[i] + my_vector2[i] + my_vector3[i];
```

Fold Functions

The `fold` family of functions reduces a vector to a single result by applying a function over all its elements. That is, given a vector of `element_type`, $V_0, V_1, V_2, ..., V_{n-1}$, a seed of type `b_type`, and a function `func`, the reduction for `foldr` is given by

$$func(V_0, func(V_1, ..., func(V_{n-2}, func(V_{n-1}, seed))));$$

Note that `foldr` start processing from the highest index position to the lowest, while `foldl` starts from the lowest index (zero), i.e. `foldl` is:

$$func(...(func(func(seed, V_0), V_1), ...)V_{n-1})$$

`foldr`	Reduce a vector by applying a function over all its elements. Start processing from the highest index to the lowest.
	`function b_type foldr(b_type function func(a_type x, b_type y), b_type seed, Vector#(vsize,a_type) vect);`

`foldl`	Reduce a vector by applying a function over all its elements. Start processing from the lowest index (zero).
	`function b_type foldl (b_type function func(b_type y, a_type x), b_type seed, Vector#(vsize,a_type) vect);`

The functions `foldr1` and `foldl1` use the first element as the seed. This means they only work on vectors of at least one element. Since the result type will be the same as the element type, there is no `b_type` as there is in the `foldr` and `foldl` functions.

`foldr1`	`foldr` function for a non-zero sized vector, using element V_{n-1} as a seed. Vector must have at least 1 element. If there is only one element, it is returned.

	`function element_type foldr1(`
	`  element_type function func(element_type x, element_type y),`
	`  Vector#(vsize,element_type) vect)`
	`  provisos (Add#(1, xxx, vsize));`

`foldl1`	`foldl` function for a non-zero sized vector, using element V_0as a seed. Vector must have at least 1 element. If there is only one element, it is returned.

	`function element_type foldl1 (`
	`  element_type function func(element_type y, element_type x),`
	`  Vector#(vsize,element_type) vect)`
	`  provisos (Add#(1, xxx, vsize));`

The `fold` function also operates over a non-empty vector, but processing is accomplished in a binary tree-like structure. Hence the depth or delay through the resulting function will be $O(log_2(vsize)$ rather than $O(vsize)$.

`fold`	Reduce a vector by applying a function over all its elements, using a binary tree-like structure. The function returns the same type as the arguments.

	`function element_type fold (`
	`  element_type function func(element_type y, element_type x),`
	`  Vector#(vsize,element_type) vect )`
	`  provisos (Add#(1, xxx, vsize));`

Example - Folds

Use fold to find the sum of the elements in a vector

```
// my_vector1 is a vector of five integers {1,2,3,4,5}
// \+ is a function which returns the sum of the elements

// This will build an adder tree, instantiating 4 adders, with a maximum
// depth or delay of 3.  If foldr1 or foldl1 were used, it would
// still instantiate 4 adders, but the delay would be 4.

my_sum =  fold (\+ , 0, my_vector1));

// my_sum = 15
```

Use fold to find the element with the maximum value

```
// my_vector1 is a vector of five integers  {2,45,5,8,32}

my_max = fold (max, my_vector1);

// my_max = 45
```

Scan Functions

The `scan` family of functions applies a function over a vector, creating a new Vector result. The `scan` function is similar to `fold`, but the intermediate results are saved and returned in a vector, instead of returning just the last result. The result of a `scan` function is a vector. That is, given a vector of `element_type`, $V_0, V_1, ..., V_{n-1}$, an initial value `initb` of type `b_type`, and a function `func`, application of the `scanr` functions creates a new vector W, where

$$\begin{aligned} W_n &= init; \\ W_{n-1} &= func(V_{n-1}, W_n); \\ W_{n-2} &= func(V_{n-2}, W_{n-1}); \\ &... \\ W_1 &= func(V_1, W_2); \\ W_0 &= func(V_0, W_1); \end{aligned}$$

scanr	Apply a function over a vector, creating a new vector result. Processes elements from the highest index position to the lowest, and fill the resulting vector in the same way. The result vector is 1 element longer than the input vector.

	`function Vector#(vsize1,b_type)`
	`  scanr(function b_type func(a_type x1, b_type x2),`
	`  b_type initb,`
	`  Vector#(vsize,a_type) vect)`
	`  provisos (Add#(1, vsize, vsize1));`

sscanr	Apply a function over a vector, creating a new vector result. The elements are processed from the highest index position to the lowest. The W_n element is dropped from the result. Input and output vectors are the same size.

	`function Vector#(vsize,b_type)`
	`  sscanr(function b_type func(a_type x1, b_type x2),`
	`  b_type initb,`
	`  Vector#(vsize,a_type) vect );`

The `scanl` function creates the resulting vector in a similar way as `scanr` except that the processing happens from the zeroth element up to the nth element.

$$\begin{aligned} W_0 &= init; \\ W_1 &= func(W_0, V_0); \\ W_2 &= func(W_1, V_1); \\ &\ldots \\ W_{n-1} &= func(W_{n-2}, V_{n-2}); \\ W_n &= func(W_{n-1}, V_{n-1}); \end{aligned}$$

The `sscanl` function drops the first result, *init*, shifting the result index by one.

scanl	Apply a function over a vector, creating a new vector result. Processes elements from the zeroth element up to the nth element. The result vector is 1 element longer than the input vector.

	`function Vector#(vsize1,a_type)`
	`  scanl(function a_type func(a_type x1, b_type x2),`
	`  a_type q,`
	`  Vector#(vsize,b_type) vect)`
	`  provisos (Add#(1, vsize, vsize1));`

`sscanl`	Apply a function over a vector, creating a new vector result. Processes elements from the left, the zeroth element up to the nth element. The first result, *init*, is dropped, shifting the result index by one. Input and output vectors are the same size.

	`function Vector#(vsize,a_type)`
	`  sscanl(function a_type func(a_type x1, b_type x2),`
	`  a_type q,`
	`  Vector#(vsize,b) vect );`

Examples - Scan

Create a vector of factorials

```
// \* is a function which returns the result of a multiplied by  b
function Bit #(16) \* (Bit #(16) b, Bit #(8) a);
   return (zeroExtend (a) * b);
endfunction

// Create a vector of factorials by multiplying each input list element
// by the previous product (the output list element), to generate
// the next product.  The seed is a Bit#(16) with a value of 1.
// The elements are processed from the zeroth element up to the nth element.

// my_vector1 = {1,2,3,4,5,6,7}
Vector#(8,Bit #(16)) my_vector2 = scanl (\*, 16'd1, my_vector1);
// 7 multipliers are generated

// my_vector2 = {1,1,2,6,24,120,720,5040}
// foldr with the same arguments would return just 5040.
```

Vector to Vector Functions

The following functions generate a new vector by changing the position of elements within the vector.

`rotate`	Move the first element to the last and shift each element to the left.
	`function Vector#(vsize,element_type)` `  rotate (Vector#(vsize,element_type) vect);`

`rotateR`	Move last element to the beginning and shift each element to the right.
	`function Vector#(vsize,element_type)` `  rotateR (Vector#(vsize,element_type) vect);`

`shiftInAt0`	Shift a new element into the vector at index 0, bumping all other elements up by one. The Nth element is dropped.

	`function Vector#(vsize,element_type)`
	`  shiftInAt0 (Vector#(vsize,element_type) vect,`
	`  element_type newElement);`

`shiftInAtN`	Shift a new element into the vector at index N, bumping all other elements down by one. The 0th element is dropped.

	`function Vector#(vsize,element_type)`
	`  shiftInAtN (Vector#(vsize,element_type) vect,`
	`  element_type newElement);`

`reverse`	Reverse element order
	`function Vector#(vsize,element_type)` `  reverse(Vector#(vsize,element_type) vect);`

`transpose`	Matrix transposition of a vector of vectors.
	`function Vector#(m,Vector#(n,element_type))` `  transpose ( Vector#(n,Vector#(m,element_type)) matrix );`

`transposeLN`	Matrix transposition of a vector of Lists.
	`function Vector#(vsize, List#(element_type))` `  transposeLN( List#(Vector#(vsize, element_type)) lvs );`

Examples - Vector to Vector Functions

Create a vector by moving the last element to the first, then shifting each element to the right.

```
// my_vector1 is a vector of elements with values {1,2,3,4,5}

my_vector2 = rotateR (my_vector1);

// my_vector2 is a vector of elements with values {5,1,2,3,4}
```

Create a vector which is the reverse of the input vector

```
// my_vector1 is a vector of elements {1,2,3,4,5}

my_vector2 = reverse (my_vector1);

// my_vector2 is a vector of elements {5,4,3,2,1}
```


Use transpose to create a new vector

```
// my_vector1 is a Vector#(3, Vector#(5, Int#(8)))
// the result, my_vector2, is a Vector #(5,Vector#(3,Int #(8)))

// my_vector1 has the values:
// {{0,1,2,3,4},{5,6,7,8,9},{10,11,12,13,14}}

my_vector2 = transpose(my_vector1);

// my_vector2 has the values:
// {{0,5,10},{1,6,11},{2,7,12},{3,8,13},{4,9,14}}
```

Monadic Operations

Within Bluespec, there are some functions which can only be invoked in certain contexts. Two common examples are: `ActionValue`, and module instantiation. ActionValues can only be invoked within an `Action` context, such as a rule block or an Action method, and can be considered as two parts - the action and the value. Module instantiation can similarly be considered, modules can only be instantiated in the module context, while the two parts are the module instantiation (the action performed) and the interface (the result returned). These situations are considered *monadic*.

When a monadic function is to be applied over a vector using map-like functions such as `map`, `zipWith`, or `replicate`, the monadic versions of these functions must be used. Moreover, the context requirements of the applied function must hold. The common application for these functions is in the generation (or instantiation) of vectors of hardware components.

`mapM`	Takes a monadic function and a vector, and applies the function to all vector elements returning the vector of corresponding results.

	`function m#(Vector#(vsize, b_type))`
	`  mapM ( function m#(b_type) func(a_type x),`
	`  Vector#(vsize, a_type) vecta )`
	`  provisos (Monad#(m));`

`mapM_`	Takes a monadic function and a vector, applies the function to all vector elements, and throws away the resulting vector leaving the action in its context.

	`function m#(void) mapM_(function m#(b_type) func(a_type x),`
	`  Vector#(vsize, a_type) vect)`
	`  provisos (Monad#(m));`

zipWithM	Take a monadic function (which takes two arguments) and two vectors; the function applied to the corresponding element from each vector would return an action and result. Perform all those actions and return the vector of corresponding results.

	`function m#(Vector#(vsize, c_type))`
	`  zipWithM( function m#(c_type) func(a_type x, b_type y),`
	`  Vector#(vsize, a_type) vecta,`
	`  Vector#(vsize, b_type) vectb )`
	`  provisos (Monad#(m));`

zipWithM_	Take a monadic function (which takes two arguments) and two vectors; the function is applied to the corresponding element from each vector. This is the same as zipWithM but the resulting vector is thrown away leaving the action in its context.

	`function m#(void)`
	`  zipWithM_(function m#(c_type) func(a_type x, b_type y),`
	`  Vector#(vsize, a_type) vecta,`
	`  Vector#(vsize, b_type) vectb )`
	`  provisos (Monad#(m));`

zipWith3M	Same as zipWithM but combines three vectors with a function. The function is applied to the corresponding element from each vector and returns an action and the vector of corresponding results.

	`function m#(Vector#(vsize, c_type))`
	`  zipWith3M( function m#(d_type)`
	`  func(a_type x, b_type y, c_type z),`
	`  Vector#(vsize, a_type) vecta,`
	`  Vector#(vsize, b_type) vectb,`
	`  Vector#(vsize, c_type) vectc )`
	`  provisos (Monad#(m));`

genWithM	Generate a vector of elements by applying the given monadic function to 0 through N-1.

	`function m#(Vector#(vsize, element_type))`
	`  genWithM(function m#(element_type) func(Integer x))`
	`  provisos (Monad#(m));`

replicateM	Generate a vector of elements by using the given monadic value repeatedly.

	`function m#(Vector#(vsize, element_type))`
	`  replicateM( m#(element_type) c)`
	`  provisos (Monad#(m));`

Examples - Creating a Vector of Registers

The following example shows some common uses of the `Vector` type. We first create a vector of registers, and show how to populate this vector. We then continue with some examples of accessing and updating the registers within the vector, as well as alternate ways to do the same.

```
// First define a variable to hold the register interfaces.
// Notice the variable is really a vector of Interfaces of type Reg,
// not a vector of modules.
Vector#(10,Reg#(DataT))   vectRegs ;

// Now we want to populate the vector, by filling it with Reg type
// interfaces, via the mkReg module.
// Notice that the replicateM function is used instead of the
// replicate function since mkReg function is creating a module.
vectRegs <- replicateM( mkReg( 0 ) ) ;

// ...

// A rule showing a read and write of one register within the
// vector.
// The readReg function is required since the selection of an
// element from vectRegs returns a Reg#(DType) interface, not the
// value of the register.  The readReg functions converts from a
// Reg#(DataT) type to a DataT type.
rule zerothElement ( readReg( vectRegs[0] ) > 20 ) ;
   // set 0 element to 0
   // The parentheses are required in this context to give
   // precedence to the selection over the write operation.
   (vectRegs[0]) <= 0 ;

   // Set the 1st element to 5
   // An alternate syntax
   vectRegs[1]._write( 5 ) ;
endrule

rule lastElement ( readReg( vectRegs[9] ) > 200 ) ;
   // Set the 9th element to -10000
   (vectRegs[9]) <= -10000 ;
endrule

// These rules defined above can execute simultaneously, since
// they touch independent registers

// Here is an example of dynamic selection,  first we define a
// register to be used as the selector.
Reg#(UInt#(4))  selector <- mkReg(0) ;
```



```
// Now define another Reg variable which is selected from the
// vectReg variable.  Note that no register is created here, just
// an alias is defined.
Reg#(DataT)  thisReg = select(vectRegs, selector ) ;

//The above statement is equivalent to:
//Reg#(DataT) thisReg = vectRegs[selector] ;


// If the selected register is greater than 20'h7_0000, then its
// value is reset to zero.  Note that the vector update function is
// not required since we are changing the contents of a register
// not the vector vectReg.
rule reduceReg( thisReg > 20'h7_0000 ) ;
   thisReg <= 0 ;
   selector <= ( selector < 9 ) ? selector + 1 : 0 ;
endrule

// As an alternative, we can define N rules which each check the
// value of one register and update accordingly.  This is done by
// generating each rule inside an elaboration-time for-loop.

Integer i;  // a compile time variable
for ( i = 0 ; i < 10 ; i = i + 1 ) begin
   rule checkValue( readReg( vectRegs[i] ) > 20'h7_0000 ) ;
      (vectRegs[i]) <= 0 ;
   endrule
end
```

Functions for Tests on Vectors

The following functions are used to test vectors - they are boolean functions, i.e. they return `True` or `False` values.

`elem`	Check if a value is an element of a vector.

	`function Bool elem (element_type x,`
	`  Vector#(vsize,element_type) vect )`
	`  provisos (Eq#(element_type));`

`any`	Test if a predicate holds for any element of a vector.
	`function Bool any(function Bool pred(element_type x1),` `  Vector#(vsize,element_type) vect );`

`all`	Test if a predicate holds for all elements of a vector.
	`function Bool all(function Bool pred(element_type x1), Vector#(vsize,element_type) vect );`

Examples - Tests on Vectors

Test that all elements of the vector `my_vector1` are positive integers

```
function Bool isPositive (Int #(32) a);
     return (a > 0)
endfunction

// function isPositive checks that "a" is a  positive integer
// if my_vector1 has n elements, n instances of the predicate
// function isPositive will be generated.

if (all(isPositive, my_vector1))
   $display ("Vector contains all negative values");
```

Test if any elements in the vector are positive integers.

```
// function isPositive checks that "a" is a  positive integer
// if my_vector1 has n elements, n instances of the predicate
// function isPositive will be generated.

if (any(pos, my_vector1))
   $display ("Vector contains some negative values");
```

Check if the integer 5 is in `my_vector`

```
// if my_vector contains n elements, elem will generate n copies
// of the eq test
if (elem(5,my_vector))
   $display ("Vector contains the integer 5");
```

Functions for Converting to and from Vectors

There are functions which convert to and from `List` and `Vector`.

`toList`	Convert a Vector to a List.
	`function List#(element_type) toList (Vector#(vsize, element_type) vect);`

`toVector`	Convert a List to a Vector.
	`function Vector#(vsize, element_type) toVector ( List#(element_type) lst);`

There are functions which convert to and from `array` and `Vector`.

arraytoVector	Convert an array to a Vector.
	`function Vector#(vsize, element_type)` `arrayToVector ( element_type[ ] arr);`

vectorToArray	Convert a Vector to an array.
	`function element_type[ ]` `vectorToArray (Vector#(vsize, element_type) vect);`

Example - Converting to and from Vectors

Convert the vector `my_vector` to a list named `my_list`

```
Vector#(5,Int#(13)) my_vector;
List#(Int#(13)) my_list = toList(my_vector);
```

Miscellaneous Functions on Vectors

joinActions	Join a number of actions together. `joinActions` is used for static elaboration only, no hardware is generated.
	`function Action joinActions (Vector#(vsize,Action) vactions);`

joinRules	Join a number of rules together.`joinRules` is used for static elaboration only, no hardware is generated.
	`function Rules joinRules (Vector#(vsize,Rules) vrules);`

mapAccumL	Map a function, but pass an accumulator from head to tail.

	`function Tuple2 #(a_type, Vector#(vsize,c_type))`
	`mapAccumL (function Tuple2 #(a_type, c_type)`
	`func(a_type x, b_type y), a_type x0,`
	`Vector#(vsize,b_type) vect );`

mapAccumR	Map a function, but pass an accumulator from tail to head.

	`function Tuple2 #(a_type, Vector#(vsize,c_type))`
	`mapAccumR(function Tuple2 #(a_type, c_type)`
	`func(a_type x, b_type y), a_type x0,`
	`Vector#(vsize,b_type) vect );`

mapPairs	Map a function over a vector consuming two elements at a time. Any straggling element is processed by the second function.

	`function Vector#(vsize2,b_type)`
	`  mapPairs (`
	`  function b_type func1(a_type x, a_type y),`
	`  function b_type func2(a_type x),`
	`  Vector#(vsize,a_type) vect )`
	`  provisos (Div#(vsize, 2, vsize2));`

Examples - Miscellaneous Functions on Vectors

Create a new vector using `mapPairs`. The function `sum` is applied to each pair of elements (the first and second, the third and fourth, etc.). If there is an uneven number of elements, the function `pass` is applied to the remaining element.

```
// sum is defined as c = a+b
function Int#(4) sum (Int #(4) a,Int #(4) b);
     Int#(4) c = a + b;
     return(c);
endfunction

// pass is defined as a
function Int#(4) pass (Int #(4) a);
     return(a);
endfunction

// my_vector1 has the elements {0,1,2,3,4}

my_vector2 = mapPairs(sum,pass,my_vector1);

// my_vector2 has the elements {1,5,4}
// my_vector2[0] = 0 + 1
// my_vector2[1] = 2 + 3
// my_vector2[3] = 4
```

C.1.8 ListN

Package name

import ListN :: * ;

Description

ListN is an alternative implementation of Vector which is preferred for list processing functions, such as head, tail, map, fold, etc. All Vector functions are available, by substituting ListN for Vector. See the Vecotr docuemntation (C.1.7) for details. If the implementation requries random access to items in the list, the Vector construct is recommended. Using ListN where Vectors is recommended (and visa-versa) can lead to very long static elaboration times.

The `ListN` package defines an abstract data type which is a listN of a specific length. Functions which create and operate on this type are also defined within this package. Because it is abstract, there are no constructors available for this type (like `Cons` and `Nil` for the `List` type).

```
struct ListN#(vsize,a_type)
      ··· abstract ···
```


Here, the type variable "a_type" represents the type of the contents of the listN while type variable "vsize" represents the length of the ListN.

C.2 Advanced Data Types

C.2.1 Complex

Package Name

import Complex :: * ;

Description

The Complex package provides a representation for complex numbers plus functions to operate on variables of this type. The basic representation is the Complex structure, which is polymorphic on the type of data it holds. For example, one can have complex numbers of type Int or of type FixedPoint. A Complex number is represented in two part, the real part (rel) and the imaginary part (img). These fields are accessible though standard structure addressing, i.e., foo.rel and foo.img where foo is of type Complex.

```
typedef struct {
        any_t  rel ;
        any_t  img ;
        } Complex#(type any_t)
deriving ( Bits, Eq ) ;
```

Types and type classes

The Complex type belongs to the Arith and Literal type classes. Each type class definition includes functions which are then also defined for the data type. The Prelude library definitions (Section B) describes which functions are defined for each type class.

Type Classes used by Complex									
	Bits	Eq	Literal	Arith	Ord	Bounded	Bit wise	Bit Reduction	Bit Extend
Complex	√	√	√	√					

Arith

The type Complex belongs to the Arith type class, hence the common infix operators (+, -, and *) are defined and can be used to manipulate variables of type Complex. Note however, that the complex multiplication (*) produces four multipliers in a combinational function; some other modules could accomplish the same function with less hardware but with greater latency.

```
instance Arith#( Complex#(any_type) )
      provisos( Arith#(any_type) ) ;
```

Literal

The Complex type is a member of the Literal class, which defines a conversion from the compile-time Integer type to Complex type with the fromInteger function. This function converts the Integer to the real part, and sets the imaginary part to 0.

```
instance Literal#( Complex#(any_type) )
   provisos( Literal#(any_type) );
```

Functions

`cmplx`	A simple constructor function is provided to set the fields.
	`function Complex#(a_type) cmplx( a_type realA, a_type imagA ) ;`

`cmplxMap`	Applies a function to each part of the complex structure. This is useful for operations such as `signExtend`, `truncate`, etc.

	`function Complex#(b_type) cmplxMap(`
	`function b_type mapFunc( a_type x),`
	`Complex#(a_type) cin ) ;`

`cmplxSwap`	Exchanges the real and imaginary parts.
	`function Complex#(a_type) cmplxSwap( Complex#(a_type) cin ) ;`

`cmplxWrite`	Displays a complex number given a prefix string, an infix string, a postscript string, and an Action function which writes each part. `cmplxWrite` is of type Action and can only be invoked in Action contexts such as Rules and Actions methods.

	`function Action cmplxWrite(String pre,`
	`String infix,`
	`String post,`
	`function Action writeaFunc( a_type x ),`
	`Complex#(a_type) cin );`

Examples - Complex Numbers

```
// The following utility function is provided for writing data
// in decimal format. An example of its use is show below.

function Action writeInt( Int#(n) ain ) ;
   $write( "%0d", ain ) ;
endfunction

// Set the fields of the complex number using the constructor function cmplx
Complex#(Int#(6)) complex_value = cmplx(-2,7) ;

// Display complex_value  as ( -2 + 7i ).
// Note that writeInt is passed as an argument to the cmplxWrite function.
cmplxWrite( "( ", " + ", "i)", writeInt, complex_value );
```



```
// Swap the real and imaginary parts.
swap_value = cmplxSwap( complex_value ) ;

// Display the swapped values. This will display ( -7 + 2i).
cmplxWrite( "( ", " + ", "i)", writeInt, swap_value );
```

C.2.2 FixedPoint

Package Name

import FixedPoint :: * ;

Description

The `FixedPoint` library package defines a type for representing fixed-point numbers and corresponding functions to operate and manipulate variables of this type.

A fixed-point number represents signed real numbers which have a fixed number of binary digits (bits) before and after the binary point. The type constructor for a fixed-point number takes two numeric types as argument; the first (isize) defines the number of bits to the left of the binary point (the integer part), while the second (fsize) defines the number of bits to the right of the binary point, (the fractional part).

The following data structure defines this type, while some utility functions provide the reading of the integer and fractional parts.

```
typedef struct {
                Int#(TAdd#(isize,fsize))  fxpt ;
                }
FixedPoint#(numeric type isize, numeric type fsize )
deriving( Eq, Bits ) ;
```

Types and type classes

The `FixedPoint` type belongs to the following type classes; `Eq`, `Bits`, `Bounded`, `Arith`, `Literal`, `Ord`, and `Bitwise`. Each type class definition includes functions which are then also defined for the data type. The Prelude library definitions (Section B) describes which functions are defined for each type class.

Type Classes used by `FixedPoint`									
	`Bits`	`Eq`	`Literal`	`Ord`	`Bounded`	`Bit` wise	`Bit` Reduce	`Bit` Extend	`Arith`
`FixedPoint`	X	X	X	X	X	X			X

`Bounded`

The range of values, v, representable with a signed fixed-point number of type `FixedPoint#(isize, fsize)` is $+(2^{isize-1} - 2^{-fsize}) \leq v \leq -2^{isize-1}$. This range is provided by the members of `Bounded` type class to which `FixedPoint` belongs. The function `epsilon` returns the smallest representable quantum by a specific type, 2^{-fsize}. For example, a variable v of type `FixedPoint#(2,3)` type can represent numbers from 3.875 ($3\frac{7}{8}$) to -4.0 in intervals of $\frac{1}{8} = 0.125$, i.e. epsilon is 0.125. The type `FixedPoint#(5,0)` is equivalent to `Int#(5)`.

```
instance Bounded#( FixedPoint#(i,f) )
```


epsilon	Returns the value of epsilon which is the smallest representable quantum by a specific type, 2^{-fsize}.
	function FixedPoint#(i,f) epsilon () ;

Arith

The type FixedPoint belongs to the Arith type class, hence the common infix operators (+, -, and *) are defined and can be used to manipulate variables of type FixedPoint.

```
instance Arith#( FixedPoint#(i,f) )
   provisos( Add#(1, xxx, i) ) ;       //   i >= 1
```

Literal

The type FixedPoint belongs to the Literal type class, which allows conversion from (compile-time) type Integer to type FixedPoint. Note that only the integer part is assigned.

```
instance Literal#( FixedPoint#(i,f) )
      provisos ( Add#(1, xxx, i) );    //   i >= 1
```

Ord

In addition to equality and inequality comparisons, FixedPoint variables can be compared by the relational operators provided by the Ord type class. i.e., <, >, <=, and >=.

```
instance Ord#( FixedPoint#(i,f) )
   provisos( Add#(1, xxx, i) ) ;       //   i >= 1
```

Bitwise

Left and right shifts are provided for FixedPoint variables as part of the Bitwise type class. Note that the shift right (>>) function does an arithmetic shift, thus preserving the sign of the operand. Note that a right shift of 1 is equivalent to a division by 2, except when the operand is equal to −epsilon. The other methods of Bitwise type class are not provided since they have no operational meaning on FixedPoint variables; the use of these generates an error message.

```
instance Bitwise#( FixedPoint#(i,f) )
   provisos( Add#(1, xxx, i) ) ;       //   i >= 1
```

Functions

Utility functions are provided to extract the integer and fractional parts.

fxptGetInt	Extracts the integer part of the FixedPoint number.
	function Int#(isize) fxptGetInt (FixedPoint#(isize, fsize) x) provisos(Add#(1, xxx, isize)) ; // isize >= 1

fxptGetFrac	Extracts the factional part of the FixedPoint number.
	function UInt#(fsize) fxptGetFrac (FixedPoint#(isize, fsize) x);

To convert run-time `Int` and `UInt` values to type `FixedPoint`, the following conversion functions are provided. Both of these functions invoke the necessary extension of the source operand.

`fromInt`	Converts run-time `Int` values to type `FixedPoint`.

	`function FixedPoint#(ir,fr) fromInt( Int#(ia) inta )`
	`  provisos ( Add#(1, xxA, ir ),  // ir >= 1`
	`  Add#(ia,xxB, ir ) );  // ir >= ia`

`fromUInt`	Converts run-time `UInt` values to type `FixedPoint`.

	`function FixedPoint#(ir,fr) fromUInt( UInt#(ia) uinta )`
	`  provisos ( Add#(ia,  1, ia1),  // ia1 = ia + 1`
	`  Add#(ia1,xxB, ir ) );  // ir >= ia1`

Non-integer compile time constants may be specified by a rational number which is a ratio of two integers. For example, one-third may be specified by `fromRational(1,3);`, while π can be specified as `fromRational( 31415926, 10000000);` .

`fromRational`	Specify a `FixedPoint` with a rational number which is the ratio of two integers.

	`function FixedPoint#(i,f) fromRational(`
	`  Integer numerator, Integer denominator)`
	`  provisos ( Add#(1, xxA, i ) ) ;  // i >= 1`

At times, a full precision multiplication may be required, where the result is sum of the field sizes of the operands. Note that the operand do not have to be the same type (sizes), as is required for the infix multiplication (*) operator.

`fxptMult`	Function for full precision multiplication, where the result is the sum of the field sizes of the operands.

	`function FixedPoint#(ri,rf)  fxptMult( FixedPoint#(ai,af) x,`
	`  FixedPoint#(bi,bf) y  )`
	`  provisos(  Add#(ai,bi,ri),  // ri = ai + bi`
	`  Add#(af,bf,rf),  // rf = af + bf`
	`  Add#(TAdd#(ai,af), TAdd#(bi,bf), TAdd#(ri,rf)) );`

`fxptTruncate` is a general truncate function which converts variables to `FixedPoint#(ai,af)` to type `FixedPoint#(ri,rf)`, where $ai \geq ri$ and $af \geq rf$. This function truncates bits as appropriate from the most significant integer bits and the least significant fractional bits.

fxptTruncate	Truncates bits as appropriate from the most significant integer bits and the least significant fractional bits.

```
function FixedPoint#(ri,rf) fxptTruncate(
             FixedPoint#(ai,af) a )
   provisos( Add#(xxA,ri,ai),    // ai >= ri
             Add#(xxB,rf,af),    // af >= rf
             Add#(xxC,TAdd#(ri,rf),TAdd#(ai,af))  );
                                 // ai+af >= ri+rf
```

`fxptSignExtend` is a general sign extend function which converts variables of type `FixedPoint#(ai,af)` to type `FixedPoint#(ri,rf)`, where $ai \leq ri$ and $af \leq rf$. The integer part is sign extended, while additional 0 bits are added to least significant end of the fractional part.

fxptSignExtend	General sign extend function where the integer part is sign extended while additional 0 bits are added to the least significant end of the fractional part.

```
function FixedPoint#(ri,rf) fxptSignExtend(
             FixedPoint#(ai,af) a )
   provisos( Add#(xxA,ai,ri),      // ri >= ai
             Add#(fdiff,af,rf),    // rf >= af
             Add#(xxC,TAdd#(ai,af),TAdd#(ri,rf)) );
                                   // ri+rf >= ai+af
```

fxptZeroExtend	A general zero extend function.

```
function FixedPoint#(ri,rf) fxptZeroExtend(
             FixedPoint#(ai,af) a )
   provisos( Add#(xxA,ai,ri),    // ri >= ai
             Add#(xxB,af,rf),    // rf >= af
             Add#(xxC,TAdd#(ai,af),TAdd#(ri,rf)) ) ;
                                 // ri+rf >= ai+af
```

Displaying `FixedPoint` values in a simple bit notation would result in a difficult to read pattern. The following write utility function is provided to ease in their display. Note that the use of this function adds many multipliers and adders into the design which are only used for generating the output and not the actual circuit.

fxptWrite	Displays a FixedPoint value in a decimal format, where fwidth give the number of digits to the right of the decimal point. fwidth must be in the inclusive range of 0 to 10. The displayed result is truncated without rounding.
	function Action fxptWrite(Integer fwidth, FixedPoint#(i,f) a)

Examples - Fixed Point Numbers

```
// The following code writes "x is 0.5156250"
FixedPoint#(1,6) x = half + epsilon ;
$write( "x is " ) ; fxptWrite( 7, x ) ; $display("" ) ;
```

C.2.3 OInt

Package Name

import OInt :: * ;

Description

The OInt#(n) type is an abstract type that can store a number in the range "0..n-1". The representation of a OInt#(n) takes up n bits, where exactly one bit is a set to one, and the others are zero, i.e., it is a *one-hot* decoded version of the number. The reason to use a OInt number is that the select operation is more efficient than for a binary-encoded number; the code generated for select takes advantage of the fact that only one of the bits may be set at a time.

Types and type classes

Definition of OInt

```
typedef ...  OInt #(numeric type n) ... ;
```

Type Classes used by OInt									
	Bits	Eq	Literal	Arith	Ord	Bounded	Bit wise	Bit Reduction	Bit Extend
OInt	√	√	√			√			

Functions

A binary-encoded number can be converted to an OInt.

toOInt	Converts from a bit-vector in unsigned binary format to an OInt. An out-of-range number gives an unspecified result.
	function OInt#(n) toOInt(Bit#(k) k) provisos(Log#(n,k)) ;

An OInt can be converted to a binary-encoded number.

fromOInt	Converts an OInt to a bit-vector in unsigned binary format.
	function Bit#(k) fromOInt(OInt#(n) o) provisos(Log#(n,k)) ;

An `OInt` can be used to select an element from a Vector in an efficient way.

`select`	The Vector `select` function, where the type of the index is an `OInt`.
	`function a_type select(Vector#(vsize, a_type) vecta, OInt#(vsize) index) provisos (Bits#(a_type, sizea));`

C.3 Control Structures

C.3.1 StmtFSM

Package Name

import StmtFSM :: * ;

Description

The `StmtFSM` package provides a procedural way of defining finite state machines (FSMs) which are automatically synthesized.

First, one uses the `Stmt` sublanguage to compose the actions of an FSM using sequential, parallel, conditional and looping structures. This sublanguage is within the *expression* syntactic category, i.e., a term in the sublanguage is an expression whose value is of type `Stmt`. This value can be bound to identifiers, passed as arguments and results of functions, held in static data structures, etc., like any other value. Finally, the FSM can be instantiated into hardware, multiple times if desired, by passing the `Stmt` value to the module constructor `mkFSM`. The resulting module interface has type `FSM`, which has methods to start the FSM and to wait until it completes.

The `Stmt` sublanguage

The state machine is automatically constructed from the procedural description given in the `Stmt` definition. Appropriate state counters are created and rules are generated internally, corresponding to the transition logic of the state machine. The use of rules for the intermediate state machine generation ensures that resource conflicts are identified and resolved, and that implicit conditions are properly checked before the execution of any action.

The names of generated rules (which may appear in conflict warnings) have suffixes of the form "`l<nn>c<nn>`", where the `<nn>` are line or column numbers, referring to the statement which gave rise to the rule.

A term in the `Stmt` sublanguage is an expression, introduced at the outermost level by the keywords `seq` or `par`. Note that within the sublanguage, `if`, `while` and `for` statements are interpreted as statements in the sublanguage and not as ordinary statements, except when enclosed within `Action/endAction` keywords.

exprPrimary ::= *seqFsmStmt* | *parFsmStmt*

fsmStmt ::= *exprFsmStmt*
| *seqFsmStmt*
| *parFsmStmt*
| *ifFsmStmt*
| *whileFsmStmt*
| *repeatFsmStmt*
| *forFsmStmt*

exprFsmStmt ::= *regWrite* ;
| *expression* ;


```
seqFsmStmt      ::= seq fsmStmt { fsmStmt } endseq
parFsmStmt      ::= par fsmStmt { fsmStmt } endpar
ifFsmStmt       ::= if expression fsmStmt
                    [ else fsmStmt ]
whileFsmStmt    ::= while ( expression )
                        loopBodyFsmStmt
forFsmStmt      ::= for ( fsmStmt ; expression ; fsmStmt )
                        loopBodyFsmStmt
repeatFsmStmt   ::= repeat ( expression )
                        loopBodyFsmStmt
loopBodyFsmStmt ::= fsmStmt
                  | break ;
                  | continue ;
```

The simplest kind of statement is an *exprFsmStmt*, which can be a register assignment (Section 8.4) or, more generally, any expression of type `Action` (including action method calls (Section 9.9) and `action-endaction` blocks (Section 9.6)) or of type `Stmt`. Statements of type `Action` execute within exactly one clock cycle, but of course the scheduling semantics described in Section 6.2 may affect exactly which clock cycle it executes in. For example, if the actions in a statement interfere with actions in some other rule, the statement may be delayed by the schedule until there is no interference. In all the descriptions of statements below, the descriptions of time taken by a construct are minimum times; they could take longer because of scheduling semantics.

Statements can be composed into sequential, parallel, conditional and loop forms. In the sequential form (`seq-endseq`), the contained statements are executed one after the other. The `seq` block terminates when its last contained statement terminates, and the total time (number of clocks) is equal to the sum of the individual statement times.

In the parallel form (`par-endpar`), the contained statements ("threads") are all executed in parallel. Statements in each thread may or may not be executed simultaneously with statements in other threads, depending on scheduling conflicts; if they cannot be executed simultaneously they will be interleaved, in accordance with normal scheduling. The entire `par` block terminates when the last of its contained threads terminates, and the minimum total time (number of clocks) is equal to the maximum of the individual thread times.

In the conditional form (`if (`b`)` s_1 `else` s_2), the boolean expression b is first evaluated. If true, s_1 is executed, otherwise s_2 (if present) is executed. The total time taken is $t + 1$ cycles, if the chosen branch takes t cycles.

In the `while (`b`)` s loop form, the boolean expression b is first evaluated. If true, s is executed, and the loop is repeated. Each time the condition is evaluated, it takes takes 1 cycle, so the total time is $1 + n \times (t + 1)$ cycles, where n is the number of times the loop is executed (possibly zero) and t is the time for the loop body statement.

The `for (`s_1`;`b`;`s_2`)` s_B loop form is equivalent to:

s_1`; while (`b`) seq` s_B`;` s_2 `endseq`

i.e., the initializer s_1 is executed first. Then, the condition b is executed and, if true, the loop body s_B is executed followed by the "increment" statement s_2. The b, s_B, s_2 sequence is repeated as long as b evaluates true.

Similarly, the `repeat (`n`)` s_B loop form is equivalent to:

s_1`; while (`$repeat\_count < n$`) seq` s_B`;` $repeat_count <= repeat\_count + 1$ `endseq`

where the value of *repeat_count* is initialized to 0. During execution, the condition ($repeat\_count < n$) is executed and, if true, the loop body s_B is executed followed by the "increment" statement $repeat\_count <= repeat\_count + 1$. The sequence is repeated as long as $repeat\_count < n$ evaluates true.

In all the loop forms, the loop body statements can contain the keywords `continue` or `break`, with the usual semantics, i.e., `continue` immediately jumps to the start of the next iteration, whereas `break` jumps out of the loop to the loop sequel.

It is important to note that this use of loops, within a `Stmt` context, expresses time-based (temporal) behavior. Section 8.7 describes the use of loops to express static structure, i.e., loops that are unrolled during static elaboration.

Interfaces and Methods

Two interfaces are defined with this package, `FSM` and `Once`. The `FSM` interface defines a basic state machine interface while the `Once` interface encapsulates the notion of an action that should only be performed once. A `Stmt` value can be instatiated into a module that presents an interface of type `FSM`.

Interfaces	
Name	Description
`FSM`	The state machine interface
`Once`	Used when an action should only be performed once

- `FSM` Interface

  The `FSM` interface provides three methods; `start`, `waitTillDone`, and `done`. Once instantiated, the FSM can be started by calling the `start` method. One can wait for the FSM to stop running by waiting explicitly on the boolean value returned by the `done` method. Alternatively, one can use the `waitTillDone` method in any action context (including from within another FSM), which (because of an implicit condition) cannot execute until this FSM is done.

```
interface FSM;
    method Action start();
    method Action waitTillDone();
    method Bool   done();
endinterface: FSM
```

`FSM` Interface		
Methods		
Name	Type	Description
`start`	`Action`	Begins state machine execution. This can only be called when the state machine is not executing.
`waitTillDone`	`Action`	Does not do any action, but is only ready when the state machine is done.
`done`	`Bool`	Asserted when the state machine is done and is ready to rerun.

- `Once` Interface

  The `Once` interface encapsulates the notion of an action that should only be performed once. The `start` method performs the action that has been encapuslated in the `Once` module. After `start` has been called `start` cannot be called again (an implicit condition will enforce this). If the `clear` method is called, the `start` method can be called once again.


```
interface Once;
    method Action start();
    method Action clear();
    method Bool   done() ;
endinterface: Once
```

Once Interface		
Methods		
Name	Type	Description
start	Action	Performs the action that has been encapsulated in the Once module, but once start has been called it cannot be called again (an implicit condition will enforce this).
clear	Action	If the clear method is called, the start method can be called once again.
done	Bool	Asserted when the state machine is done and is ready to rerun.

Modules

Instantiation is performed by passing a `Stmt` value ino the module constructor `mkFSM`. The state machine is automatically constructed from the procedural decription given in the definition described by state machine of type `Stmt` named `seq_stmt`. During construction, one or more registers of appropriate widths are created to track state execution. Upon `start` action, the registers are loaded and subsequent state changes then decrement the registers.

```
module mkFSM#( Stmt seq_stmt ) ( FSM );
```

The `mkAutoFSM` module is like `mkFSM` above, except the state machine runs automatically immediately after reset and a `$finish(0)` is called upon completion. This is useful for test benches. Thus, it has no interface, that is, it has an empty interface.

```
module mkAutoFSM#( seq_stmt ) ();
```

The `mkOnce` function is used to create a `Once` interface where the action argument has been encapsulated and will be performed when `start` is called.

```
module mkOnce#( Action a ) ( Once );
```

The implementation for `Once` is a 1 bit state machine (with a state register named `onceReady`) allowing the action argument to occur only one time. The ready bit is initially `True` and then cleared when the action is performed. It might not be performed right away, because of implicit conditions or scheduling conflicts.

Name	BSV Module Declaration	Description
mkFSM	`module mkFSM#(Stmt seq_stmt)(FSM);`	Instantiate a `Stmt` value into a module that presents an interface of type `FSM`.
mkAutoFSM	`module mkAutoFSM#(Stmt seq_stmt)();`	Like `mkFSM`, except that state machine simulation is automatically started and a `$finish(0)`) is called upon completion.
mkOnce	`module mkOnce#( Action a )( Once );`	Used to create a `Once` interface where the action argument has been encapsulated and will be performed when `start` is called.

Functions

There is one function, `await`, provided by the `StmtFSM` package. `await` is used to create an action which can only execute when the condition is `True`. The action does not do anything. `await` is useful to block the execution of an action until a condition becomes `True`.

Name	Function Declaration	Description
`await`	`function Action await( Bool cond ) ;`	Creates an Action which does nothing, but can only execute when the condition is `True`.

Example - Initializing a single-ported SRAM.

Since the SRAM has only a single port, we can write to only one location in each clock. Hence, we need to express a temporal sequence of writes for all the locations to be initialized.

```
Reg#(int) i, j;     // instantiate two register interfaces
mkRegU ri (i);      // create register with interface i
mkRegU rj (j);      // create register with interface j

// Define fsm behavior
Stmt s = seq
             for (i <= 0; i < M; i <= i + 1)
                 for (j <= 0; j < N; j <= j + 1)
                     sram.write (i, j, i+j);
         endseq

FSM fsm();          // instantiate FSM interface
mkFSM#(s) (fsm);    // create fsm with interface fsm and behavior s

...

rule initSRAM (start_reset);
    fsm.start;      // Start the fsm
endrule
```

When the `start_reset` signal is true, the rule kicks off the SRAM initialization. Other rules can wait on `fsm.done`, if necessary, for the SRAM initialization to be completed.

In this example, the `seq-endseq` brackets are used to enter the `Stmt` sublanguage, and then `for` represents `Stmt` sequencing (instead of its usual role of static generation). Since `seq-endseq` contains only one statement (the loop nest), `par-endpar` brackets would have worked just as well.

Example - Defining and instantiating a state machine.

```
import StmtFSM :: *;
import FIFO    :: *;

module testSizedFIFO();

  // Instantiation of DUT
  FIFO#(Bit#(16))  dut <- mkSizedFIFO(5);

  // Instantiation of reg's i and j
```



```
  Reg#(Bit#(4))       i  <- mkRegA(0);
  Reg#(Bit#(4))       j  <- mkRegA(0);

  // Action description with stmt notation
  Stmt driversMonitors =
   (seq
     // Clear the fifo
     dut.clear;

     // Two secuential blocks running in parallel
     par
       // Enque 2 times the Fifo Depth
        for(i <= 1; i <= 10; i <= i + 1)
        seq
          dut.enq({0,i});
          $display(" Enque %d", i);
        endseq

       // Wait until the fifo is full and then deque
       seq
         while (i < 5)
         seq
           noAction;
         endseq
         while (i <= 10)
         action
           dut.deq;
           $display("Value read %d", dut.first);
         endaction
       endseq

     endpar

     $finish(0);
   endseq);

   // stmt instantiation
   FSM  test  <- mkFSM(driversMonitors);

   // A register to control the start rule
   Reg#(Bool) going <- mkReg(False);

   // This rule kicks off the test FSM, which then runs to completion.
   rule start (!going);
      going <= True;
      test.start;
   endrule
endmodule
```

Example - Defining and instantiating a state machine to control speed changes

```
import StmtFSM::*;
import Common::*;

interface SC_FSM_ifc;
```



```
   method Speed xcvrspeed;
   method Bool  devices_ready;
   method Bool  out_of_reset;
endinterface

module mkSpeedChangeFSM(Speed new_speed, SC_FSM_ifc ifc);
   Speed initial_speed = FS;

   Reg#(Bool) outofReset_reg <- mkReg(False);
   Reg#(Bool) devices_ready_reg <- mkReg(False);
   Reg#(Speed) device_xcvr_speed_reg <- mkReg(initial_speed);

   // the following lines define the FSM using the Stmt sublanguage
   // the state machine is of type Stmt, with the name speed_change_stmt
   Stmt speed_change_stmt =
   (seq
       action outofReset_reg <= False; devices_ready_reg <= False; endaction
       noAction; noAction;

       device_xcvr_speed_reg <= new_speed;
       noAction; noAction;

       outofReset_reg <= True;
       if (device_xcvr_speed_reg==HS)
  seq noAction; noAction; endseq
       else
  seq noAction; noAction; noAction; noAction; noAction; noAction; endseq

       devices_ready_reg <= True;
    endseq);
   // end of the state machine definition

   // the statemachine is instantiated using mkFSM
   FSM speed_change_fsm <- mkFSM(speed_change_stmt);

   // the rule change_speed starts the state machine
   // the rule checks that previous actions of the state machine have completed
   rule change_speed ((device_xcvr_speed_reg != new_speed || !outofReset_reg) &&
      speed_change_fsm.done);
      speed_change_fsm.start;
   endrule

   method xcvrspeed = device_xcvr_speed_reg;
   method devices_ready = devices_ready_reg;
   method out_of_reset = outofReset_reg;
endmodule
```

Example - Defining a state machine and using the `await` function

```
   // This statement defines this brick's desired behavior as a state machine:
   // the subcomponents are to be executed one after the other:
   Stmt brickAprog =
     seq
        // Since the following loop will be executed over many clock
```



```
      // cycles, its control variable must be kept in a register:
      for (i <= 0; i < 0-1; i <= i+1)
         // This sequence requests a RAM read, changing the state;
         // then it receives the response and resets the state.
         seq
            action
               // This action can only occur if the state is Idle
               // the await function will not let the statements
               // execute until the condition is met
               await(ramState==Idle);
               ramState <= DesignReading;
               ram.request.put(tagged Read i);
            endaction
            action
               let rs <- ram.response.get();
               ramState <= Idle;
               obufin.put(truncate(rs));
            endaction
         endseq
      // Wait a little while:
      for (i <= 0; i < 200; i <= i+1)
         action
         endaction
      // Set an interrupt:
      action
         inrpt.set;
      endaction
   endseq
   );
// end of the state machine definition

FSM brickAfsm  <-  mkFSM#(brickAprog);  //instantiate the state machine

// A register to remember whether the FSM has been started:
Reg#(Bool) notStarted();
mkReg#(True) the_notStarted(notStarted);

// The rule which starts the FSM, provided it hasn't been started
// previously and the brick is enabled:
rule start_Afsm (notStarted && enabled);
   brickAfsm.start;                //start the state machine
   notStarted <= False;
endrule
```

C.4 Connecting Modules

The packages in this section, `GetPut`, `Connectable`, `ClientServer`, `CGetPut`, and `BGetPut` provide components, primarily interfaces, which are useful and easy, to connect hardware elements in a design.

The basic interfaces, `Get` and `Put` are defined in the package `GetPut`. The typeclass `Connectable` indicates that two related types can be connected together. The package `ClientServer` provides interfaces using `Get` and `Put` for modules that have a request-response type of interface. The packages

`CGetPut` and `BGetPut` define types of `Get` and `Put` interfaces that can be connected directly with wires and without additional hardware between the interfaces.

C.4.1 GetPut

Package Name

import GetPut :: *;

Description

`Get` and `Put` are simple interfaces, consisting of one method each, `get` and `put`, respectively. This package provides the interfaces `Get`, `Put`, and `GetPut`. This package also provides modules which provide the `GetPut` interface as a `FIFO` implementation, but these interfaces can be used in many additional hardware implementations.

Interfaces and methods

The `Get` interface defines a `get` method, similar to a `dequeue`, which retrieves an item from an interface and removes it at the same time. The `Put` interface defines a `put` method, similar to an `enqueue`, which gives an item to an interface. A module providing these interfaces can be designed to have implicit conditions on the `get`/`put` to ensure that the `get`/`put` is not performed when the module is not ready. This would ensure that a rule containing `get` method would not fire if the element associated with it is empty and that a rule containing `put` method would not fire if the element is full.

Interfaces			
Interface Name	Parameter name	Parameter Description	Restrictions
`Get`	*element_type*	type of the element being retrieved by the `Get`	must be in `Bits` class
`Put`	*element_type*	type of the element being added by the `Put`	must be in `Bits` class
`GetPut`	*element_type*	type of the element being retrieved and added	must be in `Bits` class

`Get`

The `Get` interface is where you retrieve (get) data from an object. The `Get` interface is provides a single method, `get`, which retrieves an item of data from an interface and removes it from the object. A `get` is similar to a `dequeue`, but it can be associated with any interface. A `Get` interface is more abstract than a `FIFO` interface; it does not describe the underlying hardware.

Get				
Method			Argument	
Name	Type	Description	Name	Description
`get`	ActionValue	returns an item from an interface and removes it from the object		

```
interface Get#(type element_type);
    method ActionValue#(element_type) get();
endinterface: Get
```

Example - adding your own `Get` interface:


```
module mkMyFifoUpstream (Get#(int));
...
   method ActionValue#(int) get();
       f.deq;
       return f.first;
   endmethod
```

`Put`

The `Put` interface is where you can give (put) data to an object. The `Put` interface provices a single method, `put`, which gives an item to an interface. A `put` is similar to a `enqueue`, but it can be associated with any interface. A `Put` interface is more abstract than a `FIFO` interface; it does not describe the underlying hardware.

Put				
Method			Argument	
Name	Type	Description	Name	Description
`put`	Action	gives an item to an interface	x1	data to be added to the object must be of type `element_type`

```
interface Put #(type a);
    method Action put(a x1);
endinterface: Put
```

Example - adding your own `Put` interface:

```
module mkMyFifoDownstream (Put#(int));
...
   method Action put(int x);
       F.enq(x);
   endmethod
```

`GetPut`

The library also defines an interface `GetPut` which associates `Get` and `Put` interfaces into a `Tuple2`.

```
typedef Tuple2 #(Get#(element_type), Put#(element_type)) GetPut #(type element_type);
```

Type classes

The class `Connectable` (Section C.4.2) is meant to indicate that two related types can be connected in some way. It does not specify the nature of the connection.

A `Get` and `Put` is an example of connectable items. One object will `put` an element into the interface and the other object will `get` the element from the interface.

```
instance Connectable #(Get#(a), Put#(a));
```

Modules

There are three modules provided by the `GetPut` package which provide the `GetPut` interface with a type of `FIFO`. These FIFOs use `Get` and `Put` interfaces instead of the usual `enq` interfaces. To use any of these modules the `FIFO` package must be imported. You can also write your own modules providing a `GetPut` interface for other hardware structures.

mkGPFIFO	Creates a FIFO of depth 2 with a GetPut interface.
	`module mkGPFIFO (GetPut#(element_type))` `  provisos (Bits#(element_type, width_elem));`

mkGPFIFO1	Creates a FIFO of depth 1 with a GetPut interface.
	`module mkGPFIFO1 (GetPut#(element_type))` `  provisos (Bits#(element_type, width_elem));`

mkGPSizedFIFO	Creates a FIFO of depth n with a GetPut interface.
	`module mkGPSizedFIFO# (Integer n) (GetPut#(element_type))` `  provisos (Bits#(element_type, width_elem));`

Functions

There are two functions defined in the `GetPut` package that change a `FIFO` interface to a `Get` or `Put` interface. Given a FIFO we can use the function `fifoToGet` to obtain a `Get` interface, which is a combination of `deq` and `first`. Given a FIFO we can use the function `fifoToPut` to obtain a `Put` interface using `enq`.

The package defines an additional function, `peekGet`, which returns the first item without removing it from the object. There are scheduling concerns when using `peekGet`; because of the implicit condition, it will only fire if there is data available.

fifoToGet	Returns a Get interface.
	`function Get#(element_type) fifoToGet(FIFO#(element_type) f);`

fifoToPut	Returns a Put interface.
	`function Put#(element_type) fifoToPut(FIFO#(element_type) f);`

peekGet	Returns first item without removing it from the object. Will not fire if data is not available.
	`function element_type peekGet(Get#(element_type) g;)`

Example of creating a FIFO with a `GetPut` interface


```
import GetPut::*;
import FIFO::*;

...
module mkMyModule (MyInterface);
   GetPut #(StatusInfo) aFifoOfStatusInfoStructures <- mkGPFIFO;
...
endmodule: mkMyModule
```

Example of a protocol monitor

This is an example of how you might write a protocol monitor that watches bus traffic between a bus and a bus target device

```
import GetPut::*;
import FIFO::*;

// Watch bus traffic beteween a bus and a bus target
interface ProtocolMonitorIfc;
   // These subinterfaces are defined inside the module
   interface Put#(Bus_to_Target_Request)  bus_to_targ_req_ifc;
   interface Put#(Target_to_Bus_Response) targ_to_bus_resp_ifc;
endinterface
...
module mkProtocolMonitor (ProtocolMonitorIfc);
   // Input FIFOs that have Put interfaces added a few lines down
   FIFO #(Bus_to_Target_Request) bus_to_targ_reqs <- mkFIFO;
   FIFO #(Target_To_Bus_Response) targ_to_bus_resps <- mkFIFO;
...
   // Define the subinterfaces: attach Put interfaces to the FIFOs, and
   //   then make those the module interfaces
   interface bus_to_targ_req_ifc = fifoToPut (bus_to_targ_reqs);
   interface targ_to_bus_resp_ifc = fifoToPut (targ_to_bus_resps);
end module: mkProtocolMonitor

// Top-level module: connect mkProtocolMonitor to the system:
module mkSys (Empty);
   ProtocolMonitorIfc pmon <- mkProtocolInterface;
...
   rule pass_bus_req_to_interface;
       let x <- bus.bus_ifc.get;     // definition not shown
       pmon.but_to_targ_ifc.put (x);
   endrule
...
endmodule: mkSys
```

C.4.2 Connectable

Package Name

import Connectable :: * ;

Description

The `Connectable` package contains the definitions for the class `Connectable` and two instances of `Connectables`; for `Tuple2`s and `Vectors`.

Types and Type-Classes

The class `Connectable` is meant to indicate that two related types can be connected in some way. It does not specify the nature of the connection. The `Connectables` type class defines the module `mkConnection`, which is used to connect the pairs.

```
typeclass Connectable #(type a, type b)
    module mkConnection#(a x1, b x2)(Empty);
endtypeclass
```

An example of connectable items is a `Get` and `Put`. One object will `put` an element into an interface and the other object will `get` the element from the interface.

```
instance Connectable #(Get#(a), Put#(a));
```

If we have `Tuple2` of connectable items then the pair is also connectable, simply by connecting the individual items.

```
instance Connectable #(Tuple2 #(a, c), Tuple2 #(b, d))
  provisos (Connectable#(a, b), Connectable#(c, d));
```

The proviso shows that the first component of one tuple connects to the first component of the other tuple, likewise, the second components connect as well. In the above statement, `a` connects to `b` and `c` connects to `d`. This is used by `ClientServer` (Section C.4.3) to connect the `Get` of the `Client` to the `Put` of the `Server` and visa-versa.

Two `Vectors` are connectable if their elements are connectable.

```
instance Connectable #(Vector#(n, a), Vector#(n, b))
  provisos (Connectable#(a, b));
```

C.4.3 ClientServer

Package Name

import ClientServer :: * ;

Description

The `ClientServer` package provides two interfaces, `Client` and `Server` which can be used to define modules which have a request-response type of interface. The `GetPut` package must be imported when using this package because the `Get` and `Put` interface types are used.

Interfaces and methods

The interfaces `Client` and `Server` can be used for modules that have a request-response type of interface (e.g. a RAM). The server accepts requests and generates responses, the client accepts responces and generates requests. There are no assumptions about how many (if any) responses a request generates

Interfaces			
Interface Name	Parameter name	Parameter Description	Restrictions
`Client`	*req_type*	type of the client request	must be in the Bits class
	resp_type	type of the client response	must be in the Bits class
`Server`	*req_type*	type of the server request	must be in the Bits class
	resp_type	type of the server response	must be in the Bits class

`Client`

The `Client` interface provides two sub-interfaces, `request` and `response`. From a `Client`, one `gets` a request and `puts` a response.

Client SubInterface		
Name	Type	Description
`request`	`Get#(req_type)`	the interface through which the outside world retrieves (gets) a request
`response`	`Put#(resp_type)`	the interface through which the outside world returns (puts) a response

```
interface Client #(type req_type, type resp_type);
    interface Get#(req_type)  request;
    interface Put#(resp_type) response;
endinterface: Client
```

`Server`

The `Server` interface provides two sub-interfaces, `request` and `response`. From a `Server`, one `puts` a request and `gets` a response.

Server SubInterface		
Name	Type	Description
`request`	`Put#(req_type)`	the interface through which the outside world returns (puts) a request
`response`	`Get#(resp_type)`	the interface through which the outside world retrieves (gets) a response

```
interface Server #(type req_type, type resp_type);
    interface Put#(req_type)  request;
    interface Get#(resp_type) response;
endinterface: Server
```

`ClientServer`

A `Client` can be connected to a `Server` and vice versa. The `request` (which is a `Get` interface) of the client will connect to `response` (which is a `Put` interface) of the `Server`. By making the `ClientServer` tuple an instance of the `Connectable` typeclass, you can connect the `Get` of the client to the `Put` of the server, and the `Put` of the client to the `Get` of the server.

```
instance Connectable #(Client#(req_type, resp_type), Server#(req_type, resp_type));
instance Connectable #(Server#(req_type, resp_type), Client#(req_type, resp_type));
```

This `Tuple2` can be redefined to be called `ClientServer`

```
typedef Tuple2 #(Client#(req_type, resp_type), Server#(req_type,resp_type))
                ClientServer #(type req_type, type resp_type);
```

Example Connecting a bus to a target

```
interface Buf_Ifc;
   interface Server#(RQ, RS) to_targ ;
   interface Client#(RQ, RS) to_initiator;
endinterface

typedef Server#(RQ, RS) Target_Ifc;
typedef Client#(RQ, RS) Initiator_Ifc;
```



```
module mkSys (Empty);
   // Instantiate subsystems
   Bus_Ifc             bus       <- mkBus;
   Target_Ifc          targ      <- mkTarget;
   Initiator_Ifc       initor    <- mkInitiator;

   // Connect bus and targ ("to_targ" is a Get i/f, targ is a Put i/f)
   Empty x <- mkConnection (bus.to_targ, targ);

   // Connect bus and initiator ("to_initor" is a Out i/f, initor is a Get i/f)
   mkConnection (bus.to_initor, initor);
   // Since mkConnection returns an interface of type Empty, it does
   // not need to be specified (but may be as above)

...
endmodule: mkSys
```

C.4.4 CGetPut

The interfaces `CGet` and `CPut` are similar to `Get` and `Put`, but the interconnection of them (via `Connectable`) is implemented with a credit based FIFO. This means that the `CGet` and `CPut` interfaces have completely registered input and outputs, and furthermore that additional register buffers can be introduced in the connection path without any ill effect (except an increase in latency, of course). The interface types are abstract (to avoid any non-proper use of the credit signaling protocol). In the absence of additional register buffers, the round-trip time for communication between the two interfaces is 4 clock cycles. Call this number r. The first argument to the type, n, specifies that transfers will occur for a fraction n/r of clock cycles (note that the used cycles will not necessarily be evenly spaced). n also specifies the depth of the buffer used in the receiving interface (the transmitter side always has only a single buffer). So (in the absence of additional buffers) use $n = 4$ to allow full-bandwidth transmission, at the cost of sufficient registers for quadruple buffering at one end; use $n = 1$ for minimal use of registers, at the cost of reducing the bandwidth to one quarter; use intermediate values to select the optimal trade-off if appropriate.

Note

For compiler reasons the actual interfaces are called `CGetS` and `CPutS` with `CGet` and `CPut` being type abbreviations. Hopefully this will be fixed soon.

```
typedef CGetS#(n, a, SizeOf#(a)) CGet #(type n, type a);

typedef CPutS#(n, a, SizeOf#(a)) CPut #(type n, type a);
```

Create one end of the credit based FIFO. Access to it is via a `Put` interface.

```
module mkCGetPut(Tuple2 #(CGetS#(n, a, sa), Put#(a)))
  provisos (Bits#(a, sa), Add#(1, k, n), Add#(n, 1, n1), Log#(n1, ln));
```

Create the other end of the credit based FIFO. Access to it is via a `Get` interface.

```
module mkGetCPut(Tuple2 #(Get#(a), CPutS#(n, a, sa)))
  provisos (Bits#(a, sa), Add#(1, k, n), Log#(n, ln));
```

Create a buffer that can be inserted along a connection path.

```
module mkCGetCPut(Tuple2 #(CGetS#(n, a, sa), CPutS#(n, a, sa)))
  provisos (Bits#(a, sa));
```

The `CGet` and `CPut` interface are connectable.


```
instance Connectable #(CGetS#(n, a, sa), CPutS#(n, a, sa));

instance Connectable #(CPutS#(n, a, sa), CGetS#(n, a, sa));
```

The same idea may be extended to clients and servers.

```
typedef CClientS#(n, a, SizeOf#(a), b, SizeOf#(b))
                                          CClient #(type n, type a, type b);
typedef CServerS#(n, a, SizeOf#(a), b, SizeOf#(b))
                                          CServer #(type n, type a, type b);

module mkClientCServer(Tuple2 #(Client#(a, b), CServerS#(n, a, sa, b, sb)))
  provisos (Bits#(a, sa), Bits#(b, sb), Add#(1, k, n));

module mkCClientServer(Tuple2 #(CClientS#(n, a, sa, b, sb), Server#(a, b)))
  provisos (Bits#(a, sa), Bits#(b, sb), Add#(1, k, n));
```

C.4.5 BGetPut

The interfaces `BGet` and `BPut` are similar to `Get` and `Put`, but the interconnection of them (via `Connectable` or in Verilog) is implemented with a simple protocol that allows all inputs and outputs to be directly connected. Furthermore, all wires go to registers and have no Bluespec SystemVerilog handshaking. The protocol makes no assumptions about setup time and hold time for the registers at each end; so these interfaces may be used when the two ends have different clocks. In all other circumstances, however, the `CGetPut` package will probably be preferable. In particular, the BGetPut protocol is very slow. The protocol consist of the sender putting the value to be sent on the `pvalue` output, and then toggling the `ppresent` wire. The receiver acknowledges the receipt by toggling the `gcredit` wire. Both `ppresent` and `gcredit` start out low.

```
interface BGetS #(type sa);
    method Bit#(sa) gvalue();
    method Bool gpresent();
    method Action gcredit(Bool x1);
endinterface: BGetS

interface BGetS #(type sa);
    method Bit#(sa) gvalue();
    method Bool gpresent();
    method Action gcredit(Bool x1);
endinterface: BGetS

typedef BGetS#(SizeOf#(a)) BGet #(type a);
typedef BPutS#(SizeOf#(a)) BPut #(type a);
typedef Tuple2 #(BGet#(a), Put#(a)) BGetPut #(type a);
typedef Tuple2 #(Get#(a), BPut#(a)) GetBPut #(type a);
```

Create one end of the buffer. Access to it is via a `Put` interface.

```
module mkBGetPut(Tuple2 #(BGetS#(sa), Put#(a)))
  provisos (Bits#(a, sa));
```

Create the other end of the buffer. Access to it is via a `Get` interface.

```
module mkGetBPut(Tuple2 #(Get#(a), BPutS#(sa)))
  provisos (Bits#(a, sa));
```

The `BGet` and `BPut` interface are connectable.


```
instance Connectable #(BGetS#(sa), BPutS#(sa));

instance Connectable #(BPutS#(sa), BGetS#(sa));
```

The same idea may be extended to clients and servers.

```
typedef BClientS#(SizeOf#(a), SizeOf#(b)) BClient #(type a, type b);
typedef BServerS#(SizeOf#(a), SizeOf#(b)) BServer #(type a, type b);
typedef Tuple2 #(BClient#(a, b), Server#(a, b)) BClientServer #(type a, type b);
typedef Tuple2 #(Client#(a, b), BServer#(a, b)) ClientBServer #(type a, type b);
```

A `BClient` can be connected to a `BServer` and vice versa.

```
instance Connectable #(BClientS#(a, b), BServerS#(a, b));
instance Connectable #(BServerS#(a, b), BClientS#(a, b));


module mkClientBServer(Tuple2 #(Client#(a, b), BServerS#(sa, sb)))
  provisos (Bits#(a, sa), Bits#(b, sb));


module mkBClientServer(Tuple2 #(BClientS#(sa, sb), Server#(a, b)))
  provisos (Bits#(a, sa), Bits#(b, sb));
```

C.5 Useful Circuits

C.5.1 LFSR

The `LFSR` package implements Linear Feedback Shift Registers (LFSRs). LFSRs can be used to obtain pseudorandom sequences, though their linearity permits easy cryptanalysis.[8] The interface allows the value in the shifter register to be set (with `seed`), read (with `value`), and shifted (with `next`). When the value is shifted the least significant bit will be fed back according to the polynomial used when the LFSR was created. When a LFSR is created the start value is 1.

```
interface LFSR #(type a);
    method Action seed(a x1);
    method a value();
    method Action next();
endinterface: LFSR
```

The `mkPolyLFSR` function creates a LFSR given a polynomial specified by the exponents that have a non-zero coefficient. For example the polynominal $x^7 + x^3 + x^2 + x$ is used by the expression "`mkPolyLFSR (Cons(7, Cons(3, Cons(2, Cons(1, Nil)))))`".

```
module mkPolyLFSR#(List#(Integer) taps) (LFSR#(Bit#(n)));
```

The `mkFeedLFSR` function creates a LFSR where the polynomial is specified by the mask used for feedback. If "`r`" is the state of the LFSR the next state is "`if r[0] == 1`" "`then (r >> 1) ^ feed`" "`else r >> 1`", where "`feed`" is the argument to `mkFeedLFSR`.

```
module mkFeedLFSR#( Bit#(n) feed )( LFSR#(Bit#(n)) );
```

Some maximal length LFSRs. Many more can be found at `http://www-2.cs.cmu.edu/ ~koopman/lfsr/`

[8] see `http://en.wikipedia.org/wiki/Linear_feedback_shift_register` for details


```
module mkLFSR_4 (LFSR#(Bit#(4)));
mkLFSR_4   =  mkFeedLFSR( 4'h9 );

module mkLFSR_8 (LFSR#(Bit#(8)));
mkLFSR_8   =  mkFeedLFSR( 8'h8E );

module mkLFSR_16 (LFSR#(Bit#(16)));
mkLFSR_16  =  mkFeedLFSR( 16'h8016 );

module mkLFSR_32 (LFSR#(Bit#(32)));
mkLFSR_32  =  mkFeedLFSR( 32'h80000057 );
```

The `mkRCounter` function creates a counter with a LFSR interface. This is useful during debugging when a non-random sequence is desired. This function can be used in place of the other mkLFSR module constructors, without changing any method calls or behavior.

```
module mkRCounter#( Bit#(n) seed ) ( LFSR#(Bit#(n)) );
```

C.5.2 CompletionBuffer

A `CompletionBuffer` is like a FIFO except that entering elements can be done out-of-order. To enter something into the completion buffer a token is necessary. A token can be obtained with the `reserve` method. This token is then used in the `complete` method to enter the actual item. Finally, the `drain` method takes items out of the buffer; the items are delivered in the order of the tokens that were checked out.

The n represents the size of the completion buffer, and a is the item type.

```
interface CompletionBuffer #(type n, type a);
    method Get#(CBToken#(n)) reserve();
    method Put#(Tuple2 #(CBToken#(n), a)) complete();
    method Get#(a) drain();
endinterface: CompletionBuffer
```

The `CBToken` type is abstract to avoid confusion.

```
typedef union tagged { ... } CBToken #(type n) ...;
```

The `mkCompletionBuffer` function creates a completion buffer. The `mkCompletionBuffer` function creates a completion buffer.

```
module mkCompletionBuffer(CompletionBuffer#(n, a))
  provisos (Bits#(a, sa), Log#(n, ln), Log#(n, TLog#(n)), Add#(1, ln, ln1));
```

C.5.3 UniqueWrappers

Package

import UniqueWrappers :: * ;

Description

The `UniqueWrappers` package takes a piece of combinational logic which is to be shared and puts it into its own protective shell or *wrapper* to prevent its duplication. This is used in instances where a separately synthesized module is not possible. It allows the designer to use a piece of logic at several places in a design without duplicating it at each site.

There are times where it is desired to use a piece of logic at several places in a design, but it is too bulky or otherwise expensive to duplicate at each site. Often the right thing to do is to make the

piece of logic into a separately synthesized module – then, if this module is instantiated only once, it will not be duplicated, and the tool will automatically generate the scheduling and multiplexing logic to share it among the sites which use its methods. Sometimes, however, this is not convenient. One reason might be that the logic is to be incorporated into a sub-module of the design which is itself polymorphic – this will probably cause difficulties in observing the constraints necessary for a module which is to be separately synthesized. And if a module is *not* separately synthesized, the tool will inline its logic freely wherever it is used, and thus duplication will not be prevented as desired.

This package covers the case where the logic to be shared is combinational and cannot be put into a separately synthesized module. It may be thought of as surrounding this combinational function with a protective shell, a *unique wrapper*, which will prevent its duplication. The module `mkUniqueWrapper` takes a one-argument function as a parameter; both the argument type `a` and the result type `b` must be representable as bits, that is, they must both be in the `Bits` typeclass.

Interfaces

The `UniqueWrappers` package provides an interface, `Wrapper`, with one actionvalue method, `func`, which takes an argument of type `a` and produces a method of type `ActionValue#(b)`. If the module is instantiated only once, the logic implementing its parameter will be instantiated just once; the module's method may, however, be used freely at several places.

Although the function supplied as the parameter is purely combinational and does not change state, the method is of type `ActionValue`. This is because actionvalue methods have `enable` signals and these signals are needed to organize the scheduling and multiplexing between the calling sites.

Variants of the interface `Wrapper` are also provided for handling functions of two or three arguments; the interfaces have one and two extra parameters respectively. In each case the result type is the final parameter, following however many argument type parameters are required.

Wrapper Interfaces	
`Wrapper`	This interface has one actionvalue method, `func`, which takes an argument of type `a_type` and produces an actionvalue of type `ActionValue#(b_type)`.
	`interface Wrapper#(type a_type, type b_type);` `  method ActionValue#(b_type) func (a_type x);`

`Wrapper2`	Similar to the `Wrapper` interface, but it takes two arguments.
	`interface Wrapper2#(type a1_type, type a2_type, type b_type);` `  method ActionValue#(b_type) func (a1_type x, a2_type y);`

`Wrapper3`	Similar to the `Wrapper` interface, but it takes three arguments.

	`interface Wrapper3#(type a1_type, type a2_type, type a3_type,`
	`  type b_type);`
	`  method ActionValue#(b_type) func (a1_type x, a2_type y, a3_type z);`

Modules

The interfaces `Wrapper`, `Wrapper2`, and `Wrapper3` are provided by the modules `mkUniqueWrapper`, `mkUniqueWrapper2`, and `mkUniqueWrapper3`. These modules vary only in the number of aguments in the parameter function.

If a function has more than three arguments, it can always be rewritten or wrapped as one which takes the arguments as a single tuple; thus the one-argument version `mkUniqueWrapper` can be used with this function.

`mkUniqueWrapper`	
	Takes a function, `func`, with a single parameter `x` and provides the interface `Wrapper`.

	`module mkUniqueWrapper#(function b_type func(a_type x))`
	`  (Wrapper#(a_type, b_type))`
	`  provisos (Bits#(a_type, sizea), Bits#(b_type, sizeb));`

`mkUniqueWrapper2`	
	Takes a function, `func`, with a two parameters, `x` and `y`, and provides the interface `Wrapper2`.

	`module mkUniqueWrapper2#(function b_type func(a1_type x, a2_type y))`
	`  (Wrapper2#(a1_type, a2_type, b_type))`
	`  provisos (Bits#(a1_type, sizea1), Bits#(a2_type, sizea2),`
	`  Bits#(b_type, sizeb));`

`mkUniqueWrapper3`	
	Takes a function, `func`, with a three parameters, `x`, `y`, and `z`, and provides the interface `Wrapper3`.

	`module mkUniqueWrapper3#(function b_type`
	`  func(a1_type x, a2_type y, a3_type z))`
	`  (Wrapper3#(a1_type, a2_type, a3_type, b_type))`
	`  provisos (Bits#(a1_type, sizea1), Bits#(a2_type, sizea2),`
	`  Bits#(a3_type, sizea3), Bits#(b_type, sizeb));`

Example: Complex Multiplication

```
// This module defines a single hardware multiplier, which is then
// used by multiple method calls to implement complex number
// multiplication (a + bi)(c + di)

typedef Int#(18) CFP;

module mkComplexMult1Fifo( ArithOpGP2#(CFP) ) ;
  FIFO#(ComplexP#(CFP))  infifo1 <- mkFIFO;
  FIFO#(ComplexP#(CFP))  infifo2 <- mkFIFO;
  let arg1 = infifo1.first ;
  let arg2 = infifo2.first ;

  FIFO#(ComplexP#(CFP))  outfifo <- mkFIFO;

  Reg#(CFP)  rr <- mkReg(0) ;
  Reg#(CFP)  ii <- mkReg(0) ;
```



```
  Reg#(CFP)  ri <- mkReg(0) ;
  Reg#(CFP)  ir <- mkReg(0) ;

  // Declare and instantiate an interface that takes 2 arguments, multiplies them
  // and returns the result.  It is a Wrapper2 because there are 2 arguments.
  Wrapper2#(CFP,CFP, CFP) smult <- mkUniqueWrapper2( \* ) ;

  // Define a sequence of actions
  // Since smult is a UnqiueWrapper the method called is smult.func
  Stmt multSeq =
  seq
     action
        let mr <- smult.func( arg1.rel,  arg2.rel ) ;
        rr <= mr ;
     endaction
     action
        let mr <- smult.func( arg1.img, arg2.img ) ;
        ii <= mr ;
     endaction
     action
        // Do the first add in this step
        let mr <- smult.func( arg1.img,  arg2.rel ) ;
        ir <= mr ;
        rr <= rr - ii ;
     endaction
     action
        let mr <- smult.func( arg1.rel, arg2.img );
        ri <= mr ;
        // We are done with the inputs so deq the in fifos
        infifo1.deq ;
        infifo2.deq ;
     endaction
     action
        let ii2 = ri + ir ;
        let res = Complex{ rel: rr , img: ii2 } ;
        outfifo.enq( res ) ;
     endaction
  endseq;

  // Now convert the sequence into a FSM ;
  //  Bluespec can assign the state variables, and pick up implict
  //  conditions of the actions
  FSM multfsm <- mkAutoFSM;
  rule startFSM;
     multfsm.start;
  endrule
endmodule
```

C.6 Local Bus Access

C.6.1 LBus

The `LBus` package provides a way to create registers that are accessible through some type of local bus (e.g., PCI).

The LBSReg type is normally never seen by a user of the LbSModule; it is only needed when creating new kinds of local bus registers. This LBSReg interface is what the local bus uses to access a register.

```
interface LBSReg #(type sa, type sd);
    method LbAddr#(sa)            lbsAddr();
    method Action                 lbsSet( Bit#(sd) x1);
    method ActionValue#(Bit#(sd)) lbsGet();
endinterface: LBSReg
```

Note that the lbsGet method allows an action to be performed when the local bus reads the value. This allows implementing, e.g., clear-on-read registers.

The type LbAddr is the address used to get and set register from the local bus. (This type is exported abstractly.)

```
typedef union tagged {
    Bit#(sa) LbAddr;
} LbAddr #(type sa) deriving (Literal, Eq, Bits);
```

The local bus registers are collected automagically by ModuleCollect monad. An LbSModule#(sa,sd,i) corresponds to a Module#(i) except that it also keeps a set of registers. The address is *sa* bits wide and data items are *sd* bits wide.

```
typedef ModuleCollect#(LBSItem#(sa, sd), i) LbSModule#(type sa, type sd, type i);
```

- **Creating Registers** The mkLbRegRW module creates a register that looks like a normal register in the module that creates it, but it is also accessible from the local bus at the given address.

  ```
  module [LbSModule#(sa, sd)]
     mkLbRegRW#( LbAddr#(sa) aw, Integer an, r_type x)
                ( Reg#(r_type))
     provisos (Bits#(r_type, sr), Add#(k, sr, sd));
  ```

  The mkLbRegRO module creates a register that looks like a normal register in the module that creates it, but it is also accessible from the local bus at the given address. From the local bus the register is read-only; attempts to write it have no effect. The created register has to have a bit width smaller than or equal to the local bus width. If it is smaller it will padded with zeroes on the left.

  ```
  module [LbSModule#(sa, sd)] mkLbRegRO#(LbAddr#(sa) aw, Integer an, r x)(Reg#(r))
     provisos (Bits#(r, sr), Add#(k, sr, sd));

  interface Accum #(type n);
      method Action add(Bit#(n) x1);
      method Bit#(n) value();
  endinterface: Accum

  module [LbSModule#(sa, sd)] mkLbAccum#(LbAddr#(sa) aw, Integer an, Bit#(k) x)(Accum#(k))
     provisos (Add#(k, i, sd));
  ```

  The mkLbOffset function can be used to add an offset to all local bus register addresses in an LbSModule.

  ```
  module [LbSModule#(sa,sd)] mkLbOffset#(LbAddr#(sa) a, LbSModule#(sa, sd, i) m)(i);
  ```

- **Collecting registers together** The external interface of a local bus is as follows. It is through this interface that register accesses normally happen.


```
interface ILBus #(type sa, type sd);
    method Action req(Bool valid, LbRWop op, Bit#(sa) addr, Bit#(sd) dat);
    method Bit#(sd) rdDat();
    method Bit#(1) ack();
    method Bit#(1) inrpt();
endinterface: ILBus

interface ILbLeaf#(type sa, type sd);

interface ILbNode#(type sa, type sd);

instance Connectable#(ILbLeaf#(sa, sd), ILbNode#(sa, sd));
```

Given a `LbSModule` with a set of register we can extract the local bus interface and the normal interface.

```
module [Module] mkLbLeaf#(LbSModule#(sa, sd, i) lm)(IWithLBus#(ILbLeaf#(sa, sd), i));
```

The `LbSModule` is used to collect individual registers. Once the registers have been collected into an `ILbLeaf` interface these interfaces can be collected together.

```
typedef ModuleCollect#(ILbLeaf#(sa, sd), i) LbAModule#(type sa, type sd, type i);
```

The `mkLbBranch` module make a `LbAModule` out of the result from `mkLbLeaf`.

```
module [LbAModule#(sa,sd)] mkLbBranch#(Module#(IWithLBus#(ILbLeaf#(sa, sd), i)) m)(i);
```

The `mkLbTop` module combines local bus register clusters. It introduces a one cycle latency on both request and response.

```
module [Module] mkLbTop#(Module#(Fan#(ILBus#(sa, sd), Vector#(n, ILbNode#(sa, sd)))) mkFanout,
  LbAModule#(sa, sd, i) lm) (IWithLBus#(ILBus#(sa, sd), i));
```

C.7 Multiple Clock Domains and Clock Generators

The BSV `Clocks` library provide features to access and change the default clock. Moreover, there are hardware primitives to generate clocks of various shapes, plus several primitives which allow the safe crossing of signals and data from one clock domain to another.

C.7.1 Clock Generators and Manipulation

`mkAbsoluteClock` provides a parameterizable clock generation module, with its first rising edge (start) and period defined by parameters. This module is not synthesizable.

```
module mkAbsoluteClock #( Integer start,
                          Integer period
                        ) ( Clock );
```

`mkAbsoluteClockFull` provides a fully parameterizable clock generation module. `initValue` is held until time start, and then the clock oscillates with from, with `not(initValue)` held for time `compValTime` followed by `initValue` held for time `initValTime`. Hence the clock period after startup is `compValTime + initValTime`. This module is not synthesizable, it is provided by the Verilog module `ClockGen.v`

```
module mkAbsoluteClockFull #( Integer start,
                              Bit#(1) initValue,
                              Integer compValTime,
                              Integer initValTime
                            ) ( Clock );
```


The mkClock module creates a Clock type from a one-bit oscillator input, and a Boolean gate condition. There are no relations between the current clock and the clock generated by this module.

```
module mkClock ( one_bit_type oscillator, Bool gate, Clock clkout)
   provisos(  Bits#(one_bit_type, 1)) ;
```

The `mkGatedClock` module adds (logic and) a Boolean gate condition to an existing clock, thus creating a another clock in the same family. The gate condition is controlled by the tick Action method of the interface, while the source clock is the current clock of the module which instantiates this module (or the "clocked_by" argument of mkGatedClock itself).

```
module mkGatedClock ( Bool b, Clock clkout );
```

An alternate interface for the module

```
module mkGatedClock2 ( Clock clk_in, Bool gate, Clock clkout );
```

C.7.2 Clock Multiplexing

Bluespec provides two clock multiplexing primitives: a simple combinational multiplexor and a stateful module which generates an appropriate reset signal when the clock changes. The `mkClockMux` module is a simple combinational multiplexor, which selects between the aClk and bClk. The provided Verilog module does not provide any glitch detection or removal logic; it is the responsibility of the user to provide additional logic to provide glitch-free behavior. The `mkClockMux` module uses three arguments and provides a Clock interface. The `aClk` is selected if `ab` is True, while `bClk` is selected otherwise. The underlying Verilog module is `ClockMux.v`.

```
module mkClockMux ( Bool ab, Clock aClk, Clock bClk, Clock clkout ) ;
```

The `mkClockSelect` module is a clock multiplexor containing additional logic which generates a reset whenever a new clock is selected. As such the interface for the module includes an `Action` method to select the clock (if `ab` is True clock_out is taken from `aClk`), provides a `Clock` interface, and also a `Reset` interface. The interface definition is described here.

```
interface SelectClkIfc ;
   method    Action select ( Bool  ab ) ;
   interface Clock  clock_out ;
   interface Reset  reset_out ;
endinterface
```

The constructor for the module uses two clock arguments, and provides the `SelectClkIfc` interface. The underlying Verilog module is `ClockSelect.v`; it is expected that users can substitute their own modules to meet any additional requirements they may have. The parameter `stages` is the number of clock cycles in which the reset is asserted after the clock changes.

```
module mkClockSelect #( Integer stages
                      ) ( Clock aClk,
                          Clock bClk,
                          SelectClkIfc ifcout ) ;
```

C.7.3 Clock Division

A clock divider provide a derived clock and also a `ClkNextRdy` signal, which indicates that divided clock will rise in the next cycle. This signal is associated with the input clock, and can only be used within that clock domain.

See mkSyncRegToSlow, mkSyncRegToFast, mkSyncFIFOToSlow, and mkSyncFIFOToFast for some specialized synchronizers which can be used with divided clocks, and other systems when the clock edges are known to be aligned.

Define a new type and then the interface

```
typedef Bool ClkNextRdy ;

interface ClockDividerIfc ;
    interface Clock      fastClock ;           // The original clock
    interface Clock      slowClock ;           // The derived clock
    method    ClkNextRdy clockReady() ;        //
endinterface
```

The divider parameter may be any integer greater than 1. For even dividers the generated clock's duty cycle is 50%, while for odd dividers, the duty cycle is $(divider/2)/divider$. The current clock (or the clocked_by argument) is used as the source clock. The mkClockDividerOffset module provides a clock divider, where the rising edge can be defined relative to other clock dividers which have the same divisor. An offset of value 2, will produce a rising edge one fast clock after a divider with offset 1. The mkClockDivider modules are provided by the Verilog module ClockDiv.v

```
module mkClockDivider #( Integer divisor
                       )( ClockDividerIfc ifc ) ;

module mkClockDividerOffset #( Integer divisor,
                               Integer offset
                             )( ClockDividerIfc ifc ) ;
```

The mkClockInverter module generates a inverted clock having the same period but opposite phase as the source clock.

```
module mkClockInverter ( ClockDividerIfc ifc ) ;
```

C.7.4 Bit Synchronizers

The Sync Bit interface provides a send method which transmits one bit of information from one clock domain to the read method in a second domain.

```
interface SyncBitIfc #(type one_bit_type) ;
   method Action       send ( one_bit_type bitData ) ;
   method one_bit_type read () ;
endinterface
```

The mkSyncBit, mkSyncBitFromCC and mkSyncBitToCC modules provide a SyncBitIfc across clock domains. The send method is in the one clock domain, and the read method is in a second clock domain. The FromCC version and ToCC versions differ in that the former moves data from the current clock (module's clock), while the later move data into the current clock domain The hardware implementation is a two register synchronizer, which can be found in SyncBit.v in the Bluespec Verilog library directory. The mkSyncBit15 module (one and a half) and its variants provide the same interface as the mkSyncBit modules, but the underlying hardware is slightly modified. For these synchronizers, the first register clocked by the destination clock triggers on the falling edge of the clock. The Verilog can in found in SyncBit15.v in the Bluespec Verilog library directory. The mkSyncBit1 module also provides the same interface, but only uses one register in the destination domain. Synchronizers like this, which use with only one register, are not generally used since meta-stable output is very probable. However, one can use this synchronizer provided special meta-stable resistant flops are selected during physical synthesis or (for example) if the output is immediately registered. The mkSyncBit05 module is similar, but the destination register triggers on the falling edge of the clock.


```
module mkSyncBit ( Clock sClkIn, Reset sRst,
                   Clock dClkIn,
                   SyncBitIfc #(one_bit_type) ifc )
   provisos( Bits#(one_bit_type, 1)) ;

module mkSyncBitFromCC ( Clock dClkIn,
                         SyncBitIfc #(one_bit_type) ifc )
   provisos( Bits#(one_bit_type, 1)) ;

module mkSyncBitToCC ( Clock sClkIn, Reset sRstIn,
                       SyncBitIfc #(one_bit_type) ifc )
   provisos( Bits#(one_bit_type, 1)) ;

module mkSyncBit15 ( Clock sClkIn, Reset sRst,
                     Clock dClkIn,
                     SyncBitIfc #(one_bit_type) ifc )
   provisos( Bits#(one_bit_type, 1)) ;

module mkSyncBit15FromCC ( Clock dClkIn,
                           SyncBitIfc #(one_bit_type) ifc )
   provisos( Bits#(one_bit_type, 1)) ;

module mkSyncBit15ToCC ( Clock sClkIn, Reset sRstIn,
                         SyncBitIfc #(one_bit_type) ifc )
   provisos( Bits#(one_bit_type, 1)) ;

module mkSyncBit1 ( Clock sClkIn, Reset sRst,
                    Clock dClkIn,
                    SyncBitIfc #(one_bit_type) ifc )
   provisos( Bits#(one_bit_type, 1)) ;

module mkSyncBit1FromCC ( Clock dClkIn,
                          SyncBitIfc #(one_bit_type) ifc )
   provisos( Bits #(one_bit_type, 1)) ;

module mkSyncBit1ToCC ( Clock sClkIn, Reset sRstIn,
                        SyncBitIfc #(one_bit_type) ifc )
   provisos( Bits#(one_bit_type, 1)) ;

module mkSyncBit05 ( Clock sClkIn, Reset sRst,
                     Clock dClkIn,
                     SyncBitIfc #(one_bit_type) ifc )
   provisos( Bits#(one_bit_type, 1)) ;

module mkSyncBit05FromCC ( Clock dClkIn,
                           SyncBitIfc #(one_bit_type) ifc )
   provisos( Bits#(one_bit_type, 1)) ;

module mkSyncBit05ToCC ( Clock sClkIn, Reset sRstIn,
                         SyncBitIfc #(one_bit_type) ifc )
   provisos( Bits#(one_bit_type, 1)) ;
```


C.7.5 Pulse Synchronizers

The Sync Pulse interface provides an Action `send` method which when invoked causes a pulse on a `read` method in a second clock domain.

```
interface SyncPulseIfc ;
   method Action send  () ;
   method Bool   pulse () ;
endinterface
```

The `mkSyncPulse`, `mkSyncPulseFromCC` and `mkSyncPulseToCC` modules provide clock domain crossing modules for pulses. When the `send` method is called from the one clock domain, a pulse will be seen on the `read` method in the second. Note that there is no handshaking between the domains, so when sending data from a fast clock domain to a slower one, not all pulses sent may be seen in the slower receiving clock domain. The pulse delay is two destination clocks cycles. The hardware implementation can be found in SyncPulse.v in the Bluespec Verilog library directory.

```
module mkSyncPulse ( Clock sClkIn, Reset sRstIn,
                     Clock dClkIn,
                     SyncPulseIfc ifc ) ;

module mkSyncPulseFromCC ( Clock dClkIn,
                           SyncPulseIfc ifc ) ;

module mkSyncPulseToCC ( Clock sClkIn, Reset sRstIn,
                         SyncPulseIfc ifc ) ;
```

The `mkSyncHandshake`, `mkSyncHandshakeFromCC` and `mkSyncHandshakeToCC` modules provide clock domain crossing modules for pulses in a similar way as `mkSyncPulse` modules, except that a handshake is provided in the `mkSyncHandshake` versions. The handshake enforces that another send does not occur before the first pulse crosses to the other domain. The pulse delay from the `send` method to the `read` method is two destination clocks. The `send` method is re-enabled in two destination clock cycles plus two source clock cycles after the `send` method is called.

```
module mkSyncHandshake ( Clock sClkIn, Reset sRstIn,
                         Clock dClkIn,
                         SyncPulseIfc ifc ) ;

module mkSyncHandshakeFromCC ( Clock dClkIn,
                               SyncPulseIfc ifc ) ;

module mkSyncHandshakeToCC ( Clock sClkIn, Reset sRstIn,
                             SyncPulseIfc ifc ) ;
```

C.7.6 Word Synchronizers

Word synchronizers use the common `Reg` interface (redescribed below), but there are a few subtle differences which the designer should be aware. First, the *_read* and *_write* methods are in difference clock domains, and second the *_write* method has an implicit "ready" condition which means that some synchronization modules cannot be written every clock cycle. Both of these conditions are handled automatically by the Bluespec compiler relieving the designer of these checks.

```
interface Reg #(a_type);
    method Action _write(a_type x1);
    method a_type _read();
endinterface: Reg
```


The mkSyncReg, mkSyncRegToCC and mkSyncRegFromCC modules provide word synchronization across clock domains. The crossing are handshaked, such that a second write cannot occur until the first is acknowledge by the destination side. The destination read is registered. The hardware implementation can be found in SyncRegister.v in the Bluespec Verilog library directory.

```
module mkSyncReg #( a_type initValue
                  )( Clock sClkIn, Reset sRstIn,
                     Clock dClkIn,
                     Reg #(a_type) ifc )
   provisos (Bits#(a_type,sa) ) ;

module mkSyncRegFromCC #( a_type initValue
                        )( Clock dClkIn,
                           Reg #(a_type) ifc)
   provisos (Bits#(a_type,sa)) ;

module mkSyncRegToCC #( a_type initValue
                      )( Clock sClkIn, Reset sRstIn,
                         Reg #(a_type) ifc)
   provisos (Bits#(a_type,sa)) ;
```

C.7.7 FIFO Synchronizers

The sync FIFO interface defines an interface similar to the FIFO interface, except it does not have a clear method.

```
interface SyncFIFOIfc #(type a_type) ;
   method Action enq ( a_type sendData ) ;
   method Action deq () ;
   method a_type first () ;
endinterface
```

The mkSyncFIFO, mkSyncFIFOFromCC and mkSyncFIFOToCC modules provide FIFOs for sending data across clock domains. The enq method is in one domain, while the deq and first methods are in a second domain. Depth of the instantiated FIFO may be increased to the next power of 2, FIFOs of depth 1 are allowed. The hardware implementation can be found in SyncFIFO.v in the Bluespec Verilog library directory.

```
module mkSyncFIFO #( Integer depthIn
                   )( Clock sClkIn, Reset sRstIn,
                      Clock dClkIn,
                      SyncFIFOIfc #(a_type) ifc )
   provisos (Bits#(a_type,sa));
```

A variation of the Sync FIFO which allow the empty and full signals to be registered. Registering the signals can give better synthesis results, since a comparator is removed from the empty or full path. However, there is an additional cycle of latency before the empty or full signal is visible.

```
module mkSyncFIFOFull #( Integer depthIn,
                         Bool regEmpty,
                         Bool regFull
                       )( Clock sClkIn, Reset sRstIn,
                          Clock dClkIn,
                          SyncFIFOIfc #(a_type) ifc )
   provisos (Bits#(a_type,sa));
```



```
module mkSyncFIFOFromCC #( Integer depthIn
                         )( Clock dClkIn,
                            SyncFIFOIfc #(a_type) ifc)
   provisos (Bits#(a_type,sa));

module mkSyncFIFOToCC #( Integer depthIn
                       )( Clock sClkIn, Reset sRstIn,
                          SyncFIFOIfc #(a_type) ifc)
   provisos (Bits#(a_type,sa));

module mkSyncFIFOFromCCFull #( Integer depthIn,
                               Bool regEmpty,
                               Bool regFull
                             )( Clock dClkIn,
                                SyncFIFOIfc #(a_type) ifc)
   provisos (Bits#(a_type,sa));

module mkSyncFIFOToCCFull #( Integer depthIn,
                             Bool regEmpty,
                             Bool regFull
                           )( Clock sClkIn, Reset sRstIn,
                             SyncFIFOIfc #(a_type) ifc)
   provisos (Bits#(a_type,sa));
```

C.7.8 Asynchronous RAMs

```
interface DualPortRamIfc #(type addr_t, type data_t);
   method Action      write( addr_t wr_addr, data_t  din );
   method data_t      read ( addr_t rd_addr);
endinterface: DualPortRamIfc

module mkDualRam( DualPortRamIfc #(addr_t, data_t) )
   provisos ( Bits#(addr_t,sa),
              Bits#(data_t,da) ) ;
```

C.7.9 A Crossing Primitive using Only Wires

```
interface ReadOnly #( type a_type ) ;
   method a_type _read() ;
endinterface

module mkNullCrossing( Clock dClk, a_type dataIn,
                       ReadOnly#(a_type) ifc )
   provisos (Bits#(a_type, sizeOfa)) ;
```

C.7.10 Specialized Crossing Primitives

The `mkSyncRegToSlow` and `mkSyncSyncRegToFast` are specialized crossing primitives which can be used to transport data when clock edges are aligned, between the domains. The divided clocks and the appropriate interface needed for the module would typically be generated using the `mkClockDivider` module. The crossing primitive is implemented via a single register, clocked by slower (divided) clock. For a fast to slow crossing, the register is only writable when `clockReady` bit of the divider interface is asserted. This is an implicit condition of the module which prevent erroneous writes. For a slow to fast crossing both the read and write methods are always available.


```
module mkSyncRegToSlow #( a_type initValue
                        )( ClockDividerIfc divider,
                           Reset slowRstIn,
                           Reg #(a_type) ifc )
   provisos (Bits#(a_type,sizea)) ;
```

The `mkSyncFIFOAlignedEdges` module is a specialized crossing primitive which can be used to transport data when clock edges are aligned, between a fast sClkIn and a slower dClkIn. The derived clock and the `ClkNextRdy` signal would typically be generated using the `mkClockDivider` module. The crossing primitive is implemented via a FIFO with the specified depth clocked by dClkIn. The FIFO is only writable when `syncBit` is asserted and the FIFO is not Full.

```
module mkSyncFIFOToSlow #( Integer depth
                         )( ClockDividerIfc divider,
                            Reset slowRstIn,
                            SyncFIFOIfc #(a_type) ifc )
   provisos (Bits#(a_type,sizea)) ;

module mkSyncFIFOToFast #( Integer depth
                         )( ClockDividerIfc divider,
                            Reset slowRstIn,
                            SyncFIFOIfc #(a_type) ifc )
   provisos (Bits#(a_type,sizea)) ;
```

C.7.11 Reset Generation and Synchronization

Reset generation allows the conversion of a Boolean type to a Reset type, where the reset is associated with the default (or `clocked_by`) clock domain. Two modules provide this function, `mkReset` and `mkResetSync`, where each module has one parameter, `stages`. The `stages` argument is the number of full clock cycles the output reset is held after the input reset is deasserted. Specifying a 0 for the `stages` argument results in the creation of a simple wire between the `bRstIn` and `rstOut`. That is, the reset is asserted immediately and not held after the `bRstIn` is deasserted. It becomes the designer's responsibility to ensure that `bRstIn` is asserted for sufficient time to allow the design to reset properly. Note that the Boolean input `bRstIn` is asserted low, and can be taken from any clock domain.

The difference between `mkReset` and `mkResetSync` is that for the former, the assertion of reset is immediate, while the later asserts reset at the next rising edge of the clock. Note that the use mkResetSync is less common, since the reset requires clock edges to take effect; failure to assert reset for a clock edge will not result in a reset being seen at `rstOut`.

```
module mkReset #( Integer stages
                )( Bool bRstIn,
                   Reset rstOut ) ;

module mkResetSync #( Integer stages
                    )( Bool bRstIn,
                       Reset rstOut ) ;
```

To synchronize resets from one clock domain to another, several modules are provided. The `mkAsyncReset` family is similar to the `mkReset` module; the `stages` argument has the same behavior. Note that `sClkIn` is unused, but specified to be in the style of other synchronization modules.

```
module mkAsyncReset #( Integer stages
                     )( Clock sClkIn, Reset sRst,
                        Clock dClkIn,
                        Reset dRstOut ) ;
```



```
module mkAsyncResetFromCC #( Integer stages
                           )( Clock dClkIn,
                              Reset dRstOut ) ;
```

The less common `mkSyncReset` modules are provided for convenience, but these modules *require* that `sRst` be held during a positive edge of `dClkIn` for the reset assertion to be noticed.

```
module mkSyncReset #( Integer stages
                    )( Clock sClkIn, Reset sRst,
                       Clock dClkIn,
                       Reset dRstOut ) ;


module mkSyncResetFromCC #( Integer stages
                          )( Clock dClkIn,
                             Reset dRstOut ) ;
```

For testbenches, in which an absolute clock is being created, it is helpful to generate a reset for that clock. The module `mkInitialReset` generates a reset on the first clock that it receives. The reset is asserted for a number of cycles specified by the parameter, which must be greater than zero. This module is not consider synthesizable.

```
module mkInitialReset #( Integer cycles
                       )( Reset rstOut ) ;
```

C.8 RAMs

C.8.1 `RAM` and `TRAM`

The `RAM` type is used for various types of memories. The memory is a `Server` which accepts read or write requests. A read request will generate a response containing the read data. The latency for a `RAM` is arbitrary, it does not even have to be a fixed latency.

Note, the types of the address and data are arbitrary.

```
typedef Server#(RAMreq#(adr, dta), dta) RAM #(type adr, type dta);


typedef Client#(RAMreq#(adr, dta), dta) RAMclient #(type adr, type dta);


typedef union tagged {
    adr Read;
    Tuple2#(adr, dta) Write;
} RAMreq #(type adr, type dta) deriving (Eq, Bits);
```

The `TRAM` type represents a tagged RAM. It is similar to the `RAM` interface, but each read request has an additional tag that will be part of the response for a read.

```
typedef Server#(TRAMreq#(tag, adr, dta), TRAMresp#(tag, dta))
                         TRAM #(type tag, type adr, type dta);


typedef
  Client#(TRAMreq#(tag, adr, dta), TRAMresp#(tag, dta))
             TRAMclient #(type tag, type adr, type dta);
```



```
typedef tagged union {
    TRAMreqRead#(tag, adr, dta) Read;
    TRAMreqWrite#(tag, adr, dta) Write;
} TRAMreq #(type tag, type adr, type dta) deriving (Eq, Bits);

typedef struct {
    tg  tag;
    adr address;
} TRAMreqRead #(type tg, type adr, type dta) deriving (Eq, Bits);

typedef struct {
    dta value;
    adr address;
} TRAMreqWrite #(type tg, type adr, type dta) deriving (Eq, Bits);

typedef struct {
    tg  tag;
    dta value;
} TRAMresp #(type tg, type dta) deriving (Eq, Bits);
```

The `tagRAM` function converts a RAM to a TRAM by putting a tag FIFO next to it. The FIFO size is specified by the first argument.

```
module tagRAM#(Integer sz, Module#(RAM#(adr, dta)) mkRAM)(TRAM#(tg, adr, dta))
  provisos (Bits#(tg, stg));
```

C.8.2 SyncSRAM

The `SyncSRAM` package contains definitions of the low level type for connecting to synchronous SRAMs. It is not intended for programming with directly; it is only used to interface to internal and external SRAMs. The `SyncSRAMS` type is the type of an SRAM. An SRAM is a "server" in the sense that it accepts a request every clock cycle and delivers a response every clock cycle. The type has three parameters, `lat`, the latency in clock cycles, `adrs`, the size of the address, and `dtas`, the size of the data.

```
typedef
  Server#(SyncSRAMrequest#(lat, adrs, dtas), Bit#(dtas))
            SyncSRAMS #(type lat, type adrs, type dtas);
```

Correspondingly, `SyncSRAMC` is the type of a user (client) of an SRAM.

```
typedef
  Client#(SyncSRAMrequest#(lat, adrs, dtas), Bit#(dtas))
            SyncSRAMC #(type lat, type adrs, type dtas);
```

An SRAM request is simply the wires to the SRAM.

Note

SyncSRAMrequest should really be a struct, but we get nice wire names by using an interface.

```
interface (SyncSRAMrequest :: # -> # -> # -> *) #(type lat, type adrs, type dtas);
    method Bit#(adrs) addr();
    method Bit#(dtas) wdata();
    method Bit#(1) we();
    method Bit#(1) ena();
endinterface: (SyncSRAMrequest :: # -> # -> # -> *)
```

Note, it is important that the latency argument is accurate. Various SRAM adapters rely on the latency information in the type to do the right thing.

C.8.3 SRAM and TSRAM

The SRAM package contains functions for wrapping a raw SRAM so that it has the more convenient RAM interface. The mkWrapSRAM function takes a SyncSRAM module and turns it into a RAM module.

```
module mkWrapSRAM#(Module#(SyncSRAMS#(lat, adrs, dtas)) mkRam)(RAM#(adr, dta))
  provisos (Bits#(adr, adrs),
    Bits#(dta, dtas),
    Add#(1, lat, lat1),
    Add#(4, lat, lat4),
    Log#(lat4, llat));
```

The wrapSRAM module generates a SyncSRAMC client and a RAM server. The client interface can be exported and hooked up to an external SRAM, or hooked up to an internally generated SRAM.

```
module wrapSRAM(Tuple2 #(SyncSRAMC#(lat, adrs, dtas), RAM#(adr, dta)))
  provisos (Bits#(adr, adrs),
    Bits#(dta, dtas),
    Add#(1, lat, lat1),
    Add#(4, lat, lat4),
    Log#(lat4, llat));
```

Both the mkWrapSRAM and wrapSRAM modules add two cycles of latency to the SRAM latency. The reason for this is that the raw interface to the SRAM has fully "registered" inputs and outputs (which is necessary for many SRAMs).

Note

The current implementation of these functions is broken, it adds three extra cycles of latency.

The TSRAM package corresponds to the SRAM package, but for tagged SRAMs.

```
module mkWrapSTRAM#(Module#(SyncSRAMS#(lat, adrs, dtas)) mkRam)
                                                (TRAM#(tg, adr, dta))
  provisos (Bits#(adr, adrs),
    Bits#(dta, dtas),
    Bits#(tg, tgs),
    Add#(1, lat, lat1),
    Log#(lat1, llat));


module wrapSTRAM(Tuple2 #(SyncSRAMC#(lat, adrs, dtas), TRAM#(tg, adr, dta)))
  provisos (Bits#(adr, adrs),
    Bits#(dta, dtas),
    Bits#(tg, tgs),
    Add#(1, lat, lat1),
    Log#(lat1, llat));
```

C.8.4 SPSRAM

The SPSRAM package is used to generate internal single ported SRAMs (for the LSI libraries). The argument specifies the size of the SRAM. The SRAM has a one cycle latency.

```
module mkSPSRAM#(Integer nwords)(SyncSRAMS#(1, adrs, dtas));
```


C.8.5 DPSRAM

The DPSRAM package contains is used to generate internal dual ported SRAMs (for the LSI libraries). The argument specifies the size of the SRAM.

```
module mkDPSRAM#(Integer nwords)(Tuple2 #(SyncSRAMS#(1, adrs, dtas),
                                          SyncSRAMS#(1, adrs, dtas)));
```

C.8.6 SRAMFile

The SRAMFile package is used to generate single ported SRAMs, where the initial contents is taken from a file. The arguments specify the file name and the size of the SRAM. The SRAM has a one cycle latency.

```
mkSRAMFile :: (IsModule m c) => String -> Integer -> m (SyncSRAMS 1 adrs dtas)
```

C.9 Miscellaneous

C.9.1 Assert

The Assert package contains definitions to test assertions in the code.

Compile time assertion. Can be used anywhere a compile-time statement is valid.

```
function Module#(Void) staticAssert(Bool b, String s);
```

Run time assertion. Can be used anywhere an Action is valid, and is tested whenever it is executed.

```
function Action dynamicAssert(Bool b, String s);
```

Continuous run-time assertion (expected to be True on each clock). Can be used anywhere a module instantiation is valid.

```
function Action continuousAssert(Bool b);
```

C.9.2 Probe

Package

import Probe :: * ;

Description

A Probe is a primitive used to ensure that a signal of interest is not optimized away by the compiler and that it is given a known name. In terms of BSV syntax, the Probe primitive it used just like a register except that only a write method exists. Since reads are not possible, the use of a Probe has no effect on scheduling. In the generated Verilog, the associated signal will be named just like the port of any Verilog module, in this case <instance_name>$PROBE. No actual Probe instance will be created however. The only side effects of a BSV Probe instantiation relate to the naming and retention of the associated signal in the generated Verilog.

Interfaces

```
interface Probe #(type a_type);
    method Action _write(a_type x1);
endinterface: Probe
```


Modules

The module `mkProbe` is used to instantiate a `Probe`.

`mkProbe`	Instantiates a `Probe`
	`module mkProbe(Probe#(a_type))` `  provisos (Bits#(a_type, sizea));`

Example - Creating and writing to registers and probes

```
import FIFO::*;
import ClientServer::*;
import GetPut::*;
import Probe::*;

typedef Bit#(32) LuRequest;
typedef Bit#(32) LuResponse;

module mkMesaHwLpm(ILpm);
   // Create registers for requestB32 and responseB32
   Reg#(LuRequest) requestB32 <- mkRegU();
   Reg#(LuResponse) responseB32 <- mkRegU();

   // Create a probe responseB32_probe
   Probe#(LuResponse) responseB32_probe <- mkProbe();
   ....
   // Define the interfaces:
   ....
      interface Get  response;
         method get() ;
            actionvalue
               let resp <- completionBuffer.drain.get();
               // record response for debugging purposes:
               let {r,t} = resp;
               responseB32 <= r;        // a write to a register
               responseB32_probe <= r;  // a write to a probe

               // count responses in status register
               return(resp);
            endactionvalue
         endmethod: get
      endinterface: response
    .....
endmodule
```

C.9.3 Reserved

Package

import Reserved :: * ;

Description

`Reserved` defines an abstract data type which only has the purpose of taking up space. It is useful when defining a `struct` where you need to enforce a certain layout and want to use the type checker

to enfoce that the value is not accidently used. One can enforce a layout unsafely with `Bit#(n)`, but `Reserved#(n)` gives safety. A value of type `Reserved#(n)` takes up exactly `n` bits.

`typedef` ··· *abstract* ··· `Reserved#(type n);`

Type classes

Type Classes used by `Reserved`									
	`Bits`	`Eq`	`Literal`	`Arith`	`Ord`	`Bounded`	`Bit wise`	`Bit Reduction`	`Bit Extend`
`Reserved`	√	√			√	√			

- `Bits`

  Converting `Reserved` to or from bits yields an unspecified value ("_").

  The only purpose is to allow the value to exist in hardware (at port boundaries and in states). The user should have no reason to use `pack/unpack` directly.

- `Eq` and `Ord`

  Any two `Reserved` values are considered to be equal.

- `Bounded`

  The upper and lower bound return unspecified values ("_").

Example: Structure with a 8 bits reserved.

```
typedef struct {
   Bit#(8)               header;      // Frame.header
   Vector#(2, Bit#(8))   payload;     // Frame.payload
   Reserved#(8)          dummy;       // Can't access 8 bits reserved
   Bit#(8)               trailer;     // Frame.trailer
} Frame;
```

header	payload0	payload1	dummy	trailer
8	8	8	8	8

C.9.4 ZBus

BSV provides the `ZBus` library to allow users to implement and use tri-state buses. Since BSV does not support high-impedance or undefined values internally, the library encapsulates the tri-state bus implementation in a module that can only be accessed through predefined interfaces which do not allow direct access to internal signals (which could potentially have high-impedance or undefined values).

The Verilog implementation of the tri-state module includes a number of primitive sub-modules that are implemented using Verilog tri-state wires. The BSV representation of the bus, however, only models the values of the bus at the associated interfaces and thus the need to represent high-impedance or undefined values in BSV is avoided. The interfaces are defined as follows:

```
interface ZBusClientIFC #(type t) ;
   method Action      drive(t value);
   method t           get();
   method Bool        fromBusValid();
endinterface
```



```
interface ZBusBusIFC #(type t) ;
   method Action      fromBusSample(ZBit#(t) value, Bool isValid);
   method ZBit#(t)    toBusValue();
   method Bool        toBusCtl();
endinterface

interface ZBusDualIFC #(type t) ;
   method ZBusBusIFC#(t)    busIFC;
   method ZBusClientIFC#(t) clientIFC;
endinterface
```

The `ZBusClientIFC` allows a BSV module to connect to the tri-state bus. For this interface There are no tri-state values as either method arguments or return values. The `ZBusBuslIFC` interface connects to the bus structure itself (using tri-state values). The `ZBusDualIFC` interface includes one ZBusBusIFC and one ZBusClientIFC. For a given bus, one `ZBusDualIFC` interface is associated with each bus client. The library also provides a module constructor function, `mkZBusBuffer`, which allows the user to create a module which provides the `ZBusDualIFC` interface.

```
module mkZBusBuffer (ZBusDualIFC #(t))
   provisos (Eq#(t), Bits#(t,st));
```

This module provides essentially the functionality of a tri-state buffer. The following code fragment creates a tri-state buffer (with an interface named buffer_0) for a 32 bit signal.

```
   ZBusDualIFC#(Bit#(32)) buffer_0();
   mkZBusBuffer inst_buffer_0(buffer_0);
```

This code fragment drives a value of `12` onto the associated bus.

```
   buffer_0.clientIFC.drive(12);
```

The `get()` and `fromBusValid()` methods (associated with the `ZBusClientIFC` interface) allow each bus client to access the current value on the bus. If the bus is in an invalid state (i.e. has a high-impedance value or an undefined value because it is being driven by more than one client simultaneously), then the `get()` method will return `0` and the `fromBusValid()` method will return `False`. In all other cases, the `fromBusValid()` method will return `True` and the `get()` method will return the current value of the bus. Finally, the `ZBus` library provides the `mkZBus` module constructor function.

```
module mkZBus#(List#(ZBusBusIFC#(t)) ifc_list)(Empty)
   provisos (Eq#(t), Bits#(t, st));
```

This function takes a list of `ZBusBusIFC` interfaces as arguments and creates a module which ties them all together in a bus. The following code fragment demonstrates its use.

```
   ZBusDualIFC#(Bit#(32)) buffer_0();
   mkZBusBuffer inst_buffer_0(buffer_0);

   ZBusDualIFC#(Bit#(32)) buffer_1();
   mkZBusBuffer inst_buffer_1(buffer_1);

   ZBusDualIFC#(Bit#(32)) buffer_2();
   mkZBusBuffer inst_buffer_2(buffer_2);

   List#(ZBusIFC#(Bit#(32))) ifc_list;
```



```
bus_ifc_list = cons(buffer_0.busIFC,
                    cons(buffer_1.busIFC,
                         cons(buffer_2.busIFC,
                                  nil)));

Empty bus_ifc();
mkZBus#(bus_ifc_list) inst_bus(bus_ifc);
```

C.9.5 OVLAssertions

Package

import OVLAssertions :: * ;

Description

The OVLAssertions package provides the BSV interfaces and wrapper modules necessary to allow BSV designs to include assertion checkers from the Open Verification Library (OVL). The OVL includes a set of assertion checkers that verify specific properties of a design. For more details on the complete OVL, refer to the Accellera Standard OVL Library Reference Manual.

Interfaces and Methods

The following interfaces are defined for use with the assertion modules. Each interface has one or more `Action` methods. Each method takes a single argument which is either a `Bool` or polymorphic.

AssertTest_IFC

Used for assertions that check a test expression on every clock cycle.

AssertTest_IFC				
Method		Argument		
Name	Type	Name	Type	Description
`test`	`Action`	`test_value`	`a_type`	Expression to be checked.

```
interface AssertTest_IFC #(type a_type);
   method Action test(a_type test_value);
endinterface
```

AssertSampleTest_IFC

Used for assertions that check a test expression on every clock cycle only if the sample, indicated by the boolean value `sample_test` is asserted.

AssertSampleTest_IFC				
Method		Argument		
Name	Type	Name	Type	Description
`sample`	`Action`	`sample_test`	`Bool`	Assertion only checked if `sample_test` is asserted.
`test`	`Action`	`test_value`	`a_type`	Expression to be checked.

```
interface AssertSampleTest_IFC #(type a_type);
   method Action sample(Bool sample_test);
   method Action test(a_type test_value);
endinterface
```


AssertStartTest_IFC

Used for assertions that check a test expression only subsequent to a start_event, specified by the Boolean value `start_test`.

AssertStartTest_IFC				
Method		Argument		
Name	Type	Name	Type	Description
`start`	`Action`	`start_test`	`Bool`	Assertion only checked after start is asserted.
`test`	`Action`	`test_value`	`a_type`	Expression to be checked.

```
interface AssertStartTest_IFC #(type a_type);
   method Action start(Bool start_test);
   method Action test(a_type test_value);
endinterface
```

AssertStartStopTest_IFC

Used to check a test expression between a start_event and a stop_event.

AssertStartStopTest_IFC				
Method		Argument		
Name	Type	Name	Type	Description
`start`	`Action`	`start_test`	`Bool`	Assertion only checked after start is asserted.
`stop`	`Action`	`stop_test`	`Bool`	Assertion only checked until the stop is asserted.
`test`	`Action`	`test_value`	`a_type`	Expression to be checked.

```
interface AssertStartStopTest_IFC #(type a_type);
   method Action start(Bool start_test);
   method Action stop(Bool stop_test);
   method Action test(a_type test_value);
endinterface
```

AssertTransitionTest_IFC

Used to check a test expression that has a specified start state and next state, i.e. a transition.

AssertTransitionTest_IFC				
Method		Argument		
Name	Type	Name	Type	Description
`test`	`Action`	`test_value`	`a_type`	Expression that should transition to the `next_value`.
`start`	`Action`	`start_test`	`a_type`	Expression that indicates the start state for the assertion check. If the value of `start_test` equals the value of `test_value`, the check is performed.
`next`	`Action`	`next_value`	`a_type`	Expression that indicates the only valid next state for the assertion check.

```
interface AssertTransitionTest_IFC #(type a_type);
   method Action test(a_type test_value);
   method Action start(a_type start_value);
   method Action next(a_type next_value);
endinterface
```


AssertQuiescentTest_IFC

Used to check that a test expression is equivalent to the specified expression when the sample state is asserted.

AssertQuiescentTest_IFC				
Method		Argument		
Name	Type	Name	Type	Description
sample	Action	sampe_test	Bool	Expression which initiates the quiescent assertion check when it transistions to true.
state	Action	state_value	a_type	Expression that should have the same value as check_value
check	Action	check_value	a_type	Expression state_value is compared to.

```
interface AssertQuiescentTest_IFC #(type a_type);
   method Action sample(Bool sample_test);
   method Action state(a_type state_value);
   method Action check(a_type check_value);
endinterface
```

AssertFifoTest_IFC

Used with assertions checking a FIFO structure.

AssertFifoTest_IFC				
Method		Argument		
Name	Type	Name	Type	Description
push	Action	push_value	a_type	Expression which indicates the number of push operations that will occur during the current cycle.
pop	Action	pop_value	a_type	Expression which indicates the number of pop operations that will occur during the current cycle.

```
interface AssertFifoTest_IFC #(type a_type, type b_type);
   method Action push(a_type push_value);
   method Action pop(b_type pop_value);
endinterface
```

Datatypes

Each assertion checker has a defined set of parameters. The parameters `severity_level`, `property_type`, `msg`, and `coverage_level` are common to all assertion checkers.

Common Parameters for all Assertion Checkers	
Parameter	Valid Values * indicates default value
severity_level	OVL_FATAL *OVL_ERROR OVL_WARNING OVL_Info
property_type	*OVL_ASSERT OVL_ASSUME OVL_IGNORE
msg	*VIOLATION
coverage_level	OVL_COVER_NONE *OVL_COVER_ALL OVL_COVER_SANITY OVL_COVER_BASIC OVL_COVER_CORNER OVL_COVER_STATISTIC

Each assertion checker may also use some subset of the following parameters.

Other Parameters for Assertion Checkers	
Parameter	Valid Values
action_on_new_start	OVL_IGNORE_NEW_START OVL_RESET_ON_NEW_START OVL_ERROR_ON_NEW_START
edge_type	OVL_NOEDGE OVL_POSEDGE OVL_NEGEDGE OVL_ANYEDGE
necessary_condition	OVL_TRIGGER_ON_MOST_PIPE OVL_TRIGGER_ON_FIRST_PIPE OVL_TRIGGER_ON_FIRST_NOPIPE
inactive	OVL_ALL_ZEROS OVL_ALL_ONES OVL_ONE_COLD

Other Parameters for Assertion Checkers	
Parameter	Valid Values
num_cks	Int#(32)
min_cks	Int#(32)
max_cks	Int#(32)
min_ack_cycle	Int#(32)
max_ack_cycle	Int#(32)
max_ack_length	Int#(32)
req_drop	Int#(32)
deassert_count	Int#(32)
depth	Int#(32)
value	a_type
min	a_type
max	a_type
check_overlapping	Bool
check_missing_start	Bool
simultaneous_push_pop	Bool

Modules

Each module in this package corresponds to an assertion checker from the Open Verification Library (OVL). The BSV name for each module is the same as the OVL name with `bsv_` appended to the beginning of the name.

Module	`bsv_assert_always`
Description	Concurrent assertion that the value of the expression is always `True`.
Interface Used	`AssertTest_IFC`
Parameters	common assertion parameters
Module Declaration	`module bsv_assert_always#(OVLDefaults#(Bool) defaults)` `(AssertTest_IFC#(Bool));`

Module	`bsv_assert_always_on_edge`
Description	Checks that the test expression evaluates `True` whenever the sample method is asserted.
Interface Used	`AssertSampleTest_IFC`
Parameters	common assertion parameters `edge_type` (default value = `OVL_NOEDGE`)
Module Declaration	`module bsv_assert_always_on_edge#(OVLDefaults#(Bool)` `defaults)(AssertSampleTest_IFC#(Bool));`

Module	`bsv_assert_change`
Description	Checks that once the start method is asserted, the expression will change value within `num_cks` cycles.
Interface Used	`AssertStartTest_IFC`
Parameters	common assertion parameters `action_on_new_start` (default value = `OVL_IGNORE_NEW_START`) `num_cks` (default value = 1)

Module Declaration	`module bsv_assert_change#(OVLDefaults#(a_type) defaults)`
	`(AssertStartTest_IFC#(a_type))`
	`provisos (Bits#(a_type, sizea),`
	`Bounded#(a_type), Eq#(a_type));`

Module	`bsv_assert_cycle_sequence`
Description	Ensures that if a specified necessary condition occurs,it is followed by a specified sequence of events.
Interface Used	`AssertTest_IFC`
Parameters	common assertion parameters `necessary_condition` (default value = `OVL_TRIGGER_ON_MOST_PIPE`)

Module Declaration	`module bsv_assert_cycle_sequence#(OVLDefaults#(a_type)`
	`defaults)(AssertTest_IFC#(a_type))`
	`provisos (Bits#(a_type, sizea),`
	`Bounded#(a_type), Eq#(a_type));`

Module	`bsv_assert_decrement`
Description	Ensures that the expression decrements only by the value specifiedR.
Interface Used	`AssertTest_IFC`
Parameters	common assertion parameters `value` (default value = 1)

Module Declaration	`module bsv_assert_decrement#(OVLDefaults#(a_type) defaults)`
	`(AssertTest_IFC#(a_type))`
	`provisos (Bits#(a_type, sizea), Literal#(a_type),`
	`Bounded#(a_type), Eq#(a_type));`

Module	`bsv_assert_delta`
Description	Ensures that the expression always changes by a value within the range specified by `min` and `max`.
Interface Used	`AssertTest_IFC`
Parameters	common assertion parameters `min` (default value = 1) `max` (default value = 1)

Module Declaration	`module bsv_assert_delta#(OVLDefaults#(a_type) defaults)`
	`(AssertTest_IFC#(a_type))`
	`provisos (Bits#(a_type, sizea), Literal#(a_type),`
	`Bounded#(a_type), Eq#(a_type));`

Module	`bsv_assert_even_parity`
Description	Ensures that value of a specified expression has even parity, that is an even number of bits in the expression are active high.
Interface Used	`AssertTest_IFC`
Parameters	common assertion parameters

Module Declaration	`module bsv_assert_even_parity#(OVLDefaults#(a_type)`
	`defaults) (AssertTest_IFC#(a_type))`
	`provisos (Bits#(a_type, sizea),`
	`Bounded#(a_type), Eq#(a_type));`

Module	`bsv_assert_fifo_index`
Description	Ensures that a FIFO-type structure never overflows or underflows. This checker can be configured to support multiple pushes (FIFO writes) and pops (FIFO reads) during the same clock cycle.
Interface Used	`AssertFifoTest_IFC`
Parameters	common assertion parameters `depth` (default value = 1) `simultaneous_push_pop` (default value = `True`)

Module Declaration	`module bsv_assert_fifo_index#(OVLDefaults#(Bit#(0))`
	`defaults)(AssertFifoTest_IFC#(a_type, b_type))`
	`provisos (Bits#(a_type, sizea), Bits#(b_type, sizeb));`

Module	`bsv_assert_frame`
Description	Checks that once the start method is asserted, the test expression evaluates true not before `min_cks` clock cycles and not after `max_cks` clock cycles.
Interface Used	`AssertStartTest_IFC`
Parameters	common assertion parameters `action_on_new_start` (default value = `OVL_IGNORE_NEW_START`) `min_cks` (default value = 1) `max_cks` (default value = 1)
Module Declaration	`module bsv_assert_frame#(OVLDefaults#(Bool) defaults)` `(AssertStartTest_IFC#(Bool));`

Module	`bsv_assert_handshake`
Description	Ensures that the specified request and acknowledge signals follow a specified handshake protocol.
Interface Used	`AssertStartTest_IFC`
Parameters	common assertion parameters `action_on_new_start` (default value = `OVL_IGNORE_NEW_START`) `min_ack_cycle` (default value = 1) `max_ack_cycle` (default value = 1)
Module Declaration	`module bsv_assert_handshake#(OVLDefaults#(Bool) defaults)` `(AssertStartTest_IFC#(Bool));`

Module	`bsv_assert_implication`
Description	Ensures that a specified consequent expression is `True` if the specified antecedent expression is `True`.
Interface Used	`AssertStartTest_IFC`
Parameters	common assertion parameters
Module Declaration	`module bsv_assert_implication#(OVLDefaults#(Bool) defaults)` `(AssertStartTest_IFC#(Bool));`

Module	`bsv_assert_increment`
Description	ensure that the test expression always increases by the value of specified by `value`.
Interface Used	`AssertTest_IFC`
Parameters	common assertion parameters `value` (default value = 1)

Module Declaration	`module bsv_assert_increment#(OVLDefaults#(a_type) defaults)`
	`(AssertTest_IFC#(a_type))`
	`provisos (Bits#(a_type, sizea), Literal#(a_type),`
	`Bounded#(a_type), Eq#(a_type));`

Module	`bsv_assert_never`
Description	Ensures that the value of a specified expression is never `True`.
Interface Used	`AssertTest_IFC`
Parameters	common assertion parameters
Module Declaration	`module bsv_assert_never#(OVLDefaults#(Bool) defaults)` `(AssertTest_IFC#(Bool));`

Module	`bsv_assert_never_unknown`
Description	Ensures that the value of a specified expression contains only 0 and 1 bits when a qualifying expression is `True`.
Interface Used	`AssertStartTest_IFC`
Parameters	common assertion parameters

Module Declaration	`module bsv_assert_never_unknown#(OVLDefaults#(a_type)`
	`defaults)(AssertStartTest_IFC#(a_type))`
	`provisos (Bits#(a_type, sizea),`
	`Bounded#(a_type), Eq#(a_type));`

Module	`bsv_assert_never_unknown_async`
Description	Ensures that the value of a specified expression always contains only 0 and 1 bits
Interface Used	`AssertTest_IFC`
Parameters	common assertion parameters

Module Declaration	`module bsv_assert_never_unknown_async#(OVLDefaults#(a_type)`
	`defaults)(AssertTest_IFC#(a_type))`
	`provisos (Bits#(a_type, sizea), Literal#(a_type),`
	`Bounded#(a_type), Eq#(a_type));`

Module	`bsv_assert_next`
Description	Ensures that the value of the specified expression is true a specified number of cycles after a start event.
Interface Used	`AssertStartTest_IFC`
Parameters	common assertion parameters `num_cks` (default value = 1) `check_overlapping` (default value = `True`) `check_missing_start` (default value = `False`)
Module Declaration	`module bsv_assert_next#(OVLDefaults#(Bool) defaults)` `(AssertStartTest_IFC#(Bool));`

Module	`bsv_assert_no_overflow`
Description	Ensures that the value of the specified expression does not overflow.
Interface Used	`AssertTest_IFC`
Parameters	common assertion parameters `min` (default value = `minBound`) `max` (default value = `maxBound`)
Module Declaration	`module bsv_assert_no_overflow#(OVLDefaults#(a_type) defaults) (AssertTest_IFC#(a_type)) provisos (Bits#(a_type, sizea), Bounded#(a_type), Eq#(a_type));`

Module	`bsv_assert_no_transition`
Description	Ensures that the value of a specified expression does not transition from a start state to the specified next state.
Interface Used	`AssertTransitionTest_IFC`
Parameters	common assertion parameters
Module Declaration	`module bsv_assert_no_transition#(OVLDefaults#(a_type) defaults) (AssertTransitionTest_IFC#(a_type)) provisos (Bits#(a_type, sizea), Bounded#(a_type), Eq#(a_type));`

Module	`bsv_assert_no_underflow`
Description	Ensures that the value of the specified expression does not underflow.
Interface Used	`AssertTest_IFC`
Parameters	common assertion parameters `min` (default value = `minBound`) `max` (default value = `maxBound`)
Module Declaration	`module bsv_assert_no_underflow#(OVLDefaults#(a_type) defaults)(AssertTest_IFC#(a_type)) provisos (Bits#(a_type, sizea), Bounded#(a_type), Eq#(a_type));`

Module	`bsv_assert_odd_parity`
Description	Ensures that the specified expression had odd parity; that an odd number of bits in the expression are active high.
Interface Used	`AssertTest_IFC`
Parameters	common assertion parameters
Module Declaration	`module bsv_assert_odd_parity#(OVLDefaults#(a_type) defaults)(AssertTest_IFC#(a_type)) provisos (Bits#(a_type, sizea), Bounded#(a_type), Eq#(a_type));`

Module	`bsv_assert_one_cold`
Description	Ensures that exactly one bit of a variable is active low.
Interface Used	`AssertTest_IFC`
Parameters	common assertion parameters `inactive` (default value = `OLV_ONE_COLD`)

Module Declaration	`module bsv_assert_one_cold#(OVLDefaults#(a_type) defaults)`
	`(AssertTest_IFC#(a_type))`
	`provisos (Bits#(a_type, sizea),`
	`Bounded#(a_type), Eq#(a_type))`

Module	`bsv_assert_one_hot`
Description	Ensures that exactly one bit of a variable is active high.
Interface Used	`AssertTest_IFC`
Parameters	common assertion parameters

Module Declaration	`module bsv_assert_one_hot#(OVLDefaults#(a_type) defaults)`
	`(AssertTest_IFC#(a_type))`
	`provisos (Bits#(a_type, sizea),`
	`Bounded#(a_type), Eq#(a_type));`

Module	`bsv_assert_proposition`
Description	Ensures that the test expression is always combinationally `True`. Like `assert_always` except that the test expression is not sampled by the clock.
Interface Used	`AssertTest_IFC`
Parameters	common assertion parameters
Module Declaration	`module bsv_assert_proposition#(OVLDefaults#(Bool) defaults)` `(AssertTest_IFC#(Bool));`

Module	`bsv_assert_quiescent_state`
Description	Ensures that the value of a specified state expression equals a corresponding check value if a specified sample event has transitioned to TRUE.
Interface Used	`AssertQuiescentTest_IFC`
Parameters	common assertion parameters

Module Declaration	`module bsv_assert_quiescent_state#(OVLDefaults#(a_type)`
	`defaults)(AssertQuiescentTest_IFC#(a_type))`
	`provisos (Bits#(a_type, sizea),`
	`Bounded#(a_type), Eq#(a_type));`

Module	bsv_assert_range
Description	Ensure that an expression is always within a specified range.
Interface Used	AssertTest_IFC
Parameters	common assertion parameters min (default value = minBound) max (default value = maxBound)
Module Declaration	module bsv_assert_range#(OVLDefaults#(a_type) defaults) (AssertTest_IFC#(a_type)) provisos (Bits#(a_type, sizea), Bounded#(a_type), Eq#(a_type));

Module	bsv_assert_time
Description	Ensures that the expression remains True for a specified number of clock cycles after a start event.
Interface Used	AssertStartTest_IFC
Parameters	common assertion parameters action_on_new_start (default value = OVL_IGNORE_NEW_START) num_cks (default value = 1)
Module Declaration	module bsv_assert_time#(OVLDefaults#(Bool) defaults) (AssertStartTest_IFC#(Bool));

Module	bsv_assert_transition
Description	Ensures that the value of a specified expression transitions properly froma start state to the specified next state.
Interface Used	AssertTransitionTest_IFC
Parameters	common assertion parameters
Module Declaration	module bsv_assert_transition#(OVLDefaults#(a_type) defaults)(AssertTransitionTest_IFC#(a_type)) provisos (Bits#(a_type, sizea), Bounded#(a_type), Eq#(a_type));

Module	bsv_assert_unchange
Description	Ensures that the value of the specified expression does not change during a specified number of clock cycles after a start event initiates checking.
Interface Used	AssertStartTest_IFC
Parameters	common assertion parameters action_on_new_start (default value = OVL_IGNORE_NEW_START) num_cks (default value = 1)
Module Declaration	module bsv_assert_unchange#(OVLDefaults#(a_type) defaults) (AssertStartTest_IFC#(a_type)) provisos (Bits#(a_type, sizea), Bounded#(a_type), Eq#(a_type));

Module	`bsv_assert_width`
Description	Ensures that when the test expression goes high it stays high for at least `min` and at most `max` clock cycles.
Interface Used	`AssertTest_IFC`
Parameters	common assertion parameters `min_cks` (default value = 1) `max_cks` (default value = 1)
Module Declaration	`module bsv_assert_width#(OVLDefaults#(Bool) defaults)` `(AssertTest_IFC#(Bool));`

Module	`bsv_assert_win_change`
Description	Ensures that the value of a specified expression changes in a specified window between a start event and a stop event.
Interface Used	`AssertStartStopTest_IFC`
Parameters	common assertion parameters

Module Declaration	`module bsv_assert_win_change#(OVLDefaults#(a_type)`
	`defaults)(AssertStartStopTest_IFC#(a_type))`
	`provisos (Bits#(a_type, sizea),`
	`Bounded#(a_type), Eq#(a_type));`

Module	`bsv_assert_win_unchange`
Description	Ensures that the value of a specified expression does not change in a specified window between a start event and a stop event.
Interface Used	`AssertStartStopTest_IFC`
Parameters	common assertion parameters

Module Declaration	`module bsv_assert_win_unchange#(OVLDefaults#(a_type)`
	`defaults)(AssertStartStopTest_IFC#(a_type))`
	`provisos (Bits#(a_type, sizea),`
	`Bounded#(a_type), Eq#(a_type));`

Module	`bsv_assert_window`
Description	Ensures that the value of a specified event is `True` between a specified window between a start event and a stop event.
Interface Used	`AssertStartStopTest_IFC`
Parameters	common assertion parameters
Module Declaration	`module bsv_assert_window#(OVLDefaults#(Bool) defaults)` `(AssertStartStopTest_IFC#(Bool));`

Module	bsv_assert_zero_one_hot
Description	ensure that exactly one bit of a variable is active high or zero.
Interface Used	AssertTest_IFC
Parameters	common assertion parameters
Module Declaration	module bsv_assert_zero_one_hot#(OVLDefaults#(a_type) defaults)(AssertTest_IFC#(a_type)) provisos (Bits#(a_type, sizea), Bounded#(a_type), Eq#(a_type));

Example using bsv_assert_increment

This example checks that a test expression is always incremented by a value of 3. The assertion passes for the first 10 increments and then starts failing when the increment amount is changed from 3 to 1.

```
import OVLAssertions::*;     // import the OVL Assertions package

module assertIncrement (Empty);

   Reg#(Bit#(8)) count <- mkReg(0);
   Reg#(Bit#(8)) test_expr <- mkReg(0);

   let defaults = mkOVLDefaults;
   // override the default increment value and set = 3
   defaults.value = 3;

   // instantiate an instance of the module bsv_assert_increment using
   // the name assert_mod and the interface AssertTest_IFC
   AssertTest_IFC#(Bit#(8)) assert_mod <- bsv_assert_increment(defaults);

   rule every (True);              // Every clock cycle
      assert_mod.test(test_expr);  // the assertion is checked
   endrule

   rule increment (True);
      count <= count + 1;
      if (count < 10)                 // for 10 cycles
          test_expr <= test_expr + 3; // increment the expected amount
      else if (count < 15)
          test_expr <= test_expr + 1; // then start incrementing by 1
      else
          $finish;
   endrule
endmodule
```


Index

