Bluespec™ SystemVerilog Reference Guide

Revision: 24 November 2008

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).

If the first character of an instance name is an underscore, (`_`), the compiler will not generate this instance in the Verilog hierarchy name. This can be useful for removing submodules from the hierarchical naming.

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`

With the exception of plain decimal literals (that have neither a bit width or a base), 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). An optional `-` is part of the syntax for plain decimal literals so that it is possible to construct negative constants whose negation is not in the range of the type being constructed (e.g. `Int#(4) x = -8;` since `8` is not a valid `Int#(4)`, but `-8` is).

Examples:

```
125
-16
'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 `"···"` and must be contained on a single source line.

stringLiteral ::= `"` ··· string characters ··· `"`

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* [`(..)`]
	|	*packageIde* `::` `*`
importDecl	::=	`import` *importItem* { `,` *importItem* } `;`
importItem	::=	*packageIde* `::` `*`
packageStmt	::=	[*attributeInstances*] *moduleDef*
	|	*interfaceDecl*
	|	*typeDef*
	|	*varDecl* | *varAssign*
	|	[*attributeInstances*] *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.

An export item can specify an identifier defined elsewhere within this package optionally followed by `(..)`. The identifier then becomes accessible outside this package. An export item can also specify an identifier from an imported package. In that case, the imported identifier is re-exported from this package, so that it is accessible by importing this package (without requiring the import of its source package). It is also possible to re-export all of the identifiers from an imported package by using the following syntax: `export packageIde::*`.

If there are any export statements in a package, then only those items are exported. If there are no export statements, by default all identifiers defined in this package (and no identifiers from any imported packages) 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. Also overloaded
                //     function inLiteralRange to determine if an Integer
                //     is in the range of the target 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 extend, 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 ceiling 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 (or 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. The pseudo-function is also sometimes written as `valueOf`; both are considered correct.

exprPrimary	::=	`valueof (` *type* `)`
	|	`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));
   Integer maxindex = valueof(n) - 1;
   Int#(k) index;
   index = fromInteger(maxindex);
   ...
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;
```

[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*."


```
    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 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	::=	[*attributeInstances*] `interface` *typeDefType* `;` { *interfaceMemberDecl* } `endinterface` [`:` *typeIde*]
typeDefType	::=	*typeIde* [*typeFormals*]
typeFormals	::=	`#` `(` *typeFormal* { `,` *typeFormal* }`)`
typeFormal	::=	[`numeric`] `type` *typeIde*
interfaceMemberDecl	::=	*methodProto* | *subinterfaceDecl*
methodProto	::=	[*attributeInstances*] `method` *type identifier* `(` [*methodProtoFormals*] `)` `;`
methodProtoFormals	::=	*methodProtoFormal* { `,` *methodProtoFormal* }
methodProtoFormal	::=	[*attributeInstances*] *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

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

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:

```
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 parameter is non-numeric. The user can override this default by specifying `numeric` in the interface declaration.

The Standard Prelude defines a standard interface called `Empty` which contains no methods, i.e., its definition is:

```
interface Empty;
endinterface
```

This is often used for top-level modules that integrate a testbench and a design-under-test, and for modules like `mkConnection`(C.6.2) that just take interface arguments and do not themselves offer any interesting interface.

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 ::= [*attributeInstances*]
  `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(...);
```

Since `Clock` and `Reset` are both interface types, they can be used in interface declarations. Example:

```
interface ClockTickIfc ;
   method Action tick() ;
   interface Clock  new_clk ;
endinterface
```

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	::=	*moduleProto* { *moduleStmt* } `endmodule` [`:` *identifier*]
moduleProto	::=	`module` [`[` *type* `]`] *identifier* [*moduleFormalParams*] `(` [*moduleFormalArgs*] `)` [*provisos*]`;`
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 keyword `parameter` specifies a Verilog parameter is to be generated; without the keyword a Verilog port is generated. A Verilog parameter requires that the value is a constant at elaboration. When the module is instantiated, the actual expression provided for the parameter must be something that can be computed using normal Verilog elaboration rules. The bluespec compiler will check for this. The `parameter` keyword is only relevant when the module is marked with the `*synthesize*` attribute.

Inside the module, the `parameter` keyword can be used for a parameter `n` that is used, for example, for constants in expressions, register initialization values, and so on. However, `n` cannot be used for structural variations in the module, such as declaring an array of `n` registers. Such structural decisions (*generate* decisions) are taken by the Bluespec compiler, and cannot currently be postponed into the Verilog.

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*
	|	<*module*>*For*
	|	<*module*>*While*
	|	*varDecl* | *varAssign*
	|	*varDo* | *varDeclDo*
	|	*functionDef*
	|	*functionStmt*
	|	*systemTaskStmt*
	|	(*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 may 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*
`(` [*moduleActualParamArg* { `,` *moduleActualParamArg* }] `)`

moduleActualParamArg ::= *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 list of parameters and arguments, if the module is defined to have parameters and arguments. Note that the parameters and the arguments are within a single set of parentheses, the parameters listed first, and there is no `#` before the 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 only 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 grab 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` [*type*] *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	::=	[*attributeInstances*] `rule` *identifier* [*ruleCond*] `;` *ruleBody* `endrule` [`:` *identifier*]
ruleCond	::=	`(` *condPredicate* `)`
condPredicate	::=	*exprOrCondPattern* { `&&&` *exprOrCondPattern* }
exprOrCondPattern	::=	*expression*
	|	*expression* `matches` *pattern*
ruleBody	::=	{ *actionStmt* }

A rule is optionally preceded by an *attributeInstances*; 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.

In order to be synthesizable, a module must meet the following characteristics:

- The module must be of type `Module` and not of any other module type that can be defined with `ModuleCollect`;
- Its interface must be fully specified; there can be no polymorphic types in the interface;
- Its interface is a type whose methods and subinterfaces are all convertible to wires (see Section 5.8.2).
- All other inputs to the module must be convertible to Bits (see Section 5.8.2).

[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.

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.

```
(* 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).

Separate synthesis of a module can affect scheduling. This is because input wires to the module, such as method arguments, now become a fixed resource that must be shared, whereas without separate synthesis, module inlining allows them to be bypassed (effectively replicated). Consider a module representing a register file containing 32 registers, with a method `read(j)` that reads the value of the j'th register. Inside the module, this just indexes an array of registers. When separately synthesized, the argument `j` becomes a 5-bit wide input port, which can only be driven with one value in any given clock. Thus, two rules that invoke `read(3)` and `read(11)`, for example, will conflict and then they cannot fire in the same clock. If, however, the module is not separately synthesized, the module and the `read()` method are inlined, and then each rule can directly read its target register, so the rules can fire together in the same clock. Thus, in general, the addition of a synthesis boundary can restrict behaviors.

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)
   "Cannot synthesize ‘mkGCD’: Its interface is 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
```

5.8.2 Module Interfaces and Arguments

As mentioned above, a module is synthesizable if its interface is convertible to wires.

- An interface is convertible to wires if all methods and subinterfaces are convertible to wires.
- A method is convertible to wires if
  - all arguments are convertible to bits;
  - it is an `Action` method or it is an `ActionValue` or value method where the return value is convertible to bits.
- `Clock`, `Reset`, and `Inout` subinterfaces are convertible to wires.
- A Vector interface can be synthesized as long as the type inside the Vector is of type `Clock`, `Reset`, `Inout` or a type which is convertible to bits.

To be convertible to bits, a type must be in the `Bits` typeclass.

For a module to be synthesizable its arguments must be of type `Clock`, `Reset`, `Inout`, or a type convertible to bits. Vectors of the preceeding types are also synthesizable. If a module has one or more arguments which are not one of the above types, the module is not synthesizable. For example, if an argument is a datatype, such as `Integer`, which is not in the `Bits` typeclass, then the module cannot be separately synthesized.

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.

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:

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

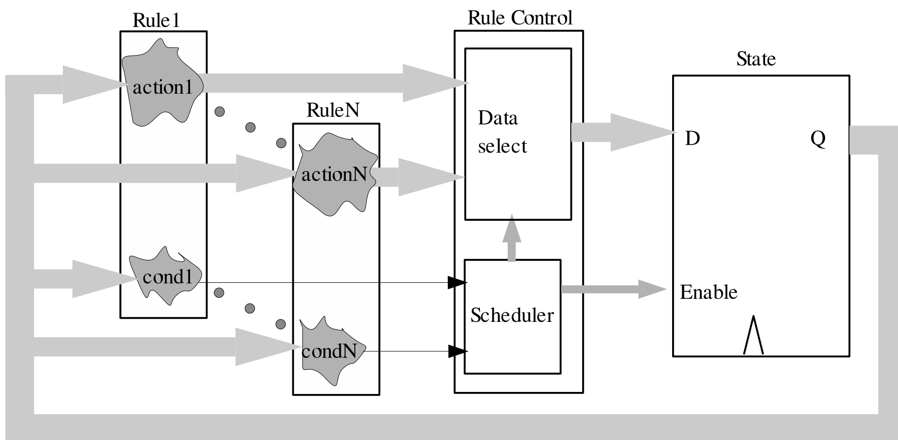

Figure 1: A general scheme for mapping an N-rule system into clocked synchronous hardware.

- 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).

6.2.4 Mapping specific hardware models

Annotations on the methods of a module are used by the BSV compiler to model the hardware behavior into TRS semantics. For example, all reads from a register must be scheduled before any writes to the same resgister. That is to say, any rule which reads from a register must be scheduled *earlier* than any other rule which writes to it. More generally, there exist scheduling constraints for specific hardware modules which describe how methods interact within the schedule. The scheduling annotations describe the constraints enforced by the BSV compiler.

The meanings of the scheduling annotations are:

`C`	conflicts
`CF`	conflict-free

[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.

`SB`	sequence before
`SBR`	sequence before restricted (cannot be in the same rule)
`SA`	sequence after
`SAR`	sequence after restricted (cannot be in the same rule)

Below is an example of the scheduling annotations for a register:

Scheduling Annotations Register		
	read	write
read	CF	SB
write	SA	SBR

The table describes the following scheduling constraints:

- Two `read` methods would be conflict-free (`CF`), that is, you could have multiple methods that read from the same register in the same rule, sequenced in any order.
- A `write` is sequenced after (`SA`) a `read`.
- A `read` is sequenced before (`SB`) a write.
- And finally, if you have two `write` methods, one must be sequenced before the other, and they cannot be in the same rule, as indicated by the annotation `SBR`.

The scheduling annotations are specific to the TRS model desired and a single hardware component can have multiple TRS models. For example, a register may be implemented using a `mkReg` module or a `mkConfigReg` module, which are identical except for their scheduling annotations.

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. Enum labels may be repeated in different enum definitions. 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 form `deriving (Ord)` is not currently supported for enums.

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* `;`
| *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 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` (section C.1.1) and `Vector` (section C.2) data types. Array 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 should have a defined bit width. The size of the updated range (determined by the two literal indices or by the size of the right-hand side) must be less than or equal to the size of the target bit vector.

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	::=	*lValue* `<=` *expression*
	|	`(` *expression* `)` `<=` *expression*

The left-hand side must contain a writeable interface type, such as `Reg#(`*t*`)` (for some type *t* that has a representation in bits). It is either an *lValue* or a parenthesized expression (e.g., the register interface could be selected from an array of register interfaces or returned from a function). The right-hand side must have the same type as the left-hand side would have if it were typechecked as an expression (including read desugaring, as described below). 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 simple 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 different method call:

> *r*._read ()

The implicit addition of the *._read* method call to variables of type `Reg#(`*t*`)` is the simplest example of *read desugaring*.

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
```

8.4.1 Registers and square-bracket notation

Register writes can be combined with the square-bracket notation.

regWrite	::=	*lValue arrayIndexes* `<=` *expression*
arrayIndexes	::=	`[` *expression* `]` { `[` *expression* `]` }

There are two different ways to interpret this combination. First, it can mean to select a register out of a collection of registers and write it.

Example. Updating a register in an array of registers:

```
List#(Reg#(int)) regs;
...
regs[3] <= regs[3] + 1;    // increment the register at position 3
```

Note that when the square-bracket notation is used on the right-hand side, read desugaring is also applied[6]. This allows the expression `regs[3]` to be interpreted as a register read without unnecessary clutter.

The indexed register assignment notation can also 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). This interpretation is just a shorthand for a whole register update where only the selected element is updated. In other words,

[6]To suppress read desugaring use `asReg` or `asIfc`


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

can be 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 tool infers the appropriate meaning for an indexed register write based on the types available and the context:

```
Reg#(Bit#(32)) x;
x[3] <= e;
List#(Reg#(a)) x;
y[3] <= e;
```

In the former case, `x` is a register containing an array of items (in this example a bit vector), so the statement updates the third item in this array (a single bit) and stores the updated bit vector in the register. In the latter case, `y` is an array of registers, so register at position 3 in the array is updated. In the former case, multiple writes to different indices in a single rule with non-exclusive conditions are forbidden (because they would be multiple conflicting writes to the same register)[7], writing the final result back to the register. In the latter case, multiple writes to different indices will be allowed, because they are writes to different registers (though multiple writes to the same index, under non-exclusive conditions would not be allowed, of course).

It also is possible to mix these notations, i.e., writing a single statement to perform a partial update of a register in an array of registers.

Example: Mixing types of square-bracket notation in a register write

```
List#(Reg#(bit[3:0])) ys;
...
y[4][3] <= e;        // Update bit 3 of the register at position 4
```

8.4.2 Registers and range notation

Just as there is a range notation for bit extraction and variable assignments, there is also a range notation for register writes.

regWrite ::= *lValue* `[` *expression* `:` *expression* `]` `<=` *expression*

The index expressions in the range notation follow the same rules as the corresponding expressions in variable assignment range updates (they must be static expressions and if they are not literal constants the right-hand side should have a defined bit width). Just as the indexed, partial register writes described in the previous subsection, multiple range-notation register writes cannot be mixed in the same rule[8].

Example: A range-notation register write

```
Reg#(Bit#(32)) r;

r[23:12] <= e; // Update a 12-bit range in the middle of r
```

[7] If multiple partial register writes are desired the best thing to do is to assign the register's value to a variable and then do cascaded variable assignments (as described in section 8.2)

[8] As described in the preceding footnote, using variable assignment is the best way to achive this effect, if desired.

8.4.3 Registers and struct member selection

regWrite ::= *lValue* . *identifier* `<=` *expression*

As with the square-bracket notation, a register update involving a field selection can mean one of two things. First, for a register containing a structure, it means update the particular field of the register value and write the result back to the register.

Example: Updating a register containing a structure

```
typedef struct { Bit#(32) a; Bit#(16) b; } Foo deriving(Bits);
...
Reg#(Foo) r;
...
r.a <= 17;
```

Second, it can mean to select the named field out of a compile-time structure that *contains* a register and write that register.

Example: Writing a register contained in a structure

```
typedef struct { Reg#(Bit#(32)) c; Reg#(Bit#(16)) d; } Baz;
...
Baz b;
...
b.a <= 23;
```

In both cases, the same notation is used and the compiler infers which interpretation is appropriate. As with square-bracket selection, struct member selection implies read desugaring, unless inhibited by `asReg` or `asIfc`.

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	::=	*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*
	|	*<functionBody>For*
	|	*<functionBody>While*
	|	*varDecl* | *varAssign*
	|	*varDo* | *varDeclDo*
	|	*functionDef*
	|	*functionStmt*
	|	*systemTaskStmt*
	|	`(` *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
```

8.8.1 Definition of functions by assignment

A function can also be defined using the following syntax.

functionProto ::= `function` *type identifier* `(` [*functionFormals*] `)` [*provisos*] `=` *expression* `;`

The part up to and including the *provisos* is the same as the standard syntax shown in Section 8.8. Then, instead of a semicolon, we have an assignment to an expression that represents the function body. The expression can of course use the function's formal arguments, and it must have the same type as the return type of the function.

Example 1. The factorial function, using recursion (from above:)

```
function int factorial (int n) =  (n<=1 ? 1 : n * factorial(n-1));
```

Example 2. Turning a method into a function. The following function definition:

```
  function int f1 (FIFO#(int) i);
     return i.first();
  endfunction
```

could be rewritten as:

```
function int f2(FIFO#(int) i) = i.first();
```

8.8.2 Function types

The function type is required for functions defined at the top level of a package and for recursive functions (such as the factorial examples above). You may choose to leave out the types within a function definition at lower levels for non-recursive functions,

If not at the top level of a package, Example 2 from the previous section could be rewritten as:

```
  function f1(i);
     return i.first();
  endfunction
```


or, if defining the function by assignment:

```
function f1 (i) = i.first();
```

Note that currently incomplete type information will be ignored. If, in the above example, partial type information were provided, it would be the same as no type information being provided. This may cause a type-checking error to be reported by the compiler.

```
function int f1(i) = i.first();  // The function type int is specified
                                 // The argument type is not specified
```

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.

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

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*
| *functionStmt*
| *systemTaskStmt*
| `(` *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*
| <*action*>*For*
| <*action*>*While*
| *regWrite*
| *varDecl* | *varAssign*
| *varDo* | *varDeclDo*
| *functionStmt*
| *systemTaskStmt*
| `(` *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*
	|	<*actionValue*>*For*
	|	<*actionValue*>*While*
	|	*regWrite*
	|	*varDecl* | *varAssign*
	|	*varDo* | *varDeclDo*
	|	*functionStmt*
	|	*systemTaskStmt*
	|	`(` *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.

Example: Using an actionvalue block to define a pop in a FIFO.

```
import FIFO :: *;

// Interface FifoWithPop combines first with deq
interface FifoWithPop#(type t);
   method Action enq(t data);
   method Action clear;
   method ActionValue#(t) pop;
endinterface

// Data is an alias of Bit#(8)
typedef Bit#(8) Data;

// The next function makes a deq and first from a fifo and returns an actionvalue block
function ActionValue#(t) fifoPop(FIFO#(t) f) provisos(Bits#(t, st));
   return(
      actionvalue
         f.deq;
         return f.first;
      endactionvalue
   );
endfunction

// Module mkFifoWithPop
(* synthesize, always_ready = "clear" *)
module mkFifoWithPop(FifoWithPop#(Data));

   // A fifo of depth 2
   FIFO#(Data) fifo <- mkFIFO;

   // methods
   method enq = fifo.enq;
   method clear = fifo.clear;
   method pop = fifoPop(fifo);
endmodule
```


9.8 Function calls

Function calls are expressed in the usual notation, i.e., a function applied to its arguments, listed in parentheses. If a function does not have any arguments, the parentheses are optional.

exprPrimary ::= *functionCall*

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

A function which has a result type of `Action` can be used as a statement when in the appropriate context.

functionStmt ::= *functionCall* `;`

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, if any, listed in parentheses. If the method does not have any arguments the parentheses are optional.

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. If the tagged union value does not have the tag corresponding to the member selection, the value is undefined. Example:

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

In this expression, if `orand` does not have the `Indexed` tag, the value is undefined. 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
    method 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	::=	[*attributeInstances*] `rules` [`:` *identifier*] *rulesStmt* `endrules` [`:` *identifier*]
rulesStmt	::=	*varDecl* | *varAssign*
	|	*rule*

A rule expression is optionally preceded by an *attributeInstances*; 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 }: begin
                                           pc <= pc+1;
                                           rf[r1] <= rf[r2] + rf[r3];
                                         end;
          tagged Jz {.r1, .r2}        : if (r1 == 0)
                                           begin
                                             pc <= r2;
                                           end;
       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,
```



```
   reset_prefix = "reset_b",
   clock_prefix = "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`). Integer literals can include the wildcard character `?` (example: `4'b00??`).

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. Integer literals can include the wildcard character `?`. An integer literal containing a wildcard will match any constant obtained by replacing each wildcard character by a valid digit. For example, `'h12?4` will match any constant between `'h1204` and `'h12f4` inclusive.
- 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 {regAddr: .ra, regIndex: .ri} : x = mem[ra+ri];
     endcase
```

Example:

```
Reg#(Bit#(16)) rg <- mkRegU;
  rule r;
    case (rg) matches
      'b_0000_000?_0000_0000: $display("1");
      'o_0?_00: $display("2");
      'h_?_0: $display("3");
      default: $display("D");
    endcase
  endrule
```

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 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 (` e_1 `matches` p_1 `&&&` e_2 `&&&` e_3 `matches` p_3 `)`
  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 tuple or struct 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.

Example:

```
   Reg#(Bit#(32)) a <-  mkReg(0);
   Tuple2#(Bit#(32),  Bool) data;

   rule r1;
      match {.in, .start} = data;
      //using "in" as a local variable
      a <= in;
   endrule
```


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.5.1 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.5.1.

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 `extend()` 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 `Int#(32)`.

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 (B.2.7). 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 (B.4), 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);
    method Action  _write (a x);
    method 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)
```

Since it is also occasionally desired to have automatically read interfaces that are not registers, BSV also provides a notation for more general suppression of read desugaring, producing an expression that always represents an interface itself:

```
asIfc(ifc)
```

12.6 FIFOs

Package FIFO (C.1.2) defines several useful interfaces and modules for FIFOs:

```
interface FIFO#(type a);
    method Action  enq (a x);
    method Action  deq;
    method a       first;
    method 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 (C.1.2) 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#(type a);
    method Action  enq (a x);
    method Action  deq;
    method a       first;
    method Action  clear;
    method Bool    notFull;
    method Bool    notEmpty;
endinterface: FIFOF
```



```
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. There are two types of system tasks; statements which are conceptually equivalent to `Action` functions, and calls which are conceptually equivalent to `ActionValue` and `Value` functions. Calls can be used within statements.

systemTaskStmt	::=	*systemTaskCall* `;`

12.8.1 Displaying information

systemTaskStmt	::=	*displayTaskName* `(` [*expression* [`,` *expression*]] `);`
displayTaskName	::=	`$display` \| `$displayb` | `$displayo` \| `$displayh`
	|	`$write` \| `$writeb` | `$writeo` \| `$writeh`

These system task statements 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[9]:

`%d`	Output a number in decimal format
`%b`	Output a number in binary format
`%o`	Output a number in octal format

[9] 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.

`%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
`%m`	Output hierarchical name

The values output are sized automatically to the largest possible value, with leading zeros, or in the case of decimal values, leading spaces. The automatic sizing of displayed data can be overridden by inserting a value `n` indicating the size of the displayed data. If `n=0` the output will be sized to minimum needed to display the data without leading zeros or spaces.

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.

Example:

```
$display ("%t", $time);
```

For display statements in different rules, the outputs will appear in the usual logical scheduling order of the rules. For multiple display statements within a single rule, technically there is no defined ordering in which the outputs should appear, since all the display statements are Actions within the rule and technically all Actions happen *simultaneously* in the atomic transaction. However, as a convenience to the programmer, the compiler will arrange for the display outputs to appear in the normal textual order of the source text, taking into accout the usual flow around if-then-elses, statically elaborated loops, and so on. However, for a rule that comprises separately compiled parts (for example, a rule that invokes a method in a separately compiled module), the system cannot guarantee the ordering of display statements across compilation boundaries. Within each separately compiled part, the display outputs will appear in source text order, but these groups may appear in any order. In particular, verification engineers should be careful about these benign (semantically equivalent) reorderings when checking the outputs for correctness.

12.8.2 $format

systemTaskCall ::= `$format` `(` [*expression* [`,` *expression*]] `)`

Bluespec also supports `$format`, a display related system task that does not exist in Verilog. `$format` takes the same arguments as the `$display` family of system tasks. However, unlike `$display` (which is a function call of type Action), `$format` is a value function which returns an object of type `Fmt` (section B.2.8). `Fmt` representations of data objects can thus be written hierarchically and applied to polymorphic types. The `FShow` package, described in Section C.7.8, defines a typeclass based on this capability.

Example:

```
typedef struct {OpCommand command;
        Bit#(8)   addr;
        Bit#(8)   data;
        Bit#(8)   length;
        Bool      lock;
        } Header deriving (Eq, Bits, Bounded);
```



```
function Fmt fshow (Header value);
    return ($format("<HEAD ")
        +
        fshow(value.command)
        +
        $format(" (%0d)", value.length)
        +
        $format(" A:%h",  value.addr)
        +
        $format(" D:%h>", value.data));
endfunction
```

12.8.3 File data type

`File` is a defined type in BSV which is defined as:

```
    typedef union tagged {
        void     InvalidFile ;
        Bit#(31) MCD;
        Bit#(31) FD;
    } File;
```

Type Classes for `File`									
	`Bits`	`Eq`	`Literal`	`Arith`	`Ord`	`Bounded`	`Bitwise`	`Bit Reduction`	`Bit Extend`
`File`	√	√					√		

Note: `Bitwise` operations are valid only for sub-type `MCD`.

12.8.4 Opening and closing file operations

systemTaskCall ::= `$fopen` `(` *fileName* [`,` *fileType*] `)`

systemTaskStmt ::= `$fclose` `(` *fileIdentifier* `)` `;`

The `$fopen` system call is of type `ActionValue` and can be used anywhere an `ActionValue` is expected. The argument *fileName* is of type String. `$fopen` returns a *fileIdentifier* of type `File`. If there is a *fileType* argument, the *fileIdentifier* returned is a file descriptor of type `FD`. If there is not a *fileType* argument, the *fileIdentifier* returned is a multi channel descriptor of type `MCD`.

One file of type `MCD` is pre-opened for append, `stdout_mcd` (value 1).

Three files of type `FD` are pre-opened; they are `stdin` (value 0), `stdout` (value 1), and `stderr` (value 2). `stdin` is pre-opened for reading and `stdout` and `stderr` are pre-opened for append.

The `fileType` determines, according to the following table, how other files of type `FD` are opened:

File Types for File Descriptors	
Argument	Description
`"r"` or `"rb"`	open for reading
`"w"` or `"wb"`	truncate to zero length or create for writing
`"a"` or `"ab"`	append; open for writing at end of file, or create for writing
`"r+"`, or `"r+b"`, or `"rb+"`	open for update (reading and writing)
`"w+"`, or `"w+b"`, or `"wb+"`	truncate or create for update
`"a+"`, or `"a+b"`, or `"ab+"`	append; open or create for update at end of file

The `$fclose` system call is of type `Action` and can be used in any context where an action is expected.

12.8.5 Writing to a file

systemTaskStmt	::=	*fileTaskName* (*fileIdentifier* , [*expression* [, *expression*]]) ;
fileTaskName	::=	`$fdisplay` \| `$fdisplayb` | `$fdisplayo` \| `$fdisplayh`
	|	`$fwrite` \| `$fwriteb` | `$fwriteo` \| `$fwriteh`

These system task calls are conceptually function calls of type `Action`, and can be used in any context where an action is expected. They correspond to the display tasks (`$display`, `$write`) but they write to specific files instead of to the standard output. They accept the same arguments (Section 12.8.1) as the tasks they are based on, with the addition of a first parameter *fileIdentifier* which indicates where to direct the file output.

Example:

```
Reg#(int) cnt <- mkReg(0);
let fh <- mkReg(InvalidFile) ;
let fmcd <- mkReg(InvalidFile) ;

rule open (cnt == 0 ) ;
   // Open the file and check for proper opening
   String dumpFile =  "dump_file1.dat" ;
   File lfh <- $fopen( dumpFile, "w" ) ;
   if ( lfh == InvalidFile )
      begin
         $display("cannot open %s", dumpFile);
         $finish(0);
      end
   cnt <= 1 ;
   fh <= lfh ;               // Save the file in a Register
endrule

rule open2 (cnt == 1 ) ;
   // Open the file and check for proper opening
   // Using a multi-channel descriptor.
   String dumpFile =  "dump_file2.dat" ;
   File lmcd <- $fopen( dumpFile ) ;
   if ( lmcd == InvalidFile )
      begin
         $display("cannot open %s", dumpFile );
         $finish(0);
      end
   lmcd = lmcd | stdout_mcd ;   // Bitwise operations with File MCD
   cnt <= 2 ;
   fmcd <= lmcd ;               // Save the file in a Register
endrule

rule dump (cnt > 1 );
   $fwrite( fh , "cnt = %0d\n", cnt);    // Writes to dump_file1.dat
   $fwrite( fmcd , "cnt = %0d\n", cnt);  // Writes to dump_file2.dat
   dump_file2.dat                        //  and stdout
   cnt <= cnt + 1;
endrule
```

12.8.6 Formatting output to a string

systemTaskStmt	::=	*stringTaskName* `(` *ifcIdentifier* `,` [*expression* [`,` *expression*]] `)` `;`
stringTaskName	::=	`$swrite` \| `$swriteb` | `$swriteo` \| `$swriteh` | `$sformat`

These system task calls are analogous to the `$fwrite` family of system tasks. They are conceptually function calls of type `Action`, and accept the same type of arguments as the corresponding `$fwrite` tasks, except that the first parameter must now be an interface with an `_write` method that takes an argument of type `Bit#(n)`.

The task `$sformat` is similar to `$swrite`, except that the second argument, and only the second argument, is interpreted as a format string. This format argument can be a static string, or it can be a dynamic value whose content is interpreted as the format string. No other arguments in `$sformat` are interpreted as format strings. `$sformat` supports all the format specifies supported by `$display`, as documented in 12.8.1.

The Bluespec compiler de-sugars each of these task calls into a call of an `ActionValue` version of the same task. For example:

```
$swrite(foo, "The value is %d", count);
```

de-sugars to

```
let x <- $swriteAV("The value is %d", count);
foo <= x;
```

An `ActionValue` value version is available for each of these tasks. The associated syntax is given below.

systemTaskCall	::=	*stringAVTaskName* `(` [*expression* [`,` *expression*]] `)`
stringAVTaskName	::=	`$swriteAV` \| `$swritebAV` | `$swriteoAV` \| `$swritehAV` | `$sformatAV`

The `ActionValue` versions of these tasks can also be called directly by the user.

Use of the system tasks described in this section allows a designer to populate state elements with dynamically generated debugging strings. These values can then be viewed using other display tasks (using the `%s` format specifier) or output to a `VCD` file for examination in a waveform viewer.

12.8.7 Reading from a file

systemTaskCall	::=	`$fgetc` `(` *fileIdentifier* `)`
systemTaskStmt	::=	`$ungetc` `(` *expression*`,` *fileIdentifier* `)` `;`

The `$fgetc` system call is a function of type `ActionValue#(int)` which returns an `int` from the file specified by *fileIdentifier*.

The `$ungetc` system statement is a function of type `Action` which inserts the character specified by *expression* into the buffer specified by *fileIdentifier*.

Example:

```
rule open ( True ) ;
   String readFile = "gettests.dat";
   File lfh <- $fopen(readFile, "r" ) ;

   int i <- $fgetc( lfh );
   if ( i != -1 )
      begin
```

```
         Bit#(8) c = truncate( pack(i) ) ;
      end
   else // an error occurred.
      begin
         $display( "Could not get byte from %s",
            readFile ) ;
      end

   $fclose ( lfh ) ;
   $finish(0);
endrule
```

12.8.8 Flushing output

systemTaskStmt ::= `$fflush` `(` [*fileIdentifier*] `)` `;`

The system call `$fflush` is a function of type `Action` and can be used in any context where an actions is expected. The `$fflush` function writes any buffered output to the file(s) specified by the *fileIdentifier*. If no argument is provided, `$fflush` writes any buffered output to all open files.

12.8.9 Stopping simulation

systemTaskStmt ::= `$finish` [`(` *expression* `)`] `;` *systemTaskStmt* ::= `$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.

12.8.10 VCD dumping

systemTaskStmt ::= `$dumpvars` | `$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.

12.8.11 Time functions

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.

12.8.12 Real functions

There are two system tasks defined for the `Real` data type (section B.2.6), used to convert between `Real` and IEEE standard 64-bit vector representation, `$realtobits` and `$bitstoreal`. They are identical to the Verilog functions.

systemTaskCall ::= `$realtobits` `(` *expression* `)`

systemTaskCall ::= `$bitstoreal` `(` *expression* `)`

12.8.13 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 BSV's attribute syntax.

attributeInstances ::= *attributeInstance*
{ *attributeInstance* }

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.

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.

The attributes `synthesize` and `noinline` control code generation for top-level modules and functions, respectively.

Attribute name	Section	Top-level module definitions	Top-level function definitions
synthesize	13.1.1	√	
noinline	13.1.2		√

13.1.1 `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 `noinline`

The `noinline` attribute is applied to functions, instructing the compiler to generate a separate module for the function. This module is instantiated as many times as required by its callers. When used in complicated calling situations, the use of the `noinline` attribute can simplify and speed up compilation. The `noinline` attribute can only be applied to functions that are defined at the top level and the inputs and outputs of the function must be in the typeclass `Bits`.

Example:

```
(* noinline *)
function Bit#(LogK) popCK(Bit#(K) x);
  return (popCountTable(x));
endfunction: popCK
```

13.2 Interface attributes

Interface attributes express protocol and naming requirements for generated Verilog interfaces. They are considered during generation of the Verilog module which uses the interface. These attributes can be applied to synthesized modules, methods, interfaces, and subinterfaces at the top level only. If the module is not synthesized, the attribute is ignored. The following table shows which attributes can be applied to which elements. These attributes cannot be applied to `Clock`, `Reset`, or `Inout` subinterface declarations.

Attribute name	Section	Synthesized module definitions	Interface type declarations	Methods of interface type declarations	Subinterfaces of interface type declarations
ready=	13.2.1			√	
enable=	13.2.1			√	
result=	13.2.1			√	
prefix=	13.2.1			√	√
port=	13.2.1			√	
always_ready	13.2.2	√	√	√	√
always_enabled	13.2.2	√	√	√	√

There is a direct correlation between interfaces in Bluespec and ports in the generated Verilog. These attributes can be applied to interfaces to control 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. In the ActionValue method `method1` shown below

```
method ActionValue#( type_out )  method1 ( type_in data_in ) ;
```

the following ports are generated:

`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 Renaming attributes

ready= and enable= Ready and enable ports use `RDY_` and `EN_` as the default prefix to the method names. The attributes `ready=` "*string*" and `enable=` "*string*" replace the prefix annotation and method name with the specified string as the name instead. These attributes may be associated with method declarations (*methodProto*) only (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 as those are annotated as `always_enabled`.

result= By default the output port for value methods and `ActionValue` methods use the method name. The attribute `result` = "*string*" causes the output to be renamed to the specified string. This is useful when the desired port names must begin with an upper case letter, which is not valid for a method name. These attributes may be associated with method declarations (*methodProto*) only (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.

prefix= and port= By default, the input ports for methods are named by using the name of the method as the prefix and the name of the method argument as the suffix, into `method_argument`. The prefix and/or suffix name can be replaced by the attributes `prefix=` "*string*" and `port=` "*string*". By combining these attributes any desired string can be generated. The `prefix=` attribute replaces the method name and the `port=` attribute replaces the argument name in the generated Verilog port name. The prefix string may be empty, in which case the joining underscore is not added.

The `prefix=` attribute may be associated with method declarations (*methodProto*) or sub-interface declarations (*subinterfaceDecl*). The `port=` attribute may be associated with each method prototype argument in the interface declaration (*methodProtoFormal*) (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 (Section 5.2.1.) Using the `prefix` attribute on the subinterface is a way of replacing the sub-interface name. This is demonstrated in the example in Section 13.2.3.

13.2.2 Port protocol attributes

The port protocol attributes `always_enabled` and `always_ready` remove unnecessary ports. In all cases the compiler verifies that the attributes are correctly applied.

The attribute `always_enabled` specifies that no enable signal will be generated for the associated interface methods. The methods will be executed on every clock cycle and the compiler verifies that the caller does this.

The attribute `always_ready` specifies that no ready signals will 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 associated with the method declarations (*methodProto*), the sub-interface declarations (*subinterfaceDecl*), or the interface declaration (*interfaceDecl*) itself. In these cases, the attribute does not take any arguments. Example:

```
interface Test;
   (* always_enabled *)
   method ActionValue#(Bool) check;
endinterface: Test
```

The attributes can also be associated with a module, in which case the attribute can have as an argument the list of methods to which the attribute is applied. When associated with a module, the attributes are applied when the interface is implemented by 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 this example, note that only the `enq` method of the subifc interface is `always_ready`. Other methods of the interface, such as `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 when applying the attribute to a module. Example:

```
(* always_enabled *)
module mkServer (ILookup);
```

13.2.3 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

Attribute name	Section	Module definitions	`rule` definitions	`rules` expressions
fire_when_enabled	13.3.1		√	
no_implicit_conditions	13.3.2		√	
descending_urgency	13.3.3	√	√	√
execution_order	13.3.4	√	√	√
mutually_exclusive	13.3.5	√	√	√
conflict_free	13.3.6	√	√	√
preempts	13.3.7	√	√	√

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, but some can also be added to module definitions.

The scheduling attributes are are only applied when the module is synthesized.

13.3.1 `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 the rule conditions are true, the attribute checks that there are no scheduling conflicts with other rules that will prevent it from firing. This is statically verified by the compiler. If the rule won't fire, the compiler will report an error.

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,
   reset_prefix = "reset_b",
   clock_prefix = "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);
     counter <= 1;
   endrule

   //  Next rule updates the counter.
   rule updateCounter;
     counter <= counter + 1;
   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. If the rule `updateCounter` is more urgent, only the rule `updateCounter` will fire. In this case, the assertion `fire_when_enabled` will be violated and the compiler will produce an error message. Note that without the assertion `fire_when_enabled` the compilation process will be correct.

13.3.2 `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; only the explicit rule predicate controls whether the rule can fire 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,
   reset_prefix = "reset_b",
   clock_prefix = "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;
      if (counter != '1)
        counter <= counter + 1;
    endrule

    // Next rule resets the counter to a value stored in the valueFifo
    (* no_implicit_conditions *)
    rule resetCounter (counter == '1);
      counter <= valueFifo.first();
      valueFifo.deq();
    endrule

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

    // Update the valueFifo
    method Action setReset(CounterType a);
      valueFifo.enq(a);
    endmethod
endmodule
```

The assertion `no_implicit_conditions` is incorrect 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. 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 `descending_urgency`

The compiler maps rules into clocks, as described in Section 6.2.2. In each clock, amongst all the rules that can fire 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 their conditions are true so both can execute. 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 one of three syntactic positions:

- It can be placed just before the `module` keyword in a module definitions (Section 5.3), in which case it can refer directly to any of the rules inside the module.
- It can be placed just before the `rule` keyword in a rule definition, (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 `rules` keyword in a rules expression (Section 9.13), in which case it can refer directly to any of the rules in the expression.

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 warning 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,
   reset_prefix = "reset_b",
   clock_prefix = "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 may 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 execution_order

With the `execution_order` attribute, the designer can specify that, when two rules fire in the same cycle, one rule should sequence before the other. This attribute is similar to the `descending_urgency` attribute (section 13.3.3) except that it specifies the execution order instead of the urgency order. The `execution_order` attribute may occur in the same syntactic positions as the `descending_urgency` attribute (Section 13.3.3) and takes a similar argument, a string containing a comma-separated list of rule names. Example:

```
(* execution_order = "r1, r2, r3" *)
```


This example specifies that r1 should execute before r2 which, in turn, should execute before r3.

If two rules cannot execute in the order specified, because of method calls which must sequence in the opposite order, for example, then the two rules are forced to conflict.

13.3.5 mutually_exclusive

The scheduler always attempts to deduce when two rules are mutually exclusive (based on their predicates). However, this deduction can fail even when two rules are actually exclusive, either because the scheduler effort limit is exceeded or because the mutual exclusion depends on a higher-level invariant that the scheduler does not know about. The mutually_exclusive attribute allows the designer to overrule the scheduler's deduction and forces the generated schedule to treat the annotated rules as exclusive. The mutually_exclusive attribute may occur in the same syntactic positions as the descending_urgency attribute (Section 13.3.3) and takes a similar argument, a string containing a comma-separated list of rule names. Example:

```
(* mutually_exclusive = "r1, r2, r3" *)
```

This example specifies that every pair of rules that are in the annotation (i.e (r1, r2), (r1, r3), and (r2, r3)) is a mutually-exclusive rule pair.

Since an asserted mutual exclusion does not come with a proof of this exclusion, the compiler will insert code that will check and generate a runtime error if two rules ever execute during the same clock cycle during simulation. This allows a designer to find out when their use of the mutually_exclusive attribute is incorrect.

13.3.6 conflict_free

Like the mutually_exclusive rule attribute (section 13.3.5), the conflict_free rule attribute is a way to overrule the scheduler's deduction about the relationship between two rules. However, unlike rules that are annotated mutually_exclusive, rules that are conflict_free may fire in the same clock cycle. Instead, the conflict_free attribute asserts that the annotated rules will not make method calls that are inconsistent with the generated schedule when they execute.

The conflict_free attribute may occur in the same syntactic positions as the descending_urgency attribute (Section 13.3.3) and takes a similar argument, a string containing a comma-separated list of rule names. Example:

```
(* conflict_free = "r1, r2, r3" *)
```

This example specifies that every pair of rules that are in the annotation (i.e (r1, r2), (r1, r3), and (r2, r3)) is a conflict-free rule pair.

For example, two rules may both conditionally enqueue data into a FIFO with a single enqueue port. Ordinarily, the scheduler would conclude that the two rules conflict since they are competing for a single method. However, if they are annotated as conflict_free the designer is asserting that when one rule is enqueuing into the FIFO, the other will not be, so the conflict is apparent, not real. With the annotation, the schedule will be generated as if any conflicts do not exist and code will be inserted into the resulting model to check if conflicting methods are actually called by the conflict free rules during simulation.

It is important to know the conflict_free attribute's capabilities and limitations. The attribute works with more than method calls that totally conflict (like the single enqueue port). During simulation, it will check and report any method calls amongst conflict_free rules that are inconsistent with the generated schedule (including registers being read after they have been written and wires being written after they are read). On the other hand, the conflict_free attribute does not overrule the scheduler's deductions with respect to resource usage (like uses of a multi-ported register file).

13.3.7 `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 list can be executed and are scheduled to fire, then none of the rules specified in the second list 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 `r1` preempts `r2`, then the compiler forces a conflict and gives `r1` priority. If `r1` is able to fire, but is not scheduled to, then `r2` 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 Evaluation behavior attributes

13.4.1 `split` and `nosplit`

Attribute name	Section	Action statements	ActionValue statements
split/nosplit	13.4.1	√	√

The `split/nosplit` attributes are applied to `Action` and `ActionValue` statements, but cannot preceed certain expressions inside an `action/endaction` including `return`, variable declarations, instantiations, and `function` statements.

When a rule contains an `if` (or `case`) statement, the compiler has the option either of splitting the rule into two mutually exclusive rules, or leaving it as one rule for scheduling but using MUXes in the production of the action. Rule splitting can sometimes be desirable because the two split rules are scheduled independently, so non-conflicting branches of otherwise conflicting rules can be scheduled concurrently. Splitting also allows the split fragments to appear in different positions in the logical execution order, providing the effect of condition dependent scheduling.

Splitting is turned *off* by default for two reasons:

- When a rule contains many `if` statements, it can lead to an exponential explosion in the number of rules. A rule with 15 `if` statements might split into 2^{15} rules, depending on how independent the statements and their branch conditions are. An explosion in the number of rules can dramatically slow down the compiler and cause other problems for later compiler phases, particularly scheduling.

- Splitting propagates the branch condition of each `if` to the predicates of the split rules. Resources required to compute rule predicates are reserved on every cycle. If a branch condition requires a scarce resource, this can starve other parts of the design that want to use that resource.

The `split` and `nosplit` attributes override any compiler flags, either the default or a flag entered on the command line (`-split-if`).

The `split` attribute splits all branches in the statement immediately following the attribute statement, which must be an `Action` statement. A `split` immediately preceeding a binding (e.g. `let`) statement is not valid. If there are nested `if` or `case` statements within the split statement, it will continue splitting recursively through the branches of the statement. The `nosplit` attribute can be used to disable rule splitting within nested `if` statements.

Example:

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

   rule finish ;
     (*split*)
     if (a == 3)
        begin
           done <= True;
        end
     else
        (*nosplit*)
        if (a == 0)
           begin
              done <= False;
              a    <= 1;
           end
        else
           begin
              done <= False;
           end
   endrule
endmodule
```

To enable rule splitting for an entire design, use the compiler flag `-split-if` at compile time. See the user guide for more information on compiler flags. You can enable rule splitting for an entire design with the `-split-if` flag and then disable the effect for specific rules, by specifying the `nosplit` attribute before the rules you do not want to split.

13.5 Input clock and reset attributes

The following attributes control the definition and naming of clock oscillator, clock gate, and reset ports. The attributes can only be applied to top-level module definitions.

Attribute name	Section	Top-level module
clock_prefix=	13.5.1	√
gate_prefix=	13.5.1	√
reset_prefix=	13.5.1	√
gate_input_clocks=	13.5.2	√
gate_all_clocks	13.5.2	√
default_clock_osc=	13.5.3	√
default_clock_gate=	13.5.3	√
default_gate_inhigh	13.5.3	√
default_gate_unused	13.5.3	√
default_reset=	13.5.3	√
clock_family=	13.5.4	√
clock_ancestors=	13.5.4	√

13.5.1 Clock and reset prefix naming attributes

The generated port renaming attributes `clock_prefix=`, `gate_prefix=`, and `reset_prefix=` rename the ports for the clock oscillators, clock gates, and resets in a module by specifying a prefix string to be added to each port name. The prefix is used *only* when a name is not provided for the port, (as described in Sections 13.5.3 and 13.6.1), requiring that the port name be created from the prefix and argument name. The attributes are associated with a module and are only applied when the module is synthesized.

Clock Prefix Naming Attributes		
Attribute	Default name	Description
`clock_prefix=`	CLK	Provides the prefix string to be added to port names for all the clock oscillators in a module.
`gate_prefix=`	CLK_GATE	Provides the prefix string to be added to port names for all the clock gates in a module.
`reset_prefix=`	RST_N	Provides the prefix string to be added to port names for all the resets in a module.

If a prefix is specified as the empty string, then no prefix will be used when creating the port names; that is the argument name alone will be used as the name.

Example:

```
(* synthesize, clock_prefix = "CK" *)
module mkMod(Clock clk2, ModIfc ifc);
```

generates the following in the Verilog:

```
module mkMod (CK, RST_N, CK_clk2, ...
```

Where `CK` is the default clock (using the user-supplied prefix), `RST_N` is the default reset (using the default prefix), and `CK_clk2` is the oscillator for the input `clk2` (using the user-supplied prefix).

13.5.2 Gate synthesis attributes

When a module is synthesized, one port, for the oscillator, is created for each clock input (including the default clock). The gate for the clock is defaulted to a logical 1. The attributes `gate_all_clocks` and `gate_input_clocks=` specify that a second port be generated for the gate.

The attribute `gate_all_clocks` will add a gate port to the default clock and to all input clocks. The attribute `gate_input_clocks=` is used to individually specify each input clock which should have a gate supplied by the parent module.

If an input clock is part of a vector of clocks, the gate port will be added to all clocks in the vector. Example:

```
(* gate_input_clock = "clks, c2" *)
module mkM(Vector#(2, Clock) clks, Clock c2);
```

In this example, a gate port will be added to both the clocks in the vector `clks` and the clock `c2`. A gate port cannot be added to just one of the clocks in the vector `clks`.

The `gate_input_clocks=` attribute can be used to add a gate port to the default clock. Example:

```
( * gate_input_clocks = "default_clock" * )
```

Note that by having a gate port, the compiler can no longer assume the gate is always logical 1. This can cause an error if the clock is connected to a submodule which requires the gate to be logical 1.

The gate synthesis attributes are associated with a module and are only applied when the module is synthesized.

13.5.3 Default clock and reset naming attributes

The default clock and reset naming attributes are associated with a module and are only applied when the module is synthesized.

The attributes `default_clock_osc=`, `default_clock_gate=`, and `default_reset=` provide the names for the default clock oscillator, default gate, and default reset ports for a module. When a name for the default clock or reset is provided, any prefix attribute for that port is ignored.

The attributes `default_gate_inhigh` and `default_gate_unused` indicate that a gate port should not be generated for the default clock and whether the gate is always logical 1 or unused. The default is `default_gate_inhigh`. This is only necessary when the attribute `gate_all_clocks` (section 13.5.2) has been used.

The attributes `no_default_clock` and `no_default_reset` are used to remove the ports for the default clock and the default reset.

Default Clock and Reset Naming Attributes	
Attribute	Description
`default_clock_osc=`	Provides the name for the default oscillator port.
`no_default_clock`	Removes the port for the default clock.
`default_clock_gate=`	Provides the name for the default gate port.
`default_gate_inhigh`	Removes the gate ports for the module and the gate is always high.
`default_gate_unused`	Removes the gate ports for the module and the gate is unused.
`default_reset=`	Provides the name for the default reset port.
`no_default_reset`	Removes the port for the default reset.

13.5.4 Clock family attributes

The `clock_family` and `clock_ancestors` attributes indicate to the compiler that clocks are in the same domain in situations where the compiler may not recognize the relationship. For example, when clocks split in synthesized modules and are then recombined in a subsequent module, the compiler may not recognize that they have a common ancestor. The `clock_ancestors` and `clock_family` attributes allow the designer to explicitly specify the family relationship between the clocks. These attributes are applied to modules only.

The `clock_ancestors` attribute specifies an ancestry relationship between clocks. A clock is a gated version of its ancestors. In other words, if `clk1` is an ancestor of `clk2` then `clk2` is a gated version of `clk1`, as specified in the following statement:

```
(* clock_ancestors = "clk1 AOF clk2" *)
```

Multiple ancestors as well as multiple independent groups can be listed in a single attribute statement. For example:

```
(* clock_ancestors = "clk1 AOF clk2 AOF clk3, clk1 AOF clk4, clka AOF clkb" *)
```

The above statement specifies that `clk1` is an ancestor of `clk2`, which is itself an ancestor of `clk3`; that `clk1` is also an ancestor of `clk4`; and that `clka` is an ancestor of `clkb`. You can also repeat the attribute statement instead of including all clock ancestors in a single statement. Example:

```
(* clock_ancestors = "clk1 AOF clk2 AOF clk3" *)
(* clock_ancestors = "clk1 AOF clk4" *)
(* clock_ancestors = "clka AOF clkb" *)
```

For clocks which do not have an ancestor relationship, but do share a common ancestor, you can use the `clock_family` attribute. Clocks which are in the same family have the same oscillator with a different gate. To be in the same family, one does not have to be a gated version of the other, instead they may be gated versions of a common ancestor.

```
(* clock_family = "clk1, clk2, clk3" *)
```

Note that `clock_ancestors` implies `same_family`.

13.6 Module argument attributes

The attributes in this section are applied to module arguments. The following table shows which type of module argument each attribute can be applied to. Each attribute can be applied to vectors of arguments as well.

Attribute name	Section	Clock/ vector of clock	Reset/ vector of reset	Value argument	Inout/ vector of inouts
osc=	13.6.1	√			
gate=	13.6.1	√			
gate_inhigh	13.6.1	√			
gate_unused	13.6.1	√			
reset=	13.6.1		√		
clocked_by=	13.6.2		√	√	√
reset_by=	13.6.3			√	√
port=	13.6.4			√	√

13.6.1 Argument-level clock and reset naming attributes

The non-default clock and reset inputs to a module will have a port name created using the argument name and any associated prefix for that port type. This name can be overridden on a per-argument basis by supplying argument-level attributes that specify the names for the ports.

These attributes are applied to the clock module arguments, except for `reset=` which is applied to the reset module arguments.

Argument-level Clock and Reset Naming Attributes		
Attribute	Applies to	Description
`osc=`	Clock or vector of clocks module arguments	Provides the full name of the oscillator port.
`gate=`	Clock or vector of clocks module arguments	Provides the full name of the gate port.
`gate_inhigh`	Clock or vector of clocks module arguments	Indicates that the gate port should be omitted and the gate is assumed to be high.
`gate_unused`	Clock or vector of clocks module arguments	Indicates that the gate port should be omitted and is never used within the module.
`reset=`	Reset or vector of resets module arguments	Provides the full name of the reset port.

Example:

```
(* synthesize *)
module mkMod((* osc="ACLK", gate="AGATE" *) Clock clk,
             (* reset="RESET" *) Reset rst,
             ModIfc ifc);
```

generates the following in the Verilog:

```
module mkMod(CLK, RST_N, ACLK, AGATE, RESET, ...
```

The attributes can be applied to the base name generated for a vector of clocks, gates or resets. Example:

```
(* synthesize *)
module mkMod((* osc="ACLK", gate="AGATE" *) Vector#(2, Clock) clks,
             (* reset="ARST" *) Vector#(2, Reset) rsts,
             ModIfc ifc);
```

generates the following in the Verilog:

```
module mkMod(CLK, RST_N, ACLK_0, AGATE_0, ACLK_1, AGATE_1, ARST_0, ARST_1,...
```

13.6.2 `clocked_by=`

The attribute `clocked_by=` allows the user to assert which clock a reset, inout, or value module argument is associated with, to specify that the argument has `no_clock`, or to associate the argument with the `default_clock`. If the `clocked_by=` attribute is not provided, the default clock will be used for inout and value arguments; the clock associated with a reset argument is dervied from where the reset is connected.

Examples:

```
module mkMod (Clock c2, (* clocked_by="c2" *) Bool b,
              ModIfc ifc);
module mkMod (Clock c2, (* clocked_by="default_clock" *) Bool b,
              ModIfc ifc);
module mkMod (Clock c2, (* clocked_by="c2" *) Reset rstIn,
                        (* clocked_by="default_clock" *) Inout q_inout,
                        (* clocked_by="c2" *) Bool b,
              ModIfc ifc);
```

To specify that an argument is not associated with any clock domain, the clock `no_clock` is used. Example:

```
module mkMod (Clock c2, (* clocked_by="no_clock" *) Bool b,
              ModIfc ifc);
```

13.6.3 reset_by=

The attribute `reset_by=` allows the user to assert which reset an inout or value module argument is associated with, to specify that the argument has `no_reset`, or to associate the argument with the `default_reset`. If the `reset_by=` attribute is not provided, the default reset will be used.

Examples:

```
module mkMod (Reset r2, (* reset_by="r2" *) Bool b,
              ModIfc ifc);

module mkMod (Reset r2, (* reset_by="default_reset" *) Inout q_inout,
              ModIfc ifc);
```

To specify that the port is not associated with any reset, `no_reset` is used. Example:

```
module mkMod (Reset r2, (* reset_by="no_reset" *) Bool b,
              ModIfc ifc);
```

13.6.4 port=

The attribute `port=` allows renaming of value module arguments. These are port-like arguments that are not clocks, resets or parameters. It provides the full name of the port generated for the argument. This is the same attribute as the `port=` attribute in Section 13.2.1, as applied to module arguments instead of interface methods.

13.7 Documentation attributes

A BSV design can specify comments to be included in the generated Verilog by use of the `doc` attribute.

Attribute name	Section	Top-level module definitions	Submodule instantiations	`rule` definitions	`rules` expressions
doc=	13.7	$\surd$	$\surd$	$\surd$	$\surd$

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.

13.7.1 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. If the module is not synthesized, the `doc` attributes are ignored.

Example:

```
(* synthesize *)
(* doc = "This is important information about the following module" *)
module mkMod (IFC);
  ...
endmodule
```

13.7.2 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 and the register is inlined (as is the default), 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;
...
```


13.7.3 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 executed. This can be achieved with a `doc` attribute attached to a BSV rule declaration or rules expression.

The `doc` attribute can be attached to any `rule..endrule` or `rules...endrules` 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	::=	*typelist* `determines` *typelist*
typelist	::=	*typeIde* | `(` *typeIde* { `,` *typeIde* } `)`
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);
    function Bool inLiteralRange(a target, Integer i);
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.).

The typeclass also provides a function `inLiteralRange` that takes an argument of type `a` and an `Integer` and returns a `Bool`. In the standard `Literal` typeclass this boolean indicates whether or not the supplied `Integer` is in the range of legal values for the type `a`.

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):[10]

```
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);
    function data_t \/ (data_t x, data_t y);
    function data_t \% (data_t x, data_t y);
endtypeclass
```

This defines the type class `Arith` with super type class `Literal`, i.e., the proviso states that in order for a type `data_t` to be an instance of `Arith` it must also be an instance of the type class `Literal`. Further, it has six 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, multiplication, and division defined on it.

The semantics of a dependency say that once the types on the left of the `determines` keyword are fixed, the types on the right are uniquely determined. The types on either side of the list can be a single type or a list of types, in which case they are enclosed in parentheses.

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
```

[10] 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.)

For any type `t` we know that `Get#(t)` and `Put#(t)` are connectable because of the following declaration in the `GetPut` package:

```
instance Connectable#(Get#(element_type), Put#(element_type));
```

In the `Connectable` dependency above, it states that `a` determines `b`. Therefore, you know that if `a` is `Get#(t)`, the *only* possibility for `b` is `Put#(t)`.

Example of a typeclass definition with lists of types in the dependencies:

```
typeclass Extend #(type a, type b, type c)
    dependencies ((a,c) determines b, (b,c) determines a);
endtypeclass
```

An example of a case where the dependencies are not commutative:

```
typeclass Bits#(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 require that `b` had to be 16; but the 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* | *moduleDef* }
		`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 *moduleDef*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.

Example:

```
typedef Bit#(8) MyType;
// MyType is an alias of Bit#(8)

typedef SizeOf#(MyType) NumberOfBits;
// NumberOfBits is a numeric type, its value is 8

Integer ordinaryNumber = valueOf(NumberOfBits);
// valueOf converts a numeric type into Integer
```

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 - the function map, as defined in the package `Vector` (C.2.8):


```
function Vector#(vsize, b_type) map (function  b_type  func (a_type x),
                                    Vector#(vsize, a_type) xvect);
    Vector#(vsize, b_type) yvect = newVector;

    for (Integer j = 0; j < valueof(vsize); j=j+1)
        yvect[j] = func (xvect[j]);

    return yvect;
endfunction: map

function int sqr (int x);
   return x * x;
endfunction: sqr

Vector#(100,int) avect = ...; // initialize vector avect

Vector#(100,int) bvect = map (sqr, avect);
```

The function `map` is polymorphic, i.e., is defined for any size type `vsize` and value types `a_type` and `b_type`. It takes two arguments:

- A function `func` with input of type `a_type` and output of type `b_type`.
- A vector `xvect` of size `vsize` containing values of type `a_type`.

Its result is a new vector `yvect` that is also of size `vsize` and containing values of type `b_type`, such that `yvect[j]=func(xvect[j])`. In the last line of the example, we call `map` passing it the `sqr` function and the vector `avect` to produce a vector `bvect` that contains the squared versions of all the elements of vector `avect`.

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 `map`, the identifier `func` is a function-valued identifier, and the expression `func (xsize [j])` invokes the function.

The function `map` could be called with a variety of arguments:

```
// Apply the extend function to each element of avect
Vector#(13, Bit#(5)) avect;
Vector#(13, Bit#(10)) bvect;
...
bvect = map(extend, avect);
```

or

```
// test all elements of avect for even-ness
Vector#(100,Bool) bvect = map (isEven, avect);
```

In other words, `map` captures, in one definition, the generic idea of applying some function to all elements of a vector and returning all the results in another vector. 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 element_type);
    ... usual enq() and deq() methods ...
```



```
    method Bool search (element_type key);

endinterface: SearchableFIFO

module mkSearchableFIFO#(function Bool test_func
                         (element_type x, element_type key))
                         (SearchableFIFO#(element_type));
    ...
    method Bool search (element_type key);
        ... apply test_func(x, key) 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 `key`, and it searches the FIFO using that key, returning `True` if the search succeeds.

Inside the `mkSearchableFIFO` module, the method applies some element test predicate `test_func` to each element of the FIFO and ORs all the results. The particular element-test function `test_func` 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. Since polymporphic modules cannot be synthesized, for synthesis a non-polymorphic version of the module would have to be instantiated.

15 Embedding Verilog in a BSV design

This section describes how to embed a Verilog module in a BSV module. This is the method to utilize existing Verilog components, Verilog components generated by other tools or 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 through BVI import. To embed a Verilog module you create a BSV wrapper around a Verilog module, defining the Verilog port connections and associating the BSV parameters with the Verilog ports.

externModuleImport ::= `import "BVI"` [*identifier* `=`] *moduleProto*
  { *moduleStmt* }
  { *importBVIStmt* }
  `endmodule` [`:` *identifier*]

The body consists of a sequence of *importBVIStmts*:

importBVIStmt	::=	*parameterBVIStmt*
	|	*methodBVIStmt*
	|	*portBVIStmt*
	|	*inputClockBVIStmt*
	|	*defaultClockBVIStmt*
	|	*outputClockBVIStmt*

| *inputResetBVIStmt*
| *defaultResetBVIStmt*
| *noResetBVIStmt*
| *ouputResetBVIStmt*
| *ancestorBVIStmt*
| *sameFamilyBVIStmt*
| *scheduleBVIStmt*
| *pathBVIStmt*
| *inoutBVIStmt*

The optional *identifier* immediately following the "BVI" is the name of the Verilog module to be imported. This will usually be found in a Verilog file of the same name (*identifier.v*). If this *identifier* is excluded, it is assumed that the Verilog module name is the same as the BSV name of the module.

The *moduleProto* is the first line in the module definition as described in Section 5.3.

The BSV wrapper returns an interface. All arguments and return values must be in the Bits class or be of type Clock, Reset, or a subinterface which meets these requirements. Note that the BSV module's parameters have no inherent relationship to the Verilog module's parameters. The BSV wrapper is used to connect the Verilog ports to the BSV parameters, performing any data conversion, such as packs or unpacks, as necessary.

Example of the header of a BVI import statement:

```
import "BVI" RWire =
   module RWire (VRWire#(a))
      provisos (Bits#(a,sa));
   ...
   endmodule: vMkRWire
```

Since the Verilog module's name matches the BSV name, the header could be also written as:

```
import "BVI"
   module RWire (VRWire#(a))
      provisos (Bits#(a,sa));
   ...
   endmodule: vMkRWire
```

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 definitions, 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 in this section.

The *importBVIStmt*s must occur at the end of the body, after the *moduleStmt*s. They may be written in any order.

The following is an example of embedding a Verilog SRAM model in BSV. The Verilog file is shown after the BSV wrapper.

```
import "BVI" mkVerilog_SRAM_model =
  module mkSRAM #(String filename) (SRAM_Ifc #(addr_t, data_t))
  provisos(Bits#(addr_t, addr_width),
           Bits#(data_t, data_width));
     parameter FILENAME      = filename;
     parameter ADDRESS_WIDTH = valueOf(addr_width);
```



```
      parameter DATA_WIDTH    = valueof(data_width);
      method request (v_in_address, v_in_data, v_in_write_not_read)
                      enable (v_in_enable);
      method v_out_data  read_response;
      default_clock clk(clk, (*unused*) clk_gate);
      default_reset no_reset;
      schedule (read_response) SB (request);
  endmodule
```

This is the Verilog module being wrapped in the above BVI import statement.

```
    module mkVerilog_SRAM_model (clk,
                                 v_in_address, v_in_data,
                                 v_in_write_not_read,
                                 v_in_enable,
                                 v_out_data);
      parameter FILENAME      = "Verilog_SRAM_model.data";
      parameter ADDRESS_WIDTH = 10;
      parameter DATA_WIDTH    = 8;
      parameter NWORDS        = (1 << ADDRESS_WIDTH);

      input                       clk;
      input  [ADDRESS_WIDTH-1:0]  v_in_address;
      input  [DATA_WIDTH-1:0]     v_in_data;
      input                       v_in_write_not_read;
      input                       v_in_enable;

      output [DATA_WIDTH-1:0]     v_out_data;
      ...
    endmodule
```

15.1 Parameter statement

The parameter statement specifies the parameter values which will be used by the Verilog module.

parameterBVIStmt ::= `parameter` *identifier* = *expression* ;

The value of *expression* is supplied to the Verilog module as the parameter named *identifier*. The *expression* must be a compile-time constant. The valid types for parameters are `String`, `Integer` and `Bit#(n)`. Example:

```
     import "BVI" ClockGen =
     module vAbsoluteClock#(Integer start, Integer period)
                           ( ClockGenIfc );
         let halfPeriod =  period/2 ;
         parameter initDelay  = start;              //the parameters start,
         parameter v1Width = halfPeriod ;           //halfPeriod and period
         parameter v2Width = period - halfPeriod ;  //must be compile-time constants
     ...
     endmodule
```

15.2 Method

The `method` statement is used to connect methods in a Bluespec interface to the appropriate Verilog wires. The syntax imitates a function prototype in that it doesn't define, but only declares. In the case of the `method` statement, instead of declaring types, it declares ports.

methodBVIStmt ::= `method` [*portId*] *identifer* [`(` [*portId* { `,` *portId* }] `)`]
[`enable` `(`*portId* `)`] [`ready` `(` *portId*`)`]
[`clocked_by` `(` *clockId*`)`] [`reset_by` `(` *resetId*`)`] `;`

The first *portId* is the output port for the method, and is only used when the method has a return value. The *identifier* 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, if there are any. There may follow up to four optional clauses (in any order): `enable` (for the enable input port if the method has an `Action` component), `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 associated reset signal, otherwise the default reset will be assumed). If no `ready` port is given, the constant value 1 is used meaning the method is always ready. The names `no_clock` and `no_reset` can be used in `clocked_by` and `reset_by` clauses indicating that there is no associated clock and no associated reset, respectively.

If the input port list is empty and none of the optional clauses are specified, the list and its parentheses may be omitted. If any of the optional clauses are specified, the empty list `()` must be shown. Example:

```
method CLOCKREADY_OUT clockready() clocked_by(clk);
```

If there was no `clocked_by` statement, the following would be allowed:

```
method CLOCKREADY_OUT clockready;
```

The BSV types of all the method's arguments and its result (if any) must all be in the `Bits` typeclass.

Any of the port names may have an attribute attached to them. The allowable attributes are `reg`, `const`, `unused`, and `inhigh`. The attributes are translated into port descriptions. Not all port attributes are allowed on all ports.

For the output ports, the ready port and the method return value, the properties `reg` and `const` are allowed. The `reg` attribute specifies that the value is coming directly from a register with no intermediate logic. The `const` attribute indicates that the value is hardwired to a constant value.

For the input ports, the input arguments and the enable port, `reg` and `unused` are allowed. In this context `reg` specifies that the value is immediately written to a register without intermediate logic. The attribute `unused` indicates that the port is not used inside the module; its value is ignored.

Additionally, for the method enable, there is the `inhigh` property, which indicates that the method is `always_enabled`, as described in Section 13.2.2. Inside the module, the value of the enable is assumed to be 1 and, as a result, the port doesn't exist. The user still gives a name for the port as a placeholder. Note that only `Action` or `ActionValue` methods can have an enable signal.

The following code fragment shows an attribute on a method enable:

```
method load(flopA, flopB) enable((*inhigh*) EN);
```

The output ports may be shared across methods (and ready signals).

15.3 Port statement

The `port` statement declares an input port, which is not part of a method, along with the value to be passed to the port. While parameters must be compile-time constants, ports can be dynamic.

The `port` statements are analogous to arguments to a BSV module, but are rarely needed, since BSV style is to interact and pass arguments through methods.

portBVIStmt	::=	`port` *identifier* [`clocked_by` `(` *clockId* `)`] [`reset_by` `(` *resetId* `)`] `=` *expression* `;`

The defining operator `<-` or `=` may be used.

The value of *expression* is supplied to the Verilog port named *identifier*. The type of *expression* must be in the `Bits` typeclass. The *expression* may be dynamic (e.g. the `_read` method of a register instantiated elsewhere in the module body), which differentiates it from a parameter statement. The Bluespec compiler cannot check that the import has specified the same size as declared in the Verilog module. If the width of the value is not the same as that expected by the Verilog module, Verilog will truncate or zero-extend the value to fit.

Example - Setting port widths to a specific width:

```
// Tie off the test ports
   port TM = 1'b0  ;  // This ties off the port TM to a 1 bit wide 0
   Bit#(w) z = 0;
   port TD = z     ;  // This ties off the port TD to w bit wide 0
```

The `clocked_by` clause is used to specify the clock domain that the port is associated with, named by *clockId*. Any clock in the domain may be used. The values `no_clock` and `default_clock`, as described in Section 15.5, may be used. If the clause is omitted, the associated clock is the default clock.

Example - BVI import statement including port statements

```
port BUS_ID clocked_by (clk2) = busId ;
```

The `reset_by` clause is used to specify the reset the port is associated with, named by *resetId*. Any reset in the domain may be used. The values `no_reset` and `default_reset`, as described in Section 15.8 may be used. If the clause is omitted, the associated reset is the default reset.

15.4 Input clock statement

The `input_clock` statement specifies how an incoming clock to a module is connected. Typically, there are two ports, the oscillator and the gate, though the connection may use fewer ports.

inputClockBVIStmt	::=	`input_clock` [*identifier*] `(` [*portsDef*] `)` `=` *expression* `;`
portsDef	::=	*portId* [`,` [*attributeInstances*] *portId*]
portId	::=	*identifier*

The defining operator `=` or `<-` may be used.

The *identifier* is the clock name which may be used elsewhere in the import to associate the clock with resets and methods via a `clocked_by` clause, as described in Sections 15.7 and 15.2. The *portsDef* statement describes the ports that define the clock. The clock value which is being connected is given by *expression*.

If the *expression* is an identifier being assigned with `=`, and the user wishes this to be the name of the clock, then the *identifier* of the clock can be omitted and the *expression* will be assumed to be the name. The clock name can be omitted in other circumstances, but then no name is associated with the clock. An unamed clock cannot be referred to elsewhere, such as in a method or reset or other statement. Example:


```
input_clock (OSC, GATE) = clk;
```

is equivalent to:

```
input_clock clk (OSC, GATE) = clk;
```

The user may leave off the gate (one port) or the gate and the oscillator (no ports). It is the designer's responsibility to ensure that not connecting ports 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. In this case, the *portsDef* would be omitted. Example of an input clock without any connection to the Verilog ports:

```
input_clock ddClk() = dClk;
```

If the clock port is specified and the gate port is to be unconnected, an attribute, either `unused` or `inhigh`, describing the gate port should be specified. The attribute `unused` indicates that the submodule doesn't care what the unconnected gate is, while `inhigh` specifies the gate is assumed in the module to be logical 1. It is an error if a clock with a gate that is not logical 1 is connected to an input clock with an `inhigh` attribute. The default when a gate port is not specified is `inhigh`, though it is recommended style that the designer specify the attribute explicitly.

To add an attribute, the usual attribute syntax, `(* attribute_name *)` immediately preceeding the object of the attribute, is used. For example, 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. So the second *portId*, for the gate port, would be marked unused.

```
input_clock ddClk (OSC, (*unused*) UNUSED) = dClk;
```

The options for specifying the clock ports in the *portsDef* clause are:

```
( )                    // there are no Verilog ports
(OSC, GATE)            // both an oscillator port and a gate port are specified
(OSC, (*unused*)GATE)  // there is no gate port and it's unused
(OSC, (*inhigh*)GATE)  // there is no gate port and it's required to be logical 1
(OSC)                  // same as (OSC, (*inhigh*) GATE)
```

In an `input_clock` statement, it is an error if both the port names and the input clock name are omitted, as the clock is then unusable.

15.5 Default clock

In BSV, each module has an implicit clock (the *current clock*) which is used to clock all instantiated submodules unless otherwise specified with a `clocked_by` clause. Other clocks to submodules must be explicitly passed as input arguments.

Every BVI import module must declare which input clock (if any) is the default clock. This default clock is the implicit clock provided by the parent module, or explicitly given via a `clocked_by` clause. The default clock is also the clock associated with methods and resets in the BVI import when no `clocked_by` clause is specified.

The simplest definition for the default clock is:

defaultClockBVIStmt ::= `default_clock` *identifier* `;`

where the *identifier* specifies the name of an input clock which is designated as the default clock.

The default clock may be unused or not connected to any ports, but it must still be declared. Example:

```
default_clock no_clock;
```

This statement indicates the implicit clock from the parent module is ignored (and not connected). Consequently, the default clock for methods and resets becomes `no_clock`, meaning there is no associated clock.

To save typing, you can merge the `default_clock` and `input_clock` statements into a single line:

defaultClockBVIStmt ::= `default_clock` [*identifier*] [`(` *portsDef* `)`] [`=` *expression*] `;`

The defining operator `=` or `<-` may be used.

This is precisely equivalent to defining an input clock and then declaring that clock to be the default clock. Example:

```
default_clock clk_src (OSC, GATE) = sClkIn;
```

is equivalent to:

```
input_clock clk_src (OSC, GATE) = sClkIn;
default_clock clk_src;
```

If omitted, the `=` *expression* in the `default_clock` statement defaults to `<- exposeCurrentClock`. Example:

```
default_clock xclk (OSC, GATE);
```

is equivalent to:

```
default_clock xclk (OSC, GATE) <- exposeCurrentClock;
```

If the portnames are excluded, the names default to `CLK, CLK_GATE`. Example:

```
default_clock xclk = clk;
```

is equivalent to:

```
default_clock xclk (CLK, CLK_GATE) = clk;
```

Alternately, if the *expression* is an identifier being assigned with `=`, and the user wishes this to be the name of the default clock, then he can leave off the name of the default clock and *expression* will be assumed to be the name. Example:

```
default_clock (OSC, GATE) = clk;
```

is equivalent to:

```
default_clock clk (OSC, GATE) = clk;
```

If an expression is provided, both the ports and the name cannot be omitted.

However, 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.

15.6 Output clock

The `output_clock` statement gives the port connections for a clock provided in the module's interface.

outputClockBVIStmt ::= `output_clock` *identifier* [`(` *portsDef* `)`]`;`

The *identifier* defines the name of the output clock, which must match a clock declared in the module's interface. Example:

```
interface ClockGenIfc;
  interface Clock gen_clk;
endinterface

import "BVI" ClockGen =
module vMkAbsoluteClock #( Integer start,
                      Integer period
                     ) ( ClockGenIfc );
   ...
   output_clock gen_clk(CLK_OUT);
endmodule
```

It is an error for the same *identifier* to be declared by more than one `output_clock` statement.

15.7 Input reset

The `input_reset` statement defines how an incoming reset to the module is connected. Typically there is one port. BSV assumes that the reset is inverted (the reset is asserted with the value 0).

inputResetBVIStmt ::= `input_reset` [*identifier*] [`(` *portId* `)`] [`clocked_by` `(` *clockId* `)`]
`=` *expression* `;`

portId ::= *identifier*

clockId ::= *identifier*

where the `=` may be replaced by `<-`.

The reset given by *expression* is to be connected to the Verilog port specified by *portId*. The *identifier* is the name of the reset and may be used elsewhere in the import to associate the reset with methods via a `reset_by` clause.

The `clocked_by` clause is used to specify the clock domain that the reset is associated with, named by *clockId*. Any clock in the domain may be used. If the clause is omitted, the associated clock is the default clock. Example:

```
input_reset rst(sRST_N) = sRstIn;
```

is equivalent to:

```
input_reset rst(sRST_N) clocked_by(clk) = sRstIn;
```

where `clk` is the identifier named in the `default_clock` statement.

If the user doesn't care which clock domain is associated with the reset, `no_clock` may be used. In this case the compiler will not check that the connected reset is associated with the correct domain. Example


```
input_reset rst(sRST_N) clocked_by(no_clock) = sRstIn;
```

If the *expression* is an identifier being assigned with =, and the user wishes this to be the name of the reset, then he can leave off the *identifier* of the reset and the *expression* will be assumed to be the name. The reset name can be left off in other circumstances, but then no name is associated with the reset. An unamed reset cannot be referred to elsewhere, such as in a method or other statement.

In the cases where a parent module needs to associate a reset with methods, but the reset is not used internally, the statement may contain a name, but not specify a port. In this case, there is no port expected in the Verilog module. Example:

```
input_reset rst() clocked_by (clk_src) = sRstIn ;
```

Example of a BVI import statement containing an `input_reset` statement:

```
import "BVI" SyncReset =
module vSyncReset#(Integer stages ) ( Reset rstIn, ResetGenIfc rstOut ) ;
   ...
   // we don't care what the clock is of the input reset
   input_reset rst(IN_RST_N) clocked_by (no_clock) = rstIn ;
   ...
endmodule
```

15.8 Default reset

In BSV, when you define a module, it has an implicit reset (the *current reset*) which is used to reset all instantiated submodules (unless otherwise specifed via a `reset_by` clause). Other resets to submodules must be explicitly passed as input arguments.

Every BVI import module must declare which reset, if any, is the default reset. The default reset is the implicit reset provided by the parent module (or explicitly given with a `reset_by`). The default reset is also the reset associated with methods in the BVI import when no `reset_by` clause is specified.

The simplest definition for the default reset is:

defaultResetBVIStmt ::= `default_reset` *identifier* `;`

where *identifier* specifies the name of an input reset which is designated as the default reset.

The reset may be unused or not connected to a port, but it must still be declared. Example:

```
default_reset no_reset;
```

The keyword `default_reset` may be omitted when declaring an unused reset. The above statement can thus be written as:

```
no_reset;       // the default_reset keyword can be omitted
```

This statement declares that the implicit reset from the parent module is ignored (and not connected). In this case, the default reset for methods becomes `no_reset`, meaning there is no associated reset.

To save typing, you can merge the `default_reset` and `input_reset` statements into a single line:

defaultResetBVIStmt ::= `default_reset` [*identifier*] [`(` *portId* `)`] [`clocked_by` `(` *clockId* `)`]
[`=` *expression*] `;`

The defining operator = or <- may be used.

This is precisely equivalent to defining an input reset and then declaring that reset to be the default. Example:

```
default_reset rst (RST_N) clocked_by (clk) = sRstIn;
```

is equivalent to:

```
input_reset rst (RST_N) clocked_by (clk) = sRstIn;
default_reset rst;
```

If omitted, = *expression* in the `default_reset` statement defaults to `<- exposeCurrentReset`. Example:

```
default_reset rst (RST_N);
```

is equivalent to

```
default_reset rst (RST_N) <- exposeCurrentReset;
```

The `clocked_by` clause is optional; if omitted, the reset is clocked by the default clock. Example:

```
default_reset rst (sRST_N) = sRstIn;
```

is equivalent to

```
default_reset rst (sRST_N) clocked_by(clk) = sRstIn;
```

where `clk` is the `default_clock`.

If `no_clock` is specified, the reset is not associated with any clock. Example:

```
input_reset rst (sRST_N) clocked_by(no_clock) = sRstIn;
```

If the *portId* is excluded, the reset port name defaults to `RST_N`. Example:

```
default_reset rstIn = rst;
```

is equivalent to:

```
default_reset rstIn (RST_N) = rst;
```

Alternatively, if the *expression* is an identifier being assigned with `=`, and the user wishes this to be the name of the default reset, then he can leave off the name of the default reset and *expression* will be assumed to be the name. Example:

```
default_reset (rstIn) = rst;
```

is equivalent to:

```
default_reset rst (rstIn) = rst;
```

Both the ports and the name cannot be omitted.

However, omitting the entire statement is equivalent to:

```
default_reset (RST_N) <- exposeCurrentReset;
```

specifying that the current reset is to be associated with all methods which do not specify otherwise.

15.9 Output reset

The `output_reset` statement gives the port connections for a reset provided in the module's interface.

outputResetBVIStmt ::= `output_reset` *identifier* [`(` *portId* `)`] [`clocked_by` `(` *clockId* `)`]`;`

The *identifier* defines the name of the output reset, which must match a reset declared in the module's interface. Example:

```
interface ResetGenIfc;
  interface Reset gen_rst;
endinterface

import "BVI" SyncReset =
module vSyncReset#(Integer stages ) ( Reset rstIn, ResetGenIfc rstOut ) ;
   ...
   output_reset gen_rst(OUT_RST_N) clocked_by(clk) ;
endmodule
```

It is an error for the same *identifier* to be declared by more than one `output_reset` statement.

15.10 Ancestor, same family

There are two statements for specifying the relationship between clocks: `ancestor` and `same_family`.

ancestorBVIStmt ::= `ancestor` `(` *clockId* `,` *clockId* `)` `;`

This statement indicates that the second named clock is an `ancestor` of the first named clock. To say that `clock1` is an `ancestor` of `clock2`, means that `clock2` is a gated version of `clock1`. This is written as:

```
ancestor (clock2, clock1);
```

For clocks which do not have an ancestor relationship, but do share a common ancestor, we have:

sameFamilyBVIStmt ::= `same_family` `(` *clockId* `,` *clockId* `)` `;`

This statement indicates that the clocks specified by the *clockIds* are in the same family (same clock domain). When two clocks are in the same family, they have the same oscillator with a different gate. To be in the same family, one does not have to be a gated version of the other, instead they may be gated versions of a common ancestor. Note that `ancestor` implies `same_family`, which then need not be explicitly stated. For example, a module which gates an input clock:

```
input_clock clk_in(CLK_IN, CLK_GATE_IN) = clk_in ;
output_clock new_clk(CLK_OUT, CLK_GATE_OUT);
ancestor(new_clk, clk_in);
```

15.11 Schedule

scheduleBVIStmt ::= `schedule` `(` *identifier* { `,` *identifier* } `)` *operatorId*
`(` *identifier* { `,` *identifier* } `)``;`

operatorId ::= `CF`
| `SB`

```
                    |   SBR
                    |   C
```

The `schedule` statement specifies the scheduling constraints between methods in an imported module. The operators relate two sets of methods; the specified relation is understood to hold for each pair of an element of the first set and an element of the second set. The order of the methods in the lists is unimportant and the parentheses may be omitted if there is only one name in the 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 two relationships for the same pair of methods. 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`. The compiler generates a warning if an annotation between a method pair is missing. 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          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 ) ;

    schedule (enq, notFull) CF (deq, first, notEmpty) ;
    schedule (first, notEmpty) CF (first, notEmpty) ;
    // CF: conflict free - methods in the first list can be scheduled
    // in any order or any number of times,  with the methods in the
    // second list - there is no conflict between the methods.
    schedule first SB deq ;
    schedule (notEmpty) SB (deq) ;
    schedule (notFull) SB (enq) ;
    // SB indicates the order in which the methods must be scheduled
    // the methods in the first list must occur (be scheduled) before
    // the methods in the second list
    // SB allows these methods to be called from one rule but the
    // SBR relationship does not.
    schedule (enq) C (enq) ;
    schedule (deq) C (deq) ;
    schedule (notFull) CF (notFull) ;
    // C: conflicts - methods in the first list conflict with the
    // methods in the second - they cannot be called in the same clock cycle.
    // if a method conflicts with itself, (enq,deq, and notFull), it
    // cannot be called more than once in a clock cycle
 endmodule
```


15.12 Path

The `path` statement indicates that there is a combinational path from the first port to the second port.

pathBVIStmt ::= `path` `(` *portId* `,` *portId* `)` `;`

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 from a value or `ActionValue` method's input parameters to its result, so this need not be specified explicitly.

The paths defined by the `path` statement are used in scheduling. A path may impact rule urgency by implying an order in how the methods are scheduled. The path is also used in checking for combinational cycles in a design. The compiler will report an error if it detects a cycle in a design. In the following example, there is a path declared between `WSET` and `WHAS`, as shown in figure 2.

```
import "BVI" RWire0 =
   module vMkRWire0 (VRWire0);
      ...
      method wset() enable(WSET) ;
      method WHAS whas ;
      schedule whas CF whas ;
      schedule wset SB whas ;
      path (WSET, WHAS) ;
   endmodule: vMkRWire0
```

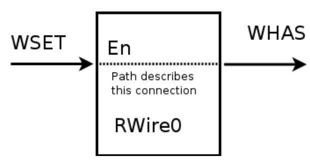

Figure 2: Path in the RWire0 Verilog module between WSET and WHAS ports

15.13 Inout

The following statements describe how to pass an `inout` port from a wrapped Verilog module through a BSV module. These ports are represented in BSV by the type `Inout`. There are two ways that an `Inout` can appear in BSV modules: as an argument to the module or as a subinterface of the interface provided by the module. There are, therefore, two ways to declare an `Inout` port in a BVI import: the statement `inout` declares an argument of the current module; and the statement `ifc_inout` declares a subinterface of the provided interface.

inoutBVIStmt ::= `inout` *portId* [`clocked_by` `(` *clockId* `)`]
[`reset_by` `(` *resetId* `)`] `=` *expression* `;`

The value of *portId* is the Verilog name of the `inout` port and *expression* is the name of an argument from the module.

inoutBVIStmt ::= `ifc_inout` *identifier* `(`*inoutId* `)` [`clocked_by` `(` *clockId* `)`]
[`reset_by` `(` *resetId* `)`] `;`

Here, the *identifier* is the name of the subinterface of the provided interface and *portId* is, again, the Verilog name of the `inout` port.

The clock and reset associated with the `Inout` are assumed to be the default clock and default reset unless explicitly specified.

Example:

```
interface Q;
   interface Inout#(Bit#(13)) q_inout;
   interface Clock c_clock;
endinterface

import "BVI" Foo =
module mkFoo#(Bool b)(Inout#(int) x, Q ifc);
   default_clock ();
   no_reset;

   inout iport = x;

   ifc_inout q_inout(qport);
   output_clock c_clock(clockport);
endmodule
```

The wrapped Verilog module is:

```
module Foo (iport, clockport, qport);
   input cccport;
   inout [31:0] iport;
   inout [12:0] qport;
   ...
endmodule
```

16 Embedding C in a BSV Design

This section describes how to declare a BSV function that is provided as a C function. This is used when there are existing C functions which the designer would like to include in a BSV module. Using the *importBDPI* syntax, the user can specify that the implementation of a BSV function is provided as a C function.

externCImport	::=	`import "BDPI"` [*identifier* `=`] `function` *type* *identifier* `(` [*CFuncArgs*] `)` [*provisos*] `;`
CFuncArgs	::=	*CFuncArg* { `,` *CFuncArg* }
CFuncArg	::=	*type* [*identifier*]

This defines a function *identifier* in the BSV source code which is implemented by a C function of the same name. A different link name (C name) can be specified immediately after the `"BDPI"`, using an optional [*identifier* =]. The link name is not bound by BSV case-restrictions on identifiers and may start with a capital letter.

Example of an import statement where the C name matches the BSV name:

```
// the C function and the BSV function are both named checksum
import "BDPI" function Bit#(32) checksum (Bit#(n), Bit#(32));
```


Example of an import statement where the C name does not match the BSV name:

```
// the C function name is checksum
// the BSV function name is checksum_raw
import "BDPI" checksum = function Bit#(32) checksum_raw (Bit#(n), Bit#(32));
```

The first *type* specifies the return type of the function. The optional *CFuncArgs* specify the arguments of the function, along with an optional identifier to name the arguments.

For instance, in the above checksum example, you might want to name the arguments to indicate that the first argument is the input value and the second argument is the size of the input value.

```
import "BDPI" function Bit#(32) checksum (Bit#(n) input_val, Bit#(32) input_size);
```

16.1 Argument Types

The types for the arguments and return value are BSV types. The following table shows the correlation from BSV types to C types.

BSV Type	C Type
`String`	`char*`
`Bit#(0) - Bit#(8)`	`unsigned char`
`Bit#(9) - Bit#(32)`	`unsigned int`
`Bit#(33) - Bit#(64)`	`unsigned long long`
`Bit#(65) -`	`unsigned int*`
`Bit#(n)`	`unsigned int*`

The *importBDPI* syntax provides the ability to import simple C functions that the user may already have. A C function with an argument of type `char` or `unsigned char` should be imported as a BSV function with an argument of type `Bit#(8)`. For `int` or `unsigned int`, use `Bit#(32)`. For `long long` or `unsigned long long`, use `Bit#(64)`. While BSV creates unsigned values, they can be passed to a C function which will treat the value as signed. This can be reflected in BSV with `Int#(8)`, `Int#(32)`, `Int#(64)`, etc.

The user may also import new C functions written to match a given BSV function type. For instance, a function on bit-vectors of size 17 (that is, `Bit#(17)`) would expect to pass this value as the C type `unsigned int` and the C function should be aware that only the first 17 bits of the value are valid data.

Wide data Bit vectors of size 65 or greater are passed by reference, as type `unsigned int*`. This is a pointer to an array of 32-bit words, where bit 0 of the BSV vector is bit 0 of the first word in the array, and bit 32 of the BSV vector is bit 0 of the second word, etc. Note that we only pass the pointer; no size value is passed to the C function. This is because the size is fixed and the C function could have the size hardcoded in it. If the function needs the size as an additional parameter, then either a C or BSV wrapper is needed. See the examples below.

Polymorphic data As the above table shows, bit vectors of variable size are passed by reference, as type `unsigned int*`. As with wide data, this is a pointer to an array of 32-bit words, where bit 0 of the BSV vector is bit 0 of the first word in the array, and bit 32 of the BSV vector is bit 0 of the second word, etc. No size value is passed to the C function, because the import takes no stance on how the size should be communicated. The user will need to handle the communication of the size, typically by adding an additional argument to the import function and using a BSV wrapper to pass the size via that argument, as follows:

```
// This function computes a checksum for any size bit-vector
// The second argument is the size of the input bit-vector
import "BDPI" checksum = function Bit#(32) checksum_raw (Bit#(n), Bit#(32));

// This wrapper handles the passing of the size
function Bit#(32) checksum (Bit#(n) vec);
   return checksum_raw(vec, fromInteger(valueOf(n)));
endfunction
```

16.2 Return types

Imported functions can be value functions, `Action` functions, or `ActionValue` functions. The acceptable return types are the same as the acceptable argument types, except that `String` is not permitted as a return type.

Imported functions with return values correlate to C functions with return values, except in the cases of wide and polymorphic data. In those cases, where the BSV type correlates to `unsigned int*`, the simulator will allocate space for the return result and pass a pointer to this memory to the C function. The C function will not be responsible for allocating memory. When the C function finishes execution, the simulator copies the result in that memory to the simulator state and frees the memory. By convention, this special argument is the first argument to the C function.

For example, the following BSV import:

```
import "BDPI" function Bit#(32) f (Bit#(8));
```

would connect to the following C function:

```
unsigned int f (unsigned char x);
```

While the following BSV import with wide data:

```
import "BDPI" function Bit#(128) g (Bit#(8));
```

would connect to the following C function:

```
void g (unsigned int* resultptr, unsigned char x);
```

16.3 Implicit pack/unpack

So far we have only mentioned `Bit` and `String` types for arguments and return values. Other types are allowed as arguments and return values, as long as they can be packed into a bit-vector. These types include `Int`, `UInt`, `Bool`, and `Maybe`, all of which have an instance in the Bits class.

For example, this is a valid import:

```
import "BDPI" function Bool my_and (Bool, Bool);
```

Since a `Bool` packs to a `Bit#(1)`, it would connect to a C function such as the following:

```
unsigned char
my_and (unsigned char x, unsigned char y);
```


In this next example, we have two C functions, `signedGT` and `unsignedGT`, both of which implement a greater-than function, returning a `Bool` indicating whether `x` is greater than `y`.

```
import "BDPI" function Bool signedGT (Int#(32) x, Int#(32) y);
import "BDPI" function Bool unsignedGT (UInt#(32) x, UInt#(32) y);
```

Because the function `signedGT` assumes that the MSB is a sign bit, we use the type-system to make sure that we only call that function on signed values by specifying that the function only works on `Int#(32)`. Similarly, we can enforce that `unsignedGT` is only called on unsigned values, by requiring its arguments to be of type `UInt#(32)`.

The C functions would be:

```
unsigned char signedGT (unsigned int x, unsigned int y);
unsigned char unsignedGT (unsigned int x, unsigned int y);
```

In both cases, the packed value is of type `Bit#(32)`, and so the C function is expected to take the its arguments as `unsigned int`. The difference is that the `signedGT` function will then treat the values as signed values while the `unsignedGT` function will treat them as unsigned values. Both functions return a `Bool`, which means the C return type is `unsigned char`.

Argument and return types to imported functions can also be structs, enums, and tagged unions. The C function will receive the data in bit form and must return values in bit form.

16.4 Other examples

Shared resources In some situations, several imported functions may share access to a resource, such as memory or the file system. If these functions wish to share file handles, pointers, or other cookies between each other, they will have to pass the data as a bit-vector, such as `unsigned int/Bit#(32)`.

When to use Action components If an imported function has a side effect or if it matters how many times or in what order the function is called (relative to other calls), then the imported function should have an `Action` component in its BSV type. That is, the functions should have a return type of `Action` or `ActionValue`.

Removing indirection for polymorphism within a range A polymorphic type will always become `unsigned int*` in the C, even if there is a numeric proviso which restricts the size. Consider the following import:

```
import "BDPI" function Bit#(n) f(Bit#(n), Bit#(8)) provisos (Add#(n,j,32));
```

This is a polymorphic vector, so the conversion rules indicate that it should appear as `unsigned int*` in the C. However, the proviso indicates that the value of n can never be greater than 32. To make the import be a specific size and not a pointer, you could use a wrapper, as in the example below.

```
import "BDPI" f = function Bit#(32) f_aux(Bit#(32), Bit#(8));

function Bit#(n) f (Bit#(n) x) provisos (Add#(n,j,32));
   return f_aux(extend(x), fromInteger(valueOf(n)));
endfunction
```

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
BVI
C
CF
SB
SBR
action                              endaction
actionvalue                         endactionvalue
ancestor
begin
bit
case                                endcase
clocked_by
default
default_clock
default_reset
dependencies
deriving
determines
else
enable
end
enum
export
for
function                            endfunction
if
ifc_inout
import
inout
input_clock
input_reset
instance                            endinstance
interface                           endinterface
let
match
matches
method                              endmethod
module                              endmodule
numeric
output_clock
output_reset
package                             endpackage
parameter
path
port
provisos
reset_by
```

`return`	
`rule`	`endrule`
`rules`	`endrules`
`same_family`	
`schedule`	
`struct`	
`tagged`	
`type`	
`typeclass`	`endtypeclass`
`typedef`	
`union`	
`valueOf`	
`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
always
always_comb
always_ff
always_latch
and
assert
assert_strobe
assign
assume
automatic
before
begin       end
bind
bins
binsof
bit
break
buf
bufif0
bufif1
byte
case        endcase
casex
casez
cell
chandle
class       endclass
clocking    endclocking
cmos
config      endconfig
const
constraint
context
continue
cover
covergroup  endgroup
coverpoint
cross
deassign
default
defparam
design
disable
dist
do
edge
else
enum
event
expect
export
extends
extern
final
first_match
for
force
foreach
forever
fork
forkjoin
function    endfunction
generate    endgenerate
genvar
highz0
highz1
if
iff
ifnone
ignore_bins
illegal_bins
import
incdir
include
initial
inout
input
inside
instance
int
integer
interface   endinterface
intersect
join
join_any
join_none
large
liblist
library
local
localparam
logic
longint
macromodule
matches
medium
modport
module      endmodule
nand
negedge
new
nmos
nor
noshowcancelled
not
notif0
notif1
null
or
output
package     endpackage
packed
parameter
pmos
posedge
primitive   endprimitive
priority
program     endprogram
property    endproperty
protected
pull0
pull1
pulldown
pullup
pulsestyle_onevent
pulsestyle_ondetect
pure
rand
randc
randcase
randsequence
rcmos
real
realtime
ref
reg
release
repeat
return
rnmos
rpmos
rtran
rtranif0
rtranif1
scalared
sequence    endsequence
shortint
shortreal
showcancelled
```

signed	
small	
solve	
specify	endspecify
specparam	
static	
string	
strong0	
strong1	
struct	
super	
supply0	
supply1	
table	endtable
tagged	
task	endtask
this	
throughout	
time	
timeprecision	
timeunit	
tran	
tranif0	
tranif1	
tri	
tri0	
tri1	
triand	
trior	
trireg	
type	
typedef	
union	
unique	
unsigned	
use	
var	
vectored	
virtual	
void	
wait	
wait_order	
wand	
weak0	
weak1	
while	
wildcard	
wire	
with	
within	
wor	
xnor	
xor	

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.

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.
`RealLiteral`	Types which can be created from real 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.

```
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.

```
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);
    function Bool   inLiteralRange(data_t target, Integer x);
endtypeclass
```

The fromInteger function converts an Integer into an element of datatype data_t. Whenever you write an integer literal in BSV(such as "0" or "1"), there is an implied fromInteger applied to it, which turns the literal into 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 values from literals just as for these predefined types.

The typeclass also provides a function inLiteralRange that takes an argument of the target type and an Integer and returns a Bool that indicates whether the Integer argument is in the legal range of the target type. For example, assuming x has type Bit#(4), inLiteralRange(x, 15) would return True, but inLiteralRange(x,22) would return False.

Literal Functions		
fromInteger	Converts an element x of datatype Integer into an element of data type data_t	
	`function data_t fromInteger(Integer x);`	

inLiteralRange	Tests whether an element x of datatype Integer is in the legal range of data type data_t
	`function Bool inLiteralRange(data_t target, Integer x);`

B.1.4 RealLiteral

RealLiteral defines the class of types which can be created from real literals.

```
typeclass RealLiteral #(type data_t);
    function data_t fromReal(Real x);
endtypeclass
```

The fromReal function converts a Real into an element of datatype data_t. Whenever you write a real literal in BSV(such as "3.14"), there is an implied fromReal applied to it, which turns the real into the specified type. By defining an instance of RealLiteral for a datatype, you can create values from reals for any type.

RealLiteral Functions	
fromReal	Converts an element x of datatype Real into an element of data type data_t
	`function data_t fromReal(Real x);`

B.1.5 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);
    function data_t \/ (data_t x, data_t y);
    function data_t \% (data_t x, data_t y);
    function data_t abs (data_t x);
    function data_t signum (data_t x);
    function data_t \** (data_t x, data_t y);
    function data_t exp_e (data_t x);
    function data_t log (data_t x);
    function data_t logb (data_t b, data_t x);
    function data_t log2 (data_t x);
    function data_t log10 (data_t x);
endtypeclass
```

The `Arith` functions provide arithmetic operations. For the arithmetic symbols, when defining an instance of the `Arith` typeclasss, the escaped operator names must be used as shown in the tables below. The `negate` name may be used instead of the operator for negation. If using or referring to these 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);`
`/`	Element `x` is divided by `y`. The definition depends on the type - many types truncate the remainder . Note: may not be synthesizable with downstream tools.
	`function data_t \/ (data_t x, data_t y);`
`%`	Returns the remainder of x/y. Obeys the identity $((x/y) * y) + (x\%y) = x$.
	`function data_t \% (data_t x, data_t y);`

Note: Division by 0 is undefined. Both $x/0$ and $x\%0$ will generate errors at compile-time and run-time for most instances.

`abs`	Returns the absolute value of `x`.
	`function data_t abs (data_t x);`
`signum`	Returns a unit value with the same sign as `x`, such that `abs(x)*signum(x) = 1`. `signum(12)` returns `1` and `signum(-12)` returns `-1`.
	`function data_t signum (data_t x);`

`**`	The element `x` is raised to the `y` power (`x**y` $= x^y$).
	`function data_t \** (data_t x, data_t y);`

`log2`	Returns the base 2 logarithm of `x` ($\log_2 x$).
	`function data_t log2(data_t x) ;`

`exp_e`	`e` is raised to the power of `x` (e^x).
	`function data_t exp_e (data_t x);`

`log`	Returns the base `e` logarithm of `x` ($\log_e x$).
	`function data_t log (data_t x);`

`logb`	Returns the base `b` logarithm of `x` ($\log_b x$).
	`function data_t logb (data_t b, data_t x);`

`log10`	Returns the base 10 logarithm of x ($\log_{10} x$).
	`function data_t log10(data_t x) ;`

B.1.6 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.1.7 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.1.8 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.

<table>
<tr><th colspan="2"><code>Bitwise</code> Functions</th></tr>
<tr><td><code>&</code></td><td>Performs an and operation on each bit in <code>x1</code> and <code>x2</code> to calculate the result.
<code>function data_t \& (data_t x1, data_t x2);</code></td></tr>
<tr><td><code>|</code></td><td>Performs an or operation on each bit in <code>x1</code> and <code>x2</code> to calculate the result.
<code>function data_t \| (data_t x1, data_t x2);</code></td></tr>
<tr><td><code>^</code></td><td>Performs an exclusive or operation on each bit in <code>x1</code> and <code>x2</code> to calculate the result.
<code>function data_t \^ (data_t x1, data_t x2);</code></td></tr>
<tr><td><code>~^</code>
<code>^~</code></td><td>Performs an exclusive nor operation on each bit in <code>x1</code> and <code>x2</code> to calculate the result.
<code>function data_t \~^ (data_t x1, data_t x2);</code>
<code>function data_t \^~ (data_t x1, data_t x2);</code></td></tr>
<tr><td><code>~</code>
<code>invert</code></td><td>Performs a unary negation operation on each bit in <code>x1</code>. When using this function, the corresponding Verilog operator, <code>~</code>, may be used.
<code>function data_t invert (data_t x1);</code></td></tr>
<tr><td><code><<</code></td><td>Performs a left shift operation of <code>x1</code> by the number of bit positions given by <code>x2</code>.
<code>function data_t \<< (data_t x1, Nat x2);</code></td></tr>
<tr><td><code>>></code></td><td>Performs a right shift operation of <code>x1</code> by the number of bit positions given by <code>x2</code>.
<code>function data_t \>> (data_t x1, Nat x2);</code></td></tr>
</table>

B.1.9 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 `&`.

<table>
<tr><th colspan="2"><code>BitReduction</code> Functions</th></tr>
<tr><td rowspan="2"><code>reduceAnd</code>

<code>&</code></td><td>Performs an and bit reduction operation between the elements of <code>d</code> to calculate the result.</td></tr>
<tr><td><code>function x#(1) reduceAnd (x#(n) d);</code></td></tr>
<tr><td rowspan="2"><code>reduceOr</code>

<code>|</code></td><td>Performs an or bit reduction operation between the elements of <code>d</code> to calculate the result.</td></tr>
<tr><td><code>function x#(1) reduceOr (x#(n) d);</code></td></tr>
<tr><td rowspan="2"><code>reduceXor</code>

<code>^</code></td><td>Performs an xor bit reduction operation between the elements of <code>d</code> to calculate the result.</td></tr>
<tr><td><code>function x#(1) reduceXor (x#(n) d);</code></td></tr>
<tr><td rowspan="2"><code>reduceNand</code>

<code>~&</code></td><td>Performs an nand bit reduction operation between the elements of <code>d</code> to calculate the result.</td></tr>
<tr><td><code>function x#(1) reduceNand (x#(n) d);</code></td></tr>
<tr><td rowspan="2"><code>reduceNor</code>

<code>~|</code></td><td>Performs an nor bit reduction operation between the elements of <code>d</code> to calculate the result.</td></tr>
<tr><td><code>function x#(1) reduceNor (x#(n) d);</code></td></tr>
<tr><td rowspan="2"><code>reduceXnor</code>

<code>~^</code>
<code>^~</code></td><td>Performs an xnor bit reduction operation between the elements of <code>d</code> to calculate the result.</td></tr>
<tr><td><code>function x#(1) reduceXnor (x#(n) d);</code></td></tr>
</table>

B.1.10 BitExtend

`BitExtend` defines types on which bit extension operations are defined.

```
typeclass BitExtend #(numeric type m, numeric type n, type x);  // n > m
    function x#(n) extend (x#(m) d);
    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 of one size and changes it to an input of another size, as described in the tables below. It is recommended that `extend` be used in place of `zeroExtend` or `signExtend`, as it will automatically perform the correct operation based on the data type of the argument.

`BitExtend` Functions	
`extend`	Performs either a zeroExtend or a signExtend as appropriate, depending on the data type of the argument (zeroExtend for an unsigned argument, signExtend for a signed argument).
	`function x#(n) extend (x#(m) d)` `   provisos (Add#(k, m, n));`

`zeroExtend`	Use of `extend` instead is recommended. 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, m, n));`

`signExtend`	Use of `extend` instead is recommended. Adds extra sign 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, m, n));`

`truncate`	Removes bits from the MSB of argument `d` of size `n` to make the datatype size `m`.
	`function x#(m) truncate (x#(n) d)` `   provisos (Add#(k, n, m));`

B.2 Data Types

Every variable and every expression in BSV has a *type*. Prelude defines the data types which are always available. An `instance` declaration defines a data type as belonging to a type class. 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 three exceptions; `bit`, `int`, and `real`, which are predefined for backwards compatibility.

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` and the result is rounded toward negative infinity. Division by 0 is undefined. `function Integer div(Integer x, Integer y);`

`mod`	Element `x` is divided by element `y` using the `div` function and the remainder is returned as an `Integer` value. `div` and `mod` satisfy the identity $(div(x,y)*y)+mod(x,y) == x$. Division by 0 is undefined. `function Integer mod(Integer x, Integer y);`

`quot`	Element `x` is divided by element `y` and the result is truncated (rounded towards 0). Division by 0 is undefined. `function Integer quot(Integer x, Integer y);`

`rem`	Element `x` is divided by element `y` using the `quot` function and the remainder is returned as an `Integer` value. `quot` and `rem` satisfy the identity $(quot(x,y) * y) + rem(x,y) == x$. Division by 0 is undefined. `function Integer rem(Integer x, Integer y);`

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 Real

The `Real` type is a data type used for real values and functions.

Real numbers are of the form:

real_number ::= [*sign*]*unsign_num*[`.`*unsign_num*] *exp* [*sign*]*unsign_num*
| [*sign*]*unsign_num* `.` *unsign_num*

sign ::= `+` | `-`

exp ::= `e` | `E`

unsign_num ::= *decimal_digit* { [`_`]*decimal_digit* }

decimal_digit ::= `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`

If there is a decimal point, there must be digits following the decimal point. An exponent can start with either an `E` or an `e`, followed by an optional sign (`+` or `-`), followed by digits. There cannot be an exponent or a sign without any digits. Any of the numeric components may include an underscore, but an underscore cannot be the first digit of the number.

Unlike Integer, Real numbers are of limited precision. They are represented as IEEE floating point numbers of 64 bit length, as defined by the IEEE standard.

Because `Real` is not part of the `Bits` typeclass, the `Real` type is used for static elaboration only; all values must be resolved at compile time.

There are many functions defined for `Real` types, provided in the `Real` package (Section C.4.1). To use these functions, the `Real` package must be imported.

Type Classes for Real										
	Bits	Eq	Literal	Real Literal	Arith	Ord	Bounded	Bitwise	Bit Reduction	Bit Extend
Real		√	√	√	√	√				

Real type aliases	
real	The SystemVerilog name `real` is an alias for `Real`
	`typedef Real real;`

There are two system tasks defined for the `Real` data type, used to convert between `Real` and IEEE standard 64-bit vector representation (`Bit#(64)`).

Real system tasks	
$realtobits	Converts from a Real to the IEEE 64-bit vector representation.
	`function Bit#(64) $realtobits (Real x) ;`

$bitstoreal	Converts from a 64-bit vector representation to a Real.
	`function Real $bitstoreal (Bit#(64) x) ;`

B.2.7 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.2.8 Fmt

The `Fmt` primitive type provides a representation of arguments to the `$display` family of system tasks (Section 12.8.1) that can be manipulated in BSV code. `Fmt` representations of data objects can be written hierarchically and applied to polymorphic types.

Objects of type `Fmt` can be supplied directly as arguments to system tasks in the `$display` family. An object of type `Fmt` is returned by the `$format` (Section 12.8.2) system task.

The `Fmt` type is part of the `Arith` class, but only the addition (`+`) operator is defined. All other `Arith` operators will produce an error message.

Type Classes for `Fmt`									
	Bits	Eq	Literal	Arith	Ord	Bounded	Bitwise	Bit Reduction	Bit Extend
Fmt			√	√					

B.2.9 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.2.10 Tuples

Tuples are predefined structures which group a small number of values together. The following pseudocode explains the structure of the tuples. You cannot define your own tuples, but must use the six predefined tuples, `Tuple2` through `Tuple7`. As shown, `Tuple2` groups two items together, `Tuple3` groups three items together, up through `Tuple7` which groups seven items together.

```
typedef struct{
    a tpl_1;
    b tpl_2;
} Tuple2 #(type a, type b) deriving (Bits, Eq, Bounded);
```



```
typedef struct{
    a tpl_1;
    b tpl_2;
    c tpl_3;
 } Tuple3 #(type a, type b, type c) deriving (Bits, Eq, Bounded);

typedef struct{
    a tpl_1;
    b tpl_2;
    c tpl_3;
    d tpl_4;
 } Tuple4 #(type a, type b, type c, type d) deriving (Bits, Eq, Bounded);

typedef struct{
    a tpl_1;
    b tpl_2;
    c tpl_3;
    d tpl_4;
    e tpl_5;
 } Tuple5 #(type a, type b, type c, type d, type e)
   deriving (Bits, Eq, Bounded);

typedef struct{
    a tpl_1;
    b tpl_2;
    c tpl_3;
    d tpl_4;
    e tpl_5;
    f tpl_6;
 } Tuple6 #(type a, type b, type c, type d, type e, type f)
   deriving (Bits, Eq, Bounded);

typedef struct{
    a tpl_1;
    b tpl_2;
    c tpl_3;
    d tpl_4;
    e tpl_5;
    f tpl_6;
    g tpl_7;
 } Tuple7 #(type a, type b, type c, type d, type e, type f, type g)
   deriving (Bits, Eq, Bounded);
```

Type Classes for `Tuples`

	Bits	Eq	Literal	Arith	Ord	Bounded	Bitwise	Bit Reduction	Bit Extend
TupleN	$\surd$	$\surd$			$\surd$	$\surd$			

Tuples cannot be manipulated like normal structures; you cannot create values of and select fields from tuples as you would a normal structure. Values of these types can be created only by applying a predefined family of constructor functions.

Tuple Constructor Functions	
`tuple2 (e1, e2)`	Creates a variable of type Tuple2 with component values e1 and e2.
`tuple3 (e1, e2, e3)`	Creates a variable of type Tuple3 with values e1, e2, and e3.
`tuple4 (e1, e2, e3, e4)`	Creates a variable of type Tuple4 with component values e1, e2, e3, and e4.
`tuple5 (e1, e2, e3, e4, e5)`	Creates a variable of type Tuple5 with component values e1, e2, e3, e4, and e5.
`tuple6 (e1, e2, e3, e4, e5, e6)`	Creates a variable of type Tuple6 with component values e1, e2, e3, e4, e5, and e6.
`tuple7 (e1, e2, e3, e4, e5, e6, e7)`	Creates a variable of type Tuple7 with component values e1, e2, e3, e4, e5, e6, and e7.

Fields of these types can be extracted only by applying a predefined family of selector functions.

Tuple Extract Functions	
`tpl_1 (x)`	Extracts the first field of x from a Tuple2 to Tuple7.
`tpl_2 (x)`	Extracts the second field of x from a Tuple2 to Tuple7.
`tpl_3 (x)`	Extracts the third field of x from a Tuple3 to Tuple7.
`tpl_4 (x)`	Extracts the fourth field of x from a Tuple4 to Tuple7.
`tpl_5 (x)`	Extracts the fifth field of x from a Tuple5, Tuple 6, or Tuple7.
`tpl_6 (x)`	Extracts the sixth field of x from a Tuple6 or Tuple7.
`tpl_7 (x)`	Extracts the seventh field of x from a Tuple7.

B.2.11 Clock

`Clock` is an abstract type of two components: a single `Bit` oscillator and a `Bool` gate.

```
typedef ... Clock ;
```

`Clock` is in the `Eq` type class, meaning two values can be compared for equality.

Type Classes for `Clock`

	Bits	Eq	Literal	Arith	Ord	Bounded	Bitwise	Bit Reduction	Bit Extend
Clock		√							

B.2.12 Reset

`Reset` is an abstract type.

```
typedef ... Reset ;
```

`Reset` is in the `Eq` type class, meaning two fields can be compared for equality.

Type Classes for `Reset`

	Bits	Eq	Literal	Arith	Ord	Bounded	Bitwise	Bit Reduction	Bit Extend
Reset		√							

B.2.13 Inout

An `Inout` type is a first class type that is used to pass Verilog inouts through a BSV module.

```
Inout#(a);
```

Example of declaring a variable named `foo` of the type `Inout`:

```
Inout#(int) foo;
```

An `Inout` type is a valid subinterface type (like `Clock` and `Reset`). A value of an `Inout` type is `clocked_by` and `reset_by` a particular Clock and Reset.

`Inout`s are connectable via the `Connectable` typeclass. The use of `mkConnection` instantiates a Verilog module `InoutConnect`. The `Inout`s must be on the same clock and the same reset. The clock and reset of the `Inout`s may be different than the clock and reset of the parent module of the `mkConnection`.

```
instance Connectable#(Inout#(a, x1), Inout#(a, x2))
   provisos (Bit#(a,sa));
```

B.2.14 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.

`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.2.15 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);`

rJoinMutuallyExclusive	
	Union of two sets of rule, with rules in the all rules in the left set annotated as mutually exclusive with all rules in the right set.No relationship between the rules in the left set or between the rules in the right set is assumed. This annotation is used in scheduling and checked during simulation.
	`function Rules rJoinMutuallyExclusive(Rules x, Rules y);`

rJoinExecutionOrder	
	Union of two sets of rule, with the rules in the left set executing before the rules in the right set.No relationship between the rules in the left set or between the rules in the right set is assumed. If any pair of rules cannot execute in the specified order in the same clock cycle, that pair of rules will conflict.
	`function Rules rJoinExecutionOrder(Rules x, Rules y);`

rJoinConflictFree	
	Union of two sets of rule, with the rules in the left set annotated as conflict-free with the rules in the right set. This assumption is used during scheduling and checked during simulation. No relationship between the rules in the left set or between the rules in the right set is assumed.
	`function Rules rJoinConflictFree(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 $n1 + n2 = n3$
`Mul`	`Mul#(n1,n2,n3)`	Assert $n1 * n2 = n3$
`Div`	`Div#(n1,n2,n3)`	Assert ceiling $n1/n2 = n3$
`Max`	`Max#(n1,n2,n3)`	Assert $\max(n1, n2) = n3$
`Log`	`Log#(n1,n2)`	Assert ceiling $\log_2(n1) = n2$.

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 $n1 - n2$
`TMul`	`TMul#(n1,n2)`	Calculate $n1 * n2$
`TDiv`	`TDiv#(n1,n2)`	Calculate ceiling $n1/n2$
`TLog`	`TLog#(n1)`	Calculate ceiling $\log_2(n1)$
`TExp`	`TExp#(n1)`	Calculate 2^{n1}

Examples using other arithmetic functions:

```
Int#(TAdd#(5,n));                   // defines a signed integer n+5 bits wide
                                    // n must be in scope somewhere

typedef TAdd#(vsize, 8) Bigsize;    // defines a new type Bigsize which
                                    // is 8 bits wider than vsize

typedef Bit#(TLog#(n)) CBToken#(numeric 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.3.3 valueOf and SizeOf pseudo-functions

Prelude provides these pseudo-functions to convert between types and numeric values. The pseudo-function `valueof` (or `valueOf`) is used to convert a numeric type into the corresponding Integer value. The pseudo-function `SizeOf` is used to convert a type t into the numeric type representing its bit size.

`valueof` `valueOf`	Converts a numeric type into its Integer value.
	`function Integer valueOf (t) ;`

Example:

```
module mkFoo (Foo#(n));
  UInt#(n) x;
  Integer y = valueOf(n);
endmodule
```

`SizeOf`	Converts a type into a numeric type representing its bit size.
	`function t SizeOf#(any_type)` `   provisos (Bits#(any_type, sa)) ;`

Example:

```
any_type x = structIn;
Bit#(SizeOf#(any_type)) = pack(structIn);
```

B.4 Registers and Wires

Register and Wire Interfaces		
Name	Section	Description
`Reg`	B.4.1	Register interface.
`RWire`	B.4.2	Similar to a register with output wrapped in a `Maybe` type to indicate validity.
`Wire`	B.4.3	Interchangeable with a `Reg` interface, validity of the data is implicit.
`BypassWire`	B.4.4	Implementation of the Wire interface where the _write method is always_enabled.
`DWire`	B.4.5	Implementation of the Wire interface where the _read method is always_ready.
`PulseWire`	B.4.6	`RWire` without any data.
`ReadOnly`	B.4.7	Interface which provides a value.

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.

`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));`

Scheduling Annotations mkReg, mkRegU, mkRegA		
	read	write
read	CF	SB
write	SA	SBR

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)) ;`

Scheduling Annotations mkRWire		
	wget	wset
wget	CF	SA
wset	SB	C

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));`

Scheduling Annotations mkWire		
	read	write
read	CF	SA
write	SB	C

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).

`mkBypassWire`	Creates a `BypassWire`. The write method is always_enabled.
	`module mkBypassWire(Wire#(element_type))` `   provisos (Bits#(element_type, element_width));`

Scheduling Annotations mkBypassWire		
	read	write
read	CF	SA
write	SB	C

B.4.5 DWire

DWire is an implementation of the Wire interface where the `_read` method is an `always_ready` method and thus has no implicit conditions. Unlike the BypassWire however, the `_write method` need not be always enabled. On cycles when a DWire is written to, the `_read` method returns that value. On cycles when no value is written, the `_read` method instead returns a default value that is specified as an argument during instantiation.

`mkDWire`	Creates a `DWire`. The read method is always_ready.
	`module mkDWire#(a_type defaultval)(Wire#(element_type))` `   provisos (Bits#(element_type, element_width));`

Scheduling Annotations mkDWire		
	read	write
read	CF	SA
write	SB	C

B.4.6 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` and `mkPulseWireOR` modules are provided to create a `PulseWire`. The `mkPulseWireOR` is nearly identical to the `mkPulseWire` module except that the `send` method in the `mkPulseWireOR` does not conflict with itself. Calling the `send` method for a `mkPulseWire` from 2 rules causes the two rules to conflict while in the `mkPulseWireOR` there is no conflict. In other words, the `mkPulseWireOR` acts a logical "OR".

`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);`

`mkPulseWireOR`	Returns a `PulseWire` which acts like a logical "Or". The `send` method of the same wire can be used in two different rules without conflict.
	`module mkPulseWireOR(PulseWire);`

Scheduling Annotations mkPulseWire		
	_read	send
_read	CF	SA
send	SB	C

Scheduling Annotations mkPulseWireOR		
	_read	send
_read	CF	SA
send	SB	SBR

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.4.7 ReadOnly

`ReadOnly` is an interface which provides a value. The `_read` shorthand can be used to read the value.

Interfaces and Methods

`ReadOnly` Interface				
Method			Arguments	
Name	Type	Description	Name	Description
`_read`	`a_type`	Reads the data	`a_type`	Data to be read, of datatype `type`.


```
interface ReadOnly #( type a_type ) ;
   method a_type _read() ;
endinterface
```

B.5 Miscellaneous Functions

B.5.1 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);`

B.5.2 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));`

`signedMul`	Performs full precision multiplication on two Int#(n) operands of different sizes.
	`function Int#(m) signedMul(Int#(n) x, Int#(k) y)` `  provisos (Add#(n,k,m));`

`unsignedMul`	Performs full precision multiplication on two unsigned UInt#(n) operands of different sizes.	`function UInt#(m) unsignedMul(UInt#(n) x, UInt#(k) y) provisos (Add#(n,k,m));`

B.5.3 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));`

`composeM`	Creates a new monadic function, `m#(c)`, made up of functions, `f` and `g`. `c(a) = f(g(a))`	`function (function m#(c_type) (a_type x0)) composeM(function m#(c_type) f(b_type x1), function m#(b_type) g(a_type x2)) provisos # (Monad#(m));`

`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 (function c_type new (b_type y, a_type x)) flip (function c_type old (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) );
```


B.5.4 Bit Functions

The following functions operate on Bit#(n) variables.

`msb`	Returns the most significant bit of `x`
	`function Bit#(1) msb(Bit#(n) x)` `   provisos(Add#(1,k,n));`

`parity`	Returns the parity of the bit argument `v`. Example: parity(5'b1) = 1, parity(5'b3) = 0;
	`function Bit#(1) parity(Bit#(n) v);`

`reverseBits`	Reverses the order of the bits in the argument `x`.
	`function Bit#(n) reverseBits(Bit#(n) x);`

`countOnes`	Returns the count of the number of 1's in the bit vector `bin`.

	`function UInt#(lgn1) countOnes ( Bit#(n) bin )`
	`  provisos (Add#(1, n, n1), Log#(n1, lgn1),`
	`             Add#(1, xx, lgn1) );`

`countZerosMSB`	For the bit vector `bin`, count the number of 0s until the first 1, starting from the most significant bit (MSB).
	`function UInt#(lgn1) countZerosMSB ( Bit#(n) bin )` `  provisos (Add#(1, n, n1), Log#(n1, lgn1) );`

`countZerosLSB`	For the bit vector `bin`, count the number of 0s until the first 1, starting from the least significant bit (LSB).
	`function UInt#(lgn1) countZerosLSB ( Bit#(n) bin )` `  provisos (Add#(1, n, n1), Log#(n1, lgn1) );`

`truncateLSB`	Truncates a `Bit#(m)` to a `Bit#(n)` by dropping bits starting with the LSB.
	`function Bit#(n) truncateLSB(Bit#(m) x)` `   provisos(Add#(n,k,m));`

B.5.5 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);`

`when`	Adds an implicit condition onto an expression.
	`function a when(Bool condition, a arg);`

Example - adding the implicit condition readCount==0 to the action

```
function Action startReadSequence (BAddr startAddr,
                                   UInt#(6) count);
   return  when ((readCount == 0), // implicit condition of the action
   (action
       readAddr    <= startAddr ;
       readCount   <= count ;
    endaction));
endfunction

rule readSeq;         // readCount==0 is an implicit condition
    startReadSequence (addr, count);
endrule
```

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 Foundation Libraries

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 AzureIP™ Foundation libraries. To use any of these libraries in a package you must explicitly import the package using an `import` clause.

Bluespec's AzureIP™ intellectual property (IP) accelerates hardware design and modeling. There are two AzureIP library families, Foundation and Premium:

- Foundation is an extensive family of components, types and functions that are included with the Bluespec toolsets for use in your models and designs – they serve as a foundational base for your modeling and implementation work.
- Premium is the designation for Bluespec's fee-based AzureIP.

C.1 Storage and Data Structures

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.

<table>
<tr><td rowspan="2">mkRegFile</td><td>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.</td></tr>
<tr><td><pre>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));</pre></td></tr>
</table>

<table>
<tr><td rowspan="2">mkRegFileFull</td><td>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).</td></tr>
<tr><td><pre>module mkRegFileFull#(RegFile#(index_t, data_t))
  provisos (Bits#(index_t, size_index),
            Bits#(data_t, size_data)
            Bounded#(index_t));</pre></td></tr>
</table>

<table>
<tr><td rowspan="2">mkRegFileWCF</td><td>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).</td></tr>
<tr><td><pre>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));</pre></td></tr>
</table>

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, which allows white spaces (including new lines, tabs, underscores, and form-feeds), comments, binary and hexadecimal numbers. Length and base format must not be specified for the numbers.

<table>
<tr><td rowspan="2">mkRegFileLoad</td><td>Create a RegFile using the file to provide the initial contents of the array.</td></tr>
<tr><td><pre>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));</pre></td></tr>
</table>

`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) );`	

```
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 `FIFOLevel`. 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`

FIFOLevel	Indicates the level or current number of items stored in the FIFO	common methods
		`notFull`
		`notEmpty`
		`isLessThan`
		`isGreaterThan`

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.

Guarded (G) versions of `FIFOF` are available which allow more control over implicit conditions. With the guarded versions the user can specify whether the enqueue or dequeue side is guarded.

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
```

The `FIFO` and `FIFOF` interfaces belong to the `toGet` and `toPut` typeclasses. You can use the `toGet` and `toPut` functions to convert `FIFO` and `FIFOF` interfaces to `Get` and `Put` interfaces (Section C.6.1).

Modules

The FIFO and FIFOF interface types are provided by the module constructors: mkFIFO, mkFIFO1, mkSizedFIFO, mkDepthParamFIFO, 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. Most FIFOs do not allow simultaneous enqueue and dequeue operations when the FIFO is full. The exception is the pipeline FIFO (mkLFIFO), which does allow simultaneous enqueue and dequeue operations in the same clock cycle when full, as shown in the following table.

Simultaneous enq and deq behavior			
FIFO type	Condition		
	empty	not empty not full	full
mkFIFO mkFIFOF		√	
mkFIFO1 mkFIFOF1			
mkLFIFO mkLFIFOF		√	√
mkLFIFO1 mkLFIFOF1			√

For creating a FIFOF interface use the "F" version of the module, such as mkFIFOF.

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.

There is also available a guarded (G) version of each of the FIFOFs. The guarded FIFOF takes two Boolean parameters; ugenq and ugdeq. Setting either parameter TRUE indicates the method (enq for ugenq, deq for ugdeq) is unguarded. If both are TRUE the FIFOF behaves the same as an unguarded FIFOF. If both are FALSE the behavior is the same as a regular FIFOF.

Module Name	BSV Module Declaration *For all modules, width_any may be 0*
FIFO of depth 2.	
mkFIFO mkFIFOF mkUGFIFOF	`module mkFIFO (FIFO#(element_type))` `   provisos (Bits#(element_type, width_any));`

Guarded FIFOF of depth 2.	
mkGFIFOF	`module mkGFIFOF (Bool ugenq, Bool ugdeq)(FIFOF#(element_type))` `   provisos (Bits#(element_type, width_any));`

FIFO of depth 1	
mkFIFO1 mkFIFOF1 mkUGFIFOF1	`module mkFIFO1(FIFO#(element_type))` `   provisos (Bits#(element_type, width_any));`

Guarded `FIFOF` of depth 1	
`mkGFIFOF1`	`module mkGFIFOF1(Bool ugenq, Bool ugdeq)(FIFOF#(element_type))` `   provisos (Bits#(element_type, width_any));`

`FIFO` of given depth n	

`mkSizedFIFO`
`mkSizedFIFOF`
`mkUGSizedFIFOF`
	`module mkSizedFIFO(Integer n)(FIFO#(element_type))`
	`provisos (Bits#(element_type, width_any));`

Guarded `FIFOF` of given depth n	

`mkGSizedFIFOF`	`module mkGSizedFIFOF(Bool ugenq, Bool ugdeq, Integer n)`
	`                    (FIFOF#(element_type))`
	`  provisos (Bits#(element_type, width_any));`

`FIFO` of given depth n where n is a Verilog parameter or computed from compile-time constants and Verilog parameters.	

`mkDepthParamFIFO`
`mkDepthParamFIFOF`
`mkUGDepthParamFIFOF`
	`module mkDepthParamFIFO(UInt#(32) n)(FIFO#(element_type))`
	`  provisos (Bits#(element_type, width_any));`

Guarded `FIFOF` of given depth n where n is a Verilog parameter or computed from compile-time constants and Verilog parameters.	

`mkGDepthParamFIFOF`	`module mkGDepthParamFIFOF`
	`           (Bool ugenq, Bool ugdeq, UInt#(32) n)`
	`           (FIFOF#(element_type))`
	`  provisos (Bits#(element_type, width_any));`

Pipeline `FIFO` of depth 1. `deq` and `enq` can be simultaneously applied in the same clock cycle when the `FIFO` is full.	

`mkLFIFO`
`mkLFIFOF`
`mkUGLFIFOF`
	`module mkLFIFO (FIFO#(element_type))`
	`  provisos (Bits#(element_type, width_any));`

Guarded pipeline FIFOF of depth 1. deq and enq can be simultaneously applied in the same clock cycle when the FIFOF is full.	
mkGLFIFOF	`module mkGLFIFOF (Bool ugenq, Bool ugdeq)(FIFOF#(element_type)) provisos (Bits#(element_type, width_any));`

Functions

The FIFO package provides a function `fifofToFifo` to convert an interface of type `FIFOF` to an interface of type `FIFO`.

Converts a FIFOF interface to a FIFO interface.	
fifofToFifo	`function FIFO#(a) fifofToFifo (FIFOF#(a) f);`

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
```

Scheduling Annotations

Scheduling constraints describe how methods interact within the schedule. For example, a `clear` to a given FIFO must be sequenced after (`SA`) an `enq` to the same FIFO. That is, when both `enq` and `clear` execute in the same cycle, the resulting FIFO state is empty. For correct rule behavior the rule executing `enq` must be scheduled before the rule calling `clear`.

The table below lists the scheduling annotations for the FIFO modules `mkFIFO`, `mkSizedFIFO`, and `mkFIFO1`.

Scheduling Annotations mkFIFO, mkSizedFIFO, mkFIFO1				
	enq	first	deq	clear
enq	C	CF	CF	SB
first	CF	CF	SB	SB
deq	CF	SA	C	SB
clear	SA	SA	SA	SBR

The table below lists the scheduling annotations for the pipeline FIFO module, `mkLFIFO`. The pipeline FIFO has a few more restrictions since there is a combinational path between the `deq` side and the `enq` side, thus restricting `deq` calls before `enq`.

Scheduling Annotations mkLFIFO				
	enq	first	deq	clear
enq	C	SA	SAR	SB
first	SB	CF	SB	SB
deq	SBR	SA	C	SB
clear	SA	SA	SA	SBR

The `FIFOF` modules add the `notFull` and `notEmpty` methods. These methods have SB annotations with the Action methods that change FIFO state. These SB annotations model the atomic behavior of a FIFO, that is when `enq`, `deq`, or `clear` are called the state of `notFull` and `notEmpty` are changed. This is no different than the annotations on `mkReg` (which is `read` SB `write`), where actions are atomic and the execution module is one rule fires at a time. This does differ from a pure hardware module of a FIFO or register where the state does not change until the clock edge.

Scheduling Annotations mkFIFOF, mkSizedFIFOF, mkFIFOF1						
	enq	notFull	first	deq	notEmpty	clear
enq	C	SA	CF	CF	SA	SB
notFull	SB	CF	CF	SB	CF	SB
first	CF	CF	CF	SB	CF	SB
deq	CF	SA	SA	C	SA	SB
notEmpty	SB	CF	CF	SB	CF	SB
clear	SA	SA	SA	SA	SA	SBR

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	Comments

`mkFIFO`
`mkFIFOF`
`mkUGFIFOF`
`mkGFIFOF`
	`FIFO2.v` `FIFO20.v`	

`mkFIFO1`
`mkFIFOF1`
`mkUGFIFOF1`
`mkGFIFOF1`
	`FIFO1.v` `FIFO10.v`	

`mkSizedFIFO`
`mkSizedFIFOF`
`mkUGSizedFIFOF`
`mkGSizedFIFOF`
	`SizedFIFO.v` `SizedFIFO0.v`
	`FIFO1.v` `FIFO10.v`
	`FIFO2.v` `FIFO20.v`
		If the depth of the FIFO = 1, then `FIFO1.v` and `FIFO10.v` are used, if the depth = 2, then `FIFO2.v` and `FIFO20.v` are used.

`mkDepthParamFIFOF`
`mkUGDepthParamFIFOF`
`mkGDepthParamFIFOF`
	`SizedFIFO.v` `SizedFIFO0.v`	

`mkLFIFO`
`mkLFIFOF`
`mkUGLFIFOF`
`mkGLFIFOF`
	`FIFOL1.v` `FIFOL10.v`	

C.1.4 FIFOLevel

Package Name

import FIFOLevel :: * ;

Description

The BSV `FIFOLevel` library provides enhanced FIFO interfaces and modules which include methods to indicate the level or the current number of items stored in the FIFO. Both single clock and dual clock (separate clocks for the enqueue and dequeue sides) versions are included in this package.

Interfaces and methods

The `FIFOLevelIfc` interface defines methods to compare the current level to `Integer` constants for a single clock. The `SyncFIFOLevelIfc` defines the same methods for dual clocks; thus it provides methods for both the source (enqueue) and destination (dequeue) clock domains. Instead of methods to compare the levels, the `FIFOCountIfc` and `SyncFIFOCountIfc` define methods to return counts of the FIFO contents, for single clocks and dual clocks respectively.

Interface Name	Parameter name	Parameter Description	Requirements of modules implementing the ifc
`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 and >2
`FIFOCountIfc`	*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 and >2
`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 and must be a power of 2 and >=2
`SyncFIFOCountIfc`	*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 and must be a power of 2 and >=2

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 dependent on the FIFO depth. The `fifoDepth` parameter must be > 2.

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` compile-time constant
`isGreaterThan`	Bool	Returns `True` if the depth of the `FIFO` is greater than the `Integer` constant, `c1`.	c1	an `Integer` compile-time constant


```
interface FIFOLevelIfc#( type element_type, numeric 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 ) ;
endinterface
```

In addition to common FIFO methods, the `FIFOCountIfc` interface defines a method to return the current number of elements as an bit-vector. See Section C.1.2 for details on `enq`, `deq`, `first`, `clear`, `notFull`, and `notEmpty`. Note that the `FIFOCountIfc` interface has a type parameter for the `fifoDepth`. This numeric type parameter is needed, since the width of the counter is dependent on the FIFO depth. The `fifoDepth` parameter must be > 2.

FIFOCountIfc		
Method		
Name	Type	Description
`count`	UInt#(TLog#(TAdd#(fifoDepth,1)))	Returns the number of items in the `FIFO`.

```
interface FIFOCountIfc#( type element_type, numeric type fifoDepth) ;
   method Action enq ( element_type sendData ) ;
   method Action deq () ;
   method element_type first () ;

   method Bool notFull ;
   method Bool notEmpty ;

   method UInt#(TLog#(TAdd#(fifoDepth,1))) count;

   method Action clear;
endinterface
```

The interfaces `SyncFIFOLevelIfc` and `SyncFIFOCountIfc` are dual clock versions of the `FIFOLevelIfc` and `FIFOCountIfc`. Methods are provided for both source and destination clock domains. The following table describes the dual clock `notFull` and `notEmpty` methods, as well as the dual clock `clear` methods, which are common to both interfaces. Note that the `SyncFIFOLevelIfc` and `SyncFIFOCountIfc` interfaces each have a type parameter for `fifoDepth`. This numeric type parameter is needed, since the width of the counter is dependent on the FIFO depth. The `fifoDepth` parameter must be a power of 2 and >= 2.

Common Dual Clock Methods		
Name	Type	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.
`sClear`	Action	Clears the FIFO from the source side.
`dClear`	Action	Clears the FIFO from the destination side.

In addition to common FIFO methods (Section C.1.2) and the common dual clock methods above, 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.

SyncFIFOLevelIfc Methods				
Method			Argument	
Name	Type	Description	Name	Description
`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 it 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 it 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, numeric 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 ;

   method Bool sIsLessThan   ( Integer c1 ) ;
   method Bool sIsGreaterThan( Integer c1 ) ;
   method Bool dIsLessThan   ( Integer c1 ) ;
   method Bool dIsGreaterThan( Integer c1 ) ;
```



```
   method Action sClear;
   method Action dClear;
endinterface
```

In addition to common FIFO methods (Section C.1.2) and the common dual clock methods above, the `SyncFIFOCountIfc` interface defines methods to return the current number of elements. Methods are provided for both the source (enqueue side) and destination (dequeue side) clock domains.

SyncFIFOCountIfc		
Method		
Name	Type	Description
`sCount`	UInt#(TLog#(TAdd#(fifoDepth,1)))	Returns the number of items in the `FIFO` from the source side.
`dCount`	UInt#(TLog#(TAdd#(fifoDepth,1)))	Returns the number of items in the `FIFO` from the destination side.

```
interface SyncFIFOCountIfc#( type element_type, numeric 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 ;

   method UInt#(TLog#(TAdd#(fifoDepth,1))) sCount;
   method UInt#(TLog#(TAdd#(fifoDepth,1))) dCount;

   method Action sClear;
   method Action dClear;
endinterface
```

The `FIFOLevelIFC`, `SyncFIFOLevelIfc`, `FIFOCountIfc`, and `SyncFIFOCountIfc` interfaces belong to the `toGet` and `toPut` typeclasses. You can use the `toGet` and `toPut` functions to convert these interfaces to `Get` and `Put` interfaces (Section C.6.1).

Modules

The module `mkFIFOLevel` provides the `FIFOLevelIfc` interface. 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.

There is also available a guarded (G) version of `FIFOLevel` which takes three Boolean parameters; `ugenq`, `ugdeq`, and `ugcount`. Setting any of the parameters to `TRUE` indicates the method (`enq` for `ugenq`, `deq` for `ugdeq`, and `isLessThan`, `isGreaterThan` for `ugcount`) is unguarded. If all three are `FALSE` the behavior is the same as a regular `FIFOLevel`.

Module Name	BSV Module Declaration

`mkFIFOLevel`	`module mkFIFOLevel (`
	`FIFOLevelIfc#(element_type, fifoDepth) )`
	`provisos( Bits#(element_type, width_element )`
	`Log#(TAdd#(fifoDepth,1),cntSize) ) ;`

	Comment: `width_element` may be 0

Module Name	BSV Module Declaration

mkGFIFOLevel	`module mkGFIFOLevel#(Bool ugenq, Bool ugdeq, Bool ugcount)`
	`( FIFOLevelIfc#(element_type, fifoDepth) )`
	`provisos( Bits#(element_type, width_element ),`
	`Log#(TAdd#(fifoDepth,1),cntSize));`

	Comment: `width_element` may be 0

The module `mkFIFOCount` provides the interface `FIFOCountIfc`. There is also available a guarded (G) version of `FIFOCount` which takes three Boolean parameters; `ugenq`, `ugdeq`, and `ugcount`. Setting any of the parameters to `TRUE` indicates the method (`enq` for `ugenq`, `deq` for `ugdeq`, and `count` for `ugcount`) is unguarded. If all three are `FALSE` the behavior is the same as a regular `FIFOCount`.

Module Name	BSV Module Declaration

mkFIFOCount	`module mkFIFOCount(`
	`FIFOCountIfc#(element_type, fifoDepth) ifc )`
	`provisos (Bits#(element_type, width_element));`

	Comment: `width_element` may be 0

Module Name	BSV Module Declaration

mkGFIFOCount	`module mkGFIFOCount#(Bool ugenq, Bool ugdeq, Bool ugcount)`
	`( FIFOCountIfc#(element_type, fifoDepth) ifc )`
	`provisos (Bits#(element_type, width_element));`

	Comment: `width_element` may be 0

The modules `mkSyncFIFOLevel` and `mkSyncFIFOCount` are dual clock FIFOs, 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.

The module `mkSyncFIFOLevel` provides the `SyncFIFOLevelIfc` interface.

Module Name	BSV Module Declaration

mkSyncFIFOLevel	`module mkSyncFIFOLevel(`
	`Clock sClkIn, Reset sRstIn,`
	`Clock dClkIn,`
	`SyncFIFOLevelIfc#(element_type, fifoDepth) ifc )`
	`provisos( Bits#(element_type, width_element),`
	`Add#(1,fifoDepth,fifoDepth1),`
	`Log#(fifoDepth1,cntSz) ) ;`

	Comment: `width_element` may be 0

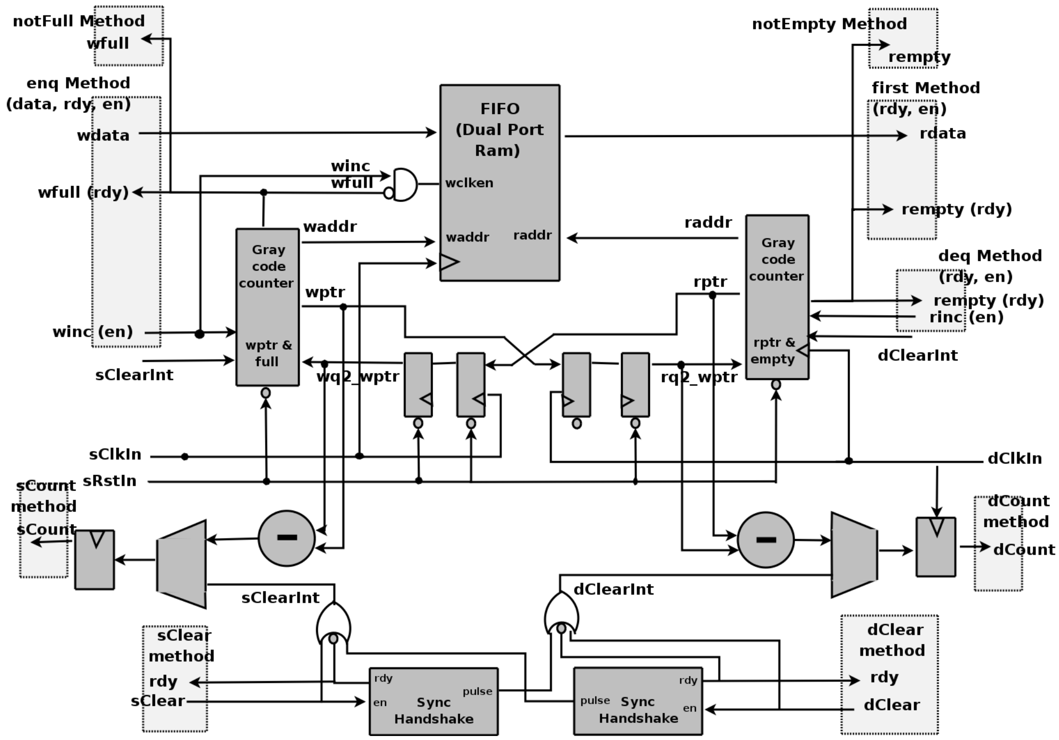

Figure 3: SyncFIFOCount

The module `mkSyncFIFOCount`, as shown in Figure 3 provides the `SyncFIFOCountIfc` interface. Because of the synchronization latency, the count reports may be different between the source and the destination clocks. Note however, that the `sCount` and `dCount` reports give pessimistic values with the appropriate side. That is, the count `sCount` (on the enqueue clock) will report the exact count of items in the FIFO or a larger count. The larger number is due to the synchronization delay in observing the dequeue action. Likewise, the `dCount` (on the dequeue clock) returns the exact count or a smaller count. The maximum disparity between `sCount` and `dCount` depends on the difference in clock periods between the source and destination clocks.

The module provides `sClear` and `dClear` methods, both of which cause the contents of the FIFO to be removed. Since the clears must be synchronized and acknowledged from one domain to the other, there is a non-trivial delay before the FIFO recovers from the clear and can accept additional enqueues or dequeues (depending on which side is cleared). The calling of either method immediately disables other activity in the calling domain. That is, calling `sClear` in cycle `n` causes the enqueue to become unready in the next cycle, `n+1`. Likewise, calling `dClear` in cycle `n` causes the dequeue to become unready in the next cycle, `n+1`.

After the `sClear` method is called, the FIFO appears empty on the dequeue side after three `dClock` edges. Three `sClock` edges later, the FIFO returns to a state where new items can be enqueued. The latency is due to the full handshaking synchronization required to send the clear signal to `dClock` and receive the acknowledgement back.

For the `dClear` method call, the enqueue side is cleared in three `sClkIn` edges and items can be enqueued at the fourth edge. All items enqueued at or before the clear are removed from the FIFO.

Note that there is a ready signal associated with both `sClear` and `dClear` methods to ensure that the clear is properly sent between the clock domains. Also, `sRstIn` must be synchronized with the `sClkIn`.

Module Name	BSV Module Declaration
mkSyncFIFOCount	`module mkSyncFIFOCount( Clock sClkIn, Reset sRstIn, Clock dClkIn, SyncFIFOCountIfc#(element_type, fifoDepth) ifc ) provisos( Bits#(element_type, width_element));`
	Comment: `width_element` may be 0

Example

The following example shows the use of `SyncFIFOLevel` 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 moveData ( burstOut ) ;
   let dataToSend = fifo.first ;
   fifo.deq ;
   ...
   burstOut <= !dFifoAlmostEmpty;

endrule
```

Verilog Modules

The modules described in this section correspond to the following Verilog modules, which are found in the Bluespec Verilog library, `$BLUESPECDIR/Verilog/`.

BSV Module Name	Verilog Module Names	
`mkFIFOLevel` `mkFIFOCount`	`SizedFIFO.v`	`SizedFIFO0.v`
`mkSyncFIFOLevel` `mkSyncFIFOCount`	`SyncFIFOLevel.v`	

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 (section 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));`

Scheduling Annotations mkConfigReg, mkConfigRegU, mkConfigRegA		
	read	write
read	CF	CF
write	CF	SBR

C.1.6 DReg

Package Name

import DReg :: * ;

Description

The `DReg` package allows a designer to create registers which store a written value for only a single clock cycle. The value written to a DReg is available to read one cycle after the write. If more than one cycle has passed since the register has been written however, the value provided by the register is instead a default value (that is specified during module instantiation). These registers are useful when wanting to send pulse values that are only asserted for a single clock cycle. The DReg is the register equivalent of a DWire B.4.5.

Modules

The `DReg` package provides three modules; `mkDReg` creates a register with a given reset/default value and synchronous reset logic, `mkDRegU` creates a register without any reset (but which still takes a default value as an argument), and `mkDRegA` creates a register with a given reset/default value and asynchronous reset logic.

`mkDReg`	Make a register with a given reset/default value. Reset logic is synchronous
	`module mkDReg#(a_type dflt_rst_val)(Reg#(a_type)) provisos (Bits#(a_type, sizea));`

`mkDRegU`	Make a register without any reset but with a specified default; initial simulation value is alternating 01 bits.
	`module mkDRegU#(a_type dflt_val)(Reg#(a_type)) provisos (Bits#(a_type, sizea));`

`mkDRegA`	Make a register with a given reset/default value. Reset logic is asynchronous.
	`module mkDRegA#(a_type, dflt_rst_val)(Reg#(a_type)) provisos (Bits#(a_type, sizea));`

Scheduling Annotations mkDReg, mkDRegU, mkDRegA		
	read	write
read	CF	SB
write	SA	SBR

C.1.7 RevertingVirtualReg

Package Name

import RevertingVirtualReg :: * ;

Description

The `RevertingVirtualReg` package allows a designer to force a schedule when scheduling attributes cannot be used. Since scheduling attributes cannot be put on methods, this allows a designer to control the schedule between two methods, or between a method and a rule by adding a virtual register between the two. The module `RevertingVirtualReg` creates a virtual register; no actual state elements are generated.

Modules

The `RevertingVirtualReg` package provides the module `mkRevertingVirtualReg`. The properties of the module are:

- it schedules exactly like an ordinary register;
- it reverts to its reset value at the end of each clock cycle.

These imply that all allowed reads will return the reset value (since they precede any writes in the cycle); thus the module neither needs nor instantiates any actual state element.

`mkRevertingVirtualReg`	Creates a virtual register reverting to the reset value at the end of each clock cycle.
	`module mkRevertingVirtualReg#(a_type rst)(Reg#(a_type))` `  provisos (Bits#(a_type, sizea));`

Scheduling Annotations mkRevertingVirtualReg		
	read	write
read	CF	SB
write	SA	SBR

Example Use `mkRevertingVirtualReg` to create the execution order of the_rule followed by the_method

```
Reg#(Bool) virtualReg <- mkRevertingVirtualReg(True);

rule the_rule (virtualReg);  // reads virtualReg
   ...
endrule

method Action the_method;
   virtualReg <= False;      // writes virtualReg
   ...
endmethod
```

In a given cycle, reads always precede writes for a register. Therefore the reading of `virtualReg` by `the_rule` will precede the writing of `virtualReg` in `the_method`. The execution order will be `the_rule` followed by `the_method`.

C.1.8 BRAM

Package Name

import BRAM :: * ;

Description

This package provides Block RAMS for use in Xilinx FPGAs. The `ClientServer` package must also be imported when using the `BRAM` package.

Types and type classes

The `BRAM` package defines a structure, `BRAMRequest`, along with types `BRAMServer` and `BRAMClient`.

```
typedef struct {Bool write;
                addr address;
                data datain;
               } BRAMRequest#(type addr_t, type data_t) deriving(Bits, Eq);
```

```
typedef Server#(BRAMRequest#(addr_t, data_t), data_t)
                BRAMServer#(type addr_t, type data_t);
typedef Client#(BRAMRequest#(addr_t, data_t), data_t)
                BRAMClient#(type addr_t, type data_t);
```

Interfaces and Methods

The `BRAM` package defines the `BRAM` interface.

```
interface BRAM#(type addr_t type data_t);
   interface BRAMServer#(addr_t, data_t) portA;
   interface BRAMServer#(addr_t, data_t) portB;
endinterface: BRAM
```

Modules

The `BRAM` package provides the following modules: `mkSyncBRAM`, `mkBRAM`, and `mkSyncBRAMLoadEither`. These modules correspond to the Xilinx Dual-Port Block RAM with two write ports.

<table>
<tr><td rowspan="2"><code>mkSyncBRAM</code></td><td>Creates a 2-port BRAM with two clocks; <code>clkA</code> is shared by the primary read and write port, <code>clkB</code> is shared by the secondary read and write port. Resets must be synched to the clock domains. There is no default clock or reset domain.</td></tr>
<tr><td><pre>module mkSyncBRAM#
      (Clock clkA, Reset rstNA, Clock clkB, Reset rstNB)
      (BRAM#(addr_t, data_t))
   provisos (Bits(addr_t, addr_sz), Bits#(data_t, data_sz),
        Add#(z, 1, addr_sz), Add#(x, 1, data_sz),
        Bounded#(addr_t));</pre></td></tr>
</table>

`mkBRAM`	Creates a 2-port BRAM with a single clock.	

	<pre>module mkBRAM(BRAM#(addr_t, data_t))
	   provisos(Bits#(addr_t, addr_sz),Bits#(data_t, data_sz),
	       Add#(z, 1, addr_sz),Add#(x, 1, data_sz),
	       Bounded#(addr_t));</pre>	

`mkSyncBRAMLoadEither`	Creates a dual-clock, two port `BRAM` and loads the initial values from a file containing either hex or binary values. Resets must be synched to the clock domains. There is no default clock or reset domain.

	<pre>module mkSyncBRAMLoadEither#
	          (Clock clkA,Reset rstNA,Clock clkB,
	           Reset rstNB, String file, Integer binary)
	          (BRAM#(addr_t, data_t))
	   provisos(Bits#(addr_t, addr_sz),Bits#(data_t, data_sz),
	       Add#(z, 1, addr_sz), Add#(x, 1, data_sz),
	       Bounded#(addr_t));</pre>

Verilog Modules

`BRAM` modules correspond to the following Verilog modules, which are found in the Bluespec Verilog library, `$BLUESPECDIR/Verilog/`.

BSV Module Name	Verilog Module Names
`mkSyncBRAM` `mkBRAM`	`BRAM.v`
`mkSyncBRAMLoadEither`	`BRAMLoad.v`

C.1.9 BRAMFIFO

Description

The BRAMFIFOs are FIFOs which utilize the Xilinx Block RAMs, as implemented in the `BRAM` package, described in Section C.1.8.

This package is provided as both a compiled library package and as BSV source code to facilitate customization. The source code file can be found in the `$BLUESPECDIR/BSVSource` directory. To customize a package, copy the file into a local directory and then include the local directory in the path when compiling. This is done by specifying the path with the `-p` option as described in the BSV Users Guide.

Packages

To include a package in your design, use the `import` syntax.

```
import BRAMFIFO :: * ;
```

Interfaces

The `BRAMFIFO` package uses `FIFOF`, `FIFO`, and `SyncFIFOIfc` interfaces, as defined in the `FIFOF`, `FIFO`, (both in Section C.1.3) and `Clocks` (Section C.8.7) packages.

Modules

`mkSizedBRAMFIFOF`	Provides a Xilinx BRAM based FIFOF of a given depth, `n`.

	`module mkSizedBRAMFIFOF#(Integer n) (FIFOF#(element_type))`
	`   provisos (Bits(element_type, width_any),`
	`             Add#(1,z,width_any));`

`mkSizedBRAMFIFO`	Provides a Xilinx BRAM based FIFO of a given depth, `n`.

	`module mkSizedBRAMFIFO#(Integer n)(FIFO#(element_type))`
	`   provisos(Bits#(t, width_element),`
	`            Add#(1, z, width_element) );`

`mkSyncBRAMFIFO`	Provides a Xilinx BRAM based FIFO for sending data across clock domains. The `enq` method is in the source `sClkIn` domain, while the `deq` and `first` methods are in the destination `dClkIn` domain. The input and output clocks, along with the input and output resets, are explicitly provided. The default clock and reset are ignored.

	`module mkSyncBRAMFIFO#(Integer depth,`
	`                       Clock sClkIn, Reset sRstIn,`
	`                       Clock dClkIn, Reset dRstIn)`
	`                       (SyncFIFOIfc#(element_type))`
	`   provisos(Bits#(element_type, width_element),`
	`            Add#(1, z, width_element));`

`mkSyncBRAMFIFOToCC`	Provides a Xilinx BRAM based FIFO to send data from a second clock domain into the current clock domain. The output clock and reset are the current clock and reset.

	`module mkSyncBRAMFIFOToCC#(Integer depth,`
	`                           Clock sClkIn, Reset sRstIn)`
	`                           (SyncFIFOIfc#(element_type))`
	`   provisos(Bits#(element_type, width_element),`
	`            Add#(1, z, width_element));`

`mkSyncBRAMFIFOFromCC`	Provides a Xilinx BRAM based FIFO to send data from the current clock domain into a second clock domain. The input clock and reset are the current clock and reset.

	`module mkSyncBRAMFIFOFromCC#(Integer depth,`
	`                             Clock dClkIn, Reset dRstIn)`
	`                             (SyncFIFOIfc#(element_type))`
	`   provisos(Bits#(element_type, width_element),`
	`            Add#(1, z, width_element));`

C.2 Aggregation: Vectors

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)) v1;

From the type, you can see that this will back into a 35-bit vector (5 elements, each with 7 bits).

	34		bit positions		0	
MSB	v1[4]	v1[3]	v1[2]	v1[1]	v1[0]	LSB

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.

C.2.1 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. This function can lead to large compile times, so it can be an inefficient way to create and populate a vector. Instead, the designer should build a vector, then set each element to a value.

```
function Vector#(vsize1, element_type)
     cons (element_type elem, Vector#(vsize, element_type) vect)
  provisos (Add#(1, vsize, vsize1));
```

`nil`	Defines a vector of size zero.
	`function Vector#(0, element_type) nil;`

<table>
<tr><td rowspan="2"><code>append</code></td><td>Append two vectors containing elements of the same type, returning the combined vector. The resulting vector will contain all the elements of <code>vecta</code> followed by all the elements of <code>vectb</code>.</td></tr>
<tr><td><pre>function Vector#(vsize, element_type)
      append(Vector#(v0size,element_type) vecta
              Vector#(v1size,element_type) vectb
  provisos (Add#(v0size, v1size, vsize)); //vsize = vsize0 + v1size</pre></td></tr>
</table>

<table>
<tr><td rowspan="2"><code>concat</code></td><td>Append (concatenate) many vectors, that is a vector of vectors into one vector.</td></tr>
<tr><td><pre>function Vector#(mvsize,element_type)
      concat(Vector#(m,Vector#(n,element_type)) xss)
  provisos (Mul#(m,n,mvsize));</pre></td></tr>
</table>

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 new vector, `my_vector`, of 5 elements of datatytpe `Integer` with elements 0, 1, 2, 3 and 4.

```
Vector #(5, Integer) my_vector = genVector;
// my_vector is a 5 element vector {0,1,2,3,4}
```

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,}
```

Add an element to `my_vector`, creating a bigger vector `my_vector1`.

```
Vector#(3, Integer) my_vector = genVector();
// my_vector = {0, 1, 2, 3}

let my_vector1 = cons(4, a);
// my_vector1 = {4, 0, 1, 2, 3}
```

Append vectors, `my_vector` and `my_vector1`, resulting in a vector `my_vector2`.


```
Vector#(3, Integer) my_vector = genVector();
// my_vector = {0, 1, 2, 3}

Vector#(3, Integer) my_vector1 = genVector();
// my_vector1 = {5, 6, 7, 8}

let my_vector2 = append(my_vector, my_vector1);
// my_vector2 = {0, 1, 2, 3, 5, 6, 7, 8}
```

Obtain a vector, `my_vector`, from a two dimensions vector, `matrix`.

```
Vector#(3, Vector#(3, Integer)) matrix;
for (Integer i = 0; i < 3; i = i + 1)
matrix[i] = genVector;

// matrix[0] = {0, 1, 2}
// matrix[1] = {3, 4, 5}
// matrix[2] = {6, 7, 8}

let my_vector = concat (matrix);
// my_vector = {0, 1, 2, 3, 4, 5, 6, 7, 8}
```

C.2.2 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 or update an element within it. Extracts or updates the ith element, where the first element is [0]. Index i must be of an indexable type; (e.g. `Integer`, `Bit#(n)`, `Int#(n)` or `UInt#(n)`.). The square-bracket notation for vectors can also be used with register writes.
	`anyVector[i];` `anyVector[i] = newValue;`

`select`	The select function is another form of the subscript notation ([i]), mainly provided for backwards-compatibility. The select function is also useful as an argument to higher-order functions. The subscript notation is generally recommended because it will report a more useful position for any selection errors.
	`function element_type` `      select(Vector#(vsize,element_type) vect, idx_type index);`

`update`	Update an element in a vector returning a new vector with one element changed/updated. This function does not change the given vector. This is another form of the subscript notation (see above), mainly provided for backwards compatibility. The update function may also be useful as an argument to a higher-order function. The subscript notation is generally recommended because it will report a more useful position for any update errors.

	`function Vector#(vsize, element_type)`
	`      update(Vector#(vsize, element_type) vectIn,`
	`             idx_type index,`
	`             element_type newElem);`

`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 highest (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));`

<table>
<tr><td><code>take</code></td><td>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.</td></tr>
<tr><td></td><td><pre>function Vector#(vxize2,element_type)
      take (Vector#(vsize,element_type) vect)
  provisos (Add#(vsize2,xxx,vsize)); // vsize2 <= vsize.</pre></td></tr>
</table>

<table>
<tr><td><code>drop</code>
<code>takeTail</code></td><td>Drop a number of elements from the vector starting at the 0th position. The elements in the result vector will be in the same order as the input vector.</td></tr>
<tr><td></td><td><pre>function Vector#(vxize2,element_type)
      drop (Vector#(vsize,element_type) vect)
  provisos (Add#(vsize2,xxx,vsize)); // vsize2 <= vsize.

function Vector#(vxize2,element_type)
      takeTail (Vector#(vsize,element_type) vect)
  provisos (Add#(vsize2,xxx,vsize)); // vsize2 <= vsize.</pre></td></tr>
</table>

<table>
<tr><td><code>takeAt</code></td><td>Take a number of elements starting at <code>startPos</code>. <code>startPos</code> must be a compile-time constant. If the <code>startPos</code> and vector size cause the function to go past the end of the vector, an error will be returned.</td></tr>
<tr><td></td><td><pre>function Vector#(vsize2,element_type)
       takeAt (Integer startPos, Vector#(vsize,element_type) vect)
  provisos (Add#(vsize2,xxx,vsize)); // vsize2 <= vsize</pre></td></tr>
</table>

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 or [ ] will generate a MUX

newvalue = select (my_vector, index);
newvalue = myvalue[index];
// newvalue = 9
```

Update the element of a vector, `my_vector`, at the position of index.

```
// my_vector is a vector of elements {6,7,8,9,10,11}
// index = 3

my_vector = update (my_vector, index, 0);
my_vector[index] = 0;
// my_vector = {6,7,8,0,10,11}
```

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
```

Extract the last element of the vector `my_vector`.

```
// my_vector is a vector of elements {6,7,8,9,10,11}

newvalue = last(my_vector);
// newvalue = 11
```

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 vector, `my_vector2`, of size 4 by removing the tail (last) 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 = init (my_vector1);
// my_vector2 is a vector of 4 elements {0,1,2,3}
```

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}
```

C.2.3 Vector to Vector Functions

The following functions generate a new vector by changing the position of elements within the vector.

`rotate`	Move the zeroth element to the highest and shift each element lower by one. For example, the element at index `n` moves to index `n-1`.
	`function Vector#(vsize,element_type)` `      rotate (Vector#(vsize,element_type) vect);`

`rotateR`	Move last element to the zeroth element and shift each element up by one. For example, the element at index `n` moves to index `n+1`.
	`function Vector#(vsize,element_type)` `      rotateR (Vector#(vsize,element_type) vect);`

`rotateBy`	Shift each element `n` places. The last `n` elements are moved to the begining, the element at index `0` moves to index `n`, index `1` to index `n+1`, etc.

	`function Vector#(vsize, element_type)`
	`      rotateBy (Vector#(vsize,element_type) vect, UInt#(log(v)) n)`
	`  provisos (Log#(vsize, logv);`

`shiftInAt0`	Shift a new element into the vector at index 0, bumping the index of all other element up by one. The highest 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 the index of 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 input vector rotated by 2 places.

```
// my_vector1 is a vector of elements {1,2,3,4,5}

my_vector2 = rotateBy {my_vector1, 2};
// my_vector2 = {4,5,1,2,3}
```

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}}
```

C.2.4 Tests on Vectors

The following functions are used to test vectors. The first three functions 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));`

```
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 );
```

The following two functions return the number of elements in the vector which match a condition.

`countElem`	Returns the number of elements in the vector which are equal to a given value. The return value is in the range of 0 to vsize.

```
function UInt#(logv1) countElem (element_type x,
                                 Vector#(vsize, element_type) vect)
  provisos (Eq#(element_type), Add#(vsize, 1, vsize1),
            Log#(vsize1, logv1));
```

`countIf`	Returns the number of elements in the vector which satisfy a given predicate function. The return value is in the range of 0 to vsize.

```
function UInt#(logv1) countIf (function Bool pred(element_type x1)
                               Vector#(vsize, element_type) vect)
  provisos (Add#(vsize, 1, vsize1), Log#(vsize1, logv1));
```

The following two functions return the index of an element.

`findElem`	Returns the index of the first element in the vector which equals a given value. Returns an `Invalid` if not found or `Valid` with a value of 0 to vsize-1 if found.

```
function Maybe#(UInt#(logv)) findElem (element_type x,
                             Vector#(vsize, element_type) vect)
  provisos (Eq#(element_type), Add#(xx1, 1, vsize),
            Log#(vsize, logv));
```

<table>
<tr><td rowspan="2"><code>findIndex</code></td><td>Returns the index of the first element in the vector which satisfies a given predicate. Returns an <code>Invalid</code> if not found or <code>Valid</code> with a value of 0 to vsize-1 if found.</td></tr>
<tr><td><pre>function Maybe#(UInt#(logv)) findIndex
                          (function Bool pred(element_type x1)
                           Vector#(vsize, element_type) vect)
  provisos (Add#(xx1,1,vsize), Log#(vsize, logv));</pre></td></tr>
</table>

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(isPositive, 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");
```

Count the number of elements which match the integer provided.

```
// my_vector1 is a vector of {1,2,1,4,3}
x = countElem ( 1, my_vector1);
// x = 2
y = countElem (4, my_vector1);
// y = 1
```

Find the index of an element which equals a predicate.

```
let f = findIndex ( beIsGreaterThan( 3 ) , my_vector );
 if ( f matches tagged Valid .indx )
    begin
       printBE ( my_vector[indx] ) ;
       $display ("Found data > 3 at index %d ", indx ) ;
 else
    begin
       $display ( "Did not find data > 3" ) ;
    end
```

C.2.5 Bit-Vector Functions

The following functions operate on bit-vectors.

`rotateBitsBy`	Shift each bit to a higher index by `n` places. The last `n` bits are moved to the begininng and the bit at index `(0)` moves to index `(n)`.
	`function Bit#(n) rotateBitsBy (Bit#(n) bvect, UInt#(logn) n)` `  provisos (Log#(n,logn), Add#(1,xxx,n));`

`countOnesAlt`	Returns the number of elements equal to 1 in a bit-vector. (This function differs slightly from the Prelude version of countOnes and has fewer provisos.)
	`function UInt#(logn1) countOnesAlt (Bit#(n) bvect)` `  provisos (Add#(1,n,n1), Log#(n1,logn1));`

C.2.6 Functions on Vectors of Registers

`readVReg`	Returns the values from reading a vector of registers (interfaces).
	`function Vector#(n,a) readVReg ( Vector#(n, Reg#(a)) vrin) ;`

`writeVReg`	Returns an Action which is the write of all registers in `vr` with the data from `vdin`.
	`function Action writeVReg ( Vector#(n, Reg#(a)) vr,` `                            Vector#(n,a) vdin) ;`

C.2.7 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.

`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})
```


C.2.8 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 `extend` function to each element of `avector` into a new vector, `resultvector`.

```
Vector#(13,Bit#(5))   avector;
Vector#(13,Bit#(10))  resultvector;
...
resultvector = map( extend, avector ) ;
```

This is equivalent to saying:

```
for (Integer i=0; i<13; i=i+1)
    resultvector[i] = extend(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}
```

C.2.9 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 );`

<table>
<tr><td rowspan="2"><code>zipWithAny</code></td><td>Combine two vectors with a function; result is as long as the smaller vector.</td></tr>
<tr><td><pre>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));</pre></td></tr>
</table>

<table>
<tr><td rowspan="2"><code>zipWith3</code></td><td>Combine three vectors with a function.</td></tr>
<tr><td><pre>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);</pre></td></tr>
</table>

<table>
<tr><td rowspan="2"><code>zipWithAny3</code></td><td>Combine three vectors with a function; result is as long as the smallest vector.</td></tr>
<tr><td><pre>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));</pre></td></tr>
</table>

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];
```

C.2.10 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(function b_type 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 (function b_type 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(`
	`         function element_type 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 (`
	`         function element_type 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 $O(vsize)$.

<table>
<tr><td rowspan="2"><code>fold</code></td><td>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.</td></tr>
<tr><td><pre>function element_type fold (
         function element_type func(element_type y, element_type x),
         Vector#(vsize,element_type) vect)
  provisos (Add#(1, xxx, vsize));</pre></td></tr>
</table>

<table>
<tr><td rowspan="2"><code>mapPairs</code></td><td>Map a function over a vector consuming two elements at a time. Any straggling element is processed by the second function.</td></tr>
<tr><td><pre>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));</pre></td></tr>
</table>

<table>
<tr><td rowspan="2"><code>joinActions</code></td><td>Join a number of actions together. <code>joinActions</code> is used for static elaboration only, no hardware is generated.</td></tr>
<tr><td><pre>function Action joinActions (Vector#(vsize,Action) vactions);</pre></td></tr>
</table>

<table>
<tr><td rowspan="2"><code>joinRules</code></td><td>Join a number of rules together.<code>joinRules</code> is used for static elaboration only, no hardware is generated.</td></tr>
<tr><td><pre>function Rules joinRules (Vector#(vsize,Rules) vrules);</pre></td></tr>
</table>

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
// make sure you leave a space after the \+ and before the ,

// 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 (\+ , 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
```

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.2.11 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

$$\begin{aligned}
W_n &= init; \\
W_{n-1} &= func(V_{n-1}, W_n); \\
W_{n-2} &= func(V_{n-2}, W_{n-1}); \\
&\dots \\
W_1 &= func(V_1, W_2); \\
W_0 &= func(V_0, W_1);
\end{aligned}$$

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));`

```
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 );`

```
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 n^{th} element.

$$\begin{aligned} W_0 &= init; \\ W_1 &= func(W_0, V_0); \\ W_2 &= func(W_1, V_1); \\ &... \\ W_{n-1} &= func(W_{n-2}, V_{n-2}); \\ W_n &= func(W_{n-1}, V_{n-1}); \end{aligned}$$

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 n^{th} 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));`

```
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 zeroth element up to the n^{th} element. The first result, *init*, is dropped, shifting the result index up 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_type) vect );`

```
function Vector#(vsize,a_type)
        sscanl(function a_type func(a_type x1, b_type x2),
               a_type q,
               Vector#(vsize, b_type) vect );
```

`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 );`

```
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 );`

```
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 );
```

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 (extend (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 $n^{th}$ 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.
```

C.2.12 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.

<table>
<tr><td rowspan="2"><code>mapM</code></td><td>Takes a monadic function and a vector, and applies the function to all vector elements returning the vector of corresponding results.</td></tr>
<tr><td><pre>function m#(Vector#(vsize, b_type))
         mapM (function m#(b_type) func(a_type x),
                 Vector#(vsize, a_type) vecta)
   provisos (Monad#(m));</pre></td></tr>
</table>

<table>
<tr><td rowspan="2"><code>mapM_</code></td><td>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.</td></tr>
<tr><td><pre>function m#(void) mapM_(function m#(b_type) func(a_type x),
                        Vector#(vsize, a_type) vect)
  provisos (Monad#(m));</pre></td></tr>
</table>

<table>
<tr><td rowspan="2"><code>zipWithM</code></td><td>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.</td></tr>
<tr><td><pre>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));</pre></td></tr>
</table>

<table>
<tr><td rowspan="2"><code>zipWithM_</code></td><td>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 <code>zipWithM</code> but the resulting vector is thrown away leaving the action in its context.</td></tr>
<tr><td><pre>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));</pre></td></tr>
</table>

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
```

C.2.13 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);
```

C.2.14 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 Vector documentation (C.2) for details. If the implementation requires 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.3 Aggregation: Lists

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.

C.3.1 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}
```

C.3.2 Extracting Elements and Sub-Lists

`[i]`	The square-bracket notation is available to extract an element from a list or update an element within it. Extracts or updates the ith element, where the first element is [0]. Index i must be of an indexable type; (e.g. `Integer`, `Bit#(n)`, `Int#(n)` or `UInt#(n)`.). The square-bracket notation for lists can also be used with register writes.
	`anyList[i];` `anyList[i] = newValue;`

`select`	The select function is another form of the subscript notation ([i]), mainly provided for backwards-compatibility. The select function is also useful as an argument to higher-order functions. The subscript notation is generally recommended because it will report a more useful position for any selection errors.
	`function element_type` `      select(List#(element_type) alist, idx_type index);`

update	Update an element in a list returning a new list with one element changed/updated. This function does not change the given list. This is another form of the subscript notation (see above), mainly provided for backwards compatibility. The update function may also be useful as an argument to a higher-order function. The subscript notation is generally recommended because it will report a more useful position for any update errors.

```
function List#(element_type)
      update(List#(element_type) alist,
                   idx_type index,
                   element_type newElem);
```

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}
```

C.3.3 List to List Functions

`rotate`	Move the first element to the last and shift each element to the next higher index.
	`function List#(element_type) rotate (List#(element_type) alist);`

`rotateR`	Move last element to the beginning and shift each element to the next lower index.
	`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}}
```

C.3.4 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 eqt Test
if (elem(5,my_list))
   $display ("List contains the integer 5");
```

C.3.5 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.

<table>
<tr><td rowspan="2"><code>zip</code></td><td>Combine two lists into a list of Tuples.</td></tr>
<tr><td><pre>function List#(Tuple2 #(a_type, b_type))
         zip(List#(a_type) lista,
              List#(b_type) listb);</pre></td></tr>
</table>

<table>
<tr><td rowspan="2"><code>zip3</code></td><td>Combine 3 lists into a list of Tuple3.</td></tr>
<tr><td><pre>function List#(Tuple3 #(a_type, b_type, c_type))
         zip3(List#(a_type) lista,
               List#(b_type) listb,
               List#(c_type) listc);</pre></td></tr>
</table>

<table>
<tr><td rowspan="2"><code>zip4</code></td><td>Combine 4 lists into a list of Tuple4.</td></tr>
<tr><td><pre>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);</pre></td></tr>
</table>

<table>
<tr><td rowspan="2"><code>unzip</code></td><td>Separate a list of pairs (i.e. a <code>Tuple2#(a,b)</code>) into a pair of two lists.</td></tr>
<tr><td><pre>function Tuple2#(List#(a_type), List#(b_type))
         unzip(List#(Tuple2 #(a_type, b_type)) listab);</pre></td></tr>
</table>

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})
```

C.3.6 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 `extend` function to each element of `alist` creating a new list, `resultlist`.

```
List#(Bit#(5))   alist;
List#(Bit#(10))  resultlist;
...
resultlist = map( extend, alist ) ;
```

This is equivalent to saying:

```
for (Integer i=0; i<13; i=i+1)
    resultlist[i] = extend(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}
```

C.3.7 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];
```

C.3.8 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);`

```
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 );
```

`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);`

<table>
<tr><td rowspan="2"><code>mapPairs</code></td><td>Map a function over a list consuming two elements at a time. Any straggling element is processed by the second function.</td></tr>
<tr><td><pre>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);</pre></td></tr>
</table>

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
```

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.3.9 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.

$$\begin{aligned} W_0 &= init; \\ W_1 &= func(W_0, L_0); \end{aligned}$$

$$
\begin{aligned}
W_2 &= func(W_1, L_1);\\
&...\\
W_{n-1} &= func(W_{n-2}, L_{n-2});\\
W_n &= func(W_{n-1}, L_{n-1});
\end{aligned}
$$

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);`

```
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 );
```

`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 );
```

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 (extend (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}
```

C.3.10 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));
```

C.4 Math

C.4.1 Real

Package Name

import Real :: * ;

Description

The `Real` library package defines functions to operate on and manipulate real numbers. Real numbers are numbers with a fractional component. They are also of limited precision. The `Real` data type is described in section B.2.6.

Constants

The constant `pi` (π) is defined.

pi	The value of the constant pi (π).

```
Real pi;
```

Trigonometric Functions

The following trigonometric functions are provided: `sin`, `cos`, `tan`, `sinh`, `cosh`, `tanh`, `asin`, `acos`, `atan`, `asinh`, `acosh`, `atanh`, and `atan2`.

sin	Returns the sine of `x`.

```
function Real sin (Real x);
```

`cos`	Returns the cosine of `x`.	
	`function Real cos (Real x);`	

`tan`	Returns the tangent of `x`.
	`function Real tan (Real x);`

`sinh`	Returns the hyperbolic sine of `x`.
	`function Real sinh (Real x);`

`cosh`	Returns the hyperbolic cosine of `x`.
	`function Real cosh (Real x);`

`tanh`	Returns the hyperbolic tangent of `x`.
	`function Real tanh (Real x);`

`asinh`	Returns the inverse hyperbolic sine of `x`.
	`function Real asinh (Real x);`

`acosh`	Returns the inverse hyperbolic cosine of `x`.
	`function Real acosh (Real x);`

`atanh`	Returns the inverse hyperbolic tangent of `x`.	
	`function Real atanh (Real x);`	

`atan2`	Returns `atan(`x/y`)`. `atan2(1,x)` is equivalent to `atan(x)`, but provides more precision when required by the division of `x/y`.
	`function Real atan2 (Real y, Real x);`

Arithmetic Functions

`pow`	The element x is raised to the y power. An alias for `**`. `pow(x,y)` = `x**y` = x^y.
	`function Real pow (Real x, Real y);`

`sqrt`	Returns the square root of `x`. Returns an error if `x` is negative.
	`function Real sqrt (Real x);`

Conversion Functions

The following four functions are used to convert a `Real` to an `Integer`.

`trunc`	Converts a `Real` to an `Integer` by removing the fractional part of `x`, which can be positive or negative. `trunc(1.1) = 1`, `trunc(-1.1)= -1`.
	`function Integer trunc (Real x);`

`round`	Converts a `Real` to an `Integer` by rounding to the nearest whole number. `.5` rounds up in magnitude. `round(1.5) = 2`, `round(-1.5)= -2`.
	`function Integer round (Real x);`

`ceil`	Converts a `Real` to an `Integer` by rounding to the higher number, regardless of sign. `ceil(1.1) = 2`, `ceil(-1.1) = -1`.	`function Integer ceil (Real x);`

`floor`	Converts a `Real` to an `Integer` by rounding to the lower number, regardless of sign. `floor(1.1) = 1`, `floor(-1.1) = -2`.	`function Integer floor (Real x);`

There are also two system functions `$realtobits` and `$bitstoreal`, defined in the Prelude (section B.2.6) which provide conversion to and from IEEE 64-bit vectors (`Bit#(64)`).

Introspection Functions

`isInfinite`	Returns `True` if the value of `x` is infinite, `False` if `x` is finite.	`function Bool isInfinite (Real x);`

`isNegativeZero`	Returns `True` if the value of `x` is negative zero.	`function Bool isNegativeZero (Real x);`

`splitReal`	Returns a Tuple containing the whole (n) and fractional (f) parts of `x` such that $n + f = x$. Both values have the same sign as `x`. The absolute value of the fractional part is guaranteed to be in the range [0,1).	`function Tuple2#(Integer, Real) splitReal (Real x);`

`decodeReal`	Returns a Tuple3 containing the sign, the fraction, and the exponent of a real number. The `Bool` represents the sign and is `True` for positive and `False` for negative. The second part (the first Integer) represents the fractional part as a signed Integer value. This can be converted to an `Int#(54)` (52 bits, plus hidden plus sign). The last value is a signed Integer representing the exponent, which can be be converted to an `Int#(12)` . The real number is represented exactly as ($fractional \times 2^{exp}$).	`function Tuple3#(Bool, Integer, Integer) decodeReal (Real x);`

C.4.2 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`. The remaining arithmetic operators are not defined for the `Complex` type. Note however, that some functions generate more hardware than may be expected. 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. The complex division operator (/) produces 6 multipliers, and a divider and may not always be synthesizable with downstream tools.

```
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 ) ;`

<table>
<tr><td rowspan="2"><code>cmplxMap</code></td><td>Applies a function to each part of the complex structure. This is useful for operations such as <code>extend</code>, <code>truncate</code>, etc.</td></tr>
<tr><td><pre>function Complex#(b_type) cmplxMap(
                          function b_type mapFunc(a_type x),
                          Complex#(a_type) cin) ;</pre></td></tr>
</table>

<table>
<tr><td rowspan="2"><code>cmplxSwap</code></td><td>Exchanges the real and imaginary parts.</td></tr>
<tr><td><pre>function Complex#(a_type) cmplxSwap(Complex#(a_type) cin) ;</pre></td></tr>
</table>

<table>
<tr><td rowspan="2"><code>cmplxWrite</code></td><td>Displays a complex number given a prefix string, an infix string, a postscript string, and an Action function which writes each part. <code>cmplxWrite</code> is of type Action and can only be invoked in Action contexts such as Rules and Actions methods.</td></tr>
<tr><td><pre>function Action cmplxWrite(String pre,
                           String infix,
                           String post,
                           function Action writeaFunc(a_type x),
                           Complex#(a_type) cin);</pre></td></tr>
</table>

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.4.3 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; `Bits`, `Eq`, `Literal`, `RealLiteral`, `Arith`, `Ord`, `Bounded`, 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	Real Literal	Arith	Ord	Bounded	Bit wise	Bit Reduce	Bit Extend
FixedPoint	√	√	√	√	√	√	√	√		

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#(isize, fsize) )
   provisos( Add#(isize, fsize, TAdd#(isize,fsize) ),
             Add#(1, xxx, isize) ) ;       //   isize >= 1
```

RealLiteral The type `FixedPoint` belongs to the `RealLiteral` type class, which allows conversion from type `Real` to type `FixedPoint`.

Example:

```
FixedPoint#(3,10) p = 3.14159;
```

```
instance RealLiteral#( FixedPoint# (isize, fsize) )
   provisos( Add#(isize, fsize, TAdd#(isize,fsize) ),
             Add#(1, xxx, isize) ) ;       //   isize >= 1
```

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`. The arithmetic operators `/` and `%` are not defined.

```
instance Arith#( FixedPoint#(isize, fsize) )
   provisos( Add#(isize, fsize, TAdd#(isize,fsize) ),
             Add#(1, xxx, isize) ) ;       //   isize >= 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#(isize, fsize) )
   provisos( Add#(1, xxx, isize) ) ;       //   isize >= 1
```

Bounded The type `FixedPoint` belongs to the `Bounded` type class. The range of values, v, representable with a signed fixed-point number of type `FixedPoint#(isize, fsize)` is $+(2^{isize-1} - 2^{-fsize}) \leq v \leq -2^{isize-1}$. 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 1.875 ($1\frac{7}{8}$) to -2.0 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#(isize, fsize) ) ;
   provisos( Add#(1, xxx, isize) ) ;       //   isize >= 1
```

`epsilon`	Returns the value of `epsilon` which is the smallest representable quantum by a specific type, 2^{-fsize}.
	`function FixedPoint#(isize, fsize) epsilon () ;`

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#(isize, fsize) )
   provisos( Add#(1, xxx, isize) ) ;       //   isize >= 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)`.

fromRational — Specify a `FixedPoint` with a rational number which is the ratio of two integers.

```
function FixedPoint#(isize, fsize) fromRational(
                Integer numerator, Integer denominator)
   provisos ( Add#(1, xxA, isize ) ) ;          // isize >= 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#(isize, fsize) 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("" ) ;
```

A `Real` value can automatically be converted to a `FixedPoint` value:

```
FixedPoint#(3,10) foo = 2e-3;

FixedPoint#(2,3) x = 1.625 ;
```

C.4.4 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.5 FSM

C.5.1 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*
	|	*returnFsmStmt*
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
returnFsmStmt	::=	`return ;`
repeatFsmStmt	::=	`repeat (` *expression* `)`
		loopBodyFsmStmt
loopBodyFsmStmt	::=	*fsmStmt*
	|	`break ;`
	|	`continue ;`

The simplest kind of statement is an *exprFsmStmt*, which can be a register assignment or, more generally, any expression of type `Action` (including action method calls and `action-endaction` blocks or of type `Stmt`. Statements of type `Action` execute within exactly one clock cycle, but of course the scheduling semantics 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 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 evaluates true , the loop body is executed, so the total time is $n \times t$ 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:

`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.

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. The user must not use `waitTillDone` until after the FSM has been started because the FSM comes out of a reset as 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 into 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 `mkFSMWithPred` module is like `mkFSM` above, except that the module constructor takes an additional boolean argument (the predicate). The predicate condition is added to the condition of each rule generated to create the FSM. This capability is useful when using the FSM in conjuction with other rules and/or FSMs. It allows the designer to explicitly specify to the compiler the conditions under which the FSM will run. This can be used to eliminate spurious rule conflict warnings (between rules in the FSM and other rules in the design).

```
module mkFSMWithPred#( Stmt seq_stmt, Bool pred ) ( FSM );
```

The `mkAutoFSM` module is also 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.
mkFSMWithPred	`module mkFSMWithPred#(Stmt seq_stmt, Bool pred)(FSM);`	Like mkFSM, except that the module constructor takes an additional predicate condition as an argument. The predicate condition is added to the condition of each rule generated to create the 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 are two functions, await and delay, provided by the StmtFSM package.

The await function 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.

The delay function is used to execute noAction for a specified number of cycles. The function is provided the value of the delay and returns a Stmt.

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.
delay	`function Stmt delay( a_type value ) ;`	Creates a Stmt which executes noAction for value number of cycles. a_type must be in the Arith class and Bits class and $<$ 32 bits.

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;  // same as: delay(2);
```



```
      device_xcvr_speed_reg <= new_speed;
      noAction; noAction;  // same as: delay(2);

      outofReset_reg <= True;
      if (device_xcvr_speed_reg==HS)
         seq noAction; noAction; endseq
         // or seq delay(2); endseq
      else
         seq noAction; noAction; noAction; noAction; noAction; noAction; endseq
        // or seq delay(6); 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
```

Creating FSM Server Modules

Instantiation of an FSM server module is performed in a manner analogous to that of a standard FSM module constructor (such as `mkFSM`). Whereas `mkFSM` takes a `Stmt` value as an argument, howver, `mkFSMServer` takes a function as an argument. More specifically, the argument to `mkFSMServer` is a function which takes an argument of type `a` and returns a value of type `RStmt#(b)`.

```
    module mkFSMServer#(function RStmt#(b) seq_func (a input)) ( FSMServer#(a, b) );
```

The `RStmt` type is a polymorphic generalization of the `Stmt` type. A sequence of type `RStmt#(a)` allows valued `return` statements (where the return value is of type `a`). Note that the `Stmt` type is equivalent to `RStmt#(Bit#(0))`.

```
typedef RStmt#(Bit#(0)) Stmt;
```

The `mkFSMServer` module constructor provides an interface of type `FSMServer#(a, b)`.

```
interface FSMServer#(type a, type b);;
    interface Server#(a, b) server;
    method Action abort();
endinterface
```

The `FSMServer` interface has one subinterface of type `Server#(a, b)` (from the `ClientServer` package) as well as an `Action` method called `abort`; The `abort` method allows the FSM inside the `FSMServer` module to be halted if the client FSM is halted.

An `FSMServer` module is accessed using the `callServer` function. `callServer` takes two arguments. The first is the interface of the `FSMServer` module. The second is the input value being passed to the module.


```
result <- callServer(serv_ifc, value);
```

Note the special left arrow notation that is used pass the server result to a register (or more generally to any state element with a `Reg` interface). A simple example follows showing the definition and use of a `mkFSMServer` module.

Example - Defining and instantiating an FSM Server Module

```
// State elements to provide inputs and store results
Reg#(Bit#(8))  count   <- mkReg(0);
Reg#(Bit#(16)) partial <- mkReg(0);
Reg#(Bit#(16)) result  <- mkReg(0);

// A function which creates a server sequence to scale a Bit#(8)
// input value by and integer scale factor. The scaling is accomplished
// by a sequence of adds.
function RStmt#(Bit#(16)) scaleSeq (Integer scale, Bit#(8) value);
   seq
      partial <= 0;
      repeat (fromInteger(scale))
         action
            partial <= partial + {0,value};
         endaction
      return partial;
   endseq;
endfunction

// Instantiate a server module to scale the input value by 3
FSMServer#(Bit#(8), Bit#(16)) scale3_serv  <- mkFSMServer(scaleSeq(3));

// A test sequence to apply the server
let test_seq =  seq
                   result <- callServer(scale3_serv, count);
                   count <= count + 1;
                endseq;

let test_fsm <- mkFSM(test_seq);

// A rule to start test_fsm
rule start;
   test_fsm.start;
endrule
// finish after 6 input values
rule done (count == 6);
   $finish;
endrule
```

C.6 Connectivity

The packages in this section provide useful components, primarily interfaces, 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 package `CGetPut` defines a type of the `Get` and `Put` interfaces that is implemented with a credit based FIFO.

C.6.1 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.

Typeclasses

The `GetPut` package defines two typeclasses; `ToGet` and `ToPut`.

`ToGet` defines the class to which the function `toGet` can be applied to create an associated `Get` interface.

```
typeclass ToGet#(a, b);
   function Get#(b) toGet(a ax);
endtypeclass
```

`ToPut` defines the class to which the function `toPut` can be applied to create an associated `Put` interface.

```
typeclass ToPut#(a, b);
   function Put#(b) toPut(a ax);
endtypeclass
```

Instances of `ToGet` and `ToPut` are defined for the following interfaces. The `toGet` and `toPut` functions convert these interfaces to a `Get` and `Put` interface respectively.

```
FIFO
FIFOF
SyncFIFOIfc
FIFOLevelIfc
SyncFIFOLevelIfc
FIFOCountIfc
SyncFIFOCountIfc
```

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 element_type);
    method Action put(element_type 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.6.2) 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#(element_type), Put#(element_type));
```

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 functions `toGet` and `toPut` (C.6.1) are recommended instead of the `fifoToGet` and `fifoToPut` functions.

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. It is recommended that you use the function `toGet` (C.6.1) instead of this function.
	`function Get#(element_type) fifoToGet(FIFO#(element_type) f);`

`fifoToPut`	Returns a `Put` interface. It is recommended that you use the function `toPut` (C.6.1) instead of this function.
	`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.6.2 Connectable

Package Name

import Connectable :: * ;

Description

The `Connectable` package contains the definitions for the class `Connectable` and two instances of `Connectables`; `Tuple`s and `Vector`s.

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
```

Instances

Get and Put One instance of the typeclass of `Connectable` is `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));
```


Tuples 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.6.3) to connect the `Get` of the `Client` to the `Put` of the `Server` and visa-versa.

This is extensible to all Tuples (`Tuple3`, `Tuple4`, etc.). As long as the items are connectable, the Tuples are connectable.

Vector Two `Vector`s are connectable if their elements are connectable.

```
instance Connectable#(Vector#(n, a), Vector#(n, b))
  provisos (Connectable#(a, b));
```

InOut `Inout`s are connectable via the `Connectable` typeclass. The use of `mkConnection` instantiates a Verilog module `InoutConnect`. The `Inout`s must be on the same clock and the same reset. The clock and reset of the `Inout`s may be different than the clock and reset of the parent module of the `mkConnection`.

```
instance Connectable#(Inout#(a, x1), Inout#(a, x2))
   provisos (Bit#(a,sa));
```

C.6.3 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 Bus_Ifc;
   interface Server#(RQ, RS) to_targ ;
   interface Client#(RQ, RS) to_initor;
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 ifc, targ is a Put ifc)
   Empty x <- mkConnection (bus.to_targ, targ);

   // Connect bus and initiator ("to_initor" is a Out ifc, initor is a Get ifc)
   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.6.4 CGetPut

Package Name

import CGetPut :: * ;

Description

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).

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.

Interfaces and methods

The interface types are abstract to avoid any non-proper use of the credit signaling protocol.

Interfaces			
Interface Name	Parameter name	Parameter Description	Restrictions
`CGet`	n	depth of the buffer used in the receiving interface	must be a numeric type
	element_type	type of the element being retrieved by the `CGet`	must be in `Bits` class
`CPut`	n	depth of the buffer used in the receiving interface	must be a numeric type
	element_type	type of the element being added by the `CPut`	must be in `Bits` class

- `CGet`

```
interface CGet#(numeric type n, type element_type);
      ...Abstract...
```

- CPut

```
interface CPut#(numeric type n, type element_type);
      ...Abstract...
```

- Connectables

  The CGet and CPut interfaces are connectable.

```
instance Connectable#(CGet#(n, element_type), CPut#(n, element_type));

instance Connectable#(CPut#(n, element_type), CGet#(n, element_type));
```

- CClient and CServer

  The same idea may be extended to clients and servers.

```
interface CClient#(type n, type req_type, type resp_type);
interface CServer#(type n, type req_type, type resp_type);
```

Modules

mkCGetPut	Create an n depth FIFO with a CGet interface on the dequeue side and a Put interface on the enqueue side.
	`module mkCGetPut(Tuple2#(CGet#(n, element_type), Put#(element_type))) provisos (Bits#(element_type));`

mkGetCPut	Create an n depth FIFO with a Get interface on the dequeue side and a CPut interface on the enqueue side.
	`module mkGetCPut(Tuple2#(Get#(element_type), CPut#(n, element_type))) provisos (Bits#(element_type));`

mkClientCServer	Create a CServer with a mkCGetPut and a mkGetCPut. Provides a CServer interface and a regular Client interface.
	`module mkClientCServer( Tuple2#(Client#(req_type, resp_type), CServer#(n, req_type, resp_type))) provisos (Bits#(req_type), Bits#(resp_type));`

`mkCClientServer`	Create a `CClient` with a `mkCGetPut` and a `mkGetCPut`. Provides a `CClient` interface and a regular `Server` interface.	
	`module mkCClientServer( Tuple2#(CClient#(n, req_type, resp_type), Server#(req_type, resp_type))) provisos (Bits#(req_type), Bits#(resp_type));`	

C.7 Utilities

C.7.1 LFSR

Package

import LFSR :: * ;

Description

The LFSR package implements Linear Feedback Shift Registers (LFSRs). LFSRs can be used to obtain reasonable pseudo-random numbers for many purposes (though not good enough for cryptography). The `seed` method must be called first, to prime the algorithm. Then values may be read using the `value` method, and the algorithm stepped on to the next value by the `next` method. When a LFSR is created the start value, or seed, is 1.

Interfaces and Methods

The `LFSR` package provides an interface, `LFSR`, which contains three methods; `seed`, `value`, and `next`. To prime the LFSR the `seed` method is called with the parameter `seed_value`, of datatype `a_type`. The value is read with the `value` method. The `next` method is used to shift the register on to the next value.

LFSR Interface				
Method			Arguments	
Name	Type	Description	Name	Description
`seed`	`Action`	Sets the value of the shift register.	`a_type` `seed_value`	datatype of the seed value the initial value
`value`	`a_type`	returns the value of the shift register		
`next`	`Action`	signals the shift register to shift to the next value.		

```
interface LFSR #(type a_type);
    method Action seed(a_type seed_value);
    method a_type value();
    method Action next();
endinterface: LFSR
```

Modules

The module `mkFeedLFSR` creates a LFSR where the polynomial is specified by the mask used for feedback.

`mkFeedLFSR`	Creates a LFSR where the polynomial is specified by the mask (*feed*) used for feedback.	
	`module mkFeedLFSR#( Bit#(n) feed )( LFSR#(Bit#(n)) );`	

For example, the polynominal $x^7+x^3+x^2+x+1$ is defined by the expression `mkFeedLFSR#(8'b1000_1111)`

Using the module `mkFeedLFSR`, the following maximal length LFSR's are defined in this package.

Module Name	feed	Module Definition
`mkLFSR_4`	4'h9 x^3+1	`module mkLFSR_4 (LFSR#(Bit#(4)));`
`mkLFSR_8`	8'h8E	`module mkLFSR_8 (LFSR#(Bit#(8)));`
`mkLFSR_16`	16'h8016	`module mkLFSR_16 (LFSR#(Bit#(16)));`
`mkLFSR_32`	32'h80000057	`module mkLFSR_32 (LFSR#(Bit#(32)));`

For example,

```
mkLFSR_4   =  mkFeedLFSR( 4'h9 );
```

The module `mkLFSR_4` instantiates the interface `LFSR` with the value `Bit#(4)` to produce a 4 bit shift register. The module uses the polynomial defined by the mask 4'h9 (x^3+1) and the module `mkFeedLFSR`.

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.

`mkRCounter`	Creates a counter with a LFSR interface.
	`module mkRCounter#( Bit#(n) seed ) ( LFSR#(Bit#(n)) );`

Example - Random Number Generator

```
import GetPut::*;
import FIFO::*;
import LFSR::*;

//  We want 6-bit random numbers, so we will use the 16-bit version of
//  LFSR and take the most significant six bits.

//  The interface for the random number generator is parameterized on bit
//  length.  It is a "get" interface, defined in the GetPut package.

typedef Get#(Bit#(n)) RandI#(type n);

module mkRn_6(RandI#(6));
```



```
  // First we instantiate the LFSR module
  LFSR#(Bit#(16)) lfsr <- mkLFSR_16 ;

  // Next comes a FIFO for storing the results until needed
  FIFO#(Bit#(6)) fi <-  mkFIFO ;

  // A boolean flag for ensuring that we first seed the LFSR module
  Reg#(Bool) starting <- mkReg(True) ;

  // This rule fires first, and sends a suitable seed to the module.
  rule start (starting);
      starting <= False;
      lfsr.seed('h11);
  endrule: start

  // After that, the following rule runs as often as it can, retrieving
  // results from the LFSR module and enqueing them on the FIFO.
  rule run (!starting);
      fi.enq(lfsr.value[10:5]);
      lfsr.next;
  endrule: run

  // The interface for mkRn_6 is a Get interface.  We can produce this from a
  // FIFO using the fifoToGet function.  We therefore don't need to define any
  // new methods explicitly in this module: we can simply return the produced
  // Get interface as the "result" of this module instantiation.
  return fifoToGet(fi);
endmodule
```

C.7.2 Randomizable

Description

The Randomizable package includes interfaces and modules to generate random values of a given data type.

This package is provided as both a compiled library package and as BSV source code to facilitate customization. The source code file can be found in the `$BLUESPECDIR/BSVSource` directory. To customize a package, copy the file into a local directory and then include the local directory in the path when compiling. This is done by specifying the path with the `-p` option as described in the BSV Users Guide.

Packages

To include a package in your design, use the `import` syntax.

```
import Randomizable :: * ;
```

Interfaces and Methods

Randomize Interface		
Name	Type	Description
`cntrl`	`Interface`	Control interface provided by the module.
`next`	`ActionValue`	Returns the next value of type `a`.

```
interface Randomize#(type a);
   interface Control cntrl;
   method ActionValue#(a) next();
endinterface
```

Control Interface		
Name	Type	Description
`init`	`Control`	Action method to initialize the randomizer.

```
interface Control ;
   method Action init();
endinterface
```

Modules

The `Randomizable` package includes two modules which return random values of type `a`. The difference between the two modules is how the min and max values are determined. The module `mkGenericRandomizer` uses the min and max values of the type, while the module `mkConstrainedRandomizer` uses arguments to set the min and max values. The type `a` must be in the `Bounded` class for both modules.

`mkGenericRandomizer`	This module provides a `Randomize` interface, which will return the next random value when the `next` method is invoked. The `min` and `max` values are the values defined by the type `a` which must be in the `Bounded` class.
	`module mkGenericRandomizer (Randomize#(a))` `   provisos (Bits#(a, sa), Bounded#(a));`

`mkConstrainedRandomizer`	This module provides a `Randomize` interface, which will give the next random value when the `next` method is invoked. When instantiated, the `min` and `max` values are provided as arguments. Type `a` must be in the `Bounded` class.
	`module mkConstrainedRandomizer#(a min, a max) (Randomize#(a))` `   provisos (Bits#(a, sa), Bounded#(a));`

Example

The `mkTLMRandomizer` module, defined within the TLM package (Section C.10.1), uses the Randomize package to generate random values for TLM packets. The `mkConstrainedRandomizer` module is for fields with specific allowed values or ranges, while the `mkGenericRandomizer` module is for field where all values of the type are allowed.

```
module mkTLMRandomizer#(Maybe#(TLMCommand) m_command) (Randomize#(TLMRequest#(`TLM_TYPES)))
   provisos(Bits#(RequestDescriptor#(`TLM_TYPES), s0),
    Bounded#(RequestDescriptor#(`TLM_TYPES)),
    Bits#(RequestData#(`TLM_TYPES), s1),
```



```
 Bounded#(RequestData#(`TLM_TYPES))
 );

...
// Use mkGeneric Randomizer - entire range valid
Randomize#(RequestDescriptor#(`TLM_TYPES)) descriptor_gen <- mkGenericRandomizer;
Randomize#(Bit#(2))                        log_wrap_gen   <- mkGenericRandomizer;
Randomize#(RequestData#(`TLM_TYPES))       data_gen       <- mkGenericRandomizer;

// Use mkConstrainedRandomizer to Avoid UNKNOWN
Randomize#(TLMCommand) command_gen <- mkConstrainedRandomizer(READ, WRITE);
Randomize#(TLMBurstMode) burst_mode_gen <- mkConstrainedRandomizer(INCREMENT, WRAP);

// Use mkConstrainedRandomizer to set legal sizes between 1 and 16
Randomize#(TLMUInt#(`TLM_TYPES)) burst_length_gen <- mkConstrainedRandomizer(1,16);
```

C.7.3 Arbiter

Description

The Arbiter package includes interfaces and modules to implement two different arbiters: a fair arbiter with changing priorities (round robin) and a sticky arbiter, also round robin, but which gives the current owner priority.

This package is provided as both a compiled library package and as BSV source code to facilitate customization. The source code file can be found in the `$BLUESPECDIR/BSVSource` directory. To customize a package, copy the file into a local directory and then include the local directory in the path when compiling. This is done by specifying the path with the `-p` option as described in the BSV Users Guide.

Packages

To include a package in your design, use the `import` syntax.

```
import Arbiter :: * ;
```

Interfaces and Methods

The Arbiter package includes three interfaces: a arbiter client interface, an arbiter request interface and an arbiter interface which is a vector of client interfaces.

ArbiterClient_IFC The `ArbiterClient_IFC` interface has two methods: an Action method to make the request and a Boolean value method to indicate the request was granted. The lock method is unused in this implementation.

```
interface ArbiterClient_IFC;
   method Action request();
   method Action lock();
   method Bool grant();
endinterface
```

ArbiterRequest_IFC The `ArbiterRequest_IFC` interface has two methods: an Action method to grant the request and a Boolean value method to indicate there is a request. The lock method is unused in this implementation.

```
interface ArbiterRequest_IFC;
   method Bool request();
   method Bool lock();
   method Action grant();
endinterface
```

The `ArbiterClient_IFC` interface and the `ArbiterRequest_IFC` interface are connectable.

```
instance Connectable#(ArbiterClient_IFC,  ArbiterRequest_IFC);
```

Arbiter_IFC The `Arbiter_IFC` has a subinterface which is a vector of `ArbiterClient_IFC` interfaces. The number of items in the vector equals the number of clients.

```
interface Arbiter_IFC#(type count);
   interface Vector#(count, ArbiterClient_IFC) clients;
endinterface
```

Modules

The `mkArbiter` module is a fair arbiter with changing priorities (round robin). The `mkStickyArbiter` gives the current owner priority - they can hold priority as long as they keep requesting it. The modules all provide a `Arbiter_IFC` interface.

`mkArbiter`	This module is a fair arbiter with changing priorities (round robin). If `fixed` is `True`, the current client holds the priority, if `fixed` is `False`, it moves to the next client. `mkArbiter` provides a `Arbiter_IFC` interface. Initial priority is given to client 0.
	`module mkArbiter#(Bool fixed) (Arbiter_IFC#(count));`

`mkStickyArbiter`	As long as the client currently with the grant continues to assert `request`, it can hold the grant. It provides a `Arbiter_IFC` interface.
	`module mkStickyArbiter (Arbiter_IFC#(count));`

C.7.4 GrayCounter

Description

The GrayCounter package provides an interface and a module to implement a gray-coded counter with methods for both binary and Gray code. This package is designed for use in the `BRAMFIFO` module, Section C.1.9. Since BRAMs have registered address inputs, the binary outputs are not registered. The counter has two domains, source and destination. Binary and Gray code values are written in the source domain. Both types of values can be read from the source and the destination domains.

This package is provided as both a compiled library package and as BSV source code to facilitate customization. The source code file can be found in the `$BLUESPECDIR/BSVSource` directory. To customize a package, copy the file into a local directory and then include the local directory in the path when compiling. This is done by specifying the path with the `-p` option as described in the BSV Users Guide.

Package Name

To include a package in your design, use the `import` syntax.

```
import GrayCounter :: * ;
```

Types

The GrayCounter package uses the type `Gray`, defined in the Gray package, Section C.7.5. The Gray package is imported by the GrayCounter package.

Interfaces and Methods

The GrayCounter package includes one interface, `GrayCounter`.

GrayCounter Interface Methods		
Name	Type	Description
`incr`	Action	Increments the counter by 1
`decr`	Action	Decrements the counter by 1
`sWriteBin`	Action	Writes a binary value into the counter in the source domain.
`sReadBin`	`Bit#(n)`	Returns a binary value from the source domain of the counter. The output is not registered
`sWriteGray`	Action	Writes a Gray code value into the counter in the source domain.
`sReadGray`	`Gray#(n)`	Returns the Gray code value from the source domain of the counter. The output is registered.
`dReadBin`	`Bit#(n)`	Returns the binary value from the destination domain of the counter. The output is not registered.
`dReadGray`	`Gray#(n)`	Returns the Gray code value from the destination domain of the counter. The output is registered.

```
interface GrayCounter#(numeric type n);
   method    Action      incr;
   method    Action      decr;
   method    Action      sWriteBin(Bit#(n) value);
   method    Bit#(n)     sReadBin;
   method    Action      sWriteGray(Gray#(n) value);
   method    Gray#(n)    sReadGray;
   method    Bit#(n)     dReadBin;
   method    Gray#(n)    dReadGray;
endinterface: GrayCounter
```

Modules

The module `mkGrayCounter` instantiates a Gray code counter with methods for both binary and Gray code.

`mkGrayCounter`	Instantiates a Gray counter with an initial value `initval`.
	`module mkGrayCounter#(Gray#(n) initval, Clock dClk, Reset dRstN) (GrayCounter#(n)) provisos(Add#(1, msb, n));`

C.7.5 Gray

Description

The `Gray` package defines a datatype, `Gray` and functions for working with the Gray type. This type is used by the `GrayCounter` package.

This package is provided as both a compiled library package and as BSV source code to facilitate customization. The source code file can be found in the `$BLUESPECDIR/BSVSource` directory. To customize a package, copy the file into a local directory and then include the local directory in the path when compiling. This is done by specifying the path with the `-p` option as described in the BSV Users Guide.

Package Name

To include a package in your design, use the `import` syntax.

```
import Gray :: * ;
```

Types and type classes

The datatype `Gray` is a representation for Gray code values. The basic representation is the `Gray` structure, which is polymorphic on the size of the value.

```
typedef struct {
                Bit#(n) code;
               } Gray#(numeric type n) deriving (Bits, Eq);
```

Functions

`grayEncode`	This function takes a binary value of type `Bit#(n)` and returns a `Gray` type with the Gray code value.
	`function Gray#(n) grayEncode(Bit#(n) value) provisos(Add#(1, msb, n));`

`grayDecode`	This function takes a Gray code value of size `n` and returns the binary value.
	`function Bit#(n) grayDecode(Gray#(n) value) provisos(Add#(1, msb, n));`

`grayIncrDecr`	This functions takes a Gray code value and a Boolean, `decrement`. If `decrement` is True, the value returned is one less than the input value. If `decrement` is False, the value returned is one greater.	

	`function Gray#(n) grayIncrDecr(Bool decrement,`
	`Gray#(n) value)`
	`provisos(Add#(1, msb, n));`	

`grayIncr`	Takes a Gray code value and returns a Gray code value incremented by 1.
	`function Gray#(n) grayIncr(Gray#(n) value)` `provisos(Add#(1, msb, n));`

`grayDecr`	Takes a Gray code value a returns a Gray code value decremented by 1.
	`function Gray#(n) grayDecr(Gray#(n) value)` `provisos(Add#(1, msb, n));`

C.7.6 CompletionBuffer

Package

import CompletionBuffer :: * ;

Description

A `CompletionBuffer` is like a FIFO except that the order of the elements in the buffer is independent of the order in which the elements are entered. Each element obtains a token, which reserves a slot in the buffer. Once the element is ready to be entered into the buffer, the token is used to place the element in the correct position. When removing elements from the buffer, the elements are delivered in the order specified by the tokens, not in the order that the elements were written.

Completion Buffers are useful when multiple tasks are running, which may complete at different times, in any order. By using a completion buffer, the order in which the elements are placed in the buffer can be controlled, independent of the order in which the data becomes available.

Interface and Methods

The CompletionBuffer interface provides three subinterfaces. The `reserve` interface, a `Get`, allows the caller to reserve a slot in the buffer by returning a token holding the identity of the slot. When data is ready to be placed in the buffer, it is added to the buffer using the `complete` interface of type `Put`. This interface takes a pair of values as its argument - the token identifying its slot, and the data itself. Finally, using the `drain` interface, of type `Get`, data may be retrieved from the buffer in the order in which the tokens were originally allocated. Thus the results of quick tasks might have to wait in the buffer while a lengthy task ahead of them completes.

The type of the elements to be stored is `element_type`. The type of the required size of the buffer is a numeric type `n`, which is also the type argument for the type for the tokens issued, `CBToken`. This allows the type-checking phase of the synthesis to ensure that the tokens are the appropriate size for the buffer, and that all the buffer's internal registers are of the correct sizes as well.

CompletionBuffer Interface		
Name	Type	Description
reserve	Get	Used to reserve a slot in the buffer. Returns a token, CBToken #(n), identifying the slot in the buffer.
complete	Put	Enters the element into the buffer. Takes as arguments the slot in the buffer, CBToken#(n), and the element to be stored in the buffer.
drain	Get	Removes an element from the buffer. The elements are returned in the order the tokens were allocated.

```
interface CompletionBuffer #(numeric type n, type element_type);
    interface Get#(CBToken#(n))                         reserve;
    interface Put#(Tuple2 #(CBToken#(n), element_type)) complete;
    interface Get#(element_type)                        drain;
endinterface: CompletionBuffer
```

Datatypes

The `CBToken` type is abstract to avoid misuse.

```
typedef union tagged { ... } CBToken #(numeric type n) ...;
```

Modules

The `mkCompletionBuffer` module is used to instantiate a completion buffer. It takes no size arguments, as all that information is already contained in the type of the interface it produces.

mkCompletionBuffer	Creates a completion buffer.
	`module mkCompletionBuffer(CompletionBuffer#(n, element_type))` `  provisos (Bits#(element_type, sizea))`

Example- Using a Completion Buffer in a server farm of multipliers

A server farm is a set of identical servers, which can each perform the same task, together with a controller. The controller allocates incoming tasks to any server which happens to be available (free), and sends results back to its caller.

The time needed to complete each task depends on the value of the multiplier argument; there is therefore no guarantee that results will become available in the order the tasks were started. It is required, however, that the controller return results to its caller in the order the tasks were received. The controller accordingly must instantiate a special mechanism for this purpose. The appropriate mechanism is a Completion Buffer.

```
import List::*;
import FIFO::*;
import GetPut::*;
import CompletionBuffer::*;

typedef Bit#(16) Tin;
typedef Bit#(32) Tout;
```



```
// Multiplier interface
interface Mult_IFC;
    method Action  start (Tin m1, Tin m2);
    method ActionValue#(Tout)    result();
endinterface

typedef Tuple2#(Tin,Tin) Args;
typedef 8 BuffSize;
typedef CBToken#(BuffSize) Token;

// This is a farm of multipliers, mkM. The module
// definition for the multipliers mkM is not provided here.
// The interface definition, Mult_IFC, is provided.
module mkFarm#( module#(Mult_IFC) mkM ) ( Mult_IFC );

   // make the buffer twice the size of the farm
   Integer n = div(valueof(BuffSize),2);

   // Declare the array of servers and instantiate them:
   Mult_IFC mults[n];
   for (Integer i=0; i<n; i=i+1)
      begin
         Mult_IFC s <- mkM;
         mults[i] = s;
      end

   FIFO#(Args) infifo <- mkFIFO;

   // instantiate the Completion Buffer, cbuff, storing values of type Tout
   // buffer size is Buffsize, data type of values is Tout
   CompletionBuffer#(BuffSize,Tout) cbuff <- mkCompletionBuffer;

   // an array of flags telling which servers are available:
   Reg#(Bool) free[n];
   // an array of tokens for the jobs in progress on the servers:
   Reg#(Token) tokens[n];
   // this loop instantiates n free registers and n token registers
   // as well as the rules to move data into and out of the server farm
   for (Integer i=0; i<n; i=i+1)
      begin
         // Instantiate the elements of the two arrays:
         Reg#(Bool) f <- mkReg(True);
         free[i] = f;
         Reg#(Token) t <- mkRegU;
         tokens[i] = t;

         Mult_IFC s = mults[i];

         // The rules for sending tasks to this particular server, and for
         // dealing with returned results:
         rule start_server (f); // start only if flag says it's free
            // Get a token
            CBToken#(BuffSize) new_t <- cbuff.reserve.get;
```

```
            Args a = infifo.first;
            Tin a1 = tpl_1(a);
            Tin a2 = tpl_2(a);
            infifo.deq;

            f <= False;
            t <= new_t;
            s.start(a1,a2);
         endrule

         rule end_server (!f);
            Tout x <- s.result;
            // Put the result x into the buffer, at the slot t
            cbuff.complete.put(tuple2(t,x));
            f <= True;
         endrule
      end

   method Action start (m1, m2);
      infifo.enq(tuple2(m1,m2));
   endmethod

   // Remove the element from the buffer, returning the result
   // The elements will be returned in the order that the tokens were obtained.
   method result = cbuff.drain.get;
endmodule
```

C.7.7 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.7.8 FShow

Package

import FShow :: * ;

Description

The `FShow` package defines the typeclass `FShow`. `FShow` includes a single member function, `fshow`. When applied to an object which is an instance of `FShow`, the `fshow` function returns an object of type `Fmt` (Section B.2.8).

This package is provided as both a compiled library package and as BSV source code to facilitate customization. The source code file can be found in the `$BLUESPECDIR/BSVSource` directory. To customize a package, copy the file into a local directory and then include the local directory in the path when compiling. This is done by specifying the path with the `-p` option as described in the BSV Users Guide.

Typeclasses

`FShow` defines the class of types to which the function `fshow` can be applied to create an associated `Fmt` representation.

```
typeclass FShow#(type t);
   function Fmt fshow(t value);
endtypeclass
```

The package defines instances of FShow for many commonly used datatypes. Users can create their own FShow instances for other types (or redefine the instances included in the FShow package).

FShow Instances		
String	Returns a Fmt object showing the value of the string.	`instance FShow#(String);`
Bool	Returns a Fmt object showing True or False.	`instance FShow#(Bool);`
Maybe#(a)	Returns a Fmt object showing Valid and the value, or just Invalid.	`instance FShow#(Maybe#(a))` `provisos(FShow#(a));`
Int#(n)	Returns a Fmt object showing n in a decimal format.	`instance FShow#(Int#(n));`
Bit#(n)	Returns a Fmt object showing n in a hexadecimal format.	`instance FShow#(Bit#(n));`
FIFOF_#(a) FIFOF#(a)	Returns a Fmt object showing the first element and Empty/Full state of the FIFO.	`instance FShow#(FIFOF_#(a))` `provisos(FShow#(a));`
Vector#(n, a)	Returns a Fmt object showing <V elem1 elem2 ...>, where the elemn are the elements of the vector.	`instance FShow#(Vector#(n, a))` `provisos(FShow#(a));`
List#(a)	Returns a Fmt object showing <List elem1 elem2 ...>, where the elemn are the elements of the list.	`instance FShow#(List#(a))` `provisos(FShow#(a));`

FixedPoint#(i,f)	Returns a Fmt object showing FP int.frac where int is the integer part and frac is the fractional part of the fixed point number.	
	`instance FShow#(FixedPoint#(i,f))` `   provisos(Add#(i,f, TAdd#(i,f)),Add#(1,ignore, i));`	

Tuple2#(a,b)	Returns a Fmt object showing Tuple2(a, b).

	`instance FShow#(Tuple2#(a, b))`
	`   provisos(FShow#(a), FShow#(b));`
	`   function Fmt fshow (Tuple2#(a, b) value);`
	`      return $format("Tuple2(", fshow(tpl_1(value)), ",",`
	`                                fshow(tpl_2(value)),")");`

Tuple3#(a,b,c)	Returns a Fmt object showing Tuple3(a, b, c).

	`instance FShow#(Tuple3#(a, b, c))`
	`   provisos(FShow#(a), FShow#(b), FShow#(c));`
	`   function Fmt fshow (Tuple3#(a, b, c) value);`
	`      return $format("Tuple3(", fshow(tpl_1(value)), ",",`
	`                                fshow(tpl_2(value)),",",`
	`                                fshow(tpl_3(value)),")");`

Functions

fshow	Returns a Fmt representation when applied to a value
	`function Fmt fshow(t value);`

concatWith	Concantenates a String (x) with two other arguments a and b, both of type Fmt.
	`function Fmt concatWith(String x, Fmt a, Fmt b);` `   return (a + $format(x) + b);`

Modules

dbgProbe	This module is used like a Probe except that the sampled value (to be viewed in waves) is the ascii representation of fshow(value).
	`module dbgProbe (Probe#(a))` `   provisos(FShow#(a));`

Example

```
package FShowExample;

import Probe::*;
```

```
import FShow::*;
import Vector::*;

/// Define some types ....

typedef Vector#(3,Bool) VOB;
typedef Tuple2#(Bit#(2), Bit#(2)) TUP;

typedef enum {READ, WRITE, UNKNOWN}  OpCommand   deriving(Bounded, Bits, Eq);

typedef struct {OpCommand command;
Bit#(8)   addr;
Bit#(8)   data;
Bit#(8)   length;
Bool      lock;
} Header deriving (Eq, Bits, Bounded);

typedef union tagged {Header  Descriptor;
      Bit#(8) Data;
      } Request deriving(Eq, Bits, Bounded);

/// Define FShow instances for the ones that aren't already in FShow.bsv

instance FShow#(OpCommand);
   function Fmt fshow (OpCommand label);
      case (label)
 READ:    return fshow("READ ");
 WRITE:   return fshow("WRITE");
 UNKNOWN: return fshow("UNKNOWN");
      endcase
   endfunction
endinstance

instance FShow#(Header);
   function Fmt fshow (Header value);
      return ($format("<HEAD ")
      +
      fshow(value.command)
      +
      $format(" (%0d)", value.length)
      +
      $format(" A:%h",  value.addr)
      +
      $format(" D:%h>", value.data));
   endfunction
endinstance

instance FShow#(Request);
   function Fmt fshow (Request request);
      case (request) matches
 tagged Descriptor .a:
    return fshow(a);
 tagged Data .a:
    return $format("<DATA %h>", a);
```



```
      endcase
   endfunction
endinstance

(* synthesize *)
module mkFShowExample (Empty);

   Reg#(Bit#(32)) value <- mkReg(1234);
   Reg#(Bit#(16)) count <- mkReg(0);

   // Probes to send "fshow" strings to waves
   Probe#(VOB)     vob_probe <- dbgProbe;
   Probe#(TUP)     tup_probe <- dbgProbe;
   Probe#(Request) req_probe <- dbgProbe;

   rule every;
      // generate some values
      VOB v_of_bools  = unpack(truncate(count));
      TUP a_tuple     = unpack(truncate(count));
      Request request = unpack(truncate(value));

      // send signals to waves.
      vob_probe <= v_of_bools;
      tup_probe <= a_tuple;
      req_probe <= request;

      // show use with $display
      $display("  A Vector: ", fshow(v_of_bools));
      $display("   A Tuple: ", fshow(a_tuple));
      $display(" A Request: ", fshow(request));
      $display("----------------------------------");

      // update values
      value <= (value << 1) | {0, (value[31] ^ value[21] ^ value[1] ^ value[01])};
      count <= count + 1;
      if (count == 30) $finish;
   endrule

endmodule
```

C.7.9 Assert

Package

import Assert :: *;

Description

The `Assert` package contains definitions to test assertions in the code.

Functions

staticAssert	Compile time assertion. Can be used anywhere a compile-time statement is valid.	
	`module staticAssert(Bool b, String s);`	

dynamicAssert	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);`

continuousAssert	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, String s);`

Examples using Assertions:

```
import Assert:: *;
module mkAssert_Example ();
  // A static assert is checked at compile-time
  // This code checks that the indices are within range
  for (Integer i=0; i<length(cs); i=i+1)
      begin
        Integer new_index = (cs[i]).index;
        staticAssert(new_index < valueOf(n),
            strConcat("Assertion index out of range: ", integerToString(new_index)));
      end

  rule always_fire (True);
       counter <= counter + 1;
  endrule
  // A continuous assert is checked on each clock cycle
  continuousAssert (!fail, "Failure: Fail becomes True");

  // A dynamic assert is checked each time the rule is executed
  rule test_assertion (True);
     dynamicAssert (!fail, "Failure: Fail becomes True");
  endrule
endmodule: mkAssert_Example
```

C.7.10 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.7.11 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 a don't care (?).

  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 don't care (?) 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

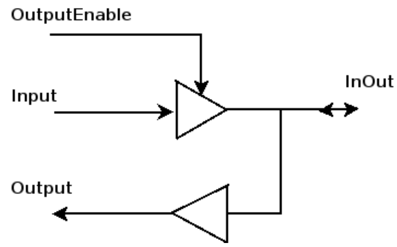

Figure 4: TriState Buffer

C.7.12 TriState

Package

import TriState :: * ;

Description

The `TriState` package implements a tri-state buffer, as shown in Figure 4. Depending on the value of the `output_enable`, `inout` can be an input or an output.

The buffer has two inputs, an `input` of type `value_type` and a Boolean `output_enable` which determines the direction of `inout`. If `output_enable` is True, the signal is coming in from `input` and out through `inout` and `output`. If `output_enable` is False, then a value can be driven in from `inout`, and the `output` value will be the value of `inout`. The behavior is described in the tables below.

output_enable = 0 output = inout		
Inputs		
input	inout	output
0	0	0
0	1	1
1	0	0
1	1	1

output_enable = 1 output = in inout = in		
	Outputs	
input	inout	output
0	0	0
1	1	1

This module is not supported in Bluesim.

Interfaces and Methods

The `TriState` interface is composed of an `InOut` interface and a `_read` method. The `_read` method is similar to the `_read` method of a register in that to read the method you reference the interface in an expression.

TriState Interface		
Name	Type	Description
`inout`	`InOut#(value_type)`	`Inout` subinterface providing a value of type `value_type`
`_read`	`value_type`	Returns the value of `output`

```
(* always_ready, always_enabled *)
interface TriState#(type value_type);
   interface Inout#(value_type)   inout;
   method    value_type           _read;
endinterface: TriState
```

Modules and Functions

The TriState package provides a module constructor function, mkTriState, which provides the TriState interface. The interface includes an InOut subinterface and the value of output.

mkTriState	Creates a module which provides the TriState interface.

	`module mkTriState#(Bool output_enable, value_type input)`
	`                  (TriState#(value_type))`
	`   provisos(Bits#(value_type, size_value));`

Verilog Modules

The TriState module is implemented by the Verilog module TriState.v which can be found in the Bluespec Verilog library, $BLUESPECDIR/Verilog/.

C.7.13 ZBus

Package

import ZBus :: * ;

Description

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.

A ZBus consists of a series of clients hanging off of a bus. The combination of the client and the bus is provided by the ZBusDualIFC interface which consists of 2 subinterfaces, the client and the bus. The client subinterface is provided by the ZBusClientIFC interface. The bus subinterface is provided by the ZBusBusIFC interface. The user never needs to manipulate the bus side, this is all done internally. The user builds the bus out of ZBusDualIFCs and then drives values onto the bus and reads values from the bus using the ZBusClientIFC.

Interfaces and Methods

There are three interfaces are defined in this package; ZBusDualIFC, ZBusClientIFC, and ZBusBusIFC.

The ZBusDualIFC interface provides two subinterfaces; a ZBusBusIFC and a ZBusClientIFC. For a given bus, one ZBusDualIFC interface is associated with each bus client.

ZBusDualIFC		
Name	Type	Description
busIFC	ZBusBusIFC#()	The subinterface providing the bus side of the ZBus.
clientIFC	ZBusClientIFC#(t)	The subinterface providing the client side to the ZBus.

```
interface ZBusDualIFC #(type value_type) ;
   interface ZBusBusIFC#(value_type)    busIFC;
   interface ZBusClientIFC#(value_type) clientIFC;
endinterface
```

The `ZBusClientIFC` allows a BSV module to connect to the tri-state bus. The `drive` method is used to drive a value onto the bus. The `get()` and `fromBusValid()` methods 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.

ZBusClientIFC				
Method			Argument	
Name	Type	Description	Name	Description
drive	Action	Drives a current value on to the bus	value	The value being put on the bus, datatype of `value_type`.
get	value_type	Returns the current value on the bus.		
fromBusValid	Bool	Returns `False` if the bus has a high-impedance value or is undefined.		

```
interface ZBusClientIFC #(type value_type) ;
   method Action      drive(value_type value);
   method value_type  get();
   method Bool        fromBusValid();
endinterface
```

The `ZBusBusIFC` interface connects to the bus structure itself using tri-state values. This interface is never accessed directly by the user.

```
interface ZBusBusIFC #(type value_type) ;
   method Action      fromBusSample(ZBit#(value_type) value, Bool isValid);
   method ZBit#(t)    toBusValue();
   method Bool        toBusCtl();
endinterface
```

Modules and Functions

The library provides a module constructor function, `mkZBusBuffer`, which allows the user to create a module which provides the `ZBusDualIFC` interface. This module provides the functionality of a tri-state buffer.

mkZBusBuffer	Creates a module which provides the ZBusDualIFC interface.	
	`module mkZBusBuffer (ZBusDualIFC #(value_type))` `   provisos (Eq#(value_type), Bits#(value_type, size_value));`	

The mkZBus module constructor function takes a list of ZBusBusIFC interfaces as arguments and creates a module which ties them all together in a bus.

mkZBus	Ties a list of ZBusBusIFC interfaces together in a bus.	
	`module mkZBus#(List#(ZBusBusIFC#(value_type)) ifc_list)(Empty)` `   provisos (Eq#(value_type), Bits#(value_type, size_value));`	

Examples - ZBus

Creating a tri-state buffer for a 32 bit signal. The interface is named buffer_0.

```
ZBusDualIFC#(Bit#(32)) buffer_0();
mkZBusBuffer inst_buffer_0(buffer_0);
```

Drive a value of 12 onto the associated bus.

```
buffer_0.clientIFC.drive(12);
```

The following code fragment demonstrates the use of the module mkZBus.

```
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.7.14 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 (`http://www.accellera.org`).

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

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

Setting Assertion Parameters

Each assertion checker module has a set of associated parameter values that can be customized for each module instantiation. The values for these parameters are passed to each checker module in the form of a single struct argument of type `OVLDefaults#(a)` A typical use scenario is illustrated below:

```
let defaults = mkOVLDefaults;

defaults.min_clks = 2;
defaults.max_clks = 3;

AssertTest_IFC#(Bool) assertWid <- bsv_assert_width(defaults);
```

The defaults struct (created by `mkOVLDefaults`) includes one field for each possible parameter. Initially each field includes the associated default value. By editing fields of the struct, individual parameter values can be modified as needed to be non-default values. The modified `defaults` struct is then provided as a module argument during instantiation.

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));`

<table>
<tr><td>Module</td><td><code>bsv_assert_decrement</code></td></tr>
<tr><td>Description</td><td>Ensures that the expression decrements only by the value specifiedR.</td></tr>
<tr><td>Interface Used</td><td><code>AssertTest_IFC</code></td></tr>
<tr><td>Parameters</td><td>common assertion parameters
<code>value</code> (default value = 1)</td></tr>
<tr><td>Module Declaration</td><td><pre>
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));
</pre></td></tr>
</table>

<table>
<tr><td>Module</td><td><code>bsv_assert_delta</code></td></tr>
<tr><td>Description</td><td>Ensures that the expression always changes by a value within the range specified by <code>min</code> and <code>max</code>.</td></tr>
<tr><td>Interface Used</td><td><code>AssertTest_IFC</code></td></tr>
<tr><td>Parameters</td><td>common assertion parameters
<code>min</code> (default value = 1)
<code>max</code> (default value = 1)</td></tr>
<tr><td>Module Declaration</td><td><pre>
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));
</pre></td></tr>
</table>

<table>
<tr><td>Module</td><td><code>bsv_assert_even_parity</code></td></tr>
<tr><td>Description</td><td>Ensures that value of a specified expression has even parity, that is an even number of bits in the expression are active high.</td></tr>
<tr><td>Interface Used</td><td><code>AssertTest_IFC</code></td></tr>
<tr><td>Parameters</td><td>common assertion parameters</td></tr>
<tr><td>Module Declaration</td><td><pre>
module bsv_assert_even_parity#(OVLDefaults#(a_type)
               defaults) (AssertTest_IFC#(a_type))
    provisos (Bits#(a_type, sizea),
              Bounded#(a_type), Eq#(a_type));
</pre></td></tr>
</table>

<table>
<tr><td>Module</td><td><code>bsv_assert_fifo_index</code></td></tr>
<tr><td>Description</td><td>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.</td></tr>
<tr><td>Interface Used</td><td><code>AssertFifoTest_IFC</code></td></tr>
<tr><td>Parameters</td><td>common assertion parameters
<code>depth</code> (default value = 1)
<code>simultaneous_push_pop</code> (default value = <code>True</code>)</td></tr>
<tr><td>Module Declaration</td><td><pre>
module bsv_assert_fifo_index#(OVLDefaults#(Bit#(0))
            defaults)(AssertFifoTest_IFC#(a_type, b_type))
    provisos (Bits#(a_type, sizea), Bits#(b_type, sizeb));
</pre></td></tr>
</table>

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));`

<table>
<tr><td>Module</td><td><code>bsv_assert_no_overflow</code></td></tr>
<tr><td>Description</td><td>Ensures that the value of the specified expression does not overflow.</td></tr>
<tr><td>Interface Used</td><td><code>AssertTest_IFC</code></td></tr>
<tr><td>Parameters</td><td>common assertion parameters
<code>min</code> (default value = <code>minBound</code>)
<code>max</code> (default value = <code>maxBound</code>)</td></tr>
<tr><td>Module Declaration</td><td><pre>module bsv_assert_no_overflow#(OVLDefaults#(a_type)
               defaults) (AssertTest_IFC#(a_type))
    provisos (Bits#(a_type, sizea),
              Bounded#(a_type), Eq#(a_type));</pre></td></tr>
</table>

<table>
<tr><td>Module</td><td><code>bsv_assert_no_transition</code></td></tr>
<tr><td>Description</td><td>Ensures that the value of a specified expression does not transition from a start state to the specified next state.</td></tr>
<tr><td>Interface Used</td><td><code>AssertTransitionTest_IFC</code></td></tr>
<tr><td>Parameters</td><td>common assertion parameters</td></tr>
<tr><td>Module Declaration</td><td><pre>module bsv_assert_no_transition#(OVLDefaults#(a_type)
             defaults) (AssertTransitionTest_IFC#(a_type))
    provisos (Bits#(a_type, sizea),
              Bounded#(a_type), Eq#(a_type));</pre></td></tr>
</table>

<table>
<tr><td>Module</td><td><code>bsv_assert_no_underflow</code></td></tr>
<tr><td>Description</td><td>Ensures that the value of the specified expression does not underflow.</td></tr>
<tr><td>Interface Used</td><td><code>AssertTest_IFC</code></td></tr>
<tr><td>Parameters</td><td>common assertion parameters
<code>min</code> (default value = <code>minBound</code>)
<code>max</code> (default value = <code>maxBound</code>)</td></tr>
<tr><td>Module Declaration</td><td><pre>module bsv_assert_no_underflow#(OVLDefaults#(a_type)
               defaults)(AssertTest_IFC#(a_type))
    provisos (Bits#(a_type, sizea),
              Bounded#(a_type), Eq#(a_type));</pre></td></tr>
</table>

<table>
<tr><td>Module</td><td><code>bsv_assert_odd_parity</code></td></tr>
<tr><td>Description</td><td>Ensures that the specified expression had odd parity; that an odd number of bits in the expression are active high.</td></tr>
<tr><td>Interface Used</td><td><code>AssertTest_IFC</code></td></tr>
<tr><td>Parameters</td><td>common assertion parameters</td></tr>
<tr><td>Module Declaration</td><td><pre>module bsv_assert_odd_parity#(OVLDefaults#(a_type)
               defaults)(AssertTest_IFC#(a_type))
    provisos (Bits#(a_type, sizea),
              Bounded#(a_type), Eq#(a_type));</pre></td></tr>
</table>

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);


   // set the default values
   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
```

Using The Library

In order to use the OVLAssertions package, a user must first download the source OVL library from Accellera (`http://www.accellera.org`). In addition, that library must be made available when building a simulation executable from the BSV generated Verilog.

If the bsc compiler is being used to generate the Verilog simulation executable, the `BSC_VSIM_FLAGS` environment variable can be used to set the required simulator flags that enable use of the OVL library.

For instance, if the `iverilog` simulator is being used and the OVL library is located in the directory `shared/std_ovl`, the `BSC_VSIM_FLAGS` environment variable can be set to `¨-I shared/std_ovl -Y .vlib -y shared/std_ovl -DOVL_VERILOG=1 -DOVL_ASSERT_ON=1¨`. These flags:

- Add `shared/std_ovl` to the Verilog and include search paths.
- Set `.vlib` as a possible file suffix.
- Set flags used in the OVL source code.

The exact flags to be used will differ based on what OVL behavior is desired and which Verilog simulator is being used.

C.8 Multiple Clock Domains and Clock Generators

Package Name

Import Clocks :: * ;

Description

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.

The `Clocks` package uses the data types `Clock` and `Reset` as well as clock functions which are described below but defined in the `Prelude` package.

Each section describes a related group of modules, followed by a table indicating the Verilog modules used to implement the BSV modules.

Types and typeclasses

The `Clocks` package uses the abstract data types `Clock` and `Reset`, which are defined in the `Prelude` package. These are first class objects. Both `Clock` and `Reset` are in the `Eq` type class, meaning two values can be compared for equality.

`Clock` is an abstract type of two components: a single `Bit` oscillator and a `Bool` gate.

```
typedef ... Clock ;
```

`Reset` is an abstract type.

```
typedef ... Reset ;
```

Type Classes for `Clock` and `Reset`									
	Bits	Eq	Literal	Arith	Ord	Bounded	Bitwise	Bit Reduction	Bit Extend
Clock		√							
Reset		√							

Example: Declaring a new clock

```
Clock clk0;
```

Example: Instantiating a register with clock and reset


```
Reg#(Byte) a <- mkReg(0, clocked_by clks0, reset_by rst0);
```

Functions

The following functions are defined in the `Prelude` package but are used with multiple clock domains.

Clock Functions	
`exposeCurrentClock`	This function returns a value of type `Clock`, which is the current clock of the module.
	`module exposeCurrentClock ( Clock c );`

`exposeCurrentReset`	This function returns a value of type `Reset`, which is the current reset of the module.
	`module exposeCurrentReset ( Reset r );`

Both `exposeCurrentClock` and `exposeCurrentReset` use the module instantiation syntax (`<-`) to return the value. Hence these can only be used from within a module.

Example: setting a reset to the current reset

```
Reset reset_value <- exposeCurrentReset;
```

Example: setting a clock to the current clock

```
Clock clock_value <- exposeCurrentClock;
```

`sameFamily`	A Boolean function which returns `True` if the clocks are in the same family, `False` if the clocks are not in the same family. Clocks in the same family have the same oscillator but may have different gate conditions.
	`function Bool sameFamily ( Clock clka, Clock clkb ) ;`

`isAncestor`	A Boolean function which returns `True` if `clka` is an ancestor of `clkb`, that is `clkb` is a gated version of `clka` (`clka` itself may be gated) or if `clka` and `clkb` are the same clock. The ancestry relation is a partial order (ie., reflexive, transitive and antisymmetric).
	`function Bool isAncestor ( Clock clka, Clock clkb ) ;`

`clockOf`	Returns the current clock of the object `obj`.
	`function Clock clockOf ( a_type obj ) ;`

`noClock`	Specifies a *null* clock, a clock where the oscillator never rises.
	`function Clock noClock() ;`

`resetOf`	Returns the current reset of the object `obj`.
	`function Reset resetOf ( a_type obj ) ;`

`noReset`	Specifies a *null* reset, a reset which is never asserted.
	`function Reset noReset() ;`

C.8.1 Clock Generators and Clock Manipulation

Description

This section provides modules to generate new clocks and to modify the existing clock.

The modules `mkAbsoluteClock`, `mkAbsoluteClockFull`, `mkClock`, and `mkUngatedClock` all define a new clock, one not based on the current clock. Both `mkAbsoluteClock` and `mkAbsoluteClockFull` define new oscillators and are not synthesizable. `mkClock` and `mkUngatedClock` use an existing oscillator to create a clock, and is synthesizable. The modules, `mkGatedClock` and `mkGatedClockFromCC` use existing clocks to generate another clock in the same family.

Interfaces and Methods

The `MakeClockIfc` supports user-defined clocks with irregular waveforms created with `mkClock` and `mkUngatedClock`, as opposed to the fixed-period waveforms created with the `mkAbsoluteClock` family.

MakeClockIfc Interface				
Method and subinterfaces			Arguments	
Name	Type	Description	Name	Description
`setClockValue`	`Action`	Changes the value of the clock at the next edge of the clock	`value`	Value the clock will be set to, must be a one bit type
`getClockValue`	`one_bit_type`	Retrieves the last value of the clock		
`setGateCond`	`Action`	Changes the gating condition	`gate`	Must be of the type `Bool`
`getGateCond`	`Bool`	Retrieves the last gating condition set		
`new_clk`	`Interface`	Clock interface provided by the module		

```
interface MakeClockIfc#(type one_bit_type);
   method Action       setClockValue(one_bit_type value) ;
   method one_bit_type getClockValue() ;
```



```
   method Action       setGateCond(Bool gate) ;
   method Bool         getGateCond() ;
   interface Clock     new_clk ;
endinterface
```

The `GatedClockIfc` is used for adding a gate to an existing clock.

GatedClockIfc Interface				
Method and subinterfaces			Arguments	
Name	Type	Description	Name	Description
`setGateCond`	`Action`	Changes the gating condition	`gate`	Must be of the type `Bool`
`getGateCond`	`Bool`	Retrieves the last gating condition set		
`new_clk`	`Interface`	Clock interface provided by the module		

```
interface GatedClockIfc ;
   method    Action setGateCond(Bool gate) ;
   method    Bool   getGateCond() ;
   interface Clock  new_clk ;
endinterface
```

Modules

The `mkClock` module creates a Clock type from a one-bit oscillator and a Boolean gate condition. There is no family relationship between the current clock and the clock generated by this module. The initial values of the oscillator and gate are passed as parameters to the module. When the module is out of reset, the oscillator value can be changed using the `setClockValue` method and the gate condition can be changed by calling the `setGateCond` method. The oscillator value and gate condition can be queried with the `getClockValue` and `getGateCond` methods, respectively. The clock created by `mkClock` is available as the `new_clk` subinterface. When setting the gate condition, the change does not affect the generated clock until it is low, to prevent glitches.

The `mkUngatedClock` module is an ungated version of the `mkClock` module. It takes only an oscillator argument (no gate argument) and returns the same `new_clock` interface. Since there is no gate, an error is returned if the design calls the `setGetCond` method. The `getGateCond` method always returns True.

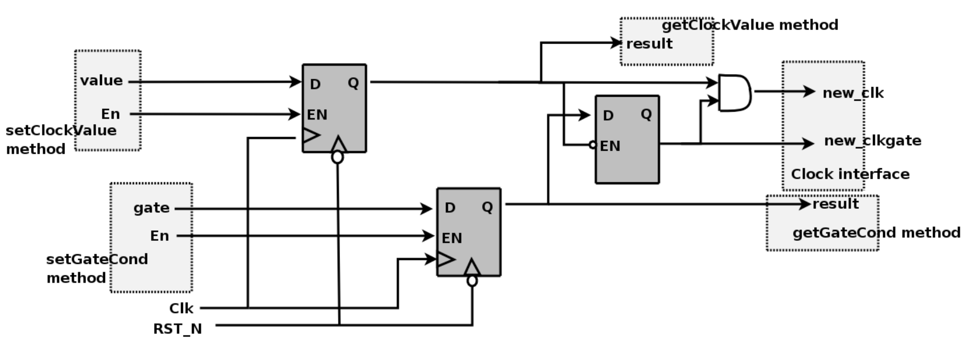

Figure 5: Clock Generator

mkClock	Creates a Clock type from a one-bit oscillator input, and a Boolean gate condition. There is no family relationship between the current clock and the clock generated by this module.
	`module mkClock #( one_bit_type initVal, Bool initGate) ( MakeClockIfc#(one_bit_type) ifc ) provisos( Bits#(one_bit_type, 1) ) ;`

mkUngatedClock	Creates an ungated Clock type from a one-bit oscillator input. There is no family relationship between the current clock and the clock generated by this module.
	`module mkUngatedClock #( one_bit_type initVal) ( MakeClockIfc#(one_bit_type) ifc ) provisos( Bits#(one_bit_type, 1) ) ;`

The `mkGatedClock` module adds (logic and) a Boolean gate condition to an existing clock, thus creating another clock in the same family. The source clock is provided as the argument `clk_in`. The gate condition is controlled by an asynchronously-reset register inside the module. The register is set with the `setGateCond` Action method of the interface and can be read with `getGateCond` method. The reset value of the gate condition register is provided as an instantiation parameter. The clock for the register (and thus these set and get methods) is the default clock of the module; to specify a clock other than the default clock, use the `clocked_by` directive.

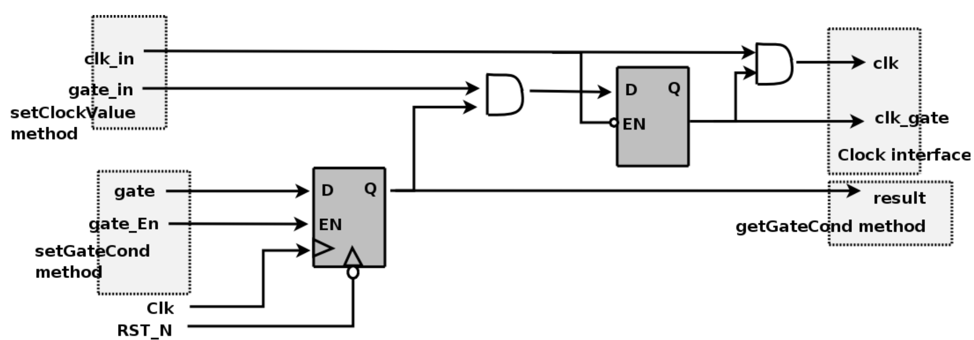

Figure 6: Gated Clock Generator

mkGatedClock	Creates another clock in the same family by adding logic and a Boolean gate condition to the current clock.
	`module mkGatedClock#(Bool v) ( Clock clk_in, GatedClockIfc ifc );`

For convenience, we provide an alternate version in which the source clock is the default clock of the module

mkGatedClockFromCC	An alternate interface for the module `mkGatedClock` in which the source clock is the default clock of the module.
	`module mkGatedClockFromCC#(Bool v) ( GatedClockIfc ifc );`

The modules `mkAbsoluteClock` and `mkAbsoluteClockFull` provide parametizable clock generation modules which are *not* synthesizable, but may be useful for testbenches. In `mkAbsoluteClock`, the first rising edge (start) and the period are defined by parameters. Additional parameters are provided by `mkAbsoluteClockFull`.

mkAbsoluteClock	The first rising edge (start) and period are defined by parameters. This module is not synthesizable.

	`module mkAbsoluteClock #( Integer start,`
	`Integer period )`
	`( Clock );`

mkAbsoluteClockFull	The value `initValue` is held until time `start`, and then the clock oscillates. The value `not(initValue)` is 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.

	`module mkAbsoluteClockFull #( Integer start,`
	`Bit#(1) initValue,`
	`Integer compValTime,`
	`Integer initValTime )`
	`( Clock );`

Verilog Modules

The BSV modules correspond to the following Verilog modules, which are found in the Bluespec Verilog library, `$BLUESPECDIR/Verilog/`.

BSV Module Name	Verilog Module Name
mkAbsoluteClock mkAbsoluteClockFull	ClockGen.v
mkClock mkUngatedClock	MakeClock.v
mkGatedClock mkGatedClockFromCC	GatedClock.v

C.8.2 Clock Multiplexing

Description

Bluespec provides two gated clock multiplexing primitives: a simple combinational multiplexor and a stateful module which generates an appropriate reset signal when the clock changes. The first multiplexor uses the interface `MuxClockIfc`, which includes an `Action` method to select the clock along with a `Clock` subinterface. The second multiplexor uses the interface `SelectClockIfc` which also has a `Reset` subinterface.

Ungated versions of these modules are also provided. The ungated versions are identical to the gated versions, except that the input and output clocks are ungated.

Interfaces and Methods

MuxClockIfc Interface				
Method and subinterfaces			Arguments	
Name	Type	Description	Name	Description
`select`	`Action`	Method used to select the clock based on the Boolean value `ab`	`ab`	if True, `clock_out` is taken from `aclk`
`clock_out`	`Interface`	Clock interface		

```
interface MuxClkIfc ;
   method    Action select ( Bool  ab ) ;
   interface Clock  clock_out ;
endinterface
```

SelectClockIfc Interface				
Method and subinterfaces			Arguments	
Name	Type	Description	Name	Description
`select`	`Action`	Method used to select the clock based on the Boolean value `ab`	`ab`	if True, clock_out is taken from `aclk`
`clock_out`	`Interface`	Clock interface		
`reset_out`	`Interface`	Reset interface		

```
interface SelectClkIfc ;
   method    Action select ( Bool  ab ) ;
   interface Clock  clock_out ;
   interface Reset  reset_out ;
endinterface
```

Modules

The `mkClockMux` module is a simple combinational multiplexor with a registered clock selection signal, which selects between clock inputs `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 two arguments and provides a Clock interface. The `aClk` is selected if `ab` is True, while `bClk` is selected otherwise.

The `mkUngatedClockMux` module is identical to the `mkClockMux` module except that the input and output clocks are ungated. The signals `aClkgate`, `bClkgate`, and `outClkgate` in figure 7 don't exist.

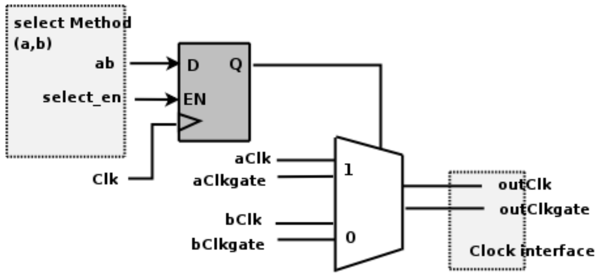

Figure 7: Clock Multiplexor

`mkClockMux`	Simple combinational multiplexor, which selects between aClk and bClk.	
	`module mkClockMux ( Clock aClk, Clock bClk )` `                  ( MuxClkIfc ) ;`	

`mkUngatedClockMux`	Simple combinational multiplexor, which selects between aClk and bClk. None of the clocks are gated.
	`module mkUngatedClockMux ( Clock aClk, Clock bClk )` `                         ( MuxClkIfc ) ;`

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 constructor for the module uses two clock arguments, and provides the `MuxClockIfc` 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 selection changes.

The `mkUngatedClockSelect` module is identical to the `mkClockSelect` module except that the input and output clocks are ungated. The signals `aClkgate`, `bClkgate`, and `outClk_gate` in figure 8 don't exist.

`mkClockSelect`	Clock Multiplexor containing additional logic which generates a reset whenever a new clock is selected.

	`module mkClockSelect #( Integer stages,`
	`                          Clock aClk,`
	`                          Clock bClk,`
	`                        ( SelectClockIfc ) ;`

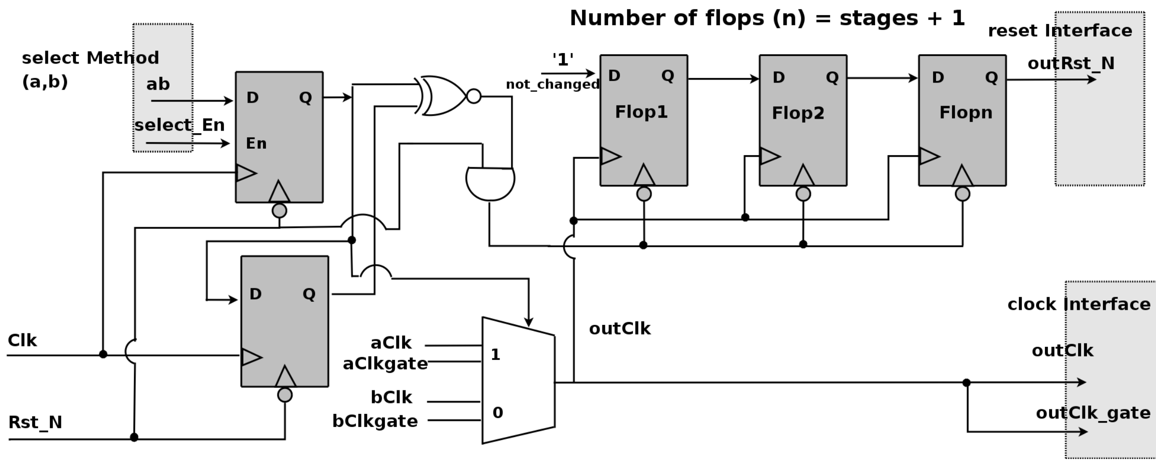

Figure 8: Clock Multiplexor with reset

`mkUngatedClockSelect`	Clock Multiplexor containing additional logic which generates a reset whenever a new clock is selected. The input and output clocks are ungated.
	`module mkUngatedClockSelect #( Integer stages, Clock aClk, Clock bClk, ( SelectClockIfc ) ;`

Verilog Modules

The BSV modules correspond to the following Verilog modules, which are found in the Bluespec Verilog library, `$BLUESPECDIR/Verilog/`.

BSV Module Name	Verilog Module Name
`mkClockMux`	`ClockMux.v`
`mkClockSelect`	`ClockSelect.v`
`mkUngatedClockMux`	`UngatedClockMux.v`
`mkUngatedClockSelect`	`UngatedClockSelect.v`

C.8.3 Clock Division

Description

A clock divider provides a derived clock and also a `ClkNextRdy` signal, which indicates that the 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` in Section C.8.10 for some specialized synchronizers which can be used with divided clocks, and other systems when the clock edges are known to be aligned.

Data Types

The `ClkNextRdy` is a Boolean signal which indicates that the slow clock will rise in the next cycle.


```
typedef Bool ClkNextRdy ;
```

Interfaces and Methods

ClockDividerIfc Interface		
Name	Type	Description
`fastClock`	`Interface`	The original clock
`slowClock`	`Interface`	The derived clock
`clockReady`	`Bool`	Boolean value which indicates that the slow clock will rise in the next cycle. The method is in the clock domain of the fast clock.

```
interface ClockDividerIfc ;
    interface Clock      fastClock ;
    interface Clock      slowClock ;
    method    ClkNextRdy clockReady() ;
endinterface
```

Modules

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.

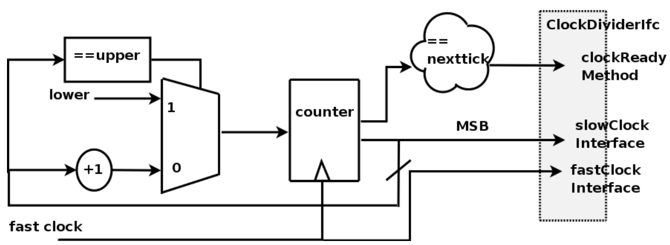

Figure 9: Clock Divider

`mkClockDivider`	Basic clock divider.
	`module mkClockDivider #( Integer divisor ) ( ClockDividerIfc ) ;`

`mkGatedClockDivider`	A gated verison of the basic clock divider.
	`module mkGatedClockDivider #( Integer divisor )( ClockDividerIfc ) ;`

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. `mkClockDivider` is just `mkClockDividerOffset` with an offset of value 0.

`mkClockDividerOffset`	Provides a clock divider, where the rising edge can be defined relative to other clock dividers which have the same divisor.	
	`module mkClockDividerOffset #( Integer divisor, Integer offset ) ( ClockDividerIfc ) ;`	

The `mkClockInverter` and `mkGatedClockInverter` modules generate an inverted clock having the same period but opposite phase as the current clock. The `mkGatedClockInverter` is a gated version of `mkClockInverter`. The output clock includes a gate signal derived from the gate of the input clock.

`mkClockInverter`	Generates an inverted clock having the same period but opposite phase as the current clock.
	`module mkClockInverter ( ClockDividerIfc ) ;`

`mkGatedClockInverter`	A gated version of `mkClockInverter`.
	`module mkGatedClockInverter ( ClockDividerIfc ifc ) ;`

Verilog Modules

The BSV modules correspond to the following Verilog modules, which are found in the Bluespec Verilog library, `$BLUESPECDIR/Verilog/`.

BSV Module Name	Verilog Module Name
`mkClockDivider` `mkClockDividerOffset`	`ClockDiv.v`
`mkGatedClockDivider`	`GatedClockDiv.v`
`mkClockInverter`	`ClockInverter.v`
`mkGatedClockInverter`	`GatedClockInverter.v`

C.8.4 Bit Synchronizers

Description

Bit synchronizers are used to safely transfer one bit of data from one clock domain to another. More complicated synchronizers are provided in later sections.

Interfaces and Methods

The `SyncBitIfc` interface provides a `send` method which transmits one bit of information from one clock domain to the `read` method in a second domain.

SyncBitIfc Interface				
Methods			Arguments	
Name	Type	Description	Name	Description
`send`	`Action`	Transmits information from one clock domain to the second domain	`bitData`	One bit of information transmitted
`read`	`one_bit_type`	Reads one bit of data sent from a different clock domain		

```
interface SyncBitIfc #(type one_bit_type) ;
   method Action       send ( one_bit_type bitData ) ;
   method one_bit_type read () ;
endinterface
```

Modules

The `mkSyncBit`, `mkSyncBitFromCC` and `mkSyncBitToCC` modules provide a `SyncBitIfc` across clock domains. The send method is in one clock domain, and the read method is in a second clock domain, as shown in Figure 10. The `FromCC` and `ToCC` versions differ in that the `FromCC` module moves data *from* the current clock (module's clock), while the `ToCC` module moves data *to* 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.

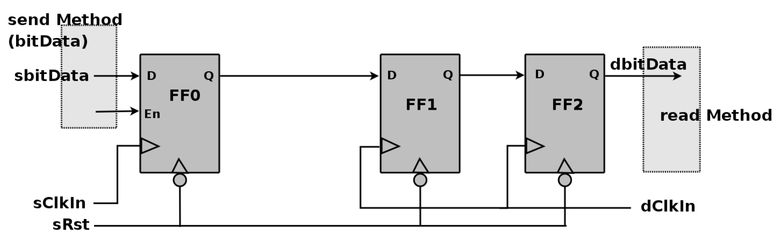

Figure 10: Bit Synchronizer

`mkSyncBit`	Moves data across clock domains. The in and out clocks, along with the input reset, are explicitly provided. The default clock and reset are ignored.
	`module mkSyncBit #( Clock sClkIn, Reset sRst, Clock dClkIn ) ( SyncBitIfc #(one_bit_type) ) provisos( Bits#(one_bit_type, 1)) ;`

`mkSyncBitFromCC`	Moves data from the current clock (the module's clock) to a different clock domain. The input clock and reset are the current clock and reset.
	`module mkSyncBitFromCC #( Clock dClkIn ) ( SyncBitIfc #(one_bit_type) ) provisos( Bits#(one_bit_type, 1)) ;`

mkSyncBitToCC	Moves data into the current clock domain. The output clock is the current clock. The current reset is ignored.	
	`module mkSyncBitToCC #( Clock sClkIn, Reset sRstIn ) ( SyncBitIfc #(one_bit_type) ) provisos( Bits#(one_bit_type, 1)) ;`	

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, as shown in Figure 11. For these synchronizers, the first register clocked by the destination clock triggers on the falling edge of the clock.

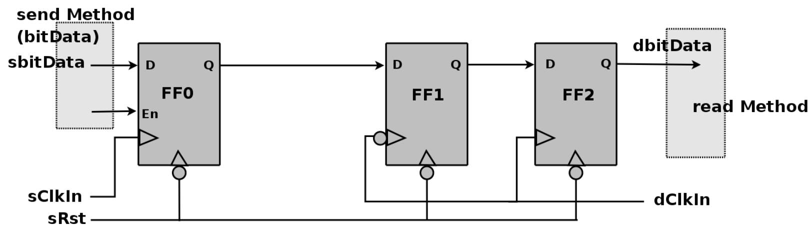

Figure 11: Bit Synchronizer 1.5 - first register in destination domain triggers on falling edge

mkSyncBit15	Similar to `mkSyncBit` except it triggers on the falling edge of the clock. The in and out clocks, along with the input reset, are explicitly provided. The default clock and reset are ignored.
	`module mkSyncBit15 #( Clock sClkIn, Reset sRst, Clock dClkIn ) ( SyncBitIfc #(one_bit_type) ) provisos( Bits#(one_bit_type, 1)) ;`

mkSyncBit15FromCC	Moves data from the current clock and is triggered on the falling edge of the clock. The input clock and reset are the current clock and reset.
	`module mkSyncBit15FromCC #(Clock dClkIn) (SyncBitIfc #(one_bit_type)) provisos( Bits#(one_bit_type, 1)) ;`

mkSyncBit15ToCC	Moves data into the current clock domain and is triggered on the falling edge of the clock. The output clock is the current clock. The current reset is ignored.
	`module mkSyncBit15ToCC #( Clock sClkIn, Reset sRstIn ) ( SyncBitIfc #(one_bit_type) ) provisos( Bits#(one_bit_type, 1)) ;`

The `mkSyncBit1` module, shown in Figure 12, also provides the same interface but only uses one register in the destination domain. Synchronizers like this, which use only one register, are not generally used since meta-stable output is more 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.

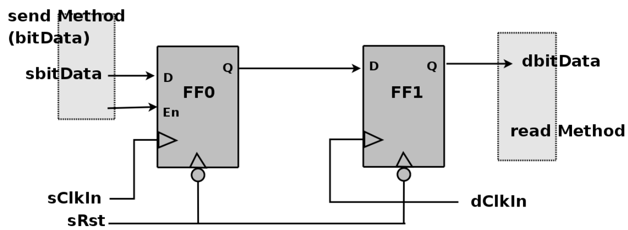

Figure 12: Bit Synchronizer 1.0 - single register in destination domain

`mkSyncBit1`	Moves data from one clock domain to another clock domain, with only one register in the destination domain. The in and out clocks, along with the input reset, are explicitly provided. The default clock and reset are ignored.

	<pre>module mkSyncBit1 #(Clock sClkIn, Reset sRst,
	                      Clock dClkIn)
	                    (SyncBitIfc #(one_bit_type))
	   provisos(Bits#(one_bit_type, 1)) ;</pre>

`mkSyncBit1FromCC`	Moves data from the current clock domain, with only one register in the destination domain. The input clock and reset are the current clock and reset.

	<pre>module mkSyncBit1FromCC #(Clock dClkIn)
	                          (SyncBitIfc #(one_bit_type))
	   provisos(Bits #(one_bit_type, 1)) ;</pre>

`mkSyncBit1ToCC`	Moves data into the current clock domain, with only one register in the destination domain. The output clock is the current clock. The current reset is ignored.

	<pre>module mkSyncBit1ToCC #(Clock sClkIn, Reset sRstIn)
	                        (SyncBitIfc #(one_bit_type))
	   provisos(Bits#(one_bit_type, 1)) ;</pre>

The `mkSyncBit05` module is similar to `mkSyncBit1`, but the destination register triggers on the falling edge of the clock, as shown in Figure 13.

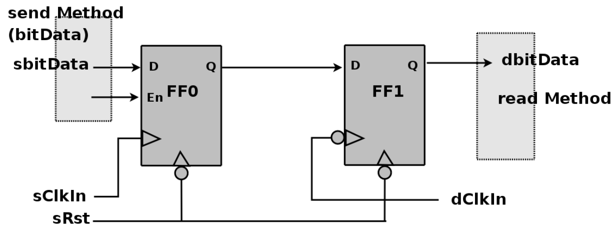

Figure 13: Bit Synchronizer .5 - first register in destination domain triggers on falling edge

`mkSyncBit05`	Moves data from one clock domain to another clock domain, with only one register in the destination domain. The destination register triggers on the falling edge of the clock. The in and out clocks, along with the input reset, are explicitly provided. The default clock and reset are ignored.

	`module mkSyncBit05 #( Clock sClkIn, Reset sRst,`
	`                       Clock dClkIn )`
	`                     ( SyncBitIfc #(one_bit_type) )`
	`   provisos( Bits#(one_bit_type, 1)) ;`

`mkSyncBit05FromCC`	Moves data from the current clock domain, with only one register in the destination domain, the destination register triggers on the falling edge of the clock. The input clock and reset are the current clock and reset.

	`module mkSyncBit05FromCC #( Clock dClkIn )`
	`                          (SyncBitIfc #(one_bit_type) )`
	`   provisos( Bits#(one_bit_type, 1)) ;`

`mkSyncBit05ToCC`	Moves data into the current clock domain, with only one register in the destination domain, the destination register triggers on the falling edge of the clock. The output clock is the current clock. The current reset is ignored.

	`module mkSyncBit05ToCC #( Clock sClkIn, Reset sRstIn )`
	`                        ( SyncBitIfc #(one_bit_type) )`
	`   provisos( Bits#(one_bit_type, 1)) ;`

Verilog Modules

The BSV modules correspond to the following Verilog modules, which are found in the Bluespec Verilog library, `$BLUESPECDIR/Verilog/`.

BSV Module Name	Verilog Module Name
mkSyncBit mkSyncBitFromCC mkSyncBitToCC	SyncBit.v
mkSyncBit15 mkSyncBit15FromCC mkSyncBit15ToCC	SyncBit15.v
mkSyncBit1 mkSyncBit1FromCC mkSyncBit1ToCC	SyncBit1.v
mkSyncBit05 mkSyncBit05FromCC mkSyncBit05ToCC	SyncBit05.v

C.8.5 Pulse Synchronizers

Description

Pulse synchronizers are used to transfer a pulse from one clock domain to another.

Interfaces and Methods

The `SyncPulseIfc` interface provides an Action method, `send`, which when invoked generates a True value on the `pulse` method in a second clock domain.

SyncPulseIfc Interface		
Methods		
Name	Type	Description
send	Action	Starts transmittling a pulse from one clock domain to the second clock domain.
pulse	Bool	Where the pulse is received in the second domain. `pulse` is True if a pulse is recieved in this cycle.

```
interface SyncPulseIfc ;
   method Action send  () ;
   method Bool   pulse () ;
endinterface
```

Modules

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.

mkSyncPulse	Sends a pulse from one clock domain to another. The in and out clocks, along with the input reset, are explicitly provided. The default clock and reset are ignored.
	`module mkSyncPulse #( Clock sClkIn, Reset sRstIn, Clock dClkIn ) ( SyncPulseIfc ) ;`

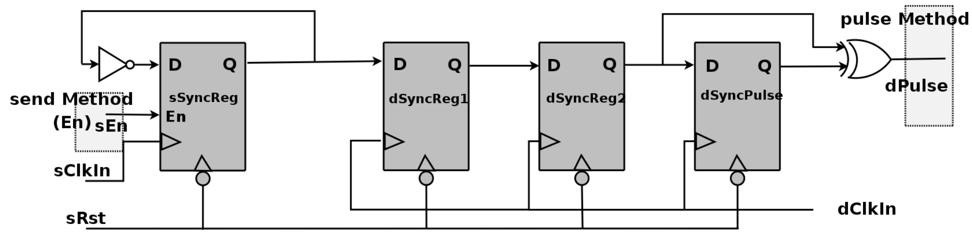

Figure 14: Pulse Synchronizer - no handshake

`mkSyncPulseFromCC`	Sends a pulse from the current clock domain to the other clock domain. The input clock and reset are the current clock and reset.
	`module mkSyncPulseFromCC #( Clock dClkIn )` `                          ( SyncPulseIfc ) ;`

`mkSyncPulseToCC`	Sends a pulse from the other clock domain to the current clock domain. The output clock is the current clock. The current reset is ignored.
	`module mkSyncPulseToCC #( Clock sClkIn, Reset sRstIn )` `                        ( SyncPulseIfc ) ;`

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. Note that this only guarantees that the pulse is seen in one clock cycle of the destination; it does not guarantee that the system on that side reacted to the pulse before it was gone. It is up to the designer to ensure this, if necessary.

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.

`mkSyncHandshake`	Sends a pulse from one clock domain to another clock domain with handshaking. The in and out clocks, along with the input reset, are explicitly provided. The default clock and reset are ignored.

	`module mkSyncHandshake #( Clock sClkIn, Reset sRstIn,`
	`                          Clock dClkIn )`
	`                        ( SyncPulseIfc ) ;`

`mkSyncHandShakeFromCC`	Sends a pulse with a handshake from the current clock domain. The input clock and reset are the current clock and reset.
	`module mkSyncHandshakeFromCC #( Clock dClkIn )` `                              ( SyncPulseIfc ) ;`

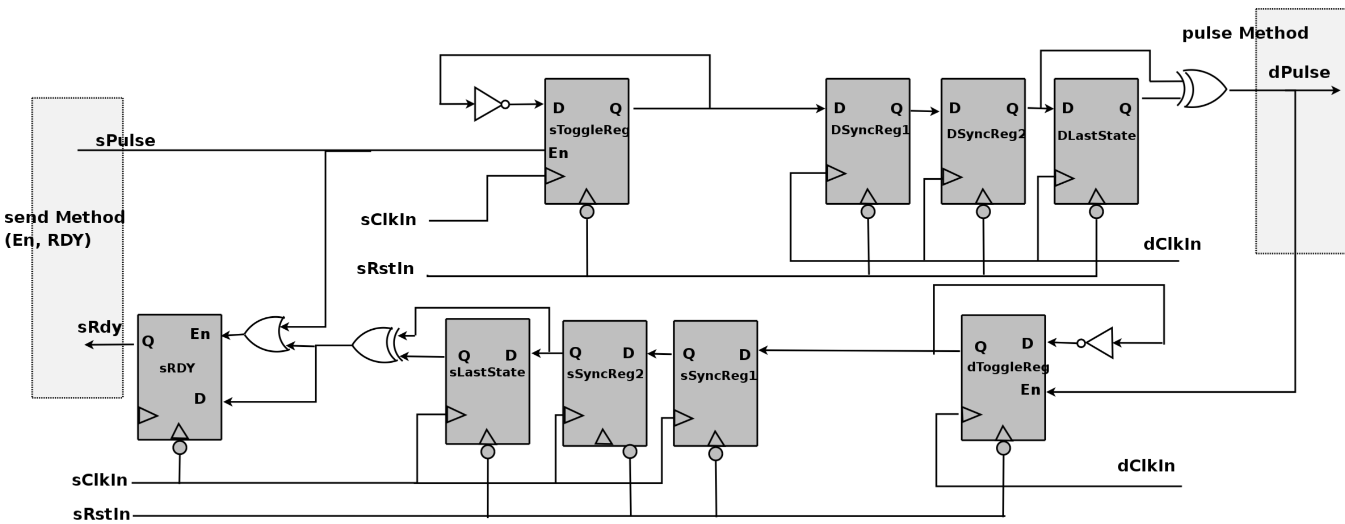

Figure 15: Pulse Synchronizer with handshake

`mkSyncHandshakeToCC`	Sends a pulse with a handshake to the current clock domain. The output clock is the current clock. The current reset is ignored.

	`module mkSyncHandshakeToCC #( Clock sClkIn,`
	`Reset sRstIn )`
	`( SyncPulseIfc ) ;`

Verilog Modules

The BSV modules correspond to the following Verilog modules, which are found in the Bluespec Verilog library, `$BLUESPECDIR/Verilog/`.

BSV Module Name	Verilog Module Name

`mkSyncPulse`
`mkSyncPulseFromCC`
`mkSyncPulseToCC`
	`SyncPulse.v`

`mkSyncHandshake`
`mkSyncHandshakeFromCC`
`mkSyncHandshakeToCC`
	`SyncHandshake.v`

C.8.6 Word Synchronizers

Description

Word synchronizers are used to provide word synchronization across clock domains. The crossings are handshaked, such that a second write cannot occur until the first is acknowledged (that the data has been received, but the value may not have been read) by the destination side. The destination read is registered.

Interfaces and Methods

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 different 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 tedious checks.

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		

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

Modules

The mkSyncReg, mkSyncRegToCC and mkSyncRegFromCC modules provide word synchronization across clock domains.

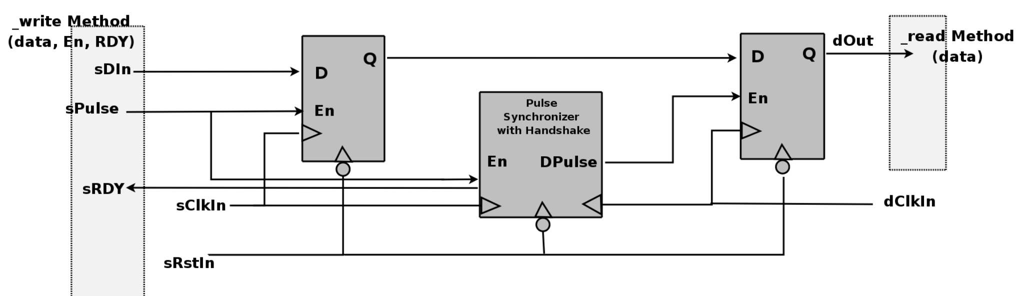

Figure 16: Register Synchronization Module (see Figure 15 for the pulse synchronizer with handshake)

mkSyncReg	Provides word synchronization across clock domains. The in and out clocks, along with the input reset, are explicitly provided. The default clock and reset are ignored.
	`module mkSyncReg #( a_type initValue, Clock sClkIn, Reset sRstIn, Clock dClkIn ) ( Reg #(a_type) ) provisos (Bits#(a_type, sa) ) ;`

`mkSyncRegFromCC`	Provides word synchronization from the current clock domain. The input clock and reset are the current clock and reset.
	`module mkSyncRegFromCC #( a_type initValue, Clock dClkIn ) ( Reg #(a_type) ) provisos (Bits#(a_type, sa)) ;`

```
module mkSyncRegFromCC #( a_type initValue,
                          Clock dClkIn )
                        ( Reg #(a_type) )
   provisos (Bits#(a_type, sa)) ;
```

`mkSyncRegToCC`	Provides word synchronization to the current clock domain. The output clock is the current clock. The current reset is ignored.
	`module mkSyncRegToCC #( a_type initValue, Clock sClkIn, Reset sRstIn ) ( Reg #(a_type) ) provisos (Bits#(a_type, sa)) ;`

```
module mkSyncRegToCC #( a_type initValue,
                        Clock sClkIn, Reset sRstIn )
                      ( Reg #(a_type) )
   provisos (Bits#(a_type, sa)) ;
```

Verilog Modules

The BSV modules correspond to the following Verilog modules, which are found in the Bluespec Verilog library, `$BLUESPECDIR/Verilog/`.

BSV Module Name	Verilog Module Name

`mkSyncReg`
`mkSyncRegFromCC`
`mkSyncRegToCC`
	`SyncRegister.v`

C.8.7 FIFO Synchronizers

Description

The FIFO synchronizers use FIFOs to synchronize data being sent across clock domains. Additional FIFO synchronizers, `SyncFIFOLevel` and `SyncFIFOCount` can be found in the `FIFOLevel` package (Section C.1.4).

Interfaces and Methods

The sync FIFO interface defines an interface similar to the FIFOF interface, except it does not have a `clear` method.

SyncFIFOIfc Interface				
Method			Arguments	
Name	Type	Description	Name	Description
enq	Action	Adds an entry to the FIFO	sendData	Data to be added
deq	Action	Removes the first entry from the FIFO		
first	a_type	Returns the first entry		
notFull	Bool	Returns True if there is space and you can enq into the FIFO		
notEmpty	Bool	Returns True if there are elements in the FIFO and you can deq from the FIFO		

```
interface SyncFIFOIfc #(type a_type) ;
   method Action enq ( a_type sendData ) ;
   method Action deq () ;
   method a_type first () ;
   method Bool notFull () ;
   method Bool notEmpty () ;
endinterface
```

Modules

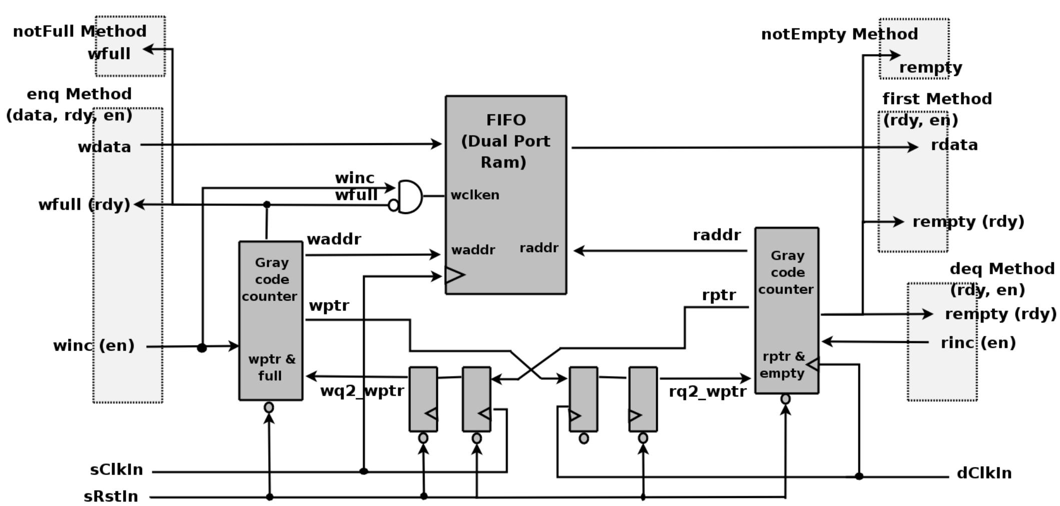

Figure 17: Synchronization FIFOs

The mkSyncFIFO, mkSyncFIFOFromCC and mkSyncFIFOToCC modules provide FIFOs for sending data across clock domains. Data items enqueued on the source side will arrive at the destination side and remain there until they are dequeued. The depth of the FIFO is specified by the depth parameter.

<table>
<tr><td rowspan="2"><code>mkSyncFIFO</code></td><td>Provides a FIFO for sending data across clock domains. The <code>enq</code> method is in the source (<code>sClkIn</code>) domain, while the <code>deq</code> and <code>first</code> methods are in the destination (<code>dClkIn</code>) domain. The in and out clocks, along with the input reset, are explicitly provided. The default clock and reset are ignored.</td></tr>
<tr><td><pre>module mkSyncFIFO #(Integer depth,
                     Clock sClkIn, Reset sRstIn,
                     Clock dClkIn)
                   (SyncFIFOIfc #(a_type))
   provisos (Bits#(a_type, sa));</pre></td></tr>
</table>

<table>
<tr><td rowspan="2"><code>mkSyncFIFOFromCC</code></td><td>Provides a FIFO to send data from the current clock domain into a second clock domain. The input clock and reset are the current clock and reset.</td></tr>
<tr><td><pre>module mkSyncFIFOFromCC #(Integer depth,
                           Clock dClkIn)
                         (SyncFIFOIfc #(a_type))
   provisos (Bits#(a_type, sa));</pre></td></tr>
</table>

<table>
<tr><td rowspan="2"><code>mkSyncFIFOToCC</code></td><td>Provides a FIFO to send data from a second clock domain into the current clock domain. The output clock is the current clock. The current reset is ignored.</td></tr>
<tr><td><pre>module mkSyncFIFOToCC #(Integer depth,
                         Clock sClkIn, Reset sRstIn)
                       (SyncFIFOIfc #(a_type))
   provisos (Bits#(a_type, sa));</pre></td></tr>
</table>

The sync FIFOFull modules are 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.

<table>
<tr><td rowspan="2"><code>mkSyncFIFOFull</code></td><td>Provides a registered FIFO for sending data across clock domains. The in and out clocks, along with the input reset, are explicitly provided. The default clock and reset are ignored.</td></tr>
<tr><td><pre>module mkSyncFIFOFull #(Integer depth,
                         Bool regEmpty,
                         Bool regFull,
                         Clock sClkIn, Reset sRstIn,
                         Clock dClkIn)
                       (SyncFIFOIfc #(a_type))
   provisos (Bits#(a_type, sa));</pre></td></tr>
</table>

<table>
<tr><td rowspan="2"><code>mkSyncFIFOFromCCFull</code></td><td>Provides a registered FIFO to send data from the current clock domain into a second clock domain. The input clock and reset are the current clock and reset.</td></tr>
<tr><td><pre>module mkSyncFIFOFromCCFull #(Integer depth,
                               Bool regEmpty,
                               Bool regFull,
                               Clock dClkIn)
                             (SyncFIFOIfc #(a_type))
   provisos (Bits#(a_type, sa));</pre></td></tr>
</table>

<table>
<tr><td rowspan="2"><code>mkSyncFIFOToCCFull</code></td><td>Provides a registered FIFO to send data from a second clock domain into the current clock domain. The output clock is the current clock. The current reset is ignored.</td></tr>
<tr><td><pre>module mkSyncFIFOToCCFull #(Integer depth,
                             Bool regEmpty,
                             Bool regFull,
                             Clock sClkIn, Reset sRstIn)
                           (SyncFIFOIfc #(a_type))
   provisos (Bits#(a_type, sa));</pre></td></tr>
</table>

Verilog Modules

The BSV modules correspond to the following Verilog modules, which are found in the Bluespec Verilog library, `$BLUESPECDIR/Verilog/`.

BSV Module Name	Verilog Module Name

`mkSyncFIFO`
`mkSyncFIFOFromCC`
`mkSyncFIFOFromCC`
`mkSyncFIFOFull`
`mkSyncFIFOFromCCFull`
`mkSyncFIFOToCCFull`
	`SyncFIFO.v`

C.8.8 Asynchronous RAMs

Description

An asynchronous RAM provides a domain crossing by having its read and write methods in separate clock domains.

Interfaces and Methods

DualPortRamIfc Interface				
Method			Arguments	
Name	Type	Description	Name	Description
write	Action	Writes data to a an address in a RAM	wr_addr	Address of datatype addr_t
			din	Data of datatype data_t
read	data_d	Reads the data from the RAM	rd_addr	Address to be read from

```
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
```

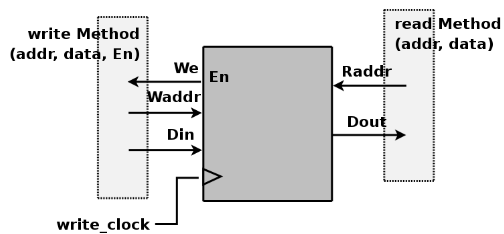

Figure 18: Ansynchronous RAM

mkDualRam	Provides an asynchronous RAM for when the read and the write methods are in separate clock domains. The write method is clocked by the default clock, the read method is not clocked.
	`module mkDualRam( DualPortRamIfc #(addr_t, data_t) ) provisos ( Bits#(addr_t, sa), Bits#(data_t, da) ) ;`

Verilog Modules

The BSV modules correspond to the following Verilog modules, which are found in the Bluespec Verilog library, $BLUESPECDIR/Verilog/.

BSV Module Name	Verilog Module Name
mkDualRam	DualPortRam.v

C.8.9 Null Crossing Primitives

Description

In these primitives, no synchronization is actually done. It is up to the designer to verify that it is safe for the signal to be used in the other domain. The mkNullCrossingWire is a wire synchronizer. The older mkNullCrossing primitive is deprecated.

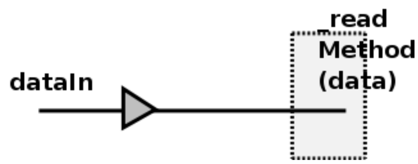

Figure 19: Wire synchronizer

Modules

The `mkNullCrossingWire` module, shown in Figure 19, uses the `ReadOnly` interface which is defined in the Prelude library B.4.7.

`mkNullCrossingWire`	Defines a synchronizer that contains only a wire. It is left up to the designer to ensure the clock crossing is safe.
	`module mkNullCrossingWire #( Clock dClk, a_type dataIn ) ( ReadOnly#(a_type) ) provisos (Bits#(a_type, sa)) ;`

Example: instantiating a null synchronizer

```
// domain2sig is domain1sig synchronized to clk0 with just a wire.
ReadOnly#(Bit#(2)) domain2sig <- mkNullCrossingWire (clk0, domain1sig);
```

Note: no synchronization is actually done. This is purely a way to tell BSC that it is safe to use the signal in the other domain. It is the responsibility of the designer to verify that this is correct.

There are some restrictions on the use of a `mkNullCrossingWire`. The expression used as the data argument must not have an implicit condition, and there cannot be another rule which is required to schedule before any method called in the expression.

`mkNullCrossingWire`s may not be used in sequence to pass a signal across multiple clock boundaries without synchronization. Once a signal has been crossed from one domain to a second domain without synchronization, it cannot be subsequently passed unsynchronized to a third domain (or back to the first domain).

Verilog Modules

The BSV modules correspond to the following Verilog modules, which are found in the Bluespec Verilog library, `$BLUESPECDIR/Verilog/`.

BSV Module Name	Verilog Module Name
`mkNullCrossingWire`	`BypassWire.v`

C.8.10 Specialized Crossing Primitives

Description

The `mkSyncRegToSlow` and `mkSyncRegToFast` 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 (Section C.8.3).

The crossing primitive is implemented via a single register, clocked by the slower (divided) clock. For a fast to slow crossing, the register is only writable when the `clockReady` bit of the divider interface is asserted. This is an implicit condition of the write method module which prevents erroneous writes. For a slow to fast crossing both the read and write methods are always available.

Modules

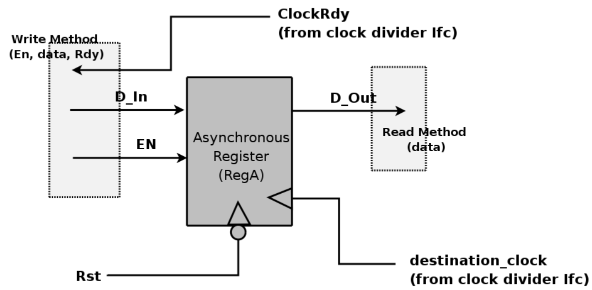

Figure 20: Fast to Slow Crossing

`mkSyncRegToSlow`	Provides a register to transport data when the clock edges are aligned between domains. This module moves data from a fast to a slow domain. The register is only writable when the `clockReady` bit of the divider is asserted.
	`module mkSyncRegToSlow #( a_type initValue, ClockDividerIfc divider, Reset slowRstIn ) ( Reg #(a_type) ) provisos (Bits#(a_type, sa)) ;`

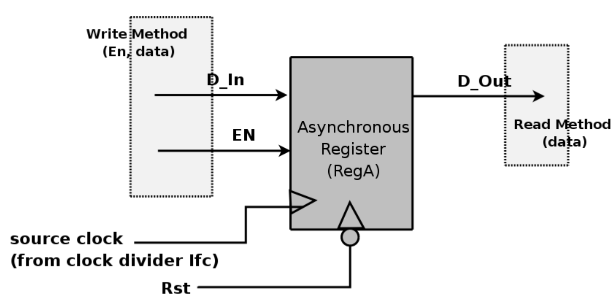

Figure 21: Slow to Fast Crossing

`mkSyncRegToFast`	Provides a register to transport data when the clock edges are aligned between domains. This module moves data from a slow to a fast domain. The read and write methods are always available.
	`module mkSyncRegToFast #( a_type initValue, ClockDividerIfc divider, Reset slowRstIn ) ( Reg #(a_type) ) provisos (Bits#(a_type, sa)) ;`

The `mkSyncFIFOToSlow` and `mkSyncFIFOToFast` modules are specialized crossing primitives which can be used to transport data when clock edges are aligned, between a fast clock domain and a slower clock domain. The derived clock and the `ClkNextRdy` signal would typically be generated using the `mkClockDivider` module. The synchronous FIFOs are clocked by the slower (divided) clock. The `SyncFIFOIfc` is detailed in Section C.8.7.

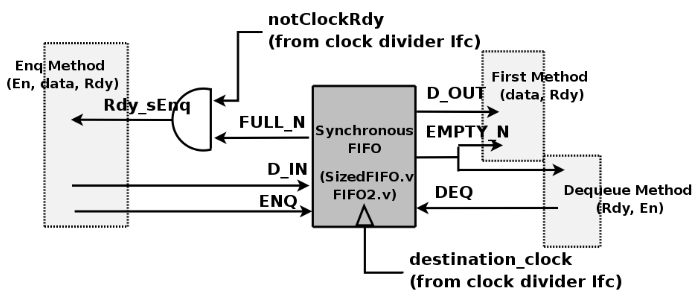

Figure 22: Aligned clocks with FIFO - to slower domain

`mkSyncFIFOToSlow`	Provides a FIFO with specified depth to transport data from a fast clock domain to a slower clock domain when clock edges are aligned. The crossing primitive is implemented via a FIFO with the specified depth clocked by `dClkIn`. The FIFO is enqueued only when the `syncBit` is asserted and the FIFO is not full.
	`module mkSyncFIFOToSlow #( Integer depth, ClockDividerIfc divider, Reset slowRstIn ) ( SyncFIFOIfc #(a_type) ) provisos (Bits#(a_type, sa)) ;`

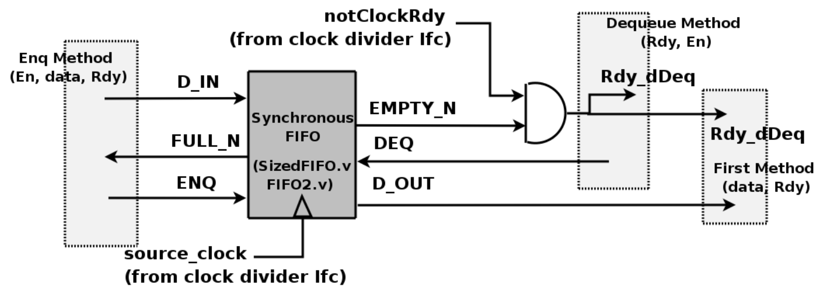

Figure 23: Aligned clocks with FIFO - to faster domain

`mkSyncFIFOToFast`	Provides a FIFO with specified depth to transport data from a slower clock domain to a faster clock domain when clock edges are aligned. The crossing primitive is implemented via a FIFO with the specified depth clocked by sClkIn (the source clock is the slower clock). The FIFO is dequeued only when the `syncBit` is asserted and the FIFO is not empty.
	`module mkSyncFIFOToFast #( Integer depth, ClockDividerIfc divider, Reset slowRstIn ) ( SyncFIFOIfc #(a_type) ) provisos (Bits#(a_type, sa)) ;`

Verilog Modules

The BSV modules correspond to the following Verilog modules, which are found in the Bluespec Verilog library, `$BLUESPECDIR/Verilog/`.

BSV Module Name	Verilog Module Name
`mkSyncRegToSlow` `mkSyncRegToFast`	`RegA.v`
`mkSyncFIFOToSlow` `mkSyncFIFOToFast`	`FIFO2.v` `SizedFIFO.v`

C.8.11 Reset Synchronization and Generation

Description

This section describes the interfaces and modules used to synchronize reset signals from one clock domain to another and to create reset signals. Reset generation converts a Boolean type to a Reset type, where the reset is associated with the default or `clocked_by` clock domain.

Interfaces and Methods

The `MakeResetIfc` interface is provided by the reset generators `mkReset` and `mkResetSync`.

MakeResetIfc Interface		
Method		
Name	Type	Description
assertReset	Action	Method used to assert the reset
isAsserted	Bool	Indicates whether the reset is asserted
new_rst	Reset	Generated output reset

```
interface MakeResetIfc;
   method Action assertReset();
   method Bool isAsserted();
   interface Reset new_rst;
endinterface
```

The interface `MuxRstIfc` is provided by the `mkResetMux` module.

MuxRstIfc Interface				
Method			Arguments	
Name	Type	Description	Name	Description
select	Action	Method used to select the reset based on the Boolean value ab	ab	Value determines which input reset to select
reset_out	Reset	Generated output reset		

```
interface MuxRstIfc;
   method Action select ( Bool ab );
   interface Reset reset_out;
endinterface
```

Modules

Reset Synchronization To synchronize resets from one clock domain to another, both synchronous and asynchronous modules are provided. The `stages` argument is the number of full clock cycles the output reset is held for after the input reset is deasserted. This is shown as the number of flops in figures 24 and 25. Specifying a 0 for the `stages` argument results in the creation of a simple wire between `sRst` and `dRstOut`.

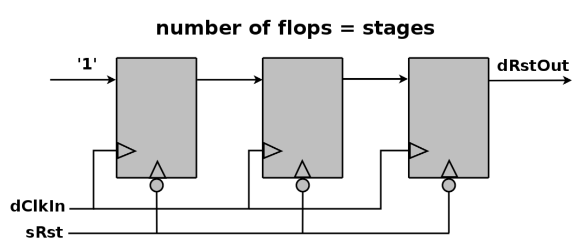

Figure 24: Module for asynchronous resets

`mkAsyncReset`	Provides synchronization of a source reset (`sRst`) to the destination domain. The output reset occurs immediately once the source reset is asserted.

	`module mkAsyncReset #( Integer stages,`
	`Reset sRst,`
	`Clock dClkIn )`
	`( Reset ) ;`

`mkAsyncResetFromCR`	Provides synchronization of the current reset to the destination domain. There is no source reset `sRst` argument because it is taken from the current reset. The output reset occurs immediately once the current reset is asserted.

	`module mkAsyncResetFromCR #( Integer stages,`
	`Clock dClkIn )`
	`( Reset ) ;`

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 detected. Both `mkSyncReset` and `mkSyncResetFromCR` use the model in figure 25.

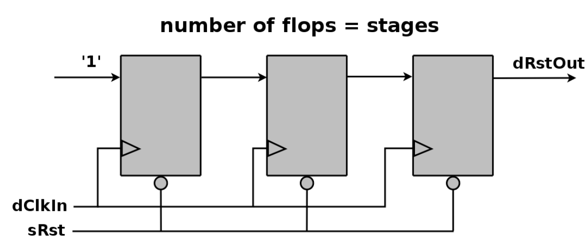

Figure 25: Module for synchronous resets

`mkSyncReset`	Provides synchronization of a source reset (`sRst`) to the destination domain. The reset is asserted at the next rising edge of the clock.

	`module mkSyncReset #( Integer stages`
	`Reset sRst,`
	`Clock dClkIn )`
	`( Reset ) ;`

<table>
<tr><td rowspan="2"><code>mkSyncResetFromCR</code></td><td>Provides synchronization of the current reset to the destination domain. The reset is asserted at the next rising edge of the clock.</td></tr>
<tr><td><pre>module mkSyncResetFromCR #(Integer stages
                            Clock dClkIn)
                          (Reset) ;</pre></td></tr>
</table>

Example: instantiating a reset synchronizer

```
// 2 is the number of stages
Reset rstn2 <- mkAsyncResetFromCR (2, clk0);

// if stages = 0, the default reset is used directly
Reset rstn0 <- mkAsyncResetFromCR (0, clk0);
```

Reset Generation Two modules are provided for reset generation, `mkReset` and `mkResetSync`, where each module has one parameter, `stages`. The `stages` parameter is the number of full clock cycles the output reset is held after the `inRst`, as seen in figure 26, is deasserted. Specifying a 0 for the `stages` parameter results in the creation of a simple wire between the input register and the output reset. That is, the reset is asserted immediately and not held after the input reset is deasserted. It becomes the designer's responsibility to ensure that the input reset is asserted for sufficient time to allow the design to reset properly. The reset is controlled using the `assertReset` method of the `MakeResetIfc` interface.

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 use of `mkResetSync` is less common, since the reset requires clock edges to take effect; failure to assert reset for a clock edge will result in a reset not being seen at the output reset.

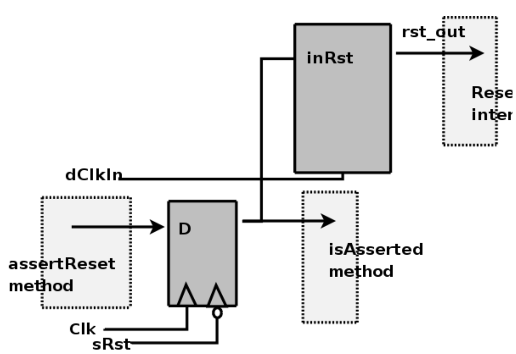

Figure 26: Module for generating resets

`mkReset`	Provides conversion of a Boolean type to a Reset type, where the reset is associated with `dClkIn`. This module uses the model in figure 26. `startInRst` indicates the reset value of the register. If `startInRst` is True, the reset value of the register is 0, which means the output reset will be asserted whenever the currentReset (`sRst`) is asserted. `rst_out` will remain asserted for the number of clock cycles given by the stages parameter after `sRst` is deasserted. If `startInRst` is False, the output reset will not be asserted when `sRst` is asserted, but only when the `assert_reset` method is invoked. At the start of simulation `rst_out` will only be asserted if `startinRst` is True and `sRst` is initially asserted.

	`module mkReset #( Integer stages,`
	`                  Bool startInRst,`
	`                  Clock dClkIn )`
	`                ( MakeResetIfc ) ;`

`mkResetSync`	Provides conversion of a Boolean type to a Reset type, where the reset is associated with `dClkIn` and the assertion of reset is at the next rising edge of the clock. This module uses the model in figure 26. `startInRst` indicates the reset value of the register. If `startInRst` is True, the reset value of the register is 0, which means the output reset will be asserted whenever the currentReset (`sRst`) is asserted. `rst_out` will remain asserted for the number of clock cycles given by the stages parameter after `sRst` is deasserted. If `startInRst` is False, the output reset will not be asserted when `sRst` is asserted, but only when the `assert_reset` method is invoked. At the start of simulation `rst_out` will only be asserted if `startinRst` is True and `sRst` is initially asserted.

	`module mkResetSync #( Integer stages,`
	`                      Bool startInRst,`
	`                      Clock dClkIn )`
	`                    ( MakeResetIfc ) ;`

A reset multiplexor `mkResetMux`, as seen in figure 27, creates one reset signal by selecting between two existing reset signals.

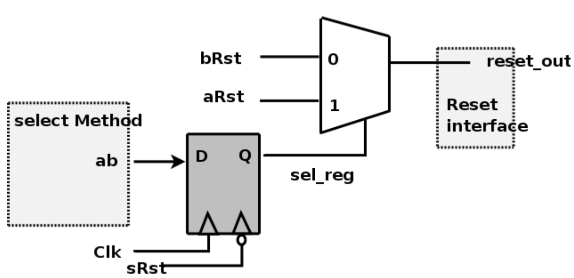

Figure 27: Reset Multiplexor

mkResetMux	Multiplexor which selects between two input resets, aRst and bRst, to create a single output reset rst_out. The reset is selected through a Boolean value provided to the select method where True selects aRst.	
	`module mkResetMux #( Reset aRst, Reset bRst ) ( MuxRstIfc rst_out ) ;`	

For testbenches, in which an absolute clock is being created, it is helpful to generate a reset for that clock. The module `mkInitialReset` is available for this purpose. It generates a reset which is asserted at the start of simulation. The reset is asserted for the number of cycles specified by the parameter `cycles`, counting the start of time as 1 cycle. Therefore, a `cycles` value of 1 will cause the reset to turn off at the first clock tick. This module is not synthesizable.

mkInitialReset	Generates a reset for cycles cycles, where the cycles parameter must be greater than zero. The clocked_by clause indicates the clock the reset is associated with. This module is not synthesizable.
	`module mkInitialReset #( Integer cycles ) ( Reset ) ;`

Example:

```
Clock c <- mkAbsoluteClock (10, 5);
// a reset associated with clock c:
Reset r <- mkInitialReset (2, clocked_by c);
```

When two reset signals need to be combined so that some logic can be reset when either input reset is asserted, the `mkResetEither` module can be used.

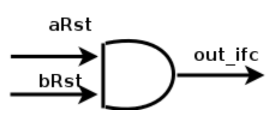

Figure 28: Reset Either

mkResetEither	Generates a reset which is asserted whenever either input reset is asserted.
	`module mkResetEither ( Reset aRst, Reset bRst) ( Reset out_ifc );`

Example:

```
Reset r <- mkResetEither(rst1, rst2);
```


`mkResetInverter`	Generates an inverted Reset.	
	`module mkResetInverter#(Reset in)` `(Reset);`	

Verilog Modules

The BSV modules correspond to the following Verilog modules, which are found in the Bluespec Verilog library, `$BLUESPECDIR/Verilog/`.

BSV Module Name	Verilog Module Name	Comments
`mkASyncReset`	`SyncReset0.v`	when stages==0
`mkASyncResetFromCR`	`SyncResetA.v`	
`mkSyncReset`	`SyncReset0.v`	when stages==0
`mkSyncResetFromCR`	`SyncReset.v`	
`mkReset`	`MakeReset0.v`	when stages==0
	`MakeResetA.v`	instantiates `SyncResetA`
`mkResetSync`	`MakeReset0.v`	when stages==0
	`MakeReset.v`	instantiates `SyncReset`
`mkResetMux`	`ResetMux.v`	
`mkResetEither`	`ResetEither.v`	
`mkResetInverter`	`ResetInverter.v`	

C.9 Special Collections

C.9.1 ModuleCollect

Package

import ModuleCollect :: * ;

Description

The `ModuleCollect` package provides the capability of adding additional items, such as configuration bus connections, to a design in such a way that it does not change the structure of the design. This section provides a brief overview of the package. For more description of its usage, see the `CBus` package (C.9.2), which utilizes `ModuleCollect`. There is also a detailed example and more complete discussion of the `CBus` package in the configbus tutorial in the BSV/tutorials directory.

An ordinary Bluespec module, when instantiated, adds its own state elements and rules to the growing accumulation of state elements and rules defined in the design. In some designs, for example a configuration bus, additional items, such as the logic for the bus address decoding must be accumulated as well. While there is a need to add these items, it is also desirable to keep these additional design details separate from the main design, keeping the natural structure of the design intact.

The `ModuleCollect` mechanism allows the designer to *hide* the details of the additional interfaces. A module which is going to be synthesized must contain only rules and state elements, as the compiler does not know how to handle the additional items. Therefore, the collection must be brought into the open, or exposed, before the module can be synthesized. The `ModuleCollect` package provides the mechanisms to allow these additional items to be collected, processed and exposed.

Types and Type Classes

The `ModuleCollect` type is a variation on `Module` that allows additional items, other than states and rules, to be collected while elaborating the module structure. A module defining the accumulation of a special collection will have the type of `ModuleCollect` which is defined as follows:

```
struct ModuleCollect#(a_type, ifc)
      ··· abstract ···
```

where `a_type` defines the type of the items being collected. The collection is kept as a `List`, therfore each item in the collection must have the same type. The collection is associated with `ifc`, the device module interface.

Your new type of module is a `ModuleCollect` defined to collect a specific type. It is often convenient to give a name to your new type of module using the `typedef` keyword.

For example:

```
typedef ModuleCollect#(element_type, ifc_device)
        MyModuleType#(type ifc_device)
```

specifies a type named `MyModuleType`.

An ordinary module, one not collecting anything other than rules and state elements has the type `Module`. When no type is explicitly given, the compiler fixes it to `Module` when the module is synthesized. But for a module accumulating a collection, the type must be explicitly given, and it is supplied in square brackets immediately after the keyword `module`. Because in our example above, the new type alias only takes one argument, the interface, we can use it here without arguments:

```
module [MyModuleType] mkSubDesign#(x,y) (IfcType) ;
```

Since only modules of type `Module` can be synthesized the collection be exposed before synthesis, by applying the function `exposeCollection`. The module type of the function `exposeCollection` is `Module`, so once the collection has been exposed the design is ready for synthesis.

Interfaces

The `IWithCollection` interface couples the normal module interface (the `device` interface) with the collection of collected items (the `collection` interface). This is the interface provided by the `exposeCollection` function. It separates the collection list and the device module interface, to allow the module to be synthesized.

```
interface IWithCollection #(type collection_type, type item_type);
   interface item_type device();
   interface List#(collection_type) collection();
endinterface: IWithCollection
```

Modules and Functions

In the course of evaluating a module body during its instantiation, an item may be added to the current collection by using the function `addToCollection`.

`addToCollection`	Adds an item to the collection.
	`function ModuleCollect#(a_type, ifc)` `        addToCollection(a_type item);`

Once a set of items has been collected, those items must be exposed before synthesis. The `exposeCollection` module constructor is used to bring the collection out into the open. The `exposeCollection` module takes as an argument a `ModuleCollect` module (`m`) with interface `ifc`, and provides an `IWithCollection` interface.

`exposeCollection`	Expose the collection to allow the module to be synthesized.
	`module exposeCollection#(ModuleCollect#(a_type, ifc) m)` `                        (IWithCollection#(a_type, ifc));`

Finally, the `ModuleCollect` package provides a function, `mapCollection`, to apply a function to each item in the current collection.

`mapCollection`	Apply a function to each item added to the collection within the second argument.

	`function ModuleCollect#(a_type, ifc)`
	`   mapCollection(function a_type x1(a_type x1),`
	`                 ModuleCollect#(a_type, ifc) x2);`

Example - Assertion Wires

```
// This example shows excerpts of a design which places various
// test conditions (Boolean expressions) at random places in a design,
// and lights an LED (setting an external wire to 1), if the condition
// is ever satisfied.

import ModuleCollect::*;
import List::*;
import Vector::*;
import Assert::*;

// The desired interface at the top level is:
interface AssertionWires#(type n);
   method Bit#(n) wires;
   method Action clear;
endinterface

// The "wires" method tells which conditions have been set, and the
// "clear" method resets them all to 0.
// The items in our extra collection will be interfaces of the
// following type:

interface AssertionWire;
   method Integer index;   //Indicates which wire is to be set if
   method Bool fail;       // fail method ever returns true.
   method Action clear;
endinterface

// We next define the "AssertModule" type.  This is to behave like an
// ordinary module providing an interface of type "i", except that it
// also can collect items of type "AssertionWire":
```

```
typedef ModuleCollect#(AssertionWire, i) AssertModule#(type i);

typedef Tuple2#(AssertionWires#(n), i) AssertIfc#(type i, type n);

...

// The next definition shows how items are added to the collection.
// This is the module which will be instantiated at various places in
// the design, to test various conditions.  It takes one static
// parameter, "ix", to specify which wire is to carry this condition,
// and one dynamic parameter (one varying at run-time) "c", giving the
// value of the condition itself.

interface AssertionReg;
   method Action set;
   method Action clear;
endinterface

module [AssertModule] mkAssertionReg#(Integer ix)(AssertionReg);

   Reg#(Bool) cond <- mkReg(False);

   // an item is defined and added to the collection
   let item = (interface AssertionWire;
                 method index;
                    return (ix);
                 endmethod
                 method fail;
                    return(cond);
                 endmethod
                 method Action clear;
                     cond <= False;
                 endmethod
               endinterface);
   addToCollection(item);
   ...
endmodule

// the collection must be exposed before synthesis
module [Module] exposeAssertionWires#(AssertModule#(i) mkI)(AssertIfc#(i, n));

   IWithCollection#(AssertionWire, i) ecs <- exposeCollection(mkI);

   ...(c_ifc is created from the list ecs.collection)

   // deliver the array of values in the registers
   let dut_ifc = ecs.device;

   // return the values in the collection, and the ifc of the device
   return(tuple2(c_ifc, dut_ifc));
endmodule
```


C.9.2 CBus

Package

import CBus :: * ;

Description

The CBus package provides the interface, types and modules to implement a configuration bus capability providing access to the control and status registers in a given module hierarchy. This package utilizes the ModuleCollect package and functionality, as described in section C.9.1. The ModuleCollect package allows items in addition to usual state elements and rules to be accumulated. This is required to collect up the interfaces of the control status registers included in a module and to add the associated logic and ports required to allow them to be accessed via a configuration bus.

This package is provided as both a compiled library package and as BSV source code to facilitate customization. The source code file can be found in the $BLUESPECDIR/BSVSource directory. To customize a package, copy the file into a local directory and then include the local directory in the path when compiling. This is done by specifying the path with the -p option as described in the BSV Users Guide.

For a more complete discussion of the CBus package, consult the configbus tutorial in the BSV/tutorials directory.

Types and Type Classes

The type CBusItem defines the type of item to be collected by ModuleCollect. The items to be collected are the same as the ifc which we will later expose, so we use a type alias:

```
typedef CBus#(size_address, size_data)
        CBusItem #(type size_address, type size_data);
```

The type ModWithCBus defines the type of module which is collecting CBusItems. An ordinary module, one not collecting anything other than state elements and rules, has the type Module. Since CBusItems are being collected, a module type ModWithCBus is defined. When the module type is not Module, the type must be specified in square brackets immediately after the module keyword in the module definition.

```
typedef ModuleCollect#(CBusItem#(size_address, size_data), item)
        ModWithCBus#(type size_address, type size_data, type item);
```

Interface and Methods

The CBus interface provides read and write methods to access control status registers. It is polymorphic in terms of the size of the address bus (size_address) and size of the data bus (size_data).

CBus Interface	
Name	Description
write	Writes the data value to the register if and only if the value of addr matches the address of the register.
read	Returns the value of the associated register if and only if addr matches the register address. In all other cases the read method returns an Invalid value.


```
interface CBus#(type size_address, type size_data);
   method Action write(Bit#(size_address) addr, Bit#(size_data) data);
   (* always_ready *)
   method ActionValue#(Bit#(size_data)) read(Bit#(size_address) addr);
endinterface
```

The `IWithCBus` interface combines the `CBus` interface with a normal module interface. It is defined as a structured interface with two sub-interfaces: `cbus_ifc` (the associated configuration bus interface) and `device_ifc` (the associated device interface). It is polymorphic in terms of the type of the configuation bus interface and the type of the device interface.

```
interface IWithCBus#(type cbus_IFC, type device_IFC);
   interface cbus_IFC cbus_ifc;
   interface device_IFC device_ifc;
endinterface
```

Modules

The `collectCBusIFC` module takes as an argument a module with an `IWithCBus` interface, adds the associated `CBus` interface to the current collection (using `addToCollection` from the `ModuleCollect` package), and returns a module with the normal interface. Note that `collectCBusIFC` is of module type `ModWithCBus`.

<table>
<tr><td rowspan="2"><code>collectCBusIFC</code></td><td>Adds the <code>CBus</code> to the collection and returns a module with just the device interface.</td></tr>
<tr><td><pre>module [ModWithCBus#(size_address, size_data)]
        collectCBusIFC#(Module#(IWithCBus#(
                           CBus#(size_address,size_data),i)) m)(i);</pre></td></tr>
</table>

The `exposeCBusIFC` module is used to create an `IWithCBus` interface given a module with a normal interface and an associated collection of `CBusItem`s. This module takes as an argument a module of type `ModWithCBus` and provides an interface of type `IWithCBus`. The `exposeCBusIFC` module exposes the collected `CBusItem`s, processes them, and provides a new combined interface. This module is synthesizable, because it is of type `Module`.

<table>
<tr><td rowspan="2"><code>exposeCBusIFC</code></td><td>A module wrapper that takes a module with a normal interface, processes the collected CBusItems and provides an IWithCBus interface.</td></tr>
<tr><td><pre>module [Module] exposeCBusIFC#(ModWithCBus#(
           size_address, size_data, item) sm)
           (IWithCBus#(CBus#(size_address, size_data), item));</pre></td></tr>
</table>

The `CBus` package provides a set of module primitives each of which adds a `CBus` interface to the collection and provides a normal `Reg` interface from the local block point of view. These modules are used in designs where a normal register would be used, and can be read and written to as registers from within the design.

<table>
<tr><td rowspan="2"><code>mkCBRegR</code></td><td>A wrapper to provide a read only <code>CBus</code> interface to the collection and a normal <code>Reg</code> interface to the local block.</td></tr>
<tr><td><pre>module [ModWithCBus#(size_address, size_data)]
       mkCBRegR#(CRAddr#(size_address2) addr, r x)
                 (Reg#(r))
   provisos (Bits#(r, sr), Add#(k, sr, size_data),
             Add#(ignore, size_address2, size_address));</pre></td></tr>
</table>

<table>
<tr><td rowspan="2"><code>mkCBRegRW</code></td><td>A wrapper to provide a read/write <code>CBus</code> interface to the collection and a normal <code>Reg</code> interface to the local block.</td></tr>
<tr><td><pre>module [ModWithCBus#(size_address, size_data)]
       mkCBRegRW#(CRAddr#(size_address2) addr, r x)
                 (Reg#(r))
   provisos (Bits#(r, sr), Add#(k, sr, size_data),
             Add#(ignore, size_address2, size_address));</pre></td></tr>
</table>

<table>
<tr><td rowspan="2"><code>mkCBRegW</code></td><td>A wrapper to provide a write only <code>CBus</code> interface to the collection and a normal <code>Reg</code> interface to the local block.</td></tr>
<tr><td><pre>module [ModWithCBus#(size_address, size_data)]
       mkCBRegW#(CRAddr#(size_address2) addr, r x)
                 (Reg#(r))
   provisos (Bits#(r, sr), Add#(k, sr, size_data),
             Add#(ignore, size_address2, size_address));</pre></td></tr>
</table>

<table>
<tr><td rowspan="2"><code>mkCBRegRC</code></td><td>A wrapper to provide a read/clear <code>CBus</code> interface to the collection and a normal <code>Reg</code> interface to the local block. This register can read from the config bus but the write is clear mode; for each write bit a 1 means clear, while a 0 means don't clear.</td></tr>
<tr><td><pre>module [ModWithCBus#(size_address, size_data)]
       mkCBRegRC#(CRAddr#(size_address2) addr, r x)
                 (Reg#(r))
   provisos (Bits#(r, sr), Add#(k, sr, size_data),
             Add#(ignore, size_address2, size_address));</pre></td></tr>
</table>

The `mkCBRegFile` module wrapper adds a `CBus` interface to the collection and provides a `RegFile` interface to the design. This module is used in designs as a normal `RegFile` would be used.

`mkCBRegFile`	A wrapper to provide a normal `RegFile` interface and automatically add the `CBus` interface to the collection.
	`module [ModWithCBus#(size_address, size_data)] mkCBRegFile#(Bit#(size_address) reg_addr, Bit#(size_address) size) (RegFile#(Bit#(size_address), r)) provisos (Bits#(r, sr), Add#(k, sr, size_data));`

Example

Provided here is a simple example of a CBus implementation. The example is comprised of three packages: `CfgDefines`, `Block`, and `Tb`. The `CfgDefines` package contains the definition for the configuration bus, `Block` is the design block, and `Tb` is the testbench which executes the block.

The `Block` package contains the local design. As seen in Figure 29, the configuration bus registers look like a single field from the CBus (`cfgResetAddr, cfgStateAddr, cfgStatusAddr`), while each field (`reset, init, cnt`, etc.) in the configuration bus registers looks like a regular register from from the local block point of view.

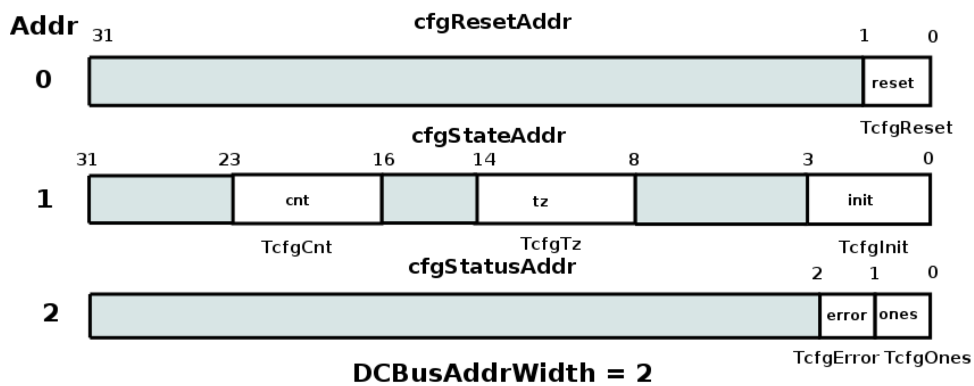

Figure 29: CBus Registers used in Block example

```
import CBus::*;       // this is a Bluespec library
import CfgDefines::*; // user defines - address,registers, etc

interface Block;
   // TODO: normally this block would have at least a few methods
   // Cbus interface is hidden, but it is there
endinterface

// In order to access the CBus at this parent, we need to expose the bus.
// Only modules of type [Module] can be synthesized.
module [Module] mkBlock(IWithCBus#(DCBus, Block));
   let ifc <- exposeCBusIFC( mkBlockInternal );
   return ifc;
endmodule

// Within this module the CBus looks like normal Registers.
// This module can't be synthesized directly.
```



```
// How these registers are combined into CBus registers is
// defined in the CfgDefines package.

module [DModWithCBus] mkBlockInternal( Block );
   // all registers are read/write from the local block point of view
   // config register interface types can be
   //   mkCBRegR  -> read only from config bus
   //   mkCBRegRW -> read/write from config bus
   //   mkCBRegW  -> write only from config bus
   //   mkCBRegRC -> read from config bus, write is clear mode
   //                i.e. for each bit a 1 means clear, 0 means don't clear
   // reset bit is write only from config bus
   // we presume that you use this bit to fire some local rules, etc
   Reg#(TCfgReset)  reg_reset_reset    <- mkCBRegW(cfg_reset_reset,    0 /* init val */);

   Reg#(TCfgInit)   reg_setup_init     <- mkCBRegRW(cfg_setup_init,    0 /* init val */);
   Reg#(TCfgTz)     reg_setup_tz       <- mkCBRegRW(cfg_setup_tz,      0 /* init val */);
   Reg#(TCfgCnt)    reg_setup_cnt      <- mkCBRegRW(cfg_setup_cnt,     1 /* init val */);

   Reg#(TCfgOnes)   reg_status_ones    <- mkCBRegRC(cfg_status_ones,   0 /* init val */);
   Reg#(TCfgError)  reg_status_error   <- mkCBRegRC(cfg_status_error,  0 /* init val */);

   // USER: you know have registers, so do whatever it is you do with registers :)
   // for instance
   rule bumpCounter ( reg_setup_cnt != unpack('1) );
      reg_setup_cnt <= reg_setup_cnt + 1;
   endrule

   rule watch4ones ( reg_setup_cnt == unpack('1) );
      reg_status_ones <= 1;
   endrule
endmodule
```

The CfgDefines package contains the user defines describing how the local registers are combined into the configuration bus.

```
package CfgDefines;
import CBus::*;

////////////////////////////////////////////////////////////////////////////////
/// basic defines
////////////////////////////////////////////////////////////////////////////////
// width of the address bus, it's easiest to use only the width of the bits needed
// but you may have other reasons for passing more bits around (even if some address
// bits are always 0)
typedef  2 DCBusAddrWidth;  // roof( log2( number_of_config_registers ) )

// the data bus width is probably defined in your spec
typedef 32 DCBusDataWidth;  // how wide is the data bus

////////////////////////////////////////////////////////////////////////////////
// Define the CBus
////////////////////////////////////////////////////////////////////////////////
typedef CBus#(  DCBusAddrWidth,DCBusDataWidth)          DCBus;
```

```
typedef CRAddr#(DCBusAddrWidth,DCBusDataWidth)          DCAddr;
typedef ModWithCBus#(DCBusAddrWidth, DCBusDataWidth, i) DModWithCBus#(type i);

////////////////////////////////////////////////////////////////////////////////
/// Configuration Register Types
////////////////////////////////////////////////////////////////////////////////
// these are configuration register from your design.  The basic
// idea is that you want to define types for each individual field
// and later on we specify which address and what offset bits these
// go to.  This means that config register address fields can
// actually be split across modules if need be.
//
typedef bit      TCfgReset;

typedef Bit#(4)  TCfgInit;
typedef Bit#(6)  TCfgTz;
typedef UInt#(8) TCfgCnt;

typedef bit      TCfgOnes;
typedef bit      TCfgError;

////////////////////////////////////////////////////////////////////////////////
/// configuration bus addresses
////////////////////////////////////////////////////////////////////////////////
Bit#(DCBusAddrWidth) cfgResetAddr  = 0; //
Bit#(DCBusAddrWidth) cfgStateAddr  = 1; //
Bit#(DCBusAddrWidth) cfgStatusAddr = 2; // maybe you really want this to be 0,4,8 ???

////////////////////////////////////////////////////////////////////////////////
/// Configuration Register Locations
////////////////////////////////////////////////////////////////////////////////
// DCAddr is a structure with two fields
//     DCBusAddrWidth a ; // this is the address
//                        // this does a pure comparison
//     Bit#(n)        o ; // this is the offset that this register
//                        // starts reading and writting at

DCAddr cfg_reset_reset  = DCAddr {a: cfgResetAddr, o:  0};  // bits 0:0

DCAddr cfg_setup_init   = DCAddr {a: cfgStateAddr, o:  0};  // bits 0:0
DCAddr cfg_setup_tz     = DCAddr {a: cfgStateAddr, o:  4};  // bits 9:4
DCAddr cfg_setup_cnt    = DCAddr {a: cfgStateAddr, o: 16};  // bits 24:16

DCAddr cfg_status_ones  = DCAddr {a: cfgStatusAddr, o:  0};  // bits 0:0
DCAddr cfg_status_error = DCAddr {a: cfgStatusAddr, o:  0};  // bits 1:1

////////////////////////////////////////////////////////////////////////////////
///
////////////////////////////////////////////////////////////////////////////////
endpackage
```

The `Tb` package executes the block.

```
import CBus::*;        // bluespec library
```



```
import CfgDefines::*;  // address defines, etc
import Block::*;       // test block with cfg bus
import StmtFSM::*;     // just for creating a test sequence

(* synthesize *)
module mkTb ();
   // In order to access this cfg bus we need to use IWithCBus type
   IWithCBus#(DCBus,Block) dut <- mkBlock;

   Stmt test =
   seq
      // write the bits need to the proper address
      // generally this comes from software or some other packing scheme
      // you can, of course, create functions to pack up several fields
      // and drive that to bits of the correct width
      // For that matter, you could have your own shadow config registers
      // up here in the testbench to do the packing and unpacking for you
      dut.cbus_ifc.write( cfgResetAddr, unpack('1) );

      // put some ones in the status bits
      dut.cbus_ifc.write( cfgStateAddr, unpack('1) );

      // show that only the valid bits get written
      $display("TOP: state = %x at ", dut.cbus_ifc.read( cfgStateAddr ), $time);

      // clear out the bits
      dut.cbus_ifc.write( cfgStateAddr, 0 );

      // but the 'ones' bit was set when it saw all ones on the count
      // so read it to see that...
      $display("TOP: status = %x at ", dut.cbus_ifc.read( cfgStatusAddr ), $time);

      // now clear it
      dut.cbus_ifc.write( cfgStatusAddr, 1 );

      // see that it's clear
      $display("TOP: status = %x at ", dut.cbus_ifc.read( cfgStatusAddr ), $time);

      // and if we had other interface methods, that where not part of CBUS
      // we would access them via dut.device_ifc
   endseq;
   mkAutoFSM( test );
endmodule
```

C.10 AzureIP Libraries

This section describes the Bluespec AzureIP library components. These components can be used to build complex, fully synthesizable designs. Each component is provided in one or more BSV packages, defining the interfaces and data structures used to communicate to other components.

These library components are provided as BSV source code to facilitate customization. Users can easily understand and then extend the IP to implement additional features as required for their applications. The source code files can be found in the `$BLUESPECDIR/BSVSource` directory. To modify the files, copy the files into a local directory and use the -p compile option, as described in the BSV Users Guide, to include the local directory in your path.

C.10.1 TLM

Description

The TLM package includes definitions of interfaces, data structures, and module constructors which allow users to create and modify bus-based designs in a manner that is independent of any one specific bus protocol. Bus operations are defined in terms of generic bus payload data structures. Other protocol specific packages include transactor modules that convert a stream of TLM bus operations into corresponding operations in a specific bus protocol. Designs created using the TLM package are thus more portable (because that they allow the core design to be easily applied to multiple bus protocols). In addition, since the specific signalling details of each bus protocol are encapsulated in pre-designed transactors, users are not required to learn, re-implement, and re-verify existing standard protocols.

Packages

The elements of the TLM library are defined within TLM package.

To include the package in your design, use the `import` syntax.

```
import TLM :: * ;
```

Data Structures

The two basic data structures defined in the `TLM` package are `TLMRequest` and `TLMResponse`. By using these types in a design, the underlying bus protocol can be changed without having to modify the interactions with the `TLM` objects.

TLMRequest A TLM request contains either control information and data, or data alone. A `TLMRequest` is tagged as either a `RequestDescriptor` or `RequestData`. A `RequestDescriptor` contains control information and data while a `RequestData` contains only data.

```
typedef union tagged {RequestDescriptor#(‘TLM_TYPES) Descriptor;
                      RequestData#(‘TLM_TYPES) Data;
                     } TLMRequest#(‘TLM_TYPE_PRMS) deriving(Eq, Bits, Bounded);
```

RequestDescriptor The table below describes the components of a `RequestDescriptor` and the valid values for each of its members.

RequestDescriptor		
Member Name	DataType	Valid Values
command	TLMCommand	READ, WRITE, UNKNOWN
mode	TLMMode	REGULAR, DEBUG, CONTROL
addr	TLMAddr#(‘TLM_TYPES)	Bit#(addr_size)
data	TLMData#(‘TLM_TYPES)	Bit#(data_size)
burst_length	TLMUint#(‘TLM_TYPES)	UInt#(uint_size)
byte_enable	TLMByteEn#(‘TLM_TYPES)	Bit#(TDiv#(data_size, 8))
burst_mode	TLMBurstMode	INCR, CNST, WRAP, UNKNOWN
burst_size	TLMBurstSize#(‘TLM_TYPES)	Bit#(TLog#(TDiv#(data_size, 8)))
prty	TLMUInt#(‘TLM_TYPES)	UInt#(uint_size)
thread_id	TLMId#(‘TLM_TYPES)	Bit#(id_size)
transaction_id	TLMId#(‘TLM_TYPES)	Bit#(id_size)
export_id	TLMId#(‘TLM_TYPES)	Bit#(id_size)
custom	TLMCustom#(‘TLM_TYPES)	cstm_type

```
typedef struct {TLMCommand                command;
                TLMMode                   mode;
                TLMAddr#(‘TLM_TYPES)      addr;
                TLMData#(‘TLM_TYPES)      data;
                TLMUInt#(‘TLM_TYPES)      burst_length;
                TLMByteEn#(‘TLM_TYPES)    byte_enable;
                TLMBurstMode              burst_mode;
                TLMBurstSize#(‘TLM_TYPES) burst_size;
                TLMUInt#(‘TLM_TYPES)      prty;
                TLMId#(‘TLM_TYPES)        thread_id;
                TLMId#(‘TLM_TYPES)        transaction_id;
                TLMId#(‘TLM_TYPES)        export_id;
                TLMCustom#(‘TLM_TYPES)    custom;
                } RequestDescriptor#(‘TLM_TYPE_PRMS) deriving (Eq, Bits, Bounded);
```

RequestData The table below describes the components of a `RequestData` and the valid values for its members.

RequestData		
Member Name	DataType	Valid Values
data	TLMData#(‘TLM_TYPES)	Bit#(data_size)
transaction_id	TLMId#(‘TLM_TYPES)	Bit#(id_size)
custom	TLMCustom#(‘TLM_TYPES)	cstm_type

```
typedef struct {TLMData#(‘TLM_TYPES)   data;
                TLMId#(‘TLM_TYPES)     transaction_id;
                TLMCustom#(‘TLM_TYPES) custom;
               } RequestData#(‘TLM_TYPE_PRMS) deriving (Eq, Bits, Bounded);
```

TLMResponse The table below describes the components of a `TLMResponse` and the valid values for its members.

TLMResponse		
Member Name	DataType	Valid Values
command	TLMCommand	READ, WRITE, UNKNOWN
data	TLMData#(‘TLM_TYPES)	Bit#(data_size)
status	TLMStatus	SUCCESS, ERROR, NO_RESPONSE
prty	TLMUInt#(‘TLM_TYPES)	UInt#(uint_size)
thread_id	TLMId#(‘TLM_TYPES)	Bit#(id_size)
transaction_id	TLMId#(‘TLM_TYPES)	Bit#(id_size)
export_id	TLMId#(‘TLM_TYPES)	Bit#(id_size)
custom	TLMCustom#(‘TLM_TYPES)	cstm_type

```
typedef struct {TLMCommand             command;
                TLMData#(‘TLM_TYPES)   data;
                TLMStatus              status;
                TLMUInt#(‘TLM_TYPES)   prty;
                TLMId#(‘TLM_TYPES)     thread_id;
                TLMId#(‘TLM_TYPES)     transaction_id;
                TLMId#(‘TLM_TYPES)     export_id;
                TLMCustom#(‘TLM_TYPES) custom;
                } TLMResponse#(‘TLM_TYPE_PRMS) deriving (Eq, Bits, Bounded);
```

Configurable Parameters

In the above BSV code definitions the compiler macros ‘TLM_TYPE_PRMS and ‘TLM_TYPES are used in the typedef statements. A ’define statement is a preprocessor construct used to place prepackaged text values into a file, as described in Section 2.7.1. In this case, the macros contain parameters to be used in the data definitions. Placing the parameters in a separate file allows them to be easily modified for different protocol requirements. For convenience, we have predefined a few useful definitions for use in the TLM package.

The TLM_TYPE_PRMS macro contains type definition parameters which are used in the interface definitions or as arguments to TLM types and interfaces.

The TLM_TYPES macro is used when providing the interface or using the data type. TLM_TYPES is still polymorphic.

The macro TLM_STD_TYPES provides specific values for the polymorphic values defined above. The values defined in TLM_STD_TYPES are common values. The user can change any of the values or define other corresponding macros (with different values) as appropriate for a given design.

The macros are found in the file TLM.defines. A sample of the contents of the file are displayed below.

```
‘define TLM_TYPE_PRMS numeric type id_size, numeric type addr_size, \
                      numeric type data_size, numeric type uint_size, type cstm_type
‘define TLM_TYPES id_size, addr_size, data_size, uint_size, cstm_type
‘define TLM_STD_TYPES  4, 32, 32, 10, Bit#(0)
```

Interfaces

The TLM interfaces define how TLM blocks interconnect and communicate. The TLM package includes two basic interfaces: The TLMSendIFC interface and the TLMRecvIFC interface. These interfaces use basic Get and Put subinterfaces as the requests and responses, as described in Section C.6.1. The TLMSendIFC interface generates (Get) requests and receives (Put) responses. The TLMRecvIFC interface receives (Put) requests and generates (Get) responses. Additional TLM interfaces are built up from these basic blocks.

TLMSendIFC The `TLMSendIFC` interface transmits the requests and receives the responses.

TLMSendIFC Interface		
Name	Type	Description
`tx`	`Get#(TLMRequest#(‘TLM_TYPES))`	Transmits a request through the `Get` interface
`rx`	`Put#(TLMResponse#(‘TLM_TYPES))`	Receives a response through the `Put` interface

```
interface TLMSendIFC#(‘TLM_TYPE_PRMS);
   interface Get#(TLMRequest#(‘TLM_TYPES))  tx;
   interface Put#(TLMResponse#(‘TLM_TYPES)) rx;
endinterface
```

TLMRecvIFC The `TLMRecvIFC` interface receives the requests and transmits the responses.

TLMRecvIFC Interface		
Name	Type	Description
`tx`	`Get#(TLMResponse#(‘TLM_TYPES))`	Transmits the response through the `Get` interface
`rx`	`Put#(TLMRequest#(‘TLM_TYPES))`	Receives the request through the `Put` interface

```
interface TLMRecvIFC#(‘TLM_TYPE_PRMS);
   interface Get#(TLMResponse#(‘TLM_TYPES)) tx;
   interface Put#(TLMRequest#(‘TLM_TYPES))  rx;
endinterface
```

As illustrated in Figure 30, a `TLMSendIFC` is connectable to a `TLMRecvIFC`, just as a `Get` is connectable to a `Put`. A transmitted request (`tx`) from a `TLMSendIFC` is received (`rx`) by the `TLMRecvIFC` and visa versa.

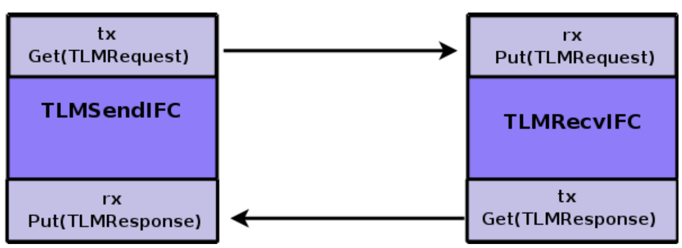

Figure 30: Connecting TLM Send And Receive Interfaces

```
instance Connectable#(TLMSendIFC#(‘TLM_TYPES), TLMRecvIFC#(‘TLM_TYPES));
```

A module with a `TLMSendIFC` interface creates a stream of requests. A module with a `TLMRecvIFC` interface receives the requests and transmits responses. Some bus protocols have separate channels for read and write operations. In these cases it is useful to have interfaces which bundle together two sends or two receives. The `TLMReadWriteSendIFC` interface includes two send interfaces while the `TLMReadWriteRecvIFC` interface bundles two receives.

TLMReadWriteSendIFC The `TLMReadWriteSendIFC` interface is composed of two `TLMSendIFC` subinterfaces, one for a read channel and one for a write channel.

```
interface TLMReadWriteSendIFC#(`TLM_TYPE_PRMS);
   interface TLMSendIFC#(`TLM_TYPES) read;
   interface TLMSendIFC#(`TLM_TYPES) write;
endinterface
```

TLMReadWriteRecvIFC The `TLMReadWriteRecvIFC` interface is composed of two `TLMRecvIFC` subinterfaces, one for a read channnel and one for a write channel.

```
interface TLMReadWriteRecvIFC#(`TLM_TYPE_PRMS);
   interface TLMRecvIFC#(`TLM_TYPES) read;
   interface TLMRecvIFC#(`TLM_TYPES) write;
endinterface
```

As illustreated in Figure 31, the `TLMReadWriteSendIFC` and `TLMReadWriteRecvIFC` interfaces are connectable as well.

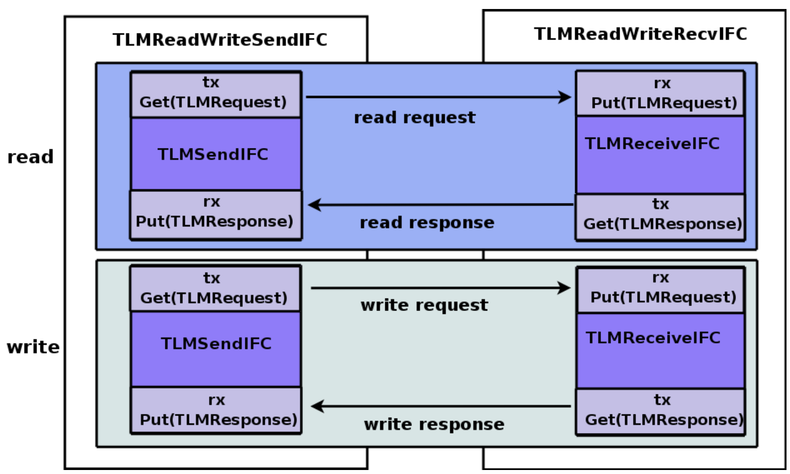

Figure 31: TLM Read/Write Interfaces

```
instance Connectable#(TLMReadWriteSendIFC#(`TLM_TYPES), TLMReadWriteRecvIFC#(`TLM_TYPES));
```

TLMTransformIFC The `TLMTransformIFC` provides a single `TLMRecvIFC` interface and a single `TLMSendIFC` interface. This interface is useful in modules which convert one stream of TLM operations into another. It is the interface provided by `mkTLMReducer` module for instance.

```
interface TLMTransformIFC#(`TLM_TYPE_PRMS);
   interface TLMRecvIFC#(`TLM_TYPES) in;
   interface TLMSendIFC#(`TLM_TYPES) out;
endinterface
```

Modules

The TLM package includes modules for creating and modifying TLM objects: `mkTLMRandomizer`, `mkTLMSource`, and `mkTLMReducer`. Two TLM RAM modules are also provided: `mkTLMRam` which provides a single read/write port and `mkTLMReadWriteRam` which provides two ports, a separate one for reads and a separate one for writes.

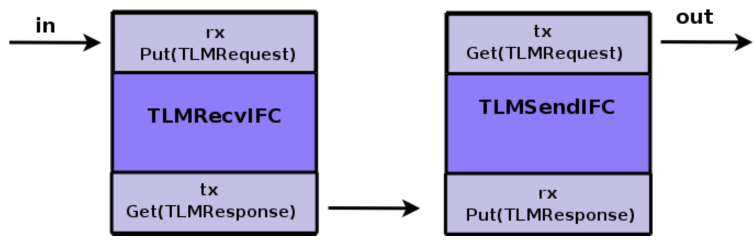

Figure 32: TLMTransformIFC Interface

mkTLMRandomizer	Creates a stream of random TLM operations. The argument `m_command` is a `Maybe` type which determines if the `TLMRequests` will be reads, writes, or both. A value of `Valid READ` will generate only reads, a value of `Valid WRITE` will generate only writes, and an `Invalid` value will generate both reads and writes. The `Randomize` interface is defined in the `Randomizable` package.	

```
module mkTLMRandomizer#(Maybe#(TLMCommand) m_command)
                       (Randomize#(TLMRequest#(`TLM_TYPES)))
   provisos(Bits#(RequestDescriptor#(`TLM_TYPES), s0),
            Bounded#(RequestDescriptor#(`TLM_TYPES)),
            Bits#(RequestData#(`TLM_TYPES), s1),
            Bounded#(RequestData#(`TLM_TYPES)));
```

mkTLMSource	Creates a wrapper around the `mkTLMRandomize` module. The provided interface is now a `TLMSendIFC` interface which both sends `TLMRequests` and receives `TLMResponses`. The argument `m_command` has the same meaning as in `mkTLMRandomizer`. The `verbose` argument controls whether or not `$display` outputs are provide when sending and receiving TLM objects.

```
module mkTLMSource#(Maybe#(TLMCommand) m_command, Bool verbose)
                   (TLMSendIFC#(`TLM_STD_TYPES));
```

mkTLMReducer	Converts a stream of (arbitrary) TLM operations into a stream with only single reads and single writes.

```
module mkTLMReducer (TLMTransformIFC#(`TLM_TYPES))
   provisos(Bits#(TLMRequest#(`TLM_TYPES), s0),
            Bits#(TLMResponse#(`TLM_TYPES), s1),
            Bits#(RequestDescriptor#(`TLM_TYPES), s2));
```


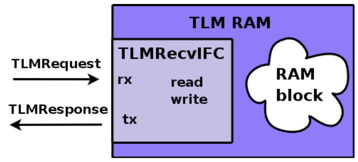

Figure 33: TLMRAM

`mkTLMRam`	Creates a TLM RAM with a single port for read and write operations. Provides the `TLMRecvIFC` interface. The `verbose` argument controls whether or not `$display` output is provided when performing a memory operation. The `id` argument provides an identifier for the instantiation which is used in the `$display` output if the `verbose` flag is asserted.

	`module mkTLMRam#(parameter Bit#(4) id, Bool verbose)`
	`(TLMRecvIFC#(‘TLM_TYPES))`
	`provisos(Bits#(TLMRequest#(‘TLM_TYPES), s0),`
	`Bits#(TLMResponse#(‘TLM_TYPES), s1));`

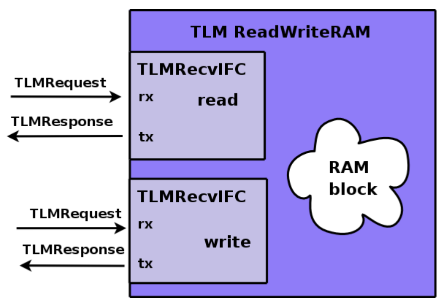

Figure 34: TLMReadWriteRAM

`mkTLMReadWriteRam`	Creates a RAM with separate ports for read and write operations. Provides the `TLMReadWriteRecvIFC` interface. The `verbose` argument controls whether or not `$display` output is provided when performing a memory operation. The `id` argument provides an identifier for the instantiation which is used in the `$display` output if the `verbose` flag is asserted.

	`module mkTLMReadWriteRam#(parameter Bit#(4) id, Bool verbose)`
	`(TLMReadWriteRecvIFC#(‘TLM_TYPES))`
	`provisos(Bits#(TLMRequest#(‘TLM_TYPES), s0),`
	`Bits#(TLMResponse#(‘TLM_TYPES), s1));`

The `mkTLMCBusAdapter` module creates an adapter which allows the `CBus` (Section C.9.2) to be accessed via a TLM interface.

`mkTLMCBusAdapter`	Takes a `TLMCBus` interface as an argument. Provides the `TLMRecvIFC` interface.	

```
module mkTLMCBusAdapter#(TLMCBus#(`TLM_TYPES, caddr_size) cfg)
                        (TLMRecvIFC#(`TLM_TYPES))
   provisos(Bits#(TLMRequest#(`TLM_TYPES),  s0),
    Bits#(TLMResponse#(`TLM_TYPES), s1),
    Add#(ignore, caddr_size, addr_size));
```

`mkTLMCBusAdapterToReadWrite`	Takes a `TLMCBus` interface as an argument. Provides the `TLMReadWriteRecvIFC` interface. This configuration provides separate ports for read and write operations.

```
module mkTLMCBusAdapterToReadWrite#
                  (TLMCBus#(`TLM_TYPES, caddr_size) cfg)
                  (TLMReadWriteRecvIFC#(`TLM_TYPES))
   provisos(Bits#(TLMRequest#(`TLM_TYPES),  s0),
    Bits#(TLMResponse#(`TLM_TYPES), s1),
    Add#(ignore, caddr_size, addr_size));
```

Functions

`createBasicRequestDescriptor`	Returns a generic TLM request with default values.

```
function RequestDescriptor#(`TLM_TYPES)
         createBasicRequestDescriptor()
   provisos(Bits#(RequestDescriptor#(`TLM_TYPES), s0));
```

`createBasicTLMResponse`	Returns a generic TLM response with default values.

```
function TLMResponse#(`TLM_TYPES) createBasicTLMResponse()
   provisos(Bits#(TLMResponse#(`TLM_TYPES), s0));
```

C.10.2 AXI

Description

The AXI library includes interface, transactor, module and function definitions to implement the Advanced eXtensible Interface (AXI) protocol with Bluespec SystemVerilog. The BSV AXI library groups the AXI data and protocols into reusable, parameterized interfaces, which interact with TLM interfaces. An AXI bus is implemented using AXI transactors to connect TLM interfaces on one side with AXI interfaces on the other side.

The AXI library supports the following AXI Bus protocol features:

- Basic and Burst Transfers
- Aligned and Unaligned Transfers

The AXI library does not support the following AXI Bus protocol features:

- Exclusive/Locked Access
- Low Power Interface
- Cache Transaction Attributes

The basic structure of an AXI write bus is show in figure 35. The structure of a read bus is similar. (Note that the nature of the AXI protocol is such that the read and write buses operate totally independently of each other).

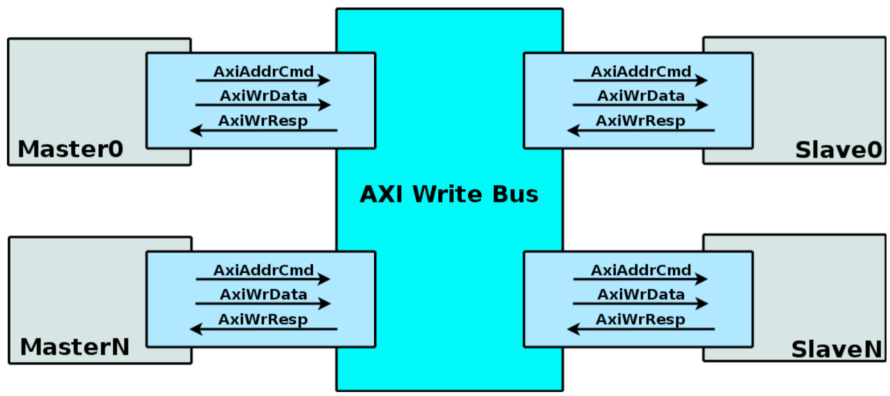

Figure 35: AXI Write Bus Example

The corresponding BSV AXI implementation is shown in figure 36. TLM Write requests are received via the `TLMRecvIFC` interfaces of the master transactors. The request is then transmitted via the `AxiWrMaster` interface out onto the AXI bus and on to the appropriate slave transactor. The slave transactor receives the request via the `AxiWrSlave` interface, translates the request back into a stream of TLM objects, and then transmits those objects via the `TLMSendIFC` interface. The TLM response from the write operation follows the same path in reverse.

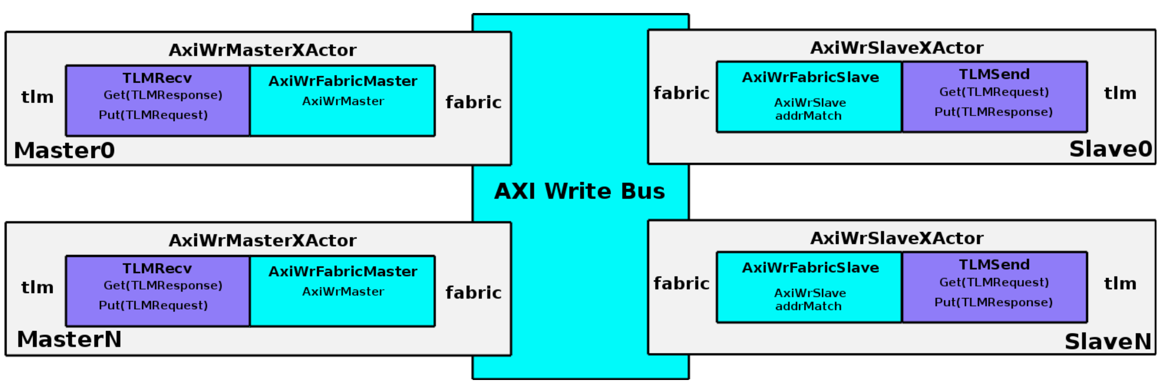

Figure 36: BSV AXI Write Bus Implementation Using TLM Transactors

Packages

The transactors, interfaces, data structures, modules, and functions for implementing the AXI bus are defined in the AXI package.

To include a package in your design, use the import syntax.

```
import Axi :: * ;
```

Data Structures

Inside the transactor modules, the AXI data is organized into the following data structures: the address data is defined by AxiAddrCmd, the read response is defined by AxiRdResp, the write data is defined by AxiWrData and the write response is defined by AxiWrResp.

AxiAddrCmd The AXI Address Bus is defined by a structure, AxiAddrCmd, the components of which are described in the following table.

AxiAddrCmd		
Member Name	DataType	Valid Values
id	AxiId#(‘TLM_TYPES)	Bit#(id_size)
len	AxiLen	Bit#(4)
size	AxiSize	Bit#(3)
burst	AxiBurst	FIXED, INCR, WRAP
lock	AxiLock	NORMAL, EXCLUSIVE, LOCKED
cache	AxiCache	Bit#(4)
prot	AxiProt	Bit#(3)
addr	AxiAddr#(‘TLM_TYPES)	Bit#(addr_size)

```
typedef struct {
                AxiId#(‘TLM_TYPES)   id;
                AxiLen               len;
                AxiSize              size;
                AxiBurst             burst;
                AxiLock              lock;
                AxiCache             cache;
                AxiProt              prot;
                AxiAddr#(‘TLM_TYPES) addr;
                } AxiAddrCmd#(‘TLM_TYPE_PRMS) deriving(Bits,Eq);
```

AxiRdResp The AXI Read Bus is defined by the AxiRdResp structure, the components of which are described in the following table.

AxiRdResp		
Member Name	DataType	Valid Values
id	AxiId#(‘TLM_TYPES)	Bit#(id_size)
data	AxiData#(‘TLM_TYPES)	Bit#(data_size)
resp	AxiResp	OKAY, EXOKAY, SLVERR, DECERR
last	Bool	True, False

```
typedef struct {
                AxiId#(`TLM_TYPES)   id;
                AxiData#(`TLM_TYPES) data;
                AxiResp              resp;
                Bool                 last;
                } AxiRdResp#(`TLM_TYPE_PRMS) deriving(Bits,Eq);
```

The AXI Write Bus is defined by two structures, `AxiWrData` and `AxiWrResp`.

AxiWrData The components of `AxiWrData` are described in the following table.

AxiWrData		
Member Name	DataType	Valid Values
`id`	`AxiId#(`TLM_TYPES)`	`Bit#(id_size)`
`data`	`AxiData#(`TLM_TYPES)`	`Bit#(data_size)`
`strb`	`AxiByteEn#(`TLM_TYPES)`	`Bit#(TDiv#(data_size, 8))`
`last`	Bool	True, False

```
typedef struct {
                AxiId#(`TLM_TYPES)     id;
                AxiData#(`TLM_TYPES)   data;
                AxiByteEn#(`TLM_TYPES) strb;
                Bool                   last;
                } AxiWrData#(`TLM_TYPE_PRMS) deriving(Bits,Eq);
```

AxiWrResp The components of `AxiWrResp` are described in the following table.

AxiWrResp		
Member Name	DataType	Valid Values
`id`	`AxiId#(`TLM_TYPES)`	`Bit#(id_size)`
`resp`	`AxiResp`	OKAY, EXOKAY, SLVERR, DECERR

```
typedef struct {
                AxiId#(`TLM_TYPES)   id;
                AxiResp              resp;
                } AxiWrResp#(`TLM_TYPE_PRMS) deriving(Bits,Eq);
```

Bus Interfaces

This section describes the AXI bus master and slave interfaces used by the AXI transactor modules. Since the AXI protocol supports read and write operations on separate buses, two flavors of each interface exist, one for reads and one for writes.

AxiRdMaster The `AxiRdMaster` interface issues AXI read requests and receives AXI read responses.

```
interface AxiRdMaster#(`TLM_TYPE_PRMS);
   // Address Outputs
   method AxiId#(`TLM_TYPES)   arID;
```



```
   method AxiAddr#(`TLM_TYPES) arADDR;
   method AxiLen              arLEN;
   method AxiSize             arSIZE;
   method AxiBurst            arBURST;
   method AxiLock             arLOCK;
   method AxiCache            arCACHE;
   method AxiProt             arPROT;
   method Bool                arVALID;

   // Address Inputs
   method Action arREADY(Bool value);

   // Response Outputs
   method Bool                  rREADY;

   // Response Inputs
   method Action rID   (AxiId#(`TLM_TYPES)   value);
   method Action rDATA (AxiData#(`TLM_TYPES) value);
   method Action rRESP (AxiResp              value);
   method Action rLAST (Bool                 value);
   method Action rVALID(Bool                 value);
endinterface
```

AxiWrMaster The `AxiWrMaster` interface issues AXI write requests and receives AXI write responses.

```
interface AxiWrMaster#(`TLM_TYPE_PRMS);
   // Address Outputs
   method AxiId#(`TLM_TYPES)   awID;
   method AxiAddr#(`TLM_TYPES) awADDR;
   method AxiLen              awLEN;
   method AxiSize             awSIZE;
   method AxiBurst            awBURST;
   method AxiLock             awLOCK;
   method AxiCache            awCACHE;
   method AxiProt             awPROT;
   method Bool                awVALID;

   // Address Inputs
   method Action awREADY(Bool value);

   // Data Outputs
   method AxiId#(`TLM_TYPES)     wID;
   method AxiData#(`TLM_TYPES)   wDATA;
   method AxiByteEn#(`TLM_TYPES) wSTRB;
   method Bool                   wLAST;
   method Bool                   wVALID;

   // Data Inputs
   method Action wREADY(Bool value);

   // Response Outputs
   method Bool                   bREADY;
```

```
   // Response Inputs
   method Action bID   (AxiId#(`TLM_TYPES) value);
   method Action bRESP (AxiResp            value);
   method Action bVALID(Bool               value);
endinterface
```

AxiRdSlave The `AxiRdSlave` interface receives AXI read requests and returns AXI read responses.

```
interface AxiRdSlave#(`TLM_TYPE_PRMS);
   // Address Inputs
   method Action arID   (AxiId#(`TLM_TYPES)   value);
   method Action arADDR (AxiAddr#(`TLM_TYPES) value);
   method Action arLEN  (AxiLen               value);
   method Action arSIZE (AxiSize              value);
   method Action arBURST(AxiBurst             value);
   method Action arLOCK (AxiLock              value);
   method Action arCACHE(AxiCache             value);
   method Action arPROT (AxiProt              value);
   method Action arVALID(Bool                 value);

   // Address Outputs
   method Bool arREADY;

   // Response Inputs
   method Action rREADY(Bool value);

   // Response Outputs
   method AxiId#(`TLM_TYPES)   rID;
   method AxiData#(`TLM_TYPES) rDATA;
   method AxiResp              rRESP;
   method Bool                 rLAST;
   method Bool                 rVALID;
endinterface
```

AxiWrSlave The `AxiWrSlave` interface receives AXI write requests and returns AXI write responses.

```
interface AxiWrSlave#(`TLM_TYPE_PRMS);
   // Address Inputs
   method Action awID   (AxiId#(`TLM_TYPES)   value);
   method Action awADDR (AxiAddr#(`TLM_TYPES) value);
   method Action awLEN  (AxiLen               value);
   method Action awSIZE (AxiSize              value);
   method Action awBURST(AxiBurst             value);
   method Action awLOCK (AxiLock              value);
   method Action awCACHE(AxiCache             value);
   method Action awPROT (AxiProt              value);
   method Action awVALID(Bool                 value);

   // Address Outputs
```



```
   method Bool awREADY;

   // Data Inputs
   method Action wID   (AxiId#(`TLM_TYPES)     value);
   method Action wDATA (AxiData#(`TLM_TYPES)   value);
   method Action wSTRB (AxiByteEn#(`TLM_TYPES) value);
   method Action wLAST (Bool                   value);
   method Action wVALID(Bool                   value);

   // Data Ouptuts
   method Bool wREADY;

   // Response Inputs
   method Action bREADY(Bool value);

   // Response Outputs
   method AxiId#(`TLM_TYPES) bID;
   method AxiResp            bRESP;
   method Bool               bVALID;
endinterface
```

The `AxiRdMaster` and `AxiRdSlave` interfaces as well as the `AxiWrMaster` and `AxiWrSlave` interfaces are connectable.

```
instance Connectable#(AxiRdMaster#(`TLM_TYPES), AxiRdSlave#(`TLM_TYPES));

instance Connectable#(AxiWrMaster#(`TLM_TYPES), AxiWrSlave#(`TLM_TYPES));
```

Fabric Interfaces

When used in the context of a bus or switch, AXI transactor modules must communicate with address decoding logic. As with the BSV implementation of the AHB bus, bus fabric interfaces are provided to support this communication. Unlike the AHB protocol however, with the AXI bus protocol no explicit communication between the arbiter and the master transactor modules is required. Thus the `AxiRdFabricMaster` and `AxiWrFabricMaster` interfaces are simply wrappers around the bus interfaces themselves.

```
interface AxiRdFabricMaster#(`TLM_TYPE_PRMS);
   interface AxiRdMaster#(`TLM_TYPES) bus;
endinterface

interface AxiWrFabricMaster#(`TLM_TYPE_PRMS);
   interface AxiWrMaster#(`TLM_TYPES) bus;
endinterface
```

The `AxiRdFabricSlave` and `AxiWrFabricSlave` interfaces each provide an `addrMatch` method which given an AXI address returns an Boolean value indicating whether the given address maps to the associated slave. By polling this method for each slave on the bus, the decoding logic can determine the appropriate destination for each bus transaction.

```
interface AxiRdFabricSlave#(`TLM_TYPE_PRMS);
   interface AxiRdSlave#(`TLM_TYPES) bus;
   method Bool addrMatch(AxiAddr#(`TLM_TYPES) value);
endinterface
```

```
interface AxiWrFabricSlave#(`TLM_TYPE_PRMS);
   interface AxiWrSlave#(`TLM_TYPES) bus;
   method Bool addrMatch(AxiAddr#(`TLM_TYPES) value);
endinterface
```

Transactor Interfaces

Each AXI transactor module provides AXI and TLM interfaces to implement a translation between a stream of TLM operations and the AXI bus protocol. Each transactor has two subinterfaces: a subinterface for the connection with the AXI bus and a subinterface to send and receive TLM objects. The AXI library package includes two master transactor interfaces and two slave transactor interfaces; The `AXIRdMasterXActor` and `AXIWrMasterXActor` interfaces for masters and the `AXIRdSlaveXActor` and `AXIWrSlaveXActor` interfaces for slaves. Since the AXI protocol supports read and write transaction on separate buses, two transactor implementations are required for masters and two implementations for slaves. The AXI subinterface definitions can be found in section C.10.2. The TLM interfaces are described in Section C.10.1.

AxiRdMasterXActorIFC The `AxiRdMasterXActorIFC` has two subinterfaces: an `AxiRdFabricMaster` subinterface and a `TLMRecvIFC` subinterface. The associated transactor converts TLM read requests into the AXI protocol, and converts the AXI response back into TLM.

```
interface AxiRdMasterXActorIFC#(`TLM_TYPE_PRMS);
   interface TLMRecvIFC#(`TLM_TYPES)        tlm;
   interface AxiRdFabricMaster#(`TLM_TYPES) fabric;
endinterface
```

AxiWrMasterXActorIFC The `AxiWrMasterXActorIFC` has two subinterfaces: an `AxiWrFabricMaster` subinterface and a `TLMRecvIFC` subinterface. The associated transactor converts TLM write requests into the AXI protocol, and converts the AXI response back into TLM.

```
interface AxiWrMasterXActorIFC#(`TLM_TYPE_PRMS);
   interface TLMRecvIFC#(`TLM_TYPES)        tlm;
   interface AxiWrFabricMaster#(`TLM_TYPES) fabric;
endinterface
```

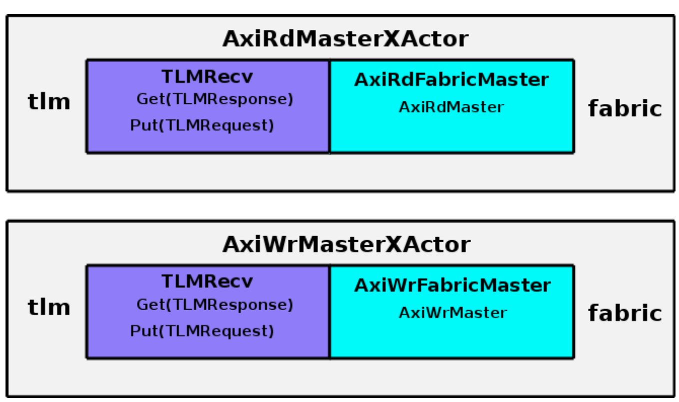

Figure 37: AXIMasterXActor Interfaces (Read and Write Versions)

AxiRdSlaveXActorIFC The `AxiRdSlaveXActorIFC` has two subinterfaces: an `AxiRdFabricSlave` subinterface and a `TLMSendIFC` subinterface. The associated transactor converts an AXI read request into TLM and the TLM response back into the AXI protocol.

```
interface AxiRdSlaveXActorIFC#(`TLM_TYPE_PRMS);
   interface TLMSendIFC#(`TLM_TYPES)       tlm;
   interface AxiRdFabricSlave#(`TLM_TYPES) fabric;
endinterface
```

AxiWrSlaveXActorIFC The `AxiWrSlaveXActorIFC` has two subinterfaces: an `AxiWrFabricSlave` subinterface and a `TLMSendIFC` subinterface. The associated transactor converts an AXI write request into TLM and the TLM response back into the AXI protocol.

```
interface AxiWrSlaveXActorIFC#(`TLM_TYPE_PRMS);
   interface TLMSendIFC#(`TLM_TYPES)       tlm;
   interface AxiWrFabricSlave#(`TLM_TYPES) fabric;
endinterface
```

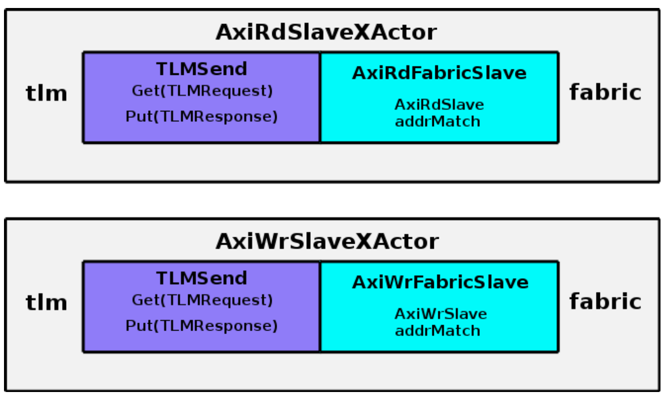

Figure 38: AXISlaveXActor Interfaces (Read and Write Versions)

Modules

The following constructors are used to create AXI transactor modules. Versions with associated synthesis boundaries are also available. These versions are called `mkAxiRdMasterStd`, `mkAxiWrMasterStd`, `mkAxiRdSlaveStd`, and `mkAxiWrSlaveStd`. The specific TLM parameter values for these synthesized versions are as specified by the preprocessor macro `TLM_STD_TYPES` (see section C.10.1).

`mkAxiRdMaster`	Creates an AXI master read transactor module. Provides an `AxiRdMasterXActorIFC` interface.
	`module mkAxiRdMaster (AxiRdMasterXActorIFC#(`TLM_TYPES))` ` provisos(Bits#(TLMRequest#(`TLM_TYPES), s0),` `            Bits#(TLMResponse#(`TLM_TYPES), s1));`

`mkAxiWrMaster`	Creates an AXI master write transactor module. Provides an `AxiWrMasterXActorIFC` interface.	

	`module mkAxiWrMaster (AxiWrMasterXActorIFC#(‘TLM_TYPES))`
	`   provisos(Bits#(TLMRequest#(‘TLM_TYPES), s0),`
	`            Bits#(TLMResponse#(‘TLM_TYPES), s1));`	

`mkAxiRdSlave`	Creates an AXI slave read transactor module. Provides an `AxiRdSlaveXActorIFC` interface.

	`module mkAxiRdSlave#(function Bool`
	`                       addr_match(AxiAddr#(‘TLM_TYPES) addr))`
	`                       (AxiRdSlaveXActorIFC#(‘TLM_TYPES))`
	`   provisos(Bits#(TLMRequest#(‘TLM_TYPES), s0),`
	`            Bits#(TLMResponse#(‘TLM_TYPES), s1),`
	`            Bits#(RequestDescriptor#(‘TLM_TYPES), s2));`

`mkAxiWrSlave`	Creates an AXI slave write transactor module. Provides an `AxiWrSlaveXActorIFC` interface.

	`module mkAxiWrSlave#(function Bool`
	`                       addr_match(AxiAddr#(‘TLM_TYPES) addr))`
	`                       (AxiWrSlaveXActorIFC#(‘TLM_TYPES))`
	`   provisos(Bits#(TLMRequest#(‘TLM_TYPES), s0),`
	`            Bits#(TLMResponse#(‘TLM_TYPES), s1),`
	`            Bits#(RequestDescriptor#(‘TLM_TYPES), s2));`

The following two module constructors are each used to create an AXI bus fabric. `mkAxiRdBus` is used to create a read bus while `mkAxiWrBus` is used to create a write bus.

`mkAxiRdBus`	Given a vector of `AxiRdFabricMaster` interfaces and a vector of `AxiRdFabricSlave` interfaces, `mkAxiRdBus` creates an AXI read bus.

	`module mkAxiRdBus#(Vector#(master_count,`
	`                             AxiRdFabricMaster#(‘TLM_TYPES)) masters,`
	`                    Vector#(slave_count,`
	`                             AxiRdFabricSlave#(‘TLM_TYPES))slaves) (Empty);`

<table>
<tr><td rowspan="2"><code>mkAxiWrBus</code></td><td>Given a vector of <code>AxiWrFabricMaster</code> interfaces and a vector of <code>AxiWrFabricSlave</code> interfaces, <code>mkAxiWrBus</code> creates an AXI write bus.</td></tr>
<tr><td><pre>module mkAxiWrBus#(Vector#(master_count,
                           AxiWrMaster#(`TLM_TYPES)) masters,
                   Vector#(slave_count,
                           AxiWrSlave#(`TLM_TYPES)) slaves) (Empty);</pre></td></tr>
</table>

The following module is used to add probe signals for each of the AXI bus signals. This facilitates debugging and waveform viewing of the created bus fabric.

<table>
<tr><td rowspan="2"><code>mkAxiMonitor</code></td><td>Adds a probe module for each of the AXI bus signals. The <code>include_pc</code> value indicates whether or not the monitor module should include an instantiation of an AXI protocol checker module (available from ARM). If the protocol checker is not available, the value of <code>include_pc</code> should be set to False.</td></tr>
<tr><td><pre>module mkAxiMonitor#(Bool include_pc,
                     AxiWrMaster#(`TLM_TYPES) master_wr,
                     AxiWrSlave#(`TLM_TYPES) slave_wr,
                     AxiRdMaster#(`TLM_TYPES) master_rd,
                     AxiRdSlave#(`TLM_TYPES) slave_rd)
                     (AxiMonitor#(`TLM_TYPES));</pre></td></tr>
</table>

C.10.3 AHB

Description

The AHB library includes interface, transactor, module and function definitions to implement the AHB protocol with Bluespec SystemVerilog. The BSV AHB library groups the AHB data and protocols into reusable, parameterized interfaces, which interact with TLM interfaces. An AHB bus is implemented using AHB transactors - interfaces which connect TLM interfaces on one side with AHB interfaces on the other side.

The AHB library supports the following AHB Bus protocol features:

- Basic and Burst Transfers
- Locked Transfers

The AHB library does not support the following AHB Bus protocol features:

- Early Burst Termination
- Split Transfers
- Retry Transfers

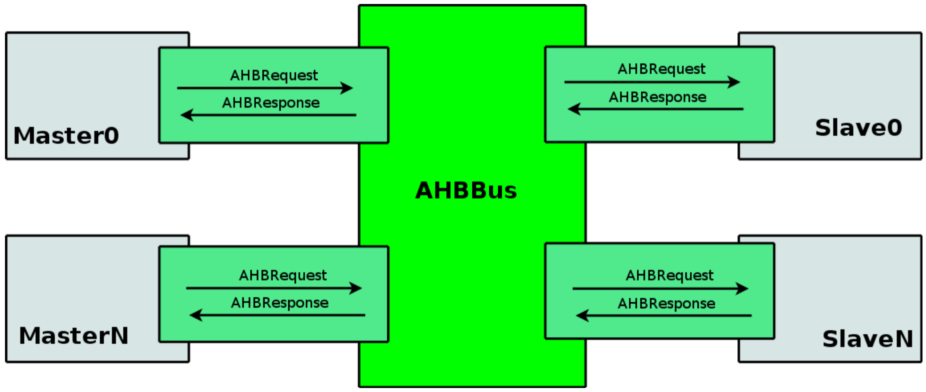

Figure 39: AHB Bus Example

Packages

The transactors, interfaces, data structures, and modules for the AHB bus are defined within the `AHB` package.

To include a package in your design, use the `import` syntax.

```
import AHB :: * ;
```

Data Structures

Inside the transactor modules, the AHB data is organized into the following data structures: the address and control information is defined by `AHBCntrl`, the write data is defined by `AHBData`. These two structures are bundled into an `AHBRequest`. Finally, the response data is defined by `AHBResponse`.

AHBRequest An AHB request is defined by the `AHBRequest` structure as described below.

AHBRequest		
Member	DataType	Valid Values
cntrl	`AHBCntrl#(‘TLM_TYPES)`	see above
data	`AHBData`	`Bit#(data_size)`

```
typedef struct {
               AHBCntrl#(‘TLM_TYPES)    cntrl;
               AHBData#(‘TLM_TYPES)     data;
              } AHBRequest#(‘TLM_TYPE_PRMS) ‘dv;
```

AHBCntrl The control fields in an `AHBRequest` are described by the `AHBCntrl` structure, the components of which are defined in the following table.

AHBCntrl		
Member	DataType	Valid Values
command	`AHBWrite`	READ, WRITE
size	`AHBSize`	BITS8, BITS16, BITS32, BITS64, BITS128, BITS256, BITS512, BITS1024
burst	`AHBBurst`	SINGLE, INCR, WRAP4, INCR4, WRAP8, INCR8, WRAP16, INCR16
transfer	`AHBTransfer`	IDLE, BUSY, NONSEQ, SEQ
prot	`AHBProt`	`Bit#(4)`
addr	`AHBAddr#(‘TLM_TYPES)`	`Bit#(addr_size)`

```
typedef struct {
                AHBWrite            command;
                AHBSize             size;
                AHBBurst            burst;
                AHBTransfer         transfer;
                AHBProt             prot;
                AHBAddr#(‘TLM_TYPES) addr;
               } AHBCntrl#(‘TLM_TYPE_PRMS) ‘dv;
```

AHBResponse An `AHBResponse` consists of a status fields and data (when responding to a read request). The components of the structure are described in the following table.

AHBResponse		
Member	DataType	Valid Values
status	`AHBResp`	OKAY, ERROR, RETRY, SPLIT
data	`AHBData`	`Bit#(data_size)`

```
typedef struct {
                AHBResp             status;
                AHBData#(‘TLM_TYPES) data;
               } AHBResponse#(‘TLM_TYPE_PRMS) ‘dv;
```

Bus Interfaces

The two basic bus interfaces included in the AHB library are the `AHBMaster` interface and the `AHBSlave` interface.

AHBMaster The `AHBMaster` interface issues AHB requests and receives AHB responses.

```
interface AHBMaster#(‘TLM_TYPE_PRMS);
   // Outputs
   method AHBAddr#(‘TLM_TYPES)  hADDR;
   method AHBData#(‘TLM_TYPES)  hWDATA;
   method AHBWrite              hWRITE;
   method AHBTransfer           hTRANS;
   method AHBBurst              hBURST;
   method AHBSize               hSIZE;
   method AHBProt               hPROT;
```

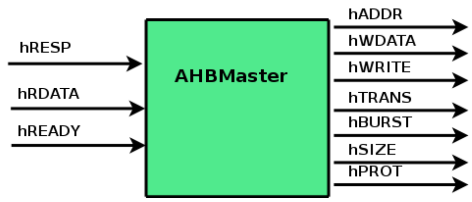

Figure 40: AHB Master Interface

```
   // Inputs
   method Action      hRDATA(AHBData#(`TLM_TYPES) data);
   method Action      hREADY(Bool                 value);
   method Action      hRESP (AHBResp              response);
endinterface
```

AHBSlave The `AHBSlave` interface receives AHB requests and returns AHB responses.

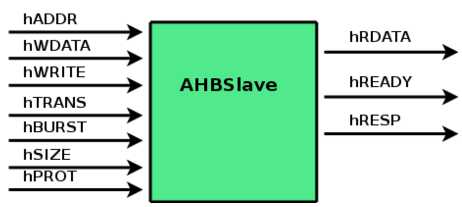

Figure 41: AHB Slave Interface

```
interface AHBSlave#(`TLM_TYPE_PRMS);
    // Inputs
   method Action      hADDR (AHBAddr#(`TLM_TYPES) addr);
   method Action      hWDATA(AHBData#(`TLM_TYPES) data);
   method Action      hWRITE(AHBWrite             value);
   method Action      hTRANS(AHBTransfer          value);
   method Action      hBURST(AHBBurst             value);
   method Action      hSIZE (AHBSize              value);
   method Action      hPROT (AHBProt              value);

   // Outputs
   method AHBData#(`TLM_TYPES) hRDATA;
   method Bool                 hREADY;
   method AHBResp              hRESP;
endinterface
```

The `AHBMaster` and `AHBSlave` interfaces are connectable.

```
instance Connectable#(AHBMaster#(`TLM_TYPES), AHBSlave#(`TLM_TYPES));
```

Fabric Interfaces

When used in the context of a bus or switch, AHB Master and Slave modules must communicate with the arbiter and with address decoding logic. Two additional interfaces are provided to support this communication.

AHBMasterArbiter The `AHBMasterArbiter` interface connects the master module with the bus arbiter. Through this interface, the master can request control of the bus and determine when control has been granted.

```
interface AHBMasterArbiter;
   method Bool        hBUSREQ();
   method Bool        hLOCK  ();
   method Action      hGRANT (Bool value);
endinterface
```

AHBSlaveSelector The `AHBSlaveSelector` interface provides an `addrMatch` method which given an AHB address returns an Boolean value indicating whether the given address maps to the associated slave. By polling this method for each slave on the bus, the decoding logic can determine the appropriate destination for each bus transaction. The `AHBSlaveSelector` interface also provides a `select` method by which the decoding logic can indicate which slave is the selected destination.

```
interface AHBSlaveSelector#(`TLM_TYPE_PRMS);
   method Bool   addrMatch(AHBAddr#(`TLM_TYPES) value);
   method Action select   (Bool                 value);
endinterface
```

AHBFabricMaster The `AHBFabricMaster` interface bundles two sub-interfaces, an `AHBMaster` interface and an `AHBMasterArbiter` interface. It is this interface that is provided as an argument when constructing an AHB bus and as the bus side interface of an AHB master transactor module.

```
interface AHBFabricMaster#(`TLM_TYPE_PRMS);
   interface AHBMaster#(`TLM_TYPES)  bus;
   interface AHBMasterArbiter        arbiter;
endinterface
```

AHBFabricSlave The `AHBFabricSlave` interface bundles two sub-interfaces, an `AHBSlave` interface and an `AHBSlaveSelector` interface. It is this interface that is provided as an argument when constructing an AHB bus and as the bus side interface of an AHB slave transactor module

```
interface AHBFabricSlave#(`TLM_TYPE_PRMS);
   interface AHBSlave#(`TLM_TYPES)         bus;
   interface AHBSlaveSelector#(`TLM_TYPES) selector;
endinterface
```

Transactor Interfaces

An AHB transactor module provides AHB and TLM interfaces to implement a translation between a stream of TLM operations and the AHB bus protocol. Each transactor has two subinterfaces: a subinterface for the connection with the AHB bus and a subinterface to send and receive TLM objects.

The AHB library package includes two transactor interfaces; The `AHBMasterXActor` interface for the master and `AHBSlaveXActor` interface for the slave. The AHB protocol doesn't separate read and write transactions, so there is a single transactor implementation for masters and a single implementation for slaves.

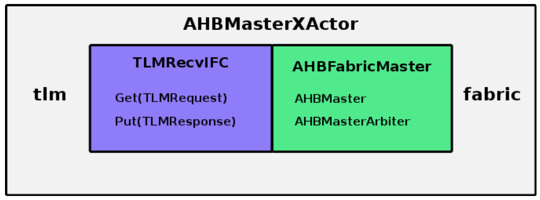

Figure 42: AHBMasterXActor Interface

AHBMasterXActorIFC The `AHBMasterXActorIFC` has two subinterfaces: an `AHBFabricMaster` subinterface and a `TLMRecvIFC` subinterface. The TLM interface is described in Section C.10.1. The transactor converts TLM requests into the AHB protocol, and converts the AHB response back into TLM.

```
interface AHBMasterXActorIFC#(`TLM_TYPE_PRMS)
   interface TLMRecvIFC#(`TLM_TYPES)      tlm;
   interface AHBFabricMaster#(`TLM_TYPES) fabric;
endinterface
```

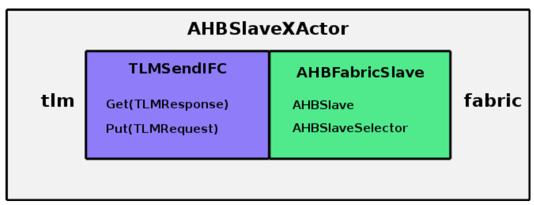

Figure 43: AHBSlaveXActor Interface

AHBSlaveXActorIFC The `AHBSlaveXActorIFC` has two subinterfaces: `AHBFabricSlave` subinterface and a `TLMSendIFC` subinterface. The TLM interface is described in Section C.10.1. The transactor converts an AHB request into TLM and the TLM response back into the AHB protocol.

```
interface AHBSlaveXActorIFC#(`TLM_TYPE_PRMS);
   interface TLMSendIFC#(`TLM_TYPES)     tlm;
   interface AHBFabricSlave#(`TLM_TYPES) fabric;
endinterface
```

Modules

The following constructors are used to create AHB transactor modules. Versions with associated synthesis boundaries are also available. These versions are called `mkAHBMasterStd`, and `mkAHBSlaveStd`. The specific TLM parameter values for these synthesized versions are as specified by the preprocessor macro `TLM_STD_TYPES` (see section C.10.1).

mkAHBMaster	Creates an AHB Master transactor module. Provides a AHBMasterXActorIFC interface.	`module mkAHBMaster (AHBMasterXActorIFC#(‘TLM_TYPES))`
		`   provisos(Bits#(TLMRequest#(‘TLM_TYPES), s0),`
		`            Bits#(TLMResponse#(‘TLM_TYPES), s1),`
		`            Bits#(RequestDescriptor#(‘TLM_TYPES), s2));`

mkAHBSlave	Creates an AHB Slave transactor module. Provides an AHBSlaveXActorIFC interface.	`module mkAHBSlave#(function Bool addr_match(AHBAddr#(‘TLM_TYPES)addr))`
		`                   (AHBSlaveXActorIFC#(‘TLM_TYPES))`
		`   provisos(Bits#(TLMRequest#(‘TLM_TYPES), s0),`
		`            Bits#(TLMResponse#(‘TLM_TYPES), s1),`
		`            Bits#(RequestDescriptor#(‘TLM_TYPES), s2));`

The following module constructor is used to create an AHB bus fabric.

mkAHBBus	Given a vector of AHBFabricMaster interfaces and a vector of AHBFabricSlave interfaces, mkAHBBus creates an AHB bus fabric.	`module mkAHBBus#(Vector#(master_count,`
		`                          AHBFabricMaster#(‘TLM_TYPES)) masters,`
		`                 Vector#(slave_count,`
		`                          AHBFabricSlave#(‘TLM_TYPES))  slaves) (Empty);`

The following module is used to add probe signals for each of the AHB bus signals. This facilitates debugging and waveform viewing of the created bus fabric.

mkAHBMonitor	Adds a probe module for each of the AHB bus signals. The include_pc value indicates whether or not the monitor module should include an instantiation of an AHB protocol checker module (available from ARM). If the protocol checker is not available, the value of include_pc should be set to False.	`module mkAHBMonitor#(Bool include_pc,`
		`                     AHBMaster#(‘TLM_TYPES) master,`
		`                     AHBSlave#(‘TLM_TYPES)  slave)`
		`                    (AHBMonitor#(‘TLM_TYPES));`

Index

Function and Module by Package

Packages provided as BSV source code