B-Lang

Bluespec™ SystemVerilog Language Reference Guide

Revision: 17 February 2024

Copyright © 2000 – January 2020: Bluespec, Inc.
January 2020 onwards: various open-source contributors

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 IEEE. The SystemVerilog standard is owned and maintained by IEEE.

SystemC is a trademark of IEEE. The SystemC standard is owned and maintained by IEEE.

Bluespec is a trademark of Bluespec, Inc.

Contents

Table of Contents 3

1 Introduction 9

1.1 Libraries and the Standard Prelude . . . 9

1.2 BSV and BH . . . 10

1.3 Meta notation for grammar . . . 10

1.4 Overview of Program Structure . . . 10

2 Lexical elements 13

2.1 Whitespace and comments . . . 13

2.2 Identifiers and keywords . . . 14

2.3 Integer literals . . . 14

2.3.1 Type conversion of integer literals . . . 15

2.4 Real literals . . . 16

2.4.1 Type conversion of real literals . . . 16

2.5 String literals . . . 17

2.5.1 Type conversion of string literals . . . 17

2.6 Don't-care values . . . 17

2.7 Compiler directives . . . 18

2.7.1 File inclusion: ‘include and ‘line . . . 18

2.7.2 Macro definition and substitution: ‘define and related directives . . . 18

2.7.3 Conditional compilation: ‘ifdef and related directives . . . 19

3 Packages and the outermost structure of a BSV design 20

3.1 Scopes, name clashes and qualified identifiers . . . 21

3.2 Importing the Standard Prelude and Libraries . . . 22

4 Types 22

4.1 Polymorphism . . . 24

4.2 Provisos (brief intro) . . . 24

4.2.1 The pseudo-function `valueof` (or `valueOf`) . . . 26

4.2.2 The pseudo-function `stringof` (or `stringOf`) . . . 27

4.3 A brief introduction to `deriving` clauses . . . 27

5 Modules and interfaces, and their instances **27**

5.1 Explicit state via module instantiation, not variables . . . 28

5.2 Interface declaration . . . 29

5.2.1 Subinterfaces . . . 31

5.3 Module definition . . . 32

5.4 Module and interface instantiation . . . 34

5.4.1 Short form instantiation . . . 34

5.4.2 Long form instantiation . . . 35

5.5 Interface definition (definition of methods) . . . 36

5.5.1 Shorthands for Action and ActionValue method definitions . . . 37

5.5.2 Definition of subinterfaces . . . 38

5.5.3 Definition of methods and subinterfaces by assignment . . . 39

5.6 Rules in module definitions . . . 40

5.7 Examples . . . 40

5.8 Synthesizing Modules . . . 43

5.8.1 Type Polymorphism . . . 44

5.8.2 Module Interfaces and Arguments . . . 45

5.9 Other module types . . . 45

6 Static and dynamic semantics **46**

6.1 Static semantics . . . 46

6.1.1 Type checking . . . 46

6.1.2 Proviso checking and bit-width constraints . . . 47

6.1.3 Static elaboration . . . 47

6.2 Dynamic semantics . . . 47

6.2.1 Reference semantics . . . 48

6.2.2 Mapping into efficient parallel clocked synchronous hardware . . . 48

6.2.3 How rules are chosen to fire . . . 50

6.2.4 Mapping specific hardware models . . . 51

7 User-defined types (type definitions) **51**

7.1 Type synonyms . . . 52

7.2 Enumerations . . . 53

7.3 Structs and tagged unions . . . 54

8 Type classes (overloading groups) and provisos **57**
8.1 Provisos . . . 58
8.2 Type class declarations . . . 58
8.3 Instance declarations . . . 60
8.4 The `Bits` type class (overloading group) . . . 61
8.5 The `SizeOf` pseudo-function . . . 62
8.6 Deriving `Bits` . . . 62
8.7 Deriving `Eq` . . . 64
8.8 Deriving `Bounded` . . . 64
8.9 Deriving `FShow` . . . 64
8.10 Deriving type class instances for isomorphic types . . . 66
8.11 Monad . . . 67

9 Variable declarations and statements **67**
9.1 Variable and array declaration and initialization . . . 67
9.2 Variable assignment . . . 68
9.3 Implicit declaration and initialization . . . 69
9.4 Register reads and writes . . . 70
9.4.1 Registers and square-bracket notation . . . 71
9.4.2 Registers and range notation . . . 72
9.4.3 Registers and struct member selection . . . 72
9.5 Begin-end statements . . . 73
9.6 Conditional statements . . . 73
9.7 Loop statements . . . 75
9.7.1 While loops . . . 75
9.7.2 For loops . . . 75
9.8 Function definitions . . . 76
9.8.1 Definition of functions by assignment . . . 77
9.8.2 Function types . . . 78
9.8.3 Higher-order functions . . . 78

10 Expressions **80**
10.1 Don't-care expressions . . . 80
10.2 Conditional expressions . . . 81
10.3 Unary and binary operators . . . 81
10.4 Bit concatenation and selection . . . 82
10.5 Begin-end expressions . . . 83
10.6 Actions and action blocks . . . 84

10.7 Actionvalue blocks . . . 85
10.8 Function calls . . . 87
10.9 Method calls . . . 87
10.10 Static type assertions . . . 88
10.11 Struct and union expressions . . . 89
10.11.1 Struct expressions . . . 89
10.11.2 Struct member selection . . . 89
10.11.3 Tagged union expressions . . . 90
10.11.4 Tagged union member selection . . . 90
10.12 Interface expressions . . . 91
10.12.1 Differences between interfaces and structs . . . 92
10.13 Rule expressions . . . 93

11 Pattern matching **94**
11.1 Case statements with pattern matching . . . 96
11.2 Case expressions with pattern matching . . . 97
11.3 Pattern matching in if statements and other contexts . . . 98
11.4 Pattern matching assignment statements . . . 99

12 Finite state machines **100**
12.1 The `Stmt` sublanguage . . . 100
12.2 FSM Interfaces and Methods . . . 102
12.3 FSM Modules . . . 104
12.4 FSM Functions . . . 105
12.5 Creating FSM Server Modules . . . 109
12.6 FSM performance caveat . . . 110

13 Important primitives **110**
13.1 The types `bit` and `Bit` . . . 110
13.1.1 Bit-width compatibility . . . 111
13.2 UInt, Int, int and Integer . . . 111
13.3 String and Char . . . 111
13.4 Tuples . . . 111
13.5 Registers . . . 112
13.6 FIFOs . . . 113
13.7 FIFOFs . . . 114
13.8 System tasks and functions . . . 114
13.8.1 Displaying information . . . 114

13.8.2 $format . . . 115
13.8.3 Opening and closing file operations . . . 116
13.8.4 Writing to a file . . . 117
13.8.5 Formatting output to a string . . . 118
13.8.6 Reading from a file . . . 118
13.8.7 Flushing output . . . 119
13.8.8 Stopping simulation . . . 119
13.8.9 VCD dumping . . . 119
13.8.10 Time functions . . . 120
13.8.11 Real functions . . . 120
13.8.12 Testing command line input . . . 120

14 Guiding the compiler with attributes **120**
14.1 Verilog module generation attributes . . . 121
14.1.1 `synthesize` . . . 121
14.1.2 `noinline` . . . 121
14.2 Interface attributes . . . 122
14.2.1 Renaming attributes . . . 122
14.2.2 Port protocol attributes . . . 123
14.2.3 Interface attributes example . . . 124
14.3 Scheduling attributes . . . 125
14.3.1 `fire_when_enabled` . . . 125
14.3.2 `no_implicit_conditions` . . . 126
14.3.3 `descending_urgency` . . . 127
14.3.4 `execution_order` . . . 129
14.3.5 `mutually_exclusive` . . . 130
14.3.6 `conflict_free` . . . 130
14.3.7 `preempts` . . . 131
14.4 Evaluation behavior attributes . . . 131
14.4.1 `split` and `nosplit` . . . 131
14.5 Input clock and reset attributes . . . 133
14.5.1 Clock and reset prefix naming attributes . . . 133
14.5.2 Gate synthesis attributes . . . 134
14.5.3 Default clock and reset naming attributes . . . 134
14.5.4 Clock family attributes . . . 135
14.6 Module argument and parameter attributes . . . 136
14.6.1 Argument-level clock and reset naming attributes . . . 136

14.6.2 `clocked_by=` . . . 137
14.6.3 `reset_by=` . . . 138
14.6.4 `port=` . . . 138
14.6.5 `parameter=` . . . 138
14.7 Documentation attributes . . . 138
14.7.1 Modules . . . 139
14.7.2 Module instantiation . . . 139
14.7.3 Rules . . . 141

15 Embedding RTL in a BSV design **141**
15.1 Parameter . . . 143
15.2 Method . . . 144
15.3 Port . . . 145
15.4 Input clock . . . 146
15.5 Default clock . . . 147
15.6 Output clock . . . 148
15.7 Input reset . . . 149
15.8 Default reset . . . 150
15.9 Output reset . . . 151
15.10 Ancestor, same family . . . 152
15.11 Schedule . . . 152
15.12 Path . . . 153
15.13 Interface . . . 154
15.14 Inout . . . 155

16 Embedding C in a BSV Design **156**
16.1 Argument Types . . . 156
16.2 Return types . . . 157
16.3 Implicit pack/unpack . . . 158
16.4 Other examples . . . 158

A Keywords **160**

B A Brief History of BH (Bluespec Haskell/Classic) and BSV (Bluespec SystemVerilog) **164**

Index **165**

1 Introduction

BSV (Bluespec SystemVerilog) is aimed at hardware designers who are using or expect to use Verilog [IEE05], VHDL [IEE02], SystemVerilog [IEE13], or SystemC [IEE12] to design ASICs or FPGAs. It is also aimed at people creating *synthesizable* models, transactors, and verification components to run on FPGA emulation platforms. BSV substantially extends the design 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 reference guide 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 Libraries and the Standard Prelude

This reference guide focuses on the BSV *language* (syntax and semantics). However, as with most languages, that is only part of the story; the utility of a language depends equally on the *libraries* that come with it.

A separate document, *Bluespec Compiler (BSC) Libraries Reference Guide* [BL05],[1] describes libraries (BSV packages) that come with *bsc* and are useful across a broad range of hardware designs. A part of those libraries, called the Standard Prelude, is automatically imported into every BSV design because it contains universally useful basic definitions.

The *Libraries Reference Guide* is extensive (over 300 pages) and ever-growing as new libraries are added to the repository. The libraries include many useful data types, many kinds of registers, wires and FIFOs; register files, BRAMs and memory interaces; Vectors; math functions; pseudo-random

[1] https://github.com/B-Lang-org/bsc/tree/main/doc/libraries_ref_guide

number generators; counters; interconnects; facilities for multiple Clock and Reset domains, and more.

Another repository, `bsc-contrib`[2], contains more libraries contributed by various people. These are offered "as-is", i.e., they are not part of *bsc*'s continuous-integration regression tests.

1.2 BSV and BH

BSV (Bluespec SystemVerilog) and BH (Bluespec Haskell, or Bluespec Classic) are actually just two different syntaxes for the same language (exactly the same semantics; just two alternative parsers in the *bsc* compiler).[3] A design project can freely mix packages from the two languages (`.bsv` and `.bs` files, respectively). BSV code is "SystemVerilog-ish" and BH code is "Haskell-ish" in flavor; choosing between them is largely a matter of personal preference. Note however, there are currently (Spring 2022) a few language features that are only available in BSV and not yet in BH; we hope to eliminate this gap over time.

1.3 Meta notation for grammar

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.

1.4 Overview of Program Structure

The sections that follow in this Reference Guide are organized according to the structure of the grammar of BSV. For a newcomer to BSV, that organization may not quickly convey an intuition or mental model of program structure and where each part fits. In this section we provide a top-down overview of BSV program structure so that the reader can locate where each grammatical construct may appear within a complete BSV program.

A complete BSV program is a collection of files, where each file contains one BSV *package*. One package may import another, making the top-level identifiers of the latter visible and usable in the former. Figure 1 shows the structure of a sample program.

Figure 2 shows the kind of top-level constructs one may find in a BSV package. Section 3 describes packages, and Section 3.1 provides more detail about scopes, controlling the import and export of names, and resolving name clashes.

Figure 3 shows what goes into an *interface* declaration (Section 5.2). These are *method* declarations and sub-interface declarations (since interfaces can be nested hierarchically).

Figure 4 shows what goes into a *module* declaration (Section 5.3). These are value, function and (sub)-module declarations and definitions, (sub-)module instantiations, rules, and definitions of methods and sub-interfaces implemented (offered) by the module.

Figure 5 shows what goes into a *rule* (Section 5.6). Each rule has a *rule condition* and a *rule body*

[2] https://github.com/B-Lang-org/bsc-contrib/tree/main/Libraries
[3] See Appendix B for a brief history of how this came to be.

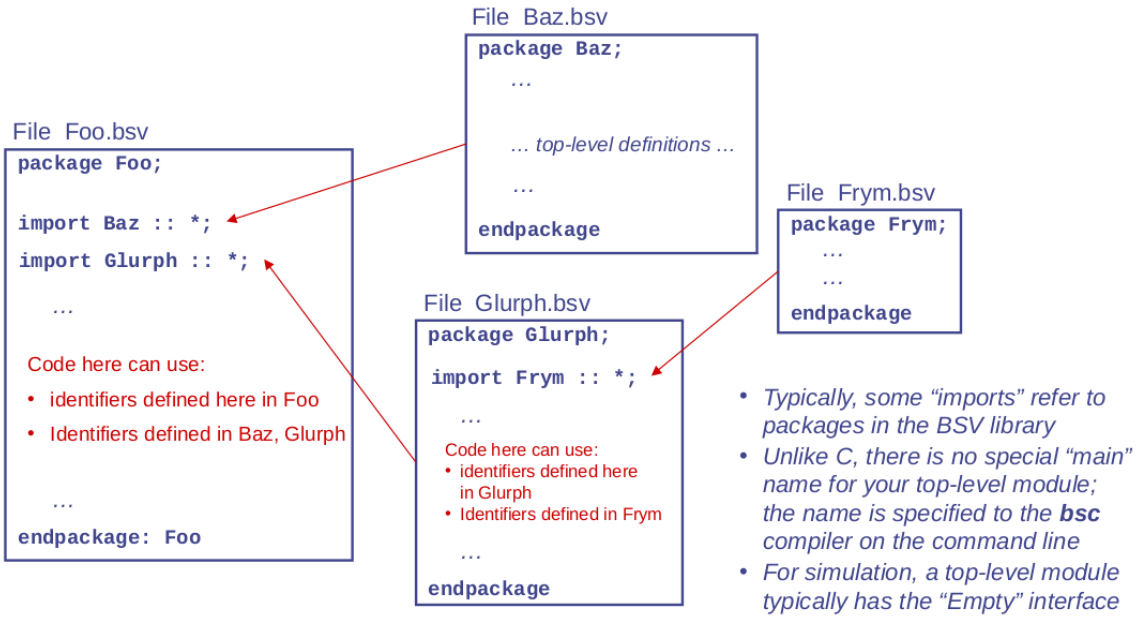

Figure 1: Overall structure of a BSV program.

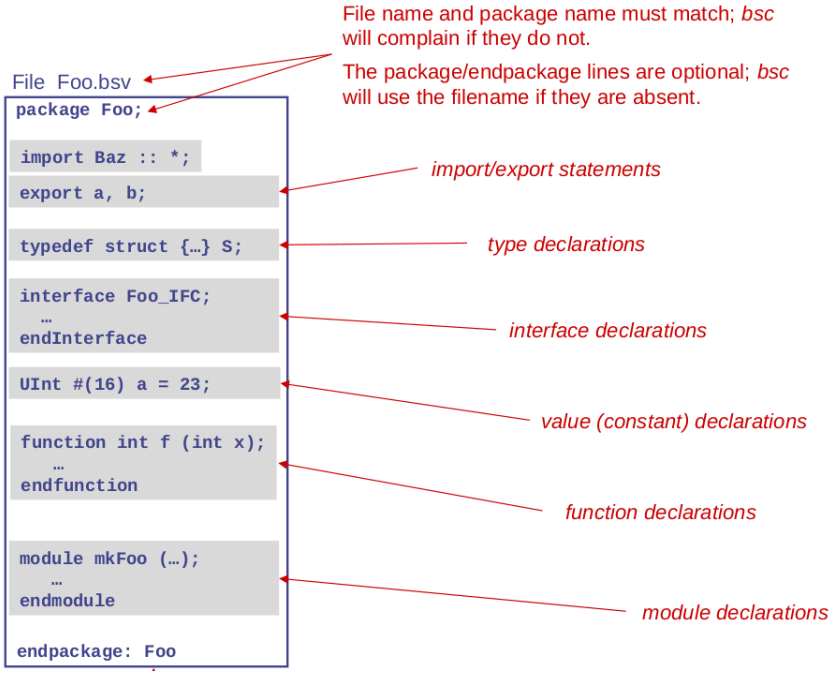

Figure 2: Contents of a BSV package.

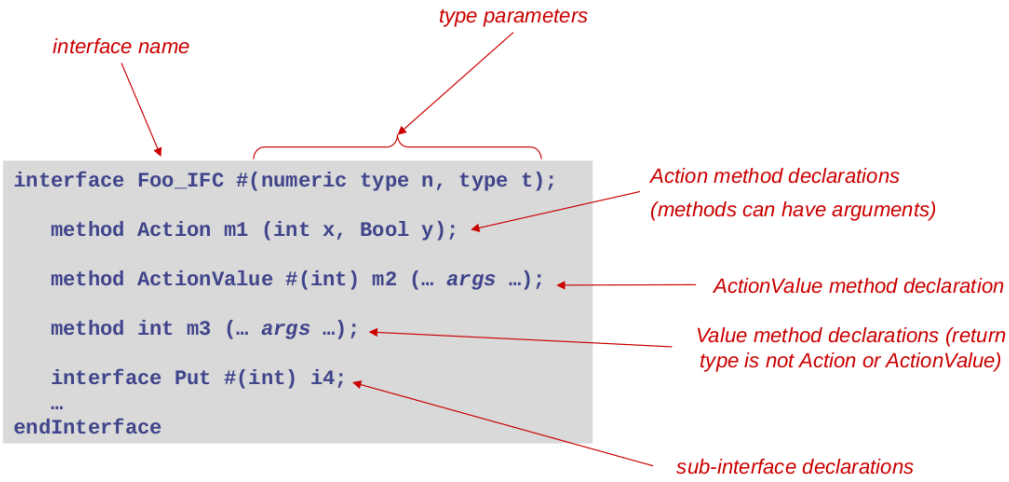

Figure 3: Contents of a BSV interface declaration.

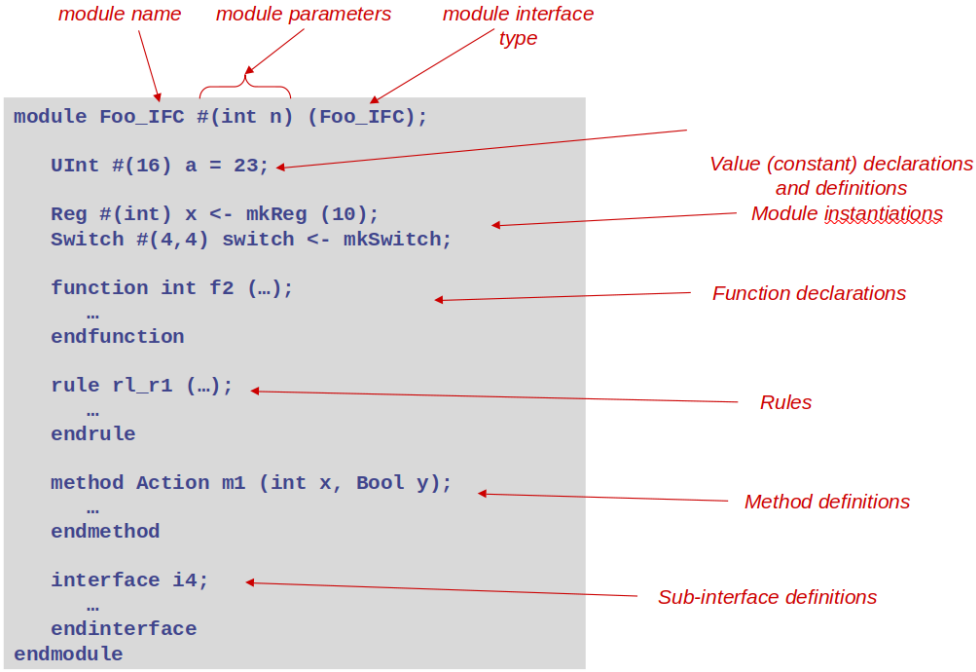

Figure 4: Contents of a BSV module declaration.

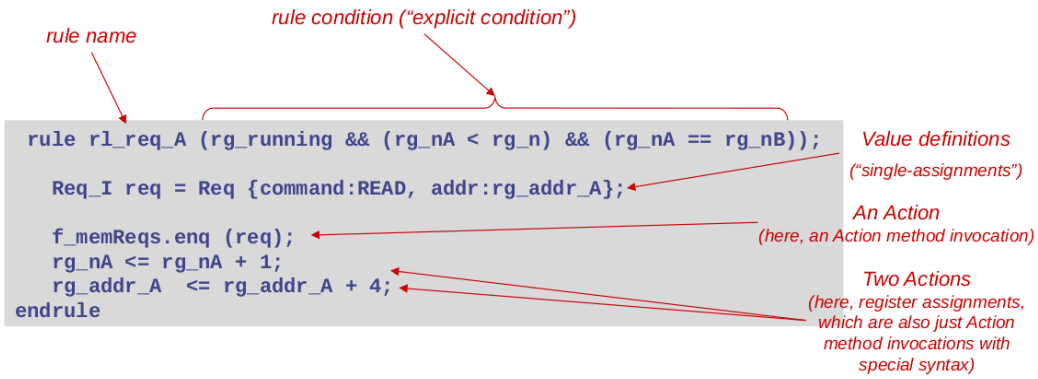

Figure 5: Contents of a BSV rule.

containing local definitions and *actions* which perform the semantic actions of the rule. Actions are typically invocations of methods in other modules.

Figure 6 shows what goes into an interface method definition (Section 5.5). These are similar in

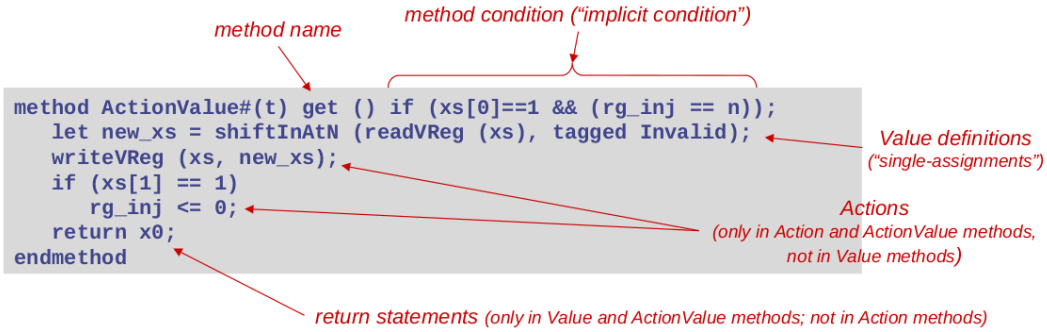

Figure 6: Contents of a BSV method definition (in a module).

structure to rules containing a condition and a body (in fact, a method definition is semantically a fragment of a rule that invokes it).

All of the above describe the *textual* structure of a BSV program. When compiled to hardware, it undergoes *static elaboration*: an instance of the top-level module instantiates its sub-modules, in turn, instantiate their sub-modules, and so on, recursively, forming a tree structure (a nesting structure). This struucture is illustrated in Figure 7

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.

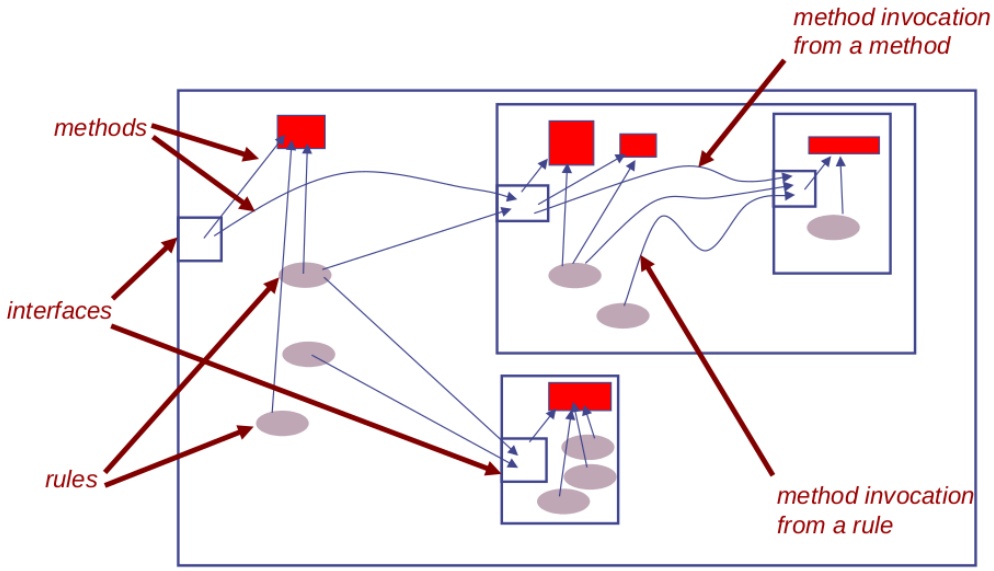

Figure 7: The (fixed) hardware strucuture of a BSV program after static elaboration.

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 13.8).

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`
	|	*sizedIntLiteral*
	|	*unsizedIntLiteral*
sizedIntLiteral	::=	*bitWidth baseLiteral*
unsizedIntLiteral	::=	[*sign*] *baseLiteral*
	|	[*sign*] *decNum*
baseLiteral	::=	(`'d` | `'D`) *decDigitsUnderscore*
	|	(`'h` | `'H`) *hexDigitsUnderscore*
	|	(`'o` | `'O`) *octDigitsUnderscore*
	|	(`'b` | `'B`) *binDigitsUnderscore*
decNum	::=	*decDigits* [*decDigitsUnderscore*]
bitWidth	::=	*decDigits*
sign	::=	`+` | `-`
decDigits	::=	{ `0...9` }
decDigitsUnderscore	::=	{ `0...9`, `_` }
hexDigitsUnderscore	::=	{ `0...9`, `a...f`, `A...F`, `_` }
octDigitsUnderscore	::=	{ `0...7`, `_` }
binDigitsUnderscore	::=	{ `0,1`, `_` }

An integer literal is a sized integer literal if a specific *bitWidth* is given (e.g., `8'o255`). There is no leading *sign* (`+` or `-`) in the syntax for sized integer literals; instead we provide unary prefix `+` or `-` operators that can be used in front of any integer expression, including literals (see Section 10). An optional sign (`+` or `-`) is part of the syntax for unsized 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
32'h_FF_FF_FF_FF
```

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 8.

An integer literal is a sized literal if a specific *bitWidth* is given (e.g., `8'o255`), in which case the literal is assumed to have type `bit [`$w-1$`:0]`. The compiler implicitly applies the function `fromSizedInteger` 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 `fromSizedInteger` is defined, i.e., for the types `Bit`, `UInt` and `Int`. The function `fromSizedInteger` is part of the `SizedLiteral` typeclass, described in *Libraries Reference Guide*.

If the literal is an unsized integer literal (a specific *bitWidth* 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 function `fromInteger` is part of the `Literal` typeclass, described in *Libraries Reference Guide*.

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

Real number literals are written with the usual Verilog notation:

realLiteral	::=	*decNum*[`.` *decDigitsUnderscore*] *exp* [*sign*] *decDigitsUnderscore*
	|	*decNum* `.` *decDigitsUnderscore*
sign	::=	`+` | `-`
exp	::=	`e` | `E`
decNum	::=	*decDigits* [*decDigitsUnderscore*]
decDigits	::=	{ `0...9` }
decDigitsUnderscore	::=	{ `0...9`, `_` }

There is no leading sign (`+` or `-`) in the syntax for real literals. Instead, we provide the unary prefix `+` and `-` operators that can be used in front of any expression, including real literals (Section 10).

If the real literal contains 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 real literal.

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

Examples:

```
1.2
0.6
2.4E10                  // exponent can be e or E
5e-3
325.761_452_e-10        // underscores are ignored
9.2e+4
```

2.4.1 Type conversion of real literals

Real literals can be used to specify values for real types. By default, real literals are assumed to have the type `Real`. BSV uses its systematic overloading resolution mechanism to perform these type conversions. Overloading resolution is described in more detail in Section 8. There are additional functions defined for `Real` types, provided in the `Real` package described in *Libraries Reference Guide*.

The function `fromReal` (described in *Libraries Reference Guide*) converts a value of type `Real` into a value of another datatype. 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 of that type from real literals.

The type `FixedPoint`, defined in the `FixedPoint` package, defines a type for representing fixed point numbers. The `FixedPoint` type has an instance of `RealLiteral` defined for it and contains functions for operating on fixed-point real numbers.

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
```

2.5.1 Type conversion of string literals

String literals are used to specify values for string types. BSV uses its systematic overloading resolution mechanism to perform these type conversions. Overloading resolution is described in more detail in Section 8.

Whenever you write a string literal in BSV there is an implicit `fromString` applied to it, which defaults to type `String`.

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	::=	*moduleDef*
	|	*interfaceDecl*

| *typeDef*
| *varDecl* | *varAssign*
| *functionDef*
| *typeclassDef*
| *typeclassInstanceDef*
| *externModuleImport*

packageIde ::= *Identifier*

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

An export item can specify an identifier defined elsewhere within the package, making the identifier accessible oustside the 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. When the exported identifier is not a structure or union type definition, the `(..)` has no effect on the exported identifier.

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. To export all identifiers from packages `Foo` and `Bar`, add the statement: `export Foo ::*`

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. We recommend, however, 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 Importing the Standard Prelude and Libraries

The Standard Prelude is imported *implicitly* into every BSV package; there is no explicit `import` statement. All other library packages need an `import` statement.

Although not a requirement, as a matter of good style we recommend against using any of the Standard Prelude names for new definitions in your programs. That would require the entity's name to be qualified with the package name, and may also be confusing to others who read the code.

4 Types

BSV provides a strong, static type-checking environment; every variable and every expression in BSV has a *type*. Variables must be assigned values which have compatible types. Type checking, which occurs before program elaboration or execution, ensures that object types are compatible and applied functions are valid for the context and type.

Data types in `BSV` are case sensitive. The first character of a type is almost always uppercase, the only exceptions being the types `int` and `bit` for compatibility with Verilog.

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*	

The *Libraries Reference Guide* describes the `Prelude` package, which defines many common datatypes, and the Foundation library packages which define many more datatypes. And, users can define new types (Section 7). The following tables list some of the more commonly used types.

Common Bit Types Defined in Prelude (*Libraries Reference Guide*) Bit types are synthesizable	
Type	Description
`Bit#(n)`	Polymorphic data type containing n bits
`UInt#(n)`	Unsigned fixed-width representation of an integer value of n bits
`Int#(n)`	Signed fixed-width representation of an integer value of n bit
`Bool`	Type which can have two values, `True` or `False`
Maybe	Used to tag values as `Valid` or `Invalid`, where valid values contain data
Tuples	Predefined structures which group a small number of values together

Common Non-Bit Types Defined in Prelude (*Libraries Reference Guide*)	
Type	Description
Integer	Non-synthesizable data type used for integer values and functions
Real	Non-synthesizable data type which can represent numbers with a fractional component
String, Char	Data type representing string literals
Fmt	Representation of arguments to the `$display` family of tasks

Common Interface Types Defined in Prelude and Foundation Library Packages	
Type	Description
`Reg`	Register interface
`FIFO`	FIFO interfaces
`Clock`	Abstract type with a oscillator and a gate
`Reset`	Abstract type for a reset
`Inout`	Type used to pass Verilog inouts through a BSV module

Types Used by the Compiler	
Type	Description
`Action`	An expression intended to act on the state of the circuit
`ActionValue`	An expression intended to act on the state of the circuit
`Rules`	Used to represent one or more rules as a first class type
`Module`	A hardware module containing sub-modules, rules and an interface

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 *numeric* 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 (16 is a numeric type)
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 $m \geq 0$.

Type parameters can also be strings (known as *string* types). These are not common, but are quite useful in the generics library, described in the *Libraries Reference Guide*. Examples:

```
MetaData#("Prelude","Maybe",PrimUnit,2)
MetaConsNamed#("Valid",1,1)
```

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 b
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 8.1, and the general facility of type classes (overloading groups), of which provisos form a part, is described in Section 8. 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 8.5, which converts a type to a numeric type.

4.2.2 The pseudo-function `stringof` (or `stringOf`)

A function `stringof` (or `stringOf`) similar to `valueof` exists to convert a string type to a string value. Example:

```
instance CShow'#(Meta#(MetaConsNamed#(name, idx, nfields), a))
 provisos (CShowSummand#(a));
   function cshow'(tagged Meta {x});
     return $format(stringOf(name), " {", cshowSummandNamed(x), "}");
   endfunction
endinstance
```

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 8. 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(FShow)    // Meaning:  automatically define fshow to convert
                   // to a Fmt representation for $display functions

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 subhierarchy 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.[4]

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:

[4] 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*."

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

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

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

5.2 Interface declaration

In BSV an interface contains members that are called *methods* (an interface may also contain subinterfaces, which are described in Section 5.2.1). To first order, a method can be regarded exactly like a function, i.e., it is a procedure that takes zero or more arguments and returns a result. Thus, method declarations inside interface declarations look just like function prototypes, the only difference being the use of the keyword `method` instead of the keyword `function`. Each method represents 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` | `string`] `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.[5] The `pop` method takes no arguments, and simply returns an enable wire which, when asserted, will discard the element from the top of the stack. The `top` method takes no arguments, and returns a value of type `int`, i.e., the element at the top of the stack.

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

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

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

```
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:

[5] The type `Action` is discussed in more detail in Section 10.6.

```
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` (see *Libraries Reference Guide*) 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` *typeDefType* `;`

where *typeDefType* 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 *Libraries Reference Guide*). 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          ::= [ attributeInstances ]
                       moduleProto
                           { moduleStmt }
                       endmodule [ : identifier ]

moduleProto        ::= module [ [ type ] ] identifier
                       [ moduleFormalParams ] ( [ moduleFormalArgs ] ) [ provisos ];

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

moduleFormalParam  ::= [ attributeInstances ] [ parameter ] type identifier

moduleFormalArgs   ::= [ attributeInstances ] type
                   |   [ attributeInstances ] type identifier
                       { , [ attributeInstances ] 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 *bsc* 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 compiler *bsc*, 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 8.

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*
	|	*varDo* | *varDeclDo*
	|	*functionCall*
	|	*systemTaskStmt*
	|	(*expression*)
	|	*returnStmt*
	|	*varDecl* | *varAssign*
	|	*functionDef*
	|	*moduleDef*
	|	<*module*>*BeginEndStmt*
	|	<*module*>*If* | <*module*>*Case*
	|	<*module*>*For* | <*module*>*While*

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 ::= [*attributeInstances*]
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.

An optional documentation attribute (Section 14.7) placed before the module instantiation will place a comment in the generated Verilog file.

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

Deprecated: long-form instantiation was originally introduced into BSV for compatibility with SystemVerilog. In practice, people rarely use this, preferring the short-form instead. As a matter of more universally recognizable style, we suggest using the short form.

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.

An optional documentation attribute (Section 14.7) placed before the module instantiation will place a comment in the generated Verilog file.

moduleInst	::=	[*attributeInstances*] *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 of the long form instantiation 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 11). 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 10.13, 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 9.8 on function definitions.

See also Section 10.12 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 10.6 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 10.7), 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

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 ;
                       { interfaceStmt }
                     endinterface [ : identifier ]

interfaceStmt    ::= methodDef
                 |   subinterfaceDef
                 |   expressionStmt

expressionStmt   ::= varDecl | varAssign
                 |   functionDef
                 |   moduleDef
                 |   <expression>BeginEndStmt
                 |   <expression>If | <expression>Case
                 |   <expression>For | <expression>While
```

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 *interfaceStmt*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 10.6 and 10.7 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 10.12) 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 14.3. 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 must be unique within a module.

The *ruleCond*, if present, may be a boolean expression, or it may be a pattern-match (pattern matching is described in Section 11). 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 10.6, 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 10.13 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 8.

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.[6]

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.

[6] 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.

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 compiler which module(s) are to be synthesized. A BSV module that is marked for code generation is said to be a *synthesized* module.

Polymporphic modules cannot be synthesized as-is (since hardware datapaths, register sizes, etc. will depend on the particular types in each instance). A common pattern is: we define a complex, polymorphic module M; we define one or more very short (2-3 lines each) module definitions M_1, M_2, ... each of which *instantiates* M at a specific type. These module, then, can be synthesized into hardware.

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` (see *Libraries Reference Guide*);
- 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).

A module can be marked for synthesis in one of two ways.

1. A module can be annotated with the `synthesize` attribute (see section 14.1.1). The appropriate syntax is shown below.

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

2. Alternatively, the `-g` compiler flag can be used on the *bsc* compiler 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 *bsc* 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 preceding 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.

5.9 Other module types

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

There are many types of modules in BSV. The default BSV module type is `Module #(ifc)`. When instantiated, a `Module` adds state elements and rules to the accumulation of elements and rules already in the design. This is the only synthesizable module type, but other types can exist. For instance, the type `ModuleCollect#(t,ifc)` (see *Libraries Reference Guide*) allows items other than states and rules to be collected while elaborating the module structure.

For most applications the modules in the design will be of type `Module` and the type can be inferred. When you write:

```
module mkMod(Ifc);
...
endmodule
```

the compiler doesn't force this code to be specific to the basic `Module` type, although it usually will be. BSV allows this syntax to be used for any type of module; what you are declaring here is a polmorphic module. In fact, it is really just a function that returns a module type. But instead of returning back the type `Module`, it returns back any type `m` with the proviso that `m` is a module. That is expressed with the proviso:

```
IsModule#(m,c)
```

However, if the code for `mkMod` uses a feature that is specific to one type of module, such as trying to add to the collection in a `ModuleCollect` module, then type inference will discover that your module can't be any module type `m`, but must be a specific type (such as `ModuleCollect` in this example).

In that case, you need to declare that the module `mkMod` works for a specific module type using the bracket syntax:

```
module [ModuleCollect#(t)] mkMod(Ifc);
```

In some instances, type inference will determine that the module must be the specific type `Module`, and you may get a signature mismatch error stating that where the code said any module type `m`, it really has to be `Module`. This can be fixed by explicitly stating the module type in the module declaration:

```
module [Module] mkMod(Ifc);
```

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 8.1). 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.*[7] 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.

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

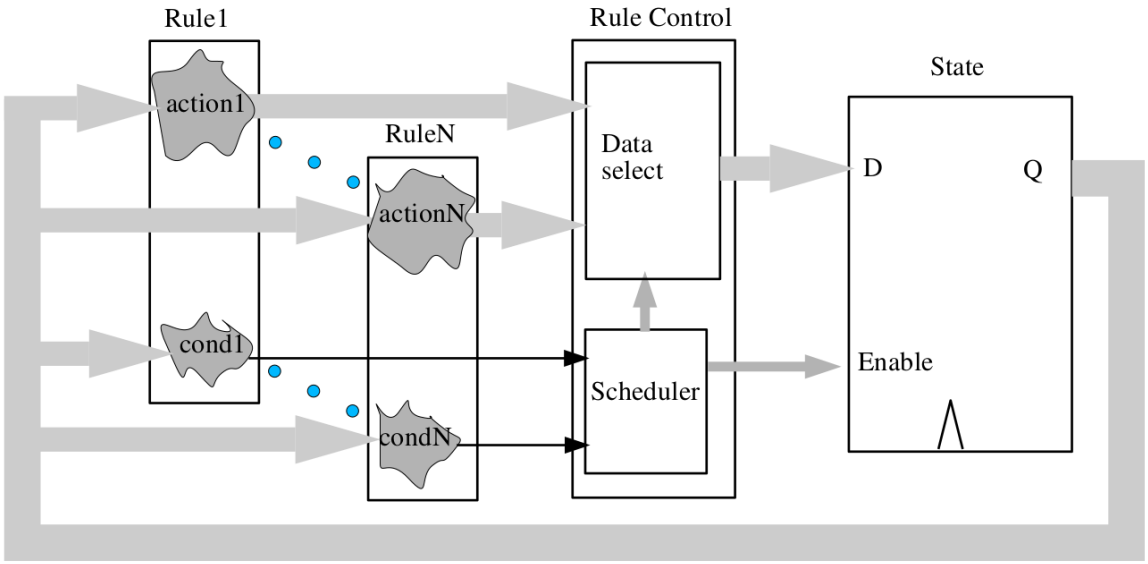

Figure 8: A general scheme for mapping an N-rule system into clocked synchronous hardware.

Figure 8 illustrates a general scheme to compose rule components when mapping the design to clocked synchronous hardware. The State box lumps together all the state elements in the BSV design (as described earlier, state elements are explicitly specified in BSV). The BSV compiler produces a rule-control circuit which conceptually takes all the enable (cond) signals and all the data (action) outputs and controls which of the data outputs are actually captured at the next clock in the state elements. The enable signals feed a *scheduler* circuit that decides which of the rules will actually fire. The scheduler, in turn, controls data multiplexers that select which data outputs reach the data inputs of state elements, and controls which state elements are enabled to capture the new data values. Firing a rule simply means that the scheduler selects its data output and clocks it into the next state.

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

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

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

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

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

Note that the scheme in Figure 8 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 14.3 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 [8] 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 14.3.3). 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 be executed an unpredictable number of times 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.

[8] 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.

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 register. 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
`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)

The annotations `SA` and `SAR` are provided for documentation purposes only; they are not supported in the BSV language.

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 must be defined at the top level of a package.

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` | `string`] `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 Tuple3#(a, a, a) Triple#(type a);

typedef 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;
```

The `typedef` statement must always be at the top level of a package, not within a module. To introduce a local name within a module, use `Alias`, `NumAlias` or `StrAlias` (see *Libraries Reference Guide*). Since these introduce new names which are type variables as opposed to types, the new names must begin with lower case letters. `Alias` is used for types which can be the types of variables, while `NumAlias` and `StrAlias` are used to give new names to numeric and string types. Example:

```
module mkMod(Ifc)
  provisos (Alias#(Bit#(crc_size), crc));

module mkRAM(RAMIfc)
  provisos (NumAlias#(addr_size, TLog#(buff_size)));
```

7.2 Enumerations

typedefEnum ::= `typedef enum {` *typedefEnumElements* `}` *Identifier* [*derives*] `;`
typedefEnumElements ::= *typedefEnumElement* { `,` *typedefEnumElement* }
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 8. 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. Also defined in `deriving (FShow)`, which defines a `Fmt` type for the labels for use with `$Display` functions. These specifications can be combined, e.g., `deriving (Bits, Eq, Bounded, FShow)`. All these default choices for representation, equality and bounds can be overridden (see Section 8).

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 11), 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 | string ] 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 8. 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 10.11 on struct and union expressions describes how to construct struct and union values and to access and update members. Section 11 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 13.4 for an important, predefined set of struct types called *Tuples* for adhoc structs of between two and eight members.

8 Type classes (overloading groups) and provisos

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

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 8.2). Further, a type class is explicitly populated with a new instance type t, using an `instance` declaration (Section 8.3), in which the programmer provides the specifications for the overloaded functions for the type t.

8.1 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 10.10) to some expression that narrows down its type.
- Add a proviso to the surrounding construct.

8.2 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` | `string`] `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):[9]

```
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);
```

[9] 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.)

```
      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 mkConnection#(a x1, b x2) (Empty);
endtypeclass
```

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`.

8.3 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 *types* are an instance of type class *typeclassIde* with the given provisos. The *varAssigns*, *functionDefs* and *moduleDefs* 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`, `Bounded`, and `FShow`. 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);
```

8.4 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 Bit#(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 (Bit#(2) x);
        if      (x == 3) return Red;
        else if (x == 2) return Green;
        else if (x == 1) return Blue;
        else ? //Illegal opcode; return unspecified value
    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 8.6).

8.5 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
```

8.6 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 8.4 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`.

8.7 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.

8.8 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`).

8.9 Deriving `FShow`

The intent of the `FShow` type class is to format values for use with the `$display` family of functions.

When attaching a `deriving(FShow)` clause to a user-defined type, the instance derived for the `FShow` type class can be described as follows:

- For an enum type, the output contains the enumerated value.

  Example:

```
typedef enum { Red, Blue, Green } Colors deriving (FShow);
...
      $display("Basic enum");
      Colors be0 = Red;
      Colors be1 = Blue;
      Colors be2 = Green;
      $display(fshow(be0));
      $display(fshow(be1));
      $display(fshow(be2));
```

Displays:

```
Basic enum
Red
Blue
Green
```

- For a struct type, the output contains the struct name, with each value prepended with the name of the field. The values are formatted with `FShow` according to the data type.

  ```
  Struct_name {field1_name: value1, field2_name: value2....}
  ```

  Example:

  ```
  typedef struct {
     Bool      val_bool;
     Bit#(8)   val_bit;
     UInt#(16) val_uint;
     Int#(32)  val_int;
  } BasicS deriving (FShow);
  ...
        $display("Basic struct");
        BasicS bs1 =
            BasicS { val_bool: True, val_bit: 22,
                     val_uint: 'hABCD, val_int: -'hABCD };
        $display(fshow(bs1));
  ```

  Displays:

  ```
  Basic struct
  BasicS { val_bool: True, val_bit: 'h16, val_uint: 43981, val_int:      -43981 }
  ```

- For a tagged union type, the output contains the name of the tag followed by the value. The values are formatted with `FShow` according to the data type.

  ```
  tagged Tag1 value1
  tagged Tag2 value2
  ```

  Example:

  ```
  typedef union tagged {
     Bool      Val_bool;
     Bit#(8)   Val_bit;
     UInt#(16) Val_uint;
  ```

```
   Int#(32)  Val_int;
} BasicU deriving (FShow);
...
      $display("Basic tagged union");
      BasicU bu0 = tagged Val_bool True;
      BasicU bu1 = tagged Val_bit 22;
      BasicU bu2 = tagged Val_uint 'hABCD;
      BasicU bu3 = tagged Val_int -'hABCD;
      $display(fshow(bu0));
      $display(fshow(bu1));
      $display(fshow(bu2));
      $display(fshow(bu3));
```

Displays:

```
Basic tagged union
tagged Val_bool True
tagged Val_bit  'h16
tagged Val_uint 43981
tagged Val_int      -43981
```

8.10 Deriving type class instances for isomorphic types

Generally speaking, the `deriving(...)` clause can only be used for the predefined type classes `Bits`, `Eq`, `Bounded`, and `FShow`. 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);
```

8.11 Monad

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

The `Monad` typeclass (idea taken directly from Haskell) is an abstraction which allows different composition strategies and is useful for combining computations into more complex computations. Monads are certain types with `bind` and `return` operations that satisfy certain mathematical properties.

BSV programmers use the `Module` and `Action` monads all the time (often without being aware of it). The `Monad` typeclass allows you to define and use new monads, just as you can in Haskell. It is a rather large subject to describe all the possibilities and opportunities of monad-based programming; we refer readers to the extensive literature on the topic, particularly in the Haskell ecosystem.

9 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.

9.1 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` and `Vector` data types (see *Libraries Reference Guide*). 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.

The type of array variables is generally expressed anonymously, using the bracket syntax. It is equivalent to the `Array` type (see *Libraries Reference Guide*), which can be used when an explicit type name is needed.

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};
```

9.2 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 9.4.

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 10.11.

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 10.11.

9.3 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.

9.4 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
```

9.4.1 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[10]. 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,

```
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)[11], 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

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

[11] 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 9.2)

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

9.4.2 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[12].

Example: A range-notation register write

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

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

9.4.3 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`.

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

9.5 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
```

9.6 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 11, 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 11.

9.7 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 12 on FSMs for an alternative use of loops to express time-based (temporal) behavior.

9.7.1 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];
```

9.7.2 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];
```

9.8 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 8). 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	::=	[*attributeInstances*] *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	::=	*returnStmt*
	|	*varDecl* | *varAssign*
	|	*functionDef*
	|	*moduleDef*
	|	<*functionBody*>*BeginEndStmt*
	|	<*functionBody*>*If* | <*functionBody*>*Case*
	|	<*functionBody*>*For* | <*functionBody*>*While*
returnStmt	::=	`return` *expression* `;`

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

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

Example. The boolean negation function:

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

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

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

Example. The factorial function, using a loop:

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

Example. The factorial function, using recursion:

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

9.8.1 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 9.8. 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();
```

9.8.2 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
```

9.8.3 Higher-order functions

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

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` (see *Libraries Reference Guide*):

```
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, in BSV, higher-order functions may be used in *synthesizable* code, i.e., the compiler can produce RTL hardware. This often comes as a surprise to people familiar with Verilog/SystemVerilog and VHDL. The key insight is that static elaboration effectively "flattens" out all designs—higher-order functions (like ordinary functions) get substituted and inlined—and there is no problem synthesizing the elaborated design.

10 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*
	|	*realLiteral*
	|	*stringLiteral*
	|	*systemFunctionCall*
	|	(*expression*)
	|	··· see other productions ···

10.1 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
```

10.2 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 11, 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 11, on pattern matching.

10.3 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.

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

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

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

is parsed as follows:

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

and not as follows:

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

10.4 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 an acceptable index type (e.g. `Integer`, `Bit#(n)`, `Int#(n)`, or `UInt#(n)`). 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 and `bit[0]` is the least significant bit. 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 10.10). 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.

10.5 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*]
{ *expressionStmt* }
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.

expressionStmt ::= *varDecl* | *varAssign*
| *functionDef*
| *moduleDef*
| <*expression*>*BeginEndStmt*
| <*expression*>*If* | <*expression*>*Case*
| <*expression*>*For* | <*expression*>*While*

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);
```

10.6 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 9.4 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	::=	*regWrite*
	|	*varDo* | *varDeclDo*
	|	*functionCall*
	|	*systemTaskStmt*
	|	`(` *expression* `)`
	|	*actionBlock*
	|	*varDecl* | *varAssign*
	|	*functionDef*
	|	*moduleDef*
	|	<*action*>*BeginEndStmt*
	|	<*action*>*If* | <*action*>*Case*
	|	<*action*>*For* | <*action*>*While*

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;
```

10.7 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	::=	*regWrite*
	|	*varDo* | *varDeclDo*
	|	*functionCall*
	|	*systemTaskStmt*
	|	`(` *expression* `)`
	|	*returnStmt*
	|	*varDecl* | *varAssign*
	|	*functionDef*
	|	*moduleDef*
	|	<*actionValue*>*BeginEndStmt*
	|	<*actionValue*>*If* | <*actionValue*>*Case*
	|	<*actionValue*>*For* | <*actionValue*>*While*

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 a method of type `(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
```

10.8 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.

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 9.8.3.

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
```

10.9 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
```

10.10 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).

10.11 Struct and union expressions

Section 7.3 describes how to define struct and union types. Section 9.1 describes how to declare variables of such types. Section 9.2 describes how to update variables of such types.

10.11.1 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.

10.11.2 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 9.2.

10.11.3 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 };
```

10.11.4 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 11.

Update of tagged union variables is described in Section 9.2.

10.12 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	::=	*methodDef*
	|	*subinterfaceDef*
	|	*expressionStmt*
expressionStmt	::=	*varDecl* | *varAssign*
	|	*functionDef*
	|	*moduleDef*
	|	<*expression*>*BeginEndStmt*
	|	<*expression*>*If* | <*expression*>*Case*
	|	<*expression*>*For* | <*expression*>*While*

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
```

10.12.1 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.

10.13 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	::=	*rule* | *expressionStmt*
expressionStmt	::=	*varDecl* | *varAssign*
	|	*functionDef*
	|	*moduleDef*
	|	<*expression*>*BeginEndStmt*
	|	<*expression*>*If* | <*expression*>*Case*
	|	<*expression*>*For* | <*expression*>*While*

A rule expression is optionally preceded by an *attributeInstances*; these are described in Section 14.3. 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
```

11 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*	
	|	*realLiteral*	
	|	*stringLiteral*	
	|	*Identifier*	Enum label
taggedUnionPattern	::=	`tagged` *Identifier* [*pattern*]	
structPattern	::=	*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.

A struct pattern consists of an identifier followed by braces, where the identifier 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 13.4).

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. Literals, including integer literals, string literals, and real literals, can include the wildcard character `?`. A 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.

11.1 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 9.6. 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
```

11.2 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
```

11.3 Pattern matching in if statements and other contexts

If statements are described in Section 9.6. As the grammar shows, the predicate (*condPredicate*) can be a series of pattern matches and expressions, separated by `&&&`. Example:

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

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

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

expressions, and succeed if they evaluate to `True`. If the whole *condPredicate* succeeds, the bound pattern variables are available in the corresponding "consequent" arm of the construct.

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

- Conditional expressions (Section 10.2):

  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 10.13):

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

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

- Conditions of methods (Sections 5.5 and 10.12):

  `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 (11.2).

```
// 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
```

11.4 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
```

12 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.

The FSM sublanguage in BSV does not add any fundamental new power or semantics to the language. The analogy is "structured programming" (while-loops, for-loops, and if-then-else constructs) *vs. goto* statements (arbitrary jumps) in conventional programming languages. Structured programming more clearly and succinctly expresses certain common control structures (and therefore are also easier to modify and maintain) than unconstrained *gotos*. Similarly, BSV's FSM sublanguage more clearly and succinctly expresses certain common sequential and concurrent process structures that could, in principle, have all been coded directly with rules.

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 10.6), obeying all the scheduling semantics of actions (Section 6.2).

FSMs are particularly useful in testbenches, where one orchestrates sequential, parallel and concurrent stimulus generation for a DUT (Design Under Test).

In order to use this sublanguage, it is necessary to import the `StmtFSM` package:

```
import StmtFSM :: * ;
```

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 first-class 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.

12.1 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" or "processes") 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.

12.2 FSM 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`.

There is a one clock cycle delay after the `start` method is asserted before the FSM starts. This insulates the `start` method from many of the FSM schedule constraints that change depending on what computation is included in each specific FSM. Therefore, it is possible that the StmtFSM is enabled when the `start` method is called, but not on the next cycle when the FSM actually starts. In this case, the FSM will stall until the conditions allow it to continue.

Interfaces	
Name	Description
`FSM`	The state machine interface
`Once`	Used when an action should only be performed once

- `FSM` Interface

  The `FSM` interface provides four methods; `start`, `waitTillDone`, `done` and `abort`. 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. The `done` method is `True` before the FSM has run the first time. 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`. The `abort` method immediately exits the execution of the FSM.

```
interface FSM;
    method Action start();
    method Action waitTillDone();
    method Bool   done();
    method Action abort();
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. State machine comes out of reset as done.
`abort`	`Action`	Exits execution of the state machine.

- `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.

12.3 FSM 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.

12.4 FSM 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  <-  mkRegU;     // instantiate register with interface i
Reg#(int) j  <-  mkRegU;     // instantiate 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 sequential 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
```

12.5 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 from within an FSM statement block. `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 to 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
```

12.6 FSM performance caveat

We mentioned, earlier, that FSMs do not add any fundamental new power or semantics to the language. Anything expressed using `StmtFSM` can also be written directly using rules; such a rendering is usually more verbose and its logical process structure is usually less transparent. However, writing explicit rules does enable the programmer to perform certain optimizations that would not happen in `StmtFSM` compilation. For example, in a `StmtFSM` for-loop with an incrementing index register, can the `Action` for the index-register increment be merged into any of the `Action`s in the for-loop body (potential conflicts)? The human programmer can often answer this question easily, whereas `StmtFSM` compiler takes a pessimistic extra cycle after the loop body for this action.

13 Important primitives

These primitives are available via the Standard Prelude package and other standard libraries. See also *Libraries Reference Guide* more useful libraries.

13.1 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 10.4.

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

```
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 13.4) 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.

13.1.1 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 *Libraries Reference Guide*) 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.

13.2 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 *Libraries Reference Guide* 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.

13.3 String and Char

The type `String` is defined in the Standard Prelude package (see *Libraries Reference Guide*) along with the type `Char`. Strings and chars are mostly used in system tasks (such as `$display`). Strings can be concatenated using the `+` infix operator or, equivalently, the `strConcat` function. Strings can be tested for equality and inequality using the `==` and `!=` operators. String literals, written in double-quotes, are described in Section 2.5.

The `Char` type provides the ability to traverse and modify the characters of a string. The `Char` type can be tested for equality and inequality using the `==` and `!=` operators. The `Char` type can also be compared for ordering through the `Ord` class.

13.4 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      ...             Tuple8#(type a, ..., type h) ...;
```

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

```
tuple2 (e1, e2)
tuple3 (e1, e2, e3)
...
tuple8 (e1, e2, e3, ..., e8)
```

where the expressions `e`J evaluate to the component values of the tuples. The tuple types are defined in the Standard Prelude package (see *Libraries Reference Guide*).

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

```
tpl_1 (e)
tpl_2 (e)
...
tpl_8 (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
```

13.5 Registers

The most elementary module available in BSV is the register (see *Libraries Reference Guide*), 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 9.4 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)
```

13.6 FIFOs

Package `FIFO` (see *Libraries Reference Guide*) 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`.

13.7 FIFOFs

Package `FIFOF` (see *Libraries Reference Guide*) 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.

13.8 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* ;

13.8.1 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`, `Char`, `Bit#(n)` (i.e., of type `bit[n-1:0]`), `Integer`, or any type that is a member of the overloading group `Bits`. Values of type `Char` are treated as a `String` of one character, by implicitly converting to `String`. 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` and `Char` 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[13]:

`%d`	Output a number in decimal format
`%b`	Output a number in binary format
`%o`	Output a number in octal format
`%h`	Output a number in hexadecimal format
`%c`	Output a character with given ASCII code
`%s`	Output a string (argument must be a string)
`%t`	Output a number in time format
`%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 10.7) 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.

13.8.2 $format

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

[13] 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.

BSV 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` (see *Libraries Reference Guide*). `Fmt` representations of data objects can thus be written hierarchically and applied to polymorphic types. The `FShow` typeclass is 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
```

13.8.3 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`.

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

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

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.

13.8.4 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 13.8.1) 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
```

13.8.5 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 13.8.1.

The *bsc* 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.

13.8.6 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
```

13.8.7 Flushing output

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

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

13.8.8 Stopping simulation

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

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

The `$finish` task causes simulation to stop immediately and exit back to the operating system. The `$stop` task causes simulation to suspend immediately and enter an interactive mode. The optional argument expressions can be 0, 1 or 2, and control the verbosity of the diagnostic messages printed by the simulator. the default (if there is no argument expression) is 1.

13.8.9 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.

13.8.10 Time functions

systemFunctionCall ::= `$time` | `$stime`

These system function calls are conceptually of `ActionValue` type (see Section 10.7), 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.

13.8.11 Real functions

There are two system tasks defined for the `Real` data type (see *Libraries Reference Guide*), 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* `)`

13.8.12 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 10.7), 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.

14 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.

14.1 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	14.1.1	√	
noinline	14.1.2		√

14.1.1 `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.

14.1.2 `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
```

14.2 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=	14.2.1			√	
enable=	14.2.1			√	
result=	14.2.1			√	
prefix=	14.2.1			√	√
port=	14.2.1			√	
always_ready	14.2.2	√	√	√	√
always_enabled	14.2.2	√	√	√	√

There is a direct correlation between interfaces in BSV 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.

BSV 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.

14.2.1 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 subinterface 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 subinterface declarations to aid the renaming of interface hierarchies. By default, interface hierarchies are named by prefixing the subinterface 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 subinterface name. This is demonstrated in the example in Section 14.2.3.

14.2.2 Port protocol attributes

The port protocol attributes `always_enabled` and `always_ready` remove unnecessary ports. These attributes are applied to synthesized modules, methods, interfaces, and subinterfaces at the top level only. If the module is not synthesized, the attribute is ignored. 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 subinterface 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

(* synthesize *)
(* 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:

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

14.2.3 Interface attributes example

```
(* always_ready *)                  // all methods in this and all subinterface
                                    // 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
```

14.3 Scheduling attributes

Attribute name	Section	Module definitions	`rule` definitions	`rules` expressions
fire_when_enabled	14.3.1		√	
no_implicit_conditions	14.3.2		√	
descending_urgency	14.3.3	√	√	√
execution_order	14.3.4	√	√	√
mutually_exclusive	14.3.5	√	√	√
conflict_free	14.3.6	√	√	√
preempts	14.3.7	√	√	√

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.

14.3.1 `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 rules try to read and write 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.

14.3.2 `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.

14.3.3 `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 10.13), 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 submodule 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 submodule 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.

14.3.4 `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 14.3.3) 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 14.3.3) 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.

14.3.5 `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 14.3.3) 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.

14.3.6 `conflict_free`

Like the `mutually_exclusive` rule attribute (section 14.3.5), 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 14.3.3) 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).

14.3.7 `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 14.3.3), 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.

14.4 Evaluation behavior attributes

14.4.1 `split` and `nosplit`

Attribute name	Section	Action statements	ActionValue statements
split/nosplit	14.4.1	√	√

The `split/nosplit` attributes are applied to `Action` and `ActionValue` statements, but cannot precede 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 preceding 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 *Bluespec Compiler (BSC) 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.

14.5 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=	14.5.1	√
gate_prefix=	14.5.1	√
reset_prefix=	14.5.1	√
gate_input_clocks=	14.5.2	√
gate_all_clocks	14.5.2	√
default_clock_osc=	14.5.3	√
default_clock_gate=	14.5.3	√
default_gate_inhigh	14.5.3	√
default_gate_unused	14.5.3	√
default_reset=	14.5.3	√
clock_family=	14.5.4	√
clock_ancestors=	14.5.4	√

14.5.1 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 14.5.3 and 14.6.1), 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).

14.5.2 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_clocks = "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.

14.5.3 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 14.5.2) 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.

14.5.4 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 `clock_family`.

14.6 Module argument and parameter attributes

The attributes in this section are applied to module arguments and module parameters. The following table shows which type of module argument or parameter each attribute can be applied to. More information on module arguments and module parameters can be found in Section 5.3.

In most instances BSV allows arguments and parameters to be enclosed in a single set of parentheses, in which case the `#` is eliminated. However, the compiler is stricter about where attributes are placed. Only port attributes can be placed in the attribute list `()` and only parameter attributes in the parameter `#()` list.

Examples:

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

module mkMod #((* parameter="DATA_WIDTH" *) parameter Int#(8) width)
              ( ModIfc ifc );
```

Attribute	Section	Clock/ vector of clock	Reset/ vector of reset	Inout/ vector of inouts	Value argument	Parameter
osc=	14.6.1	√				
gate=	14.6.1	√				
gate_inhigh	14.6.1	√				
gate_unused	14.6.1	√				
reset=	14.6.1		√			
clocked_by=	14.6.2		√	√	√	
reset_by=	14.6.3			√	√	
port=	14.6.4			√	√	
parameter=	14.6.5					√

14.6.1 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,...
```

14.6.2 `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);
```

14.6.3 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);
```

14.6.4 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 14.2.1, as applied to module arguments instead of interface methods.

```
module mkMod (a_type initValue, (* port="B" *) Bool b, ModIfc ifc);

module mkMod ((* port="PBUS" *) Inout#(Bit#(32)) pbus, ModIfc ifc);
```

14.6.5 parameter=

The attribute `parameter=` allows renaming of parameters in the generated RTL. This is similar to the `port=` attribute, except that the `parameter=` attribute can only be used for parameters in the *moduleFormalParams* list. More detail on module parameters can be found in Section 5.3. The name provided in the `parameter=` attribute statement is the name generated for the parameter in the RTL.

Example:

```
module mkMod #((* parameter="DATA_WIDTH" *) parameter Int#(8) width)
              ( ModIfc ifc );
```

14.7 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=	14.7	√	√	√	√

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.

14.7.1 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
```

14.7.2 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;
...
```

14.7.3 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 preceded 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 Compiler (BSC) 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(...);
```

15 Embedding RTL in a BSV design

This section describes how to embed existing RTL modules, Verilog or VHDL, in a BSV module. The `import "BVI"` statement is used to utilize existing components, utilize components generated by

other tools, or to define a custom set of primitives. One example is the definition of BSV primitives (registers, FIFOs, etc.), which are implemented through import of Verilog modules. The `import "BVI"` statement creates a BSV wrapper around the RTL module so that it looks like a BSV module. Instead of ports, the wrapped module has methods and interfaces.

The `import "BVI"` statement can be used to wrap Verilog or VHDL modules. Throughout this section Verilog will be used to refer to either Verilog or VHDL. (One limitatation for VHDL is that BSV does not support two dimensional 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*
| *outputResetBVIStmt*
| *ancestorBVIStmt*
| *sameFamilyBVIStmt*
| *scheduleBVIStmt*
| *pathBVIStmt*
| *interfaceBVIStmt*
| *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`, `Inout`, 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, subinterface 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

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 BSV 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 14.2.2. 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

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 *bsc* 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
   Bit#(1) v = 0 ;
   port TM = v   ;  // 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

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 preceding 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) ;
    schedule (notFull) CF (notFull) ;
    // 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) ;
    // 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 and deq), 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 9.

```
import "BVI" RWire0 =
   module vMkRWire0 (VRWire0);
      ...
      method wset() enable(WSET) ;
```

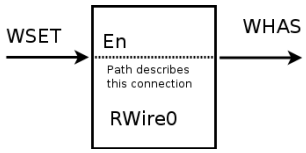

Figure 9: Path in the RWire0 Verilog module between WSET and WHAS ports

```
    method WHAS whas ;
    schedule whas CF whas ;
    schedule wset SB whas ;
    path (WSET, WHAS) ;
  endmodule: vMkRWire0
```

15.13 Interface

interfaceBVIStmt ::= `interface` *typeDefType* `;`
{ *interfaceBVIMembDecl* }
`endinterface` [`:` *typeIde*]

interfaceBVIMembDecl := *methodBVIStmt*
| *interfaceBVIStmt* `;`

An interface statement can contain two types of statements: method statements and subinterface declarations. The interface statement in BVI import is the same as any other interface statement (Section 5.2) with one difference: the method statements within the interface are BVI method statements (`methodBVIStmt` 15.2).

Example:

```
import "BVI" BRAM2 =
module vSyncBRAM2#(Integer memSize, Bool hasOutputRegister,
                   Clock clkA, Reset rstNA, Clock clkB, Reset rstNB
                    ) (BRAM_DUAL_PORT#(addr, data))
   provisos(Bits#(addr, addr_sz),
            Bits#(data, data_sz));
   ...

   interface BRAM_PORT a;
     method put(WEA, (*reg*)ADDRA, (*reg*)DIA) enable(ENA) clocked_by(clkA) reset_by(rstA);
     method DOA read() clocked_by(clkA) reset_by(rstA);
   endinterface: a

   interface BRAM_PORT b;
     method put(WEB, (*reg*)ADDRB, (*reg*)DIB) enable(ENB) clocked_by(clkB) reset_by(rstB);
     method DOB read() clocked_by(clkB) reset_by(rstB);
   endinterface: b
endmodule: vSyncBRAM2
```

Since a BVI wrapper module can only provide a single interface (`BRAM_DUAL_PORT` in this example), to provide multiple interfaces you have to create an interface hierarchy using interface statements.

The interface hierarchy provided in this example is:

```
interface BRAM_DUAL_PORT#(type addr, type data);
   interface BRAM_PORT#(addr, data) a;
   interface BRAM_PORT#(addr, data) b;
endinterface: BRAM_DUAL_PORT
```

where the subinterfaces, `a` and `b`, are defined as `interface` statements in the body of the `import "BVI"` statement.

15.14 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

[BL05] B-Lang.org. Bluespec Compiler (BSC) Libraries Reference Guide, Since 2005. https://github.com/B-Lang-org/bsc/tree/main/doc/libraries_ref_guide.

[Hoe00] J. Hoe. *Operation-Centric Hardware Description and Synthesis*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, June 2000.

[IEE02] IEEE. IEEE Standard VHDL Language Reference Manual, IEEE Std 1076-1993, 2002.

[IEE05] IEEE. IEEE Standard Verilog (R) Hardware Description Language, 2005. IEEE Std 1364-2005.

[IEE12] IEEE. IEEE Standard for Standard SystemC Language Reference Manual, January 9 2012. IEEE Std 1666-2011.

[IEE13] IEEE. IEEE Standard for System Verilog—Unified Hardware Design, Specification and Verification Language, 21 February 2013. IEEE Std 1800-2012.

[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
E
SB
SBR
action                          endaction
actionvalue                     endactionvalue
ancestor
begin
bit
case                            endcase
clocked_by
default
default_clock
default_reset
dependencies
deriving
determines
e
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
string
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		time	var
small		timeprecision	vectored
solve		timeunit	virtual
specify	endspecify	tran	void
specparam		tranif0	wait
static		tranif1	wait_order
string		tri	wand
strong0		tri0	weak0
strong1		tri1	weak1
struct		triand	while
super		trior	wildcard
supply0		trireg	wire
supply1		type	with
table	endtable	typedef	within
tagged		union	wor
task	endtask	unique	xnor
this		unsigned	xor
throughout		use	

B A Brief History of BH (Bluespec Haskell/Classic) and BSV (Bluespec SystemVerilog)

The original research on using *rules* to specify hardware behavior, and of compiling rules into synthesizable Verilog was conducted by James Hoe for his Ph.D. thesis [Hoe00] under the supervision of Prof. Arvind at MIT in the late 1990s.

In 2000, activity moved to Sandburst Corp., a fabless semiconductor startup company based in Massachusetts, producing enterprise-class and metropolitan-class network router chips. The first Bluespec language and compiler were designed at Sandburst by Lennart Augustsson, with the assistance of several others including Jacob Schwartz, Mieszko Lis, Joe Stoy, Arvind and Rishiyur Nikhil. The language was very much Haskell-influenced in its syntax, in its type system, and in its static elaboration semantics. The compiler itself was written in Haskell. During this period, the language was used only in-house, principally for hardware-models of Sandburst's chip architecture.

In 2003, a separate company, Bluespec, Inc., also in Massachusetts, was spun out from Sandburst Corp. to focus on the Bluespec language and compiler as its central product, targeted at digital design engineers worldwide. Because of the target audence's deep familiarity with Hardware Design Languages (principally Verilog and VHDL), and complete unfamiliarity with Haskell, the syntax of the Bluespec language was completely redesigned to be as similar as possible to SystemVerilog, which was just then being defined as an industry standard. Members of Bluespec, Inc. joined the IEEE SystemVerilog standardization effort, for closer alignment (the "tagged unions" and "pattern-matching" constructs in the SystemVerilog standard were a Bluespec contribution, inspired directly by Haskell).

The original Haskell-like syntax (at Sandburst Corp.) and the subsequent SystemVerilog-like syntax (at Bluespec, Inc.) are merely two different syntaxes for the same language, with the same semantics. They are simply two alternative parsing front-ends for the common *bsc* compiler. The original language is called "Bluespec Classic" or BH (for "Bluespec Haskell") and the subsequent language is called BSV (for "Bluespec SystemVerilog"). Both syntaxes continue to be accepted by *bsc*, and a single system can mix and match packages, some written in BH and some written in BSV. Almost all designs from 2005 onwards were done in BSV (with one major exception done in BH), but since open-sourcing in 2020, BH is once again seeing increased use.

From about 2005 to 2020, *bsc* was a product of Bluespec, Inc., licensed commercially to companies and non-academic institutions, with free licenses to universities for teaching and research. During this time, BSV was used for production ASIC designs in several top-ten semiconductor companies, a network devices company, and a major Internet company, and for high-level architectural modeling in a few top-ten computer companies. Bluespec, Inc. itself does all its RISC-V CPU and system designs in BSV (many open-sourced on GitHub). Amongst universities, MIT (USA), University of Cambridge (UK) and IIT Madras (India) have been and continue to be leading users.

Bluespec, Inc. has always considered the language to be fully open (i.e., anyone can freely build their own language implementation); the only proprietary artefact was Bluespec, Inc.'s own implementation, the *bsc* compiler. In 2010, at an IEEE SystemVerilog Standards Committee planning meeting, Bluespec, Inc. offered to donate the whole of the BSV language definition for incorporation into the SystemVerilog standard, but this offer did not garner enough votes for acceptance.

In 2020, Bluespec, Inc. released the *bsc* compiler (and related proprietary artefacts: Bluesim, Bluespec Development Workstation, etc.) in fully open-source form. All the source codes etc. are now hosted in open GitHub repositories, and there are now many contributors from diverse locations worldwide.

Index

`(..)` (exporting member names), 21
`*/` (close nested comment), 13
`.`, *see* structs, member selection
`/*` (open block comment), 13
`//` (one-line comment), 13
`<=` (`Reg` assignment), 70
`?` (don't-care expression), 17, 80
`[]` (bit/part select from bit array), 82
`$bitstoreal` (`Real` system function), 120
`$display`, 114
`$displayb`, 114
`$displayh`, 114
`$displayo`, 114
`$dumpoff`, 119
`$dumpon`, 119
`$dumpvars`, 119
`$fclose`, 116
`$fdisplay`, 117
`$fdisplayb`, 117
`$fdisplayh`, 117
`$fdisplayo`, 117
`$fflush`, 119
`$fgetc`, 118
`$finish`, 119
`$fopen`, 116
`$fwrite`, 117
`$fwriteb`, 117
`$fwriteh`, 117
`$fwriteo`, 117
`$realtobits` (`Real` system function), 120
`$sformat`, 118
`$sformatAV`, 118
`$stime`, 120
`$stop`, 119
`$swrite`, 118
`$swriteAV`, 118
`$swriteb`, 118
`$swritebAV`, 118
`$swriteh`, 118
`$swritehAV`, 118
`$swriteo`, 118
`$swriteoAV`, 118
`$test$plusargs`, 120
`$time`, 120
`$ungetc`, 118
`$write`, 114
`$writeb`, 114
`$writeh`, 114
`$writeo`, 114
`_read` (`Reg` interface method), 112
`_write` (`Reg` interface method), 112
`{}` (concatenation of bit arrays), 82
$ (character in identifiers), 14
_ (character in identifiers), 14
`, *see* compiler directives

actions
    `Action` (type), 84
    `action` (keyword), 84
    combining, 84
`ActionValue` (type), 85
`Add` (type provisos), 25
Alias, 53
`always_enabled` (attribute), 123
`always_ready` (attribute), 123
`ancestor`(BVI import statement), 152
`AOF` (clock ancestor attribute), 135
application
    of functions to arguments, 87
    of methods to arguments, 87
`Arith` (type class), 25
    UInt, Int type instances, 111
array
    anonymous type, 67
arrays
    update, 69
`asIfc` (interface pseudo-function), 113
`asReg` (`Reg` function), 113
assignment statements
    pattern matching in, 99
attribute
    always_enabled, 123
    always_ready, 123
    clock_ancestors=, 135
    clock_family=, 135
    clock_prefix, 133
    clocked_by=, 137
    conflict_free, 130
    default_clock_gate=, 134
    default_clock_osc=, 134
    default_reset=, 134
    defualt_gate_inhigh=, 134
    defualt_gate_unused=, 134
    descending_urgency, 127
    doc=, 138
    enable=, 122
    execution_order, 129
    fire_when_enabled, 125
    gate=, 136
    gate_default_clock, 134

gate_inhigh, 136
gate_input_clocks=, 134
gate_prefix, 133
gate_unused, 136
mutually_exclusive, 130
no_implicit_conditions, 126
noinline, 121
nosplit, 131
osc=, 136
parameter=, 138
port=, 123, 138
preempts, 131
prefix=, 123
ready=, 122
reset=, 136
reset_by=, 138
reset_prefix, 133
result=, 123
split, 131
synthesize, 121
attributes, 120
`await` (`StmtFSM` function), 105

`begin` (keyword), 73, 83
begin-end expression blocks, 83
begin-end statement blocks, 73
`bind` (`Monad` class method), 67
`Bit` (type), 110
`bit` (type), 110
`BitExtend` (type class), 25
UInt, Int type instances, 111
`BitReduction` (type class), 25
UInt, Int type instances, 111
`Bits` (type class), 25, 61
`deriving`, 62
representation of data types, 62
UInt, Int type instances, 111
`Bitwise` (type class), 25
UInt, Int type instances, 111
`Bounded` (type class), 25
`deriving`, 64
UInt, Int type instances, 111
BVI import (keyword)
in interfacing to Verilog, 141

`C` (scheduling annotations), 51
`case` (keyword), 73, 96, 97
case expression, 97
case statements
ordinary, 73
pattern matching, 96
casting, type, 88
`CF` (scheduling annotations), 51
`Char` (type), 111
`clear` (`FIFOF` interface method), 114
`clear` (`FIFO` interface method), 113
`clock_ancestors=` (attribute), 135
`clock_family=` (attribute), 135
`clock_prefix=` (attribute), 133
`clocked_by`, 34
`clocked_by=`(attribute), 137
comment
block, 13
one-line, 13
compiler directives, 18
conditional expressions, 81
pattern matching in, 99
conditional statements, 73
`conflict_free` (attribute), 130
context, *see* provisos
context too weak (overloading resolution), 58

`default` (keyword), 74, 96
`default_clock`(BVI import statement), 147
`default_clock_gate=` (attribute), 134
`default_clock_osc=` (attribute), 134
`default_gate_inhigh` (attribute), 134
`default_gate_unused` (attribute), 134
`default_reset`(BVI import statement), 150
`default_reset=` (attribute), 134
`` `define `` (compiler directive), 18
`delay` (`StmtFSM` function), 105
`deq` (`FIFOF` interface method), 114
`deq` (`FIFO` interface method), 113
`deriving`
`Bits`, 62
`Bounded`, 64
`Eq`, 64
`FShow`, 64
brief description, 27
for isomorphic types, 66
`descending_urgency` (attribute), 127
`Div` (type provisos), 25
`doc=` (attribute), 138
documentation attributes, 138
don't-care expression, *see* ?
`dumpoff`, 119
`dumpon`, 119
`dumpvars`, 119

`` `else `` (compiler directive), 19
`else` (keyword), 73
`` `elsif `` (compiler directive), 19
`Empty` (interface), 31
`enable=` (attribute), 122
`end` (keyword), 73, 83
`` `endif `` (compiler directive), 19
`endpackage` (keyword), 20

`enq` (`FIFOF` interface method), 114
`enq` (`FIFO` interface method), 113
`enum`, 53
enumerations, 53
`Eq` (type class), 25
    `deriving`, 64
    UInt, Int type instances, 111
`execution_order` (attribute), 129
`export` (keyword), 20
export, identifiers from a package, 21

`FIFO` (interface type), 113
`FIFOF` (interface type), 114
finite state machines, 100
`fire_when_enabled` (attribute), 125
`first` (`FIFOF` interface method), 114
`first` (`FIFO` interface method), 113
`fromInteger` (`Literal` class method), 16
`fromSizedInteger` (`SizedLiteral` class method), 15
`FShow` (type class)
    `deriving`, 64
FSMs, 100
function calls, 87
function definitions, 76

`gate=` (attribute), 136
`gate_default_clock` (attribute), 134
`gate_inhigh` (attribute), 136
`gate_input_clocks=` (attribute), 134
`gate_prefix=` (attribute), 133
`gate_unused` (attribute), 136
generated clock port renaming, 133
grammar, 10

higher order functions, 78

Identifier (grammar terminal), 14
identifier (grammar terminal), 14
identifiers, 14
    case sensitivity, 14
    export from a package, 21
    import into a package, 21
    qualified, 22
    static scoping, 21
    with $ as first letter, 14
`if` (keyword), 73
    in method implicit conditions, 36
if statements, 73
    pattern matching in, 98
if-else statements, 73
`‘ifdef` (compiler directive), 19
`‘ifndef` (compiler directive), 19
implicit conditions, 36
    on interface methods, 36
`import` (keyword), 20
`import "BDPI"` (keyword), 156
`import "BVI"` (keyword), 141
import, identifiers into a package, 21
`‘include` (compiler directive), 18
infix operators
    associativity, 81
    precedence, 81
    predefined, 81
`inout`(BVI import statement), 155
`input_clock`(BVI import statement), 146
`input_reset`(BVI import statement), 149
`instance`, 60
instance of type class (overloading group), 57, 60
instantiation (module), 34
`Int` (type), 111
`int` (type), 111
`Integer` (type), 111
Integer literals, 14
interface
    expression, 91
    instantiation, 34
`interface` (BVI import statement), 154
`interface` (keyword)
    in interface declarations, 29
    in interface expressions, 91
interfaces, 29
    definition of, 27
`Invalid`
    tagged union member of`Maybe` type, 57

let, 69
`‘line` (compiler directive), 18
`Literal` (type class), 25
    UInt, Int type instances, 111
Literals
    Integer, 14
    Real, 16
    String, 17
`Log` (type provisos), 25
loop statements
    statically unrolled, 75
    temporal, in FSMs, 102

macro invocation (compiler directive), 19
`match` (keyword), 99
`Max` (type provisos), 25
`Maybe` (type), 57
meta notation, *see* grammar
`method`(BVI import statement), 144
method calls, 87
methods
    of an interface, 29

pattern matching in, 99
mkAutoFSM, 104
mkFIFO (module), 113
mkFIFOF (module), 114
mkFSM, 104
mkFSMServer, 109
mkFSMWithPred, 104
mkOnce, 104
mkReg (module), 112
mkRegU (module), 112
mkSizedFIFO (module), 113
mkSizedFIFOF (module), 114
module
definition of, 32
instantiation, 34
modules
definition of, 27
module (keyword), 32
Monad (type class), 67
Mul (type provisos), 25
mutually_exclusive (attribute), 130

no_implicit_conditions (attribute), 126
noAction (empty action), 84
noinline (attribute), 121
nosplit (attribute), 131
NumAlias, 53

operators
infix, 81
prefix, 81
Ord (type class), 25, 57, 58
UInt, Int type instances, 111
osc= (attribute), 136
output_clock(BVI import statement), 148
output_reset(BVI import statement), 151
overloading groups, *see* type classes
overloading, of types, 57

pack (Bits type class overloaded function), 61
package, 20
package (keyword), 20
parameter, 32
parameter(BVI import statement), 143
parameter= (attribute), 138
path(BVI import statement), 153
pattern matching, 94
error, 97
in assignment statements, 99
in case expressions, 97
in case statements, 96
in conditional expressions, 99
in if statements, 98
in methods, 99
in rules, 99
patterns, 94
polymorphism, 24
port(BVI import statement), 145
port= (attribute), 123, 138
preempts (attribute), 131
prefix= (attribute), 123
Prelude, *see* Standard Prelude, *see* Standard Prelude
provisos, 58
brief description, 24

ready= (attribute), 122
Real literals, 16
records, *see* struct
Reg (type), 112
register assignment, 70
array element, 71
partial, 71
register writes, 70
clear, 100
reset= (attribute), 136
reset_by, 34
reset_by=(attribute), 138
reset_prefix= (attribute), 133
‘resetall (compiler directive), 19
result= (attribute), 123
return (Monad class method), 67
rules, 40
expression, 93
pattern matching in, 99
Rules (type), 93

SA (scheduling annotations), 51
same_family(BVI import statement), 152
SAR (scheduling annotations), 51
SB (scheduling annotations), 51
SBR (scheduling annotations), 51
schedule(BVI import statement), 152
scheduling annotations, 51
size types, 24
type classes for constraints, 25
SizeOf (pseudo-function on types), 62
split (Bit function), 110
split (attribute), 131
Standard Prelude, 9, 22, 59, 84, 110, 111
start, 100
static elaboration, 13, 46, 47
StmtFSM
await, 105
callServer, 109
delay, 105
mkAutoFSM, 104
mkFSMServer, 109
mkFSMwithPred, 104

mkFSM, 104
mkOnce, 104
StmtFSM (package), 100
StrAlias, 53
String (type), 111
String literals, 17
string types, 24
stringOf (pseudo-function of string types), 27
struct
type definition, 54
struct, 54
structs
member selection, 89
update, 69
subinterfaces
declaration of, 31
definition of, 38
synthesize
modules, 43
synthesize (attribute), 121
system functions, 114
$bitstoreal, 120
$realtobits, 120
$stime, 120
$test$plusargs, 120
$time, 120
system tasks, 114
$display, 114
$displayb, 114
$displayh, 114
$displayo, 114
$dumpoff, 119
$dumpon, 119
$dumpvars, 119
$fclose, 116
$fdisplay, 117
$fdisplayb, 117
$fdisplayh, 117
$fdisplayo, 117
$fflush, 119
$fgetc, 118
$finish, 119
$fopen, 116
$fwrite, 117
$fwriteb, 117
$fwriteh, 117
$fwriteo, 117
$sformat, 118
$sformatAV, 118
$stop, 119
$swrite, 118
$swriteAV, 118
$swriteb, 118
$swritebAV, 118
$swriteh, 118
$swritehAV, 118
$swriteo, 118
$swriteoAV, 118
$ungetc, 118
$write, 114
$writeb, 114
$writeh, 114
$writeo, 114

tagged, *see* union
tagged union
member selection, *see* pattern matching
member selection using dot notation, 90
type definition, 54
update, 69
tuples
expressions, 112
patterns, 112
selecting components, 112
type definition, 111
type assertions
static, 88
type casting, 88
type classes, 57
Add, 25
Arith, 25
BitExtend, 25
BitReduction, 25
Bits, 25, 45, 61, 62, 121
Bitwise, 25
Bounded, 25, 64
Div, 25
Eq, 25, 64
FShow, 64, 116
Literal, 25
Log, 25
Max, 25
Monad, 67
Mul, 25
Ord, 25, 57
instance, 60
typeclass (declaring new), 58
instance declaration, 60
type declaration, 22
type variables, 24
typedef (keyword), 51
types, 22
parameterized, 24
polymorphic, 24

UInt (type), 111
‘undef (compiler directive), 19
underscore, *see* _

`union`, 54
union tagged
    type definition, 54
`unpack` (`Bits` type class overloaded function), 61

`Valid`
    tagged union member of`Maybe` type, 57
Value Change Dump, 119
`valueOf` (pseudo-function of size types), 26
variable assignment, 68
variable declaration, 67
variable initialization, 67
variables, 67
VCD, 118, 119
`void` (type, in tagged unions), 55