B-Lang

Bluespec Compiler (BSC) Libraries Reference Guide

Revision: 17 February 2024

Trademarks and copyrights

Contents

Table of Contents 3

1 Introduction 8

2 The Standard Prelude package 9

2.1 Type classes . . . 9

2.1.1 Bits . . . 9

2.1.2 Eq . . . 10

2.1.3 Literal . . . 11

2.1.4 RealLiteral . . . 11

2.1.5 SizedLiteral . . . 12

2.1.6 Arith . . . 12

2.1.7 Ord . . . 14

2.1.8 Bounded . . . 16

2.1.9 Bitwise . . . 17

2.1.10 BitReduction . . . 18

2.1.11 BitExtend . . . 20

2.1.12 SaturatingArith . . . 21

2.1.13 Alias and NumAlias . . . 22

2.1.14 FShow . . . 22

2.1.15 StringLiteral . . . 24

2.2 Data Types . . . 24

2.2.1 Bit . . . 24

2.2.2 UInt . . . 26

2.2.3 Int . . . 26

2.2.4 Integer . . . 26

2.2.5 Bool . . . 27

2.2.6 Real . . . 28

2.2.7 String . . . 29

2.2.8 Char . . . 31

2.2.9 Fmt . . . 33

2.2.10 Void . . . 34

2.2.11 Maybe . . . 34

2.2.12 Tuples . . . 35

2.2.13 Array . . . 37

2.2.14 Ordering . . . 38

2.2.15 File . . . 38
2.2.16 Clock . . . 39
2.2.17 Reset . . . 39
2.2.18 Inout . . . 39
2.2.19 Action/ActionValue . . . 41
2.2.20 Rules . . . 42
2.3 Operations on Numeric Types . . . 43
2.3.1 Size Relationship/Provisos . . . 43
2.3.2 Size Relationship Type Functions . . . 44
2.3.3 valueOf and SizeOf pseudo-functions . . . 44
2.4 Registers and Wires . . . 45
2.4.1 Reg . . . 45
2.4.2 CReg . . . 47
2.4.3 RWire . . . 50
2.4.4 Wire . . . 51
2.4.5 BypassWire . . . 53
2.4.6 DWire . . . 53
2.4.7 PulseWire . . . 54
2.4.8 ReadOnly . . . 56
2.4.9 WriteOnly . . . 57
2.5 Miscellaneous Functions . . . 58
2.5.1 Compile-time Messages . . . 58
2.5.2 Arithmetic Functions . . . 59
2.5.3 Operations on Functions . . . 60
2.5.4 Bit Functions . . . 61
2.5.5 Integer Functions . . . 62
2.5.6 Control Flow Function . . . 62
2.6 Environment Values . . . 63
2.7 Compile-time IO . . . 64
2.8 Generics . . . 67
2.8.1 The `Generic` type class . . . 67
2.8.2 Representation types . . . 68
2.8.3 Metadata types . . . 69
2.8.4 Defining generic instances . . . 71

3 Standard Libraries **73**

3.1 Storage Structures 73

3.1.1 Register File 73

3.1.2 ConfigReg 76

3.1.3 DReg 77

3.1.4 RevertingVirtualReg 77

3.1.5 BRAM 79

3.1.6 BRAMCore 85

3.2 FIFOs 90

3.2.1 FIFO Overview 90

3.2.2 FIFO and FIFOF packages 90

3.2.3 FIFOLevel 98

3.2.4 BRAMFIFO 105

3.2.5 SpecialFIFOs 107

3.2.6 AlignedFIFOs 109

3.2.7 Gearbox 112

3.2.8 MIMO 114

3.3 Aggregation: Vectors 116

3.3.1 Creating and Generating Vectors 118

3.3.2 Extracting Elements and Sub-Vectors 120

3.3.3 Vector to Vector Functions 123

3.3.4 Tests on Vectors 126

3.3.5 Bit-Vector Functions 129

3.3.6 Functions on Vectors of Registers 129

3.3.7 Combining Vectors with Zip 129

3.3.8 Mapping Functions over Vectors 131

3.3.9 ZipWith Functions 131

3.3.10 Fold Functions 133

3.3.11 Scan Functions 135

3.3.12 Monadic Operations 137

3.3.13 Converting to and from Vectors 141

3.3.14 ListN 141

3.4 Aggregation: Lists 142

3.4.1 Creating and Generating Lists 143

3.4.2 Extracting Elements and Sub-Lists 144

3.4.3 List to List Functions 148

3.4.4 Tests on Lists 150

3.4.5 Combining Lists with Zip Functions . . . 151
3.4.6 Mapping Functions over Lists . . . 152
3.4.7 ZipWith Functions . . . 153
3.4.8 Fold Functions . . . 154
3.4.9 Scan Functions . . . 157
3.4.10 Monadic Operations . . . 159
3.5 Math . . . 160
3.5.1 Real . . . 160
3.5.2 OInt . . . 164
3.5.3 Complex . . . 165
3.5.4 FixedPoint . . . 168
3.5.5 NumberTypes . . . 175
3.6 FSM . . . 177
3.6.1 StmtFSM . . . 177
3.7 Connectivity . . . 187
3.7.1 GetPut . . . 187
3.7.2 Connectable . . . 192
3.7.3 ClientServer . . . 194
3.7.4 Memory . . . 196
3.7.5 CGetPut . . . 197
3.7.6 CommitIfc . . . 199
3.8 Utilities . . . 203
3.8.1 LFSR . . . 203
3.8.2 Randomizable . . . 205
3.8.3 Arbiter . . . 207
3.8.4 Cntrs . . . 209
3.8.5 GrayCounter . . . 211
3.8.6 Gray . . . 212
3.8.7 CompletionBuffer . . . 213
3.8.8 UniqueWrappers . . . 216
3.8.9 DefaultValue . . . 220
3.8.10 TieOff . . . 222
3.8.11 Assert . . . 222
3.8.12 Probe . . . 223
3.8.13 Reserved . . . 224
3.8.14 TriState . . . 226
3.8.15 ZBus . . . 227

3.8.16 CRC 230

3.8.17 OVLAssertions 232

3.8.18 Printf 245

3.8.19 BuildVector 246

3.9 Multiple Clock Domains and Clock Generators 247

3.9.1 Clock Generators and Clock Manipulation 249

3.9.2 Clock Multiplexing 253

3.9.3 Clock Division 255

3.9.4 Bit Synchronizers 258

3.9.5 Pulse Synchronizers 262

3.9.6 Word Synchronizers 265

3.9.7 FIFO Synchronizers 266

3.9.8 Asynchronous RAMs 269

3.9.9 Null Crossing Primitives 270

3.9.10 Reset Synchronization and Generation 272

3.10 Special Collections 277

3.10.1 ModuleContext 277

3.10.2 ModuleCollect 282

3.10.3 CBus 286

3.10.4 HList 292

3.10.5 UnitAppendList 296

Index **298**

Function and Module by Package **306**

Typeclasses **312**

1 Introduction

TBD

2 The Standard Prelude package

This section describes the type classes, data types, interfaces and functions provided by the `Prelude` package. The standard `Prelude` package is automatically included in all BSV packages. You do not need to take any special action to use any of the features defined in the `Prelude` package.

Section 3 describes BSC's collection of standard libraries. To use any of these libraries in a design you must explicitly import the library package.

2.1 Type classes

A type class groups related functions and operators and allows for instances across the various datatypes which are members of the typeclass. Hence the function names within a type class are *overloaded* across the various type class members.

A `typeclass` declaration creates a type class. An `instance` declaration defines a datatype as belonging to a type class. A datatype may belong to zero or many type classes.

The Prelude package declares the following type classes:

Prelude Type Classes	
`Bits`	Types that can be converted to bit vectors and back.
`Eq`	Types on which equality is defined.
`Literal`	Types which can be created from integer literals.
`RealLiteral`	Types which can be created from real literals.
`Arith`	Types on which arithmetic operations are defined.
`Ord`	Types on which comparison operations are defined.
`Bounded`	Types with a finite range.
`Bitwise`	Types on which bitwise operations are defined.
`BitReduction`	Types on which bitwise operations on a single operand to produce a single bit result are defined.
`BitExtend`	Types on which extend operations are defined.
`SaturatingArith`	Types with functions to describe how overflow and underflow should be handled.
`Alias`	Types which can be used interchangeably.
`NumAlias`	Types which give a new name to a numeric type.
`FShow`	Types which can convert a value to a `Fmt` representation for use with `$display` system tasks.
`StringLiteral`	Types which can be created around strings.

2.1.1 Bits

`Bits` defines the class of types that can be converted to bit vectors and back. Membership in this class is required for a data type to be stored in a state, such as a Register or a FIFO, or to be used at a synthesized module boundary. Often instance of this class can be automatically derived using the `deriving` statement.

```
typeclass Bits #(type a, numeric type n);
    function Bit#(n) pack(a x);
    function a unpack(Bit#(n) x);
endtypeclass
```

Note: the numeric keyword is not required

The functions `pack` and `unpack` are provided to convert elements to `Bit#()` and to convert `Bit#()` elements to another datatype.

Bits Functions	
pack	Converts element a of datatype data_t to a element of datatype Bit#() of size_a.
	`function Bit#(size_a) pack(data_t a);`
unpack	Converts an element a of datatype Bit#() and size_a into an element with of element type data_t.
	`function data_t unpack(Bit#(size_a) a);`

Examples

```
Reg#(int) cycle <- mkReg (0);
..
rule r;
  ...
   if (pack(cycle)[0] == 0) a = a + 1;
   else                     a = a + 2;
   if (pack(cycle)[1:0] == 3) a = a + 3;

 Int#(10) src_step    = unpack(config6[9:0]);
 Bool     src_rdy_en  = unpack(config6[16]);
```

2.1.2 Eq

Eq defines the class of types whose values can be compared for equality. Instances of the Eq class are often automatically derived using the deriving statement.

```
typeclass Eq #(type data_t);
    function Bool \== (data_t x, data_t y);
    function Bool \/= (data_t x, data_t y);
endtypeclass
```

The equality functions == and != are Boolean functions which return a value of True if the equality condition is met. When defining an instance of an Eq typeclass, the \== and \/= notations must be used. If using or referring to the functions, the standard Verilog operators == and != may be used.

Eq Functions	
==	Returns True if x is equal to y.
	`function Bool \== (data_t x, data_t y,);`
/=	Returns True if x is not equal to y.
	`function Bool \/= (data_t x, data_t y,);`

Examples

```
return (pack(i) & 3) == 0;

if (a != maxInt)
```

2.1.3 Literal

Literal defines the class of types which can be created from integer literals.

```
typeclass Literal #(type data_t);
    function data_t fromInteger(Integer x);
    function Bool   inLiteralRange(data_t target, Integer x);
endtypeclass
```

The fromInteger function converts an Integer into an element of datatype data_t. Whenever you write an integer literal in BSV(such as "0" or "1"), there is an implied fromInteger applied to it, which turns the literal into the type you are using it as (such as Int, UInt, Bit, etc.). By defining an instance of Literal for your own datatypes, you can create values from literals just as for these predefined types.

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

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

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

Examples

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

function Bool inLiteralRange(RegAddress a, Integer i);
   return(i >= 0 && i < 83);
endfunction
```

2.1.4 RealLiteral

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

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

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

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

Examples

```
FixedPoint#(is, fs) f = fromReal(n);  //n is a Real number
```

2.1.5 SizedLiteral

`SizedLiteral` defines the class of types which can be created from integer literals with a specified size.

```
typeclass SizedLiteral #(type data_t, type size_t)
  dependencies (data_t determines size_t);
    function data_t fromSizedInteger(Bit#(size_t);
endtypeclass
```

The `fromSizedInteger` function converts a literal of type `Bit#(size_t)` into an element of datatype `data_t`. Whenever you write a sized literal like `1'b0`, there is an implied `fromSizedInteger` which turns the literal into the type you are using it as, with the defined size. Instances are defined for the types `Bit`, `UInt`, and `Int`.

SizedLiteral Functions	
fromSizedInteger	Converts an element of Bit#(size_t) into an element of data type data_t
	`function data_t fromSizedInteger(Bit#(size_t));`

2.1.6 Arith

`Arith` defines the class of types on which arithmetic operations are defined.

```
typeclass Arith #(type data_t)
  provisos (Literal#(data_t));
    function data_t \+ (data_t x, data_t y);
    function data_t \- (data_t x, data_t y);
    function data_t negate (data_t x);
    function data_t \* (data_t x, data_t y);
    function data_t \/ (data_t x, data_t y);
    function data_t \% (data_t x, data_t y);
    function data_t abs (data_t x);
    function data_t signum (data_t x);
    function data_t \** (data_t x, data_t y);
```

```
    function data_t exp_e (data_t x);
    function data_t log (data_t x);
    function data_t logb (data_t b, data_t x);
    function data_t log2 (data_t x);
    function data_t log10 (data_t x);
endtypeclass
```

The `Arith` functions provide arithmetic operations. For the arithmetic symbols, when defining an instance of the `Arith` typeclass, the escaped operator names must be used as shown in the tables below. The `negate` name may be used instead of the operator for negation. If using or referring to these functions, the standard (non-escaped) Verilog operators can be used.

`Arith` Functions	
`+`	Element `x` is added to element `y`.
	`function data_t \+ (data_t x, data_t y);`

`-`	Element `y` is subtracted from element `x`.
	`function data_t \- (data_t x, data_t y);`

`negate`
 	
`-`
	Change the sign of the number. When using the function the Verilog negate operator, `-`, may be used.

	`function data_t negate (data_t x);`

`*`	Element `x` is multiplied by `y`.
	`function data_t \* (data_t x, data_t y);`

`/`	Element `x` is divided by `y`. The definition depends on the type - many types truncate the remainder . Note: may not be synthesizable with downstream tools.
	`function data_t \/ (data_t x, data_t y);`

`%`	Returns the remainder of x/y. Obeys the identity $((x/y) * y) + (x\%y) = x$.
	`function data_t \% (data_t x, data_t y);`

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

`abs`	Returns the absolute value of `x`.
	`function data_t abs (data_t x);`

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

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

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

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

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

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

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

Examples

```
real u = log(1);
real x = 128.0;
real y = log2(x);
real z = 100.0;
real v = log10(z);
real w = logb(3,9.0);
real a = -x;
real b = abs(x);
```

2.1.7 Ord

`Ord` defines the class of types for which an *order* is defined, allowing comparison operations. A complete definition of an instance of `Ord` requires defining either `<=` or `compare`.

```
typeclass Ord #(type data_t);
    function Bool \<  (data_t x, data_t y);
    function Bool \<= (data_t x, data_t y);
    function Bool \>  (data_t x, data_t y);
    function Bool \>= (data_t x, data_t y);
    function Ordering compare(data_t x, data_t y);
    function data_t min(data_t x, data_t y);
    function data_t max(data_t x, data_t y);
endtypeclass
```

The functions `<`, `<=`, `>`, and `>=` are Boolean functions which return a value of `True` if the comparison condition is met.

`Ord` Functions	
`<`	Returns `True` if `x` is less than `y`. `function Bool \<  (data_t x, data_t y);`

`<=`	Returns `True` if `x` is less than or equal to `y`. `function Bool \<=  (data_t x, data_t y);`

`>`	Returns `True` if `x` is greater than `y`. `function Bool \>  (data_t x, data_t y);`

`>=`	Returns `True` if `x` is greater than or equal to `y`. `function Bool \>=  (data_t x, data_t y);`

The function `compare` returns a value of the `Ordering` (Section 2.2.14) data type (`LT`, `GT`, or `EQ`).

`compare`	Returns the `Ordering` value describing the relationship of `x` to `y`. `function Ordering compare (data_t x, data_t y);`

The functions `min` and `max` return a value of datatype `data_t` which is either the minimum or maximum of the two values, depending on the function.

`min`	Returns the minimum of the values `x` and `y`. `function data_t min (data_t x, data_t y);`

`max`	Returns the maximum of the values `x` and `y`. `function data_t max (data_t x, data_t y);`

Examples

```
rule r1 (x <= y);

rule r2 (x > y);

function Ordering onKey(Record r1, Record r2);
   return compare(r1.key,r2.key);
endfunction
...
   List#(Record) sorted_rs = sortBy(onKey,rs);
   List#(List#(Record)) grouped_rs = groupBy(equiv,sorted_rs);

let read_count = min(reads_remaining, 16);
```

2.1.8 Bounded

Bounded defines the class of types with a finite range and provides functions to define the range.

```
typeclass Bounded #(type data_t);
    data_t minBound;
    data_t maxBound;
endtypeclass
```

The Bounded functions minBound and maxBound define the minimum and maximum values for the type data_t. Instances of the Bounded class are often automatically derived using the deriving statement.

Bounded Functions	
minBound	The minimum value the type data_t can have. `data_t minBound;`
maxBound	The maximum value the type data_t can have. `data_t maxBound;`

Examples

```
module mkGenericRandomizer (Randomize#(a))
   provisos (Bits#(a, sa), Bounded#(a));

typedef struct {
    Bit#(2) red;
    Bit#(1) blue;
} RgbColor deriving (Eq, Bits, Bounded);
```

2.1.9 Bitwise

`Bitwise` defines the class of types on which bitwise operations are defined.

```
typeclass Bitwise #(type data_t);
    function data_t \& (data_t x1, data_t x2);
    function data_t \| (data_t x1, data_t x2);
    function data_t \^ (data_t x1, data_t x2);
    function data_t \~^ (data_t x1, data_t x2);
    function data_t \^~ (data_t x1, data_t x2);
    function data_t invert (data_t x1);
    function data_t \<< (data_t x1, x2);
    function data_t \>> (data_t x1, x2);
    function Bit#(1) msb (data_t x);
    function Bit#(1) lsb (data_t x);
endtypeclass
```

The `Bitwise` functions compare two operands bit by bit to calculate a result. That is, the bit in the first operand is compared to its equivalent bit in the second operand to calculate a single bit for the result.

`Bitwise` Functions	
`&`	Performs an *and* operation on each bit in `x1` and `x2` to calculate the result.
	`function data_t \& (data_t x1, data_t x2);`

`|`	Performs an *or* operation on each bit in `x1` and `x2` to calculate the result.
	`function data_t \\| (data_t x1, data_t x2);`

`^`	Performs an *exclusive or* operation on each bit in `x1` and `x2` to calculate the result.
	`function data_t \^ (data_t x1, data_t x2);`

`~^` `^~`	Performs an *exclusive nor* operation on each bit in `x1` and `x2` to calculate the result.
	`function data_t \~^ (data_t x1, data_t x2);` `function data_t \^~ (data_t x1, data_t x2);`

`~` `invert`	Performs a *unary negation* operation on each bit in `x1`. When using this function, the corresponding Verilog operator, `~`, may be used.
	`function data_t invert (data_t x1);`

The `<<` and `>>` operators perform left and right shift operations. Whether the shift is an arithmetic shift (`Int`) or a logical shift (`Bit`, `UInt`) is dependent on how the type is defined.

`<<`	Performs a *left shift* operation of `x1` by the number of bit positions given by `x2`. `x2` must be of an acceptable index type (`Integer`, `Bit#(n)`, `Int#(n)` or `UInt#(n)`).
	`function data_t \<< (data_t x1, x2);`

`>>`	Performs a *right shift* operation of `x1` by the number of bit positions given by `x2`. `x2` must be of an acceptable index type (`Integer`, `Bit#(n)`, `Int#(n)` or `UInt#(n)`).
	`function data_t \>> (data_t x1, x2);`

The functions `msb` and `lsb` operate on a single argument.

`msb`	Returns the value of the most significant bit of `x`. Returns 0 if width of data_t is 0.
	`function Bit#(1) msb (data_t x);`

`lsb`	Returns the value of the least significant bit of `x`. Returns 0 if width of data_t is 0.
	`function Bit#(1) lsb (data_t x);`

Examples

```
function Value computeOp(AOp aop, Value v1, Value v2) ;
    case (aop) matches
        Aand : return v1 & v2;
        Anor : return invert(v1 | v2);
        Aor : return v1 | v2;
        Axor : return v1 ^ v2;
        Asll : return v1 << vToNat(v2);
        Asrl : return v1 >> vToNat(v2);
    endcase
endfunction: computeOp

Bit#(3) msb = read_counter [5:3];
Bit#(3) lsb = read_counter [2:0];
read_counter <= (msb == 3'b111) ? {msb+1,lsb+1} : {msb+1,lsb};
```

2.1.10 BitReduction

`BitReduction` defines the class of types on which the Verilog bit reduction operations are defined.

```
typeclass BitReduction #(type x, numeric type n)
    function x#(1) reduceAnd (x#(n) d);
    function x#(1) reduceOr (x#(n) d);
    function x#(1) reduceXor (x#(n) d);
    function x#(1) reduceNand (x#(n) d);
    function x#(1) reduceNor (x#(n) d);
    function x#(1) reduceXnor (x#(n) d);
endtypeclass
```

Note: the numeric keyword is not required

The `BitReduction` functions take a sized type and reduce it to one element. The most common example is to operate on a `Bit#()` to produce a single bit result. The first step of the operation applies the operator between the first bit of the operand and the second bit of the operand to produce a result. The function then applies the operator between the result and the next bit of the operand, until the final bit is processed.

Typically the bit reduction operators will be accessed through their Verilog operators. When defining a new instance of the `BitReduction` type class the BSV names must be used. The table below lists both values. For example, the BSV bit reduction *and* operator is `reduceAnd` and the corresponding Verilog operator is `&`.

`BitReduction` Functions	

`reduceAnd`
 	
`&`
	Performs an *and* bit reduction operation between the elements of `d` to calculate the result.
	
	`function x#(1) reduceAnd (x#(n) d);`

`reduceOr`
 	
`|`
	Performs an *or* bit reduction operation between the elements of `d` to calculate the result.
	
	`function x#(1) reduceOr (x#(n) d);`

`reduceXor`
 	
`^`
	Performs an *xor* bit reduction operation between the elements of `d` to calculate the result.
	
	`function x#(1) reduceXor (x#(n) d);`

`reduceNand`
 	
`~&`
	Performs an *nand* bit reduction operation between the elements of `d` to calculate the result.
	
	`function x#(1) reduceNand (x#(n) d);`

`reduceNor`
 	
`~|`
	Performs an *nor* bit reduction operation between the elements of `d` to calculate the result.
	
	`function x#(1) reduceNor (x#(n) d);`

`reduceXnor`
 	
`~^`
`^~`
	Performs an *xnor* bit reduction operation between the elements of `d` to calculate the result.
	
	`function x#(1) reduceXnor (x#(n) d);`

2.1.11 BitExtend

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

```
typeclass BitExtend #(numeric type m, numeric type n, type x);  // n > m
    function x#(n) extend (x#(m) d);
    function x#(n) zeroExtend (x#(m) d);
    function x#(n) signExtend (x#(m) d);
    function x#(m) truncate (x#(n) d);
endtypeclass
```

The `BitExtend` operations take as input of one size and changes it to an input of another size, as described in the tables below. It is recommended that `extend` be used in place of `zeroExtend` or `signExtend`, as it will automatically perform the correct operation based on the data type of the argument.

`BitExtend` Functions	

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

`zeroExtend`	Use of `extend` instead is recommended. Adds extra zero bits to the MSB of argument `d` of size `m` to make the datatype size `n`.
	
	`function x#(n) zeroExtend (x#(m) d)`
	`   provisos (Add#(k, m, n));`

`signExtend`	Use of `extend` instead is recommended. Adds extra sign bits to the MSB of argument `d` of size `m` to make the datatype size `n` by replicating the sign bit.
	
	`function x#(n) signExtend (x#(m) d)`
	`   provisos (Add#(k, m, n));`

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

Examples

```
UInt#(TAdd#(1,TLog#(n))) zz = extend(xx) + extend(yy);
Bit#(n) v1 = zeroExtend(v);
Int#(4) i_index = signExtend(i) + 4;
Bit#(32) upp = truncate(din);
r <= zeroExtend(c + truncate(r))
```

2.1.12 SaturatingArith

The `SaturatingArith` typeclass contains modified addition and subtraction functions which saturate to the values defined by maxBound or minBound when the operation would otherwise overflow or wrap-around.

There are 4 types of saturation modes which determine how an overflow or underflow should be handled, as defined by the `SaturationMode` type.

Saturation Modes	
Enum Value	Description
`Sat_Wrap`	Ignore overflow and underflow, just wrap around
`Sat_Bound`	On overflow or underflow result becomes maxBound or minBound
`Sat_Zero`	On overflow or underflow result becomes 0
`Sat_Symmetric`	On overflow or underflow result becomes maxBound or (minBound+1)

```
typedef enum { Sat_Wrap
              ,Sat_Bound
              ,Sat_Zero
              ,Sat_Symmetric
   } SaturationMode deriving (Bits, Eq);

   typeclass SaturatingArith#( type t);
      function t satPlus (SaturationMode mode, t x, t y);
      function t satMinus (SaturationMode mode, t x, t y);
      function t boundedPlus  (t x, t y) = satPlus (Sat_Bound, x, y);
      function t boundedMinus (t x, t y) = satMinus(Sat_Bound, x, y);
   endtypeclass
```

Instances of the `SaturatingArith` class are defined for `Int`, `UInt`, `Complex`, and `FixedPoint`.

`satPlus`	Modified plus function which saturates when the operation would otherwise overflow or wrap-around. The saturation value (`maxBound`, wrap, or 0) is determined by the value of `mode`, the `SaturationMode`.
	`function t satPlus (SaturationMode mode, t x, t y);`

`satMinus`	Modified minus function which saturates when the operation would otherwise overflow or wrap-around. The saturation value (`minBound`, wrap, `minBound +1`, or 0) is determined by the value of `mode`, the `SaturationMode`.
	`function t satMinus (SaturationMode mode, t x, t y);`

`boundedPlus`	Modified plus function which saturates to `maxBound` when the operation would otherwise overflow or wrap-around. The function is the same as `satPlus` where the `SaturationMode` is `Sat_Bound`.	
	`function t boundedPlus (t x, t y) = satPlus (Sat_Bound, x, y);`	

`boundedMinus`	Modified minus function which saturates to `minBound` when the operation would otherwise overflow or wrap-around. The function is the same as `satMinus` where the `SaturationMode` is `Sat_Bound`.
	`function t boundedMinus (t x, t y) = satMinus(Sat_Bound, x, y);`

Examples

```
Reg#(SaturationMode) smode <- mkReg(Sat_Wrap);

rule okdata (isOk);
   tstCount <= boundedPlus (tstCount, 1);
endrule
```

2.1.13 Alias and NumAlias

`Alias` specifies that two types can be used interchangeably, providing a way to introduce local names for types within a module. They are used in Provisos.

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

`NumAlias` is used to give a new name to a numeric type.

```
typeclass NumAlias#(numeric type a, numeric type b)
   dependencies (a determines b,
                 b determines a);
endtypeclass
```

Examples

```
Alias#(fp, FixedPoint#(i,f));
NumAlias#(TLog#(a,b), logab);
```

2.1.14 FShow

The `FShow` typeclass defines the types to which the function `fshow` can be applied. The function converts a value to an associated `Fmt` representation for use with the `$display` family of system tasks. Instances of the `FShow` class can often be automatically derived using the `deriving` statement.

```
typeclass FShow#(type t);
   function Fmt fshow(t value);
endtypeclass
```

FShow function	
fshow	Returns a Fmt representation when applied to a value `function Fmt fshow(t value);`

Instances of FShow for Prelude data types are defined in the Prelude package. Instances for non-Prelude types are documented in the type package. If an instance of FShow is not already defined for a type you can create your own instance. You can also redefine existing instances as required for your design.

FShow Instances			
Type	Fmt Object	Description	Example
String, Char	value	value of the string	Hello
Bool	True False	Bool values	True False
Int#(n)	n	n in decimal format	-17
UInt#(n)	n	n in decimal format	42
Bit#(n)	'hn	n in hex, prepended with 'h	'h43F2
Maybe#(a)	tagged Valid value tagged Invalid	FShow applied to value	tagged Valid 42 tagged Invalid
Tuple2#(a,b) Tuple3#(a,b,c) Tuple4#(a,b,c,d) ... Tuple8#(a,b,c,d,e,f,g,h)	< a, b> < a, b, c> < a, b, c, d>	FShow applied to each value	< 0, 1> < 0, 1, 2> < 0, 1, 2, 3>

Example

```
typedef enum {READ, WRITE, UNKNOWN}  OpCommand   deriving(Bounded,Bits, Eq, FShow);

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

typedef union tagged {Header  Descriptor;
      Bit#(8) Data;
      } Request deriving(Eq, Bits, Bounded);

// Define FShow instances where definition is different
// than the derived values

instance FShow#(Header);
   function Fmt fshow (Header value);
      return ($format("<HEAD ")
```

```
      +
      fshow(value.command)
      +
      $format(" (%0d)", value.length)
      +
      $format(" A:%h",  value.addr)
      +
      $format(" D:%h>", value.data));
   endfunction
endinstance

instance FShow#(Request);
   function Fmt fshow (Request request);
      case (request) matches
         tagged Descriptor .a:
            return fshow(a);
         tagged Data .a:
            return $format("<DATA %h>", a);
      endcase
   endfunction
endinstance
```

2.1.15 StringLiteral

`StringLiteral` defines the class of types which can be created from strings.

```
typeclass StringLiteral #(type data_t);
    function data_t fromString(String x);
endtypeclass
```

`StringLiteral` Functions	
`fromString`	Converts an element `x` of datatype `String` into an element of data type `data_t`
	`function data_t fromString(String x);`

2.2 Data Types

Every variable and every expression in BSV has a *type*. Prelude defines the data types which are always available. An `instance` declaration defines a data type as belonging to a type class. Each data type may belong to one or more type classes; all functions, modules, and operators declared for the type class are then defined for the data type. A data type does not have to belong to any type classes.

Data type identifiers must always begin with a capital letter. There are three exceptions; `bit`, `int`, and `real`, which are predefined for backwards compatibility.

2.2.1 Bit

To define a value of type Bit:

```
Bit#(type n);
```

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

`Bit` type aliases	
bit	The data type `bit` is defined as a single bit. This is a special case of `Bit`.
	`typedef Bit#(1) bit;`

Examples

```
Bit#(32) a;  // like 'reg [31:] a'
Bit#(1)  b;  // like 'reg a'
bit      c;  // same as Bit#(1) c
```

The `Bit` data type provides functions to concatenate and split bit-vectors.

`Bit` Functions	
{x,y}	Concatenate two bit vectors, `x` of size `n` and `y` of size `m` returning a bit vector of size `k`. The Verilog operator `{ }` is used.
	`function Bit#(k) bitconcat(Bit#(n) x, Bit#(m) y) provisos (Add#(n, m, k));`

split	Split a bit vector into two bit vectors (higher-order bits (n), lower-order bits (m)).
	`function Tuple2 #(Bit#(n), Bit#(m)) split(Bit#(k) x) provisos (Add#(n, m, k));`

Examples

```
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

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

2.2.2 UInt

The `UInt` type is an unsigned fixed width representation of an integer value.

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

Examples

```
UInt#(8)  a = 'h80;
UInt#(12) b = zeroExtend(a); // b => 'h080
UInt#(8)  c = truncate(b);   // c => 'h80
```

2.2.3 Int

The `Int` type is a signed fixed width representation of an integer value.

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

Examples

```
Int#(8)  a = 'h80;
Int#(12) b = signExtend(a);   // b => 'hF80
Int#(8)  c = truncate(d);     // c => 'h80
```

`Int` type aliases	
`int`	The data type `int` is defined as a 32-bit signed integer. This is a special case of `Int`. `typedef Int#(32) int;`

2.2.4 Integer

The `Integer` type is a data type used for integer values and functions. Because `Integer` is not part of the `Bits` typeclass, the `Integer` type is used for static elaboration only; all values must be resolved at compile time.

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

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

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

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

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

The `fromInteger` function, defined in Section 2.1.3, can be used to convert an `Integer` into any type in the `Literal` typeclass.

Examples

```
Int#(32) arr2[16];
for (Integer i=0; i<16; i=i+1)
   arr2[i] = fromInteger(i);

Integer foo = 10;
foo = foo + 1;
foo = foo * 5;
Bit#(16) var1 = fromInteger( foo );
```

2.2.5 Bool

The `Bool` type is defined to have two values, `True` and `False`.

`typedef enum {False, True} Bool;`

Type Classes for `Bool`

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

The `Bool` functions return either a value of `True` or `False`.

Bool Functions

`not` `!`	Returns True if x is false, returns False if x is true.	`function Bool not (Bool x);`

`&&`	Returns True if x *and* y are true, else it returns False.	
	`function Bool \&& (Bool x, Bool y);`	

`||`	Returns True if x *or* y is true, else it returns False.	
	`function Bool \\|| (Bool x, Bool y);`	

Examples

```
Bool  a             // A variable named a with a type of Bool
Reg#(Bool) done     // A register named done with a type of Bool
Vector#(5, Bool)    // A vector of 5 Boolean values

Bool  a = 0;     // ERROR!  You cannot do this
Bool  a = True;  // correct
if (a)           // correct
  ....
if (a == 0)      // ERROR!
  ....

Bool b1 = True;
Bool b2 = True;
Bool b3 = b1 && b2;
```

2.2.6 Real

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

Real numbers are of the form:

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

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

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

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

There are many functions defined for `Real` types, provided in the `Real` package (Section 3.5.1). To use these functions, the `Real` package must be imported.

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

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

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

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

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

Examples

```
Bit#(1) sign1 = 0;
Bit#(52) mantissa1 = 0;
Bit#(11) exp1 = 1024;
Real r1 = $bitstoreal({sign1, exp1, mantissa1});  //r1 = 2.0

Real x = pi;
let m = realToString(x));   // m = 3.141592653589793
```

2.2.7 String

Strings are mostly used in system tasks (such as $display). The String type belongs to the Eq type class; strings can be tested for equality and inequality using the == and != operators. The String type is also part of the Arith class, but only the addition (+) operator is defined. All other Arith operators will produce an error message.

Type Classes for String									
	Bits	Eq	Literal	Arith	Ord	Bounded	Bitwise	String Literal	FShow
String		√		√				√	√

String Functions	
strConcat +	Concatenates two strings (same as the + operator) `function String strConcat(String s1, String s2);`

`stringLength`	Returns the number of characters in a string	
	`function Integer stringLength (String s);`	

`stringSplit`	If the string is empty, returns `Invalid`; otherwise it returns `Valid` with the `Tuple` containing the first character as the head and the rest of the string as the tail.
	`function Maybe#(Tuple#2(Char, String)) stringSplit(String s);`

`stringHead`	Extracts the first character of a string; reports an error if the string is empty.
	`function Char stringHead(String s);`

`stringTail`	Extracts all the characters of a string after the first; reports an error if the string is empty.
	`function String stringTail(String s);`

`stringCons`	Adds a character to the front of a string. This function is the complement of `stringSplit`.
	`function String stringCons(Char c, String s);`

`stringToCharList`	Converts a `String` to a `List` of characters
	`function List#(Char) stringToCharList (String s);`

`charListToString`	Converts a `List` of characters to a `String`
	`function String charListToString (List#(Char) cs);`

`quote`	Add single quotes around a string: `‘str’`
	`function String quote (String s);`

`doubleQuote`	Add double quotes around a string: `"str"`
	`function String doubleQuote (String s);`

Examples

```
String s1 = "This is a test";
$display("first string = ", s1);

// we can use + to concatenate
String s2 = s1 + " of concatenation";
$display("Second string = ", s2);
```

2.2.8 Char

The `Char` data type is used mostly in system tasks (such as `$display`). The `Char` type provides the ability to traverse the characters of a string. The `Char` type belongs to the `Eq` type class; chars can be tested for equality and inequality using the `==` and `!=` operators.

The `Char` type belongs to the `Ord` type class.

Type Classes for `Char`									
	`Bits`	`Eq`	`Literal`	`Arith`	`Ord`	`Bounded`	`Bitwise`	`String Literal`	`FShow`
`Char`		√			√			√	√

`Char` Functions	
`charToString`	Convert a single character to a string
	`function String charToString (Char c);`

`charToInteger`	Convert a character to its ASCII numeric value
	`function Integer charToInteger (Char c);`

`integerToChar`	Convert an ASCII value to its character equivalent, returns an error if the number is out of range
	`function Char integerToChar (Integer n);`

`isSpace`	Determine if a character is whitespace (space, tab `\t`, vertical tab `\v`, newline `\n`, carriage return `\r`, linefeed `\f`)
	`function Bool isSpace (Char c);`

`isLower`	Determine if a character is a lowercase ASCII character (`a - z`)
	`function Bool isLower (Char c);`

`isUpper`	Determine if a character is an uppercase ASCII character (`A - Z`)
	`function Bool isUpper (Char c);`

`isAlpha`	Determine if a character is an ASCII letter, either upper or lowercase
	`function Bool isAlpha (Char c);`

`isDigit`	Determine if a character is an ASCII decimal digit (`0 - 9`)
	`function Bool isDigit (Char c);`

`isAlphaNum`	Determine if a character is an ASCII letter or decminal digit	`function Bool isAlphaNum (Char c);`
`isOctDigit`	Determine if a character is an ASCII octal digit (`0` - `7`)	`function Bool isOctDigit (Char c);`
`isHexDigit`	Determine if a character is an ASCII hexadecimal digit (`0` - `9`, `a` - `f`, or `A` - `F`)	`function Bool isHexDigit (Char c);`
`toUpper`	Convert an ASCII lowercase letter to uppercase; other characters are unchanged	`function Char toUpper (Char c);`
`toLower`	Convert an ASCII uppercase letter to lowercase; other characters are unchanged	`function Char toLower (Char c);`
`digitToInteger`	Convert an ASCII decimal digit to its numeric value (`0` to 0, unlike `charToInteger` which would return the ASCII code 48); returns an error if the character is not a digit.	`function Integer digitToInteger (Char c);`
`digitToBits`	Convert an ASCII decimal digit to its numeric value; returns an error if the character is not a digit. Same as `digitToInteger`, but returns the value as a bit vector; the vector can be any size, but the user will get an error if the size is too small to represent the value	`function Bit#(n) digitToBits (Char c);`
`integerToDigit`	Convert a decimal digit value (0 to 9) to the ASCII character for that digit; returns an error if the value is out of range. This function is the complement of `digitToInteger`	`function Char integerToDigit (Integer d);`
`bitsToDigit`	Convert a Bit type digit value to the ASCII character for that digit; returns an error if the value is out of range. This is the same as `integerToDigit` but for values that are Bit types	`function Char bitsToDigit (Bit#(n) d);`

`hexDigitToInteger`	Convert an ASCII decimal digit to its numeric, including hex characters `a` - `f` and `A` - `F`
	`function Integer hexDigitToInteger (Char c);`

`hexDigitToBits`	Convert an ASCII decimal digit to its numeric, including hex characters `a - f` and `A - F` returning the value as a bit vector. The vector can be any size, but an error will be returned if the size is too small to represent the value.
	`function Bit#(n) hexDigitToBits (Char c);`

`integerToHexDigit`	Convert a hexadecimal digit value (0 to 15) to the ASCII character for that digit; returns an error if the value is out of range. This function is the complement of `hexDigitToInteger`. The function returns lowercase for the letters `a` to `f`; apply the function `toUpper` to get uppercase.
	`function Char integerToHexDigit (Integer d);`

`bitsToHexDigit`	Convert a Bit type hexadecimal digit value to the ASCII character for that digit, returns an error if the value is out of range. The function returns lowercase for the letters `a` to `f`; apply the function `toUpper` to get uppercase.
	`function Char bitsToHexDigit (Bit#(n) d);`

2.2.9 Fmt

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

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

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

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

Examples

```
Reg#(Bit#(8)) count <- mkReg(0);
Fmt f = $format("(%0d)", count + 1);
$display(" XYZ ", f, " ", $format("(%0d) ", count));

\\value displayed:  XYZ (6) (5)
```

2.2.10 Void

The `Void` type is a type which has one literal `?` used for constructing concrete values of the type `void`. The `Void` type is part of the `Bits` and `Literal` typeclasses.

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

Examples

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

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

2.2.11 Maybe

The `Maybe` type is used for tagging values as either *Valid* or *Invalid*. If the value is *Valid*, the value contains a datatype `data_t`.

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

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

The `Maybe` data type provides functions to check if the value is *Valid* and to extract the valid value.

`Maybe` Functions	
`fromMaybe`	Extracts the `Valid` value out of a `Maybe` argument. If the tag is `Invalid` the default value, `defaultval`, is returned.
	`function data_t fromMaybe( data_t defaultval,` `                            Maybe#(data_t) val ) ;`

`isValid`	Returns a value of `True` if the `Maybe` argument is `Valid`.
	`function Bool isValid( Maybe#(data_t) val ) ;`

Examples

```
RWire#(int) rw_incr <- mkRWire();  //  increment method is being invoked
RWire#(int) rw_decr <- mkRWire();  //  decrement method is being invoked

rule doit;
   Maybe#(int) mbi = rw_incr.wget();
   Maybe#(int) mbd = rw_decr.wget();
   int   di        = fromMaybe (?, mbi);
   int   dd        = fromMaybe (?, mbd);
   if      ((! isValid (mbi)) && (! isValid (mbd)))
      noAction;
   else if (   isValid (mbi)  && (! isValid (mbd)))
      value2 <= value2 + di;
   else if ((! isValid (mbi)) &&    isValid (mbd))
      value2 <= value2 - dd;
   else // (   isValid (mbi)  &&    isValid (mbd))
      value2 <= value2 + di - dd;
endrule
```

2.2.12 Tuples

Tuples are predefined structures which group a small number of values together. The following pseudo code explains the structure of the tuples. You cannot define your own tuples, but must use the seven predefined tuples, `Tuple2` through `Tuple8`. As shown, `Tuple2` groups two items together, `Tuple3` groups three items together, up through `Tuple8` which groups eight items together.

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

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

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

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

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

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

typedef struct{
    a tpl_1;
    b tpl_2;
    c tpl_3;
    d tpl_4;
    e tpl_5;
    f tpl_6;
    g tpl_7;
    h tpl_8;
 } Tuple8 #(type a, type b, type c, type d, type e, type f, type g, type h)
   deriving (Bits, Eq, Bounded);
```

Type Classes for `Tuples`

	`Bits`	`Eq`	`Literal`	`Arith`	`Ord`	`Bounded`	`Bitwise`	`Bit Reduction`	`Bit Extend`
`TupleN`	√	√			√	√			

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

`Tuple` Constructor Functions	
`tuple2 (e1, e2)`	Creates a variable of type Tuple2 with component values e1 and e2.
`tuple3 (e1, e2, e3)`	Creates a variable of type Tuple3 with values e1, e2, and e3.
`tuple4 (e1, e2, e3, e4)`	Creates a variable of type Tuple4 with component values e1, e2, e3, and e4.
`tuple5 (e1, e2, e3, e4, e5)`	Creates a variable of type Tuple5 with component values e1, e2, e3, e4, and e5.
`tuple6 (e1, e2, e3, e4, e5, e6)`	Creates a variable of type Tuple6 with component values e1, e2, e3, e4, e5, and e6.
`tuple7 (e1, e2, e3, e4, e5, e6, e7)`	Creates a variable of type Tuple7 with component values e1, e2, e3, e4, e5, e6, and e7.
`tuple8 (e1, e2, e3, e4, e5, e6, e7, e8)`	Creates a variable of type Tuple8 with component values e1, e2, e3, e4, e5, e6, e7, and e8.

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

`Tuple` Extract Functions	
`tpl_1 (x)`	Extracts the first field of x from a Tuple2 to Tuple8.
`tpl_2 (x)`	Extracts the second field of x from a Tuple2 to Tuple8.
`tpl_3 (x)`	Extracts the third field of x from a Tuple3 to Tuple8.
`tpl_4 (x)`	Extracts the fourth field of x from a Tuple4 to Tuple8.
`tpl_5 (x)`	Extracts the fifth field of x from a Tuple5 to Tuple8.
`tpl_6 (x)`	Extracts the sixth field of x from a Tuple6, Tuple7 or Tuple8.
`tpl_7 (x)`	Extracts the seventh field of x from a Tuple7 or Tuple8.
`tpl_8 (x)`	Extracts the seventh field of x from a Tuple8.

Examples

```
Tuple2#( Bool, int ) foo = tuple2( True, 25 );
Bool field1 = tpl_1( foo ); // this is value 1 in the list
int  field2 = tpl_2( foo ); // this is value 2 in the list
foo = tuple2( !field1, field2 );
```

2.2.13 Array

Array variables are generally declared anonymously, using the bracket syntax. However, the type of such variables can be expressed with the type constructor `Array`, when an explicit type is needed.

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

For example, the following declarations using bracket syntax:

```
Bool arr[3];

function Bool fn(Bool bits[]);
```

are equivalent to the following declarations using the explicit type constructor:

```
Array#(Bool) arr;

function Bool fn(Array#(Bool) bits);
```

Note that, unlike `Vector`, the size of an array is not part of its type. In the first declaration, a size is given for the array `arr`. However, since `arr` is not assigned to a value, the size is unused here. If the array were assigned, the size would be used like a type declaration, to check that the assigned value has the declared size. Since it is not part of the type, this check would occur during elaboration, and not during type checking.

2.2.14 Ordering

The `Ordering` type is used as the return type for the result of generic comparators, including the `compare` function defined in the `Ord` (Section 2.1.7) type class. The valid values of `Ordering` are: `LT`, `GT`, and `EQ`.

```
typedef enum {
    LT,
    EQ,
    GT
} Ordering deriving (Eq, Bits, Bounded);
```

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

Examples

```
function Ordering onKey(Record r1, Record r2);
   return compare(r1.key,r2.key);
endfunction
```

2.2.15 File

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

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

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

Note: `Bitwise` operations are valid only for subtype `MCD`.

The `File` type is used by the system tasks for file I/O.

2.2.16 Clock

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

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

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

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

Examples

```
Clock clk <- exposeCurrentClock;

module mkTopLevel( Clock readClk, Reset readRst, Top ifc );
```

2.2.17 Reset

`Reset` is an abstract type.

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

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

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

Examples

```
Reset rst <- exposeCurrentReset;

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

interface ResetGenIfc;
  interface Reset gen_rst;
endinterface
```

2.2.18 Inout

An `Inout` type is a first class type that is used to pass Verilog inouts through a BSV module. It takes an argument which is the type of the underlying signal:

```
Inout#(type t)
```

For example, the type of an `Inout` signal which communicates boolean values would be:

```
Inout#(Bool)
```

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

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

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

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

A module with an `Inout` subinterface cannot leave that interface undefined since there is no way to create or examine inout values in BSV. For example, you cannot even write:

```
Inout#(int) i = ? ;    // not valid in BSV
```

The `Inout` type exists only so that RTL inout signals can be connected in BSV; the ultimate users of the signal will be outside the BSV code. An imported Verilog module might have an inout port that your BSV design doesn't use, but which needs to be exposed at the top level. In this case, the submodule will introduce an inout signal that the BSV cannot read or write, but merely provides in its interfaces until it is exposed at the top level. Or, a design may contain two imported Verilog modules that have inout ports that expect to be connected. You can import these two modules, declaring that they each have a port of type `Inout#(t)` and connect them together. The compiler will check that both ports are of the same type `t` and that they are in the same clock domain with the same reset. Beyond that, BSV does not concern itself with the values of the inout signals.

Examples

Instantiating a submodule with an inout and exposing it at the next level:

```
interface SubIfc;
   ...
   interface Inout#(Bool) b;
endinterface

interface TopIfc;
   ...
   interface Inout#(Bool) bus;
endinterface

module mkTop (TopIfc);
   SubIfc sub <- mkSub;
   ...
   interface bus = sub.b;
endmodule
```

Connecting two submodules, using `SubIfc` defined above:

```
module mkTop(...);
   ...
   SubIfc sub1 <- mkSub;
   SubIfc sub2 <- mkSub;
   mkConnection (sub1.b, sub2.b);
   ...
endmodule
```

2.2.19 Action/ActionValue

Any expression that is intended to act on the state of the circuit (at circuit execution time) is called an *action* and has type `Action` or `ActionValue#(a)`. The type parameter `a` represents the type of the returned value.

Type Classes for `Action/ActionValue`									
	Bits	Eq	Literal	Arith	Ord	Bounded	Bitwise	Bit Reduction	Bit Extend
Action									

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

`typedef` ··· *abstract* ··· `struct ActionValue#(type a);`

`ActionValue` type aliases	
Action	The `Action` type is a special case of the more general type `ActionValue` where nothing is returned. That is, the returns type is `(void)`.
	`typedef ActionValue#(void) Action;`

`Action` Functions	
noAction	An empty `Action`, this is an `Action` that does nothing.
	`function Action noAction();`

Examples

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

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

seq
  noAction;
endseq
```

2.2.20 Rules

A rule expression has type `Rules` and consists of a collection of individual rule constructs. Rules are first class objects, hence variables of type `Rules` may be created and manipulated. `Rules` values must eventually be added to a module in order to appear in synthesized hardware.

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

The `Rules` data type provides functions to create, manipulate, and combine values of the type `Rules`.

`Rules` Functions	
`emptyRules`	An empty rules variable.
	`function Rules emptyRules();`

`addRules`	Takes rules `r` and adds them into a module. This function may only be called from within a module. The return type `Empty` indicates that the instantiation does not return anything.
	`module addRules#(Rules r) (Empty);`

`rJoin`	Symmetric union of two sets of rules. A symmetric union means that neither set is implied to have any relation to the other: not more urgent, not execute before, etc.
	`function Rules rJoin(Rules x, Rules y);`

`rJoinPreempts`	Union of two sets of rules, with rules on the left getting scheduling precedence and blocking the rules on the right.That is, if a rule in set `x` fires, then all rules in set `y` are prevented from firing. This is the same as specifying `descending_urgency` plus a forced conflict.
	`function Rules rJoinPreempts(Rules x, Rules y);`

`rJoinDescendingUrgency`	
	Union of two sets of rule, with rules in the left having higher urgency.That is, if some rules compete for resources, then scheduling will select rules in set `x` set before set `y`. If the rules do not conflict, no conflict is added; the rules can fire together.
	`function Rules rJoinDescendingUrgency(Rules x, Rules y);`

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

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

rJoinConflictFree	
	Union of two sets of rule, with the rules in the left set annotated as conflict-free with the rules in the right set. This assumption is used during scheduling and checked during simulation. No relationship between the rules in the left set or between the rules in the right set is assumed.
	`function Rules rJoinConflictFree(Rules x, Rules y);`

Examples (This is an excerpt from a complete example in the BSV Reference Guide.)

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

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

2.3 Operations on Numeric Types

2.3.1 Size Relationship/Provisos

These classes are used in provisos to express constraints between the sizes of types.

Class	Proviso	Description
`Add`	`Add#(n1,n2,n3)`	Assert $n1 + n2 = n3$
`Mul`	`Mul#(n1,n2,n3)`	Assert $n1 * n2 = n3$
`Div`	`Div#(n1,n2,n3)`	Assert ceiling $n1/n2 = n3$
`Max`	`Max#(n1,n2,n3)`	Assert $\max(n1, n2) = n3$
`Min`	`Min#(n1,n2,n3)`	Assert $\min(n1, n2) = n3$
`Log`	`Log#(n1,n2)`	Assert ceiling $\log_2(n1) = n2$.

Examples of Provisos using size relationships:

```
instance Bits #( Vector#(vsize, element_type), tsize)
   provisos (Bits#(element_type, sizea),
             Mul#(vsize, sizea, tsize));       // vsize * sizea = tsize

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

function Vector#(mvsize,element_type)
      concat(Vector#(m,Vector#(n,element_type)) xss)
  provisos (Mul#(m,n,mvsize));                 // m * n = mvsize
```

2.3.2 Size Relationship Type Functions

These type functions are used when "defining" size relationships between data types, where the defined value need not (or cannot) be named in a proviso. They may be used in datatype definition statements when the size of the datatype may be calculated from other parameters.

Type Function	Size Relationship	Description
TAdd	TAdd#(n1,n2)	Calculate $n1 + n2$
TSub	TSub#(n1,n2)	Calculate $n1 - n2$
TMul	TMul#(n1,n2)	Calculate $n1 * n2$
TDiv	TDiv#(n1,n2)	Calculate ceiling $n1/n2$
TLog	TLog#(n1)	Calculate ceiling $\log_2(n1)$
TExp	TExp#(n1)	Calculate 2^{n1}
TMax	TMax#(n1,n2)	Calculate $max(n1, n2)$
TMin	TMin#(n1,n2)	Calculate $min(n1, n2)$

Examples using other arithmetic functions:

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

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

typedef Bit#(TLog#(n)) CBToken#(numeric type n);
                                          // defines a new parameterized type,
                                          // CBToken, which is log(n) bits wide.

typedef 8 Wordsize;                       // Blocksize is based on Wordsize
typedef TAdd#(Wordsize, 1) Blocksize;
```

2.3.3 valueOf and SizeOf pseudo-functions

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

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

Examples

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

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

Examples

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

2.4 Registers and Wires

Register and Wire Interfaces and Modules		
Name	Section	Description
`Reg`	2.4.1	Register interface
`CReg`	2.4.2	Implementation of a register with an array of `Reg` interfaces that sequence concurrently
`RWire`	2.4.3	Similar to the `Reg` interface with output wrapped in a `Maybe` type to indicate validity
`Wire`	2.4.4	Interchangeable with a `Reg` interface, validity of the data is implicit
`BypassWire`	2.4.5	Implementation of the `Wire` interface where the `_write method` is `always_enabled`
`DWire`	2.4.6	Implementation of the `Wire` interface where the `_read` method is `always_ready` by providing a default value
`PulseWire`	2.4.7	Similar to the `RWire` interface without any data
`ReadOnly`	2.4.8	Interface which provides a value
`WriteOnly`	2.4.9	Interface which writes a value

2.4.1 Reg

The most elementary module available in BSV is the register, which has a `Reg` interface. Registers are polymorphic, i.e., in principle they can hold a value of any type but, of course, ultimately registers store bits. Thus, the provisos on register modules indicate that the type of the value stored in the register must be in the `Bits` type class, i.e., the operations `pack` and `unpack` are defined on the type to convert into bits and back.

Note that all Bluespec registers are considered atomic units, which means that even if one bit is updated (written), then all the bits are considered updated. This prevents multiple rules from updating register fields in an inconsistent manner.

When scheduling register modules, reads occur before writes. That is, any rule which reads from a register must be scheduled earlier than any other rule which writes to the register. The value read from the register is the value written in the previous clock cycle.

Interfaces and Methods

The `Reg` interface contains two methods, `_write` and `_read`.

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

The `_write` and `_read` methods are rarely used. Instead, for writes, one uses the non-blocking assignment notation and, for reads, one just mentions the register interface in an expression.

`Reg` Interface				
Method			Arguments	
Name	Type	Description	Name	Description
`_write`	`Action`	writes a value `x1`	`x1`	data to be written
`_read`	`a_type`	returns the value of the register		

Modules

Prelude provides three modules to create a register: `mkReg` creates a register with a given reset value, `mkRegU` creates a register without any reset, and `mkRegA` creates a register with a given reset value and with asynchronous reset logic.

`mkReg`	Make a register with a given reset value. Reset logic is synchronous.
	`module mkReg#(parameter a_type resetval)(Reg#(a_type))` `  provisos (Bits#(a_type, sizea));`

`mkRegU`	Make a register without any reset; initial simulation value is alternating 01 bits.
	`module mkRegU(Reg#(a_type))` `  provisos (Bits#(a_type, sizea));`

`mkRegA`	Make a register with a given reset value. Reset logic is asynchronous.
	`module mkRegA#(parameter a_type resetval)(Reg#(a_type))` `  provisos (Bits#(a_type, sizea));`

Scheduling

When scheduling register modules, reads occur before writes. That is, any rule which reads from a register must be scheduled earlier than any other rule which writes to the register. The value read from the register is the value written in the previous clock cycle. Multiple rules can write to a register in a given clock cycle, with the effect that a later rule overwrites the value written by earlier rules.

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

Functions

Three functions are provided for using registers: `asReg` returns the register interface instead of the value of the register; `readReg` reads the value of a register, useful when managing vectors or lists of registers; and `writeReg` to write a value into a register, also useful when managing vectors or lists of registers.

`asReg`	Treat a register as a register, i.e., suppress the normal behavior where the interface name implicitly represents the value that the register contains (the `_read` value). This function returns the register interface, not the value of the register.
	`function Reg#(a_type) asReg(Reg#(a_type) regIfc);`

`readReg`	Read the value out of a register. Useful for giving as the argument to higher-order vector and list functions.
	`function a_type readReg(Reg#(a_type) regIfc);`

`writeReg`	Write a value into a register. Useful for giving as the argument to higher-order vector and list functions.
	`function Action writeReg(Reg#(a_atype) regIfc, a_type din);`

Examples

```
Reg#(ta) res <- mkReg(0);

// create board[x][y]
Reg#(ta) pipe[depth];
for (Integer i=0; i<depth; i=i+1) begin
   Reg#(ta) c();
   mkReg#(0) xinst(c);
   pipe[i] = asReg(c);
end

function a readReg(Reg#(a) r);
   return(r);
endfunction
```

2.4.2 CReg

The basic register modules described in 2.4.1 have scheduling annotations that do not allow two rules to read and write a register concurrently (that is, sequentially in the same clock cycle). Implementing this concurrency requires bypassing, so that a value written by one rule is visible to the next rule that wants to read the register. This can create long paths, and so explicit input from the designer is preferred. Therefore, the basic registers do not support this bypassing. If a designer wants concurrent register access between rules, they must explicitly request and manage this, by instantiating one of the `CReg` family of modules.

These concurrent registers are also known as EHRs (Ephemeral History Registers) in work by Arvind and Rosenband[1].

Modules

Prelude provides three modules to create a concurrent register: `mkCReg` creates a concurrent register with a given reset value, `mkCRegU` creates a concurrent register without any reset, and `mkCRegA` creates a concurrent register with a given reset value and with asynchronous reset logic.

mkCReg	Make a concurrent register with a given number of ports and with a given reset value. Reset logic is synchronous.

	`module mkCReg#(parameter Integer n,`
	`                parameter a_type resetval)`
	`              (Reg#(a_type) ifc[])`
	`  provisos (Bits#(a_type, sizea));`

mkCRegU	Make a concurrent register with a given number of ports and without any reset. Initial simulation value is alternating 01 bits.
	`module mkCRegU#(parameter Integer n) (Reg#(a_type) ifc[])` `  provisos (Bits#(a_type, sizea));`

mkCRegA	Make a concurrent register with a given number of ports and with a given reset value. Reset logic is asynchronous.

	`module mkCRegA#(parameter Integer n,`
	`                 parameter a_type resetval)`
	`               (Reg#(a_type) ifc[])`
	`  provisos (Bits#(a_type, sizea));`

As indicated by the bracket notation in the interface type, these modules provide an array of `Reg` interfaces, whose methods can be called concurrently in separate actions. The modules take a size parameter `n`, that indicates the size of the array to return. The minimum size is 0. An implementation may specify a maximum size (5, in the current implementation).

Scheduling

When a concurrent register is instantiated, it returns an array of `Reg` interfaces. The scheduling relationships for `_read` and `_write` in one `Reg` interface are the same as for the basic register modules in 2.4.1. The methods of lower-numbered interfaces sequence before the methods of higher-numbered interfaces.

The designer manages the concurrency of the register accesses by choosing which interfaces to assign to which rules.

[1] Daniel L. Rosenband. The Ephemeral History Register: Flexible Scheduling for Rule-Based Designs. In *Proc. MEMOCODE'04*, June 2004.

Scheduling Annotations mkCReg, mkCRegU, mkCRegA for $j < k$				
	read_j	write_j	read_k	write_k
read_j	CF	SB	SBR	SBR
write_j	SA	SBR	SBR	SBR

The `Reg` interfaces execute in sequence starting with element 0 of the array up through element $n-1$. That is, the `_read` method of the first interface will return the value stored in the register from the previous clock cycle; if the `_write` method of the first interface is called, the `_read` method of the second interface will return the written value, otherwise it returns the registered value. And so on, with each `_read` method returning the last value written to any of the lower-numbered interfaces, or the register value if none of the lower-numbered `_write` methods were called. The value registered at the end of clock cycle is the value written by the highest-numbered write method that was called.

Examples

In the following example, the two rules can be scheduled concurrently in the same clock cycle, which would not have been possible if the register `byteCount` had been instantiated as a basic `mkReg` register. Further, if both rules execute in a given clock cycle, the value read in rule `doRecv` is the updated value that was written in rule `doSend`.

```
Reg#(Bool) byteCount[2] <- mkCReg(2, True);

rule doSend (canSend);
   ...
   byteCount[0] <= byteCount[0] + len;
endrule

rule doRecv (canRecv);
   ...
   byteCount[1] <= byteCount[1] - len;
endrule
```

In the above example, `mkCReg` returns an array of `Reg` interfaces. This type is expressed implicitly using the bracket syntax. The type can also be explicitly expressed with the name `Array`, when necessary. (See Section 2.2.13 for more information on the `Array` type.) One place where this can be necessary is when the array type is a component of a larger type. A common example of this would be when instatiating a `Vector` of `mkCReg` modules:

```
Vector#(N, Array#(Reg#(T))) regs <- replicateM(mkCReg(3,0));
```

The above instantiation can also be achieved using multidimensional arrays. Instead of a vector of arrays, one can use an array of arrays:

```
Integer n = valueOf(N);
Reg#(T) regs[n][3];
for (Integer i=0; i<n; i=i+1)
   regs[i] <- mkCReg(3,0);
```

2.4.3 RWire

An RWire is a primitive stateless module whose purpose is to allow data transfer between methods and rules without the cycle latency of a register. That is, a RWire may be written in a cycle and that value can be read out in the same cycle; values are not stored across clock cycles.

When scheduling wire modules, since the value is read in the same cycle in which it is written, writes must occur before reads. That is, any rule which writes to a wire must be scheduled earlier than any other rule which reads from the wire. This is the reverse of how registers are scheduled.

Interfaces and Methods

The RWire interface is conceptually similar to a register's interface, but the output value is wrapped in a Maybe type. The wset method places a value on the wire and sets the valid signal. The read-like method, wget, returns the value and a valid signal in a Maybe type. The output is only Valid if a write has a occurred in the same clock cycle, otherwise the output is Invalid.

RWire Interface				
Method			Arguments	
Name	Type	Description	Name	Description
wset	Action	writes a value and sets the valid signal	datain	data to be sent on the wire
wget	Maybe	returns the value and the valid signal		

```
interface RWire#(type element_type) ;
   method Action wset(element_type datain) ;
   method Maybe#(element_type) wget() ;
endinterface: RWire
```

Modules

The mkRWireSBR, mkRWire, and mkUnSafeRWire modules are provided to create an RWire. The difference between the RWire modules is the scheduling annotations. In mkRWireSBR the wset is SBR with itself, allowing multiple wsets in the same clock cycle (though not, of course, in the same rule).

mkRWireSBR	Creates an RWire. Output is only valid if a write has occurred in the same clock cycle. This is the recommended module to use to create an RWire.
	`module mkRWireSBR(RWire#(element_type))` `   provisos (Bits#(element_type, element_width)) ;`

mkRWire	Creates an RWire. Output is only valid if a write has occurred in the same clock cycle. The write (wset) must be sequenced before the read (wget) and they must be in different rules.
	`module mkRWire(RWire#(element_type))` `   provisos (Bits#(element_type, element_width)) ;`

`mkUnsafeRWire`	Creates an `RWire`. Output is only valid if a write has occurred in the same clock cycle. The write (`wset`) must be sequenced before the read (`wget`) but they can be in the same rule.
	`module mkUnsafeRWire(RWire#(element_type))` `   provisos (Bits#(element_type, element_width)) ;`

Scheduling

When scheduling wire modules, since the value is read in the same cycle in which it is written, writes must occur before reads. That is, any rule which writes to a wire must be scheduled earlier than any other rule which reads from the wire. This is the reverse of how registers are scheduled.

Scheduling Annotations mkRWire		
	wget	wset
wget	CF	SAR
wset	SBR	C

Scheduling Annotations mkRWireSBR		
	wget	wset
wget	CF	SAR
wset	SBR	SBR

Scheduling Annotations mkUnsafeRWire		
	wget	wset
wget	CF	SA
wset	SB	C

Examples

```
RWire#(int) rw_incr <- mkRWire();

rule doit;
   Maybe#(int) mbi = rw_incr.wget();
   int   di        = fromMaybe (?, mbi);
   if      ((! isValid (mbi))
      noAction;
   else // (   isValid (mbi))
      value2 <= value2 + di ;
endrule

rule doitdifferently;
   case (rw_incr.wget()) matches
     { tagged Invalid  } : noAction;
     { tagged Valid .di} : value <= value + di;
   endcase
endrule

method Action increment (int di);
   rw_incr.wset (di);
endmethod
```

2.4.4 Wire

The `Wire` interface and module are similar to `RWire`, but the valid bit is hidden from the user and the validity of the read is considered an implicit condition. The `Wire` interface works like the `Reg` interface, so mentioning the name of the wire gets (reads) its contents whenever they're valid, and using `<=` writes the wire. `Wire` is an `RWire` that is designed to be interchangeable with `Reg`. You can replace a `Reg` with a `Wire` without changing the syntax.

Interfaces and Methods

```
typedef Reg#(element_type) Wire#(type element_type);
```

Wire Interface				
Method			Arguments	
Name	Type	Description	Name	Description
`_write`	`Action`	writes a value `x1`	`x1`	data to be written
`_read`	`a_type`	returns the value of the wire		

Modules

The `mkWire` and `mkUnsafeWire` modules are provided to create a `Wire`. The only difference between the two modules are the scheduling annotations. The `mkWire` version requires that the the write and the read be in different rules.

`mkWire`	Creates a `Wire`. Validity of the output is automatically checked as an implicit condition of the read method. The write and the read methods must be in different rules.
	`module mkWire(Wire#(element_type))` `provisos (Bits#(element_type, element_width));`

`mkUnsafeWire`	Creates a `Wire`. Validity of the output is automatically checked as an implicit condition of the read method. The write and the read methods can be in the same rule.
	`module mkUnsafeWire(Wire#(element_type))` `provisos (Bits#(element_type, element_width));`

Scheduling Annotations mkWire		
	_read	_write
_read	CF	SAR
_write	SBR	C

Scheduling Annotations mkUnsafeWire		
	_read	_write
_read	CF	SA
_write	SB	C

Examples

```
module mkCounter_v2 (Counter);
   Reg#(int) value2 <- mkReg(0);
   Wire#(int) w_incr <- mkWire();

   rule r_incr;
      value2 <= value2 + w_incr;
   endrule

   method int read();
      return value2;
   endmethod

   method Action increment (int di);
      w_incr <= di;
   endmethod
endmodule
```

2.4.5 BypassWire

BypassWire is an implementation of the Wire interface where the `_write` method is an `always_enabled` method. The compiler will issue a warning if the method does not appear to be called every clock cycle. The advantage of this tradeoff is that the `_read` method of this interface does not carry any implicit condition (so it can satisfy a `no_implicit_conditions` assertion or an `always_ready` method).

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

Scheduling Annotations mkBypassWire		
	_read	_write
_read	CF	SAR
_write	SBR	C

Examples

```
module mkCounter_v2 (Counter);
   Reg#(int) value2 <- mkReg(0);
   Wire#(int) w_incr <- mkBypassWire();

   rule r_incr;
      value2 <= value2 + w_incr;
   endrule

   method int read();
      return value2;
   endmethod

   method Action increment (int di);
      w_incr <= di;
   endmethod
endmodule
```

2.4.6 DWire

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

There are two modules to create a `DWire`; the only difference being the scheduling annotations. A write is always scheduled before a read, however the `mkDWire` module requires that the write and read be in different rules.

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

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

Scheduling Annotations mkDWire		
	_read	_write
_read	CF	SAR
_write	SBR	C

Scheduling Annotations mkUnsafeDWire		
	_read	_write
_read	CF	SA
_write	SB	C

Examples

```
module mkCounter_v2 (Counter);
   Reg#(int) value2 <- mkReg(0);
   Wire#(int) w_incr <- mkDWire (0);

   rule r_incr;
      value2 <= value2 + w_incr;
   endrule

   method int read();
      return value2;
   endmethod

   method Action increment (int di);
      w_incr <= di;
   endmethod
endmodule
```

2.4.7 PulseWire

Interfaces and Methods

The `PulseWire` interface is an `RWire` without any data. It is useful within rules and action methods to signal other methods or rules in the same clock cycle. Note that because the read method is called `_read`, the register shorthand can be used to get its value without mentioning the method `_read` (it is implicitly added).

`PulseWire` Interface		
Name	Type	Description
`send`	Action	sends a signal down the wire
`_read`	Bool	returns the valid signal

```
   interface PulseWire;
     method Action send();
     method Bool _read();
   endinterface
```

Modules

Four modules are provided to create a `PulseWire`, the only difference being the scheduling annotations. In the `OR` versions the `send` method does not conflict with itself. Calling the `send` method for a `mkPulseWire` from 2 rules causes the two rules to conflict while in the `mkPulseWireOR` there is no conflict. In other words, the `mkPulseWireOR` acts a logical "OR". The `Unsafe` versions allow the `send` and `_read` methods to be in the same rule.

`mkPulseWire`	The writing to this type of wire is used in rules and action methods to send a single bit to signal other methods or rules in the same clock cycle.
	`module mkPulseWire(PulseWire);`

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

`mkUnsafePulseWire`	The writing to this type of wire is used in rules and action methods to send a single bit to signal other methods or rules in the same clock cycle. The `send` and `_read` methods can be in the same rule.
	`module mkUnsafePulseWire(PulseWire);`

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

Scheduling Annotations mkPulseWire

	_read	send
_read	CF	SAR
send	SBR	C

Scheduling Annotations mkUnsafePulseWire

	_read	send
_read	CF	SA
send	SB	C

Scheduling Annotations mkPulseWireOR

	_read	send
_read	CF	SAR
send	SBR	SBR

Scheduling Annotations mkUnsafePulseWireOR

	_read	send
_read	CF	SA
send	SB	SBR

Counter Example - Using `Reg` and `PulseWire`

```
interface Counter#(type size_t);
    method Bit#(size_t) read();
    method Action load(Bit#(size_t) newval);
    method Action increment();
```

```
    method Action decrement();
endinterface

module mkCounter(Counter#(size_t));
    Reg#(Bit#(size_t)) value <- mkReg(0);          // define a Reg

    PulseWire increment_called <- mkPulseWire();   // define the PulseWires used
    PulseWire decrement_called <- mkPulseWire();   // to signal other methods or rules

    // whether rules fire is based on values of PulseWires
    rule do_increment(increment_called && !decrement_called);
        value <= value + 1;
    endrule

    rule do_decrement(!increment_called && decrement_called);
        value <= value - 1;
    endrule

    method Bit#(size_t) read();                 // read the register
        return value;
    endmethod

    method Action load(Bit#(size_t) newval);    // load the register
        value <= newval;                        // with a new value
    endmethod

    method Action increment();                  // sends the signal on the
        increment_called.send();                // PulseWire increment_called
    endmethod

    method Action decrement();                  /  sends the signal on the
        decrement_called.send();                // PulseWire decrement_called
    endmethod
endmodule
```

2.4.8 ReadOnly

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

Interfaces and Methods

ReadOnly Interface		
Method		
Name	Type	Description
_read	a_type	Reads the data

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

Functions

`regToReadOnly`	Converts a `Reg` interface into a `ReadOnly` interface. Useful for giving as the argument to higher-order vector and list functions.
	`function ReadOnly#(a_type) regToReadOnly(Reg#(a_type) regIfc);`

`pulseWireToReadOnly`	Converts a `PulseWire` interface into a `ReadOnly` interface.
	`function ReadOnly#(Bool) pulseWireToReadOnly(PulseWire ifc);`

`readReadOnly`	Takes a `ReadOnly` interface and returns a value.
	`function a_type readReadOnly(ReadOnly#(a_type) r);`

Examples

```
interface AHBSlaveIFC;
   interface AHBSlave              bus;
   interface Put#(AHBResponse)     response;
   interface ReadOnly#(AHBRequest) request;
endinterface
...
   interface ReadOnly request;
      method AHBRequest _read;
         let ctrl = AhbCtrl {command:  write_wire,
                             size:     size_wire,
                             burst:    burst_wire,
                             transfer: transfer_wire,
                             prot:     prot_wire,
                             addr:     addr_wire} ;
         let value = AHBRequest {ctrl: ctrl, data: wdata_wire};
         return value;
      endmethod
   endinterface
```

2.4.9 WriteOnly

`WriteOnly` is an interface which writes a value. The `_write` shorthand is used to write the value.

Interfaces and Methods

`WriteOnly` Interface				
Method			Arguments	
Name	Type	Description	Name	Description
`_write`	Action	Writes the data	`x`	Value to be written, of datatype `a_type`.

```
interface WriteOnly #( type a_type ) ;
   method Action _write (a_type x) ;
endinterface
```

Examples

```
interface WriteOnly#(type a);
   method Action _write(a v);
endinterface

// module with an always-enabled port to tie to a default value
   import "BVI" AlwaysWrite =
      module mkAlwaysWrite(WriteOnly#(a)) provisos(Bits#(a,sa));
        no_reset;
        parameter width = valueof(sa);
        method _write(D_IN) enable((*inhigh*)EN);
        schedule _write C _write;
   endmodule

   module mkDefaultValue1();
      WriteOnly#(UInt#(7)) d1 <- mkAlwaysWrite(clocked_by primMakeDisabledClock);
      rule handle_d1;
          d1 <= 5;
      endrule
   endmodule
```

2.5 Miscellaneous Functions

2.5.1 Compile-time Messages

`error`	Generate a compile-time error message, `s`, and halt compilation.
	`function a_type error(String s);`

`warning`	When applied to a value `v` of type `a`, generate a compile-time warning message, `s`, and continue compilation, returning `v`.
	`function a_type warning(String s, a_type v);`

`message`	When applied to a value `v` of type `a`, generate a compile-time informative message, `s`, and continue compilation, returning `v`.
	`function a_type message(String s, a_type v);`

`errorM`	Generate a compile-time error message, `s`, and halt compilation in a monad.
	`function m#(void) errorM(String s)` `  provisos (Monad#(m));`

`warningM`	Generate a compilation warning in a monad.
	`function m#(void) warningM(String s)` `  provisos (Monad#(m));`

`messageM`	Generate a compilation message in a monad.
	`function m#(void) messageM(String s)` `  provisos (Monad#(m));`

2.5.2 Arithmetic Functions

`max`	Returns the maximum of two values, x and y.
	`function a_type max(a_type x, a_type y)` `  provisos (Ord#(a_type));`

`min`	Returns the minimum of two values, x and y.
	`function a_type min(a_type x, a_type y)` `  provisos (Ord#(a_type));`

`abs`	Returns the absolute value of x.
	`function a_type abs(a_type x)` `  provisos (Arith#(a_type), Ord#(a_type));`

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

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

`signedQuot`	Performs full precision division on two Int#(n) operands of different sizes.
	`function Int#(m) signedQuot(Int#(n) x, Int#(k) y) ;`

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

2.5.3 Operations on Functions

Higher order functions are functions which take functions as arguments and/or return functions as results. These are often useful with list and vector functions.

`compose`	Creates a new function, `c`, made up of functions, `f` and `g`. That is, `c(a) = f(g(a))`

	`function (function c_type (a_type x0))`
	`     compose(function c_type f(b_type x1),`
	`             function b_type g(a_type x2));`

`composeM`	Creates a new monadic function, `m#(c)`, made up of functions, `f` and `g`. That is, `c(a) = f(g(a))`

	`function (function m#(c_type) (a_type x0))`
	`      composeM(function m#(c_type) f(b_type x1),`
	`               function m#(b_type) g(a_type x2))`
	`   provisos # (Monad#(m));`

`id`	Identity function, returns `x` when given `x`. This function is useful when the argument requires a function which doesn't do anything.
	`function a_type id(a_type x);`

`constFn`	Constant function, returns `x`.
	`function a_type constFn(a_type x, b_type y);`

`flip`	Flips the arguments `x` and `y`, returning a new function.
	`function (function c_type new (b_type y, a_type x))` `     flip (function c_type old (a_type x, b_type y));`

`curry`	This function converts a function on a pair (Tuple2) of arguments into a function which takes the arguments separately. The phrase `t0 f(t1 x, t2 y)` is the function returned by `curry`
	`function (function t0 f(t1 x, t2 y))` `     curry (function t0 g(Tuple2#(t1, t2) x));`

`uncurry`	This function does the reverse of `curry`; it converts a function of two arguments into a function which takes a single argument, a pair (Tuple2).	`function (function t0 g(Tuple2#(t1, t2) x))` `     uncurry (function t0 f(t1 x, t2 y));`

Examples

```
//using constFn to set the initial values of the registers in a list
  List#(Reg#(Resource)) items <- mapM( constFn(mkReg(initRes)),upto(1,numAdd) );

  return(pack(map(compose(head0,toList),state)));

  xs <- mapM(constFn(mkReg(False)),genList);
```

2.5.4 Bit Functions

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

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

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

`countOnes`	Returns the count of the number of 1's in the bit vector `bin`.	`function UInt#(lgn1) countOnes ( Bit#(n) bin )`
		`  provisos (Add#(1, n, n1), Log#(n1, lgn1),`
		`            Add#(1, xx, lgn1) );`

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

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

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

Examples

```
Bit#(6) f6 = truncateLSB(f);

let cmem=countZerosLSB(cfg.memoryAllocate);

let n = countOnes(neighbors);
```

2.5.5 Integer Functions

The following functions can only be used for static elaboration.

`gcd`	Calculate the greatest common divisor of two Integers. `function Integer gcd(Integer a, Integer b);`

`lcm`	Calculate the least common multiple of two Integers. `function Integer lcm(Integer a, Integer b);`

2.5.6 Control Flow Function

`while`	Repeat a function while a predicate holds.
	`function a_type while(function Bool pred(a_type x1),`
	`             function a_type f(a_type x1), a_type x);`

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

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

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

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

2.6 Environment Values

The `Environment` section of the Prelude contains some value definitions that remain static within a compilation, but may vary between compilations.

Test whether the compiler is generating C.

`genC`	Returns `True` if the compiler is generating C.
	`function Bool genC();`

Test whether the compiler is generating Verilog.

`genVerilog`	Returns `True` if the compiler is generating Verilog.
	`function Bool genVerilog();`

Examples

```
if (genVerilog)
return (t + fromInteger(adj));
```

The following two variables provide access to the names of the package being compiled and the module being synthesized as strings.

`genPackageName`	Returns a `String` containing the name of the package being compiled.
	`function String genPackageName;`

`genModuleName`	Returns a `String` containing the name of the module being synthesized.
	`function String genModuleName;`

Return the version of the compiler.

`compilerVersion`	Returns a `String` containing the compiler version. This is the same string used with the `-v` flag.
	`String compilerVersion;`

Example:

```
The statement:
     $display("compiler version = %s", compilerVersion);
produces this output:
     compiler version = version 3.8.56 (build 7084, 2005-07-22)
```

Return the build number of the version of the compiler.

`buildVersion`	Returns a `Bit#(32)` containing the build number portion of the compiler version.
	`Bit#(32) buildVersion;`

Example:

```
The statement:
     $display("The build version of the compiler is %d", buildVersion);
produces this output:
     "The build version of the compiler is 12345"
```

Get the current date and time.

`date`	Returns a `String` containing the date.
	`String date;`

Example:

```
The statement:
     $display("date = %s", date);
produces this output:
     "date = Mon Feb 6 08:39:59 EST 2006"
```

Returns the number of seconds from the epoch (1970-01-01 00:00:00) to now.

`epochTime`	Returns a `Bit#(32)` containing the number of seconds since the epoch, which is defined as 1970-01-01 00:00:00.
	`Bit#(32) epochTime;`

Example:

```
The statement:
     $display("Current epoch is %d", epochTime);
produces this output:
     "Current epoch is 1235481642"
```

2.7 Compile-time IO

These functions control file IO during elaboration. The functions are expressed as modules and can only be used as statements inside a `module...endmodule` block.

The type `Handle` is a primitive type for a file handle. The value is returned when you open a file and is used to specify the file by the other functions.

The flag `-fdir`, described in the *User Guide*, can be used to specify where relative file paths should be based from.

The type `IOMode` is an enumerated type with three values: `ReadMode`, `WriteMode`, and `AppendMode`:

```
typedef enum { ReadMode, WriteMode, AppendMode } IOMode;
```

When opening a file you specify the mode (`IOMode`) and the filename. Opening a file in write mode creates a new file; in append mode it adds to an existing file.

`openFile`	Opens a file and returns the type `Handle`.
	`module openFile #(String filename, IOMode mode) (Handle);`

The function `hClose` explicitly closes the file with the specified handle. The compiler will close any handles that are still open at the end of elaboration, or upon exiting with an error, but you shouldn't rely on this. Buffered files will be flushed when the file is closed.

`hClose`	Closes the file with the specified handle.
	`module hClose #(Handle hdl) ();`

The following functions provide query functions for handles.

`hIsEOF`	Returns a Bool indicating if the end of file has been reached for the specified handle.
	`function Bool hIsEOF (Handle hdl);`

`hIsOpen`	Returns true if the the handle `hdl` is open.
	`function Bool hIsOpen (Handle hdl);`

`hIsClosed`	Returns true if the handle `hdl` is closed.
	`function Bool hIsClosed (Handle hdl);`

`hIsReadable`	Returns true if the handle has been opened in Readable mode and can be read from.
	`function Bool hIsReadable (Handle hdl);`

`hIsWriteable`	Returns true if handle has been opened in Writeable mode and can be written to.
	`function Bool hIsWriteable (Handle hdl);`

The default buffering of files is determined by your system. If the system is buffering, you may not see any output until the handle is flushed or closed. You can override this by setting the buffering policy of the handle, so that writes are not buffered, or are line buffered. The file handle functions `hFlush`, `hGetBuffering`, and `hSetBuffering` allow you to control buffering.

At the end of elaboration, or upon exiting with an error, the compiler closes any file handles that were not otherwised closed. Any buffered files will be flushed at this point.

The data type `BufferMode` indicates the type of buffering.

```
typedef union tagged {
   void NoBuffering;
   void LineBuffering;
   Maybe#(Integer) BlockBuffering;
} BufferMode;
```

`hFlush`	Explicitly flushes the buffer with the specified handle.
	`function Action hFlush(Handle hdl);`

`hGetBuffering`	Returns the buffering policy of the file with the specified handle.
	`function ActionValue#(BufferMode) hGetBuffering(Handle hdl);`

`hSetBuffering`	Sets the buffering mode for the file with the specified handle if the file system supports it.
	`function Action hSetBuffering(Handle hdl, BufferMode mode);`

The functions `hPutStr` and `hPutStrLn` write strings to a file. The function `hPutStrLn` adds a newline to the end of the string.

`hPutStr`	Writes the string to the file with the specified handle.
	`module hPutStr #(Handle hdl, String str) ();`

`hPutStrLn`	Writes the string to the file with the specified handle and appends a newline to the end of the string.
	`module hPutStr #(Handle hdl, String str) ();`

`hPutChar`	Writes the character to the file with the specified handle.
	`module hPutChar #(Handle hdl, Char c) ();`

`hGetChar`	Reads the character from the file with the specified handle.	
	`module hGetChar #(Handle hdl) (Char);`	

`hGetChar`	Reads the character from the file with the specified handle. `module hGetChar #(Handle hdl) (Char);`

`hGetLine`	Reads a line from the file with the specified handle. `module hGetLine #(Handle hdl) (String);`

Example: Creates a file named `sysBasicWrite.log` containing the line `"Hello World"`.

```
String fname = "sysBasicWrite.log";

module sysBasicWrite ();
   Handle hdl <- openFile(fname, WriteMode);
   hPutStr(hdl, "Hello");
   hClose(hdl);
   mkSub;
endmodule

module mkSub ();
   Handle hdl <- openFile(fname, AppendMode);
   hPutStrLn(hdl, " World");
   hClose(hdl);
endmodule
```

2.8 Generics

Generics is a mechanism permitting users to derive instances of their own custom type classes. The design of generics in Bluespec is based on the GHC.Generics library in Haskell. Generics provides a way of converting arbitrary struct and tagged union/data types to and from a generic representation. In this representation product types are represented as tuples/`PrimPair`, and sum types as `Either`. The representation types are also tagged with various metadata about the types, fields and constructors, such as their name, arity and index. Users can implement a default instance for their type class, using a helper type class over the generic representation.

Due to the complexity of the types involved, writing generic instances in Bluespec SystemVerilog can be rather tedious. Thus the documentation here is instead given using the Bluespec Haskell/Classic syntax.

2.8.1 The `Generic` type class

The `Generic` type class defines a means of converting values of a datatype `a` into a generic representation `r`. The type `r` is determined by the type `a` as a functional dependency. The function `from` converts a value into its generic representation, and `to` converts a generic representation back into a value.

```
class Generic a r | a -> r where
   from :: a -> r
   to :: r -> a
```

BSC automatically derives an instance of `Generic` for all types that don't have an explicit instance. For libraries that export a type abstractly (without exporting its internals), an explicit instance is needed, to avoid exposing the internal implementation; see the source of the `Vector` library for an example of this.

2.8.2 Representation types

The following types are used in generic representations:

`data Either a b` `  = Left a | Right b`	The standard `Either` type is used to represent sum types, *i.e.* tagged unions/data with multiple constructors.

`interface PrimPair a b =`
`  fst :: a`
`  snd :: b`
	The standard `PrimPair` type is used to represent product types, *i.e.* structs/interfaces/data constructors with multiple fields. Since this is also the same underlying representation as tuples, tuple syntax (as described in the BHref guide) may be used for products in generic instances.

`interface PrimUnit = { }`	The standard `PrimUnit` type is used to represent types containing no data, *i.e.* empty structs/interfaces/data constructors.
`data (Vector :: # -> * -> *)` `  n a`	The standard `Vector` type is used to represent fixed-size collection types – `ListN` and `Vector`.
`data Conc a = Conc a`	The `Conc` type is used to wrap the (non-generic) types of fields in the sum of products.
`data ConcPrim a = ConcPrim a`	The `ConcPrim` type is used in `Generic` instances for primitive types, *e.g.* `Bit`. This type is used instead of `Conc` to avoid infinite recursion through a `Conc` instance of the generic version of a type class that defaults back to the non-generic one. Users of generics typically do not need to define instances for `ConcPrim`; this exists to help supply generic instances for several type classes that are used internally by BSC.
`data ConcPoly a = ConcPoly a`	The `ConcPoly` type exists as a workaround for a limitation of generics in dealing with higher-rank data types; see below for more details. Users typically should not need to define instances for `ConcPoly`.
`data Meta m r = Meta r`	`Meta` is used to tag a representation type with additional type-level metadata. The `m` type parameter must be one of the below metadata types

Examples

Ignoring metadata, the derived `Generic` instance for the `PrimPair` type is

```
instance Generic (PrimPair a b) (Conc a, Conc b) where
   from x = (Conc x.fst, Conc x.snd)
   to (Conc x, Conc y) = PrimPair { fst=x; snd=y; }
```

The derived `Generic` instance for the `List` type is

```
instance Generic (List a) (Either () (Conc a, Conc (List a))) where
   from Nil = Left ()
   from (Cons x y) = Right (Conc x, Conc y)
   to (Left ()) = Nil
   to (Right (Conc x, Conc y)) = Cons x y
```

In the generic representation for data/tagged unions with more than two constructors, the `Either` types are aranged to form a left-biased, balanced binary tree. This makes it possible to directly convert between nested left/right constructors, and a binary tag value corresponding to the constructor index. An example of this, in implementing a `CustomBits` type class, is given below.

Higher-rank data

Generics is not able to fully handle *higher-rank* data, *i.e.* structs/interfaces/data constructors containing type variables that are not bound as type parameters. For example, the following type is higher-rank:

```
struct Foo =
   x :: a -> a -- Higher rank
   y :: Int 8
```

If a `Generic` instance were derived for this type in the usual fasion, then the representation would contain `Conc (a -> a)`. The presence of a type variable in the representation means that it is not uniquely determined by the type `Foo`, as required by the functional dependency.

Instead, when deriving an instance for a higher-rank struct or data, a "wrapper" struct is generated for each higher-rank field. This is wrapped in the `ConcPoly` constructor, to indicate that this is not the real type of the field:

```
struct Foo_x =
  val :: a -> a

instance Generic Foo (ConcPoly Foo_x, Conc (Int 8))) where
  from a = (ConcPoly (Foo_x { val = a.x; }), Conc a.y)
  to (ConcPoly x, Conc y) = Foo { x = x.val; y = y; }
```

Users can omit an instance for `ConcPoly` to not support higher-rank data, or define some useful default behavior. For example, the `CShow` library defines an instance for ConcPoly to return a placeholder string for higher-rank fields.

2.8.3 Metadata types

The following types are used to represent metadata in generic representations. Note that these only appear at the type level tagging a `Meta` type; values of these types are not constructed.

`data (MetaData :: $ -> $ -> * -> # -> *) name pkg tyargs ncons`	Indicates that a representation is for a type (e.g. struct/data) with a name, package, tuple of type arguments and number of constructors. Types of kind `*`, `#` or `$` appearing in the type arguments are wrapped in one of the following type constructors:
	`data (StarArg :: * -> *) i`
	`data (NumArg :: # -> *) i`
	`data (StrArg :: $ -> *) i`
	`data ConArg`
	Constructor-kinded types arguments cannot be handled in general and are omitted from the `ConArg` representation type.

`data (MetaConsNamed :: $ -> # -> # -> *) name idx nfields`	Indicates that a representation is for a constructor with named fields, with a name, index in the data's constructors, and number of fields.
`data (MetaConsAnon :: $ -> # -> # -> *) name idx nfields`	Indicates that a representation is for a data constructor with anonymous fields, with a name, index in the data's constructors, and number of fields.
`data (MetaField :: $ -> # -> *) name idx`	Indicates that a representation is for a field, with a field name (either the given name for a named field or the generated field name for an anonymous field) and index in the constructor's fields.

Examples

Including metadata, the derived `Generic` instance for the `PrimPair` type is

```
instance Generic (PrimPair a b)
   (Meta (MetaData "PrimPair" "Prelude" (StarArg a, StarArg b) 1)
   (Meta (MetaConsNamed "PrimPair" 0 2)
   (Meta (MetaField "fst" 0) (Conc a),
      Meta (MetaField "snd" 1) (Conc b)))) where
from x = Meta (Meta (Meta (Conc x.fst), Meta (Conc x.snd)))
to (Meta (Meta (PrimPair (Meta (Conc a1)) (Meta (Conc a2))))) =
   PrimPair { fst = a1; snd = a2; }
```

The derived `Generic` instance for the `List` type is

```
instance Generic (List a)
   (Meta (MetaData "List" "Prelude" (StarArg a) 2)
   (Either (Meta (MetaConsAnon "Nil" 0 0) ())
   (Meta (MetaConsAnon "Cons" 1 2)
      (Meta (MetaField "_1" 0) (Conc a),
      Meta (MetaField "_2" 1) (Conc (List a)))))) where
from Nil = Meta (Left (Meta ()))
from (Cons x y) =
   Meta (Right (Meta (Meta (Conc x), Meta (Conc y))))
to (Meta (Left (Meta ()))) = Nil
to (Meta (Right (Meta ((Meta (Conc x)), (Meta (Conc y)))))) = Cons x y
```

2.8.4 Defining generic instances

The typical way for users to define a generic implementation for their type class is to define a helper type class that works over the generic representation, and then define a default instance for the original type class using `Generic` to convert to and from the generic representation. For example, one can use generics to implement a custom version of the `Bits` type class:

```
class MyBits a n | a -> n where
  mypack   :: a -> Bit n
  myunpack :: Bit n -> a

-- Explicit instances for primitive types
instance MyBits (Bit n) n where
  mypack = id
  myunpack = id

-- Generic default instance
instance (Generic a r, MyBits' r n) => MyBits a n where
  mypack   x  = mypack' $ from x
  myunpack bs = to $ myunpack' bs

class MyBits' r n | r -> n where
  mypack'   :: r -> Bit n
  myunpack' :: Bit n -> r

-- Instance for sum types
instance (MyBits' r1 n1, MyBits' r2 n2, Max n1 n2 c, Add 1 c n,
          Add p1 n1 c, Add p2 n2 c) =>
   MyBits' (Either r1 r2) n where
  mypack' (Left x) = 1'b0 ++ extend (mypack' x)
  mypack' (Right x) = 1'b1 ++ extend (mypack' x)
  myunpack' bs =
     let (tag, content) = (split bs) :: (Bit 1, Bit c)
     in case tag of
        0 -> Left $ myunpack' $ truncate content
        1 -> Right $ myunpack' $ truncate content

-- Instance for product types
instance (MyBits' r1 n1, MyBits' r2 n2, Add n1 n2 n) =>
   MyBits' (r1, r2) n where
  mypack' (x, y) = mypack' x ++ mypack' y
  myunpack' bs = let (bs1, bs2) = split bs
               in (myunpack' bs1, myunpack' bs2)

instance  MyBits' () 0 where
  mypack' () = 0'b0
  myunpack' _ = ()

instance (MyBits' a m, Bits (Vector n (Bit m)) l) =>
   MyBits' (Vector n a) l where
  mypack' v = pack $ map mypack' v
  myunpack' = map myunpack' `compose` unpack

-- Ignore all types of metadata
instance (MyBits' r n) => MyBits' (Meta m r) n where
```

```
    mypack' (Meta x) = mypack' x
    myunpack' bs = Meta $ myunpack' bs

  -- Conc instance calls back to the non-generic MyBits class
  instance (MyBits a n) => MyBits' (Conc a) n where
    mypack' (Conc x) = mypack x
    myunpack' bs = Conc $ myunpack bs
```

A more sophisticated use of generics, making use of metadata, can be found in the implementation of the `CShow` library.

3 Standard Libraries

Section 2 defined the standard Prelude package, which is automatically imported into every package. This section describes BSV's collection of standard libraries. To use any of these libraries in a package you must explicitly import the library package using an `import` clause.

3.1 Storage Structures

3.1.1 Register File

Package

```
import RegFile :: * ;
```

Description

This package provides 5-read-port 1-write-port register array modules.

Note: In a design that uses RegFiles, some of the read ports may remain unused. This may generate a warning in various downstream tool. Downstream tools should be instructed to optimize away the unused ports.

Interfaces and Methods

The `RegFile` package defines one interface, `RegFile`. The `RegFile` interface provides two methods, `upd` and `sub`. The `upd` method is an `Action` method used to modify (or update) the value of an element in the register file. The `sub` method (from "sub"script) is a `Value` method which reads and returns the value of an element in the register file. The value returned is of a datatype `data_t`.

Interface Name	Parameter name	Parameter Description	Restrictions
`RegFile`	*index_type*	datatype of the index	must be in the `Bits` class
	data_t	datatype of the element values	must be in the `Bits` class

```
interface RegFile #(type index_t, type data_t);
    method Action upd(index_t addr, data_t d);
    method data_t sub(index_t addr);
endinterface: RegFile
```

Method			Arguments	
Name	Type	Description	Name	Description
`upd`	`Action`	Change or update an element within the register file.	`addr`	index of the element to be changed, with a datatype of `index_t`
			`d`	new value to be stored, with a datatype of `data_t`
`sub`	*data_t*	Read an element from the register file and return it.	`addr`	index of the element, with a datatype of `index_t`

Modules

The `RegFile` package provides three modules: `mkRegFile` creates a RegFile with registers allocated from the `lo_index` to the `hi_index`; `mkRegFileFull` creates a RegFile from the minimum index to the maximum index; and `mkRegFileWCF` creates a RegFile from `lo_index` to `hi_index` for which the reads and the write are scheduled conflict-free. There is a second set of these modules, the `RegFileLoad` variants, which take as an argument a file containing the initial contents of the array.

`mkRegFile`	Create a RegFile with registers allocated from `lo_index` to `hi_index`. `lo_index` and `hi_index` are of the `index_t` datatype and the elements are of the `data_t` datatype.

	`module mkRegFile#( index_t lo_index, index_t hi_index )`
	`( RegFile#(index_t, data_t) )`
	`provisos (Bits#(index_t, size_index),`
	`Bits#(data_t,  size_data));`

`mkRegFileFull`	Create a RegFile from min to max index where the index is of a datatype `index_t` and the elements are of datatype `data_t`. The min and max are specified by the `Bounded` typeclass instance (0 to N-1 for N-bit numbers).

	`module mkRegFileFull#( RegFile#(index_t, data_t) )`
	`provisos (Bits#(index_t, size_index),`
	`Bits#(data_t, size_data)`
	`Bounded#(index_t) );`

`mkRegFileWCF`	Create a RegFile from `lo_index` to `hi_index` for which the reads and the write are scheduled conflict-free. For the implications of this scheduling, see the documentation for `ConfigReg` (Section 3.1.2).

	`module mkRegFileWCF#( index_t lo_index, index_t hi_index )`
	`( RegFile#(index_t, data_t) )`
	`provisos (Bits#(index_t, size_index),`
	`Bits#(data_t,  size_data));`

The `RegFileLoad` variants provide the same functionality as `RegFile`, but each constructor function takes an additional file name argument. The file contains the initial contents of the array using the Verilog hex memory file syntax, which allows white spaces (including new lines, tabs, underscores, and form-feeds), comments, binary and hexadecimal numbers. Length and base format must not be specified for the numbers.

The generated Verilog for file load variants contains `$readmemb` and `$readmemh` constructs. These statements, as well as initial blocks generally, are considered simulation-only constructs because they are not supported consistently across synthesis tools. Therefore, in the generated Verilog the initial blocks are protected with a `translate_off` directive. When using a synthesis tool which supports these constructs you can remove the directives to allow the tool to processes the `$readmemh` and `$readmemb` tasks during synthesis.

`mkRegFileLoad`	Create a RegFile using the file to provide the initial contents of the array.

	`module mkRegFileLoad#`
	`( String file, index_t lo_index, index_t hi_index)`
	`( RegFile#(index_t, data_t) )`
	`provisos (Bits#(index_t, size_index),`
	`Bits#(data_t, size_data));`

<table>
<tr><td rowspan="2"><code>mkRegFileFullLoad</code></td><td>Create a RegFile from min to max index using the file to provide the initial contents of the array. The min and max are specified by the <code>Bounded</code> typeclass instance (0 to N-1 for N-bit numbers).</td></tr>
<tr><td><pre>module mkRegFileFullLoad#(String file)
                         (RegFile#(index_t, data_t))
  provisos (Bits#(index_t, size_index),
            Bits#(data_t, size_data),
            Bounded#(index_t));</pre></td></tr>
</table>

<table>
<tr><td rowspan="2"><code>mkRegFileWCFLoad</code></td><td>Create a RegFile from <code>lo_index</code> to <code>hi_index</code> for which the reads and the write are scheduled conflict-free (see Section 3.1.2), using the file to provide the initial contents of the array.</td></tr>
<tr><td><pre>module mkRegFileWCFLoad#
           (String file, index_t lo_index, index_t hi_index)
           (RegFile#(index_t, data_t))
  provisos (Bits#(index_t, size_index),
            Bits#(data_t, size_data));</pre></td></tr>
</table>

Examples

Use `mkRegFileLoad` to create Register files and then read the values.

```
Reg#(Cntr) count <- mkReg(0);

// Create Register files to use as inputs in a testbench
RegFile#(Cntr, Fp64) vecA  <- mkRegFileLoad("vec.a.txt", 0, 9);
RegFile#(Cntr, Fp64) vecB  <- mkRegFileLoad("vec.b.txt", 0, 9);

//read the values from the Register files
rule drivein (count < 10);
      Fp64 a = vecA.sub(count);
      Fp64 b = vecB.sub(count);
      uut.start(a, b);
      count <= count + 1;
endrule
```

Verilog Modules

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

BSV Module Name	Verilog Module Name	Defined in File

`mkRegFile`
`mkRegFileFull`
`mkRegFileWCF`
	RegFile	RegFile.v

`mkRegFileLoad`
`mkRegFileFullLoad`
`mkRegFileWCFLoad`
	RegFileLoad	RegFileLoad.v

3.1.2 ConfigReg

Package

```
import ConfigReg :: * ;
```

Description

The `ConfigReg` package provides a way to create registers where each update clobbers the current value, but the precise timing of updates is not important. These registers are identical to the `mkReg` registers except that their scheduling annotations allows reads and writes to occur in either order during rule execution.

Rules which fire during the clock cycle where the register is written read a stale value (that is the value from the beginning of the clock cycle) regardless of firing order and writes which have occurred during the clock cycle. Thus if rule `r1` writes to a ConfigReg `cr` and rule `r2` reads `cr` later in the same cycle, the old or stale value of `cr` is read, not the value written in `r1`. If a standard register is used instead, rule `r2`'s execution will be blocked by `r1`'s execution or the scheduler may create a different rule execution order.

The hardware implementation is identical for the more common registers (`mkReg`, `mkRegU` and `mkRegA`), and the module constructors parallel these as well.

Interfaces

The `ConfigReg` interface is an alias of the `Reg` interface (section 2.4.1).

```
typedef Reg#(a_type) ConfigReg #(type a_type);
```

Modules

The `ConfigReg` package provides three modules; `mkConfigReg` creates a register with a given reset value and synchronous reset logic, `mkConfigRegU` creates a register without any reset, and `mkConfigRegA` creates a register with a given reset value and asynchronous reset logic.

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

`mkConfigRegU`	Make a register without any reset; initial simulation value is alternating 01 bits.
	`module mkConfigRegU(Reg#(a_type))` `  provisos (Bits#(a_type, sizea));`

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

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

3.1.3 DReg

Package

```
import DReg :: * ;
```

Description

The DReg package allows a designer to create registers which store a written value for only a single clock cycle. The value written to a DReg is available to read one cycle after the write. If more than one cycle has passed since the register has been written however, the value provided by the register is instead a default value (that is specified during module instantiation). These registers are useful when wanting to send pulse values that are only asserted for a single clock cycle. The DReg is the register equivalent of a DWire 2.4.6.

Modules

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

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

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

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

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

3.1.4 RevertingVirtualReg

Package

```
import RevertingVirtualReg :: * ;
```

Description

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

Modules

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

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

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

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

Scheduling Annotations mkRevertingVirtualReg		
	read	write
read	CF	SB
write	SA	SBR

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

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

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

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

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

3.1.5 BRAM

Package

```
import BRAM :: * ;
```

Description

The `BRAM` package provides types, interfaces, and modules to support FPGA BlockRams. The BRAM modules include FIFO wrappers to provide implicit conditions for proper flow control for the BRAM latency. Specific tools may determine whether modules are mapped to appropriate BRAM cells during synthesis.

The low-level modules that implement BRAM without implicit conditions are available in the `BRAMCore` package, Section 3.1.6. Most designs should use the `BRAM` package, `BRAMCore` should only be used if you need access to low-level core BRAM modules without implicit conditions.

Types and type classes

BRAM_Configure The `BRAM_Configure` structure specifies the underlying modules and their attributes for instantiation. Default values for the BRAM are defined with the `DefaultValue` instance and can easily be modified.

BRAM_Configure Structure			
Field	Type	Description	Allowed or Recommended Values
`memorySize`	Integer	Number of words in the BRAM	
`latency`	Integer	Number of stages in the read	1 (address is registered) 2 (address and data are registered)
`loadFormat`	LoadFormat	Describes the load file	`None` `tagged Hex` *filename* `tagged Binary` *filename*
`outFIFODepth`	Integer	The depth of the BypassFIFO after the BRAM for the BRAMServer module	latency+2
`allowWriteResponseBypass`	Bool	Determines if write responses can directly be enqueued in the output fifo (`latency = 0` for write).	

The size of the BRAM is determined by the `memorySize` field given in number of words. The width of a word is determined by the polymorphic type `data` specified in the BRAM interface. If the `memorySize` field is 0, then memory size $= 2^n$, where `n` is the number of address bits determined from the address type.

The `latency` field has two valid values; `1` indicates that the address on the read is registered, `2` indicates that both the address on the read input and the data on the read output are registered. When latency = 2, the components in the dotted box in Figure 1 are included.

The `outFIFODepth` is used to determine the depth of the Bypass FIFO after the BRAM in the `mkBRAMServer` module. This value should be `latency + 2` to allow full pipeline behavior.

The `allowWriteResponseBypass` field, when `True`, specifies that the write response is issued on the same cycle as the write request. If `False`, the write reponse is pipelined, which is the same behavior as the read request. When `True`, the schedule constraints between `put` and `get` are `put SBR get`. Otherwise, the annotation is `get CF put` (no constraint).

```
typedef struct {Integer      memorySize ;
                Integer      latency ;              // 1 or 2 can extend to 3
                LoadFormat   loadFormat;            // None, Hex or Binary
                Integer      outFIFODepth;
                Bool         allowWriteResponseBypass;
               } BRAM_Configure ;
```

The `LoadFormat` defines the type of the load file (`None`, `Hex` or `Binary`). The type `None` is used when there is no load file. When the type is `Hex` or `Binary`, the name of the load file is provided as a `String`.

```
typedef union tagged {
                      void None;
                      String Hex;
                      String Binary;
   } LoadFormat
deriving (Eq, Bits);
```

The default values are defined in this package using the `DefaultValue` instance for `BRAM_Configure`. You can modify the default values by changing this instance or by modifying specific fields in your design.

Values defined in defaultValue			
Field	Type	Value	Meaning
`memorySize`	Integer	0	2^n, where `n` is the number of address bits
`latency`	Integer	1	address is registered
`outFIFODepth`	Integer	3	latency + 2
`loadFormat`	`LoadFormat`	None	no load file is used
`allowWriteResponseBypass`	Bool	False	the write response is pipelined

```
instance DefaultValue #(BRAM_Configure);
   defaultValue = BRAM_Configure {memorySize        : 0
                                 ,latency           : 1 // No output reg
                                 ,outFIFODepth      : 3
                                 ,loadFormat        : None
                                 ,allowWriteResponseBypass  : False  };
endinstance
```

To modify a default configuration for your design, set the field you want to change to the new value. Example:

```
BRAM_Configure cfg = defaultValue ;     //declare variable cfg
             cfg.memorySize = 1024*32 ; //new value for memorySize
             cfg.loadFormat = tagged Hex "ram.txt";  //value for loadFormat
BRAM2Port#(UInt#(15), Bit#(16)) bram <- mkBRAM2Server (cfg) ;
                                       //instantiate 32K x 16 bits BRAM module
```

BRAMRequest The `BRAM` package defines 2 structures for a BRAM request: `BRAMRequest`, and the byte enabled version `BRAMRequestBE`.

BRAMRequest Structure		
Field	Type	Description
write	Bool	Indicates whether this operation is a write (True) or a read (False).
responseOnWrite	Bool	Indicates whether a response should be received from this write command
address	addr	Word address of the read or write
datain	data	Data to be written. This field is ignored for reads.

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

BRAMRequestBE The structure BRAMRequestBE allows for the byte enable signal.

BRAMRequestBE Structure		
Field	Type	Description
writeen	Bit#(n)	Byte-enable indicating whether this operation is a write (n != 0) or a read (n = 0).
responseOnWrite	Bool	Indicates whether a response should be received from this write command
address	addr	Word address of the read or write
datain	data	Data to be written. This field is ignored for reads.

```
typedef struct {Bit#(n) writeen;
                Bool    responseOnWrite;
                addr    address;
                data    datain;
               } BRAMRequestBE#(type addr, type data, numeric type n) deriving (Bits, Eq);
```

Interfaces and Methods

The interfaces for the BRAM are built on the Server interface defined in the ClientServer package, Section 3.7.3. Some type aliases specific to the BRAM are defined here.

BRAM Server and Client interface types :

```
typedef Server#(BRAMRequest#(addr, data), data) BRAMServer#(type addr, type data);
typedef Client#(BRAMRequest#(addr, data), data) BRAMClient#(type addr, type data);
```

Byte-enabled BRAM Server and Client interface types:

```
typedef Server#(BRAMRequestBE#(addr, data, n), data)
        BRAMServerBE#(type addr, type data, numeric type n);
typedef Client#(BRAMRequestBE#(addr, data, n), data)
        BRAMClientBE#(type addr, type data, numeric type n);
```

The BRAM package defines 1 and 2 port interfaces, with write-enabled and byte-enabled versions. Each BRAM port interface contains a BRAMServer#(addr, data) subinterface and a clear action, which clears the output FIFO of any pending requests. The data in the BRAM is not cleared.

BRAM1Port Interface 1 Port BRAM Interface		
Name	Type	Description
portA	BRAMServer#(addr, data)	Server subinterface
portAClear	Action	Method to clear the portA output FIFO

```
interface BRAM1Port#(type addr, type data);
   interface BRAMServer#(addr, data) portA;
   method Action portAClear;
endinterface: BRAM1Port
```

BRAM1PortBE Interface Byte enabled 1 port BRAM Interface		
Name	Type	Description
portA	BRAMServerBE#(addr, data, n)	Byte-enabled server subinterface
portAClear	Action	Method to clear the portA output FIFO

```
interface BRAM1PortBE#(type addr, type data, numeric type n);
   interface BRAMServerBE#(addr, data, n) portA;
   method Action portAClear;
endinterface: BRAM1PortBE
```

BRAM2Port Interface 2 port BRAM Interface		
Name	Type	Description
portA	BRAMServer#(addr, data)	Server subinterface for port A
portB	BRAMServer#(addr, data)	Server subinterface for port B
portAClear	Action	Method to clear the port A output FIFO
portBClear	Action	Method to clear the port B output FIFO

```
interface BRAM2Port#(type addr, type data);
   interface BRAMServer#(addr, data) portA;
   interface BRAMServer#(addr, data) portB;
   method Action portAClear;
   method Action portBClear;
endinterface: BRAM2Port
```

BRAM2PortBE Interface Byte enabled 2 port BRAM Interface		
Name	Type	Description
portA	BRAMServerBE#(addr, data, n)	Byte-enabled server subinterface for port A
portB	BRAMServerBE#(addr, data, n)	Byte-enabled server subinterface for port B
portAClear	Action	Method to clear the portA output FIFO
portBClear	Action	Method to clear the portB output FIFO

```
interface BRAM2PortBE#(type addr, type data, numeric type n);
   interface BRAMServerBE#(addr, data, n) portA;
   interface BRAMServerBE#(addr, data, n) portB;
   method Action portAClear;
   method Action portBClear;
endinterface: BRAM2PortBE
```

Modules

The BRAM modules defined in the `BRAMCore` package (Section 3.1.6) are wrapped with control logic to turn the BRAM into a server, as shown in Figure 1. The BRAM Server modules include an output FIFO and logic to control its loading and to avoid overflow. A single port, single clock byte-enabled version is provided as well as 2 port and dual clock write-enabled versions.

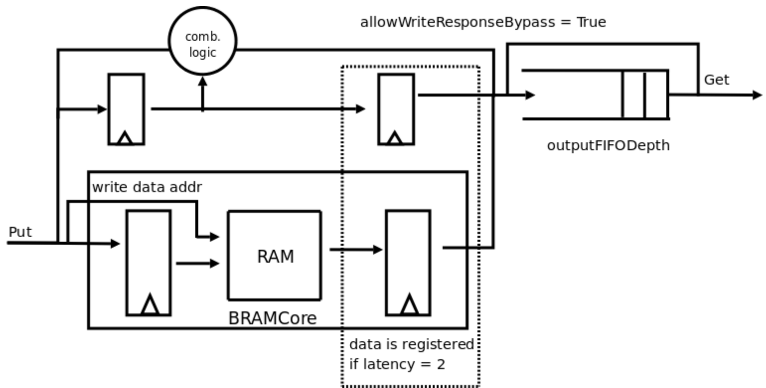

Figure 1: 1 port of a BRAM Server

`mkBRAM1Server`	BRAM Server module including an output FIFO and logic to control loading and to avoid overflow.
	`module mkBRAM1Server #( BRAM_Configure cfg ) ( BRAM1Port #(addr, data) ) provisos(Bits#(addr, addr_sz), Bits#(data, data_sz), DefaultValue#(data) );`

```
module mkBRAM1Server #( BRAM_Configure cfg )
                      ( BRAM1Port #(addr, data) )
   provisos(Bits#(addr, addr_sz),
            Bits#(data, data_sz),
            DefaultValue#(data) );
```

`mkBRAM1ServerBE`	Byte-enabled BRAM Server module.

```
module mkBRAM1ServerBE #( BRAM_Configure cfg )
                        ( BRAM1PortBE #(addr, data, n) )
   provisos(Bits#(addr, addr_sz),
            Bits#(data, data_sz),
            Div#(data_sz, n, chunk_sz),
            Mul#(chunk_sz, n, data_sz),
            DefaultValue#(data) );
```

`mkBRAM2Server`	2 port BRAM Server module.

```
module mkBRAM2Server #( BRAM_Configure cfg )
                      ( BRAM2Port #(addr, data) )
   provisos(Bits#(addr, addr_sz),
            Bits#(data, data_sz),
            DefaultValue#(data) );
```

mkBRAM2ServerBE	Byte-enabled 2 port BRAM Server module.	
	`module mkBRAM2ServerBE #( BRAM_Configure cfg ) ( BRAM2PortBE #(addr, data, n) ) provisos(Bits#(addr, addr_sz), Bits#(data, data_sz), Div#(data_sz, n, chunk_sz), Mul#(chunk_sz, n, data_sz) );`	
mkSyncBRAM2Server	2 port, dual clock, BRAM Server module. The `portA` subinterface and `portAClear` methods are in the `clkA` domain; the `portB` subinterface and `portBClear` methods are in the `clkB` domain.	
	`(* no_default_clock, no_default_reset *) module mkSyncBRAM2Server #( BRAM_Configure cfg, Clock clkA, Reset rstNA, Clock clkB, Reset rstNB ) (BRAM2Port #(addr, data) ) provisos(Bits#(addr, addr_sz), Bits#(data, data_sz), DefaultValue#(data) );`	
mkSyncBRAM2ServerBE	2 port, dual clock, byte-enabled BRAM Server module. The `portA` subinterface and `portAClear` methods are in the `clkA` domain; the `portB` subinterface and `portBClear` methods are in the `clkB` domain.	
	`(* no_default_clock, no_default_reset *) module mkSyncBRAM2ServerBE #(BRAM_Configure cfg, Clock clkA, Reset rstNA, Clock clkB, Reset rstNB ) (BRAM2PortBE #(addr, data, n)) provisos(Bits#(addr, addr_sz), Bits#(data, data_sz), Div#(data_sz, n, chunk_sz), Mul#(chunk_sz, n, data_sz) );`	

Example: Using a BRAM

```
import BRAM::*;
import StmtFSM::*;
import Clocks::*;

function BRAMRequest#(Bit#(8), Bit#(8)) makeRequest(Bool write, Bit#(8) addr, Bit#(8) data);
   return BRAMRequest{
                      write: write,
                      responseOnWrite:False,
                      address: addr,
                      datain: data
                      };
endfunction
```

```
(* synthesize *)
module sysBRAMTest();
    BRAM_Configure cfg = defaultValue;
    cfg.allowWriteResponseBypass = False;
    BRAM2Port#(Bit#(8), Bit#(8)) dut0 <- mkBRAM2Server(cfg);
    cfg.loadFormat = tagged Hex "bram2.txt";
    BRAM2Port#(Bit#(8), Bit#(8)) dut1 <- mkBRAM2Server(cfg);

   //Define StmtFSM to run tests
   Stmt test =
   (seq
       delay(10);
       ...
       action
          dut1.portA.request.put(makeRequest(False, 8'h02, 0));
          dut1.portB.request.put(makeRequest(False, 8'h03, 0));
       endaction
       action
          $display("dut1read[0] = %x", dut1.portA.response.get);
          $display("dut1read[1] = %x", dut1.portB.response.get);
       endaction
       ...
       delay(100);
    endseq);
   mkAutoFSM(test);
endmodule
```

3.1.6 BRAMCore

Package

```
import BRAMCore :: * ;
```

Description

The `BRAMCore` package, along with the `BRAM` package (Section 3.1.5) provides types, interfaces, and modules to support FPGA BlockRAMS. Specific tools may determine whether modules are mapped to appropriate BRAM cells during synthesis.

Most designs should use the the `BRAM` package instead of `BRAMCore`, as the `BRAM` package provides implicit conditions provided by FIFO wrappers. The `BRAMCore` package should be used only if you want the low-level core BRAM modules without implicit conditions.

The `BRAMCore` package contains the low-level wrappers to the BRAM Verilog and Bluesim modules. Components are provided for single and dual port, byte-enabled, loadable, and dual clock versions.

Interfaces and Methods

The `BRAMCore` package defines four variations of a BRAM interface to support single and dual port BRAMs, as well as byte-enabled BRAMs.

The `BRAM_PORT` interface declares two methods; an Action method `put`, and a value method `read`.

The `BRAM_DUAL_PORT` interface is defined as two `BRAM_PORT` subinterfaces, one for each port.

BRAM_PORT Interface				
Method			Arguments	
Name	Type	Description	Name	Description
put	Action	Read or write values in the BRAM.	write	Write enable for the port; if True the action is write, if False, the action is read.
			address	Index of the element, with a datatype of addr.
			datain	Value to be written, with a datatype of data. This value is ignored if the action is read.
read	*data*	Returns a value of type data.		

```
(* always_ready *)
interface BRAM_PORT#(type addr, type data);
   method Action put(Bool write, addr address, data datain);
   method data   read();
endinterface: BRAM_PORT

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

Byte-enabled Interfaces

The BRAM_PORT_BE and BRAM_DUAL_PORT_BE interfaces are the byte-enabled versions of the BRAM interfaces. In this version, the argument writen is of type Bit#(n), where n is the number of byte-enables. Your synthesis tools and targeted technology determine the restriction of data size and byte enable size. If $n = 0$, the action is a read.

The BRAM_DUAL_PORT_BE interface is defined as two BRAM_PORT_BE subinterfaces, one for each port.

BRAM_PORT_BE Interface				
Method			Arguments	
Name	Type	Description	Name	Description
put	Action	Read or write values in the BRAM.	writeen	Byte-enable for the port; if n != 0 write the specified bytes, if n = 0 read.
			address	Index of the elements to be read or written, with a datatype of addr.
			datain	Value to be written, with a datatype of data. This value is ignored if the action is read.
read	*data*	Returns a value of type data.		

```
(* always_ready *)
interface BRAM_PORT_BE#(type addr, type data, numeric type n);
   method Action put(Bit#(n) writeen, addr address, data datain);
   method data   read();
endinterface: BRAM_PORT_BE

interface BRAM_DUAL_PORT_BE#(type addr, type data, numeric type n);
   interface BRAM_PORT_BE#(addr, data, n) a;
   interface BRAM_PORT_BE#(addr, data, n) b;
endinterface
```

Modules

The `BRAMCore` package provides 1 and 2 port BRAM core modules, in both write-enabled and byte-enabled versions. Note that there are no implicit conditions on the methods of these modules; if these are required consider using the modules in the `BRAM` package (Section 3.1.5).

The `BRAMCore` package requires the caller to ensure the correct cycle to capture the read data, as determined by the `hasOutputRegister` flag. If `hasOutputRegister` is `True`, both the read address and the read data are registered; if `False`, only the read address is registered.

- If the output is registered (`hasOutputRegister` is `True`), the latency is 2; the read data is available 2 cycles after the request.
- If the output is not registered (`hasOutputRegister` is `False`), the latency is 1; the read data is available 1 cycle after the request.

The other argument required is `memSize`, an `Integer` specifying the memory size in number of words of type `data`.

The loadable BRAM modules require two additional arguments:

- `file` is a `String` containing the name of the load file.
- `binary` is a `Bool` indicating whether the data type of the load file is binary (`True`) or hex (`False`).

mkBRAMCore1

Single port BRAM

```
module mkBRAMCore1#(Integer memSize,
                   Bool hasOutputRegister)
                   (BRAM_PORT#(addr, data))
   provisos(Bits#(addr, addr_sz), Bits#(data, data_sz));
```

mkBRAMCore1BE

Byte-enabled, single port BRAM.

```
module mkBRAMCore1BE#(Integer memSize,
                      Bool hasOutputRegister )
                     (BRAM_PORT_BE#(addr, data, n))
   provisos(Bits#(addr, addr_sz), Bits#(data, data_sz),
            Div#(data_sz, n, chunk_sz),
            Mul#(chunk_sz, n, data_sz));
```

mkBRAMCore1Load

Loadable, single port BRAM where the initial contents are in `file`. The parameter `binary` indicates whether the contents of `file` are binary (`True`) or hex (`False`).

```
module mkBRAMCore1Load#(Integer memSize,
                        Bool hasOutputRegister,
                        String file, Bool binary )
                       (BRAM_PORT#(addr, data))
   provisos(Bits#(addr, addr_sz), Bits#(data, data_sz) );
```

`mkBRAMCore1BELoad`	Loadable, single port, byte-enabled BRAM.	

```
module mkBRAMCore1BELoad#(Integer memSize,
                          Bool hasOutputRegister,
                          String file, Bool binary)
                         (BRAM_PORT_BE#(addr, data, n))
   provisos(Bits#(addr, addr_sz), Bits#(data, data_sz),
            Div#(data_sz, n, chunk_sz),
            Mul#(chunk_sz, n, data_sz) );
```

`mkBRAMCore2`	Dual port, single clock BRAM.

```
module mkBRAMCore2#(Integer memSize,
                    Bool hasOutputRegister )
                   (BRAM_DUAL_PORT#(addr, data))
   provisos(Bits#(addr, addr_sz), Bits#(data, data_sz) );
```

`mkBRAMCore2BE`	Byte-enabled, dual port BRAM.

```
module mkBRAMCore2BE#(Integer memSize,
                      Bool hasOutputRegister
                     ) (BRAM_DUAL_PORT_BE#(addr, data, n))
   provisos(Bits#(addr, addr_sz),
            Bits#(data, data_sz),
            Div#(data_sz, n, chunk_sz),
            Mul#(chunk_sz, n, data_sz) );
```

`mkSyncBRAMCore2`	Dual port, dual clock BRAM.

```
module mkSyncBRAMCore2#(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));
```

`mkSyncBRAMCore2BE`	Dual port, dual clock byte-enabled BRAM.

```
module mkSyncBRAMCore2BE#(Integer memSize,
                          Bool hasOutputRegister,
                          Clock clkA, Reset rstNA,
                          Clock clkB, Reset rstNB)
                      (BRAM_DUAL_PORT_BE#(addr, data, n))
   provisos(Bits#(addr, addr_sz),
            Bits#(data, data_sz),
            Div#(data_sz, n, chunk_sz),
            Mul#(chunk_sz, n, data_sz) );
```

<table>
<tr><td rowspan="2"><code>mkBRAMCore2Load</code></td><td>Dual port, single clock, BRAM where the initial contents are in <code>file</code>. The parameter <code>binary</code> indicates whether the contents of <code>file</code> are binary (<code>True</code>) or hex (<code>False</code>).</td></tr>
<tr><td><pre>module mkBRAMCore2Load#(Integer memSize,
                        Bool hasOutputRegister,
                        String file, Bool binary)
                       (BRAM_DUAL_PORT#(addr, data))
   provisos(Bits#(addr, addr_sz),Bits#(data, data_sz));</pre></td></tr>
</table>

<table>
<tr><td rowspan="2"><code>mkBRAMCore2BELoad</code></td><td>Dual port, single clock, byte-enabled BRAM where the initial contents are in <code>file</code>. The parameter <code>binary</code> indicates whether the contents of <code>file</code> are binary (<code>True</code>) or hex (<code>False</code>).</td></tr>
<tr><td><pre>module mkBRAMCore2BELoad#(Integer memSize,
                          Bool hasOutputRegister,
                          String file, Bool binary)
                    (BRAM_DUAL_PORT_BE#(addr, data, n))
   provisos(Bits#(addr, addr_sz),
            Bits#(data, data_sz),
            Div#(data_sz, n, chunk_sz),
            Mul#(chunk_sz, n, data_sz));</pre></td></tr>
</table>

<table>
<tr><td rowspan="2"><code>mkSyncBRAMCore2Load</code></td><td>Dual port, dual clock BRAM with initial contents in <code>file</code>.</td></tr>
<tr><td><pre>module mkSyncBRAMCore2Load#(Integer memSize,
                        Bool hasOutputRegister,
                        Clock clkA, Reset rstNA,
                        Clock clkB, Reset rstNB,
                        String file, Bool binary)
                        (BRAM_DUAL_PORT#(addr, data))
   provisos(Bits#(addr, addr_sz), Bits#(data, data_sz));</pre></td></tr>
</table>

<table>
<tr><td rowspan="2"><code>mkSyncBRAMCore2BELoad</code></td><td>Dual port, dual clock, byte-enabledBRAM with initial contents in <code>file</code>.</td></tr>
<tr><td><pre>module mkSyncBRAMCore2BELoad#(Integer memSize,
                              Bool hasOutputRegister,
                              Clock clkA, Reset rstNA,
                              Clock clkB, Reset rstNB,
                              String file, Bool binary)
                       (BRAM_DUAL_PORT_BE#(addr, data, n))
   provisos(Bits#(addr, addr_sz),
            Bits#(data, data_sz),
            Div#(data_sz, n, chunk_sz),
            Mul#(chunk_sz, n, data_sz));</pre></td></tr>
</table>

Verilog Modules

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

BSV Module Name	Verilog Module Names
`mkBRAMCore1`	`BRAM1.v`
`mkBRAMCore1Load`	`BRAM1Load.v`
`mkBRAMCore1BE`	`BRAM1BE.v`
`mkBRAMCore1BELoad`	`BRAM1BELoad.v`
`mkBRAMCore2` `mkSyncBRAMCore2`	`BRAM2.v`
`mkBRAMCore2BE` `mkSyncBRAMCore2BE`	`BRAM2BE.v`
`mkBRAMCore2Load` `mkSyncBRAMCore2Load`	`BRAM2Load.v`
`mkBRAMCore2BELoad` `mkSyncBRAMCore2BELoad`	`BRAM2BELoad.v`

3.2 FIFOs

3.2.1 FIFO Overview

The BSC library contains multiple FIFO packages.

Package Name	Description	BSV Source provided	Section
FIFO	Defines the FIFO interface and module constructors. FIFOs provided have implicit full and empty signals. Includes pipeline FIFO (`mkLFIFO`).		3.2.2
FIFOF	Defines the FIFOF interface and module constructors. FIFOs provided have explicit full and empty signals. Includes pipeline FIFOF (`mkLFIFOF`).		3.2.2
FIFOLevel	Enhanced FIFO interfaces and modules which include methods to indicate the level or current number of items stored in the FIFO. Single and dual clock versions are provided.		3.2.3
BRAMFIFO	FIFOs which utilize the Xilinx Block RAMs.	√	3.2.4
SpecialFIFOs	Additional pipeline and bypass FIFOs	√	3.2.5
AlignedFIFOs	Parameterized FIFO module for creating synchronizing FIFOs between clock domains with aligned edges.	√	3.2.6
Gearbox	FIFOs which change the frequency and data width of data across clock domains with aligned edges. The overall data rate stays the samme.	√	3.2.7
Clocks	Generalized FIFOs to synchronize data being sent across clock domains		3.9.7

3.2.2 FIFO and FIFOF packages

Packages

```
import FIFO :: * ;
import FIFOF :: * ;
```

Description

The FIFO package defines the FIFO interface and four module constructors. The FIFO package is for FIFOs with implicit full and empty signals.

The FIFOF package defines FIFOs with explicit full and empty signals. The standard version of FIFOF has FIFOs with the enq, deq and first methods guarded by the appropriate (notFull or notEmpty) implicit conditions for safety and improved scheduling. Unguarded (UG) versions of FIFOF are available for the rare cases when implicit conditions are not desired. Guarded (G) versions of FIFOF are available which allow more control over implicit conditions. With the guarded versions the user can specify whether the enqueue or dequeue side is guarded.

Type classes

FShow The FIFOF type belongs to the FShow type class. A FIFOF can be turned into a Fmt type. The Fmt value returned depends on the values of the notEmpty and notFull methods.

FShow values for FIFOF			
notEmpty	notFull	Fmt Object	Example
True	True	<first>	3
True	False	<first> FULL	2 FULL
False	True	EMPTY	EMPTY
False	False	EMPTY	EMPTY
Note: <first> is the value of the first entry with the fshow function applied			

Interfaces and methods

The four common methods, enq, deq, first and clear are provided by both the FIFO and FIFOF interfaces.

FIFO methods				
Method			Argument	
Name	Type	Description	Name	Description
enq	Action	adds an entry to the FIFO	x1	variable to be added to the FIFO must be of type *element_type*
deq	Action	removes first entry from the FIFO		
first	*element_type*	returns first entry		the entry returned is of *element_type*
clear	Action	clears all entries from the FIFO		

```
interface FIFO #(type element_type);
    method Action enq(element_type x1);
    method Action deq();
    method element_type first();
    method Action clear();
endinterface: FIFO
```

FIFOF provides two additional methods, notFull and notEmpty.

Additional FIFOF Methods		
Name	Type	Description
notFull	Bool	returns a True value if there is space, you can enqueue an entry into the FIFO
notEmpty	Bool	returns a True value if there are elements the FIFO, you can dequeue from the FIFO

```
interface FIFOF #(type element_type);
    method Action enq(element_type x1);
    method Action deq();
    method element_type first();
    method Bool notFull();
    method Bool notEmpty();
    method Action clear();
endinterface: FIFOF
```

The `FIFO` and `FIFOF` interfaces belong to the `ToGet` and `ToPut` typeclasses. You can use the `toGet` and `toPut` functions to convert `FIFO` and `FIFOF` interfaces to `Get` and `Put` interfaces (Section 3.7.1).

Modules

The `FIFO` and `FIFOF` interface types are provided by the module constructors: `mkFIFO`, `mkFIFO1`, `mkSizedFIFO`, `mkDepthParamFIFO`, and `mkLFIFO`. Each `FIFO` is safe with implicit conditions; they do not allow an `enq` when the `FIFO` is full or a `deq` or `first` when the `FIFO` is empty.

Most FIFOs do not allow simultaneous enqueue and dequeue operations when the FIFO is full or empty. The exceptions are pipeline and bypass FIFOs. A pipeline FIFO (provided as `mkLFIFO` in this package), allows simultaneous enqueue and dequeue operations when full. A bypass FIFO allows simultaneous enqueue and dequeue operations when empty. Additional pipeline and bypass FIFOs are provided in the `SpecialFIFOs` package (Section 3.2.5). The FIFOs in the `SpecialFIFOs` package are provided as both compiled code and BSV source code, so they are customizable.

Allowed Simultaneous enq and deq by FIFO type			
	FIFO Condition		
FIFO type	empty	not empty not full	full
`mkFIFO` `mkFIFOF`		√	
`mkFIFO1` `mkFIFOF1`		NA	
`mkLFIFO` `mkLFIFOF`		√	√
`mkLFIFO1` `mkLFIFOF1`		NA	√
Modules provided in SpecialFIFOs package 3.2.5			
`mkPipelineFIFO` `mkPipelineFIFOF`		NA	√
`mkBypassFIFO` `mkBypassFIFOF`	√	NA	
`mkSizedBypassFIFOF`	√	√	
`mkBypassFIFOLevel`	√	√	

For creating a `FIFOF` interface (providing explicit `notFull` and `notEmpty` methods) use the `"F"` version of the module, for example use `mkFIFOF` instead of `mkFIFO`.

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

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

`FIFO` or `FIFOF` of given depth n	
`mkSizedFIFO` `mkSizedFIFOF`	`module mkSizedFIFO#(Integer n)(FIFO#(element_type))` `   provisos (Bits#(element_type, width_any));`

`FIFO` or `FIFOF` of given depth n where n is a Verilog parameter or computed from compile-time constants and Verilog parameters.	
`mkDepthParamFIFO` `mkDepthParamFIFOF`	`module mkDepthParamFIFO#(UInt#(32) n)(FIFO#(element_type))` `  provisos (Bits#(element_type, width_any));`

Unguarded (UG) versions of `FIFOF` are available for the rare cases when implicit conditions are not desired. When using an unguarded FIFO, the implicit conditions for correct FIFO operations are NOT considered during rule and method processing, making it possible to enqueue when full and to dequeue when empty. These modules provide the `FIFOF` interface.

Unguarded FIFOF of depth 2	
`mkUGFIFOF`	`module mkUGFIFOF#(FIFOF#(element_type))` `   provisos (Bits#(element_type, width_any));`

Unguarded `FIFOF` of depth 1	
`mkUGFIFOF1`	`module mkUGFIFO1#(FIFOF#(element_type))` `   provisos (Bits#(element_type, width_any));`

Unguarded `FIFOF` of given depth n	
`mkUGSizedFIFOF`	`module mkUGSizedFIFOF#(Integer n)(FIFOF#(element_type))` `   provisos (Bits#(element_type, width_any));`

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

mkUGDepthParamFIFOF	`module mkUGDepthParamFIFOF#(UInt#(32) n)`
	`(FIFOF#(element_type))`
	`provisos (Bits#(element_type, width_any));`

The guarded (G) versions of each of the FIFOFs allow you to specify which implicit condition you want to guard. These modules takes two Boolean parameters; `ugenq` and `ugdeq`. Setting either parameter `TRUE` indicates the relevant methods (`enq` for `ugenq`, `first` and `deq` for `ugdeq`) are unguarded. If both are `TRUE` the FIFOF behaves the same as an unguarded FIFOF. If both are `FALSE` the behavior is the same as a regular FIFOF.

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

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

Guarded FIFOF of given depth n	

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

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

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

The LFIFOs (pipeline FIFOs) allow `enq` and `deq` in the same clock cycle when the FIFO is full. Additional BSV versions of the pipeline FIFO and also bypass FIFOs (allowing simultaneous `enq` and `deq` when the FIFO is empty) are provided in the `SpecialFIFOs` package (Section 3.2.5). Both unguarded and guarded versions of the LFIFO are provided in the FIFOF package.

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

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

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

Functions

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

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

Example using the FIFO package

This example creates 2 input FIFOs and moves data from the input FIFOs to the output FIFOs.

```
import FIFO::*;

typedef Bit#(24) DataT;

// define a single interface into our example block
interface BlockIFC;
   method Action push1 (DataT a);
   method Action push2 (DataT a);
   method ActionValue#(DataT) get();
endinterface

module mkBlock1( BlockIFC  );
   Integer fifo_depth = 16;

   // create the first inbound FIFO instance
   FIFO#(DataT) inbound1 <- mkSizedFIFO(fifo_depth);

   // create the second inbound FIFO instance
   FIFO#(DataT) inbound2 <- mkSizedFIFO(fifo_depth);

   // create the outbound instance
   FIFO#(DataT) outbound <- mkSizedFIFO(fifo_depth);

   // rule for enqueue of outbound from inbound1
```

```
      // implicit conditions ensure correct behavior
      rule enq1 (True);
         DataT in_data = inbound1.first;
         DataT out_data = in_data;
         outbound.enq(out_data);
         inbound1.deq;
      endrule: enq1

      // rule for enqueue of outbound from inbound2
      // implicit conditions ensure correct behavior
      rule enq2 (True);
         DataT in_data = inbound2.first;
         DataT out_data = in_data;
         outbound.enq(out_data);
         inbound2.deq;
      endrule: enq2

      //Add an entry  to the inbound1 FIFO
      method Action push1 (DataT a);
            inbound1.enq(a);
      endmethod

      //Add an entry  to the inbound2 FIFO
      method Action push2 (DataT a);
            inbound2.enq(a);
      endmethod

      //Remove first value from outbound and return it
      method ActionValue#(DataT) get();
            outbound.deq();
            return outbound.first();
      endmethod
   endmodule
```

Scheduling Annotations

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

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

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

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

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

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

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

Verilog Modules

`FIFO` and `FIFOF` modules correspond to the following Verilog modules, which are found in the BSC Verilog library, `$BLUESPECDIR/Verilog/`.

BSV Module Name	Verilog Module Names		Comments

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

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

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

mkDepthParamFIFOF mkUGDepthParamFIFOF mkGDepthParamFIFOF	SizedFIFO.v SizedFIFO0.v	

mkLFIFO mkLFIFOF mkUGLFIFOF mkGLFIFOF	FIFOL1.v FIFOL10.v	

3.2.3 FIFOLevel

Package

```
import FIFOLevel :: * ;
```

Description

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

Interfaces and methods

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

Interface Name	Parameter name	Parameter Description	Requirements of modules implementing the ifc
`FIFOLevelIfc`	*element_type*	type of the elements stored in the `FIFO`	must be in `Bits` class
	fifoDepth	the depth of the `FIFO`	must be `numeric` type and >2
`FIFOCountIfc`	*element_type*	type of the elements stored in the `FIFO`	must be in `Bits` class
	fifoDepth	the depth of the `FIFO`	must be `numeric` type and >2
`SyncFIFOLevelIfc`	*element_type*	type of the elements stored in the `FIFO`	must be in `Bits` class
	fifoDepth	the depth of the `FIFO`	must be `numeric` type and must be a power of 2 and >=2
`SyncFIFOCountIfc`	*element_type*	type of the elements stored in the `FIFO`	must be in `Bits` class
	fifoDepth	the depth of the `FIFO`	must be `numeric` type and must be a power of 2 and >=2

In addition to common FIFO methods, the `FIFOLevelIfc` interface defines methods to compare the current level to `Integer` constants. See Section 3.2.2 for details on `enq`, `deq`, `first`, `clear`, `notFull`, and `notEmpty`. Note that `FIFOLevelIfc` interface has a type parameter for the `fifoDepth`. This numeric type parameter is needed, since the width of the counter is dependent on the FIFO depth. The `fifoDepth` parameter must be > 2.

FIFOLevelIfc				
Method			Argument	
Name	Type	Description	Name	Description
`isLessThan`	Bool	Returns `True` if the depth of the `FIFO` is less than the `Integer` constant, `c1`.	c1	an `Integer` compile-time constant
`isGreaterThan`	Bool	Returns `True` if the depth of the `FIFO` is greater than the `Integer` constant, `c1`.	c1	an `Integer` compile-time constant

```
interface FIFOLevelIfc#( type element_type, numeric type fifoDepth ) ;
   method Action enq( element_type x1 );
   method Action deq();
   method element_type first();
   method Action clear();

   method Bool notFull ;
   method Bool notEmpty ;

   method Bool isLessThan   ( Integer c1 ) ;
   method Bool isGreaterThan( Integer c1 ) ;
endinterface
```

In addition to common FIFO methods, the `FIFOCountIfc` interface defines a method to return the current number of elements as an bit-vector. See Section 3.2.2 for details on `enq`, `deq`, `first`, `clear`, `notFull`, and `notEmpty`. Note that the `FIFOCountIfc` interface has a type parameter for the `fifoDepth`. This numeric type parameter is needed, since the width of the counter is dependent on the FIFO depth. The `fifoDepth` parameter must be > 2.

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

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

   method Bool notFull ;
   method Bool notEmpty ;

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

   method Action clear;
endinterface
```

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

Common Dual Clock Methods		
Name	Type	Description
`sNotFull`	Bool	Returns `True` if the `FIFO` appears as not full from the source side clock.
`sNotEmpty`	Bool	Returns `True` if the `FIFO` appears as not empty from the source side clock.
`dNotFull`	Bool	Returns `True` if the `FIFO` appears as not full from the destination side clock.
`dNotEmpty`	Bool	Returns `True` if the `FIFO` appears as not empty from the destination side clock.
`sClear`	Action	Clears the FIFO from the source side.
`dClear`	Action	Clears the FIFO from the destination side.

In addition to common FIFO methods (Section 3.2.2) and the common dual clock methods above, the `SyncFIFOLevelIfc` interface defines methods to compare the current level to `Integer` constants. Methods are provided for both the source (enqueue side) and destination (dequeue side) clock domains.

SyncFIFOLevelIfc Methods				
Method			Argument	
Name	Type	Description	Name	Description
`sIsLessThan`	Bool	Returns `True` if the depth of the `FIFO`, as appears on the source side clock, is less than the `Integer` constant, `c1`.	c1	an `Integer` compile-time constant
`sIsGreaterThan`	Bool	Returns `True` if the depth of the `FIFO`, as it appears on the source side clock, is greater than the `Integer` constant, `c1`.	c1	an `Integer` compile-time constant.
`dIsLessThan`	Bool	Returns `True` if the depth of the `FIFO`, as it appears on the destination side clock, is less than the `Integer` constant, `c1`.	c1	an `Integer` compile-time constant
`dIsGreaterThan`	Bool	Returns `True` if the depth of the `FIFO`, as appears on the destination side clock, is greater than the `Integer` constant, `c1`.	c1	an `Integer` compile-time constant.

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

   method Bool sNotFull ;
   method Bool sNotEmpty ;
```

```
   method Bool dNotFull ;
   method Bool dNotEmpty ;

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

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

In addition to common FIFO methods (Section 3.2.2) and the common dual clock methods above, the `SyncFIFOCountIfc` interface defines methods to return the current number of elements. Methods are provided for both the source (enqueue side) and destination (dequeue side) clock domains.

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

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

   method Bool sNotFull ;
   method Bool sNotEmpty ;
   method Bool dNotFull ;
   method Bool dNotEmpty ;

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

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

The `FIFOLevelIFC`, `SyncFIFOLevelIfc`, `FIFOCountIfc`, and `SyncFIFOCountIfc` interfaces belong to the `ToGet` and `ToPut` typeclasses. You can use the `toGet` and `toPut` functions to convert these interfaces to `Get` and `Put` interfaces (Section 3.7.1).

Modules

The module `mkFIFOLevel` provides the `FIFOLevelIfc` interface. Note that the implementation allows any number of `isLessThan` and `isGreaterThan` method calls. Each call with a unique argument adds an additional comparator to the design.

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

<table>
<tr><th>Module Name</th><th>BSV Module Declaration</th></tr>
<tr><td rowspan="2"><code>mkFIFOLevel</code></td><td><pre>module mkFIFOLevel (
          FIFOLevelIfc#(element_type, fifoDepth))
   provisos(Bits#(element_type, width_element)
             Log#(TAdd#(fifoDepth,1),cntSize)) ;</pre></td></tr>
<tr><td>Comment: <code>width_element</code> may be 0</td></tr>
</table>

<table>
<tr><th>Module Name</th><th>BSV Module Declaration</th></tr>
<tr><td rowspan="2"><code>mkGFIFOLevel</code></td><td><pre>module mkGFIFOLevel#(Bool ugenq, Bool ugdeq, Bool ugcount)
           (FIFOLevelIfc#(element_type, fifoDepth))
   provisos(Bits#(element_type, width_element),
            Log#(TAdd#(fifoDepth,1),cntSize));</pre></td></tr>
<tr><td>Comment: <code>width_element</code> may be 0</td></tr>
</table>

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

<table>
<tr><th>Module Name</th><th>BSV Module Declaration</th></tr>
<tr><td rowspan="2"><code>mkFIFOCount</code></td><td><pre>module mkFIFOCount(
          FIFOCountIfc#(element_type, fifoDepth) ifc)
   provisos (Bits#(element_type, width_element));</pre></td></tr>
<tr><td>Comment: <code>width_element</code> may be 0</td></tr>
</table>

<table>
<tr><th>Module Name</th><th>BSV Module Declaration</th></tr>
<tr><td rowspan="2"><code>mkGFIFOCount</code></td><td><pre>module mkGFIFOCount#(Bool ugenq, Bool ugdeq, Bool ugcount)
         (FIFOCountIfc#(element_type, fifoDepth) ifc)
   provisos (Bits#(element_type, width_element));</pre></td></tr>
<tr><td>Comment: <code>width_element</code> may be 0</td></tr>
</table>

The modules `mkSyncFIFOLevel` and `mkSyncFIFOCount` are dual clock FIFOs, where enqueue and dequeue methods are in separate clocks domains, `sClkIn` and `dClkIn` respectively. Because of the synchronization latency, the flag indicators will not necessarily be identical between the source and the destination clocks. Note however, that the `sNotFull` and `dNotEmpty` flags always give proper (pessimistic) indications for the safe use of `enq` and `deq` methods; these are automatically included as implicit condition in the `enq` and `deq` (and `first`) methods.

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

Module Name	BSV Module Declaration
`mkSyncFIFOLevel`	`module mkSyncFIFOLevel( Clock sClkIn, Reset sRstIn, Clock dClkIn, SyncFIFOLevelIfc#(element_type, fifoDepth) ifc ) provisos( Bits#(element_type, width_element), Log#(TAdd#(fifoDepth,1),cntSize));`
	Comment: `width_element` may be 0

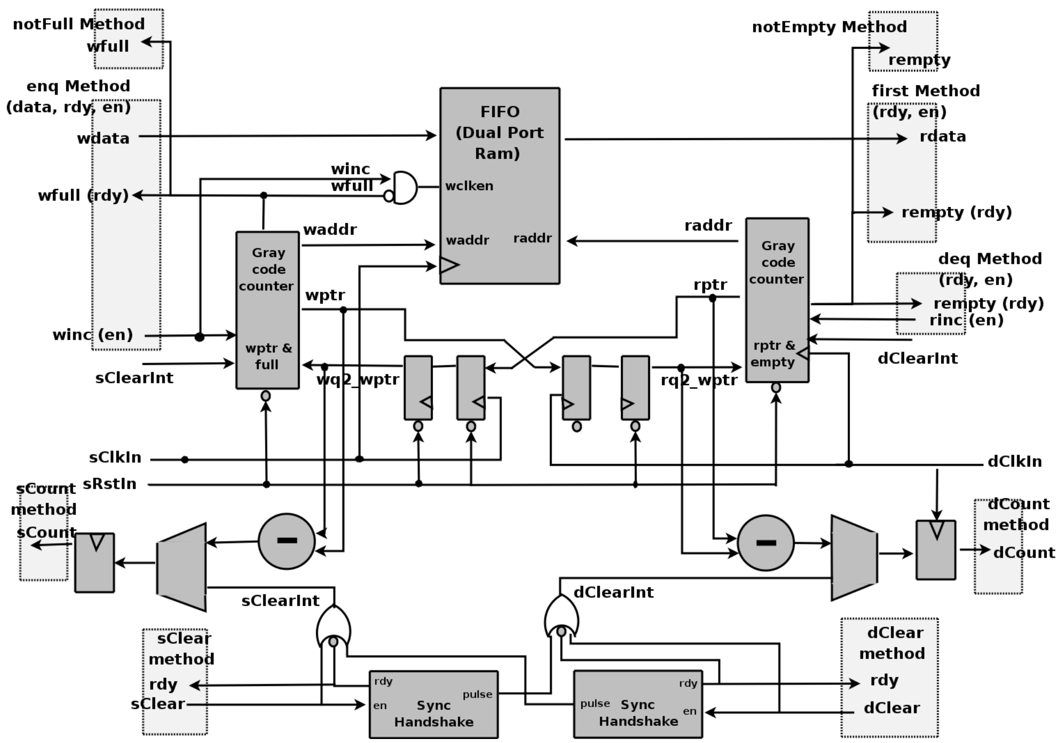

Figure 2: SyncFIFOCount

The module `mkSyncFIFOCount`, as shown in Figure 2 provides the `SyncFIFOCountIfc` interface. Because of the synchronization latency, the count reports may be different between the source and the destination clocks. Note however, that the `sCount` and `dCount` reports give pessimistic values with the appropriate side. That is, the count `sCount` (on the enqueue clock) will report the exact count of items in the FIFO or a larger count. The larger number is due to the synchronization delay in observing the dequeue action. Likewise, the `dCount` (on the dequeue clock) returns the exact count or a smaller count. The maximum disparity between `sCount` and `dCount` depends on the difference in clock periods between the source and destination clocks.

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

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

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

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

Module Name	BSV Module Declaration

`mkSyncFIFOCount`	`module mkSyncFIFOCount(`
	`          Clock sClkIn, Reset sRstIn,`
	`          Clock dClkIn,`
	`          SyncFIFOCountIfc#(element_type, fifoDepth) ifc )`
	`   provisos( Bits#(element_type, width_element));`

	Comment: `width_element` may be 0

Example

The following example shows the use of `SyncFIFOLevel` as a way to collect data into a FIFO, and then send it out in a burst mode. The portion of the design shown, waits until the FIFO is almost full, and then sets a register, `burstOut` which indicates that the FIFO should dequeue. When the FIFO is almost empty, the flag is cleared, and FIFO fills again.

```
. . .
// Define a fifo of Int(#23) with 128 entries
SyncFIFOLevelIfc#(Int#(23),128) fifo <- mkSyncFIFOLevel(sclk, rst, dclk ) ;

// Define some constants
let sFifoAlmostFull = fifo.sIsGreaterThan( 120 ) ;
let dFifoAlmostFull = fifo.dIsGreaterThan( 120 ) ;
let dFifoAlmostEmpty = fifo.dIsLessThan( 12 ) ;

// a register to indicate a burst mode
Reg#(Bool)  burstOut <- mkReg( False, clocked_by (dclk)) ;

. . .
// Set and clear the burst mode depending on fifo status
rule timeToDeque( dFifoAlmostFull && ! burstOut ) ;
   burstOut <= True ;
endrule


rule moveData ( burstOut ) ;
   let dataToSend = fifo.first ;
   fifo.deq ;
   ...
   burstOut <= !dFifoAlmostEmpty;

endrule
```

Scheduling Annotations

Scheduling constraints describe how methods interact within the schedule. The annotations for `mkFIFOLevel` and `mkSyncFIFOLevel` are the same, except that methods in different domains (source and destination) are always conflict free.

Scheduling Annotations `mkFIFOLevel`, `mkSyncFIFOLevel`								
	enq	first	deq	clear	notFull	notEmpty	isLessThan	isGreaterThan
enq	C	CF	CF	SB	SA	SA	SA	SA
first	CF	CF	SB	SB	CF	CF	CF	CF
deq	CF	SA	C	SB	SA	SA	SA	SA
clear	SA	SA	SA	SBR	SA	SA	SA	SA
notFull	SB	CF	SB	SB	CF	CF	CF	CF
notEmpty	SB	CF	SB	SB	CF	CF	CF	CF
isLessThan	SB	CF	SB	SB	CF	CF	CF	CF
isGreaterThan	SB	CF	SB	SB	CF	CF	CF	CF

The annotations for `mkFIFOCount` and `mkSyncFIFOCount` are the same, except that methods in different domains (source and destination) are always conflict free.

Scheduling Annotations `mkFIFOCount`, `mkSyncFIFOCount`							
	enq	first	deq	clear	notFull	notEmpty	count
enq	C	CF	CF	SB	SA	SA	SA
first	CF	CF	SB	SB	CF	CF	CF
deq	CF	SA	C	SB	SA	SA	SA
clear	SA	SA	SA	SBR	SA	SA	SA
notFull	SB	CF	SB	SB	CF	CF	CF
notEmpty	SB	CF	SB	SB	CF	CF	CF
count	SB	CF	SB	SB	CF	CF	CF

Verilog Modules

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

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

3.2.4 BRAMFIFO

Package

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

Description

The `BRAMFIFO` package provides FIFO interfaces and are built around a BRAM memory. The BRAM is provided in the `BRAMCore` package described in Section 3.1.6.

Interfaces

The `BRAMFIFO` package provides `FIFOF`, `FIFO`, and `SyncFIFOIfc` interfaces, as defined in the `FIFOF`, `FIFO`, (both in Section 3.2.2) and `Clocks` (Section 3.9.7) packages.

Modules

<table>
<tr><td rowspan="2">mkSizedBRAMFIFOF</td><td>Provides a <code>FIFOF</code> interface of a given depth, <code>n</code>.</td></tr>
<tr><td><pre>module mkSizedBRAMFIFOF#(Integer n) (FIFOF#(element_type))
   provisos (Bits(element_type, width_any),
             Add#(1,z,width_any));</pre></td></tr>
</table>

<table>
<tr><td rowspan="2">mkSizedBRAMFIFO</td><td>Provides a <code>FIFO</code> interface of a given depth, <code>n</code>.</td></tr>
<tr><td><pre>module mkSizedBRAMFIFO#(Integer n)(FIFO#(element_type))
   provisos(Bits#(t, width_element),
            Add#(1, z, width_element));</pre></td></tr>
</table>

<table>
<tr><td rowspan="2">mkSyncBRAMFIFO</td><td>Provides a <code>SyncFIFOIfc</code> interface to send data across clock domains. The <code>enq</code> method is in the source <code>sClkIn</code> domain, while the <code>deq</code> and <code>first</code> methods are in the destination <code>dClkIn</code> domain. The input and output clocks, along with the input and output resets, are explicitly provided. The default clock and reset are ignored.</td></tr>
<tr><td><pre>module mkSyncBRAMFIFO#(Integer depth,
                       Clock sClkIn, Reset sRstIn,
                       Clock dClkIn, Reset dRstIn)
                       (SyncFIFOIfc#(element_type))
   provisos(Bits#(element_type, width_element),
            Add#(1, z, width_element));</pre></td></tr>
</table>

<table>
<tr><td rowspan="2">mkSyncBRAMFIFOToCC</td><td>Provides a <code>SyncFIFOIfc</code> interface to send data from a second clock domain into the current clock domain. The output clock and reset are the current clock and reset.</td></tr>
<tr><td><pre>module mkSyncBRAMFIFOToCC#(Integer depth,
                           Clock sClkIn, Reset sRstIn)
                           (SyncFIFOIfc#(element_type))
   provisos(Bits#(element_type, width_element),
            Add#(1, z, width_element));</pre></td></tr>
</table>

`mkSyncBRAMFIFOFromCC`	Provides a `SyncFIFOIfc` interface to send data from the current clock domain into a second clock domain. The input clock and reset are the current clock and reset.
	`module mkSyncBRAMFIFOFromCC#(Integer depth, Clock dClkIn, Reset dRstIn) (SyncFIFOIfc#(element_type)) provisos(Bits#(element_type, width_element), Add#(1, z, width_element));`

3.2.5 SpecialFIFOs

Package

```
import SpecialFIFOs :: * ;
```

Description

The SpecialFIFOs package contains various FIFOs provided as BSV source code, allowing users to easily modify them to their own specifications. Included in the SpecialFIFOs package are pipeline and bypass FIFOs. The pipeline FIFOs are equivalent to the `mkLFIFO` (Section 3.2.2); they allow simultaneous enqueue and dequeue operations in the same clock cycle when *full*. The bypass FIFOs allow simultaneous enqueue and dequeue in the same clock cycle when *empty*. FIFOF versions, with explicit full and empty signals, are provided for both pipeline and bypass FIFOs. The package also includes the `DFIFOF`, a FIFOF with unguarded dequeue and first methods (thus they have no implicit conditions).

FIFOs in Special FIFOs package		
Module name	Interface	Description
`mkPipelineFIFO`	`FIFO`	1 element pipeline FIFO; can `enq` and `deq` simultaneously when full.
`mkPipelineFIFOF`	`FIFOF`	1 element pipeline FIFO with explicit full and empty signals.
`mkBypassFIFO`	`FIFO`	1 element bypass FIFO; can `enq` and `deq` simultaneously when empty.
`mkBypassFIFOF`	`FIFOF`	1 element bypass FIFO with explicit full and empty signals.
`mkSizedBypassFIFOF`	`FIFOF`	Bypass FIFO of given depth, with explicit full and empty signals.
`mkBypassFIFOLevel`	`FIFOLevelIfc`	Same as a `FIFOLevel` (Section 3.2.3), but can `enq` and `deq` when empty.
`mkDFIFOF`	`FIFOF`	A FIFOF with unguarded `deq` and `first` methods where the `first` method returns specified default value when the FIFO is empty.

Allowed Simultaneous enq and deq by FIFO type			
FIFO type	FIFO Condition		
	empty	not empty not full	full
`mkPipelineFIFO` `mkPipelineFIFOF`		NA	√
`mkBypassFIFO` `mkBypassFIFOF`	√	NA	
`mkSizedBypassFIFOF`	√	√	
`mkBypassFIFOLevel`	√	√	
`mkDFIFOF`	√	√	√

Interfaces and methods

The modules defined in the `SpecialFIFOs` package provide the `FIFO`, `FIFOF`, and `FIFOLevelIfc` interfaces, as shown in the table above. These interfaces are described in Section 3.2.2 (FIFO package) and Section 3.2.3 (FIFOLevel package).

Modules

Module Name	BSV Module Declaration
1-element pipeline FIFO; can `enq` and `deq` simultaneously when full.	
`mkPipelineFIFO`	`module mkPipelineFIFO (FIFO#(element_type))` `   provisos (Bits#(element_type, width_any));`

1-element pipeline FIFOF; can `enq` and `deq` simultaneously when full. Has explicit full and empty signals.	
`mkPipelineFIFOF`	`module mkPipelineFIFOF (FIFOF#(element_type))` `   provisos (Bits#(element_type, width_any));`

1-element bypass FIFO; can `enq` and `deq` simultaneously when empty.	
`mkBypassFIFO`	`module mkBypassFIFO (FIFO#(element_type))` `   provisos (Bits#(element_type, width_any));`

1-element bypass FIFOF; can `enq` and `deq` simultaneously when empty. Has explicit full and empty signals.	
`mkBypassFIFOF`	`module mkBypassFIFOF (FIFOF#(element_type))` `   provisos (Bits#(element_type, width_any));`

Bypass FIFOF of given depth `fifoDepth` with explicit full and empty signals.	

`mkSizedBypassFIFOF`	`module mkSizedBypassFIFOF#(Integer fifoDepth)`
	`(FIFOF#(element_type))`
	`provisos (Bits#(element_type, width_any));`

Bypass FIFOLevel of given depth `fifoDepth`	

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

A FIFOF with unguarded `deq` and `first` methods (thus they have no implicit conditions). The `first` method returns a specified default value when the FIFO is empty	

`mkDFIFOF`	`module mkDFIFOF#(element_type default_value)`
	`(FIFOF#(element_type))`
	`provisos (Bits#(element_type, width_any));`

3.2.6 AlignedFIFOs

Package

```
import AlignedFIFOs :: * ;
```

Description

The AlignedFIFOs package contains a parameterized FIFO module intended for creating synchronizing FIFOs between clock domains with aligned edges for both types of clock domain crossings:

- slow-to-fast crossing - every edge in the source domain implies the existence of a simultaneous edge in the destination domain
- fast-to-slow crossing - every edge in the destination domain implies the existence of a simultaneous edge in the source domain

The FIFO is parameterized on the type of store used to hold the FIFO data, which is itself parameterized on the index type, value type, and read latency. Modules to construct stores based on a single register, a vector of registers and a BRAM are provided, and the user can supply their own store implementation as well.

The FIFO allows the user to control whether or not outputs are held stable during the full slow clock cycle or allowed to transition mid-cycle. Holding the outputs stable is the safest option but it slightly increases the minimum latency through the FIFO.

A primary design goal of this FIFO is to provide an efficient and flexible family of synchronizing FIFOs between aligned clock domains which are written in BSV and are fully compatible with Bluesim. These FIFOs (particularly ones using vectors of registers) may not be the best choice for ASIC synthesis due to the muxing to select the head value in the `first` method.

Interfaces and methods

Store Interface The `AlignedFIFO` is parameterized on the type of store used to hold the FIFO data. The three types of stores provided in the `AlignedFIFO` package (single-element, vector-of-registers, and BRAM) all return a `Store` interface.

The `Store` interface has a `prefetch` method which is used by some modules (the `mkBRAMStore` in this package). If a prefetch is used, the `read` method returns the value at the previously fetched index; the value of `idx` should be ignored. If a prefetch is not used, the `read` method index value determines the returned value.

Store Interface Methods		
Name	Type	Description
`write`	Action	Writes the value at index `idx`.
`prefetch`	Action	Initiates a prefetch of the value at index `idx`.
`read`	`a`	Returns the value of type `a`. If prefetch is not used, returns the value at index `idx`. When prefetch is used, returns the value at the previously fetched index; the value of `idx` should be ignored.

```
interface Store#(type i, type a, numeric type n);
   method Action write(i idx, a value);
   method Action prefetch(i idx);
   method a read(i idx);
endinterface: Store
```

AlignedFIFO Interface The `AlignedFIFO` interface provides methods for both source (enqueue) and destination (dequeue) clock domains.

AlignedFIFO Interface Methods		
Name	Type	Description
`enq`	Action	Adds an entry to the FIFO from the source clock domain.
`first`	`a`	Returns the first entry from the FIFO in the destination clock domain.
`deq`	Action	Removes the first entry from the FIFO in the destination clock domain.
`dNotFull`	Bool	Returns `True` if the FIFO appears not full from the destination clock domain.
`dNotEmpty`	Bool	Returns `True` if the FIFO appears not empty from the destination clock domain.
`sNotFull`	Bool	Returns `True` if the FIFO appears not full from the source clock domain.
`sNotEmpty`	Bool	Returns `True` if the FIFO appears not empty from the source clock domain.
`dClear`	Action	Clears the FIFO from the destination side.
`sClear`	Action	Clears the FIFO from the source side.

```
interface AlignedFIFO#(type a);
   method Action enq(a x);
   method a first();
   method Action deq();
   method Bool dNotFull();
   method Bool dNotEmpty();
   method Bool sNotFull();
```

```
   method Bool sNotEmpty();
   method Action dClear();
   method Action sClear();
endinterface: AlignedFIFO
```

Modules

The `AlignedFIFO` module is parameterized on the type of store used to hold the FIFO data. The `AlignedFIFOs` package contains modules to construct stores based on a single register (`mkRegStore`), a vector of registers (`mkRegVectorStore`), and a BRAM (`mkBRAMStore`). Users can supply their own store implementation as well.

The `mkRegStore` instantiates a single-element store. The module returns a `Store` interface and does not use a prefetch.

Module Name	BSV Module Declaration
Implementation of a single-element store	
`mkRegStore`	`module mkRegStore(Clock sClock, Clock dClock, Store#(UInt#(0),a,0) ifc) provisos(Bits#(a,a_sz) );`

The `mkRegVectorStore` module instantiates a vector-of-registers store. The module returns a `Store` interface and does not use a prefetch.

Implementation of a vector-of-registers store	
`mkRegVectorStore`	`module mkRegVectorStore(Clock sClock, Clock dClock, Store#(UInt#(w),a,0) ifc) provisos( Bits#(a,a_sz) );`

The `mkBRAMStore2W1R` module returns a `Store` interface and uses a prefetch. This model assumes the read clock is a 2x divided version of the write clock.

A BRAM-based store where the read clock is a 2x divided version of the write clock.	
`mkBRAMStore2W1R`	`module mkBRAMStore2W1R(Clock sClock, Reset sReset, Clock dClock, Reset dReset, Store#(i,a,1) ifc) provisos( Bits#(a,a_sz), Bits#(i,w), Eq#(i) );`

The `mkBRAMStore1W2R` module returns a `Store` interface and uses a prefetch. This model assumes the write clock is a 2x divided version of the read clock.

A BRAM-based store where the write clock is a 2x divided version of read clock.	
`mkBRAMStore1W2R`	`module mkBRAMStore1W2R(Clock sClock, Reset sReset, Clock dClock, Reset dReset, Store#(i,a,1) ifc) provisos( Bits#(a,a_sz), Bits#(i,w), Eq#(i) );`

The `mkAlignedFIFO` module makes a synchronizing FIFO for aligned clocks, based on the given backing store (determined by the type of store instantiated). The store is assumed to have 2^w slots addressed from 0 to $2^w - 1$. The store will be written in the source clock domain and read in the destination clock domain.

The `enq` and `deq` methods will only be callable when the `allow_enq` and `allow_deq` inputs are high. For a slow-to-fast crossing use:

```
allow_enq = constant True
allow_deq = pre-edge signal
```

For a fast-to-slow crossing, use:

```
allow_enq = pre-edge signal
allow_deq = constant True
```

The pre-edge signal is `True` when the slow clock will rise in the next clock cycle. The `ClockNextRdy` from the `ClockDividerIfc` (Section 3.9.3) can be used as the pre-edge signal.

These settings ensure that the outputs in the slow clock domain are stable for the entire cycle. Setting both inputs to constant True reduces the minimum latency through the FIFO, but allows outputs in the slow domain to transition mid-cycle. This is less safe and can interact badly with the `$displays` in a Verilog simulation.

It is not advisable to call both `dClear` and `sClear` simultaneously.

<table>
<tr><th colspan="2">Implementation of an aligned FIFO</th></tr>
<tr><td><code>mkAlignedFIFO</code></td><td><pre>(* no_default_clock, no_default_reset *)
module mkAlignedFIFO(Clock sClock
    , Reset sReset
    , Clock dClock
    , Reset dReset
    , Store#(i,a,n) store
    , Bool allow_enq
    , Bool allow_deq
    , AlignedFIFO#(a) ifc
)
   provisos(Bits#(a,sz_a), Bits#(i,w),
             Eq#(i), Arith#(i));</pre></td></tr>
</table>

3.2.7 Gearbox

Package

```
import Gearbox :: *
```

Description

This package defines FIFO-like converters that convert N-wide data to and from 1-wide data at N-times the frequency. These converters change the frequency and the data width, while the overall data rate stays the same. The data width on the fast side is always 1, while the data width on the slow side is N. The converters are intended to be used between clock domains with aligned edges for both types of clock domain crossings (fast to slow and slow to fast). For example:

300 MHz at 8-bits	converted to	100 MHz at 24-bits	(fast to slow)
100 MHz at 24-bits	converted to	300 MHz at 8-bits	(slow to fast)

In both of these examples, the data type `a` is `Bit#(8)` and N=3.

These modules are written in pure BSV using a style that utilzies only `mkNullCrossingReg` to cross registered values between clock domains. Restricting the form of clock crossings is important to ensure that the module preserves atomic semantics and also that it is compatible with both Verilog and Bluesim backends.

Interfaces and methods

The `Gearbox` interface provides the following methods: `enq`, `deq`, `first`, `notFull` and `notEmpty`.

Gearbox Interface		
Method Name	Type	Description
`enq`	Action	Adds an entry to the converter of type `Vector#(in, a)`, where `a` is the datatype. If the input is the fast domain then `in` = 1, if the input is the slow domain, `in` = N.
`deq`	Action	Removes the first entry from the converter.
`first`	`Vector#(out, a)`	Returns the first entry from the converter. If the output domain is the fast side, `out` = 1, if the output domain is the slow side, `out` = N.
`notFull`	Bool	Returns a True value if there is space to enqueue an entry into the FIFO.
`notEmpty`	Bool	Returns a True value if there is are elements in the FIFO and you can dequeue from the FIFO.

```
interface Gearbox#(numeric type in, numeric type out, type a);
   method    Action          enq(Vector#(in, a) din);
   method    Action          deq();
   method    Vector#(out, a) first();
   method    Bool            notFull();
   method    Bool            notEmpty();
endinterface
```

Modules

The package provides two modules: `mkNto1Gearbox` for slow to fast domain crossings, and `mk1toNGearbox` to for fast to slow domain crossings. These are intended for use between clock domains with aligned edges for both types of clock domain crossings.

Note: for both modules the resets in the source and destination domains (`sReset` and `dReset`) should be asserted together, otherwise only half the unit will be in reset.

With the `mkNto1Gearbox` module, 2xN elements of data storage are provided, grouped into 2 blocks of N elements each. Each block is writable in the source (slow) domain and readable in the destination (fast) domain.

`mkNto1Gearbox`	Moves data from a slow domain to a fast domain, changing the data width from a larger width to a smaller width. The data rate stays the same. The width of the output is 1, the width of the input is N.

	`module mkNto1Gearbox(Clock sClock, Reset sReset,`
	`                      Clock dClock, Reset dReset,`
	`                      Gearbox#(in, out, a) ifc)`
	`   provisos(Bits#(a, a_sz), Add#(out, 0, 1),`
	`            Add#(out, z, in) );`

With the `mk1toNGearbox` module, 2xN elements of data storage are provided, grouped into 2 blocks of N elements each. Each block is writable in the source (fast) domain and readable in the destination (slow) domain.

`mk1toNGearbox`	Moves data from a fast domain to a slow domain, changing the data width from a smaller width to a larger width. The data rate stays the same. The width of the input is 1, the width of the output is N.

	`module mk1toNGearbox(Clock sClock, Reset sReset,`
	`                      Clock dClock, Reset dReset,`
	`                      Gearbox#(in, out, a) ifc)`
	`   provisos(Bits#(a, a_sz), Add#(in, 0, 1),`
	`            Add#(in, z, out), Mul#(2, out, elements),`
	`            Add#(1, w, elements), Add#(out, x, elements) );`

3.2.8 MIMO

Package

```
import MIMO :: *
```

Description

This package defines a Multiple-In Multiple-Out (MIMO), an enhanced FIFO that allows the designer to specify the number of objects enqueued and dequeued in one cycle. There are different implementations of the MIMO available for synthesis: BRAM, Register, and Vector.

Types and type classes

The `LUInt` type is a `UInt` defined as the log of the `n`.

```
typedef UInt#(TLog#(TAdd#(n, 1)))     LUInt#(numeric type n);
```

The `MimoConfiguration` type defines whether the `MIMO` is guarded or unguarded, and whether it is based on a BRAM. There is an instance in the `DefaultValue` type class. The default `MIMO` is guarded and not based on a BRAM.

```
typedef struct {
   Bool         unguarded;
   Bool         bram_based;
} MIMOConfiguration deriving (Eq);
```

```
instance DefaultValue#(MIMOConfiguration);
   defaultValue = MIMOConfiguration {
      unguarded:  False,
      bram_based: False
      };
endinstance
```

Interfaces and methods

The `MIMO` interface is polymorphic and takes 4 parameters: `max_in`, `max_out`, `size`, and `t`.

MIMO Interace Parameters	
Name	Description
`max_in`	Maximum number of objects enqueued in one cycle. Must be numeric.
`max_out`	Maximum number of objects dequeued in one cycle. Must be numeric.
`size`	Total size of internal storage. Must be numeric.
`t`	Data type of the stored objects

The `MIMO` interface provides the following methods: `enq`, `first`, `deq`, `enqReady`, `enqReadyN`, `deqReady`, `deqReadyN`, `count`, and `clear`.

MIMO methods			
Method			Argument
Name	Type	Description	
`enq`	Action	adds an entry to the `MIMO`	`LUInt#(max_in) count` `Vector#(max_in, t) data`
`first`	`Vector#(max_out, t)`	Returns a Vector containing `max_out` items of type `t`.	
`deq`	Action	Removes the first `count` entries	`LUInt#(max_out) count`
`enqReady`	Bool	Returns a True value if there is space to enqueue an entry	
`enqReadyN`	Bool	Returns a True value if there is space to enqueue `count` entries	`LUInt#(max_in) count`
`deqReady`	Bool	Returns a True value if there is an element to dequeue	
`deqReadyN`	Bool	Returns a True value if there is are `count` elements to dequeue	`LUInt#(max_out) count`
`count`	`LUInt#(size)`	Returns the log of the number of elements in the `MIMO`	
`clear`	Action	Clears the `MIMO`	

```
interface MIMO#(numeric type max_in, numeric type max_out, numeric type size, type t);
   method    Action               enq(LUInt#(max_in) count, Vector#(max_in, t) data);
   method    Vector#(max_out, t)  first;
   method    Action               deq(LUInt#(max_out) count);
   method    Bool                 enqReady;
   method    Bool                 enqReadyN(LUInt#(max_in) count);
   method    Bool                 deqReady;
   method    Bool                 deqReadyN(LUInt#(max_out) count);
   method    LUInt#(size)         count;
   method    Action               clear;
endinterface
```

Modules

The package provides modules to synthesize different implementations of the MIMO: the basic MIMO (`mkMIMO`), BRAM-based (`mkMIMOBRAM`), register-based (`mkMIMOReg`), and a Vector of registers (`mkMIMOV`).

All implementations must meet the following provisos:

- The object must have bit representation
- The object must have at least 2 elements of storage.
- The maximum number of objects enqueued (`max_in`) must be less than or equal to the total bits of storage (`size`)
- The maximum number of objects dequeued (`max_out`) must be less than or equal to the total bits of storage (`size`)

`mkMIMO`	The basic implementation of MIMO. Object must be at least 1 bit in size.
	`module mkMIMO#(MIMOConfiguration cfg)(MIMO#(max_in, max_out, size, t));`

`mkMIMOBRAM`	Implementation of BRAM-based MIMO. Object must be at least 1 byte in size.
	`module mkMIMOBram#(MIMOConfiguration cfg)(MIMO#(max_in, max_out, size, t));`

`mkMIMOReg`	Implementation of register-based MIMO.
	`module mkMIMOReg#(MIMOConfiguration cfg)(MIMO#(max_in, max_out, size, t));`

`mkMIMOV`	Implementation of Vector-based MIMO. The ojbect must have a default value defined.
	`module mkMIMOV(MIMO#(max_in, max_out, size, t));`

3.3 Aggregation: Vectors

Package

```
import Vector :: * ;
```

Description

The `Vector` package defines an abstract data type which is a container of a specific length, holding elements of one type. Functions which create and operate on this type are also defined within this package. Because it is abstract, there are no constructors available for this type (like `Cons` and `Nil` for the `List` type).

```
typedef struct Vector#(type numeric vsize, type element_type);
```

Here, the type variable `element_type` represents the type of the contents of the elements while the numeric type variable `vsize` represents the length of the vector.

If the elements are in the `Bits` class, then the vector is as well. Thus a vector of these elements can be stored into Registers or FIFOs; for example a Register holding a vector of type `int`. Note that a vector can also store abstract types, such as a vector of `Rules` or a vector of `Reg` interfaces. These are useful during static elaboration although they have no hardware implementation.

Typeclasses

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

Bits A vector can be turned into bits if the individual elements can be turned into bits. When packed and unpacked, the zeroth element of the vector is stored in the least significant bits. The size of the resulting bits is given by $tsize = vsize * \texttt{SizeOf}\#(element\_type)$ which is specified in the provisos.

```
instance Bits #( Vector#(vsize, element_type), tsize)
   provisos (Bits#(element_type, sizea),
             Mul#(vsize, sizea, tsize));
```

Vectors are zero-indexed; the first element of a vector `v`, is `v[0]`. When vectors are packed, they are packed in order from the LSB to the MSB.

Example. Vector#(5, Bit#(7)) v1;

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

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

Example. A vector with a structure:

```
typedef struct { Bool a, UInt#(5) b} Newstruct deriving (Bits);
Vector#(3, NewStruct) v2;
```

The structure, `Newstruct` packs into 6 bits. Therefore `v2` will pack into an 18-bit vector. And its structure would look as follows:

	17	16 - 12	11	10 - 6	5	0	
MSB	v2[2].a	v2[2].b	v2[1].a	v2[1].b	v2[0].a	v2[0].b	LSB
	v2[2]		v2[1]		v2[0]		

Eq Vectors can be compared for equality if the elements can. That is, the operators `==` and `!=` are defined.

Bounded Vectors are bounded if the elements are.

FShow The `FShow` class provides the `fshow` function which can be applied to a `Vector` and returns an associated `Fmt` object showing:

```
<V elem1 elem2 ...>
```

where the `elemn` are the elements of the vector with `fshow` applied to each element value.

3.3.1 Creating and Generating Vectors

The following functions are used to create new vectors, with and without defined elements. There are no constructors available for this abstract type (and hence no pattern-matching is available for this type) but the following functions may be used to construct values of the `Vector` type.

`newVector`	Generate a vector with undefined elements, typically used when vectors are declared.
	`function Vector#(vsize, element_type) newVector();`

`genVector`	Generate a vector containing integers 0 through n-1, vector[0] will have value 0.
	`function Vector#(vsize, Integer) genVector();`

`replicate`	Generate a vector of elements by replicating the given argument (c).
	`function Vector#(vsize, element_type) replicate(element_type c);`

`genWith`	Generate a vector of elements by applying the given function to 0 through n-1. The argument to the function is another function which has one argument of type `Integer` and returns an `element_type`.
	`function Vector#(vsize, element_type) genWith(function element_type func(Integer x1));`

`cons`	Adds an element to a vector creating a vector one element larger. The new element will be at the 0th position. This function can lead to large compile times, so it can be an inefficient way to create and populate a vector. Instead, the designer should build a vector, then set each element to a value.
	`function Vector#(vsize1, element_type) cons (element_type elem, Vector#(vsize, element_type) vect) provisos (Add#(1, vsize, vsize1));`

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

`append`	Append two vectors containing elements of the same type, returning the combined vector. The resulting vector `result` will contain all the elements of `vecta` followed by all the elements of `vectb`. `result[0]` = `vecta[0]`, `result[vsize-1]` = `vectb[v1size-1]`.

	`function Vector#( vsize, element_type )`
	`      append( Vector#(v0size,element_type) vecta,`
	`              Vector#(v1size,element_type) vectb)`
	`  provisos (Add#(v0size, v1size, vsize)); //vsize = v0size + v1size`

`concat`	Append (*con*catenate) many vectors, that is a vector of vectors into one vector. `concat(xss)[0]`will be `xss[0][0]`, provided `m` and `n` are non-zero.

	`function Vector#(mvsize,element_type)`
	`      concat(Vector#(m,Vector#(n,element_type)) xss)`
	`  provisos (Mul#(m,n,mvsize));`

Examples - Creating and Generating Vectors

Create a new vector, `my_vector`, of 5 elements of datatytpe `Int#(32)`, with elements which are undefined.

```
Vector #(5, Int#(32)) my_vector;
```

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

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

Create a vector, my_vector, of five 1's.

```
Vector #(5,Int #(32)) my_vector = replicate (1);
// my_vector is a 5 element vector {1,1,1,1,1}
```

Create a vector, `my_vector`, by applying the given function `add2` to 0 through n-1.

```
function Integer add2 (Integer a);
    Integer c = a + 2;
return(c);
endfunction

Vector #(5,Integer) my_vector = genWith(add2);

// a is the index of the vector, 0 to n-1
// my_vector = {2,3,4,5,6,}
```

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

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

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

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

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

Vector#(3, Integer) my_vector1 = genWith(add2);
// my_vector1 = {2, 3, 4}

let my_vector2 = append(my_vector, my_vector1);
// my_vector2 = {0, 1, 2, 2, 3, 4}
```

3.3.2 Extracting Elements and Sub-Vectors

These functions are used to select elements or vectors from existing vectors, while retaining the input vector.

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

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

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

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

`head`	Extract the zeroth (head) element of a vector. The vector must have at least one element.

	`function element_type`
	`      head (Vector#(vsize, element_type) vect)`
	`  provisos(Add#(1,xxx,vxize)); // vsize >= 1`

`last`	Extract the highest (tail) element of a vector. The vector must have at least one element.

	`function element_type`
	`      last (Vector#(vsize, element_type) vect)`
	`  provisos(Add#(1,xxx,vxize)); // vsize >= 1`

`tail`	Remove the head element of a vector leaving its tail in a smaller vector.

	`function Vector#(vsize,element_type)`
	`      tail (Vector#(vsize1, element_type) xs)`
	`  provisos (Add#(1, vsize, vsize1));`

`init`	Remove the last element of a vector leaving its initial part in a smaller vector.

	`function Vector#(vsize,element_type)`
	`      init (Vector#(vsize1, element_type) xs)`
	`  provisos (Add#(1, vsize, vsize1));`

`take`	Take a number of elements from a vector starting from index 0. The number of elements to take is indicated by the type of the context where this is called, and is not specified as an argument to the function.
	`function Vector#(vxize2,element_type)`
	`      take (Vector#(vsize,element_type) vect)`
	`  provisos (Add#(vsize2,xxx,vsize)); // vsize2 <= vsize.`

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

`takeAt`	Take a number of elements starting at `startPos`. `startPos` must be a compile-time constant. If the `startPos` plus the output vector size extend beyond the end of the input vector, an error will be returned.
	`function Vector#(vsize2,element_type)`
	`       takeAt (Integer startPos, Vector#(vsize,element_type) vect)`
	`  provisos (Add#(vsize2,xxx,vsize)); // vsize2 <= vsize`

Examples - Extracting Elements and Sub-Vectors

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

```
// my_vector is a vector of elements {6,7,8,9,10,11}
// index = 3
// select or [ ] will generate a MUX

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

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

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

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

Extract the zeroth element of the vector `my_vector`.

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

newvalue = head(my_vector);
// newvalue = 6
```

Extract the last element of the vector `my_vector`.

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

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

Create a vector, `my_vector2`, of size 4 by removing the head (zeroth) element of the vector `my_vector1`.

```
// my_vector1 is a vector with 5 elements {0,1,2,3,4}

Vector #(4, Int#(32)) my_vector2 = tail (my_vector1);
// my_vector2 is a vector of 4 elements {1,2,3,4}
```

Create a vector, `my_vector2`, of size 4 by removing the tail (last) element of the vector `my_vector1`.

```
// my_vector1 is a vector with 5 elements {0,1,2,3,4}

Vector #(4, Int#(32)) my_vector2 = init (my_vector1);
// my_vector2 is a vector of 4 elements {0,1,2,3}
```

Create a 2 element vector, `my_vector2`, by taking the first two elements of the vector `my_vector1`.

```
// my_vector1 is vector with 5 elements {0,1,2,3,4}

Vector #(2, Int#(4)) my_vector2 = take (my_vector1);
// my_vector2 is a 2 element vector {0,1}
```

Create a 3 element vector, `my_vector2`, by taking the last 3 elements of vector, `my_vector1`. using `takeTail`

```
// my_vector1 is Vector with 5 elements {0,1,2,3,4}

Vector #(3,Int #(4)) my_vector2 = takeTail (my_vector1);
// my_vector2 is a 3 element vector {2,3,4}
```

Create a 3 element vector, `my_vector2`, by taking the 1st - 3rd elements of vector, `my_vector1`. using `takeAt`

```
// my_vector1 is Vector with 5 elements {0,1,2,3,4}

Vector #(3,Int #(4)) my_vector2 = takeAt (1, my_vector1);
// my_vector2 is a 3 element vector {1,2,3}
```

3.3.3 Vector to Vector Functions

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

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

`rotateBy`	Shift each element `n` places. The last `n` elements are moved to the beginning, the element at index `0` moves to index `n`, index `1` to index `n+1`, etc.	`function Vector#(vsize, element_type)`
		`      rotateBy (Vector#(vsize,element_type) vect, UInt#(log(v)) n)`
		`  provisos (Log#(vsize, logv);`

`shiftInAt0`	Shift a new element into the vector at index 0, bumping the index of all other element up by one. The highest element is dropped.	`function Vector#(vsize,element_type)`
		`      shiftInAt0 (Vector#(vsize,element_type) vect,`
		`                  element_type newElement);`

`shiftInAtN`	Shift a new element into the vector at index n, bumping the index of all other elements down by one. The 0th element is dropped.	`function Vector#(vsize,element_type)`
		`      shiftInAtN (Vector#(vsize,element_type) vect,`
		`                  element_type newElement);`

`shiftOutFrom0`	Shifts out `amount` number of elements from the vector starting at index 0, bumping the index of all remaining elements down by `amount`. The shifted elements are replaced with the value `default`. This function is similar to a `>>` bit operation. `amt_type` must be of an acceptable index type (`Integer`, `Bit#(n)`, `Int#(n)` or `UInt#(n)`).
	`function Vector#(vsize,element_type) shiftOutFrom0 (element_type default, Vector#(vsize,element_type) vect, amt_type amount);`

`shiftOutFromN`	Shifts out `amount` number of elements from the vector starting at index `vsize-1` bumping the index of remaining elements up by `amount`. The shifted elements are replaced with the value `default`. This function is similar to a `<<` bit operation. `amt_type` must be of an acceptable index type (`Integer`, `Bit#(n)`, `Int#(n)` or `UInt#(n)`).
	`function Vector#(vsize,element_type) shiftOutFromN (element_type default, Vector#(vsize,element_type) vect, amt_type amount);`

`reverse`	Reverse element order
	`function Vector#(vsize,element_type) reverse(Vector#(vsize,element_type) vect);`

`transpose`	Matrix transposition of a vector of vectors.
	`function Vector#(m,Vector#(n,element_type)) transpose ( Vector#(n,Vector#(m,element_type)) matrix );`

`transposeLN`	Matrix transposition of a vector of Lists.
	`function Vector#(vsize, List#(element_type)) transposeLN( List#(Vector#(vsize, element_type)) lvs );`

Examples - Vector to Vector Functions

Create a vector by moving the last element to the first, then shifting each element to the right.

```
// my_vector1 is a vector of elements with values {1,2,3,4,5}

my_vector2 = rotateR (my_vector1);
// my_vector2 is a vector of elements with values {5,1,2,3,4}
```

Create a vector which is the input vector rotated by 2 places.

```
// my_vector1 is a vector of elements {1,2,3,4,5}

my_vector2 = rotateBy {my_vector1, 2};
// my_vector2 = {4,5,1,2,3}
```

Create a vector which shifts out 3 elements starting from 0, replacing them with the value F

```
// my_vector1 is a vector of elements {5,4,3,2,1,0}

my_vector2 = shiftOutFrom0 (F, my_vector1, 3);
// my_vector2 is a vector of elements {F,F,F,5,4,3}
```

Create a vector which shifts out 3 elements starting from n-1, replacing them with the value F

```
// my_vector1 is a vector of elements {5,4,3,2,1,0}

my_vector2 = shiftOutFromN (F, my_vector1, 3);
// my_vector2 is a vector of elements {2,1,0,F,F,F}
```

Create a vector which is the reverse of the input vector.

```
// my_vector1 is a vector of elements {1,2,3,4,5}

my_vector2 = reverse (my_vector1);
// my_vector2 is a vector of elements {5,4,3,2,1}
```

Use transpose to create a new vector.

```
// my_vector1 is a Vector#(3, Vector#(5, Int#(8)))
// the result, my_vector2, is a Vector #(5,Vector#(3,Int #(8)))

// my_vector1 has the values:
// {{0,1,2,3,4},{5,6,7,8,9},{10,11,12,13,14}}

my_vector2 = transpose(my_vector1);
// my_vector2 has the values:
// {{0,5,10},{1,6,11},{2,7,12},{3,8,13},{4,9,14}}
```

3.3.4 Tests on Vectors

The following functions are used to test vectors. The first set of functions are Boolean functions, i.e. they return `True` or `False` values.

<table>
<tr><td rowspan="2"><code>elem</code></td><td>Check if a value is an element of a vector.</td></tr>
<tr><td><pre>function Bool elem (element_type x,
                    Vector#(vsize,element_type) vect)
  provisos (Eq#(element_type));</pre></td></tr>
</table>

<table>
<tr><td rowspan="2"><code>any</code></td><td>Test if a predicate holds for any element of a vector.</td></tr>
<tr><td><pre>function Bool any(function Bool pred(element_type x1),
                  Vector#(vsize,element_type) vect);</pre></td></tr>
</table>

`all`	Test if a predicate holds for all elements of a vector.	
	`function Bool all(function Bool pred(element_type x1),` `                  Vector#(vsize,element_type) vect );`	

`or`	Combine all elements in a vector of Booleans with a logical or. Returns True if any elements in the Vector are True.
	`function Bool or (Vector#(vsize, Bool) vect );`

`and`	Combine all elements in a vector of Booleans with a logical and. Returns True if all elements in the Vector are True.
	`function Bool and (Vector#(vsize, Bool) vect );`

The following two functions return the number of elements in the vector which match a condition.

`countElem`	Returns the number of elements in the vector which are equal to a given value. The return value is in the range of 0 to vsize.

	`function UInt#(logv1) countElem (element_type x,`
	`                                 Vector#(vsize, element_type) vect)`
	`  provisos (Eq#(element_type), Add#(vsize, 1, vsize1),`
	`            Log#(vsize1, logv1));`

`countIf`	Returns the number of elements in the vector which satisfy a given predicate function. The return value is in the range of 0 to vsize.

	`function UInt#(logv1) countIf (function Bool pred(element_type x1)`
	`                               Vector#(vsize, element_type) vect)`
	`  provisos (Add#(vsize, 1, vsize1), Log#(vsize1, logv1));`

`find`	Returns the first element that satisfies the predicate or `Nothing` if there is none.

	`function Maybe#(element_type)`
	`      find (function Bool pred(element_type),`
	`            Vector#(vsize, element_type) vect);`

The following two functions return the index of an element.

<table>
<tr><td rowspan="2"><code>findElem</code></td><td>Returns the index of the first element in the vector which equals a given value. Returns an <code>Invalid</code> if not found or <code>Valid</code> with a value of 0 to vsize-1 if found.</td></tr>
<tr><td><pre>function Maybe#(UInt#(logv)) findElem (element_type x,
                                Vector#(vsize, element_type) vect)
  provisos (Eq#(element_type), Add#(xx1, 1, vsize),
            Log#(vsize, logv));</pre></td></tr>
</table>

<table>
<tr><td rowspan="2"><code>findIndex</code></td><td>Returns the index of the first element in the vector which satisfies a given predicate. Returns an <code>Invalid</code> if not found or <code>Valid</code> with a value of 0 to vsize-1 if found.</td></tr>
<tr><td><pre>function Maybe#(UInt#(logv)) findIndex
                          (function Bool pred(element_type x1)
                           Vector#(vsize, element_type) vect)
  provisos (Add#(xx1,1,vsize), Log#(vsize, logv));</pre></td></tr>
</table>

Examples -Tests on Vectors

Test that all elements of the vector `my_vector1` are positive integers.

```
function Bool isPositive (Int #(32) a);
     return (a > 0)
endfunction

// function isPositive checks that "a" is a  positive integer
// if my_vector1 has n elements, n instances of the predicate
// function isPositive will be generated.

if (all(isPositive, my_vector1))
   $display ("Vector contains all negative values");
```

Test if any elements in the vector are positive integers.

```
// function isPositive checks that "a" is a  positive integer
// if my_vector1 has n elements, n instances of the predicate
// function isPositive will be generated.

if (any(isPositive, my_vector1))
   $display ("Vector contains some negative values");
```

Check if the integer 5 is in `my_vector`.

```
// if my_vector contains n elements, elem will generate n copies
// of the eq test
if (elem(5,my_vector))
   $display ("Vector contains the integer 5");
```

Count the number of elements which match the integer provided.

```
// my_vector1 is a vector of {1,2,1,4,3}
x = countElem ( 1, my_vector1);
// x = 2
y = countElem (4, my_vector1);
// y = 1
```

Find the index of an element which equals a predicate.

```
let f = findIndex ( beIsGreaterThan( 3 ) , my_vector );
 if ( f matches tagged Valid .indx )
    begin
       printBE ( my_vector[indx] ) ;
       $display ("Found data > 3 at index %d ", indx ) ;
 else
    begin
       $display ( "Did not find data > 3" ) ;
    end
```

3.3.5 Bit-Vector Functions

The following functions operate on bit-vectors.

`rotateBitsBy`	Shift each bit to a higher index by `n` places. The last `n` bits are moved to the begininng and the bit at index `(0)` moves to index `(n)`.
	`function Bit#(n) rotateBitsBy (Bit#(n) bvect, UInt#(logn) n)` `  provisos (Log#(n,logn), Add#(1,xxx,n));`

`countOnesAlt`	Returns the number of elements equal to 1 in a bit-vector. (This function differs slightly from the Prelude version of countOnes and has fewer provisos.)
	`function UInt#(logn1) countOnesAlt (Bit#(n) bvect)` `  provisos (Add#(1,n,n1), Log#(n1,logn1));`

3.3.6 Functions on Vectors of Registers

`readVReg`	Returns the values from reading a vector of registers (interfaces).
	`function Vector#(n,a) readVReg ( Vector#(n, Reg#(a)) vrin) ;`

`writeVReg`	Returns an Action which is the write of all registers in `vr` with the data from `vdin`.
	`function Action writeVReg ( Vector#(n, Reg#(a)) vr,` `                            Vector#(n,a) vdin) ;`

3.3.7 Combining Vectors with Zip

The family of `zip` functions takes two or more vectors and combines them into one vector of `Tuples`. Several variations are provided for different resulting `Tuples`, as well as support for mis-matched vector sizes.

zip	Combine two vectors into a vector of Tuples.	`function Vector#(vsize,Tuple2 #(a_type, b_type))`
		`      zip( Vector#(vsize, a_type) vecta,`
		`           Vector#(vsize, b_type) vectb);`

zip3	Combine three vectors into a vector of Tuple3.	`function Vector#(vsize,Tuple3 #(a_type, b_type, c_type))`
		`      zip3( Vector#(vsize, a_type) vecta,`
		`            Vector#(vsize, b_type) vectb,`
		`            Vector#(vsize, c_type) vectc);`

zip4	Combine four vectors into a vector of Tuple4.	`function Vector#(vsize,Tuple4 #(a_type, b_type, c_type, d_type))`
		`      zip4( Vector#(vsize, a_type) vecta,`
		`            Vector#(vsize, b_type) vectb,`
		`            Vector#(vsize, c_type) vectc,`
		`            Vector#(vsize, d_type) vectd);`

zipAny	Combine two vectors into one vector of pairs (2-tuples); result is as long as the smaller vector.	`function Vector#(vsize,Tuple2 #(a_type, b_type))`
		`      zipAny(Vector#(m,a_type) vect1,`
		`             Vector#(n,b_type) vect2);`
		`  provisos (Max#(m,vsize,m), Max#(n, vsize, n));`

unzip	Separate a vector of pairs (i.e. a `Tuple2#(a,b))` into a pair of two vectors.	`function Tuple2#(Vector#(vsize,a_type), Vector#(vsize, b_type))` `      unzip(Vector#(vsize,Tuple2 #(a_type, b_type)) vectab);`

Examples - Combining Vectors with Zip

Combine two vectors into a vector of Tuples.

```
// my_vector1 is a vector of elements {0,1,2,3,4}
// my_vector2 is a vector of elements {5,6,7,8,9}

my_vector3 = zip(my_vector1, my_vector2);
// my_vector3 is a vector of Tuples {(0,5),(1,6),(2,7),(3,8),(4,9)}
```

Separate a vector of pairs into a Tuple of two vectors.

```
// my_vector3 is a vector of pairs {(0,5),(1,6),(2,7),(3,8),(4,9)}

Tuple2#(Vector #(5,Int #(5)),Vector #(5,Int #(5))) my_vector4 =
                                                    unzip(my_vector3);
// my_vector4 is ({0,1,2,3,4},{5,6,7,8,9})
```

3.3.8 Mapping Functions over Vectors

A function can be applied to all elements of a vector, using high-order functions such as `map`. These functions take as an argument a function, which is applied to the elements of the vector.

<table>
<tr><td rowspan="2"><code>map</code></td><td>Map a function over a vector, returning a new vector of results.</td></tr>
<tr><td><pre>function Vector#(vsize,b_type)
      map (function b_type func(a_type x),
           Vector#(vsize, a_type) vect);</pre></td></tr>
</table>

Example - Mapping Functions over Vectors

Consider the following code example which applies the `extend` function to each element of `avector` into a new vector, `resultvector`.

```
Vector#(13,Bit#(5))   avector;
Vector#(13,Bit#(10))  resultvector;
...
resultvector = map( extend, avector ) ;
```

This is equivalent to saying:

```
for (Integer i=0; i<13; i=i+1)
    resultvector[i] = extend(avector[i]);
```

Map a negate function over a Vector

```
// my_vector1 is a vector of 5 elements {0,1,2,3,4}
// negate is a function which makes each element negative

Vector #(5,Int #(32)) my_vector2 = map (negate, my_vector1);

// my_vector2 is a vector of 5 elements {0,-1,-2,-3,-4}
```

3.3.9 ZipWith Functions

The zipWith functions combine two or more vectors with a function and generate a new vector. These functions combine features of map and zip functions.

<table>
<tr><td rowspan="2"><code>zipWith</code></td><td>Combine two vectors with a function.</td></tr>
<tr><td><pre>function Vector#(vsize,c_type)
         zipWith (function c_type func(a_type x, b_type y),
                  Vector#(vsize,a_type) vecta,
                  Vector#(vsize,b_type) vectb);</pre></td></tr>
</table>

`zipWithAny`	Combine two vectors with a function; result is as long as the smaller vector.
	`function Vector#(vsize,c_type) zipWithAny (function c_type func(a_type x, b_type y), Vector#(m,a_type) vecta, Vector#(n,b_type) vectb ) provisos (Max#(n, vsize, n), Max#(m, vsize, m));`

```
function Vector#(vsize,c_type)
         zipWithAny (function c_type func(a_type x, b_type y),
                     Vector#(m,a_type) vecta,
                     Vector#(n,b_type) vectb )
  provisos (Max#(n, vsize, n), Max#(m, vsize, m));
```

`zipWith3`	Combine three vectors with a function.

```
function Vector#(vsize,d_type)
        zipWith3(function d_type func(a_type x, b_type y, c_type z),
                 Vector#(vsize,a_type) vecta,
                 Vector#(vsize,b_type) vectb,
                 Vector#(vsize,c_type) vectc );
```

`zipWithAny3`	Combine three vectors with a function; result is as long as the smallest vector.

```
function Vector#(vsize,c_type)
   zipWithAny3(function d_type func(a_type x, b_type y, c_type z),
               Vector#(m,a_type) vecta,
               Vector#(n,b_type) vectb,
               Vector#(o,c_type) vectc )
provisos (Max#(n, vsize, n), Max#(m, vsize, m), Max#(o, vsize, o));
```

Examples - ZipWith

Create a vector by applying a function over the elements of 3 vectors.

```
// the function add3 adds 3 values
function Int#(n) add3 (Int #(n) a,Int #(n) b,Int #(n) c);
    Int#(n) d = a + b +c ;
    return d;
endfunction

// Create the vector my_vector4 by adding the ith element of each of
// 3 vectors (my_vector1, my_vector2, my_vector3) to generate the ith
// element of my_vector4.

// my_vector1 = {0,1,2,3,4}
// my_vector2 = {5,6,7,8,9}
// my_vector3 = {10,11,12,13,14}

Vector #(5,Int #(8)) my_vector4 = zipWith3(add3, my_vector1, my_vector2, my_vector3);
// creates 5 instances of the add3 function in hardware.
// my_vector4 = {15,18,21,24,27}

// This is equivalent to saying:
   for (Integer i=0; i<5; i=i+1)
      my_vector4[i] = my_vector1[i] + my_vector2[i] + my_vector3[i];
```

3.3.10 Fold Functions

The `fold` family of functions reduces a vector to a single result by applying a function over all its elements. That is, given a vector of `element_type`, $V_0, V_1, V_2, ..., V_{n-1}$, a seed of type `b_type`, and a function `func`, the reduction for `foldr` is given by

$$func(V_0, func(V_1, ..., func(V_{n-2}, func(V_{n-1}, seed))));$$

Note that `foldr` start processing from the highest index position to the lowest, while `foldl` starts from the lowest index (zero), i.e. `foldl` is:

$$func(...(func(func(seed, V_0), V_1), ...)V_{n-1})$$

`foldr`	Reduce a vector by applying a function over all its elements. Start processing from the highest index to the lowest.
	`function b_type foldr(function b_type func(a_type x, b_type y),` `                        b_type seed, Vector#(vsize,a_type) vect);`

`foldl`	Reduce a vector by applying a function over all its elements. Start processing from the lowest index (zero).
	`function b_type foldl (function b_type func(b_type y, a_type x),` `                        b_type seed, Vector#(vsize,a_type) vect);`

The functions `foldr1` and `foldl1` use the first element as the seed. This means they only work on vectors of at least one element. Since the result type will be the same as the element type, there is no `b_type` as there is in the `foldr` and `foldl` functions.

`foldr1`	`foldr` function for a non-zero sized vector, using element V_{n-1} as a seed. Vector must have at least 1 element. If there is only one element, it is returned.

	`function element_type foldr1(`
	`         function element_type func(element_type x, element_type y),`
	`         Vector#(vsize,element_type) vect)`
	`  provisos (Add#(1, xxx, vsize));`

`foldl1`	`foldl` function for a non-zero sized vector, using element V_0as a seed. Vector must have at least 1 element. If there is only one element, it is returned.

	`function element_type foldl1 (`
	`         function element_type func(element_type y, element_type x),`
	`         Vector#(vsize,element_type) vect)`
	`  provisos (Add#(1, xxx, vsize));`

The `fold` function also operates over a non-empty vector, but processing is accomplished in a binary tree-like structure. Hence the depth or delay through the resulting function will be $O(log_2(vsize)$ rather than $O(vsize)$.

`fold`	Reduce a vector by applying a function over all its elements, using a binary tree-like structure. The function returns the same type as the arguments.	`function element_type fold ( function element_type func(element_type y, element_type x), Vector#(vsize,element_type) vect ) provisos (Add#(1, xxx, vsize));`

```
function element_type fold (
        function element_type func(element_type y, element_type x),
        Vector#(vsize,element_type) vect )
  provisos (Add#(1, xxx, vsize));
```

`mapPairs`	Map a function over a vector consuming two elements at a time. Any straggling element is processed by the second function.	see below

```
function Vector#(vsize2,b_type)
         mapPairs (
            function b_type func1(a_type x, a_type y),
            function b_type func2(a_type x),
            Vector#(vsize,a_type) vect )
  provisos (Div#(vsize, 2, vsize2));
```

`joinActions`	Join a number of actions together. `joinActions` is used for static elaboration only, no hardware is generated.	`function Action joinActions (Vector#(vsize,Action) vactions);`

`joinRules`	Join a number of rules together.`joinRules` is used for static elaboration only, no hardware is generated.	`function Rules joinRules (Vector#(vsize,Rules) vrules);`

Example - Folds

Use fold to find the sum of the elements in a vector.

```
// my_vector1 is a vector of five integers {1,2,3,4,5}
// \+ is a function which returns the sum of the elements
// make sure you leave a space after the \+ and before the ,

// This will build an adder tree, instantiating 4 adders, with a maximum
// depth or delay of 3.  If foldr1 or foldl1 were used, it would
// still instantiate 4 adders, but the delay would be 4.

my_sum =  fold (\+ , my_vector1));
// my_sum = 15
```

Use fold to find the element with the maximum value.

```
// my_vector1 is a vector of five integers  {2,45,5,8,32}

my_max = fold (max, my_vector1);
// my_max = 45
```

Create a new vector using mapPairs. The function sum is applied to each pair of elements (the first and second, the third and fourth, etc.). If there is an uneven number of elements, the function pass is applied to the remaining element.

```
// sum is defined as c = a+b
function Int#(4) sum (Int #(4) a,Int #(4) b);
     Int#(4) c = a + b;
     return(c);
endfunction

// pass is defined as a
function Int#(4) pass (Int #(4) a);
     return(a);
endfunction

// my_vector1 has the elements {0,1,2,3,4}

my_vector2 = mapPairs(sum,pass,my_vector1);
// my_vector2 has the elements {1,5,4}
// my_vector2[0] = 0 + 1
// my_vector2[1] = 2 + 3
// my_vector2[2] = 4
```

3.3.11 Scan Functions

The scan family of functions applies a function over a vector, creating a new vector result. The scan function is similar to fold, but the intermediate results are saved and returned in a vector, instead of returning just the last result. The result of a scan function is a vector. That is, given a vector of element_type, $V_0, V_1, ..., V_{n-1}$, an initial value initb of type b_type, and a function func, application of the scanr functions creates a new vector W, where

$$
\begin{aligned}
W_n &= init; \\
W_{n-1} &= func(V_{n-1}, W_n); \\
W_{n-2} &= func(V_{n-2}, W_{n-1}); \\
&\;... \\
W_1 &= func(V_1, W_2); \\
W_0 &= func(V_0, W_1);
\end{aligned}
$$

scanr	Apply a function over a vector, creating a new vector result. Processes elements from the highest index position to the lowest, and fill the resulting vector in the same way. The result vector is 1 element longer than the input vector.
	`function Vector#(vsize1,b_type) scanr(function b_type func(a_type x1, b_type x2), b_type initb, Vector#(vsize,a_type) vect) provisos (Add#(1, vsize, vsize1));`

`sscanr`	Apply a function over a vector, creating a new vector result. The elements are processed from the highest index position to the lowest. The W_n element is dropped from the result. Input and output vectors are the same size.
	
	`function Vector#(vsize,b_type)`
	`        sscanr(function b_type func(a_type x1, b_type x2),`
	`               b_type initb,`
	`               Vector#(vsize,a_type) vect );`

The `scanl` function creates the resulting vector in a similar way as `scanr` except that the processing happens from the zeroth element up to the n^{th} element.

$$\begin{aligned} W_0 &= init; \\ W_1 &= func(W_0, V_0); \\ W_2 &= func(W_1, V_1); \\ &\ldots \\ W_{n-1} &= func(W_{n-2}, V_{n-2}); \\ W_n &= func(W_{n-1}, V_{n-1}); \end{aligned}$$

The `sscanl` function drops the first result, *init*, shifting the result index by one.

`scanl`	Apply a function over a vector, creating a new vector result. Processes elements from the zeroth element up to the n^{th} element. The result vector is 1 element longer than the input vector.
	
	`function Vector#(vsize1,a_type)`
	`        scanl(function a_type func(a_type x1, b_type x2),`
	`              a_type q,`
	`              Vector#(vsize, b_type) vect)`
	`  provisos (Add#(1, vsize, vsize1));`

`sscanl`	Apply a function over a vector, creating a new vector result. Processes elements from the zeroth element up to the n^{th} element. The first result, *init*, is dropped, shifting the result index up by one. Input and output vectors are the same size.
	
	`function Vector#(vsize,a_type)`
	`        sscanl(function a_type func(a_type x1, b_type x2),`
	`               a_type q,`
	`               Vector#(vsize, b_type) vect );`

mapAccumL	Map a function, but pass an accumulator from head to tail.
	`function Tuple2 #(a_type, Vector#(vsize,c_type)) mapAccumL (function Tuple2 #(a_type, c_type) func(a_type x, b_type y), a_type x0, Vector#(vsize,b_type) vect );`

mapAccumR	Map a function, but pass an accumulator from tail to head.
	`function Tuple2 #(a_type, Vector#(vsize,c_type)) mapAccumR(function Tuple2 #(a_type, c_type) func(a_type x, b_type y), a_type x0, Vector#(vsize,b_type) vect );`

Examples - Scan

Create a vector of factorials.

```
// \* is a function which returns the result of a multiplied by  b
function Bit #(16) \* (Bit #(16) b, Bit #(8) a);
   return (extend (a) * b);
endfunction

// Create a vector of factorials by multiplying each input list element
// by the previous product (the output list element), to generate
// the next product.  The seed is a Bit#(16) with a value of 1.
// The elements are processed from the zeroth element up to the $n^{th}$ element.

// my_vector1 = {1,2,3,4,5,6,7}
Vector#(8,Bit #(16)) my_vector2 = scanl (\*, 16'd1, my_vector1);
// 7 multipliers are generated

// my_vector2 = {1,1,2,6,24,120,720,5040}
// foldr with the same arguments would return just 5040.
```

3.3.12 Monadic Operations

Within Bluespec, there are some functions which can only be invoked in certain contexts. Two common examples are: `ActionValue`, and module instantiation. ActionValues can only be invoked within an `Action` context, such as a rule block or an Action method, and can be considered as two parts - the action and the value. Module instantiation can similarly be considered, modules can only be instantiated in the module context, while the two parts are the module instantiation (the action performed) and the interface (the result returned). These situations are considered *monadic*.

When a monadic function is to be applied over a vector using map-like functions such as `map`, `zipWith`, or `replicate`, the monadic versions of these functions must be used. Moreover, the context requirements of the applied function must hold. The common application for these functions is in the generation (or instantiation) of vectors of hardware components.

`mapM`	Takes a monadic function and a vector, and applies the function to all vector elements returning the vector of corresponding results.	

	`function m#(Vector#(vsize, b_type))`
	`        mapM ( function m#(b_type) func(a_type x),`
	`               Vector#(vsize, a_type) vecta )`
	`   provisos (Monad#(m));`	

`mapM_`	Takes a monadic function and a vector, applies the function to all vector elements, and throws away the resulting vector leaving the action in its context.	

	`function m#(void) mapM_(function m#(b_type) func(a_type x),`
	`                        Vector#(vsize, a_type) vect)`
	`  provisos (Monad#(m));`	

`zipWithM`	Take a monadic function (which takes two arguments) and two vectors; the function applied to the corresponding element from each vector would return an action and result. Perform all those actions and return the vector of corresponding results.	

	`function m#(Vector#(vsize, c_type))`
	`         zipWithM( function m#(c_type) func(a_type x, b_type y),`
	`                   Vector#(vsize, a_type) vecta,`
	`                   Vector#(vsize, b_type) vectb )`
	`  provisos (Monad#(m));`	

`zipWithM_`	Take a monadic function (which takes two arguments) and two vectors; the function is applied to the corresponding element from each vector. This is the same as `zipWithM` but the resulting vector is thrown away leaving the action in its context.	

	`function m#(void)`
	`         zipWithM_(function m#(c_type) func(a_type x, b_type y),`
	`                   Vector#(vsize, a_type) vecta,`
	`                   Vector#(vsize, b_type) vectb )`
	`  provisos (Monad#(m));`	

<table>
<tr><td rowspan="2"><code>zipWith3M</code></td><td>Same as <code>zipWithM</code> but combines three vectors with a function. The function is applied to the corresponding element from each vector and returns an action and the vector of corresponding results.</td></tr>
<tr><td><pre>function m#(Vector#(vsize, c_type))
     zipWith3M(function m#(d_type)
                func(a_type x, b_type y, c_type z),
                Vector#(vsize, a_type) vecta,
                Vector#(vsize, b_type) vectb,
                Vector#(vsize, c_type) vectc)
  provisos (Monad#(m));</pre></td></tr>
</table>

<table>
<tr><td rowspan="2"><code>genWithM</code></td><td>Generate a vector of elements by applying the given monadic function to 0 through n-1.</td></tr>
<tr><td><pre>function m#(Vector#(vsize, element_type))
         genWithM(function m#(element_type) func(Integer x))
  provisos (Monad#(m));</pre></td></tr>
</table>

<table>
<tr><td rowspan="2"><code>replicateM</code></td><td>Generate a vector of elements by using the given monadic value repeatedly.</td></tr>
<tr><td><pre>function m#(Vector#(vsize, element_type))
         replicateM(m#(element_type) c)
  provisos (Monad#(m));</pre></td></tr>
</table>

Examples - Creating a Vector of Registers

The following example shows some common uses of the `Vector` type. We first create a vector of registers, and show how to populate this vector. We then continue with some examples of accessing and updating the registers within the vector, as well as alternate ways to do the same.

```
// First define a variable to hold the register interfaces.
// Notice the variable is really a vector of Interfaces of type Reg,
// not a vector of modules.
Vector#(10,Reg#(DataT))   vectRegs ;

// Now we want to populate the vector, by filling it with Reg type
// interfaces, via the mkReg module.
// Notice that the replicateM function is used instead of the
// replicate function since mkReg function is creating a module.
vectRegs <- replicateM( mkReg( 0 ) ) ;

// ...

// A rule showing a read and write of one register within the
// vector.
// The readReg function is required since the selection of an
// element from vectRegs returns a Reg#(DType) interface, not the
// value of the register.  The readReg functions converts from a
```

```
// Reg#(DataT) type to a DataT type.
rule zerothElement ( readReg( vectRegs[0] ) > 20 ) ;
   // set 0 element to 0
   // The parentheses are required in this context to give
   // precedence to the selection over the write operation.
   (vectRegs[0]) <= 0 ;

   // Set the 1st element to 5
   // An alternate syntax
   vectRegs[1]._write( 5 ) ;
endrule

rule lastElement ( readReg( vectRegs[9] ) > 200 ) ;
   // Set the 9th element to -10000
   (vectRegs[9]) <= -10000 ;
endrule

// These rules defined above can execute simultaneously, since
// they touch independent registers

// Here is an example of dynamic selection,  first we define a
// register to be used as the selector.
Reg#(UInt#(4))  selector <- mkReg(0) ;

// Now define another Reg variable which is selected from the
// vectReg variable.  Note that no register is created here, just
// an alias is defined.
Reg#(DataT)  thisReg = select(vectRegs, selector ) ;

//The above statement is equivalent to:
//Reg#(DataT) thisReg = vectRegs[selector] ;


// If the selected register is greater than 20'h7_0000, then its
// value is reset to zero.  Note that the vector update function is
// not required since we are changing the contents of a register
// not the vector vectReg.
rule reduceReg( thisReg > 20'h7_0000 ) ;
   thisReg <= 0 ;
   selector <= ( selector < 9 ) ? selector + 1 : 0 ;
endrule

// As an alternative, we can define N rules which each check the
// value of one register and update accordingly.  This is done by
// generating each rule inside an elaboration-time for-loop.

Integer i;  // a compile time variable
for ( i = 0 ; i < 10 ; i = i + 1 ) begin
   rule checkValue( readReg( vectRegs[i] ) > 20'h7_0000 ) ;
      (vectRegs[i]) <= 0 ;
   endrule
end
```

3.3.13 Converting to and from Vectors

There are functions which convert between Vectors and other types.

`toList`	Convert a Vector to a List.
	`function List#(element_type) toList (Vector#(vsize, element_type) vect);`

`toVector`	Convert a List to a Vector.
	`function Vector#(vsize, element_type) toVector ( List#(element_type) lst);`

`arrayToVector`	Convert an array to a Vector.
	`function Vector#(vsize, element_type) arrayToVector ( element_type[ ] arr);`

`vectorToArray`	Convert a Vector to an array.
	`function element_type[ ] vectorToArray (Vector#(vsize, element_type) vect);`

`toChunks`	Convert a value to a Vector of chunks, possibly padding the final chunk. The input type and size as well as the chunk type and size are determined from their types.
	`function Vector#(n_chunk, chunk_type) toChunks(type_x x) provisos( Bits#(chunk_type, chunk_sz), Bits#(type_x, x_sz) , Div#(x_sz, chunk_sz, n_chunk) );`

Example - Converting to and from Vectors

Convert the vector `my_vector` to a list named `my_list`.

```
Vector#(5,Int#(13)) my_vector;
List#(Int#(13)) my_list = toList(my_vector);
```

3.3.14 ListN

Package name

```
import ListN :: * ;
```

Description

ListN is an alternative implementation of Vector which is preferred for sequential list processing functions, such as head, tail, map, fold, etc. All Vector functions are available, by substituting ListN for Vector. See the Vector documentation (3.3) for details. If the implementation requires random access to items in the list, the Vector construct is recommended. Using ListN where Vectors is recommended (and visa-versa) can lead to very long static elaboration times.

The `ListN` package defines an abstract data type which is a ListN of a specific length. Functions which create and operate on this type are also defined within this package. Because it is abstract, there are no constructors available for this type (like `Cons` and `Nil` for the `List` type).

```
struct ListN#(vsize,a_type)
```
· · · *abstract* · · ·

Here, the type variable "`a_type`" represents the type of the contents of the listN while type variable "`vsize`" represents the length of the ListN.

3.4 Aggregation: Lists

Package

```
import List :: * ;
```

Description

The `List` package defines a data type and functions which create and operate on this data type. Lists are similar to Vectors, but are used when the number of items on the list may vary at compile-time or need not be strictly enforced by the type system. All elements of a list must be of the same type. The list type is defined as a tagged union as follows.

```
typedef union tagged {
    void Nil;
    struct {
        a         head;
        List #(a) tail;
    } Cons;
} List #(type a);
```

A list is tagged `Nil` if it has no elements, otherwise it is tagged `Cons`. `Cons` is a structure of a single element and the rest of the list.

Lists are most often used during static elaboration (compile-time) to manipulate collections of objects. Since `List#(element_type)` is not in the `Bits` typeclass, lists cannot be stored in registers or other dynamic elements. However, one can have a list of registers or variables corresponding to hardware functions.

Data classes

FShow The `FShow` class provides the function `fshow` which can be applied to a `List` and returns an associated `Fmt` object showing:

```
<List elem1 elem2 ...>
```

where the `elemn` are the elements of the list with `fshow` applied to each element value.

3.4.1 Creating and Generating Lists

`cons`	Adds an element to a list. The new element will be at the 0th position.
	`function List#(element_type) cons (element_type x, List#(element_type) xs);`

`upto`	Create a list of Integers counting up over a range of numbers, from m to n. If m > n, an empty list (`Nil`) will be returned.
	`List#(Integer) upto(Integer m, Integer n);`

`replicate`	Generate a list of n elements by replicating the given argument, `elem`.
	`function List#(element_type) replicate(Integer n, element_type elem);`

`append`	Append two lists, returning the combined list. The elements of both lists must be the same datatype, `element_type`. The combined list will contain all the elements of `xs` followed in order by all the elements of `ys`.
	`function List#(element_type) append(List#(element_type) xs, List#(element_type) ys);`

`concat`	Append (*con*catenate) many lists, that is a list of lists, into one list.
	`function List# (element_type) concat (List#(List#(element_type)) xss);`

Examples - Creating and Generating Lists

Create a new list, `my_list`, of elements of datatytpe `Int#(32)` which are undefined

```
List #(Int#(32)) my_list;
```

Create a list, `my_list`, of five 1's

```
List #(Int #(32)) my_list = replicate (5,32'd1);

//my_list = {1,1,1,1,1}
```

Create a new list using the `upto` function

```
List #(Integer) my_list2 = upto (1, 5);

//my_list2 = {1,2,3,4,5}
```

3.4.2 Extracting Elements and Sub-Lists

`[i]`	The square-bracket notation is available to extract an element from a list or update an element within it. Extracts or updates the ith element, where the first element is [0]. Index i must be of an acceptable index type (e.g. `Integer`, `Bit#(n)`, `Int#(n)` or `UInt#(n)`). The square-bracket notation for lists can also be used with register writes.
	`anyList[i];` `anyList[i] = newValue;`

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

```
function element_type
      select(List#(element_type) alist, idx_type index);
```

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

```
function List#(element_type)
      update(List#(element_type) alist,
                 idx_type index,
                 element_type newElem);
```

`oneHotSelect`	Select a list element with a Boolean list. The Boolean list should have exactly one element that is `True`, otherwise the result is undefined. The returned element is the one in the corresponding position to the `True` element in the Boolean list.

```
function element_type
      oneHotSelect (List#(Bool) bool_list,
                    List#(element_type) alist);
```

`head`	Extract the first element of a list. The input list must have at least 1 element, or an error will be returned.	`function element_type head (List#(element_type) listIn);`
`last`	Extract the last element of a list. The input list must have at least 1 element, or an error will be returned.	`function element_type last (List#(element_type) alist);`
`tail`	Remove the head element of a list leaving the remaining elements in a smaller list. The input list must have at least 1 element, or an error will be returned.	`function List#(element_type) tail (List#(element_type) alist);`
`init`	Remove the last element of a list the remaining elements in a smaller list. The input list must have at least one element, or an error will be returned.	`function List#(element_type) init (List#(element_type) alist);`
`take`	Take a number of elements from a list starting from index 0. The number to take is specified by the argument `n`. If the argument is greater than the number of elements on the list, the function stops taking at the end of the list and returns the entire input list.	`function List#(element_type)` `      take (Integer n, List#(element_type) alist);`
`drop`	Drop a number of elements from a list starting from index 0. The number to drop is specified by the argument `n`. If the argument is greater than the number of elements on the list, the entire input list is dropped, returning an empty list.	`function List#(element_type)` `      drop (Integer n, List#(element_type) alist);`

`filter`	Create a new list from a given list where the new list has only the elements which satisfy the predicate function.

	`function List#(element_type)`
	`      filter (function Bool pred(element_type),`
	`              List#(element_type) alist);`

`find`	Return the first element that satisfies the predicate or `Nothing` if there is none.

	`function Maybe#(element_type)`
	`      find (function Bool pred(element_type),`
	`            List#(element_type) alist);`

`lookup`	Returns the value in an association list or `Nothing` if there is no matching value.

	`function Maybe#(b_type)`
	`      lookup (a_type key, List#(Tuple2#(a_type, b_type)) alist)`
	`   provisos(Eq#(a_type));`

`takeWhile`	Returns the first set of elements of a list which satisfy the predicate function.

	`function List#(element_type)`
	`      takeWhile (function Bool pred(element_type x),`
	`                 List#(element_type) alist);`

`takeWhileRev`	Returns the last set of elements on a list which satisfy the predicate function.

	`function List#(element_type)`
	`      takeWhileRev (function Bool pred(element_type x),`
	`                    List#(element_type) alist);`

`dropWhile`	Removes the first set of elements on a list which satisfy the predicate function, returning a list with the remaining elements.

	`function List#(element_type)`
	`      dropWhile (function Bool pred(element_type x),`
	`                 List#(element_type) alist);`

`dropWhileRev`	Removes the last set of elements on a list which satisfy the predicate function, returning a list with the remaining elements.
	`function List#(element_type) dropWhileRev (function Bool pred(element_type x), List#(element_type) alist);`

Examples - Extracting Elements and Sub-Lists

Extract the element from a list, `my_list`, at the position of `index`.

```
//my_list = {1,2,3,4,5}, index = 3

newvalue = select (my_list, index);

//newvalue = 4
```

Extract the zeroth element of the list `my_list`.

```
//my_list = {1,2,3,4,5}

newvalue = head(my_list);

//newvalue = 1
```

Create a list, `my_list2`, of size 4 by removing the head (zeroth) element of the list `my_list1`.

```
//my_list1 is a list with 5 elements, {0,1,2,3,4}

List #(Int #(32)) my_list2 = tail (my_list1);
List #(Int #(32)) my_list3 = tail(tail(tail(tail(tail(my_list1);

//my_list2 = {1,2,3,4}
//my_list3 = Nil
```

Create a 2 element list, `my_list2`, by taking the first two elements of the list `my_list1`.

```
//my_list1 is list with 5 elements, {0,1,2,3,4}
List #(Int #(4)) my_list2 = take (2,my_list1);

//my_list2 = {0,1}
```

The number of elements specified to take in `take` can be greater than the number of elements on the list, in which case the entire input list will be returned.

```
//my_list1 is list with 5 elements, {0,1,2,3,4}
List #(Int #(4)) my_list2 = take (7,my_list1);

//my_list2 = {0,1,2,3,4}
```

Select an element based on a boolean list.

```
//my_list1 is a list of unsigned integers, {1,2,3,4,5}
//my_list2 is a list of Booleans, only one value in my_list2 can be True.
//my_list2 = {False, False, True, False,False, False, False}.

result = oneHotSelect (my_list2, my_list1));

//result = 3
```

Create a list by removing the initial segment of a list that meets a predicate.

```
//the predicate function is a < 2

function Bool lessthan2 (Int #(4) a);
    return (a < 2);
endfunction

//my_list1 = {0,1,2,0,1,7,8}

 List #(Int #(4)) my_result = (dropWhile(lessthan2, my_list1));

//my_result = {2,0,1,7,8}
```

3.4.3 List to List Functions

`rotate`	Move the first element to the last and shift each element to the next higher index.
	`function List#(element_type) rotate (List#(element_type) alist);`

`rotateR`	Move last element to the beginning and shift each element to the next lower index.
	`function List#(element_type) rotateR (List#(element_type) alist);`

`reverse`	Reverse element order
	`function List#(element_type) reverse(List#(element_type) alist);`

`transpose`	Matrix transposition of a list of lists.
	`function List#(List#(element_type)) transpose ( List#(List#(element_type)) matrix );`

`sort`	Uses the ordering defined for the `element_type` data type to return a list in ascending order. The type `element_type` must be in the `Ord` type class.
	`function List#(element_type) sort(List#(element_type) alist) provisos(Ord#(element_type)) ;`

`sortBy`	Generalizes the `sort` function to use an arbitrary ordering function defined by the comparison function `comparef` in place of the `Ord` instance for `element_type`.

	`function List#(element_type)`
	`   sortBy(function Ordering comparef(element_type x, element_type y),`
	`          List#(element_type) alist);`

`group`	Returns a list of the contiguous subsequences of equal elements (according to the `Eq` instance for `element_type`) found in its input list. Every element in the input list will appear in exactly one sublist of the result. Every sublist will be a non-empty list of equal elements. For any list, `x`, `concat(group(x)) == x`.
	`function List#(List#(element_type)) group (List#(element_type) alist)` `   provisos(Eq#(element_type)) ;`

`groupBy`	Generalizes the `group` function to use an arbitrary equivalence relation defined by the comparison function `eqf` in place of the Eq instance for `element_type`.

	`function List#(List#(element_type))`
	`   groupBy(function Bool eqf(element_type x, element_type y),`
	`           List#(element_type) alist);`

Examples - List to List Functions

Create a list by moving the last element to the first, then shifting each element to the right.

```
//my_list1 is a List of elements with values {1,2,3,4,5}

my_list2 = rotateR (my_list1);

//my_list2 is a List of elements with values {5,1,2,3,4}
```

Create a list which is the reverse of the input List

```
//my_list1 is a List of elements {1,2,3,4,5}

my_list2 = reverse (my_list1);

//my_list2 is a List of elements {5,4,3,2,1}
```

Use transpose to create a new list

```
//my_list1 has the values:
//{{0,1,2,3,4},{5,6,7,8,9},{10,11,12,13,14}}

my_list2 = transpose(my_list1);

//my_list2 has the values:
//{{0,5,10},{1,6,11},{2,7,12},{3,8,13},{4,9,14}}
```

Use sort to create a new list

```
//my_list1 has the values: {3,2,5,4,1}

my_list2 = sort(my_list1);

//my_list2 has the values: {1,2,3,4,5}
```

Use group to create a list of lists

```
//my_list1 is a list of elements {Mississippi}

my_list2 = group(my_list1);

//my_list2 is a list of lists:
{{M},{i},{ss},{i},{ss},{i},{pp},{i}}
```

3.4.4 Tests on Lists

`==` `!=`	Lists can be compared for equality if the elements in the list can be compared.
	`instance Eq #( List#(element_type) )` `   provisos( Eq#( element_type ) ) ;`

`elem`	Check if a value is an element in a list.
	`function Bool elem (element_type x, List#(element_type) alist )` `  proviso (Eq#(element_type));`

`length`	Determine the length of a list. Can be done at elaboration time only.
	`function Integer length (List#(element_type) alist );`

`any`	Test if a predicate holds for any element of a list.
	`function Bool any(function Bool pred(element_type x1),` `                  List#(element_type) alist );`

`all`	Test if a predicate holds for all elements of a list.
	`function Bool all(function Bool pred(element_type x1),` `                  List#(element_type) alist );`

`or`	Combine all elements in a Boolean list with a logical or. Returns True if any elements in the list are True.	
	`function Bool or (List# (Bool) bool_list);`	

`and`	Combine all elements in a Boolean list with a logical and. Returns True if all elements in the list are true.	
	`function Bool and (List# (Bool) bool_list);`	

Examples - Tests on Lists

Test that all elements of the list `my_list1` are positive integers

```
function Bool isPositive (Int #(32) a);
     return (a > 0)
endfunction

// function isPositive checks that "a" is a  positive integer
// if my_list1 has n elements, n instances of the predicate
// function isPositive will be generated.

if (all(isPositive, my_list1))
   $display ("List contains all negative values");
```

Test if any elements in the list are positive integers.

```
// function isPositive checks that "a" is a  positive integer
// if my_list1 has n elements, n instances of the predicate
// function isPositive will be generated.

if (any(pos, my_list1))
   $display ("List contains some negative values");
```

Check if the integer 5 is in `my_list`

```
// if my_list contains n elements, elem will generate n copies
// of the eqt Test
if (elem(5,my_list))
   $display ("List contains the integer 5");
```

3.4.5 Combining Lists with Zip Functions

The family of `zip` functions takes two or more lists and combines them into one list of `Tuples`. Several variations are provided for different resulting `Tuples`. All variants can handle input lists of different sizes. The resulting lists will be the size of the smallest list.

`zip`	Combine two lists into a list of Tuples.

	`function List#(Tuple2 #(a_type, b_type))`
	`        zip( List#(a_type) lista,`
	`             List#(b_type) listb);`

`zip3`	Combine 3 lists into a list of Tuple3.

	`function List#(Tuple3 #(a_type, b_type, c_type))`
	`        zip3( List#(a_type) lista,`
	`              List#(b_type) listb,`
	`              List#(c_type) listc);`

`zip4`	Combine 4 lists into a list of Tuple4.

	`function List#(Tuple4 #(a_type, b_type, c_type, d_type))`
	`        zip4( List#(a_type) lista,`
	`              List#(b_type) listb,`
	`              List#(c_type) listc,`
	`              List#(d_type) listd);`

`unzip`	Separate a list of pairs (i.e. a `Tuple2#(a,b)`) into a pair of two lists.
	`function Tuple2#(List#(a_type), List#(b_type))` `        unzip(List#(Tuple2 #(a_type, b_type)) listab);`

Examples - Combining Lists with Zip

Combine two lists into a list of Tuples

```
//my_list1 is a list of elements {0,1,2,3,4,5,6,7}
//my_list2 is a list of elements {True,False,True,True,False}

my_list3 = zip(my_list1, my_list2);

//my_list3 is a list of Tuples {(0,True),(1,False),(2,True),(3,True),(4,False)}
```

Separate a list of pairs into a Tuple of two lists

```
//my_list is a list of pairs {(0,5),(1,6),(2,7),(3,8),(4,9)}

Tuple2#(List#(Int#(5)),List#(Int#(5))) my_list2 = unzip(my_list);

//my_list2 is ({0,1,2,3,4},{5,6,7,8,9})
```

3.4.6 Mapping Functions over Lists

A function can be applied to all elements of a list, using high-order functions such as `map`. These functions take as an argument a function, which is applied to the elements of the list.

`map`	Map a function over a list, returning a new list of results.
	`function List#(b_type) map (function b_type func(a_type),` `                            List#(a_type) alist);`

Example - Mapping Functions over Lists

Consider the following code example which applies the `extend` function to each element of `alist` creating a new list, `resultlist`.

```
List#(Bit#(5))   alist;
List#(Bit#(10))  resultlist;
...
resultlist = map( extend, alist ) ;
```

This is equivalent to saying:

```
for (Integer i=0; i<13; i=i+1)
    resultlist[i] = extend(alist[i]);
```

Map a negate function over a list

```
//my_list1 is a list of 5 elements {0,1,2,3,4}
//negate is a function which makes each element negative

List #(Int #(32)) my_list2 = map (negate, my_list1);

//my_list2 is a list of 5 elements {0,-1,-2,-3,-4}
```

3.4.7 ZipWith Functions

The zipWith functions combine two or more lists with a function and generate a new list. These functions combine features of map and zip functions.

`zipWith`	Combine two lists with a function. The lists do not have to have the same number of elements.

	`function List#(c_type)`
	`      zipWith (function c_type func(a_type x, b_type y),`
	`               List#(a_type) listx,`
	`               List#(b_type) listy );`

`zipWith3`	Combine three lists with a function. The lists do not have to have the same number of elements.

	`function List#(d_type)`
	`      zipWith3(function d_type func(a_type x, b_type y, c_type z),`
	`               List#(a_type) listx,`
	`               List#(b_type) listy,`
	`               List#(c_type) listz );`

`zipWith4`	Combine four lists with a function. The lists do not have to have the same number of elements.	`function List#(e_type) zipWith4 (function e_type func(a_type x, b_type y, c_type z, d_type w), List#(a_type) listx, List#(b_type) listy, List#(c_type) listz List#(d_type) listw );`

Examples - ZipWith

Create a list by applying a function over the elements of 3 lists.

```
//the function add3 adds 3 values
function Int#(8) add3 (Int #(8) a,Int #(8) b,Int #(8) c);
    Int#(8) d = a + b +c ;
    return(d);
endfunction

//Create the list my_list4 by adding the ith element of each of
//3 lists (my_list1, my_list2, my_list3) to generate the ith
//element of my_list4.

//my_list1 = {0,1,2,3,4}
//my_list2 = {5,6,7,8,9}
//my_list3 = {10,11,12,13,14}

List #(Int #(8)) my_list4 = zipWith3(add3, my_list1, my_list2, my_list3);

//my_list4 = {15,18,21,24,27}

// This is equivalent to saying:
   for (Integer i=0; i<5; i=i+1)
      my_list4[i] = my_list1[i] + my_list2[i] + my_list3[i];
```

3.4.8 Fold Functions

The `fold` family of functions reduces a list to a single result by applying a function over all its elements. That is, given a list of `element_type`, $L_0, L_1, L_2, ..., L_{n-1}$, a seed of type `b_type`, and a function `func`, the reduction for `foldr` is given by

$$func(L_0, func(L_1, ..., func(L_{n-2}, func(L_{n-1}, seed))));$$

Note that `foldr` start processing from the highest index position to the lowest, while `foldl` starts from the lowest index (zero), i.e.,

$$func(...(func(func(seed, L_0), L_1), ...)L_{n-1})$$

foldr	Reduce a list by applying a function over all its elements. Start processing from the highest index to the lowest.	

	`function b_type foldr(b_type function func(a_type x, b_type y),`
	`                      b_type seed,`
	`                      List#(a_type) alist);`	

foldl	Reduce a list by applying a function over all its elements. Start processing from the lowest index (zero).	

	`function b_type foldl (b_type function func(b_type y, a_type x),`
	`                       b_type seed,`
	`                       List#(a_type) alist);`	

The functions `foldr1` and `foldl1` use the first element as the seed. This means they only work on lists of at least one element. Since the result type will be the same as the element type, there is no `b_type` as there is in the `foldr` and `foldl` functions.

foldr1	`foldr` function for a non-zero sized list. Uses element L_{n-1} as the seed. List must have at least 1 element.	

	`function element_type foldr1`
	`      (element_type function func(element_type x, element_type y),`
	`       List#(element_type) alist);`	

foldl1	`foldl` function for a non-zero sized list. Uses element L_0 as the seed. List must have at least 1 element.	

	`function element_type foldl1`
	`      (element_type function func(element_type y, element_type x),`
	`       List#(element_type) alist);`	

The `fold` function also operates over a non-empty list, but processing is accomplished in a binary tree-like structure. Hence the depth or delay through the resulting function will be $O(log_2(lsize)$ rather than $O(lsize)$.

fold	Reduce a list by applying a function over all its elements, using a binary tree-like structure. The function returns the same type as the arguments.	

	`function element_type fold`
	`      (element_type function func(element_type y, element_type x),`
	`       List#(element_type) alist );`	

joinActions	Join a number of actions together.	
	`function Action joinActions (List#(Action) list_actions);`	

joinRules	Join a number of rules together.
	`function Rules joinRules (List#(Rules) list_rules);`

mapPairs	Map a function over a list consuming two elements at a time. Any straggling element is processed by the second function.

	`function List#(b_type)`
	`         mapPairs (`
	`            function b_type func1(a_type x, a_type y),`
	`            function b_type func2(a_type x),`
	`            List#(a_type) alist );`

Example - Folds

```
// my_list1 is a list of five integers {1,2,3,4,5}
// \+ is a function which returns the sum of the elements

my_sum =  foldr (\+ , 0, my_list1));

// my_sum = 15
```

Use fold to find the element with the maximum value

```
// my_list1 is a list of five integers  {2,45,5,8,32}

my_max = fold (max, my_list1);

// my_max = 45
```

Create a new list using `mapPairs`. The function `sum` is applied to each pair of elements (the first and second, the third and fourth, etc.). If there is an uneven number of elements, the function `pass` is applied to the remaining element.

```
//sum is defined as c = a+b
function Int#(4) sum (Int #(4) a,Int #(4) b);
     Int#(4) c = a + b;
       return(c);
endfunction

//pass is defined as a
function Int#(4) pass (Int #(4) a);
       return(a);
endfunction

//my_list1 has the elements {0,1,2,3,4}
```

```
my_list2 = mapPairs(sum,pass,my_list1);

//my_list2 has the elements {1,5,4}
//my_list2[0] = 0 + 1
//my_list2[1] = 2 + 3
//my_list2[3] = 4
```

3.4.9 Scan Functions

The scan family of functions applies a function over a list, creating a new List result. The scan function is similar to fold, but the intermediate results are saved and returned in a list, instead of returning just the last result. The result of a scan function is a list. That is, given a list of element_type, $L_0, L_1, ..., L_{n-1}$, an initial value initb of type b_type, and a function func, application of the scanr functions creates a new list W, where

$$\begin{aligned} W_n &= init; \\ W_{n-1} &= func(L_{n-1}, W_n); \\ W_{n-2} &= func(L_{n-2}, W_{n-1}); \\ ... & \\ W_1 &= func(L_1, W_2); \\ W_0 &= func(L_0, W_1); \end{aligned}$$

scanr	Apply a function over a list, creating a new list result. Processes elements from the highest index position to the lowest, and fills the resulting list in the same way. The result list is one element longer than the input list.

```
function List#(b_type)
         scanr(function b_type func(a_type x1, b_type x2),
               b_type initb,
               List#(a_type) alist);
```

sscanr	Apply a function over a list, creating a new list result. The elements are processed from the highest index position to the lowest. Drops the W_n element from the result. Input and output lists are the same size.

```
function List#(b_type)
         sscanr(function b_type func(a_type x1, b_type x2),
                b_type initb,
                List#(a_type) alist );
```

The scanl function creates the resulting list in a similar way as scanr except that the processing happens from the zeroth element up to the nth element.

$$\begin{aligned} W_0 &= init; \\ W_1 &= func(W_0, L_0); \end{aligned}$$

$$
\begin{aligned}
W_2 &= func(W_1, L_1);\\
... & \\
W_{n-1} &= func(W_{n-2}, L_{n-2});\\
W_n &= func(W_{n-1}, L_{n-1});
\end{aligned}
$$

The `sscanl` function drops the first result, *init*, shifting the result index by one.

scanl

Apply a function over a list, creating a new list result. Processes elements from the zeroth element up to the nth element. The result list is 1 element longer than the input list.

```
function List#(a_type)
         scanl(function a_type func(a_type x1, b_type x2),
               a_type inita,
               List#(b_type) alist);
```

sscanl

Apply a function over a list, creating a new list result. Processes elements from the zeroth element up to the nth element. Drop the first result, *init*, shifting the result index by one. The length of the input and output lists are the same.

```
function List#(a_type)
         sscanl(function a_type func(a_type x1, b_type x2),
                a_type inita,
                List#(b) alist );
```

mapAccumL

Map a function, but pass an accumulator from head to tail.

```
function Tuple2 #(a_type, List#(c_type))
         mapAccumL (function Tuple2 #(a_type, c_type)
                    func(a_type x, b_type y),a_type x0,
                    List#(b_type) alist );
```

mapAccumR

Map a function, but pass an accumulator from tail to head.

```
function Tuple2 #(a_type, List#(c_type))
         mapAccumR(function Tuple2 #(a_type, c_type)
                   func(a_type x, b_type y),a_type x0,
                   List#(b_type) alist );
```

Examples - Scan

Create a list of factorials

```
//the function my_mult multiplies element a by element b
function Bit #(16) my_mult (Bit #(16) b, Bit #(8) a);
   return (extend (a) * b);
```

```
endfunction

// Create a list of factorials by multiplying each input list element
// by the previous product (the output list element), to generate
// the next product.  The seed is a Bit#(16) with a value of 1.
// The elements are processed from the zeroth element up to the nth element.
//my_list1 = {1,2,3,4,5,6,7}

List #(Bit #(16)) my_list2 = scanl (my_mult, 16'd1, my_list1);

//my_list2 = {1,1,2,6,24,120,720,5040}
```

3.4.10 Monadic Operations

Within Bluespec, there are some functions which can only be invoked in certain contexts. Two common examples are: `ActionValue`, and module instantiation. ActionValues can only be invoked within an `Action` context, such as a rule block or an Action method, and can be considered as two parts - the action and the value. Module instantiation can similarly be considered, modules can only be instantiated in the module context, while the two parts are the module instantiation (the action performed) and the interface (the result returned). These situations are considered *monadic*.

When a monadic function is to be applied over a list using map-like functions such as `map, zipWith`, or `replicate`, the monadic versions of these functions must be used. Moreover, the context requirements of the applied function must hold.

<table>
<tr><td rowspan="2"><code>mapM</code></td><td>Takes a monadic function and a list, and applies the function to all list elements returning the list of corresponding results.</td></tr>
<tr><td><pre>function m#(List#(b_type))
         mapM (function m#(b_type) func(a_type x),
                List#(a_type) alist)
   provisos (Monad#(m));</pre></td></tr>
</table>

<table>
<tr><td rowspan="2"><code>mapM_</code></td><td>Takes a monadic function and a list, applies the function to all list elements, and throws away the resulting list leaving the action in its context.</td></tr>
<tr><td><pre>function m#(void)
         mapM_(function m#(b_type) func(a_type x),
               List#(a_type) alist)
  provisos (Monad#(m));</pre></td></tr>
</table>

<table>
<tr><td rowspan="2"><code>zipWithM</code></td><td>Take a monadic function (which takes two arguments) and two lists; the function applied to the corresponding element from each list would return an action and result. Perform all those actions and return the list of corresponding results.</td></tr>
<tr><td><pre>function m#(List#(c_type))
         zipWithM(function m#(c_type) func(a_type x, b_type y),
                   List#(a_type) alist,
                   List#(b_type) blist)
  provisos (Monad#(m));</pre></td></tr>
</table>

<table>
<tr><td rowspan="2"><code>zipWith3M</code></td><td>Same as <code>zipWithM</code> but combines three lists with a function. The function is applied to the corresponding element from each list and returns an action and the list of corresponding results.</td></tr>
<tr><td><pre>function m#(List#(d_type))
     zipWith3M(function m#(d_type)
                func(a_type x, b_type y, c_type z),
                List#(a_type) alist ,
                List#(b_type) blist,
                List#(c_type) clist)
  provisos (Monad#(m));</pre></td></tr>
</table>

<table>
<tr><td rowspan="2"><code>replicateM</code></td><td>Generate a list of elements by using the given monadic value repeatedly.</td></tr>
<tr><td><pre>function m#(List#(element_type))
         replicateM(Integer n, m#(element_type) c)
  provisos (Monad#(m));</pre></td></tr>
</table>

3.5 Math

3.5.1 Real

Package

```
import Real :: * ;
```

Description

The `Real` library package defines functions to operate on and manipulate real numbers. Real numbers are numbers with a fractional component. They are also of limited precision. The `Real` data type is described in section 2.2.6.

Constants

The constant `pi` (π) is defined.

pi	The value of the constant pi (π).	
	`Real pi;`	

Trigonometric Functions

The following trigonometric functions are provided: `sin`, `cos`, `tan`, `sinh`, `cosh`, `tanh`, `asin`, `acos`, `atan`, `asinh`, `acosh`, `atanh`, and `atan2`.

sin	Returns the sine of `x`.
	`function Real sin (Real x);`

cos	Returns the cosine of `x`.
	`function Real cos (Real x);`

tan	Returns the tangent of `x`.
	`function Real tan (Real x);`

sinh	Returns the hyperbolic sine of `x`.
	`function Real sinh (Real x);`

cosh	Returns the hyperbolic cosine of `x`.
	`function Real cosh (Real x);`

tanh	Returns the hyperbolic tangent of `x`.
	`function Real tanh (Real x);`

`asinh`	Returns the inverse hyperbolic sine of `x`.	
	`function Real asinh (Real x);`	

`acosh`	Returns the inverse hyperbolic cosine of `x`.
	`function Real acosh (Real x);`

`atanh`	Returns the inverse hyperbolic tangent of `x`.
	`function Real atanh (Real x);`

`atan2`	Returns `atan(`x/y`)`. `atan2(1,x)` is equivalent to `atan(x)`, but provides more precision when required by the division of `x/y`.
	`function Real atan2 (Real y, Real x);`

Arithmetic Functions

`pow`	The element x is raised to the y power. An alias for `**`. `pow(x,y)` = `x**y` = x^y.
	`function Real pow (Real x, Real y);`

`sqrt`	Returns the square root of `x`. Returns an error if `x` is negative.
	`function Real sqrt (Real x);`

Conversion Functions

The following four functions are used to convert a `Real` to an `Integer`.

`trunc`	Converts a `Real` to an `Integer` by removing the fractional part of `x`, which can be positive or negative. `trunc(1.1) = 1`, `trunc(-1.1)= -1`.	
	`function Integer trunc (Real x);`	

`round`	Converts a `Real` to an `Integer` by rounding to the nearest whole number. `.5` rounds up in magnitude. `round(1.5) = 2`, `round(-1.5)= -2`.
	`function Integer round (Real x);`

`ceil`	Converts a `Real` to an `Integer` by rounding to the higher number, regardless of sign. `ceil(1.1) = 2`, `ceil(-1.1) = -1`.
	`function Integer ceil (Real x);`

`floor`	Converts a `Real` to an `Integer` by rounding to the lower number, regardless of sign. `floor(1.1) = 1`, `floor(-1.1) = -2`.
	`function Integer floor (Real x);`

There are also two system functions `$realtobits` and `$bitstoreal`, defined in the Prelude (section 2.2.6) which provide conversion to and from IEEE 64-bit vectors (`Bit#(64)`).

Introspection Functions

`isInfinite`	Returns `True` if the value of `x` is infinite, `False` if `x` is finite.
	`function Bool isInfinite (Real x);`

`isNegativeZero`	Returns `True` if the value of `x` is negative zero.
	`function Bool isNegativeZero (Real x);`

`splitReal`	Returns a Tuple containing the whole (n) and fractional (f) parts of `x` such that $n + f = x$. Both values have the same sign as `x`. The absolute value of the fractional part is guaranteed to be in the range [0,1).
	`function Tuple2#(Integer, Real) splitReal (Real x);`

`decodeReal`	Returns a Tuple3 containing the sign, the fraction, and the exponent of a real number. The second part (the first Integer) represents the fractional part as a signed Integer value. This can be converted to an `Int#(54)` (52 bits, plus hidden bit, plus the sign bit). The last value is a signed Integer representing the exponent, which can be be converted to an `Int#(11)` . The real number is represented exactly as ($fractional \times 2^{exp}$). The `Bool` represents the sign and is `True` for positive and positive zero, `False` for negative and negative zero. Since the second value is a signed value, the `Bool` is redundant except for zero values.
	`function Tuple3#(Bool, Integer, Integer) decodeReal (Real x);`

`realToDigits`	Deconstructs a real number into its digits. The function takes a base and a real number and returns a list of digits and an exponent (ignoring the sign). In particular, if $x \geq 0$, and `realToDigits(base,x)` returned a list of digits `d1, d2, ..., dn` and an exponent `e`, then: • `n` $\geq$ `1` • `abs(x) = 0.d1d2...dn * (base`^{e}`)` • `0` $\leq$ `di` $\leq$ `base-1`
	`function Tuple2#(List#(Integer), Integer)` `        realToDigits (Integer base, Real r);`

3.5.2 OInt

Package

```
import OInt :: * ;
```

Description

The `OInt#(n)` type is an abstract type that can store a number in the range "`0..n-1`". The representation of a `OInt#(n)` takes up n bits, where exactly one bit is a set to one, and the others are zero, i.e., it is a *one-hot* decoded version of the number. The reason to use a `OInt` number is that the `select` operation is more efficient than for a binary-encoded number; the code generated for select takes advantage of the fact that only one of the bits may be set at a time.

Types and type classes

Definition of `OInt`

```
typedef ...  OInt #(numeric type n) ... ;
```

Type Classes used by OInt									
	Bits	Eq	Literal	Arith	Ord	Bounded	Bit wise	Bit Reduction	Bit Extend
OInt	√	√	√			√			

Functions

A binary-encoded number can be converted to an OInt.

toOInt	Converts from a bit-vector in unsigned binary format to an OInt. An out-of-range number gives an unspecified result.
	`function OInt#(n) toOInt(Bit#(k) k)` `  provisos( Log#(n,k)) ;`

An OInt can be converted to a binary-encoded number.

fromOInt	Converts an OInt to a bit-vector in unsigned binary format.
	`function Bit#(k) fromOInt(OInt#(n) o)` `  provisos( Log#(n,k)) ;`

An OInt can be used to select an element from a Vector in an efficient way.

select	The Vector select function, where the type of the index is an OInt.

	`function a_type select(Vector#(vsize, a_type) vecta,`
	`                       OInt#(vsize) index)`
	`  provisos (Bits#(a_type, sizea));`

3.5.3 Complex

Package

```
import Complex :: * ;
```

Description

The Complex package provides a representation for complex numbers plus functions to operate on variables of this type. The basic representation is the Complex structure, which is polymorphic on the type of data it holds. For example, one can have complex numbers of type Int or of type FixedPoint. A Complex number is represented in two part, the real part (rel) and the imaginary part (img). These fields are accessible though standard structure addressing, i.e., foo.rel and foo.img where foo is of type Complex.

```
typedef struct {
        any_t  rel ;
        any_t  img ;
        } Complex#(type any_t)
deriving ( Bits, Eq ) ;
```

Types and type classes

The `Complex` type belongs to the `Arith`, `Literal`, `SaturatingArith`, and `FShow` type classes. Each type class definition includes functions which are then also defined for the data type. The Prelude library definitions (Section 2) describes which functions are defined for each type class.

Type Classes used by `Complex`										
	`Bits`	`Eq`	`Literal`	`Arith`	`Ord`	`Bounded`	`Bit` wise	`Bit` Reduction	`Bit` Extend	`FShow`
`Complex`	√	√	√	√						√

Arith The type `Complex` belongs to the `Arith` type class, hence the common infix operators (+, -, *, and /) are defined and can be used to manipulate variables of type `Complex`. The remaining arithmetic operators are not defined for the `Complex` type. Note however, that some functions generate more hardware than may be expected. The complex multiplication (*) produces four multipliers in a combinational function; some other modules could accomplish the same function with less hardware but with greater latency. The complex division operator (/) produces 6 multipliers, and a divider and may not always be synthesizable with downstream tools.

```
instance Arith#( Complex#(any_type) )
      provisos( Arith#(any_type) ) ;
```

Literal The `Complex` type is a member of the `Literal` class, which defines a conversion from the compile-time `Integer` type to `Complex` type with the `fromInteger` function. This function converts the Integer to the real part, and sets the imaginary part to 0.

```
instance Literal#( Complex#(any_type) )
   provisos( Literal#(any_type) );
```

SaturatingArith The `SaturatingArith` class provides the functions `satPlus`, `satMinus`, `boundedPlus`, and `boundedMinus`. These are modified plus and minus functions which saturate to values defined by the `SaturationMode` when the operations would otherwise overflow or wrap-around. The type of the complex value (`any_type`) must be in the `SaturatingArith` class.

```
instance SaturatingArith#(Complex#(any_type))
   provisos (SaturatingArith#(any_type));
```

FShow An instance of `FShow` is available provided `any_type` is a member of `FShow` as well.

```
instance FShow#(Complex#(any_type))
   provisos (FShow#(any_type));
   function Fmt fshow (Complex#(any_type) x);
      return $format("<C ", fshow(x.rel), ",", fshow(x.img), ">");
   endfunction
endinstance
```

Functions

`cmplx`	A simple constructor function is provided to set the fields.
	`function Complex#(a_type) cmplx( a_type realA, a_type imagA ) ;`

<table>
<tr><td rowspan="2"><code>cmplxMap</code></td><td>Applies a function to each part of the complex structure. This is useful for operations such as <code>extend</code>, <code>truncate</code>, etc.</td></tr>
<tr><td><pre>function Complex#(b_type) cmplxMap(
                            function b_type mapFunc(a_type x),
                            Complex#(a_type) cin) ;</pre></td></tr>
</table>

<table>
<tr><td rowspan="2"><code>cmplxSwap</code></td><td>Exchanges the real and imaginary parts.</td></tr>
<tr><td><pre>function Complex#(a_type) cmplxSwap(Complex#(a_type) cin) ;</pre></td></tr>
</table>

<table>
<tr><td rowspan="2"><code>cmplxConj</code></td><td>Negates the imaginary part.</td></tr>
<tr><td><pre>function Complex#(a_type) cmplxConj(Complex#(a_type) cin) ;</pre></td></tr>
</table>

<table>
<tr><td rowspan="2"><code>cmplxWrite</code></td><td>Displays a complex number given a prefix string, an infix string, a postscript string, and an Action function which writes each part. <code>cmplxWrite</code> is of type Action and can only be invoked in Action contexts such as Rules and Actions methods.</td></tr>
<tr><td><pre>function Action cmplxWrite(String pre,
                           String infix,
                           String post,
                           function Action writeaFunc(a_type x),
                           Complex#(a_type) cin);</pre></td></tr>
</table>

Examples - Complex Numbers

```
// The following utility function is provided for writing data
// in decimal format. An example of its use is show below.

function Action writeInt( Int#(n) ain ) ;
   $write( "%0d", ain ) ;
endfunction

// Set the fields of the complex number using the constructor function cmplx
Complex#(Int#(6)) complex_value = cmplx(-2,7) ;

// Display complex_value  as ( -2 + 7i ).
// Note that writeInt is passed as an argument to the cmplxWrite function.
```

```
   cmplxWrite( "( ", " + ", "i)", writeInt, complex_value );

   // Swap the real and imaginary parts.
   swap_value = cmplxSwap( complex_value ) ;

   // Display the swapped values. This will display ( -7 + 2i).
   cmplxWrite( "( ", " + ", "i)", writeInt, swap_value );
```

3.5.4 FixedPoint

Package

```
import FixedPoint :: * ;
```

Description

The `FixedPoint` library package defines a type for representing fixed-point numbers and corresponding functions to operate and manipulate variables of this type.

A fixed-point number represents signed numbers which have a fixed number of binary digits (bits) before and after the binary point. The type constructor for a fixed-point number takes two numeric types as argument; the first (isize) defines the number of bits to the left of the binary point (the integer part), while the second (fsize) defines the number of bits to the right of the binary point, (the fractional part).

The following data structure defines this type, while some utility functions provide the reading of the integer and fractional parts.

```
typedef struct {
                Bit#(isize) i;
                Bit#(fsize) f;
                }
FixedPoint#(numeric type isize, numeric type fsize )
    deriving( Eq ) ;
```

Types and type classes

The `FixedPoint` type belongs to the following type classes; `Bits`, `Eq`, `Literal`, `RealLiteral`, `Arith`, `Ord`, `Bounded`, `Bitwise`, `SaturatingArith`, and `FShow`. Each type class definition includes functions which are then also defined for the data type. The Prelude library definitions (Section 2) describes which functions are defined for each type class.

Type Classes used by `FixedPoint`											
	`Bits`	`Eq`	`Literal`	`Real Literal`	`Arith`	`Ord`	`Bounded`	`Bit wise`	`Bit Reduce`	`Bit Extend`	`Format`
`FixedPoint`	√	√	√	√	√	√	√	√			√

Bits The type `FixedPoint` belongs to the `Bits` type class, which allows conversion from type `Bits` to type `FixedPoint`.

```
instance Bits#( FixedPoint#(isize, fsize), bsize )
   provisos ( Add#(isize, fsize, bsize) );
```

Literal The type `FixedPoint` belongs to the `Literal` type class, which allows conversion from (compile-time) type `Integer` to type `FixedPoint`. Note that only the integer part is assigned.

```
instance Literal#( FixedPoint#(isize, fsize) )
   provisos( Add#(isize, fsize, bsize) );
```

RealLiteral The type `FixedPoint` belongs to the `RealLiteral` type class, which allows conversion from type `Real` to type `FixedPoint`.

```
instance RealLiteral#( FixedPoint# (isize, fsize) )
```

Example:

```
    FixedPoint#(4,10) mypi = 3.1415926;  //Implied fromReal
    FixedPoint#(2,14) cx = fromReal(cos(pi/4));
```

Arith The type `FixedPoint` belongs to the `Arith` type class, hence the common infix operators (+, -, *, and /) are defined and can be used to manipulate variables of type `FixedPoint`. The arithmetic operator `%` is not defined.

```
instance Arith#( FixedPoint#(isize, fsize) )
   provisos( Add#(isize, fsize, bsize) ) ;
```

For multiplication (*) and quotient (/) , the operation is calculated in full precision and the result is then rounded and saturated to the resulting size. Both operators use the rounding function `fxptTruncateRoundSat`, with mode `Rnd_Zero`, `Sat_Bound`.

Ord In addition to equality and inequality comparisons, `FixedPoint` variables can be compared by the relational operators provided by the `Ord` type class. i.e., <, >, <=, and >=.

```
instance Ord#( FixedPoint#(isize, fsize) )
   provisos( Add#(isize, fsize, bsize) ) ;
```

Bounded The type `FixedPoint` belongs to the `Bounded` type class. The range of values, v, representable with a signed fixed-point number of type `FixedPoint#(isize, fsize)` is $+(2^{isize-1} - 2^{-fsize}) \leq v \leq -2^{isize-1}$. The function `epsilon` returns the smallest representable quantum by a specific type, 2^{-fsize}. For example, a variable v of type `FixedPoint#(2,3)` type can represent numbers from 1.875 ($1\frac{7}{8}$) to -2.0 in intervals of $\frac{1}{8} = 0.125$, i.e. epsilon is 0.125. The type `FixedPoint#(5,0)` is equivalent to `Int#(5)`.

```
instance Bounded#( FixedPoint#(isize, fsize) )
   provisos( Add#(isize, fsize, bsize) ) ;
```

`epsilon`	Returns the value of `epsilon` which is the smallest representable quantum by a specific type, 2^{-fsize}.
	`function FixedPoint#(isize, fsize) epsilon () ;`

Bitwise Left and right shifts are provided for `FixedPoint` variables as part of the `Bitwise` type class. Note that the shift right (`>>`) function does an arithmetic shift, thus preserving the sign of the operand. Note that a right shift of 1 is equivalent to a division by 2, except when the operand is equal to −`epsilon`. The functions `msb` and `lsb` are also provided. The other methods of `Bitwise` type class are not provided since they have no operational meaning on `FixedPoint` variables; the use of these generates an error message.

```
instance Bitwise#( FixedPoint#(isize, fsize) )
   provisos( Add#(isize, fsize, bsize) );
```

SaturatingArith The `SaturatingArith` class provides the functions `satPlus`, `satMinus`, `boundedPlus`, and `boundedMinus`. These are modified plus and minus functions which saturate to values defined by the `SaturationMode` when the operations would otherwise overflow or wrap-around.

```
instance SaturatingArith#(FixedPoint#(isize, fsize));
```

FShow The `FShow` class provides the function `fshow` which can be applied to a type to create an associated `Fmt` representation.

```
instance FShow#(FixedPoint#(i,f));
   function Fmt fshow (FixedPoint#(i,f) value);
      Int#(i) i_part = fxptGetInt(value);
      UInt#(f) f_part = fxptGetFrac(value);
      return $format("<FP %b.%b>", i_part, f_part);
   endfunction
endinstance
```

Functions

Utility functions are provided to extract the integer and fractional parts.

`fxptGetInt`	Extracts the integer part of the `FixedPoint` number.
	`function Int#(isize) fxptGetInt ( FixedPoint#(isize, fsize) x );`

`fxptGetFrac`	Extracts the fractional part of the `FixedPoint` number.
	`function UInt#(fsize) fxptGetFrac ( FixedPoint#(isize, fsize) x );`

To convert run-time `Int` and `UInt` values to type `FixedPoint`, the following conversion functions are provided. Both of these functions invoke the necessary extension of the source operand.

`fromInt`	Converts run-time `Int` values to type `FixedPoint`.
	`function FixedPoint#(ir,fr) fromInt( Int#(ia) inta )` `   provisos ( Add#(ia, xxA, ir ) );       // ir >= ia`

`fromUInt`	Converts run-time `UInt` values to type `FixedPoint`.

	`function FixedPoint#(ir,fr) fromUInt( UInt#(ia) uinta )`
	`   provisos ( Add#(ia,  1, ia1),          // ia1 = ia + 1`
	`              Add#(ia1,xxB, ir ) );       // ir >= ia1`

Non-integer compile time constants may be specified by a rational number which is a ratio of two integers. For example, one-third may be specified by `fromRational(1,3)`.

`fromRational`	Specify a `FixedPoint` with a rational number which is the ratio of two integers.
	`function FixedPoint#(isize, fsize) fromRational( Integer numerator, Integer denominator) provisos ( Add#(isize, fsize, bsize ) ) ;`

```
function FixedPoint#(isize, fsize) fromRational(
                 Integer numerator, Integer denominator)
   provisos ( Add#(isize, fsize, bsize ) ) ;
```

At times, full precision Arithmetic functions may be required, where the operands are not the same type (sizes), as is required for the infix `Arith` operators. These functions do not overflow on the result.

`fxptAdd`

Function for full precision addition. The operands do not have to be of the same type (size) and there is no overflow on the result.

```
function FixedPoint#(ri,rf)  fxptAdd( FixedPoint#(ai,af) a,
                                      FixedPoint#(bi,bf) b  )
   provisos (Max#(ai,bi,rim)     // ri = 1 + max(ai, bi)
            ,Add#(1,rim, ri)
            ,Max#(af,bf,rf));   // rf = max (af, bf)
```

`fxptSub`

Function for full precision subtraction where the operands do not have to be of the same type (size) and there is no overflow on the result.

```
function FixedPoint#(ri,rf)  fxptSub( FixedPoint#(ai,af) a,
                                     FixedPoint#(bi,bf) b  )
   provisos (Max#(ai,bi,rim)     // ri = 1 + max(ai, bi)
            ,Add#(1,rim, ri)
            ,Max#(af,bf,rf));   // rf = max (af, bf)
```

`fxptMult`

Function for full precision multiplication, where the result is the sum of the field sizes of the operands. The operands do not have to be of the same type (size).

```
function FixedPoint#(ri,rf)  fxptMult( FixedPoint#(ai,af) x,
                                       FixedPoint#(bi,bf) y  )
   provisos ( Add#(ai,bi,ri)   // ri = ai + bi
            ,Add#(af,bf,rf)   // rf = af + bf
            ,Add#(ai,af,ab)
            ,Add#(bi,bf,bb)
            ,Add#(ab,bb,rb)
            ,Add#(ri,rf,rb) ) ;
```

fxptQuot	Function for full precision division where the operands do not have to be of the same type (size).	

	`function FixedPoint#(ri,rf)  fxptQuot (FixedPoint#(ai,af) a,`
	`                                     FixedPoint#(bi,bf) b  )`
	`   provisos (Add#(ai1,bf,ri)      // ri = ai + bf + 1`
	`            ,Add#(ai,1,ai1)`
	`            ,Add#(af,_xf,rf));   // rf >= af`	

`fxptTruncate` is a general truncate function which converts variables to `FixedPoint#(ai,af)` to type `FixedPoint#(ri,rf)`, where $ai \geq ri$ and $af \geq rf$. This function truncates bits as appropriate from the most significant integer bits and the least significant fractional bits.

fxptTruncate	Truncates bits as appropriate from the most significant integer bits and the least significant fractional bits.

	`function FixedPoint#(ri,rf) fxptTruncate(`
	`             FixedPoint#(ai,af) a )`
	`   provisos( Add#(xxA,ri,ai),    // ai >= ri`
	`             Add#(xxB,rf,af));   // af >= rf`

Two saturating fixed-point truncation functions are provided: `fxptTruncateSat` and `fxptTruncateRoundSat`. They both use the `SaturationMode`, defined in Section 2.1.12, to determine the final result.

```
typedef enum { Sat_Wrap
              ,Sat_Bound
              ,Sat_Zero
              ,Sat_Symmetric
   } SaturationMode deriving (Bits, Eq);
```

fxptTruncateSat	A saturating fixed point truncation. If the value cannot be represented in its truncated form, an alternate value, `minBound` or `maxBound`, is selected based on `smode`.

	`function FixedPoint#(ri,rf) fxptTruncateSat (`
	`             SaturationMode smode, FixedPoint#(ai,af) din)`
	`   provisos (Add#(ri,idrop,ai)`
	`             ,Add#(rf,_f,af) );`

The function `fxptTruncateRoundSat` rounds the saturated value, as determined by the value of `rmode` of type `RoundMode`. The rounding only applies to the truncation of the fractional component of the fixed-point number, though it may cause a wrap or overflow to the integer component which requires saturation.

`fxptTruncateRoundSat`	A saturating fixed point truncate function which rounds the truncated fractional component as determined by the value of `rmode` (RoundMode). If the final value cannot be represented in its truncated form, the `minBound` or `maxBound` value is returned.	
	`function FixedPoint#(ri,rf) fxptTruncateRoundSat (RoundMode rmode, SaturationMode smode, FixedPoint#(ai,af) din) provisos (Add#(ri,idrop,ai) ,Add#(rf,fdrop,af) );`	

```
typedef enum {
              Rnd_Plus_Inf
              ,Rnd_Zero
              ,Rnd_Minus_Inf
              ,Rnd_Inf
              ,Rnd_Conv
              ,Rnd_Truncate
              ,Rnd_Truncate_Zero
          } RoundMode deriving (Bits, Eq);
```

These modes are equivalent to the SystemC values shown in the table below. The rounding mode determines how the value is rounded when the truncated value is equidistant between two representable values.

Rounding Modes			
RoundMode	SystemC Equivalent	Description	Action when truncated value equidistant between values
`Rnd_Plus_Inf`	SC_RND	Round to plus infinity	Always increment
`Rnd_Zero`	SC_RND_ZERO	Round to zero	Move towards reduced magnitude (decrement positive value, increment negative value)
`Rnd_Minus_Inf`	SC_RND_MIN_INF	Round to minus infinity	Always decrement
`Rnd_Inf`	SC_RND_INF	Round to infinity	Always increase magnitude
`Rnd_Conv`	SC_RND_CONV	Round to convergence	Alternate increment and decrement based on even and odd values
`Rnd_Truncate`	SC_TRN	Truncate, no rounding	
`Rnd_Truncate_Zero`	SC_TRN_ZERO	Truncate to zero	Move towards reduced magnitude

Consider what happens when you apply the function `fxptTruncateRoundSat` to a fixed-point number. The least significant fractional bits are dropped. If the dropped bits are non-zero, the remaining fractional component rounds towards the nearest representable value. If the remaining component is exactly equidistant between two representable values, the rounding mode (rmode) determines whether the value rounds up or down.

The following table displays the rounding value added to the LSB of the remaining fractional component. When the value is equidistant (1/2), the algorithm may be dependent on whether the value of the variable is positive or negative.

Rounding Value added to LSB of Remaining Fractional Component				
RoundMode	Value of Truncated Bits			
	< 1/2	1/2		> 1/2
		Pos	Neg	
`Rnd_Plus_Inf`	0	1	1	1
`Rnd_Zero`	0	0	1	1
`Rnd_Minus_Inf`	0	0	0	1
`Rnd_Inf`	0	1	0	1
`Rnd_Conv`				
Remaining LSB = 0	0	0	0	1
Remaining LSB = 1	0	1	1	1

The final two modes are truncates and are handled differently. The `Rnd_Truncate` mode simply drops the extra bits without changing the remaining number. The `Rnd_Truncate_Zero` mode decreases the magnitude of the variable, moving the value closer to 0. If the number is positive, the function simply drops the extra bits, if negative, 1 is added.

RoundMode	Sign of Argument		Description
	Positive	Negative	
`Rnd_Truncate`	0	0	Truncate extra bits, no rounding
`Rnd_Truncate_Zero`	0	1	Add 1 to negative number if truncated bits are non-zero

Example: Truncated values by Round type, where argument is `FixedPoint#(2,3)` type and result is a `FixedPoint#(2,1)` type. In this example, we're rounding to the nearest 1/2, as determined by RoundMode.

Result by RoundMode when SaturationMode = Sat_Wrap								
Argument		RoundMode						
Binary	Decimal	`Plus_Inf`	`Zero`	`Minus_Inf`	`Inf`	`Conv`	`Trunc`	`Trunc_Zero`
10.001	-1.875	-2.0	-2.0	-2.0	-2.0	-2.0	-2.0	-1.5
10.110	-1.250	-1.0	-1.0	-1.5	-1.5	-1.0	-1.5	-1.0
11.101	-0.375	-0.5	-0.5	-0.5	-0.5	-0.5	-0.5	0.0
00.011	0.375	0.5	0.5	0.5	0.5	0.5	0.0	0.0
01.001	1.250	1.5	1.0	1.0	1.5	1.0	1.0	1.0
01.111	1.875	-2.0	-2.0	-2.0	-2.0	-2.0	1.5	1.5

`fxptSignExtend`

A general sign extend function which converts variables of type `FixedPoint#(ai,af)` to type `FixedPoint#(ri,rf)`, where $ai \leq ri$ and $af \leq rf$. The integer part is sign extended, while additional 0 bits are added to least significant end of the fractional part.

```
function FixedPoint#(ri,rf) fxptSignExtend(
              FixedPoint#(ai,af) a )
   provisos( Add#(xxA,ai,ri),      // ri >= ai
             Add#(fdiff,af,rf));   // rf >= af
```

`fxptZeroExtend`	A general zero extend function.

	`function FixedPoint#(ri,rf) fxptZeroExtend(`
	`            FixedPoint#(ai,af) a )`
	`   provisos( Add#(xxA,ai,ri),    // ri >= ai`
	`            Add#(xxB,af,rf));   // rf >= af`

Displaying `FixedPoint` values in a simple bit notation would result in a difficult to read pattern. The following write utility function is provided to ease in their display. Note that the use of this function adds many multipliers and adders into the design which are only used for generating the output and not the actual circuit.

`fxptWrite`	Displays a `FixedPoint` value in a decimal format, where `fwidth` give the number of digits to the right of the decimal point. `fwidth` must be in the inclusive range of 0 to 10. The displayed result is truncated without rounding.

	`function Action fxptWrite( Integer fwidth,`
	`                           FixedPoint#(isize, fsize) a )`
	`   provisos( Add#(i, f, b),`
	`            Add#(33,f,ff)); // 33 extra bits for computations.`

Examples - Fixed Point Numbers

```
// The following code writes "x is 0.5156250"
FixedPoint#(1,6) x = half + epsilon ;
$write( "x is " ) ; fxptWrite( 7, x ) ; $display("" ) ;
```

A `Real` value can automatically be converted to a `FixedPoint` value:

```
FixedPoint#(3,10) foo = 2e-3;

FixedPoint#(2,3) x = 1.625 ;
```

3.5.5 NumberTypes

Package

```
import NumberTypes :: * ;
```

Description

The `NumberTypes` package defines two new number types for use as index types: `BuffIndex` and `WrapNumber`.

A `BuffIndex#(sz, ln)` is an unsigned integer which wraps around, where `sz` is the number of bits in its representation and `ln` is the size of the buffer it is to index. Often `sz` will be `TLog#(ln)`. `BuffIndex` is intended to be used as the index type for buffers of arbitrary size. The values of `BuffIndex` are not ordered; you cannot determine which of two values is ahead of the other because of the wrap-around.

A `WrapNumber#(sz)` is an unsigned integer which wraps around, where `sz` is the number of bits in its representation. The range is the entire value space (i.e.2^{sz}), but should be used in situations where at any time all valid values are in at most half of that space. The ordering of values can be defined taking wrap-around into account, so that the nearer distance apart is used to determine which value is ahead of the other.

Types and type classes

A `BuffIndex` has two numeric type parameters: the size in bits of the representation (`sz`), and the length of the buffer it is to index (`ln`).

```
typedef struct { UInt#(sz) bix; } BuffIndex#(numeric type sz, numeric type ln)
   deriving (Bits, Eq);
```

A `WrapNumber#(sz)` has a single numeric type parameter, `sz`, which is the size in bits of the representation.

```
typedef struct { UInt#(sz) wn; } WrapNumber#(numeric type sz)
      deriving (Bits, Eq, Arith, Literal, Bounded);
```

Both types belong to the `Bits`, `Eq`, `Arith`, and `Literal` typeclasses. The `WrapNumber` type also belongs to the `Ord` typeclass. Each type class definition includes functions which are then also defined for the data type. The Prelude library definitions (Section 2) describes which functions are defined for each type class.

Type Classes used by `BuffIndex` and `WrapNumber`									
	`Bits`	`Eq`	`Literal`	`Arith`	`Ord`	`Bounded`	`Bit` wise	`Bit` Reduction	`Bit` Extend
`WrapNumber`	√	√	√	√	√	√			
`BuffIndex`	√	√	√	√					

Literal Both `BuffIndex` and `WrapNumber` belong to the `Literal` typeclass, which allows conversion from (compile-time) type `Integer` to these types.

For the `BuffIndex` type, the `fromInteger` and `inLiteralRange` functions are defined as:

```
instance Literal#(BuffIndex#(sz,ln));
   function fromInteger(i) = BuffIndex {bix: fromInteger(i) };
   function inLiteralRange(x,i) = (i>=0 && i < valueof(ln));
endinstance
```

Arith The type class `Arith` defines the common infix operators. Addition and subtraction are the only meaningful arithmetic operations for `WrapNumber` and `BuffIndex`.

Ord `WrapNumber` belongs to the `Ord` typeclass, so values of `WrapNumber` can be compared by the relational operators <, >, <=, and >=. Since the ordering of `WrapNumber` types takes into account wrap-around, the nearer distance apart is used to determine which value is ahead of the other.

Functions

Utility functions to convert a `BuffIndex` to a `UInt` and for adding and subtracting `BuffIndex` and `UInt` values are provided.

`unwrapBI`	Converts a `BuffIndex` to a `UInt`	
	`function UInt#(sz) unwrapBI(BuffIndex#(sz,ln) x);`	

`addBIUInt`	Adds a `UInt` to a `BuffIndex`, returning a `BuffIndex`
	`function BuffIndex#(sz,ln) addBIUInt(BuffIndex#(sz,ln) bin, UInt#(sz) i);`

`sbtrctBIUInt`	Subtracts a `UInt` from a `BuffIndex`, returning a `BuffIndex`
	`function BuffIndex#(sz,ln) sbtrctBIUInt(BuffIndex#(sz,ln) bin, UInt#(sz) i);`

Utility functions to convert between a `WrapNumber` and a `UInt`, and a function to add a `UInt` to a `WrapNumber` are provided.

`wrap`	Converts a `UInt` to a `WrapNumber`
	`function WrapNumber#(sz) wrap(UInt#(sz) x) ;`

`unwrap`	Converts a `WrapNumber` to a `UInt`
	`function UInt#(sz) unwrap (WrapNumber#(sz) x);`

`addUInt`	Adds a `UInt` to a `WrapNumber`, returning a `WrapNumber`
	`function WrapNumber#(sz) addUInt(WrapNumber#(sz) wn, UInt#(sz) i) ;`

3.6 FSM

3.6.1 StmtFSM

Package

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

Description

The `StmtFSM` package provides a procedural way of defining finite state machines (FSMs) which are automatically synthesized.

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

The `Stmt` sublanguage

The state machine is automatically constructed from the procedural description given in the `Stmt` definition. Appropriate state counters are created and rules are generated internally, corresponding to the transition logic of the state machine. The use of rules for the intermediate state machine generation ensures that resource conflicts are identified and resolved, and that implicit conditions are properly checked before the execution of any action.

The names of generated rules (which may appear in conflict warnings) have suffixes of the form "l<nn>c<nn>", where the <nn> are line or column numbers, referring to the statement which gave rise to the rule.

A term in the `Stmt` sublanguage is an expression, introduced at the outermost level by the keywords `seq` or `par`. Note that within the sublanguage, `if`, `while` and `for` statements are interpreted as statements in the sublanguage and not as ordinary statements, except when enclosed within `action/endaction` keywords.

exprPrimary	::=	*seqFsmStmt* | *parFsmStmt*
fsmStmt	::=	*exprFsmStmt*
	|	*seqFsmStmt*
	|	*parFsmStmt*
	|	*ifFsmStmt*
	|	*whileFsmStmt*
	|	*repeatFsmStmt*
	|	*forFsmStmt*
	|	*returnFsmStmt*
exprFsmStmt	::=	*regWrite* `;`
	|	*expression* `;`
seqFsmStmt	::=	`seq` *fsmStmt* { *fsmStmt* } `endseq`
parFsmStmt	::=	`par` *fsmStmt* { *fsmStmt* } `endpar`
ifFsmStmt	::=	`if` *expression* *fsmStmt*
		[`else` *fsmStmt*]
whileFsmStmt	::=	`while (` *expression* `)`
		loopBodyFsmStmt
forFsmStmt	::=	`for (` *fsmStmt* `;` *expression* `;` *fsmStmt* `)`
		loopBodyFsmStmt
returnFsmStmt	::=	`return ;`
repeatFsmStmt	::=	`repeat (` *expression* `)`
		loopBodyFsmStmt
loopBodyFsmStmt	::=	*fsmStmt*
	|	`break ;`
	|	`continue ;`

The simplest kind of statement is an *exprFsmStmt*, which can be a register assignment or, more generally, any expression of type `Action` (including action method calls and `action-endaction` blocks or of type `Stmt`. Statements of type `Action` execute within exactly one clock cycle, but of course the scheduling semantics may affect exactly which clock cycle it executes in. For example, if the actions in a statement interfere with actions in some other rule, the statement may be delayed by the schedule until there is no interference. In all the descriptions of statements below, the descriptions of time taken by a construct are minimum times; they could take longer because of scheduling semantics.

Statements can be composed into sequential, parallel, conditional and loop forms. In the sequential form (`seq-endseq`), the contained statements are executed one after the other. The `seq` block terminates when its last contained statement terminates, and the total time (number of clocks) is equal to the sum of the individual statement times.

In the parallel form (`par-endpar`), the contained statements ("threads") are all executed in parallel. Statements in each thread may or may not be executed simultaneously with statements in other threads, depending on scheduling conflicts; if they cannot be executed simultaneously they will be interleaved, in accordance with normal scheduling. The entire `par` block terminates when the last of its contained threads terminates, and the minimum total time (number of clocks) is equal to the maximum of the individual thread times.

In the conditional form (`if (`b`)` s_1 `else` s_2), the boolean expression b is first evaluated. If true, s_1 is executed, otherwise s_2 (if present) is executed. The total time taken is t cycles, if the chosen branch takes t cycles.

In the `while (`b`)` s loop form, the boolean expression b is first evaluated. If true, s is executed, and the loop is repeated. Each time the condition evaluates true , the loop body is executed, so the total time is $n \times t$ cycles, where n is the number of times the loop is executed (possibly zero) and t is the time for the loop body statement.

The `for (`s_1`;`b`;`s_2`)` s_B loop form is equivalent to:

s_1`; while (`b`) seq` s_B`;` s_2 `endseq`

i.e., the initializer s_1 is executed first. Then, the condition b is executed and, if true, the loop body s_B is executed followed by the "increment" statement s_2. The b, s_B, s_2 sequence is repeated as long as b evaluates true.

Similarly, the `repeat (`n`)` s_B loop form is equivalent to:

`while (`$repeat\_count < n$`) seq` s_B`;` $repeat\_count <= repeat\_count + 1$ `endseq`

where the value of $repeat\_count$ is initialized to 0. During execution, the condition ($repeat\_count < n$) is executed and, if true, the loop body s_B is executed followed by the "increment" statement $repeat\_count <= repeat\_count + 1$. The sequence is repeated as long as $repeat\_count < n$ evaluates true.

In all the loop forms, the loop body statements can contain the keywords `continue` or `break`, with the usual semantics, i.e., `continue` immediately jumps to the start of the next iteration, whereas `break` jumps out of the loop to the loop sequel.

It is important to note that this use of loops, within a `Stmt` context, expresses time-based (temporal) behavior.

Interfaces and Methods

Two interfaces are defined with this package, `FSM` and `Once`. The `FSM` interface defines a basic state machine interface while the `Once` interface encapsulates the notion of an action that should only be performed once. A `Stmt` value can be instatiated into a module that presents an interface of type `FSM`.

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.

Modules

Instantiation is performed by passing a `Stmt` value into the module constructor `mkFSM`. The state machine is automatically constructed from the procedural decription given in the definition described by state machine of type `Stmt` named `seq_stmt`. During construction, one or more registers of appropriate widths are created to track state execution. Upon `start` action, the registers are loaded and subsequent state changes then decrement the registers.

```
module mkFSM#( Stmt seq_stmt ) ( FSM );
```

The `mkFSMWithPred` module is like `mkFSM` above, except that the module constructor takes an additional boolean argument (the predicate). The predicate condition is added to the condition of each rule generated to create the FSM. This capability is useful when using the FSM in conjuction with other rules and/or FSMs. It allows the designer to explicitly specify to the compiler the conditions under which the FSM will run. This can be used to eliminate spurious rule conflict warnings (between rules in the FSM and other rules in the design).

```
module mkFSMWithPred#( Stmt seq_stmt, Bool pred ) ( FSM );
```

The `mkAutoFSM` module is also like `mkFSM` above, except the state machine runs automatically immediately after reset and a `$finish(0)` is called upon completion. This is useful for test benches. Thus, it has no interface, that is, it has an empty interface.

```
module mkAutoFSM#( seq_stmt ) ();
```

The `mkOnce` function is used to create a `Once` interface where the action argument has been encapsulated and will be performed when `start` is called.

```
module mkOnce#( Action a ) ( Once );
```

The implementation for `Once` is a 1 bit state machine (with a state register named `onceReady`) allowing the action argument to occur only one time. The ready bit is initially `True` and then cleared when the action is performed. It might not be performed right away, because of implicit conditions or scheduling conflicts.

Name	BSV Module Declaration	Description
`mkFSM`	`module mkFSM#(Stmt seq_stmt)(FSM);`	Instantiate a `Stmt` value into a module that presents an interface of type `FSM`.
`mkFSMWithPred`	`module mkFSMWithPred#(Stmt seq_stmt,` `Bool pred)(FSM);`	Like `mkFSM`, except that the module constructor takes an additional predicate condition as an argument. The predicate condition is added to the condition of each rule generated to create the FSM.
`mkAutoFSM`	`module mkAutoFSM#(Stmt seq_stmt)();`	Like `mkFSM`, except that state machine simulation is automatically started and a `$finish(0)`) is called upon completion.
`mkOnce`	`module mkOnce#( Action a )( Once );`	Used to create a `Once` interface where the action argument has been encapsulated and will be performed when `start` is called.

Functions

There are two functions, `await` and `delay`, provided by the `StmtFSM` package.

The `await` function is used to create an action which can only execute when the condition is `True`. The action does not do anything. `await` is useful to block the execution of an action until a condition becomes `True`.

The `delay` function is used to execute `noAction` for a specified number of cycles. The function is provided the value of the delay and returns a `Stmt`.

Name	Function Declaration	Description
`await`	`function Action await( Bool cond ) ;`	Creates an Action which does nothing, but can only execute when the condition is `True`.
`delay`	`function Stmt delay( a_type value ) ;`	Creates a `Stmt` which executes `noAction` for `value` number of cycles. `a_type` must be in the Arith class and Bits class and <32 bits.

Example - Initializing a single-ported SRAM.

Since the SRAM has only a single port, we can write to only one location in each clock. Hence, we need to express a temporal sequence of writes for all the locations to be initialized.

```
Reg#(int) i  <-  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
```

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
```

3.7 Connectivity

The packages in this section provide useful components, primarily interfaces, to connect hardware elements in a design.

The basic interfaces, `Get` and `Put` are defined in the package `GetPut`. The typeclass `Connectable` indicates that two related types can be connected together. The package `ClientServer` provides interfaces using `Get` and `Put` for modules that have a request-response type of interface. The package `CGetPut` defines a type of the `Get` and `Put` interfaces that is implemented with a credit based FIFO.

3.7.1 GetPut

Package

```
import GetPut :: *;
```

Description

A common paradigm between two blocks is the get/put mechanism: one side *gets* or retrieves an item from an interface and the other side *puts* or gives an item to an interface. These types of interfaces are used in *Transaction Level Modeling* or TLM for short. This pattern is so common in system design that BSV provides the `GetPut` library package for this purpose.

The `GetPut` package provides basic interfaces to implement the TLM paradigm, along with interface transformer functions and modules to transform to/from FIFO implementations. The `ClientServer` package in Section 3.7.3 defines more complex interfaces based on the `Get` and `Put` interfaces to support request-response interfaces. The `GetPut` package must be imported when using the `ClientServer` package.

Typeclasses

The `GetPut` package defines two typeclasses: `ToGet` and `ToPut`. The types with instances defined in these typeclasses provide the functions `toGet` and `toPut`, used to create associated `Get` and `Put` interfaces from these other types.

`ToGet` defines the class to which the function `toGet` can be applied to create an associated `Get` interface.

```
typeclass ToGet#(a, b);
   function Get#(b) toGet(a ax);
endtypeclass
```

ToPut defines the class to which the function toPut can be applied to create an associated Put interface.

```
typeclass ToPut#(a, b);
   function Put#(b) toPut(a ax);
endtypeclass
```

Instances of ToGet and ToPut are defined for the following interfaces:

Defined Instances for ToGet and ToPut			
Type (Interface)	toGet	toPut	Comments
a	✓		toGet returns value a
ActionValue#(a)	✓		toGet performs the Action and returns the value
function Action fn(a)		✓	toPut calls Action function fn with argument a
Get#(a)	✓		identity function: returns Get#(a)
Put#(a)		✓	identity function: returns Put#(a)
Reg#(a)	✓	✓	toGet returns _read, toPut calls _write
RWire#(a)	✓	✓	toGet returns wget, toPut calls wset
ReadOnly#(a)	✓		toGet returns _read
FIFO#(a)	✓	✓	toGet calls deq returns first, toPut calls enq
FIFOF#(a)	✓	✓	toGet calls deq returns first, toPut calls enq
SyncFIFOIfc#(a)	✓	✓	toGet calls deq returns first, toPut calls enq
FIFOLevelIfc#(a)	✓	✓	toGet calls deq returns first, toPut calls enq
SyncFIFOLevelIfc#(a)	✓	✓	toGet calls deq returns first, toPut calls enq
FIFOCountIfc#(a)	✓	✓	toGet calls deq returns first, toPut calls enq
SyncFIFOCountIfc#(a)	✓	✓	toGet calls deq returns first, toPut calls enq

Example - Using toPut

```
module mkTop (Put#(UInt#(64)));
   Reg#(UInt#(64)) inValue <- mkReg(0);
   Reg#(Bool) startit <- mkReg(True);
   ...
   StimIfc     stim_gen <- mkStimulusGen;

   rule startTb (startit && inValue!=0);
      // Get the value
      let val = inValue;
      stim_gen.start(val);
      startit <= False;
   endrule
   ...
   return (toPut(asReg(inValue)));
endmodule: mkTop
```

Interfaces and methods

The Get interface defines the get method, similar to a dequeue, which retrieves an item from an interface and removes it at the same time. The Put interface defines the put method, similar to an enqueue, which gives an item to an interface. Also provided is the GetS interface, which defines separate methods for the dequeue (deq) and retreiving the item (first) from the interface.

You can design your own Get and Put interfaces with implicit conditions on the get/put to ensure that the get/put is not performed when the module is not ready. This would ensure that a rule containing get method would not fire if the element associated with it is empty and that a rule containing put method would not fire if the element is full.

The following interfaces are defined in the GetPut package. They each take a single parameter, element_type which must be in the Bits typeclass.

Interfaces defined in GetPut			
Interface Name	Description	Methods	Type
Get	Retrieves item from an interface	get	ActionValue
Put	Adds an item to an interface	put	Action
GetS	Retrieves an item from an interface with 2 methods, separating the return of the value from the dequeue	first deq	Value Action
GetPut	Combination of a Get and a Put in a Tuple2	get put	ActionValue Action

Get

The Get interface is where you retrieve (get) data from an object. The Get interface is provides a single ActionValue method, get, which retrieves an item of data from an interface and removes it from the object. A get is similar to a dequeue, but it can be associated with any interface. A Get interface is more abstract than a FIFO interface; it does not describe the underlying hardware.

Get				
Method			Argument	
Name	Type	Description	Name	Description
get	ActionValue	returns an item from an interface and removes it from the object		

```
interface Get#(type element_type);
    method ActionValue#(element_type) get();
endinterface: Get
```

Example - adding your own Get interface:

```
module mkMyFifoUpstream (Get#(int));
...
   method ActionValue#(int) get();
       f.deq;
       return f.first;
   endmethod
```

Put

The Put interface is where you can give (put) data to an object. The Put interface provides a single Action method, put, which gives an item to an interface. A put is similar to an enqueue, but it can be associated with any interface. A Put interface is more abstract than a FIFO interface; it does not describe the underlying hardware.

Put				
Method			Argument	
Name	Type	Description	Name	Description
put	Action	gives an item to an interface	x1	data to be added to the object must be of type element_type

```
interface Put#(type element_type);
    method Action put(element_type x1);
endinterface: Put
```

Example - adding your own Put interface:

```
module mkMyFifoDownstream (Put#(int));
...
   method Action put(int x);
       F.enq(x);
   endmethod
```

GetS

The GetS interface is like a Get interface, but separates the get method into two methods: a first and a deq.

GetS				
Method			Argument	
Name	Type	Description	Name	Description
first	Value	returns an item from the interface		
deq	Action	Removes the item from the interface		

```
interface GetS#(type element_type);
    method element_type first();
    method Action deq();
endinterface: GetS
```

GetPut

The library also defines an interface GetPut which associates Get and Put interfaces into a Tuple2.

```
typedef Tuple2#(Get#(element_type), Put#(element_type)) GetPut#(type element_type);
```

Type classes

The class Connectable (Section 3.7.2) is meant to indicate that two related types can be connected in some way. It does not specify the nature of the connection.

A Get and Put is an example of connectable items. One object will put an element into the interface and the other object will get the element from the interface.

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

Modules

There are three modules provided by the GetPut package which provide the GetPut interface with a type of FIFO. These FIFOs use Get and Put interfaces instead of the usual enq interfaces. To use any of these modules the FIFO package must be imported. You can also write your own modules providing a GetPut interface for other hardware structures.

mkGPFIFO	Creates a FIFO of depth 2 with a `GetPut` interface.	
	`module mkGPFIFO (GetPut#(element_type))` `  provisos (Bits#(element_type, width_elem));`	

mkGPFIFO1	Creates a FIFO of depth 1 with a `GetPut` interface.
	`module mkGPFIFO1 (GetPut#(element_type))` `  provisos (Bits#(element_type, width_elem));`

mkGPSizedFIFO	Creates a FIFO of depth n with a `GetPut` interface.
	`module mkGPSizedFIFO# (Integer n) (GetPut#(element_type))` `  provisos (Bits#(element_type, width_elem));`

Functions

There are three functions defined in the `GetPut` package that change a `FIFO` interface to a `Get`, `GetS` or `Put` interface. Given a FIFO we can use the function `fifoToGet` to obtain a `Get` interface, which is a combination of `deq` and `first`. Given a FIFO we can use the function `fifoToPut` to obtain a `Put` interface using `enq`. The functions `toGet` and `toPut` (3.7.1) are recommended instead of the `fifoToGet` and `fifoToPut` functions. The function `fifoToGetS` returns the `GetS` methods as fifo methods.

fifoToGet	Returns a `Get` interface. It is recommended that you use the function `toGet` (3.7.1) instead of this function.
	`function Get#(element_type) fifoToGet(FIFO#(element_type) f);`

fifoToGetS	Returns a `GetS` interface.
	`function GetS#(element_type) fifoToGet(FIFO#(element_type) f);`

fifoToPut	Returns a `Put` interface. It is recommended that you use the function `toPut` (3.7.1) instead of this function.
	`function Put#(element_type) fifoToPut(FIFO#(element_type) f);`

Example of creating a FIFO with a `GetPut` interface

```
import GetPut::*;
import FIFO::*;

...
module mkMyModule (MyInterface);
   GetPut#(StatusInfo) aFifoOfStatusInfoStructures <- mkGPFIFO;
...
endmodule: mkMyModule
```

Example of a protocol monitor

This is an example of how you might write a protocol monitor that watches bus traffic between a bus and a bus target device

```
import GetPut::*;
import FIFO::*;

// Watch bus traffic beteween a bus and a bus target
interface ProtocolMonitorIfc;
   // These subinterfaces are defined inside the module
   interface Put#(Bus_to_Target_Request)  bus_to_targ_req_ifc;
   interface Put#(Target_to_Bus_Response) targ_to_bus_resp_ifc;
endinterface
...
module mkProtocolMonitor (ProtocolMonitorIfc);
   // Input FIFOs that have Put interfaces added a few lines down
   FIFO#(Bus_to_Target_Request) bus_to_targ_reqs <- mkFIFO;
   FIFO#(Target_To_Bus_Response) targ_to_bus_resps <- mkFIFO;
...
   // Define the subinterfaces: attach Put interfaces to the FIFOs, and
   //   then make those the module interfaces
   interface bus_to_targ_req_ifc = fifoToPut (bus_to_targ_reqs);
   interface targ_to_bus_resp_ifc = fifoToPut (targ_to_bus_resps);
end module: mkProtocolMonitor

// Top-level module: connect mkProtocolMonitor to the system:
module mkSys (Empty);
   ProtocolMonitorIfc pmon <- mkProtocolInterface;
...
   rule pass_bus_req_to_interface;
       let x <- bus.bus_ifc.get;     // definition not shown
       pmon.but_to_targ_ifc.put (x);
   endrule
...
endmodule: mkSys
```

3.7.2 Connectable

Package

```
import Connectable :: * ;
```

Description

The Connectable package contains the definitions for the class Connectable and instances of Connectables.

Types and Type-Classes

The class Connectable is meant to indicate that two related types can be connected in some way. It does not specify the nature of the connection. The Connectables type class defines the module mkConnection, which is used to connect the pairs.

```
typeclass Connectable#(type a, type b);
    module mkConnection#(a x1, b x2)(Empty);
endtypeclass
```

Instances

Get and Put One instance of the typeclass of Connectable is Get and Put. One object will put an element into an interface and the other object will get the element from the interface.

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

Tuples If we have Tuple2 of connectable items then the pair is also connectable, simply by connecting the individual items.

```
instance Connectable#(Tuple2#(a, c), Tuple2#(b, d))
  provisos (Connectable#(a, b), Connectable#(c, d));
```

The proviso shows that the first component of one tuple connects to the first component of the other tuple, likewise, the second components connect as well. In the above statement, a connects to b and c connects to d. This is used by ClientServer (Section 3.7.3) to connect the Get of the Client to the Put of the Server and visa-versa.

This is extensible to all Tuples (Tuple3, Tuple4, etc.). As long as the items are connectable, the Tuples are connectable.

Vector Two Vectors are connectable if their elements are connectable.

```
instance Connectable#(Vector#(n, a), Vector#(n, b))
  provisos (Connectable#(a, b));
```

ListN Two ListNs are connectable if their elements are connectable.

```
instance Connectable#(ListN#(n, a), ListN#(n, b))
  provisos (Connectable#(a, b));
```

Action, ActionValue An ActionValue method (or function) which produces a value can be connected to an Action method (or function) which takes that value as an argument.

```
instance Connectable#(ActionValue#(a), function Action f(a x));

instance Connectable#(function Action f(a x), ActionValue#(a));
```

A Value method (or value) can be connected to an Action method (or function) which takes that value as an argument.

```
instance Connectable#(a, function Action f(a x));

instance Connectable#(function Action f(a x), a);
```

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

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

3.7.3 ClientServer

Package

```
import ClientServer :: * ;
```

Description

The ClientServer package provides two interfaces, Client and Server which can be used to define modules which have a request-response type of interface. The GetPut package must be imported when using this package because the Get and Put interface types are used.

Interfaces and methods

The interfaces Client and Server can be used for modules that have a request-response type of interface (e.g. a RAM). The server accepts requests and generates responses, the client accepts responces and generates requests. There are no assumptions about how many (if any) responses a request generates

Interfaces			
Interface Name	Parameter name	Parameter Description	Restrictions
Client	*req_type*	type of the client request	must be in the Bits class
	resp_type	type of the client response	must be in the Bits class
Server	*req_type*	type of the server request	must be in the Bits class
	resp_type	type of the server response	must be in the Bits class

Client

The Client interface provides two subinterfaces, request and response. From a Client, one gets a request and puts a response.

Client SubInterface		
Name	Type	Description
request	Get#(req_type)	the interface through which the outside world retrieves (gets) a request
response	Put#(resp_type)	the interface through which the outside world returns (puts) a response

```
interface Client#(type req_type, type resp_type);
    interface Get#(req_type)  request;
    interface Put#(resp_type) response;
endinterface: Client
```

Server

The Server interface provides two subinterfaces, request and response. From a Server, one puts a request and gets a response.

Server SubInterface		
Name	Type	Description
request	Put#(req_type)	the interface through which the outside world returns (puts) a request
response	Get#(resp_type)	the interface through which the outside world retrieves (gets) a response

```
interface Server#(type req_type, type resp_type);
    interface Put#(req_type)  request;
    interface Get#(resp_type) response;
endinterface: Server
```

`ClientServer`

A `Client` can be connected to a `Server` and vice versa. The `request` (which is a `Get` interface) of the client will connect to `response` (which is a `Put` interface) of the `Server`. By making the `ClientServer` tuple an instance of the `Connectable` typeclass, you can connect the `Get` of the client to the `Put` of the server, and the `Put` of the client to the `Get` of the server.

```
instance Connectable#(Client#(req_type, resp_type), Server#(req_type, resp_type));
instance Connectable#(Server#(req_type, resp_type), Client#(req_type, resp_type));
```

This `Tuple2` can be redefined to be called `ClientServer`

```
typedef Tuple2#(Client#(req_type, resp_type), Server#(req_type,resp_type))
                ClientServer#(type req_type, type resp_type);
```

Example Connecting a bus to a target

```
interface Bus_Ifc;
   interface Server#(RQ, RS) to_initor ;
   interface Client#(RQ, RS) to_targ;
endinterface

typedef Server#(RQ, RS) Target_Ifc;
typedef Client#(RQ, RS) Initiator_Ifc;

module mkSys (Empty);
   // Instantiate subsystems
   Bus_Ifc             bus       <- mkBus;
   Target_Ifc          targ      <- mkTarget;
   Initiator_Ifc       initor    <- mkInitiator;

   // Connect bus and targ (to_targ is a Client ifc, targ is a Server ifc)
   Empty x <- mkConnection (bus.to_targ, targ);

   // Connect bus and initiator (to_initor is a Server ifc, initor is a Client ifc)
   mkConnection (bus.to_initor, initor);
   // Since mkConnection returns an interface of type Empty, it does
   // not need to be specified (but may be as above)
...
endmodule: mkSys
```

Functions

The `ClientServer` package includes functions which return a `Client` interface or a `Server` interface from separate request and response interfaces taken as arguments. The argument interfaces must be able to be converted to `Get` and `Put` interfaces, as indicated by the `ToGet` and `ToPut` provisos.

toGPClient	Function that returns a Client interface from two arguments (request and response interfaces). The arguments must be able to be converted to Get and Put interfaces.	

	`function Client#(req_type, resp_type)`
	`  toGPClient(req_ifc_type req_ifc, resp_ifc_type resp_ifc)`
	`provisos (ToGet#(req_ifc_type, req_type), ToPut#(resp_ifc_type, resp_type));`	

toGPServer	Function that returns a Server interface from two arguments (request and response interfaces). The arguments must be able to be converted to Get and Put interfaces.

	`function Server#(req_type, resp_type)`
	`  toGPServer(req_ifc_type req_ifc, resp_ifc_type resp_ifc)`
	`provisos (ToPut#(req_ifc_type, req_type), ToGet#(resp_ifc_type,resp_type));`

3.7.4 Memory

Package

```
import Memory :: * ;
```

Description

The Memory package provides the memory structures MemoryRequest and MemoryResponse which can be used to define a Client/Server memory structure.

Types and type classes

A MemoryRequest is a polymorphic structure of a request containing a write bit, a byte enable (byteen), the address and the data for a memory request:

```
typedef struct {
   Bool    write;
   Bit#(TDiv#(d,8)) byteen;
   Bit#(a) address;
   Bit#(d) data;
} MemoryRequest#(numeric type a, numeric type d) deriving (Bits, Eq);
```

The MemoryResponse contains the data:

```
typedef struct {
   Bit#(d)          data;
} MemoryResponse#(numeric type d) deriving (Bits, Eq);
```

Interfaces and Methods

The interfaces MemoryServer and MemoryClient are defined from the Server and Client interfaces defined in ClientServer package (Section 3.7.3) using the MemoryRequest and MemoryResponse types.

The MemoryServer accepts requests and generates responses, the MemoryClient accepts responces and generates requests. There are no assumptions about how many (if any) responses a request generates.

```
typedef Server#(MemoryRequest#(a,d), MemoryResponse#(d))
                MemoryServer#(numeric type a, numeric type d);

typedef Client#(MemoryRequest#(a,d), MemoryResponse#(d))
                MemoryClient#(numeric type a, numeric type d);
```

Default value instances are defined for both `MemoryRequest` and `MemoryResponse`:

```
instance DefaultValue#(MemoryRequest#(a,d));
   defaultValue = MemoryRequest {
      write:    False,
      byteen:   '1,
      address:  0,
      data:     0
      };
endinstance

instance DefaultValue#(MemoryResponse#(d));
   defaultValue = MemoryResponse {
      data:     0
      };
endinstance
```

An instance of the `TieOff` class (Section 3.8.10) is defined for `MemoryClient`:

```
instance TieOff#(MemoryClient#(a, d));
```

Functions

<table>
<tr><td rowspan="2"><code>updateDataWithMask</code></td><td>Replaces the original data with new data. The data must be divisible an 8-bit multiple of the mask. The mask indicates which bits to replace.</td></tr>
<tr><td><pre>function Bit#(d) updateDataWithMask(Bit#(d) origdata
                                   , Bit#(d) newdata
                                   , Bit#(d8) mask);</pre></td></tr>
</table>

3.7.5 CGetPut

Package

```
import CGetPut :: * ;
```

Description

The interfaces `CGet` and `CPut` are similar to `Get` and `Put`, but the interconnection of them (via `Connectable`) is implemented with a credit-based FIFO. This means that the `CGet` and `CPut` interfaces have completely registered input and outputs, and furthermore that additional register buffers can be introduced in the connection path without any ill effect (except an increase in latency, of course).

In the absence of additional register buffers, the round-trip time for communication between the two interfaces is 4 clock cycles. Call this number r. The first argument to the type, n, specifies that transfers will occur for a fraction n/r of clock cycles (note that the used cycles will not necessarily be evenly spaced). n also specifies the depth of the buffer used in the receiving interface (the transmitter side always has only a single buffer). So (in the absence of additional buffers) use $n = 4$ to allow full-bandwidth transmission, at the cost of sufficient registers for quadruple buffering at one end; use $n = 1$ for minimal use of registers, at the cost of reducing the bandwidth to one quarter; use intermediate values to select the optimal trade-off if appropriate.

Interfaces and methods

The interface types are abstract to avoid any improper use of the credit signaling protocol.

Interfaces			
Interface Name	Parameter name	Parameter Description	Restrictions
`CGet`	*n*	depth of the buffer used in the receiving interface	must be a numeric type
	element_type	type of the element being retrieved by the `CGet`	must be in `Bits` class
`CPut`	*n*	depth of the buffer used in the receiving interface	must be a numeric type
	element_type	type of the element being added by the `CPut`	must be in `Bits` class

- `CGet`

```
interface CGet#(numeric type n, type element_type);
      ...Abstract...
```

- `CPut`

```
interface CPut#(numeric type n, type element_type);
      ...Abstract...
```

- Connectables

  The `CGet` and `CPut` interfaces are connectable.

```
instance Connectable#(CGet#(n, element_type), CPut#(n, element_type));

instance Connectable#(CPut#(n, element_type), CGet#(n, element_type));
```

- CClient and CServer

  The same idea may be extended to clients and servers.

```
interface CClient#(type n, type req_type, type resp_type);
interface CServer#(type n, type req_type, type resp_type);
```

Modules

`mkCGetPut`	Create an n depth FIFO with a `CGet` interface on the dequeue side and a `Put` interface on the enqueue side.
	`module mkCGetPut(Tuple2#(CGet#(n, element_type), Put#(element_type))) provisos (Bits#(element_type));`

`mkGetCPut`	Create an n depth FIFO with a `Get` interface on the dequeue side and a `CPut` interface on the enqueue side.
	`module mkGetCPut(Tuple2#(Get#(element_type),`
	`CPut#(n, element_type)))`
	`provisos (Bits#(element_type));`

`mkClientCServer`	Create a `CServer` with a `mkCGetPut` and a `mkGetCPut`. Provides a `CServer` interface and a regular `Client` interface.
	`module mkClientCServer(`
	`Tuple2#(Client#(req_type, resp_type),`
	`CServer#(n, req_type, resp_type)))`
	`provisos (Bits#(req_type),`
	`Bits#(resp_type));`

`mkCClientServer`	Create a `CClient` with a `mkCGetPut` and a `mkGetCPut`. Provides a `CClient` interface and a regular `Server` interface.
	`module mkCClientServer(`
	`Tuple2#(CClient#(n, req_type, resp_type),`
	`Server#(req_type, resp_type)))`
	`provisos (Bits#(req_type),`
	`Bits#(resp_type));`

3.7.6 CommitIfc

Package

```
import CommitIfc :: * ;
```

Description

The `CommitIfc` package defines a Commit/Accept protocol and interfaces to implement a combinational connection between two modules without adding an AND gate in the connection. The protocols implemented by FIFO and Get/Put connections add an AND gate between the modules being connected. This combinational loop in the connection of the interfaces can cause complications in FPGA applications and in partitioning for FPGAs. Additionally, some synthesis tools require a connection level without any gates. By using the `CommitIfc` protocol the AND gate is moved out of the connection and into the connecting modules.

The `CommitIfc` package defines two interfaces, `SendCommit` and `RecvCommit`, which model the opposit ends of a FIFO. The protocol does not apply an execution order between the `dataout` and `ack` or `datain` and `accept` methods. That is, one can signal `accept` before the data arrives.

Interfaces

The `CommitIfc` package defines two interfaces: `SendCommit` and `RecvCommit`.

The `SendCommit` interface declares two methods: a value method `dataout` and an Action method `ack`. No execution order is applied between the `dataout` and the `ack`; one can signal that the data is accepted before the data rrives.

SendCommit Interface		
Name	Type	Description
dataout	a_type	The data being sent. There is an implicit RDY indicating the data is valid.
ack	Action	Signal that data has been accepted.

```
interface SendCommit#(type a_type);
   method a_type dataout;
   (*always_ready*)
   method Action ack;
endinterface
```

The `RecvCommit` interface declares two methods: an Action method with the data, `datain`, and a value method `accept`, returning a Bool indicating if the interface can accept data (comparable to a `notFull`).

RecvCommit Interface		
Name	Type	Description
datain	Action	Receives, or enqueues, the value `din`, of type `a_type`.
accept	Bool	A boolean indicating if the interface can accept data, comparable to a `notFull`.

```
interface RecvCommit#(type a_type);
   (*always_ready*)
   method Action datain (a_type din);
   (*always_ready*)
   method Bool accept ;
endinterface
```

Connectble Instances

The `CommitIfc` package defines instances of the `Connectable` type class for the `SendCommit` and `RecvCommit` interfaces, defining how the types can be connected. The `Connectable` type class defines a `mkConnection` module for each set of pairs.

```
instance Connectable#(SendCommit#(a_type), RecvCommit#(a_type));

instance Connectable#(RecvCommit#(a_type), SendCommit#(a_type));
```

FIFOF The `SendCommit` and `RecvCommit` interfaces can be connected to `FIFOF` interfaces.

```
instance Connectable#(SendCommit#(a_type), FIFOF#(a_type))
   provisos (Bits#(a_type, size_a));

instance Connectable#(FIFOF#(a_type), SendCommit#(a_type))
   provisos (Bits#(a_type, size_a));

instance Connectable#(RecvCommit#(a_type), FIFOF#(a_type))
   provisos (Bits#(a_type, size_a));

instance Connectable#(FIFOF#(a_type), RecvCommit#(a_type))
   provisos (Bits#(a_type, size_a));
```

SyncFIFOIfc The SendCommit and RecvCommit interfaces can be connected to SyncFIFOIfc interfaces.

```
instance Connectable#(SendCommit#(a_type), SyncFIFOIfc#(a_type))
   provisos (Bits#(a_type, size_a));

instance Connectable#(SyncFIFOIfc#(a_type), SendCommit#(a_type))
   provisos (Bits#(a_type, size_a));

instance Connectable#(RecvCommit#(a_type), SyncFIFOIfc#(a_type))
   provisos (Bits#(a_type, size_a));

instance Connectable#(SyncFIFOIfc#(a_type), RecvCommit#(a_type))
   provisos (Bits#(a_type, size_a));
```

Typeclasses

The CommitIfc package defines typeclasses for converting to these interfaces from other interface types. These must use a module since rules and wires are required.

```
typeclass ToSendCommit#(type a_type , type b_type)
   dependencies (a_type determines b_type);
   module mkSendCommit#(a_type x) (SendCommit#(b_type));
endtypeclass

typeclass ToRecvCommit#(type  a_type , type b_type)
   dependencies  (a_type determines b_type);
   module mkRecvCommit#(a_type x) (RecvCommit#(b_type));
endtypeclass
```

Instances

Instances for the ToSendCommit and ToRecvCommit type classes are defined to convert to convert from FIFO, FIFOF, SyncFIFOIfc, Get and Put interfaces.

FIFO

```
instance ToSendCommit#(FIFO#(a), a);
```

Note: ToRecvCommit#(FIFO#(a_type), a_type) is not possible, because it would need to have a notFull signal.

FIFOF The FIFOF instances assume that the fifo has proper implicit conditions.

```
instance ToSendCommit#(FIFOF#(a_type), a_type);
   module mkSendCommit #(FIFOF#(a) f) (SendCommit#(a));

instance ToRecvCommit#(FIFOF#(a_type), a_type)
   provisos(Bits#(a,sa));
   module mkRecvCommit #(FIFOF#(a) f) (RecvCommit#(a));
```

SyncFIFOIfc The `SyncFIFOIfc` instances assume that the fifo has proper implicit conditions.

```
instance ToSendCommit#(SyncFIFOIfc#(a_type), a_type);
   module mkSendCommit #(SyncFIFOIfc#(a) f) (SendCommit#(a));

instance ToRecvCommit#(SyncFIFOIfc#(a_type), a_type)
   provisos(Bits#(a,sa));
   module mkRecvCommit #(SyncFIFOIfc#(a) f) (RecvCommit#(a));
```

Get and Put These convert from `Get` and `Put` interfaces but introduce additional latency:

```
instance ToSendCommit#(Get#(a_type), a_type)
   provisos ( Bits#(a,sa));
   module mkSendCommit #(Get#(a_type) g) (SendCommit#(a_type));

instance ToRecvCommit#(Put#(a_type), a_type)
   provisos(Bits#(a,sa));
   module mkRecvCommit #(Put#(a_type) p) (RecvCommit#(a_type));
```

These add FIFOs, but maintain loopless behavior:

```
instance Connectable#(SendCommit#(a_type), Put#(a_type))
   provisos (ToRecvCommit#(Put#(a_type), a_type));

instance Connectable#(Put#(a_type), SendCommit#(a_type))
   provisos (ToRecvCommit#(Put#(a_type), a_type));

instance Connectable#(RecvCommit#(a_type), Get#(a_type))
   provisos (ToSendCommit#(Get#(a_type), a_type));

instance Connectable#(Get#(a_type), RecvCommit#(a_type))
   provisos (ToSendCommit#(Get#(a_type), a_type));
```

Client/Server Variations

The `SendCommit` and `RecvCommit` interfaces can be combined into `ClientCommit` and `ServerCommit` type interfaces, similar to the `Client` and `Server` interfaces described in Section 3.7.3.

A `Client` provides two subinterfaces, a `Get` and a `Put` The `ClientCommit` interface combines a `SendCommit` request with a `RecvCommit` response.

```
interface ClientCommit#(type req, type resp);
   interface SendCommit#(req) request;
   interface RecvCommit#(resp) response;
endinterface
```

The `mkClientfromClientCommit` module takes a `ClientCommit` interface and provides a `Client` interface:

mkClientFromClientCommit	Provides a `Client` interface from a `ClientCommit` interface.

	`module mkClientFromClientCommit#(ClientCommit#(req, resp) c)`
	`(Client#(req,resp))`
	`provisos ( Bits#(resp,_x), Bits#(req,_y));`

A `Server` interface provides a `Put` request with a `Get` response. The `ServerCommit` interface combines a `RecvCommit` request with a `SendCommit` response.

```
interface ServerCommit#(type req, type resp);
   interface RecvCommit#(req) request;
   interface SendCommit#(resp) response;
endinterface
```

`ClientCommit` and `ServerCommit` interfaces are connectable to each other.

```
instance Connectable#(ClientCommit#(req,resp), ServerCommit#(req,resp));

instance Connectable#( ServerCommit#(req,resp), ClientCommit#(req,resp));
```

`ClientCommit` and `ServerCommit` interfaces are connectable to `Clients` and `Servers`.

```
instance Connectable #(ClientCommit#(req,resp), Server#(req,resp))
   provisos ( Bits#(resp,_x), Bits#(req,_y));

instance Connectable #(Server#(req,resp), ClientCommit#(req,resp))
   provisos ( Bits#(resp,_x), Bits#(req,_y));

instance Connectable #(ServerCommit#(req,resp), Client#(req,resp))
   provisos ( Bits#(resp,_x), Bits#(req,_y));

instance Connectable #( Client#(req,resp), ServerCommit#(req,resp))
   provisos ( Bits#(resp,_x), Bits#(req,_y));
```

The `SendCommit` and `RecvCommit` can be defined as instances of `ToGet` and `ToPut`. These functions introduce a combinational loop between the Commit interface methods.

```
instance ToGet#(SendCommit#(a_type, a_type);

instance ToPut#(RecvCommit#(a_type, a_type);
```

3.8 Utilities

3.8.1 LFSR

Package

```
import LFSR :: * ;
```

Description

The `LFSR` package implements Linear Feedback Shift Registers (LFSRs). LFSRs can be used to obtain reasonable pseudo-random numbers for many purposes (though not good enough for cryptography). The `seed` method must be called first, to prime the algorithm. Then values may be read using the `value` method, and the algorithm stepped on to the next value by the `next` method. When a LFSR is created the start value, or seed, is 1.

Interfaces and Methods

The `LFSR` package provides an interface, `LFSR`, which contains three methods; `seed`, `value`, and `next`. To prime the LFSR the `seed` method is called with the parameter `seed_value`, of datatype `a_type`. The value is read with the `value` method. The `next` method is used to shift the register on to the next value.

LFSR Interface				
Method			Arguments	
Name	Type	Description	Name	Description
seed	Action	Sets the value of the shift register.	a_type seed_value	datatype of the seed value the initial value
value	a_type	returns the value of the shift register		
next	Action	signals the shift register to shift to the next value.		

```
interface LFSR #(type a_type);
    method Action seed(a_type seed_value);
    method a_type value();
    method Action next();
endinterface: LFSR
```

Modules

The module mkFeedLFSR creates a LFSR where the polynomial is specified by the mask used for feedback.

mkFeedLFSR	Creates a LFSR where the polynomial is specified by the mask (*feed*) used for feedback.
	module mkFeedLFSR#(Bit#(n) feed)(LFSR#(Bit#(n)));

For example, the polynominal $x^7+x^3+x^2+x+1$ is defined by the expression mkFeedLFSR#(8'b1000_1111)

Using the module mkFeedLFSR, the following maximal length LFSR's are defined in this package.

Module Name	feed	Module Definition
mkLFSR_4	4'h9 $x^3 + 1$	module mkLFSR_4 (LFSR#(Bit#(4)));
mkLFSR_8	8'h8E	module mkLFSR_8 (LFSR#(Bit#(8)));
mkLFSR_16	16'h8016	module mkLFSR_16 (LFSR#(Bit#(16)));
mkLFSR_32	32'h80000057	module mkLFSR_32 (LFSR#(Bit#(32)));

For example,

```
mkLFSR_4   =  mkFeedLFSR( 4'h9 );
```

The module mkLFSR_4 instantiates the interface LFSR with the value Bit#(4) to produce a 4 bit shift register. The module uses the polynomial defined by the mask 4'h9 ($x^3 + 1$) and the module mkFeedLFSR.

The mkRCounter function creates a counter with a LFSR interface. This is useful during debugging when a non-random sequence is desired. This function can be used in place of the other mkLFSR module constructors, without changing any method calls or behavior.

`mkRCounter`	Creates a counter with a LFSR interface.
	`module mkRCounter#( Bit#(n) seed ) ( LFSR#(Bit#(n)) );`

Example - Random Number Generator

```
import GetPut::*;
import FIFO::*;
import LFSR::*;

//  We want 6-bit random numbers, so we will use the 16-bit version of
//  LFSR and take the most significant six bits.

//  The interface for the random number generator is parameterized on bit
//  length.  It is a "get" interface, defined in the GetPut package.

typedef Get#(Bit#(n)) RandI#(type n);

module mkRn_6(RandI#(6));
  // First we instantiate the LFSR module
  LFSR#(Bit#(16)) lfsr <- mkLFSR_16 ;

  // Next comes a FIFO for storing the results until needed
  FIFO#(Bit#(6)) fi <-  mkFIFO ;

  // A boolean flag for ensuring that we first seed the LFSR module
  Reg#(Bool) starting <- mkReg(True) ;

  // This rule fires first, and sends a suitable seed to the module.
  rule start (starting);
      starting <= False;
      lfsr.seed('h11);
  endrule: start

  // After that, the following rule runs as often as it can, retrieving
  // results from the LFSR module and enqueing them on the FIFO.
  rule run (!starting);
      fi.enq(lfsr.value[10:5]);
      lfsr.next;
  endrule: run

  // The interface for mkRn_6 is a Get interface.  We can produce this from a
  // FIFO using the fifoToGet function.  We therefore don't need to define any
  // new methods explicitly in this module: we can simply return the produced
  // Get interface as the "result" of this module instantiation.
  return fifoToGet(fi);
endmodule
```

3.8.2 Randomizable

Package

```
import Randomizable :: * ;
```

Description

The Randomizable package includes interfaces and modules to generate random values of a given data type.

Typeclasses

The `Randomizable` package includes the `Randomizable` typeclass.

```
typeclass Randomizable#(type t);
   module mkRandomizer (Randomize#(t));
endtypeclass
```

Interfaces and Methods

Randomize Interface		
Name	Type	Description
`cntrl`	`Interface`	Control interface provided by the module.
`next`	`ActionValue`	Returns the next value of type `a`.

```
interface Randomize#(type a);
   interface Control cntrl;
   method ActionValue#(a) next();
endinterface
```

Control Interface		
Name	Type	Description
`init`	`Control`	Action method to initialize the randomizer.

```
interface Control ;
   method Action init();
endinterface
```

Modules

The `Randomizable` package includes two modules which return random values of type `a`. The difference between the two modules is how the min and max values are determined. The module `mkGenericRandomizer` uses the min and max values of the type, while the module `mkConstrainedRandomizer` uses arguments to set the min and max values. The type `a` must be in the `Bounded` class for both modules.

`mkGenericRandomizer`	This module provides a `Randomize` interface, which will return the next random value when the `next` method is invoked. The `min` and `max` values are the values defined by the type `a` which must be in the `Bounded` class.
	`module mkGenericRandomizer (Randomize#(a))` `   provisos (Bits#(a, sa), Bounded#(a));`

`mkConstrainedRandomizer`	This module provides a `Randomize` interface, which will give the next random value when the `next` method is invoked. When instantiated, the `min` and `max` values are provided as arguments. Type `a` must be in the `Bounded` class.
	`module mkConstrainedRandomizer#(a min, a max) (Randomize#(a))` `   provisos (Bits#(a, sa), Bounded#(a));`

Example

The `mkTLMRandomizer` module, shown below, uses the Randomize package to generate random values for TLM packets. The `mkConstrainedRandomizer` module is for fields with specific allowed values or ranges, while the `mkGenericRandomizer` module is for field where all values of the type are allowed.

```
module mkTLMRandomizer#(Maybe#(TLMCommand) m_command) (Randomize#(TLMRequest#(`TLM_TYPES)))
   provisos(Bits#(RequestDescriptor#(`TLM_TYPES), s0),
    Bounded#(RequestDescriptor#(`TLM_TYPES)),
    Bits#(RequestData#(`TLM_TYPES), s1),
    Bounded#(RequestData#(`TLM_TYPES))
    );

   ...
   // Use mkGeneric Randomizer - entire range valid
   Randomize#(RequestDescriptor#(`TLM_TYPES)) descriptor_gen <- mkGenericRandomizer;
   Randomize#(Bit#(2))                        log_wrap_gen   <- mkGenericRandomizer;
   Randomize#(RequestData#(`TLM_TYPES))       data_gen       <- mkGenericRandomizer;

   // Use mkConstrainedRandomizer to Avoid UNKNOWN
   Randomize#(TLMCommand) command_gen <- mkConstrainedRandomizer(READ, WRITE);
   Randomize#(TLMBurstMode) burst_mode_gen <- mkConstrainedRandomizer(INCREMENT, WRAP);

   // Use mkConstrainedRandomizer to set legal sizes between 1 and 16
   Randomize#(TLMUInt#(`TLM_TYPES)) burst_length_gen <- mkConstrainedRandomizer(1,16);
   ...
```

3.8.3 Arbiter

Package

```
import Arbiter :: * ;
```

Description

The Arbiter package includes interfaces and modules to implement two different arbiters: a fair arbiter with changing priorities (round robin) and a sticky arbiter, also round robin, but which gives the current owner priority.

Interfaces and Methods

The Arbiter package includes three interfaces: a arbiter client interface, an arbiter request interface and an arbiter interface which is a vector of client interfaces.

ArbiterClient_IFC The `ArbiterClient_IFC` interface has two methods: an Action method to make the request and a Boolean value method to indicate the request was granted. The lock method is unused in this implementation.

```
interface ArbiterClient_IFC;
   method Action request();
   method Action lock();
   method Bool grant();
endinterface
```

ArbiterRequest_IFC The `ArbiterRequest_IFC` interface has two methods: an Action method to grant the request and a Boolean value method to indicate there is a request. The lock method is unused in this implementation.

```
interface ArbiterRequest_IFC;
   method Bool request();
   method Bool lock();
   method Action grant();
endinterface
```

The `ArbiterClient_IFC` interface and the `ArbiterRequest_IFC` interface are connectable.

```
instance Connectable#(ArbiterClient_IFC,  ArbiterRequest_IFC);
```

Arbiter_IFC The `Arbiter_IFC` has a subinterface which is a vector of `ArbiterClient_IFC` interfaces. The number of items in the vector equals the number of clients.

```
interface Arbiter_IFC#(type count);
   interface Vector#(count, ArbiterClient_IFC) clients;
endinterface
```

Modules

The `mkArbiter` module is a fair arbiter with changing priorities (round robin). The `mkStickyArbiter` gives the current owner priority - they can hold priority as long as they keep requesting it. The modules all provide a `Arbiter_IFC` interface.

`mkArbiter`	This module is a fair arbiter with changing priorities (round robin). If `fixed` is `True`, the current client holds the priority, if `fixed` is `False`, it moves to the next client. `mkArbiter` provides a `Arbiter_IFC` interface. Initial priority is given to client 0.
	`module mkArbiter#(Bool fixed) (Arbiter_IFC#(count));`

`mkStickyArbiter`	As long as the client currently with the grant continues to assert `request`, it can hold the grant. It provides a `Arbiter_IFC` interface.
	`module mkStickyArbiter (Arbiter_IFC#(count));`

3.8.4 Cntrs

Package

```
import Cntrs :: * ;
```

Description

The Cntrs package provides interfaces and modules to implement typed and untyped up/down counters.

The `Count` interface and associated `mkCount` module provides an up/down counter which allows atomic simultaneous increment and decrement operations. The scheduled order of operations in a single cycle is:

```
 read SB update SB (incr,decr) SB write
```

If there are simultaneous `update`, `incr`, and `decr` operations, the final result will be:

```
update_val + incr_val - decr_val
```

A `write` sets the new value to `write_val` regardless of the other methods called in the cycle.

The `UCount` interface and associated `mkUCount` module provide an untyped version of an up/down counter; that is the counter width can be determined at elaboration time rather than type check time. The value of the counter is represented by an `UInt#(n)` where the width of the counter is determined by the `maxValue` ($0 \leq maxValue < 2^{32}$) argument. There are no methods to access the counter value directly; you can only access the value through the comparison operations.

Interfaces and Methods

The Cntrs package provides two interfaces; `Count` which is a typed interface and `UCount` which is an untyped interface.

Count Interface Methods		
Name	Type	Description
`incr`	Action	Increments the counter by `incr_val`
`decr`	Action	Decrements the counter by `decr_val`
`update`	Action	Sets the value to `update_val`. Final value will include increment and decrement values.
`_write`	Action	Sets the final value to `write_val` regardless of other operations in the cycle.
`_read`	`t`	Returns the value of the counter.

```
interface Count#(type t);
   method Action incr    (t incr_val);
   method Action decr    (t decr_val);
   method Action update  (t update_val);
   method Action _write  (t write_val);
   method t      _read;
endinterface
```

UCount Interface Methods		
Name	Type	Description
`incr`	Action	Increments the counter by `incr_val`
`decr`	Action	Decrements the counter by `decr_val`
`update`	Action	Sets the value to `update_val`. Final value will include increment and decrement values.
`_write`	Action	Sets the final value to `write_val` regardless of other operations in the cycle.
`isEqual`	`Bool`	Returns true if `val` is equal to the value of the counter.
`isLessThan`	`Bool`	Returns true if `val` is less than the value of the counter.
`isGreaterThan`	`Bool`	Returns true if `val` is greater than the value of the counter.

```
interface UCount;
   method Action update (Integer update_val);
   method Action _write (Integer write_val);
   method Action incr   (Integer incr_val);
   method Action decr   (Integer decr_val);
   method Bool isEqual  (Integer val);
   method Bool isLessThan    (Integer val);
   method Bool isGreaterThan (Integer val);
endinterface
```

Modules

`mkCount`	Instantiates a counter where read precedes update precedes an increment or decrement precedes a write. The `ModArith` provisos limits its use to module 2 arithmetic types: `UInt`, `Int`, and `Bit`. Widths of size 0 are supported.
	`module mkCount #(t resetVal) (Count#(t)) provisos ( Arith#(t) ,ModArith#(t) ,Bits#(t,st));`

```
module mkCount #(t resetVal) (Count#(t))
   provisos ( Arith#(t)
             ,ModArith#(t)
             ,Bits#(t,st));
```

`mkUCount`	Instantiates a counter which can count from 0 to `maxVal` inclusive. `maxVal` must be known at compile time. The `initValue` and `maxValue` must be Integers.
	`module mkUCount#(Integer initValue, Integer maxValue) (UCount);`

Verilog Modules

`mkCount` and `mkUCount` corresponds to the following Verilog module, which are found in the BSC Verilog library, `$BLUESPECDIR/Verilog/`.

BSV Module Name	Verilog Module Name	Defined in File
`mkCount` `mkUCount`	vCount	Counter.v

3.8.5 GrayCounter

Package

```
import GrayCounter :: * ;
```

Description

The GrayCounter package provides an interface and a module to implement a gray-coded counter with methods for both binary and Gray code. This package is designed for use in the `BRAMFIFO` module, Section 3.2.4. Since BRAMs have registered address inputs, the binary outputs are not registered. The counter has two domains, source and destination. Binary and Gray code values are written in the source domain. Both types of values can be read from the source and the destination domains.

Types

The GrayCounter package uses the type `Gray`, defined in the Gray package, Section 3.8.6. The Gray package is imported by the GrayCounter package.

Interfaces and Methods

The GrayCounter package includes one interface, `GrayCounter`.

GrayCounter Interface Methods		
Name	Type	Description
`incr`	Action	Increments the counter by 1
`decr`	Action	Decrements the counter by 1
`sWriteBin`	Action	Writes a binary value into the counter in the source domain.
`sReadBin`	`Bit#(n)`	Returns a binary value from the source domain of the counter. The output is not registered
`sWriteGray`	Action	Writes a Gray code value into the counter in the source domain.
`sReadGray`	`Gray#(n)`	Returns the Gray code value from the source domain of the counter. The output is registered.
`dReadBin`	`Bit#(n)`	Returns the binary value from the destination domain of the counter. The output is not registered.
`dReadGray`	`Gray#(n)`	Returns the Gray code value from the destination domain of the counter. The output is registered.

```
interface GrayCounter#(numeric type n);
   method    Action      incr;
   method    Action      decr;
   method    Action      sWriteBin(Bit#(n) value);
   method    Bit#(n)     sReadBin;
   method    Action      sWriteGray(Gray#(n) value);
   method    Gray#(n)    sReadGray;
   method    Bit#(n)    dReadBin;
   method    Gray#(n)   dReadGray;
endinterface: GrayCounter
```

Modules

The module `mkGrayCounter` instantiates a Gray code counter with methods for both binary and Gray code.

`mkGrayCounter`	Instantiates a Gray counter with an initial value `initval`.
	`module mkGrayCounter#(Gray#(n) initval, Clock dClk, Reset dRstN) (GrayCounter#(n)) provisos(Add#(1, msb, n));`

3.8.6 Gray

Package

```
import Gray :: * ;
```

Description

The `Gray` package defines a datatype, `Gray` and functions for working with the Gray type. This type is used by the `GrayCounter` package.

Types and type classes

The datatype `Gray` is a representation for Gray code values. The basic representation is the `Gray` structure, which is polymorphic on the size of the value.

```
typedef struct {
                Bit#(n) code;
               } Gray#(numeric type n) deriving (Bits, Eq);
```

The `Gray` type belongs to the `Literal` and `Bounded` type classes. Each type class definition includes functions which are then also defined for the data type. The Prelude library definitions (Section 2) describes which functions are defined for each type class.

Type Classes used by `Gray`									
	`Bits`	`Eq`	`Literal`	`Arith`	`Ord`	`Bounded`	`Bit` wise	`Bit` Reduction	`Bit` Extend
`Gray`	√	√	√			√			

Literal The `Gray` type is a member of the `Literal` class, which defines an encoding from the compile-time `Integer` type to `Gray` type with the `fromInteger` and `grayEncode` functions. The `fromInteger` converts the value to a bit pattern, and then calls `grayEncode`.

```
instance Literal #( Gray#(n) )
   provisos(Add#(1, msb, n));
```

Bounded The `Gray` type is a member of the `Bounded` class, which provides the functions `minBound` and `maxBound` to define the minimum and maximum Gray code values.

- minimum: `'b0`
- maximum: `'b10...0`

```
instance Bounded # ( Gray#(n) )
   provisos(Add#(1, msb, n));
```

Functions

`grayEncode`	This function takes a binary value of type `Bit#(n)` and returns a `Gray` type with the Gray code value.
	`function Gray#(n) grayEncode(Bit#(n) value)`
	`        provisos(Add#(1, msb, n));`

`grayDecode`	This function takes a Gray code value of size `n` and returns the binary value.
	`function Bit#(n) grayDecode(Gray#(n) value)`
	`        provisos(Add#(1, msb, n));`

`grayIncrDecr`	This functions takes a Gray code value and a Boolean, `decrement`. If `decrement` is True, the value returned is one less than the input value. If `decrement` is False, the value returned is one greater.
	`function Gray#(n) grayIncrDecr(Bool decrement,`
	`                              Gray#(n) value)`
	`        provisos(Add#(1, msb, n));`

`grayIncr`	Takes a Gray code value and returns a Gray code value incremented by 1.
	`function Gray#(n) grayIncr(Gray#(n) value)`
	`        provisos(Add#(1, msb, n));`

`grayDecr`	Takes a Gray code value a returns a Gray code value decremented by 1.
	`function Gray#(n) grayDecr(Gray#(n) value)`
	`        provisos(Add#(1, msb, n));`

3.8.7 CompletionBuffer

Package

```
import CompletionBuffer :: * ;
```

Description

A `CompletionBuffer` is like a FIFO except that the order of the elements in the buffer is independent of the order in which the elements are entered. Each element obtains a token, which reserves a slot in the buffer. Once the element is ready to be entered into the buffer, the token is used to place the element in the correct position. When removing elements from the buffer, the elements are delivered in the order specified by the tokens, not in the order that the elements were written.

Completion Buffers are useful when multiple tasks are running, which may complete at different times, in any order. By using a completion buffer, the order in which the elements are placed in the buffer can be controlled, independent of the order in which the data becomes available.

Interface and Methods

The CompletionBuffer interface provides three subinterfaces. The `reserve` interface, a `Get`, allows the caller to reserve a slot in the buffer by returning a token holding the identity of the slot. When data is ready to be placed in the buffer, it is added to the buffer using the `complete` interface of type `Put`. This interface takes a pair of values as its argument - the token identifying its slot, and the data itself. Finally, using the `drain` interface, of type `Get`, data may be retrieved from the buffer in the order in which the tokens were originally allocated. Thus the results of quick tasks might have to wait in the buffer while a lengthy task ahead of them completes.

The type of the elements to be stored is `element_type`. The type of the required size of the buffer is a numeric type `n`, which is also the type argument for the type for the tokens issued, `CBToken`. This allows the type-checking phase of the synthesis to ensure that the tokens are the appropriate size for the buffer, and that all the buffer's internal registers are of the correct sizes as well.

`CompletionBuffer` Interface		
Name	Type	Description
`reserve`	`Get`	Used to reserve a slot in the buffer. Returns a token, `CBToken #(n)`, identifying the slot in the buffer.
`complete`	`Put`	Enters the element into the buffer. Takes as arguments the slot in the buffer, `CBToken#(n)`, and the element to be stored in the buffer.
`drain`	`Get`	Removes an element from the buffer. The elements are returned in the order the tokens were allocated.

```
interface CompletionBuffer #(numeric type n, type element_type);
    interface Get#(CBToken#(n))                          reserve;
    interface Put#(Tuple2 #(CBToken#(n), element_type)) complete;
    interface Get#(element_type)                         drain;
endinterface: CompletionBuffer
```

Datatypes

The `CBToken` type is abstract to avoid misuse.

```
typedef union tagged { ... } CBToken #(numeric type n) ...;
```

Modules

The `mkCompletionBuffer` module is used to instantiate a completion buffer. It takes no size arguments, as all that information is already contained in the type of the interface it produces.

`mkCompletionBuffer`	Creates a completion buffer.
	`module mkCompletionBuffer(CompletionBuffer#(n, element_type))` `  provisos (Bits#(element_type, sizea))`

Example- Using a Completion Buffer in a server farm of multipliers

A server farm is a set of identical servers, which can each perform the same task, together with a controller. The controller allocates incoming tasks to any server which happens to be available (free), and sends results back to its caller.

The time needed to complete each task depends on the value of the multiplier argument; there is therefore no guarantee that results will become available in the order the tasks were started. It is required, however, that the controller return results to its caller in the order the tasks were received. The controller accordingly must instantiate a special mechanism for this purpose. The appropriate mechanism is a Completion Buffer.

```
import List::*;
import FIFO::*;
import GetPut::*;
import CompletionBuffer::*;

typedef Bit#(16) Tin;
typedef Bit#(32) Tout;

// Multiplier interface
interface Mult_IFC;
    method Action  start (Tin m1, Tin m2);
    method ActionValue#(Tout)    result();
endinterface

typedef Tuple2#(Tin,Tin) Args;
typedef 8 BuffSize;
typedef CBToken#(BuffSize) Token;

// This is a farm of multipliers, mkM. The module
// definition for the multipliers mkM is not provided here.
// The interface definition, Mult_IFC, is provided.
module mkFarm#( module#(Mult_IFC) mkM ) ( Mult_IFC );

   // make the buffer twice the size of the farm
   Integer n = div(valueof(BuffSize),2);

   // Declare the array of servers and instantiate them:
   Mult_IFC mults[n];
   for (Integer i=0; i<n; i=i+1)
      begin
         Mult_IFC s <- mkM;
         mults[i] = s;
      end

   FIFO#(Args) infifo <- mkFIFO;

   // instantiate the Completion Buffer, cbuff, storing values of type Tout
   // buffer size is Buffsize, data type of values is Tout
   CompletionBuffer#(BuffSize,Tout) cbuff <- mkCompletionBuffer;

   // an array of flags telling which servers are available:
   Reg#(Bool) free[n];
   // an array of tokens for the jobs in progress on the servers:
```

```
   Reg#(Token) tokens[n];
   // this loop instantiates n free registers and n token registers
   // as well as the rules to move data into and out of the server farm
   for (Integer i=0; i<n; i=i+1)
      begin
         // Instantiate the elements of the two arrays:
         Reg#(Bool) f <- mkReg(True);
         free[i] = f;
         Reg#(Token) t <- mkRegU;
         tokens[i] = t;

         Mult_IFC s = mults[i];

         // The rules for sending tasks to this particular server, and for
         // dealing with returned results:
         rule start_server (f); // start only if flag says it's free
            // Get a token
            CBToken#(BuffSize) new_t <- cbuff.reserve.get;

            Args a = infifo.first;
            Tin a1 = tpl_1(a);
            Tin a2 = tpl_2(a);
            infifo.deq;

            f <= False;
            t <= new_t;
            s.start(a1,a2);
         endrule

         rule end_server (!f);
            Tout x <- s.result;
            // Put the result x into the buffer, at the slot t
            cbuff.complete.put(tuple2(t,x));
            f <= True;
         endrule
      end

   method Action start (m1, m2);
      infifo.enq(tuple2(m1,m2));
   endmethod

   // Remove the element from the buffer, returning the result
   // The elements will be returned in the order that the tokens were obtained.
   method result = cbuff.drain.get;
endmodule
```

3.8.8 UniqueWrappers

Package

```
import UniqueWrappers :: * ;
```

Description

The `UniqueWrappers` package takes a piece of combinational logic which is to be shared and puts it into its own protective shell or *wrapper* to prevent its duplication. This is used in instances where a separately synthesized module is not possible. It allows the designer to use a piece of logic at several places in a design without duplicating it at each site.

There are times where it is desired to use a piece of logic at several places in a design, but it is too bulky or otherwise expensive to duplicate at each site. Often the right thing to do is to make the piece of logic into a separately synthesized module – then, if this module is instantiated only once, it will not be duplicated, and the tool will automatically generate the scheduling and multiplexing logic to share it among the sites which use its methods. Sometimes, however, this is not convenient. One reason might be that the logic is to be incorporated into a submodule of the design which is itself polymorphic – this will probably cause difficulties in observing the constraints necessary for a module which is to be separately synthesized. And if a module is *not* separately synthesized, the tool will inline its logic freely wherever it is used, and thus duplication will not be prevented as desired.

This package covers the case where the logic to be shared is combinational and cannot be put into a separately synthesized module. It may be thought of as surrounding this combinational function with a protective shell, a *unique wrapper*, which will prevent its duplication. The module `mkUniqueWrapper` takes a one-argument function as a parameter; both the argument type `a` and the result type `b` must be representable as bits, that is, they must both be in the `Bits` typeclass.

Interfaces

The `UniqueWrappers` package provides an interface, `Wrapper`, with one actionvalue method, `func`, which takes an argument of type `a` and produces a method of type `ActionValue#(b)`. If the module is instantiated only once, the logic implementing its parameter will be instantiated just once; the module's method may, however, be used freely at several places.

Although the function supplied as the parameter is purely combinational and does not change state, the method is of type `ActionValue`. This is because actionvalue methods have `enable` signals and these signals are needed to organize the scheduling and multiplexing between the calling sites.

Variants of the interface `Wrapper` are also provided for handling functions of two or three arguments; the interfaces have one and two extra parameters respectively. In each case the result type is the final parameter, following however many argument type parameters are required.

Wrapper Interfaces	
`Wrapper`	This interface has one actionvalue method, `func`, which takes an argument of type `a_type` and produces an actionvalue of type `ActionValue#(b_type)`.

```
interface Wrapper#(type a_type, type b_type);
  method ActionValue#(b_type) func (a_type x);
```

`Wrapper2`	Similar to the `Wrapper` interface, but it takes two arguments.

```
interface Wrapper2#(type a1_type, type a2_type, type b_type);
  method ActionValue#(b_type) func (a1_type x, a2_type y);
```

`Wrapper3`	Similar to the `Wrapper` interface, but it takes three arguments.

```
interface Wrapper3#(type a1_type, type a2_type, type a3_type,
                    type b_type);
  method ActionValue#(b_type) func (a1_type x, a2_type y, a3_type z);
```

Modules

The interfaces Wrapper, Wrapper2, and Wrapper3 are provided by the modules mkUniqueWrapper, mkUniqueWrapper2, and mkUniqueWrapper3. These modules vary only in the number of aguments in the parameter function.

If a function has more than three arguments, it can always be rewritten or wrapped as one which takes the arguments as a single tuple; thus the one-argument version mkUniqueWrapper can be used with this function.

mkUniqueWrapper

Takes a function, func, with a single parameter x and provides the interface Wrapper.

```
module mkUniqueWrapper#(function b_type func(a_type x))
                        (Wrapper#(a_type, b_type))
   provisos (Bits#(a_type, sizea), Bits#(b_type, sizeb));
```

mkUniqueWrapper2

Takes a function, func, with a two parameters, x and y, and provides the interface Wrapper2.

```
module mkUniqueWrapper2#(function b_type func(a1_type x, a2_type y))
                         (Wrapper2#(a1_type, a2_type, b_type))
   provisos (Bits#(a1_type, sizea1), Bits#(a2_type, sizea2),
             Bits#(b_type, sizeb));
```

mkUniqueWrapper3

Takes a function, func, with a three parameters, x, y, and z, and provides the interface Wrapper3.

```
module mkUniqueWrapper3#(function b_type
                           func(a1_type x, a2_type y, a3_type z))
                         (Wrapper3#(a1_type, a2_type, a3_type, b_type))
   provisos (Bits#(a1_type, sizea1), Bits#(a2_type, sizea2),
             Bits#(a3_type, sizea3), Bits#(b_type, sizeb));
```

Example: Complex Multiplication

```
// This module defines a single hardware multiplier, which is then
// used by multiple method calls to implement complex number
// multiplication (a + bi)(c + di)

typedef Int#(18) CFP;

module mkComplexMult1Fifo( ArithOpGP2#(CFP) ) ;
  FIFO#(ComplexP#(CFP))  infifo1 <- mkFIFO;
  FIFO#(ComplexP#(CFP))  infifo2 <- mkFIFO;
  let arg1 = infifo1.first ;
  let arg2 = infifo2.first ;
```

```
  FIFO#(ComplexP#(CFP))  outfifo <- mkFIFO;

  Reg#(CFP)  rr <- mkReg(0) ;
  Reg#(CFP)  ii <- mkReg(0) ;
  Reg#(CFP)  ri <- mkReg(0) ;
  Reg#(CFP)  ir <- mkReg(0) ;

  // Declare and instantiate an interface that takes 2 arguments, multiplies them
  // and returns the result.  It is a Wrapper2 because there are 2 arguments.
  Wrapper2#(CFP,CFP, CFP) smult <- mkUniqueWrapper2( \* ) ;

  // Define a sequence of actions
  // Since smult is a UnqiueWrapper the method called is smult.func
  Stmt multSeq =
  seq
     action
        let mr <- smult.func( arg1.rel,  arg2.rel ) ;
        rr <= mr ;
     endaction
     action
        let mr <- smult.func( arg1.img, arg2.img ) ;
        ii <= mr ;
     endaction
     action
        // Do the first add in this step
        let mr <- smult.func( arg1.img,  arg2.rel ) ;
        ir <= mr ;
        rr <= rr - ii ;
     endaction
     action
        let mr <- smult.func( arg1.rel, arg2.img );
        ri <= mr ;
        // We are done with the inputs so deq the in fifos
        infifo1.deq ;
        infifo2.deq ;
     endaction
     action
        let ii2 = ri + ir ;
        let res = Complex{ rel: rr , img: ii2 } ;
        outfifo.enq( res ) ;
     endaction
  endseq;

  // Now convert the sequence into a FSM ;
  //  Bluespec can assign the state variables, and pick up implict
  //  conditions of the actions
  FSM multfsm <- mkFSM(multSeq);
  rule startFSM;
     multfsm.start;
  endrule
endmodule
```

3.8.9 DefaultValue

Package

```
import DefaultValue :: * ;
```

Description

This package defines a type class of `DefaultValue` and instances of the type class for many commonly used datatypes. Users can create their own default value instances for other types. This type class is particularly useful for defining default values for user-defined structures.

Typeclasses

```
typeclass DefaultValue #( type t );
    t defaultValue ;
endtypeclass
```

The following instances are defined in the `DefaultValue` package. You can define your own instances for user-defined structures and other types.

`DefaultValue` Instances	
`Literal#(t)`	Any type `t` in the `Literal` class can have a default value which is defined here as 0. The types in the `Literal` class include `Bit#(n)`, `Int#(n)`, `UInt#(n)`, `Real`, `Integer`, `FixedPoint`, and `Complex`.

	`instance DefaultValue # (t)`
	`   provisos (Literal#(t));`
	`   defaultValue = fromInteger (0);`

`Bool`	The default value for a `Bool` is defined as `False`.
	`instance DefaultValue #( Bool );` `   defaultValue = False ;`

`void`	The default value for a void is defined as `?`.
	`instance DefaultValue #(void);` `   defaultValue = ?;`

`Maybe`	The default value for a `Maybe` is defined as `tagged Invalid`.
	`instance DefaultValue #( Maybe#(t) );` `   defaultValue = tagged Invalid ;`

The default value for a `Tuple` is composed of the default values of each member type. Instances are defined for `Tuple2` through `Tuple8`.

`Tuple2#(a,b)`	The default value of a `Tuple2` is the default value of element `a` and the default value of element `b`.	`instance DefaultValue #( Tuple2#(a,b) )`
		`   provisos (DefaultValue#(a)`
		`             ,DefaultValue#(b) );`
		`   defaultValue = tuple2 (defaultValue, defaultValue );`

`Vector`	The default value for a Vector replicates the element's default value type for each element.	`instance DefaultValue #( Vector#(n,t) )`
		`   provisos (DefaultValue#(t));`
		`   defaultValue = replicate (defaultValue) ;`

Examples

Example 1: Specifying the initial or reset values for a structure.

```
   Reg#(Int#(17))               rint  <- mkReg#(defaultValue); // initial value 0
   Reg#(Tuple2#(Bool,UInt#(5))) tbui  <- mkReg#(defaultValue); // value is(False,0)
   Reg#(Vector#(n,Bool)         vbool <- mkReg#(defaultValue); // initial value all False
```

Example 2: Using default values to replace the unsafe use of unpack.

```
import DefaultValue :: *;

typedef struct {
  UInt#(4) size;
  UInt#(3) depth ;
  } MyStruct
deriving (Bits, Eq);

instance DefaultValue #( MyStruct );
  defaultValue = MyStruct { size : 0,
                           depth : 1 };
endinstance
```

then you can use:

```
   Reg#(MyStruct)               mstr  <- mkReg(defaultValue);
```

instead of:

```
   Reg#(MyStruct)               mybad <- mkReg(unpack(0)); // Bad use of unpack
```

Example 3: Module instantiation which requires a large structure as an argument.

```
 ModParam modParams = defaultValue ;    // generate default value
 modParams.field1 = 5 ;                 // override some default values
 modParams.field2 = 1.4 ;
 ModIfc <- mkMod (modArgs) ;            // construct the module
```

3.8.10 TieOff

Package

```
import TieOff :: * ;
```

Description

This package provides a typeclass `TieOff#(t)` which may be userful to provide default enable methods of some interface `t`, some of which must be `always_enabled` or require some action.

Typeclasses

```
typeclass TieOff #(type t);
  module mkTieOff#(t ifc) (Empty);
endtypeclass
```

Example: Defining a `TieOff` for a `Get` interface

This is a sink module which pulls data from the `Get` interface and displays the data.

```
instance TieOff #(Get #(t) )
  provisos (Bits#(t,st),
            FShow#(t) );
  module mkTieOff ( Get#(t) ifc, Empty inf);
     rule getSink (True);
        t val <- ifc.get;
        $display( "Get tieoff %m", fshow(val) );
     endrule
  endmodule
endinstance
```

3.8.11 Assert

Package

```
import Assert :: *;
```

Description

The `Assert` package contains definitions to test assertions in the code. The `check-assert` flag must be set during compilation. By default the flag is set to `False` and assertions are ignored. The flag, when set, instructs the compiler to abort compilation if an assertion fails.

Functions

`staticAssert`	Compile time assertion. Can be used anywhere a compile-time statement is valid.
	`module staticAssert(Bool b, String s);`

`dynamicAssert`	Run time assertion. Can be used anywhere an Action is valid, and is tested whenever it is executed.	`function Action dynamicAssert(Bool b, String s);`

`continuousAssert`	Continuous run-time assertion (expected to be True on each clock). Can be used anywhere a module instantiation is valid.	`module continuousAssert#(Bool b, String s)(Empty);`

Examples using Assertions:

```
import Assert:: *;
module mkAssert_Example ();
  // A static assert is checked at compile-time
  // This code checks that the indices are within range
  for (Integer i=0; i<length(cs); i=i+1)
      begin
        Integer new_index = (cs[i]).index;
        staticAssert(new_index < valueOf(n),
            strConcat("Assertion index out of range: ", integerToString(new_index)));
      end

  rule always_fire (True);
       counter <= counter + 1;
  endrule
  // A continuous assert is checked on each clock cycle
  continuousAssert (!fail, "Failure: Fail becomes True");

  // A dynamic assert is checked each time the rule is executed
  rule test_assertion (True);
     dynamicAssert (!fail, "Failure: Fail becomes True");
  endrule
endmodule: mkAssert_Example
```

3.8.12 Probe

Package

```
import Probe :: * ;
```

Description

A `Probe` is a primitive used to ensure that a signal of interest is not optimized away by the compiler and that it is given a known name. In terms of BSV syntax, the `Probe` primitive it used just like a register except that only a write method exists. Since reads are not possible, the use of a `Probe` has no effect on scheduling. In the generated Verilog, the associated signal will be named just like the port of any Verilog module, in this case `<instance_name>$PROBE`. No actual `Probe` instance will be created however. The only side effects of a BSV `Probe` instantiation relate to the naming and retention of the associated signal in the generated Verilog.

Interfaces

```
interface Probe #(type a_type);
    method Action _write(a_type x1);
endinterface: Probe
```

Modules

The module `mkProbe` is used to instantiate a `Probe`.

`mkProbe`	Instantiates a `Probe`
	`module mkProbe(Probe#(a_type))`
	`  provisos (Bits#(a_type, sizea));`

Example - Creating and writing to registers and probes

```
import FIFO::*;
import ClientServer::*;
import GetPut::*;
import Probe::*;

typedef Bit#(32) LuRequest;
typedef Bit#(32) LuResponse;

module mkMesaHwLpm(ILpm);
   // Create registers for requestB32 and responseB32
   Reg#(LuRequest) requestB32 <- mkRegU();
   Reg#(LuResponse) responseB32 <- mkRegU();

   // Create a probe responseB32_probe
   Probe#(LuResponse) responseB32_probe <- mkProbe();
   ....
   // Define the interfaces:
   ....
      interface Get  response;
         method get() ;
            actionvalue
               let resp <- completionBuffer.drain.get();
               // record response for debugging purposes:
               let {r,t} = resp;
               responseB32 <= r;        // a write to a register
               responseB32_probe <= r;  // a write to a probe

               // count responses in status register
               return(resp);
            endactionvalue
         endmethod: get
      endinterface: response
    .....
endmodule
```

3.8.13 Reserved

Package

```
import Reserved :: * ;
```

Description

The Reserved package defines three abstract data types which only have the purpose of taking up space. They are useful when defining a struct where you need to enforce a certain layout and want to use the type checker to enforce that the value is not accidently used. One can enforce a layout unsafely with Bit#(n), but Reserved#(n) gives safety. A value of type Reserved#(n) takes up exactly n bits.

typedef ··· *abstract* ··· Reserved#(type n);

Types and Type classes

There are three types defined in the Reserved package: Reserved, ReservedZero, and ReservedOne. The Reserved type is an abstract data type which takes up exactly n bits and always returns an unspecified value. The ReservedZero and ReservedOne data types are equivalent to the Reserved type except that ReservedZero always returns '0 and ReservedOne always returns '1.

Type Classes used by Reserved									
	Bits	Eq	Literal	Arith	Ord	Bounded	Bit wise	Bit Reduction	Bit Extend
Reserved	√	√			√	√			
ReservedZero	√	√			√	√			
ReservedOne	√	√			√	√			

- Bits The only purpose of these types is to allow the value to exist in hardware (at port boundaries and in states). The user should have no reason to use pack/unpack directly.

  Converting Reserved to or from Bits returns a don't care (?).

  Converting ReservedZero to or from Bits returns a '0.

  Converting ReservedOne to or from Bits returns a '1.

- Eq and Ord

  Any two Reserved, ReservedZero, or ReservedOne values are considered to be equal.

- Bounded

  The upper and lower bound return don't care (?), '1 or '0 values depending on the type.

Example: Structure with a 8 bits reserved.

```
typedef struct {
   Bit#(8)               header;      // Frame.header
   Vector#(2, Bit#(8))   payload;     // Frame.payload
   Reserved#(8)          dummy;       // Can't access 8 bits reserved
   Bit#(8)               trailer;     // Frame.trailer
} Frame;
```

header	payload0	payload1	dummy	trailer
8	8	8	8	8

3.8.14 TriState

Package

```
import TriState :: * ;
```

Description

The `TriState` package implements a tri-state buffer, as shown in Figure 3. Depending on the value of the `output_enable`, `inout` can be an input or an output.

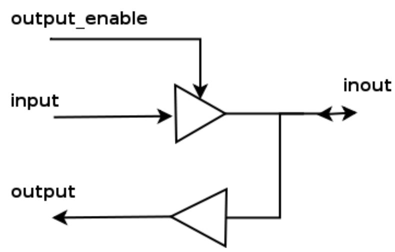

Figure 3: TriState Buffer

The buffer has two inputs, an `input` of type `value_type` and a Boolean `output_enable` which determines the direction of `inout`. If `output_enable` is True, the signal is coming in from `input` and out through `inout` and `output`. If `output_enable` is False, then a value can be driven in from `inout`, and the `output` value will be the value of `inout`. The behavior is described in the tables below.

output_enable = 0 output = inout		
Inputs		
input	inout	output
0	0	0
0	1	1
1	0	0
1	1	1

output_enable = 1 output = in inout = in		
	Outputs	
input	inout	output
0	0	0
1	1	1

This module is not supported in Bluesim.

Interfaces and Methods

The `TriState` interface is composed of an `Inout` interface and a `_read` method. The `_read` method is similar to the `_read` method of a register in that to read the method you reference the interface in an expression.

TriState Interface		
Name	Type	Description
io	Inout#(value_type)	Inout subinterface providing a value of type value_type
_read	value_type	Returns the value of output

```
(* always_ready, always_enabled *)
interface TriState#(type value_type);
   interface Inout#(value_type)   io;
   method    value_type           _read;
endinterface: TriState
```

Modules and Functions

The TriState package provides a module constructor function, mkTriState, which provides the TriState interface. The interface includes an Inout subinterface and the value of output.

mkTriState	Creates a module which provides the TriState interface.
	`module mkTriState#(Bool output_enable, value_type input) (TriState#(value_type)) provisos(Bits#(value_type, size_value));`

Verilog Modules

The TriState module is implemented by the Verilog module TriState.v which can be found in the BSC Verilog library, $BLUESPECDIR/Verilog/.

3.8.15 ZBus

Package

```
import ZBus :: * ;
```

Description

BSV provides the ZBus library to allow users to implement and use tri-state buses. Since BSV does not support high-impedance or undefined values internally, the library encapsulates the tri-state bus implementation in a module that can only be accessed through predefined interfaces which do not allow direct access to internal signals (which could potentially have high-impedance or undefined values).

The Verilog implementation of the tri-state module includes a number of primitive submodules that are implemented using Verilog tri-state wires. The BSV representation of the bus, however, only models the values of the bus at the associated interfaces and thus the need to represent high-impedance or undefined values in BSV is avoided.

A ZBus consists of a series of clients hanging off of a bus. The combination of the client and the bus is provided by the ZBusDualIFC interface which consists of 2 subinterfaces, the client and the bus. The client subinterface is provided by the ZBusClientIFC interface. The bus subinterface is provided by the ZBusBusIFC interface. The user never needs to manipulate the bus side, this is all done internally. The user builds the bus out of ZBusDualIFCs and then drives values onto the bus and reads values from the bus using the ZBusClientIFC.

Interfaces and Methods

There are three interfaces are defined in this package; ZBusDualIFC, ZBusClientIFC, and ZBusBusIFC.

The ZBusDualIFC interface provides two subinterfaces; a ZBusBusIFC and a ZBusClientIFC. For a given bus, one ZBusDualIFC interface is associated with each bus client.

ZBusDualIFC		
Name	Type	Description
busIFC	ZBusBusIFC#()	The subinterface providing the bus side of the ZBus.
clientIFC	ZBusClientIFC#(t)	The subinterface providing the client side to the ZBus.

```
interface ZBusDualIFC #(type value_type) ;
   interface ZBusBusIFC#(value_type)    busIFC;
   interface ZBusClientIFC#(value_type) clientIFC;
endinterface
```

The ZBusClientIFC allows a BSV module to connect to the tri-state bus. The drive method is used to drive a value onto the bus. The get() and fromBusValid() methods allow each bus client to access the current value on the bus. If the bus is in an invalid state (i.e. has a high-impedance value or an undefined value because it is being driven by more than one client simultaneously), then the get() method will return 0 and the fromBusValid() method will return False. In all other cases, the fromBusValid() method will return True and the get() method will return the current value of the bus.

ZBusClientIFC				
Method			Argument	
Name	Type	Description	Name	Description
drive	Action	Drives a current value on to the bus	value	The value being put on the bus, datatype of value_type.
get	value_type	Returns the current value on the bus.		
fromBusValid	Bool	Returns False if the bus has a high-impedance value or is undefined.		

```
interface ZBusClientIFC #(type value_type) ;
   method Action      drive(value_type value);
   method value_type  get();
   method Bool        fromBusValid();
endinterface
```

The ZBusBusIFC interface connects to the bus structure itself using tri-state values. This interface is never accessed directly by the user.

```
interface ZBusBusIFC #(type value_type) ;
   method Action      fromBusSample(ZBit#(value_type) value, Bool isValid);
   method ZBit#(t)    toBusValue();
   method Bool        toBusCtl();
endinterface
```

Modules and Functions

The library provides a module constructor function, `mkZBusBuffer`, which allows the user to create a module which provides the `ZBusDualIFC` interface. This module provides the functionality of a tri-state buffer.

`mkZBusBuffer`	Creates a module which provides the `ZBusDualIFC` interface.
	`module mkZBusBuffer (ZBusDualIFC #(value_type))` `   provisos (Eq#(value_type), Bits#(value_type, size_value));`

The `mkZBus` module constructor function takes a list of `ZBusBusIFC` interfaces as arguments and creates a module which ties them all together in a bus.

`mkZBus`	Ties a list of `ZBusBusIFC` interfaces together in a bus.
	`module mkZBus#(List#(ZBusBusIFC#(value_type)) ifc_list)(Empty)` `   provisos (Eq#(value_type), Bits#(value_type, size_value));`

Examples - ZBus

Creating a tri-state buffer for a 32 bit signal. The interface is named `buffer_0`.

```
ZBusDualIFC#(Bit#(32)) buffer_0();
mkZBusBuffer inst_buffer_0(buffer_0);
```

Drive a value of 12 onto the associated bus.

```
buffer_0.clientIFC.drive(12);
```

The following code fragment demonstrates the use of the module `mkZBus`.

```
ZBusDualIFC#(Bit#(32)) buffer_0();
mkZBusBuffer inst_buffer_0(buffer_0);

ZBusDualIFC#(Bit#(32)) buffer_1();
mkZBusBuffer inst_buffer_1(buffer_1);

ZBusDualIFC#(Bit#(32)) buffer_2();
mkZBusBuffer inst_buffer_2(buffer_2);

List#(ZBusIFC#(Bit#(32))) ifc_list;

bus_ifc_list = cons(buffer_0.busIFC,
                    cons(buffer_1.busIFC,
                         cons(buffer_2.busIFC,
                                nil)));

Empty bus_ifc();
mkZBus#(bus_ifc_list) inst_bus(bus_ifc);
```

3.8.16 CRC

Package

```
import CRC :: * ;
```

Description

CRC's are designed to protect against common types of errors on communication channels. The `CRC` package defines modules to calculate a check value for each 8-bit block of data, which can then be verified to determine if data was transmitted and/or received correctly. There are many commonly used and standardized CRC algorithms. The `CRC` package provides both a generalized CRC module as well as module implementations for the CRC-CCITT, CRC-16-ANSI, and CRC-32 (IEEE 802.3) standards. The size of the CRC polynomial is polymorphic and the data size is a byte (`Bit#(8)`), which is relevant for many applications. The generalized module uses five arguments to define the CRC algorithm: the CRC polynomial, the initial CRC value, a fixed bit pattern to Xor with the remainder, a boolean indicating whether to reverse the data bit order and a boolean indicating whether to reverse the result bit order. By specifying these arguments, you can implement many CRC algorithms. This package provides modules for three specific algorithms by defining the arguments for those algorithms.

Interfaces and Methods

The CRC modules provide the `CRC` interface. The `add` method is used to calculate the check value on the `data` argument. In this package, the argument is always a `Bit#(8)`.

`CRC` Interface				
Method			Arguments	
Name	Type	Description	Name	Description
`add`	`Action`	Update the CRC	`Bit#(8) data`	8-bit data block
`clear`	`Action`	Reset to the initial value		
`result`	`Bit#(n)`	Returns the current value of the check value		
`complete`	`ActionValue(Bit#(n))`	Return the result and reset		

```
interface CRC#(numeric type n);
   method    Action   add(Bit#(8) data);
   method    Action   clear();
   method    Bit#(n)  result();
   method    ActionValue#(Bit#(n)) complete();
endinterface
```

Modules

The implementation of the generalized CRC module takes the following five arguments:

- `Bit#(n)` polynomial: the crc operation polynomial, for example $x^{16} + x^{12} + x^5 + 1$ is written as 'h1021
- `Bit#(n)` initval: the initial CRC value
- `Bit#(n)` finalXor: the result is xor'd with this value if desired
- `Bool` reflectData: if True, reverse the data bit order
- `Bool` reflectRemainder: if True, reverse the result bit order

`mkCRC`	The generalized CRC module. The provisos enforce the requirement that polynomial and initial value must be at least 8 bits.

```
module mkCRC#(  Bit#(n) polynomial
      , Bit#(n) initval
      , Bit#(n) finalXor
      , Bool    reflectData
      , Bool    reflectRemainder
      )(CRC#(n))
   provisos(  Add#(8, n8, n) );
```

CRC Arguments for Common Standards					
Name	polynomial	initval	finalXor	reflectData	reflectRemainder
CRC-CCITT	`'h1021`	`'hFFFF`	`'h0000`	`False`	`False`
CRC-16-ANSI	`'h8005`	`'h0000`	`'h0000`	`True`	`True`
CRC-32 (IEEE 802.3)	`'h04C11DB7`	`'hFFFFFFFF`	`'hFFFFFFFF`	`True`	`True`

`mkCRC_CCIT`	Implements the 16-bit CRC-CCITT standard. $(x^{16} + x^{15} + x^2 + 1)$.

```
module mkCRC_CCITT(CRC#(16));
```

`mkCRC16`	Implementation of the 16-bit CRC-16-ANSI standard. $(x^{16} + x^{15} + x^2 + 1)$.

```
module mkCRC16(CRC#(16));
```

`mkCRC32`	Implementation of the 32-bit CRC-32 (IEEE 802.3) standard. $(x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x^1 + 1)$

```
module mkCRC32(CRC#(32));
```

`reflect`	The `reflect` function reverses the data bits if the value of `reflectData` is `True`.

```
function Bit#(a) reflect(Bool doIt, Bit#(a) data);
   return (doIt) ? reverseBits(data) : data;
endfunction
```

3.8.17 OVLAssertions

Package

```
import OVLAssertions :: * ;
```

Description

The OVLAssertions package provides the BSV interfaces and wrapper modules necessary to allow BSV designs to include assertion checkers from the Open Verification Library (OVL). The OVL includes a set of assertion checkers that verify specific properties of a design. For more details on the complete OVL, refer to the Accellera Standard OVL Library Reference Manual (`http://www.accellera.org`).

Interfaces and Methods

The following interfaces are defined for use with the assertion modules. Each interface has one or more `Action` methods. Each method takes a single argument which is either a `Bool` or polymorphic.

AssertTest_IFC Used for assertions that check a test expression on every clock cycle.

AssertTest_IFC				
Method		Argument		
Name	Type	Name	Type	Description
`test`	`Action`	`test_value`	`a_type`	Expression to be checked.

```
interface AssertTest_IFC #(type a_type);
   method Action test(a_type test_value);
endinterface
```

AssertSampleTest_IFC Used for assertions that check a test expression on every clock cycle only if the sample, indicated by the boolean value `sample_test` is asserted.

AssertSampleTest_IFC				
Method		Argument		
Name	Type	Name	Type	Description
`sample`	`Action`	`sample_test`	`Bool`	Assertion only checked if `sample_test` is asserted.
`test`	`Action`	`test_value`	`a_type`	Expression to be checked.

```
interface AssertSampleTest_IFC #(type a_type);
   method Action sample(Bool sample_test);
   method Action test(a_type test_value);
endinterface
```

AssertStartTest_IFC Used for assertions that check a test expression only subsequent to a start_event, specified by the Boolean value `start_test`.

AssertStartTest_IFC				
Method		Argument		
Name	Type	Name	Type	Description
`start`	`Action`	`start_test`	`Bool`	Assertion only checked after start is asserted.
`test`	`Action`	`test_value`	`a_type`	Expression to be checked.

```
interface AssertStartTest_IFC #(type a_type);
   method Action start(Bool start_test);
   method Action test(a_type test_value);
endinterface
```

AssertStartStopTest_IFC Used to check a test expression between a start_event and a stop_event.

AssertStartStopTest_IFC				
Method		Argument		
Name	Type	Name	Type	Description
start	Action	start_test	Bool	Assertion only checked after start is asserted.
stop	Action	stop_test	Bool	Assertion only checked until the stop is asserted.
test	Action	test_value	a_type	Expression to be checked.

```
interface AssertStartStopTest_IFC #(type a_type);
   method Action start(Bool start_test);
   method Action stop(Bool stop_test);
   method Action test(a_type test_value);
endinterface
```

AssertTransitionTest_IFC Used to check a test expression that has a specified start state and next state, i.e. a transition.

AssertTransitionTest_IFC				
Method		Argument		
Name	Type	Name	Type	Description
test	Action	test_value	a_type	Expression that should transition to the `next_value`.
start	Action	start_test	a_type	Expression that indicates the start state for the assertion check. If the value of `start_test` equals the value of `test_value`, the check is performed.
next	Action	next_value	a_type	Expression that indicates the only valid next state for the assertion check.

```
interface AssertTransitionTest_IFC #(type a_type);
   method Action test(a_type test_value);
   method Action start(a_type start_value);
   method Action next(a_type next_value);
endinterface
```

AssertQuiescentTest_IFC Used to check that a test expression is equivalent to the specified expression when the sample state is asserted.

AssertQuiescentTest_IFC				
Method		Argument		
Name	Type	Name	Type	Description
sample	Action	sampe_test	Bool	Expression which initiates the quiescent assertion check when it transistions to true.
state	Action	state_value	a_type	Expression that should have the same value as check_value
check	Action	check_value	a_type	Expression state_value is compared to.

```
interface AssertQuiescentTest_IFC #(type a_type);
   method Action sample(Bool sample_test);
   method Action state(a_type state_value);
   method Action check(a_type check_value);
endinterface
```

AssertFifoTest_IFC Used with assertions checking a FIFO structure.

AssertFifoTest_IFC				
Method		Argument		
Name	Type	Name	Type	Description
push	Action	push_value	a_type	Expression which indicates the number of push operations that will occur during the current cycle.
pop	Action	pop_value	a_type	Expression which indicates the number of pop operations that will occur during the current cycle.

```
interface AssertFifoTest_IFC #(type a_type, type b_type);
   method Action push(a_type push_value);
   method Action pop(b_type pop_value);
endinterface
```

Datatypes

The parameters severity_level, property_type, msg, and coverage_level are common to all assertion checkers.

Common Parameters for all Assertion Checkers	
Parameter	Valid Values * indicates default value
severity_level	OVL_FATAL *OVL_ERROR OVL_WARNING OVL_Info
property_type	*OVL_ASSERT OVL_ASSUME OVL_IGNORE
msg	*VIOLATION
coverage_level	OVL_COVER_NONE *OVL_COVER_ALL OVL_COVER_SANITY OVL_COVER_BASIC OVL_COVER_CORNER OVL_COVER_STATISTIC

Each assertion checker may also use some subset of the following parameters.

Other Parameters for Assertion Checkers	
Parameter	Valid Values
action_on_new_start	OVL_IGNORE_NEW_START OVL_RESET_ON_NEW_START OVL_ERROR_ON_NEW_START
edge_type	OVL_NOEDGE OVL_POSEDGE OVL_NEGEDGE OVL_ANYEDGE
necessary_condition	OVL_TRIGGER_ON_MOST_PIPE OVL_TRIGGER_ON_FIRST_PIPE OVL_TRIGGER_ON_FIRST_NOPIPE
inactive	OVL_ALL_ZEROS OVL_ALL_ONES OVL_ONE_COLD

Other Parameters for Assertion Checkers	
Parameter	Valid Values
num_cks	Int#(32)
min_cks	Int#(32)
max_cks	Int#(32)
min_ack_cycle	Int#(32)
max_ack_cycle	Int#(32)
max_ack_length	Int#(32)
req_drop	Int#(32)
deassert_count	Int#(32)
depth	Int#(32)
value	a_type
min	a_type
max	a_type
check_overlapping	Bool
check_missing_start	Bool
simultaneous_push_pop	Bool

Setting Assertion Parameters

Each assertion checker module has a set of associated parameter values that can be customized for each module instantiation. The values for these parameters are passed to each checker module in the form of a single struct argument of type `OVLDefaults#(a)` A typical use scenario is illustrated below:

```
let defaults = mkOVLDefaults;

defaults.min_clks = 2;
defaults.max_clks = 3;

AssertTest_IFC#(Bool) assertWid <- bsv_assert_width(defaults);
```

The defaults struct (created by `mkOVLDefaults`) includes one field for each possible parameter. Initially each field includes the associated default value. By editing fields of the struct, individual parameter values can be modified as needed to be non-default values. The modified `defaults` struct is then provided as a module argument during instantiation.

Modules

Each module in this package corresponds to an assertion checker from the Open Verification Library (OVL). The BSV name for each module is the same as the OVL name with `bsv_` appended to the beginning of the name.

Module	`bsv_assert_always`
Description	Concurrent assertion that the value of the expression is always `True`.
Interface Used	`AssertTest_IFC`
Parameters	common assertion parameters
Module Declaration	`module bsv_assert_always#(OVLDefaults#(Bool) defaults)` `(AssertTest_IFC#(Bool));`

Module	`bsv_assert_always_on_edge`
Description	Checks that the test expression evaluates `True` whenever the sample method is asserted.
Interface Used	`AssertSampleTest_IFC`
Parameters	common assertion parameters `edge_type` (default value = `OVL_NOEDGE`)
Module Declaration	`module bsv_assert_always_on_edge#(OVLDefaults#(Bool)` `defaults)(AssertSampleTest_IFC#(Bool));`

Module	`bsv_assert_change`
Description	Checks that once the start method is asserted, the expression will change value within `num_cks` cycles.
Interface Used	`AssertStartTest_IFC`
Parameters	common assertion parameters `action_on_new_start` (default value = `OVL_IGNORE_NEW_START`) `num_cks` (default value = 1)

Module Declaration	`module bsv_assert_change#(OVLDefaults#(a_type) defaults)`
	`(AssertStartTest_IFC#(a_type))`
	`provisos (Bits#(a_type, sizea),`
	`Bounded#(a_type), Eq#(a_type));`

Module	`bsv_assert_cycle_sequence`
Description	Ensures that if a specified necessary condition occurs,it is followed by a specified sequence of events.
Interface Used	`AssertTest_IFC`
Parameters	common assertion parameters `necessary_condition` (default value = `OVL_TRIGGER_ON_MOST_PIPE`)

Module Declaration	`module bsv_assert_cycle_sequence#(OVLDefaults#(a_type)`
	`defaults)(AssertTest_IFC#(a_type))`
	`provisos (Bits#(a_type, sizea),`
	`Bounded#(a_type), Eq#(a_type));`

Module	`bsv_assert_decrement`
Description	Ensures that the expression decrements only by the value specifiedR.
Interface Used	`AssertTest_IFC`
Parameters	common assertion parameters `value` (default value = 1)

Module Declaration	<pre>module bsv_assert_decrement#(OVLDefaults#(a_type) defaults)
	               (AssertTest_IFC#(a_type))
	    provisos (Bits#(a_type, sizea), Literal#(a_type),
	              Bounded#(a_type), Eq#(a_type));</pre>

Module	`bsv_assert_delta`
Description	Ensures that the expression always changes by a value within the range specified by `min` and `max`.
Interface Used	`AssertTest_IFC`
Parameters	common assertion parameters `min` (default value = 1) `max` (default value = 1)

Module Declaration	<pre>module bsv_assert_delta#(OVLDefaults#(a_type) defaults)
	               (AssertTest_IFC#(a_type))
	    provisos (Bits#(a_type, sizea), Literal#(a_type),
	              Bounded#(a_type), Eq#(a_type));</pre>

Module	`bsv_assert_even_parity`
Description	Ensures that value of a specified expression has even parity, that is an even number of bits in the expression are active high.
Interface Used	`AssertTest_IFC`
Parameters	common assertion parameters

Module Declaration	<pre>module bsv_assert_even_parity#(OVLDefaults#(a_type)
	               defaults) (AssertTest_IFC#(a_type))
	    provisos (Bits#(a_type, sizea),
	              Bounded#(a_type), Eq#(a_type));</pre>

Module	`bsv_assert_fifo_index`
Description	Ensures that a FIFO-type structure never overflows or underflows. This checker can be configured to support multiple pushes (FIFO writes) and pops (FIFO reads) during the same clock cycle.
Interface Used	`AssertFifoTest_IFC`
Parameters	common assertion parameters `depth` (default value = 1) `simultaneous_push_pop` (default value = `True`)

Module Declaration	<pre>module bsv_assert_fifo_index#(OVLDefaults#(Bit#(0))
	            defaults)(AssertFifoTest_IFC#(a_type, b_type))
	    provisos (Bits#(a_type, sizea), Bits#(b_type, sizeb));</pre>

<table>
<tr><td>Module</td><td><code>bsv_assert_frame</code></td></tr>
<tr><td>Description</td><td>Checks that once the start method is asserted, the test expression evaluates true not before <code>min_cks</code> clock cycles and not after <code>max_cks</code> clock cycles.</td></tr>
<tr><td>Interface Used</td><td><code>AssertStartTest_IFC</code></td></tr>
<tr><td>Parameters</td><td>common assertion parameters
<code>action_on_new_start</code> (default value = <code>OVL_IGNORE_NEW_START</code>)
<code>min_cks</code> (default value = 1)
<code>max_cks</code> (default value = 1)</td></tr>
<tr><td>Module Declaration</td><td><pre><code>module bsv_assert_frame#(OVLDefaults#(Bool) defaults)
                (AssertStartTest_IFC#(Bool));</code></pre></td></tr>
</table>

<table>
<tr><td>Module</td><td><code>bsv_assert_handshake</code></td></tr>
<tr><td>Description</td><td>Ensures that the specified request and acknowledge signals follow a specified handshake protocol.</td></tr>
<tr><td>Interface Used</td><td><code>AssertStartTest_IFC</code></td></tr>
<tr><td>Parameters</td><td>common assertion parameters
<code>action_on_new_start</code> (default value = <code>OVL_IGNORE_NEW_START</code>)
<code>min_ack_cycle</code> (default value = 1)
<code>max_ack_cycle</code> (default value = 1)</td></tr>
<tr><td>Module Declaration</td><td><pre><code>module bsv_assert_handshake#(OVLDefaults#(Bool) defaults)
                (AssertStartTest_IFC#(Bool));</code></pre></td></tr>
</table>

<table>
<tr><td>Module</td><td><code>bsv_assert_implication</code></td></tr>
<tr><td>Description</td><td>Ensures that a specified consequent expression is <code>True</code> if the specified antecedent expression is <code>True</code>.</td></tr>
<tr><td>Interface Used</td><td><code>AssertStartTest_IFC</code></td></tr>
<tr><td>Parameters</td><td>common assertion parameters</td></tr>
<tr><td>Module Declaration</td><td><pre><code>module bsv_assert_implication#(OVLDefaults#(Bool) defaults)
                (AssertStartTest_IFC#(Bool));</code></pre></td></tr>
</table>

<table>
<tr><td>Module</td><td><code>bsv_assert_increment</code></td></tr>
<tr><td>Description</td><td>ensure that the test expression always increases by the value of specified by <code>value</code>.</td></tr>
<tr><td>Interface Used</td><td><code>AssertTest_IFC</code></td></tr>
<tr><td>Parameters</td><td>common assertion parameters
<code>value</code> (default value = 1)</td></tr>
<tr><td>Module Declaration</td><td><pre><code>module bsv_assert_increment#(OVLDefaults#(a_type) defaults)
                (AssertTest_IFC#(a_type))
    provisos (Bits#(a_type, sizea), Literal#(a_type),
              Bounded#(a_type), Eq#(a_type));</code></pre></td></tr>
</table>

Module	`bsv_assert_never`
Description	Ensures that the value of a specified expression is never `True`.
Interface Used	`AssertTest_IFC`
Parameters	common assertion parameters
Module Declaration	`module bsv_assert_never#(OVLDefaults#(Bool) defaults)` `(AssertTest_IFC#(Bool));`

Module	`bsv_assert_never_unknown`
Description	Ensures that the value of a specified expression contains only 0 and 1 bits when a qualifying expression is `True`.
Interface Used	`AssertStartTest_IFC`
Parameters	common assertion parameters

Module Declaration	`module bsv_assert_never_unknown#(OVLDefaults#(a_type)`
	`defaults)(AssertStartTest_IFC#(a_type))`
	`provisos (Bits#(a_type, sizea),`
	`Bounded#(a_type), Eq#(a_type));`

Module	`bsv_assert_never_unknown_async`
Description	Ensures that the value of a specified expression always contains only 0 and 1 bits
Interface Used	`AssertTest_IFC`
Parameters	common assertion parameters

Module Declaration	`module bsv_assert_never_unknown_async#(OVLDefaults#(a_type)`
	`defaults)(AssertTest_IFC#(a_type))`
	`provisos (Bits#(a_type, sizea), Literal#(a_type),`
	`Bounded#(a_type), Eq#(a_type));`

Module	`bsv_assert_next`
Description	Ensures that the value of the specified expression is true a specified number of cycles after a start event.
Interface Used	`AssertStartTest_IFC`
Parameters	common assertion parameters `num_cks` (default value = 1) `check_overlapping` (default value = `True`) `check_missing_start` (default value = `False`)
Module Declaration	`module bsv_assert_next#(OVLDefaults#(Bool) defaults)` `(AssertStartTest_IFC#(Bool));`

Module	`bsv_assert_no_overflow`
Description	Ensures that the value of the specified expression does not overflow.
Interface Used	`AssertTest_IFC`
Parameters	common assertion parameters `min` (default value = `minBound`) `max` (default value = `maxBound`)
Module Declaration	`module bsv_assert_no_overflow#(OVLDefaults#(a_type) defaults) (AssertTest_IFC#(a_type)) provisos (Bits#(a_type, sizea), Bounded#(a_type), Eq#(a_type));`

Module	`bsv_assert_no_transition`
Description	Ensures that the value of a specified expression does not transition from a start state to the specified next state.
Interface Used	`AssertTransitionTest_IFC`
Parameters	common assertion parameters
Module Declaration	`module bsv_assert_no_transition#(OVLDefaults#(a_type) defaults) (AssertTransitionTest_IFC#(a_type)) provisos (Bits#(a_type, sizea), Bounded#(a_type), Eq#(a_type));`

Module	`bsv_assert_no_underflow`
Description	Ensures that the value of the specified expression does not underflow.
Interface Used	`AssertTest_IFC`
Parameters	common assertion parameters `min` (default value = `minBound`) `max` (default value = `maxBound`)
Module Declaration	`module bsv_assert_no_underflow#(OVLDefaults#(a_type) defaults)(AssertTest_IFC#(a_type)) provisos (Bits#(a_type, sizea), Bounded#(a_type), Eq#(a_type));`

Module	`bsv_assert_odd_parity`
Description	Ensures that the specified expression had odd parity; that an odd number of bits in the expression are active high.
Interface Used	`AssertTest_IFC`
Parameters	common assertion parameters
Module Declaration	`module bsv_assert_odd_parity#(OVLDefaults#(a_type) defaults)(AssertTest_IFC#(a_type)) provisos (Bits#(a_type, sizea), Bounded#(a_type), Eq#(a_type));`

Module	`bsv_assert_one_cold`
Description	Ensures that exactly one bit of a variable is active low.
Interface Used	`AssertTest_IFC`
Parameters	common assertion parameters `inactive` (default value = `OLV_ONE_COLD`)

Module Declaration	<pre>module bsv_assert_one_cold#(OVLDefaults#(a_type) defaults)
	               (AssertTest_IFC#(a_type))
	    provisos (Bits#(a_type, sizea),
	              Bounded#(a_type), Eq#(a_type))</pre>

Module	`bsv_assert_one_hot`
Description	Ensures that exactly one bit of a variable is active high.
Interface Used	`AssertTest_IFC`
Parameters	common assertion parameters

Module Declaration	<pre>module bsv_assert_one_hot#(OVLDefaults#(a_type) defaults)
	               (AssertTest_IFC#(a_type))
	    provisos (Bits#(a_type, sizea),
	              Bounded#(a_type), Eq#(a_type));</pre>

Module	`bsv_assert_proposition`
Description	Ensures that the test expression is always combinationally `True`. Like `assert_always` except that the test expression is not sampled by the clock.
Interface Used	`AssertTest_IFC`
Parameters	common assertion parameters

Module Declaration	<pre>module bsv_assert_proposition#(OVLDefaults#(Bool) defaults)
	               (AssertTest_IFC#(Bool));</pre>

Module	`bsv_assert_quiescent_state`
Description	Ensures that the value of a specified state expression equals a corresponding check value if a specified sample event has transitioned to TRUE.
Interface Used	`AssertQuiescentTest_IFC`
Parameters	common assertion parameters

Module Declaration	<pre>module bsv_assert_quiescent_state#(OVLDefaults#(a_type)
	               defaults)(AssertQuiescentTest_IFC#(a_type))
	    provisos (Bits#(a_type, sizea),
	              Bounded#(a_type), Eq#(a_type));</pre>

Module	`bsv_assert_range`
Description	Ensure that an expression is always within a specified range.
Interface Used	`AssertTest_IFC`
Parameters	common assertion parameters `min` (default value = `minBound`) `max` (default value = `maxBound`)

Module Declaration	<pre>module bsv_assert_range#(OVLDefaults#(a_type) defaults)
	               (AssertTest_IFC#(a_type))
	    provisos (Bits#(a_type, sizea),
	              Bounded#(a_type), Eq#(a_type));</pre>

Module	`bsv_assert_time`
Description	Ensures that the expression remains `True` for a specified number of clock cycles after a start event.
Interface Used	`AssertStartTest_IFC`
Parameters	common assertion parameters `action_on_new_start` (default value = `OVL_IGNORE_NEW_START`) `num_cks` (default value = 1)

Module Declaration	<pre>module bsv_assert_time#(OVLDefaults#(Bool) defaults)
	               (AssertStartTest_IFC#(Bool));</pre>

Module	`bsv_assert_transition`
Description	Ensures that the value of a specified expression transitions properly froma start state to the specified next state.
Interface Used	`AssertTransitionTest_IFC`
Parameters	common assertion parameters

Module Declaration	<pre>module bsv_assert_transition#(OVLDefaults#(a_type)
	               defaults)(AssertTransitionTest_IFC#(a_type))
	    provisos (Bits#(a_type, sizea),
	              Bounded#(a_type), Eq#(a_type));</pre>

Module	`bsv_assert_unchange`
Description	Ensures that the value of the specified expression does not change during a specified number of clock cycles after a start event initiates checking.
Interface Used	`AssertStartTest_IFC`
Parameters	common assertion parameters `action_on_new_start` (default value = `OVL_IGNORE_NEW_START`) `num_cks` (default value = 1)

Module Declaration	<pre>module bsv_assert_unchange#(OVLDefaults#(a_type) defaults)
	               (AssertStartTest_IFC#(a_type))
	    provisos (Bits#(a_type, sizea),
	              Bounded#(a_type), Eq#(a_type));</pre>

Module	`bsv_assert_width`
Description	Ensures that when the test expression goes high it stays high for at least `min` and at most `max` clock cycles.
Interface Used	`AssertTest_IFC`
Parameters	common assertion parameters `min_cks` (default value = 1) `max_cks` (default value = 1)
Module Declaration	`module bsv_assert_width#(OVLDefaults#(Bool) defaults)` `(AssertTest_IFC#(Bool));`

Module	`bsv_assert_win_change`
Description	Ensures that the value of a specified expression changes in a specified window between a start event and a stop event.
Interface Used	`AssertStartStopTest_IFC`
Parameters	common assertion parameters

Module Declaration	`module bsv_assert_win_change#(OVLDefaults#(a_type)`
	`defaults)(AssertStartStopTest_IFC#(a_type))`
	`provisos (Bits#(a_type, sizea),`
	`Bounded#(a_type), Eq#(a_type));`

Module	`bsv_assert_win_unchange`
Description	Ensures that the value of a specified expression does not change in a specified window between a start event and a stop event.
Interface Used	`AssertStartStopTest_IFC`
Parameters	common assertion parameters

Module Declaration	`module bsv_assert_win_unchange#(OVLDefaults#(a_type)`
	`defaults)(AssertStartStopTest_IFC#(a_type))`
	`provisos (Bits#(a_type, sizea),`
	`Bounded#(a_type), Eq#(a_type));`

Module	`bsv_assert_window`
Description	Ensures that the value of a specified event is `True` between a specified window between a start event and a stop event.
Interface Used	`AssertStartStopTest_IFC`
Parameters	common assertion parameters
Module Declaration	`module bsv_assert_window#(OVLDefaults#(Bool) defaults)` `(AssertStartStopTest_IFC#(Bool));`

Module	`bsv_assert_zero_one_hot`
Description	ensure that exactly one bit of a variable is active high or zero.
Interface Used	`AssertTest_IFC`
Parameters	common assertion parameters
Module Declaration	`module bsv_assert_zero_one_hot#(OVLDefaults#(a_type) defaults)(AssertTest_IFC#(a_type)) provisos (Bits#(a_type, sizea), Bounded#(a_type), Eq#(a_type));`

Example using bsv_assert_increment

This example checks that a test expression is always incremented by a value of 3. The assertion passes for the first 10 increments and then starts failing when the increment amount is changed from 3 to 1.

```
import OVLAssertions::*;     // import the OVL Assertions package

module assertIncrement (Empty);

   Reg#(Bit#(8)) count <- mkReg(0);
   Reg#(Bit#(8)) test_expr <- mkReg(0);


   // set the default values
   let defaults = mkOVLDefaults;

   // override the default increment value and set = 3
   defaults.value = 3;

   // instantiate an instance of the module bsv_assert_increment using
   // the name assert_mod and the interface AssertTest_IFC
   AssertTest_IFC#(Bit#(8)) assert_mod <- bsv_assert_increment(defaults);

   rule every (True);              // Every clock cycle
      assert_mod.test(test_expr);  // the assertion is checked
   endrule

   rule increment (True);
      count <= count + 1;
      if (count < 10)                 // for 10 cycles
          test_expr <= test_expr + 3; // increment the expected amount
      else if (count < 15)
          test_expr <= test_expr + 1; // then start incrementing by 1
      else
          $finish;
   endrule
endmodule
```

Using The Library

In order to use the OVLAssertions package, a user must first download the source OVL library from Accellera (`http://www.accellera.org`). In addition, that library must be made available when building a simulation executable from the BSV generated Verilog.

If the bsc compiler is being used to generate the Verilog simulation executable, the `BSC_VSIM_FLAGS` environment variable can be used to set the required simulator flags that enable use of the OVL library.

For instance, if the `iverilog` simulator is being used and the OVL library is located in the directory `shared/std_ovl`, the `BSC_VSIM_FLAGS` environment variable can be set to `"-I shared/std_ovl -Y .vlib -y shared/std_ovl -DOVL_VERILOG=1 -DOVL_ASSERT_ON=1"`. These flags:

- Add `shared/std_ovl` to the Verilog and include search paths.
- Set `.vlib` as a possible file suffix.
- Set flags used in the OVL source code.

The exact flags to be used will differ based on what OVL behavior is desired and which Verilog simulator is being used.

3.8.18 Printf

Package

```
import Printf:: * ;
```

Description

The `Printf` package provides the `sprintf` function to allow users to construct strings using typical C printf patterns. The function supports a full range of C-style format options.

The `sprintf` function uses two advanced features, type classes and partial function application, to implement a variable number of arguments. That is why the type signature of the function includes a proviso for `SPrintfType`, also defined in this package.

Type classes

The `Printf` package includes the `SPrintf` and `PrintfArg` typeclasses. The proviso `SPrintf` specifies that the function can take a variable number of arguments, and further the types of those arguments can be displayed. This last requirement is captured by the `PrintfArg` typeclass, which is the class of types that can be displayed.

```
typeclass SPrintfType#(type t);
  function t spr(String fmt, List#(UPrintf) args);
endtypeclass
```

The `PrintfArg` typeclass defines a separate conversion for each type in the class.

```
typeclass PrintfArg#(type t);
   function UPrintf toUPrintf(t arg);
endtypeclass
```

Functions

`sprintf`	Constructs a string given a C-style format string and any input values for that format.
	`function r sprintf(String fmt) provisos (SPrintfType#(r));`

The `sprintf` function constructs a string from a format string followed by a variable number of arguments. Examples:

```
String s1 = sprintf("Hello");

Bit#(8) x = 0;
String s2 = sprintf("x = %d", x);

Real r = 1.2;
String s3 = sprintf("x = %d, r = %g", x, r);
```

The behavior of `sprintf` depends on the types of the arguments. If the type of an argument is unclear, you may be required to give specific types to those arguments.

For instance, an integer literal can represent many types, so you need to specify which one you are using:

```
String s4 = sprintf("%d, %d", 1, 2); // ambiguous

// Example of two ways to specify the type
UInt#(8) n = 1;
String s4 = sprintf("%d, %d", n, Bit#(4)'(2));
```

When calling `sprintf` on a value whose type is not known, as in a polymorphic function, you may be required to add a proviso to the function for the type variable.

The `PrintfArg` proviso on polymorphic functions is required when the type of the argument is not known. The type class instances define the conversion functions for each printable type.

```
function Action disp(t x);
   action
      String s = sprintf("x=%d", x);
      $display(s);
   endaction
endfunction
```

will generate an error message. By adding the proviso, the function compiles correctly:

```
function Action disp(t x)
   provisos( PrintfArg#(t) );
   action
      String s = sprintf("x=%d", x);
      $display(s);
   endaction
endfunction
```

3.8.19 BuildVector

Package

```
import BuildVector :: * ;
```

Description

The `BuildVector` package provides the `BuildVector` type class to implement a vector construction function which can take any number of arguments (>0).

In pseudo code, we can show this as:

```
   function Vector#(n, a) vec(a v1, a v2, ..., a vn);
```

Examples:

```
Vector#(3, Bool) v1 = vec(True, False, True);
Vector#(4, Integer) v2 = vec(2, 17, 22, 42);
```

Functions

vec	A function for creating a Vector of size `n` from `n` arguments. The variable number of arguments is implemented via the `BuildVector` typeclass, which is a proviso of this function.
	`function r vec(a x) provisos(BuildVector#(a,r,0));`

3.9 Multiple Clock Domains and Clock Generators

Package

```
import Clocks :: * ;
```

Description

The BSV `Clocks` library provide features to access and change the default clock. Moreover, there are hardware primitives to generate clocks of various shapes, plus several primitives which allow the safe crossing of signals and data from one clock domain to another.

The `Clocks` package uses the data types `Clock` and `Reset` as well as clock functions which are described below but defined in the `Prelude` package.

Each section describes a related group of modules, followed by a table indicating the Verilog modules used to implement the BSV modules.

Types and typeclasses

The `Clocks` package uses the abstract data types `Clock` and `Reset`, which are defined in the `Prelude` package. These are first class objects. Both `Clock` and `Reset` are in the `Eq` type class, meaning two values can be compared for equality.

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

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

`Reset` is an abstract type.

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

Type Classes for `Clock` and `Reset`									
	Bits	Eq	Literal	Arith	Ord	Bounded	Bitwise	Bit Reduction	Bit Extend
Clock		√							
Reset		√							

Example: Declaring a new clock

```
Clock clk0;
```

Example: Instantiating a register with clock and reset

```
Reg#(Byte) a <- mkReg(0, clocked_by clks0, reset_by rst0);
```

Functions

The following functions are defined in the `Prelude` package but are used with multiple clock domains.

Clock Functions	
`exposeCurrentClock`	This function returns a value of type `Clock`, which is the current clock of the module.
	`module exposeCurrentClock ( Clock c );`

`exposeCurrentReset`	This function returns a value of type `Reset`, which is the current reset of the module.
	`module exposeCurrentReset ( Reset r );`

Both `exposeCurrentClock` and `exposeCurrentReset` use the module instantiation syntax (`<-`) to return the value. Hence these can only be used from within a module.

Example: setting a reset to the current reset

```
Reset reset_value <- exposeCurrentReset;
```

Example: setting a clock to the current clock

```
Clock clock_value <- exposeCurrentClock;
```

`sameFamily`	A Boolean function which returns `True` if the clocks are in the same family, `False` if the clocks are not in the same family. Clocks in the same family have the same oscillator but may have different gate conditions.
	`function Bool sameFamily ( Clock clka, Clock clkb ) ;`

`isAncestor`	A Boolean function which returns `True` if `clka` is an ancestor of `clkb`, that is `clkb` is a gated version of `clka` (`clka` itself may be gated) or if `clka` and `clkb` are the same clock. The ancestry relation is a partial order (ie., reflexive, transitive and antisymmetric).
	`function Bool isAncestor ( Clock clka, Clock clkb ) ;`

`clockOf`	Returns the current clock of the object `obj`.
	`function Clock clockOf ( a_type obj ) ;`

noClock	Specifies a *null* clock, a clock where the oscillator never rises.	
	`function Clock noClock() ;`	

resetOf	Returns the current reset of the object `obj`.
	`function Reset resetOf ( a_type obj ) ;`

noReset	Specifies a *null* reset, a reset which is never asserted.
	`function Reset noReset() ;`

invertCurrentClock	Returns a value of type `Clock`, which is the inverted current clock of the module.
	`module invertCurrentClock(Clock);`

invertCurrentReset	Returns a value of type `Reset`, which is the inverted current reset of the module.
	`module invertCurrentReset(Reset);`

3.9.1 Clock Generators and Clock Manipulation

Description

This section provides modules to generate new clocks and to modify the existing clock.

The modules `mkAbsoluteClock`, `mkAbsoluteClockFull`, `mkClock`, and `mkUngatedClock` all define a new clock, one not based on the current clock. Both `mkAbsoluteClock` and `mkAbsoluteClockFull` define new oscillators and are not synthesizable. `mkClock` and `mkUngatedClock` use an existing oscillator to create a clock, and is synthesizable. The modules, `mkGatedClock` and `mkGatedClockFromCC` use existing clocks to generate another clock in the same family.

Interfaces and Methods

The `MakeClockIfc` supports user-defined clocks with irregular waveforms created with `mkClock` and `mkUngatedClock`, as opposed to the fixed-period waveforms created with the `mkAbsoluteClock` family.

MakeClockIfc Interface				
Method and subinterfaces			Arguments	
Name	Type	Description	Name	Description
setClockValue	Action	Changes the value of the clock at the next edge of the clock	value	Value the clock will be set to, must be a one bit type
getClockValue	one_bit_type	Retrieves the last value of the clock		
setGateCond	Action	Changes the gating condition	gate	Must be of the type Bool
getGateCond	Bool	Retrieves the last gating condition set		
new_clk	Interface	Clock interface provided by the module		

```
interface MakeClockIfc#(type one_bit_type);
   method Action       setClockValue(one_bit_type value) ;
   method one_bit_type getClockValue() ;
   method Action       setGateCond(Bool gate) ;
   method Bool         getGateCond() ;
   interface Clock     new_clk ;
endinterface
```

The GatedClockIfc is used for adding a gate to an existing clock.

GatedClockIfc Interface				
Method and subinterfaces			Arguments	
Name	Type	Description	Name	Description
setGateCond	Action	Changes the gating condition	gate	Must be of the type Bool
getGateCond	Bool	Retrieves the last gating condition set		
new_clk	Interface	Clock interface provided by the module		

```
interface GatedClockIfc ;
   method    Action setGateCond(Bool gate) ;
   method    Bool   getGateCond() ;
   interface Clock  new_clk ;
endinterface
```

Modules

The mkClock module creates a Clock type from a one-bit oscillator and a Boolean gate condition. There is no family relationship between the current clock and the clock generated by this module. The initial values of the oscillator and gate are passed as parameters to the module. When the module is out of reset, the oscillator value can be changed using the setClockValue method and the gate condition can be changed by calling the setGateCond method. The oscillator value and gate condition can be queried with the getClockValue and getGateCond methods, respectively. The clock created by mkClock is available as the new_clk subinterface. When setting the gate condition, the change does not affect the generated clock until it is low, to prevent glitches.

The mkUngatedClock module is an ungated version of the mkClock module. It takes only an oscillator argument (no gate argument) and returns the same new_clock interface. Since there is no gate, an error is returned if the design calls the setGateCond method. The getGateCond method always returns True.

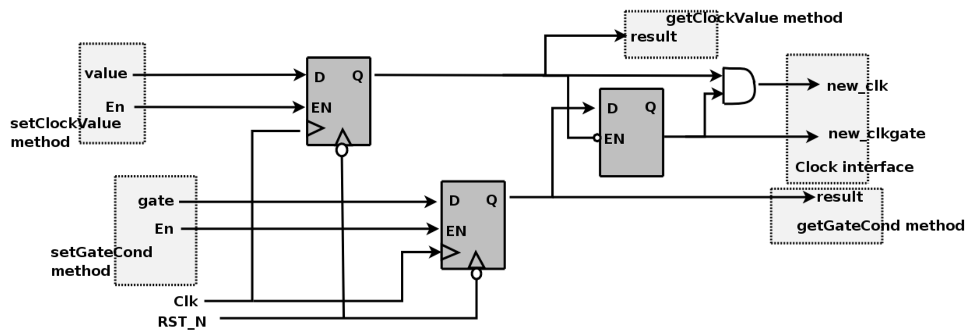

Figure 4: Clock Generator

`mkClock`	Creates a Clock type from a one-bit oscillator input, and a Boolean gate condition. There is no family relationship between the current clock and the clock generated by this module.

	`module mkClock #( one_bit_type initVal, Bool initGate)`
	`                ( MakeClockIfc#(one_bit_type) ifc )`
	`   provisos( Bits#(one_bit_type, 1) ) ;`

`mkUngatedClock`	Creates an ungated Clock type from a one-bit oscillator input. There is no family relationship between the current clock and the clock generated by this module.

	`module mkUngatedClock #( one_bit_type initVal)`
	`                ( MakeClockIfc#(one_bit_type) ifc )`
	`   provisos( Bits#(one_bit_type, 1) ) ;`

The `mkGatedClock` module adds (logic and) a Boolean gate condition to an existing clock, thus creating another clock in the same family. The source clock is provided as the argument `clk_in`. The gate condition is controlled by an asynchronously-reset register inside the module. The register is set with the `setGateCond` Action method of the interface and can be read with `getGateCond` method. The reset value of the gate condition register is provided as an instantiation parameter. The clock for the register (and thus these set and get methods) is the default clock of the module; to specify a clock other than the default clock, use the `clocked_by` directive.

`mkGatedClock`	Creates another clock in the same family by adding logic and a Boolean gate condition to the current clock.
	`module mkGatedClock#(Bool v) ( Clock clk_in, GatedClockIfc ifc );`

For convenience, we provide an alternate version in which the source clock is the default clock of the module

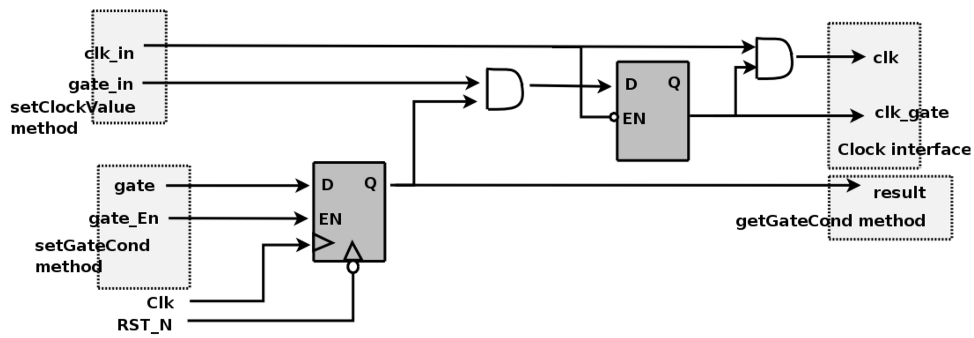

Figure 5: Gated Clock Generator

`mkGatedClockFromCC`	An alternate interface for the module `mkGatedClock` in which the source clock is the default clock of the module.
	`module mkGatedClockFromCC#(Bool v) ( GatedClockIfc ifc );`

The modules `mkAbsoluteClock` and `mkAbsoluteClockFull` provide parametizable clock generation modules which are *not* synthesizable, but may be useful for testbenches. In `mkAbsoluteClock`, the first rising edge (start) and the period are defined by parameters. These parameters are measured in Verilog delay times, which are usually specified during simulation with the `timescale` directive. Refer to the Verilog LRM for more details on delay times. s Additional parameters are provided by `mkAbsoluteClockFull`.

`mkAbsoluteClock`	The first rising edge (start) and period are defined by parameters. This module is not synthesizable.

	`module mkAbsoluteClock #( Integer start,`
	`                          Integer period )`
	`                          ( Clock );`

`mkAbsoluteClockFull`	The value `initValue` is held until time `start`, and then the clock oscillates. The value `not(initValue)` is held for time `compValTime`, followed by `initValue` held for time `initValTime`. Hence the clock period after startup is `compValTime + initValTime`. This module is not synthesizable.

	`module mkAbsoluteClockFull #( Integer start,`
	`                              Bit#(1) initValue,`
	`                              Integer compValTime,`
	`                              Integer initValTime )`
	`                              ( Clock );`

Verilog Modules

The BSV modules correspond to the following Verilog modules, which are found in the BSC Verilog library, `$BLUESPECDIR/Verilog/`.

BSV Module Name	Verilog Module Name
`mkAbsoluteClock` `mkAbsoluteClockFull`	`ClockGen.v`
`mkClock` `mkUngatedClock`	`MakeClock.v`
`mkGatedClock` `mkGatedClockFromCC`	`GatedClock.v`

3.9.2 Clock Multiplexing

Description

BSC provides two gated clock multiplexing primitives: a simple combinational multiplexor and a stateful module which generates an appropriate reset signal when the clock changes. The first multiplexor uses the interface `MuxClockIfc`, which includes an `Action` method to select the clock along with a `Clock` subinterface. The second multiplexor uses the interface `SelectClockIfc` which also has a `Reset` subinterface.

Ungated versions of these modules are also provided. The ungated versions are identical to the gated versions, except that the input and output clocks are ungated.

Interfaces and Methods

MuxClockIfc Interface				
Method and subinterfaces			Arguments	
Name	Type	Description	Name	Description
`select`	`Action`	Method used to select the clock based on the Boolean value `ab`	`ab`	if True, `clock_out` is taken from `aclk`
`clock_out`	`Interface`	Clock interface		

```
interface MuxClkIfc ;
   method    Action select ( Bool  ab ) ;
   interface Clock  clock_out ;
endinterface
```

SelectClockIfc Interface				
Method and subinterfaces			Arguments	
Name	Type	Description	Name	Description
`select`	`Action`	Method used to select the clock based on the Boolean value `ab`	`ab`	if True, clock_out is taken from `aclk`
`clock_out`	`Interface`	Clock interface		
`reset_out`	`Interface`	Reset interface		

```
interface SelectClkIfc ;
   method    Action select ( Bool  ab ) ;
   interface Clock  clock_out ;
   interface Reset  reset_out ;
endinterface
```

Modules

The `mkClockMux` module is a simple combinational multiplexor with a registered clock selection signal, which selects between clock inputs `aClk` and `bClk`. The provided Verilog module does not provide any glitch detection or removal logic; it is the responsibility of the user to provide additional logic to provide glitch-free behavior. The `mkClockMux` module uses two arguments and provides a Clock interface. The `aClk` is selected if `ab` is True, while `bClk` is selected otherwise.

The `mkUngatedClockMux` module is identical to the `mkClockMux` module except that the input and output clocks are ungated. The signals `aClkgate`, `bClkgate`, and `outClkgate` in figure 6 don't exist.

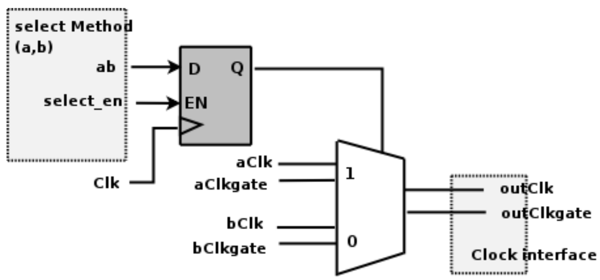

Figure 6: Clock Multiplexor

`mkClockMux`	Simple combinational multiplexor, which selects between aClk and bClk.
	`module mkClockMux ( Clock aClk, Clock bClk ) ( MuxClkIfc ) ;`

`mkUngatedClockMux`	Simple combinational multiplexor, which selects between aClk and bClk. None of the clocks are gated.
	`module mkUngatedClockMux ( Clock aClk, Clock bClk ) ( MuxClkIfc ) ;`

The `mkClockSelect` module is a clock multiplexor containing additional logic which generates a reset whenever a new clock is selected. As such, the interface for the module includes an `Action` method to select the clock (if `ab` is True clock_out is taken from `aClk`), provides a `Clock` interface, and also a `Reset` interface.

The constructor for the module uses two clock arguments, and provides the `MuxClockIfc` interface. The underlying Verilog module is `ClockSelect.v`; it is expected that users can substitute their own modules to meet any additional requirements they may have. The parameter `stages` is the number of clock cycles in which the reset is asserted after the clock selection changes.

The `mkUngatedClockSelect` module is identical to the `mkClockSelect` module except that the input and output clocks are ungated. The signals `aClkgate`, `bClkgate`, and `outClk_gate` in figure 7 don't exist.

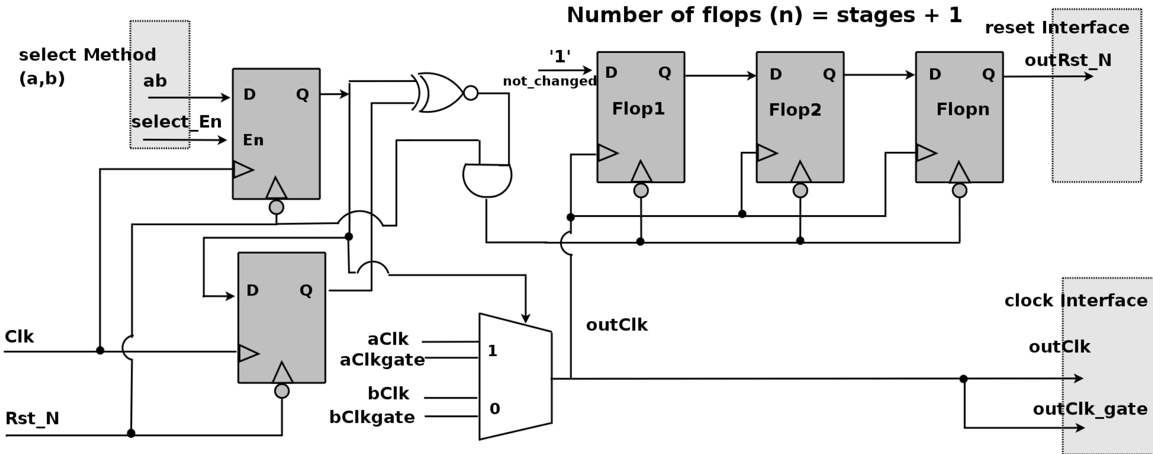

Figure 7: Clock Multiplexor with reset

`mkClockSelect`	Clock Multiplexor containing additional logic which generates a reset whenever a new clock is selected.
	`module mkClockSelect #( Integer stages, Clock aClk, Clock bClk, ( SelectClockIfc ) ;`

`mkUngatedClockSelect`	Clock Multiplexor containing additional logic which generates a reset whenever a new clock is selected. The input and output clocks are ungated.
	`module mkUngatedClockSelect #( Integer stages, Clock aClk, Clock bClk, ( SelectClockIfc ) ;`

Verilog Modules

The BSV modules correspond to the following Verilog modules, which are found in the BSC Verilog library, `$BLUESPECDIR/Verilog/`.

BSV Module Name	Verilog Module Name
`mkClockMux`	`ClockMux.v`
`mkClockSelect`	`ClockSelect.v`
`mkUngatedClockMux`	`UngatedClockMux.v`
`mkUngatedClockSelect`	`UngatedClockSelect.v`

3.9.3 Clock Division

Description

A clock divider provides a derived clock and also a `ClkNextRdy` signal, which indicates that the divided clock will rise in the next cycle. This signal is associated with the input clock, and can only be used within that clock domain.

The `AlignedFIFOs` package (Section 3.2.6) contains parameterized FIFO modules for creating synchronizing FIFOs between clock domains with aligned edges.

Data Types

The `ClkNextRdy` is a Boolean signal which indicates that the slow clock will rise in the next cycle.

```
typedef Bool ClkNextRdy ;
```

Interfaces and Methods

ClockDividerIfc Interface		
Name	Type	Description
`fastClock`	`Interface`	The original clock
`slowClock`	`Interface`	The derived clock
`clockReady`	`Bool`	Boolean value which indicates that the slow clock will rise in the next cycle. The method is in the clock domain of the fast clock.

```
interface ClockDividerIfc ;
    interface Clock      fastClock ;
    interface Clock      slowClock ;
    method    ClkNextRdy clockReady() ;
endinterface
```

Modules

The `divider` parameter may be any integer greater than 1. For even dividers the generated clock's duty cycle is 50%, while for odd dividers, the duty cycle is $(divider/2)/divider$. Since `divisor` is an integer, the remainder is truncated when divided. The current clock (or the `clocked_by` argument) is used as the source clock.

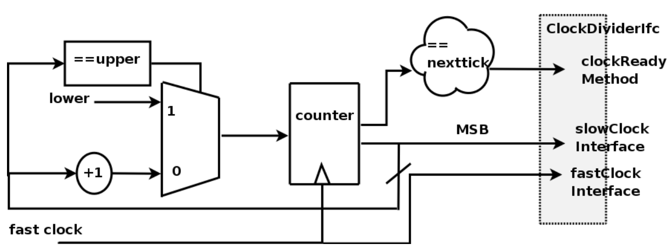

Figure 8: Clock Divider

`mkClockDivider`	Basic clock divider.	
	`module mkClockDivider #( Integer divisor ) ( ClockDividerIfc ) ;`	

`mkGatedClockDivider`	A gated verison of the basic clock divider.
	`module mkGatedClockDivider #( Integer divisor )( ClockDividerIfc ) ;`

The `mkClockDividerOffset` module provides a clock divider where the rising edge can be defined relative to other clock dividers which have the same divisor. An offset of value 2 will produce a rising edge one fast clock after a divider with offset 1. `mkClockDivider` is just `mkClockDividerOffset` with an offset of value 0.

`mkClockDividerOffset`	Provides a clock divider, where the rising edge can be defined relative to other clock dividers which have the same divisor.
	`module mkClockDividerOffset #( Integer divisor, Integer offset ) ( ClockDividerIfc ) ;`

The `mkClockInverter` and `mkGatedClockInverter` modules generate an inverted clock having the same period but opposite phase as the current clock. The `mkGatedClockInverter` is a gated version of `mkClockInverter`. The output clock includes a gate signal derived from the gate of the input clock.

`mkClockInverter`	Generates an inverted clock having the same period but opposite phase as the current clock.
	`module mkClockInverter ( ClockDividerIfc ) ;`

`mkGatedClockInverter`	A gated version of `mkClockInverter`.
	`module mkGatedClockInverter ( ClockDividerIfc ifc ) ;`

Verilog Modules

The BSV modules correspond to the following Verilog modules, which are found in the BSC Verilog library, `$BLUESPECDIR/Verilog/`.

BSV Module Name	Verilog Module Name
`mkClockDivider` `mkClockDividerOffset`	`ClockDiv.v`
`mkGatedClockDivider`	`GatedClockDiv.v`
`mkClockInverter`	`ClockInverter.v`
`mkGatedClockInverter`	`GatedClockInverter.v`

3.9.4 Bit Synchronizers

Description

Bit synchronizers are used to safely transfer one bit of data from one clock domain to another. More complicated synchronizers are provided in later sections.

Interfaces and Methods

The `SyncBitIfc` interface provides a `send` method which transmits one bit of information from one clock domain to the `read` method in a second domain.

SyncBitIfc Interface				
Methods			Arguments	
Name	Type	Description	Name	Description
`send`	`Action`	Transmits information from one clock domain to the second domain	`bitData`	One bit of information transmitted
`read`	`one_bit_type`	Reads one bit of data sent from a different clock domain		

```
interface SyncBitIfc #(type one_bit_type) ;
   method Action       send ( one_bit_type bitData ) ;
   method one_bit_type read () ;
endinterface
```

Modules

The `mkSyncBit`, `mkSyncBitFromCC` and `mkSyncBitToCC` modules provide a `SyncBitIfc` across clock domains. The send method is in one clock domain, and the read method is in a second clock domain, as shown in Figure 9. The `FromCC` and `ToCC` versions differ in that the `FromCC` module moves data *from* the current clock (module's clock), while the `ToCC` module moves data *to* the current clock domain. The hardware implementation is a two register synchronizer, which can be found in `SyncBit.v` in the BSC Verilog library directory.

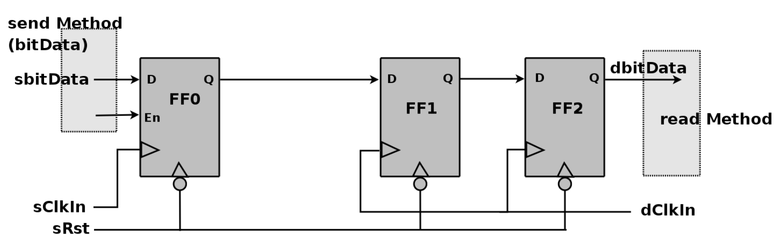

Figure 9: Bit Synchronizer

`mkSyncBit`	Moves data across clock domains. The in and out clocks, along with the input reset, are explicitly provided. The default clock and reset are ignored.
	`module mkSyncBit #( Clock sClkIn, Reset sRst, Clock dClkIn ) ( SyncBitIfc #(one_bit_type) ) provisos( Bits#(one_bit_type, 1)) ;`

mkSyncBitFromCC	Moves data from the current clock (the module's clock) to a different clock domain. The input clock and reset are the current clock and reset.	
	`module mkSyncBitFromCC #( Clock dClkIn ) ( SyncBitIfc #(one_bit_type) ) provisos( Bits#(one_bit_type, 1)) ;`	

mkSyncBitToCC	Moves data into the current clock domain. The output clock is the current clock. The current reset is ignored.	
	`module mkSyncBitToCC #( Clock sClkIn, Reset sRstIn ) ( SyncBitIfc #(one_bit_type) ) provisos( Bits#(one_bit_type, 1)) ;`	

The `mkSyncBit15` module (one and a half) and its variants provide the same interface as the `mkSyncBit` modules, but the underlying hardware is slightly modified, as shown in Figure 10. For these synchronizers, the first register clocked by the destination clock triggers on the falling edge of the clock.

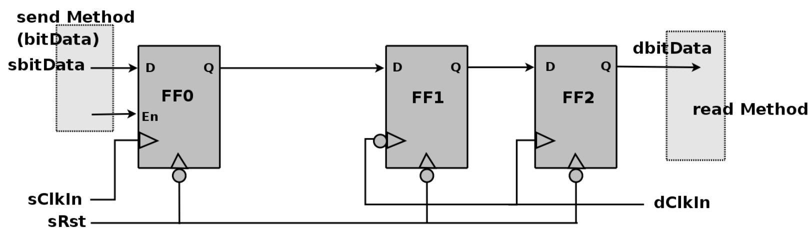

Figure 10: Bit Synchronizer 1.5 - first register in destination domain triggers on falling edge

mkSyncBit15	Similar to `mkSyncBit` except it triggers on the falling edge of the clock. The in and out clocks, along with the input reset, are explicitly provided. The default clock and reset are ignored.	
	`module mkSyncBit15 #( Clock sClkIn, Reset sRst, Clock dClkIn ) ( SyncBitIfc #(one_bit_type) ) provisos( Bits#(one_bit_type, 1)) ;`	

mkSyncBit15FromCC	Moves data from the current clock and is triggered on the falling edge of the clock. The input clock and reset are the current clock and reset.	
	`module mkSyncBit15FromCC #(Clock dClkIn) (SyncBitIfc #(one_bit_type)) provisos( Bits#(one_bit_type, 1)) ;`	

<table>
<tr><td rowspan="2"><code>mkSyncBit15ToCC</code></td><td>Moves data into the current clock domain and is triggered on the falling edge of the clock. The output clock is the current clock. The current reset is ignored.</td></tr>
<tr><td><pre>module mkSyncBit15ToCC #(Clock sClkIn, Reset sRstIn)
                         (SyncBitIfc #(one_bit_type))
   provisos(Bits#(one_bit_type, 1)) ;</pre></td></tr>
</table>

The `mkSyncBit1` module, shown in Figure 11, also provides the same interface but only uses one register in the destination domain. Synchronizers like this, which use only one register, are not generally used since meta-stable output is more probable. However, one can use this synchronizer provided special meta-stable resistant flops are selected during physical synthesis or (for example) if the output is immediately registered.

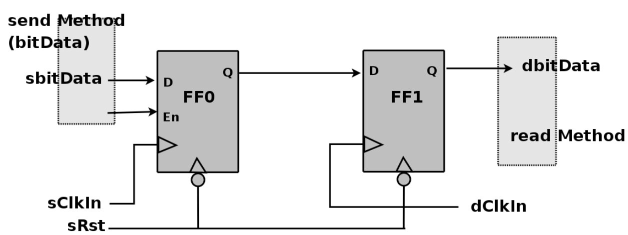

Figure 11: Bit Synchronizer 1.0 - single register in destination domain

<table>
<tr><td rowspan="2"><code>mkSyncBit1</code></td><td>Moves data from one clock domain to another clock domain, with only one register in the destination domain. The in and out clocks, along with the input reset, are explicitly provided. The default clock and reset are ignored.</td></tr>
<tr><td><pre>module mkSyncBit1 #(Clock sClkIn, Reset sRst,
                     Clock dClkIn)
                   (SyncBitIfc #(one_bit_type))
   provisos(Bits#(one_bit_type, 1)) ;</pre></td></tr>
</table>

<table>
<tr><td rowspan="2"><code>mkSyncBit1FromCC</code></td><td>Moves data from the current clock domain, with only one register in the destination domain. The input clock and reset are the current clock and reset.</td></tr>
<tr><td><pre>module mkSyncBit1FromCC #(Clock dClkIn)
                          (SyncBitIfc #(one_bit_type))
   provisos(Bits #(one_bit_type, 1)) ;</pre></td></tr>
</table>

`mkSyncBit1ToCC`	Moves data into the current clock domain, with only one register in the destination domain. The output clock is the current clock. The current reset is ignored.
	`module mkSyncBit1ToCC #( Clock sClkIn, Reset sRstIn ) ( SyncBitIfc #(one_bit_type) ) provisos( Bits#(one_bit_type, 1)) ;`

The `mkSyncBit05` module is similar to `mkSyncBit1`, but the destination register triggers on the falling edge of the clock, as shown in Figure 12.

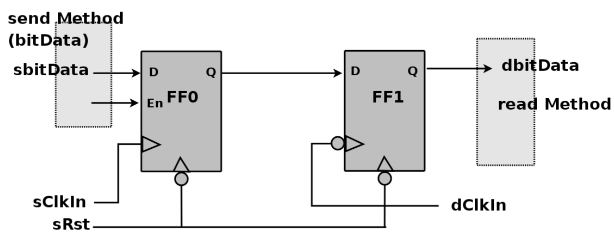

Figure 12: Bit Synchronizer .5 - first register in destination domain triggers on falling edge

`mkSyncBit05`	Moves data from one clock domain to another clock domain, with only one register in the destination domain. The destination register triggers on the falling edge of the clock. The in and out clocks, along with the input reset, are explicitly provided. The default clock and reset are ignored.
	`module mkSyncBit05 #( Clock sClkIn, Reset sRst, Clock dClkIn ) ( SyncBitIfc #(one_bit_type) ) provisos( Bits#(one_bit_type, 1)) ;`

`mkSyncBit05FromCC`	Moves data from the current clock domain, with only one register in the destination domain, the destination register triggers on the falling edge of the clock. The input clock and reset are the current clock and reset.
	`module mkSyncBit05FromCC #( Clock dClkIn ) (SyncBitIfc #(one_bit_type) ) provisos( Bits#(one_bit_type, 1)) ;`

`mkSyncBit05ToCC`	Moves data into the current clock domain, with only one register in the destination domain, the destination register triggers on the falling edge of the clock. The output clock is the current clock. The current reset is ignored.
	`module mkSyncBit05ToCC #( Clock sClkIn, Reset sRstIn ) ( SyncBitIfc #(one_bit_type) ) provisos( Bits#(one_bit_type, 1)) ;`

Verilog Modules

The BSV modules correspond to the following Verilog modules, which are found in the BSC Verilog library, `$BLUESPECDIR/Verilog/`.

BSV Module Name	Verilog Module Name

`mkSyncBit`
`mkSyncBitFromCC`
`mkSyncBitToCC`
	`SyncBit.v`

`mkSyncBit15`
`mkSyncBit15FromCC`
`mkSyncBit15ToCC`
	`SyncBit15.v`

`mkSyncBit1`
`mkSyncBit1FromCC`
`mkSyncBit1ToCC`
	`SyncBit1.v`

`mkSyncBit05`
`mkSyncBit05FromCC`
`mkSyncBit05ToCC`
	`SyncBit05.v`

3.9.5 Pulse Synchronizers

Description

Pulse synchronizers are used to transfer a pulse from one clock domain to another.

Interfaces and Methods

The `SyncPulseIfc` interface provides an Action method, `send`, which when invoked generates a True value on the `pulse` method in a second clock domain.

SyncPulseIfc Interface		
Methods		
Name	Type	Description
`send`	`Action`	Starts transmittling a pulse from one clock domain to the second clock domain.
`pulse`	`Bool`	Where the pulse is received in the second domain. `pulse` is True if a pulse is recieved in this cycle.

```
interface SyncPulseIfc ;
   method Action send  () ;
   method Bool   pulse () ;
endinterface
```

Modules

The `mkSyncPulse`, `mkSyncPulseFromCC` and `mkSyncPulseToCC` modules provide clock domain crossing modules for pulses. When the `send` method is called from the one clock domain, a pulse will be seen on the `read` method in the second. Note that there is no handshaking between the domains, so when sending data from a fast clock domain to a slower one, not all pulses sent may be seen in the slower receiving clock domain. The pulse delay is two destination clocks cycles.

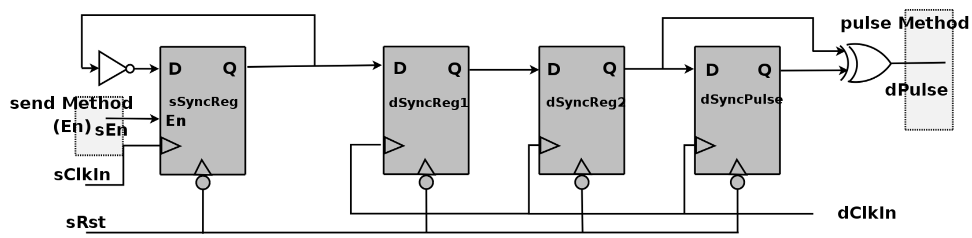

Figure 13: Pulse Synchronizer - no handshake

`mkSyncPulse`	Sends a pulse from one clock domain to another. The in and out clocks, along with the input reset, are explicitly provided. The default clock and reset are ignored.
	`module mkSyncPulse #( Clock sClkIn, Reset sRstIn, Clock dClkIn ) ( SyncPulseIfc ) ;`

`mkSyncPulseFromCC`	Sends a pulse from the current clock domain to the other clock domain. The input clock and reset are the current clock and reset.
	`module mkSyncPulseFromCC #( Clock dClkIn ) ( SyncPulseIfc ) ;`

`mkSyncPulseToCC`	Sends a pulse from the other clock domain to the current clock domain. The output clock is the current clock. The current reset is ignored.
	`module mkSyncPulseToCC #( Clock sClkIn, Reset sRstIn ) ( SyncPulseIfc ) ;`

The `mkSyncHandshake`, `mkSyncHandshakeFromCC` and `mkSyncHandshakeToCC` modules provide clock domain crossing modules for pulses in a similar way as `mkSyncPulse` modules, except that a handshake is provided in the `mkSyncHandshake` versions. The handshake enforces that another send does not occur before the first pulse crosses to the other domain. Note that this only guarantees that the pulse is seen in one clock cycle of the destination; it does not guarantee that the system on that side reacted to the pulse before it was gone. It is up to the designer to ensure this, if necessary. The modules are not ready in reset.

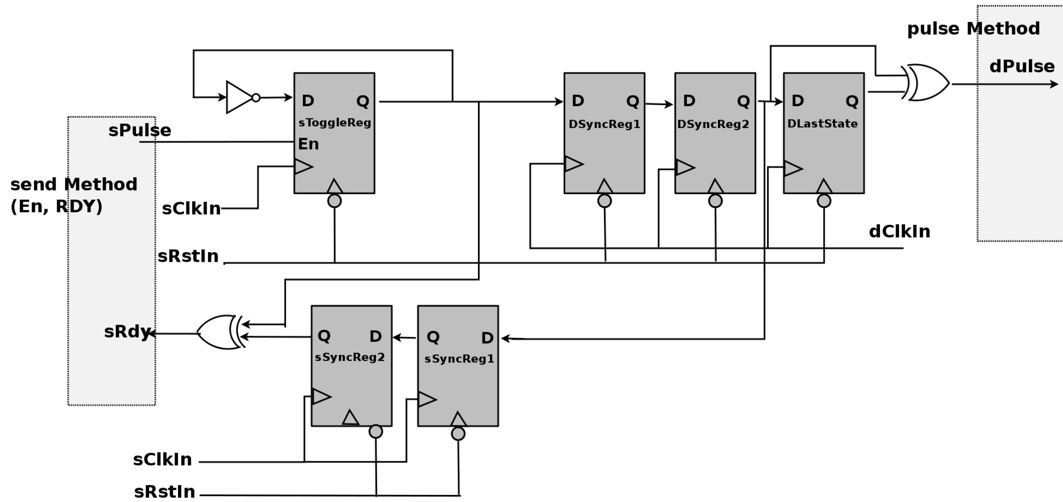

Figure 14: Pulse Synchronizer with handshake

The pulse delay from the `send` method to the `read` method is two destination clocks. The `send` method is re-enabled in two destination clock cycles plus two source clock cycles after the `send` method is called.

`mkSyncHandshake`	Sends a pulse from one clock domain to another clock domain with handshaking. The in and out clocks, along with the input reset, are explicitly provided. The default clock and reset are ignored.
	`module mkSyncHandshake #( Clock sClkIn, Reset sRstIn, Clock dClkIn ) ( SyncPulseIfc ) ;`

`mkSyncHandshakeFromCC`	Sends a pulse with a handshake from the current clock domain. The input clock and reset are the current clock and reset.
	`module mkSyncHandshakeFromCC #( Clock dClkIn ) ( SyncPulseIfc ) ;`

`mkSyncHandshakeToCC`	Sends a pulse with a handshake to the current clock domain. The output clock is the current clock. The current reset is ignored.
	`module mkSyncHandshakeToCC #( Clock sClkIn, Reset sRstIn ) ( SyncPulseIfc ) ;`

Verilog Modules

The BSV modules correspond to the following Verilog modules, which are found in the BSC Verilog library, `$BLUESPECDIR/Verilog/`.

BSV Module Name	Verilog Module Name
mkSyncPulse mkSyncPulseFromCC mkSyncPulseToCC	SyncPulse.v
mkSyncHandshake mkSyncHandshakeFromCC mkSyncHandshakeToCC	SyncHandshake.v

3.9.6 Word Synchronizers

Description

Word synchronizers are used to provide word synchronization across clock domains. The crossings are handshaked, such that a second write cannot occur until the first is acknowledged (that the data has been received, but the value may not have been read) by the destination side. The destination read is registered.

Interfaces and Methods

Word synchronizers use the common `Reg` interface (redescribed below), but there are a few subtle differences which the designer should be aware. First, the `_read` and `_write` methods are in different clock domains and, second, the `_write` method has an implicit "ready" condition which means that some synchronization modules cannot be written every clock cycle. Both of these conditions are handled automatically by BSC, relieving the designer of these tedious checks.

Reg Interface				
Method			Arguments	
Name	Type	Description	Name	Description
_write	Action	Writes a value x1	x1	Data to be written
_read	a_type	Returns the value of the register		

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

Modules

The `mkSyncReg`, `mkSyncRegToCC` and `mkSyncRegFromCC` modules provide word synchronization across clock domains.

<table>
<tr><td rowspan="2">mkSyncReg</td><td>Provides word synchronization across clock domains. The in and out clocks, along with the input reset, are explicitly provided. The default clock and reset are ignored.</td></tr>
<tr><td><pre>module mkSyncReg #(a_type initValue,
                    Clock sClkIn, Reset sRstIn,
                    Clock dClkIn)
                  (Reg #(a_type))
   provisos (Bits#(a_type, sa)) ;</pre></td></tr>
</table>

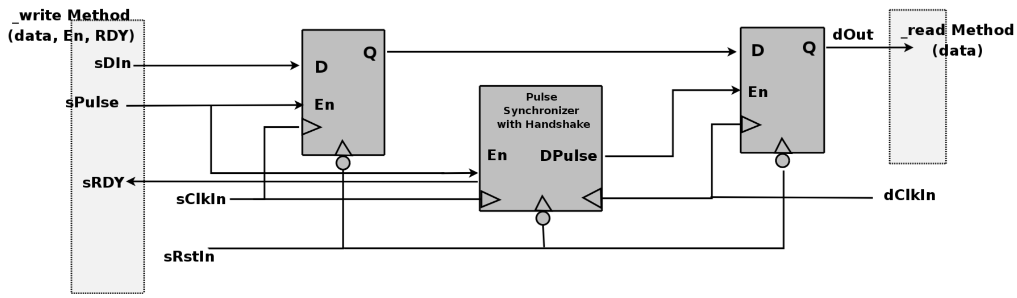

Figure 15: Register Synchronization Module (see Figure 14 for the pulse synchronizer with handshake)

mkSyncRegFromCC	Provides word synchronization from the current clock domain. The input clock and reset are the current clock and reset.
	`module mkSyncRegFromCC #( a_type initValue, Clock dClkIn ) ( Reg #(a_type) ) provisos (Bits#(a_type, sa)) ;`

```
module mkSyncRegFromCC #( a_type initValue,
                          Clock dClkIn )
                        ( Reg #(a_type) )
   provisos (Bits#(a_type, sa)) ;
```

mkSyncRegToCC	Provides word synchronization to the current clock domain. The output clock is the current clock. The current reset is ignored.

```
module mkSyncRegToCC #( a_type initValue,
                        Clock sClkIn, Reset sRstIn )
                      ( Reg #(a_type) )
   provisos (Bits#(a_type, sa)) ;
```

Verilog Modules

The BSV modules correspond to the following Verilog modules, which are found in the BSC Verilog library, `$BLUESPECDIR/Verilog/`.

BSV Module Name	Verilog Module Name
mkSyncReg mkSyncRegFromCC mkSyncRegToCC	SyncRegister.v

3.9.7 FIFO Synchronizers

Description

The `SyncFIFO` modules use FIFOs to synchronize data being sent across clock domains, providing registered full and empty signals (`notFull` and `notEmpty`). Additional FIFO synchronizers, `SyncFIFOLevel` and `SyncFIFOCount` can be found in the `FIFOLevel` package (Section 3.2.3).

Interfaces and Methods

The `SyncFIFOIfc` interface defines an interface similar to the FIFOF interface, except it does not have a `clear` method.

`SyncFIFOIfc` Interface				
Method			Arguments	
Name	Type	Description	Name	Description
`enq`	`Action`	Adds an entry to the FIFO	`sendData`	Data to be added
`deq`	`Action`	Removes the first entry from the FIFO		
`first`	`a_type`	Returns the first entry		
`notFull`	`Bool`	Returns True if there is space and you can `enq` into the FIFO		
`notEmpty`	`Bool`	Returns True if there are elements in the FIFO and you can `deq` from the FIFO		

```
interface SyncFIFOIfc #(type a_type) ;
   method Action enq ( a_type sendData ) ;
   method Action deq () ;
   method a_type first () ;
   method Bool notFull () ;
   method Bool notEmpty () ;
endinterface
```

Modules

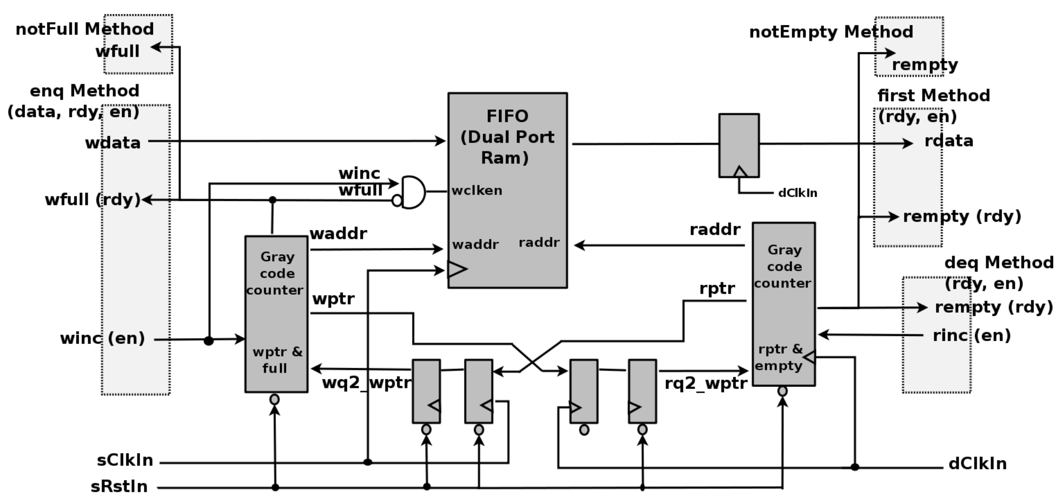

Figure 16: Synchronization FIFOs

The `mkSyncFIFO`, `mkSyncFIFOFromCC` and `mkSyncFIFOToCC` modules provide FIFOs for sending data across clock domains. Data items enqueued on the source side will arrive at the destination side and remain there until they are dequeued. The depth of the FIFO is specified by the `depth` parameter. The full and empty signals are registered. The module `mkSyncFIFO1` is a 1 element synchronized FIFO.

`mkSyncFIFO`	Provides a FIFO for sending data across clock domains. The `enq` method is in the source (`sClkIn`) domain, while the `deq` and `first` methods are in the destination (`dClkIn`) domain. The in and out clocks, along with the input reset, are explicitly provided. The default clock and reset are ignored.
	`module mkSyncFIFO #( Integer depth, Clock sClkIn, Reset sRstIn, Clock dClkIn ) ( SyncFIFOIfc #(a_type) ) provisos (Bits#(a_type, sa));`

```
module mkSyncFIFO #( Integer depth,
                     Clock sClkIn, Reset sRstIn,
                     Clock dClkIn )
                   ( SyncFIFOIfc #(a_type) )
   provisos (Bits#(a_type, sa));
```

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

```
module mkSyncFIFOFromCC #( Integer depth,
                           Clock dClkIn )
                         ( SyncFIFOIfc #(a_type) )
   provisos (Bits#(a_type, sa));
```

`mkSyncFIFOToCC`	Provides a FIFO to send data from a second clock domain into the current clock domain. The output clock is the current clock. The current reset is ignored.

```
module mkSyncFIFOToCC #( Integer depth,
                         Clock sClkIn, Reset sRstIn )
                       ( SyncFIFOIfc #(a_type) )
   provisos (Bits#(a_type, sa));
```

`mkSyncFIFO1`	Provides a 1 element FIFO for sending data across clock domains. The 1 element module does not have a dedicated output register and registers for full and empty, as in the depth > 1 module. This module should be used in clock crossing applications where complete FIFO handshaking is required, but data throughput or storage is minimal.

```
module mkSyncFIFO #( Clock sClkIn, Reset sRstIn,
                     Clock dClkIn )
                   ( SyncFIFOIfc #(a_type) )
   provisos (Bits#(a_type, sa));
```

Verilog Modules

The BSV modules correspond to the following Verilog modules, which are found in the BSC Verilog library, `$BLUESPECDIR/Verilog/`.

BSV Module Name	Verilog Module Name
mkSyncFIFO mkSyncFIFOFromCC mkSyncFIFOToCC	SyncFIFO.v
mkSyncFIFO1	SyncFIFO1.v

3.9.8 Asynchronous RAMs

Description

An asynchronous RAM provides a domain crossing by having its read and write methods in separate clock domains.

Interfaces and Methods

DualPortRamIfc Interface				
Method			Arguments	
Name	Type	Description	Name	Description
write	Action	Writes data to a an address in a RAM	wr_addr din	Address of datatype addr_t Data of datatype data_t
read	data_d	Reads the data from the RAM	rd_addr	Address to be read from

```
interface DualPortRamIfc #(type addr_t, type data_t);
   method Action      write( addr_t wr_addr, data_t  din );
   method data_t      read ( addr_t rd_addr);
endinterface: DualPortRamIfc
```

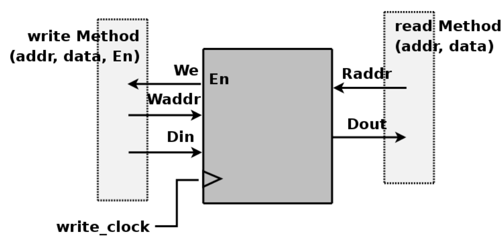

Figure 17: Ansynchronous RAM

mkDualRam	Provides an asynchronous RAM for when the read and the write methods are in separate clock domains. The write method is clocked by the default clock, the read method is not clocked.
	`module mkDualRam( DualPortRamIfc #(addr_t, data_t) ) provisos ( Bits#(addr_t, sa), Bits#(data_t, da) ) ;`

Verilog Modules

The BSV modules correspond to the following Verilog modules, which are found in the BSC Verilog library, `$BLUESPECDIR/Verilog/`.

BSV Module Name	Verilog Module Name
`mkDualRam`	`DualPortRam.v`

3.9.9 Null Crossing Primitives

Description

In these primitives, no synchronization is actually done. It is up to the designer to verify that it is safe for the signal to be used in the other domain. The `mkNullCrossingWire` is a wire synchronizer. The `mkNullCrossingReg` modules are equivalent to a register (`mkReg`, `mkRegA`, or `mkRegU` depending on the module) followed by a `mkNullCrossingWire`.

The older `mkNullCrossing` primitive is deprecated.

Interfaces

The `mkNullCrossingWire` module, shown in Figure 18, provides the `ReadOnly` interface which is defined in the Prelude library 2.4.8.

The `mkNullCrossingReg` modules provide the `CrossingReg` interface.

Interfaces and Methods

`CrossingReg` Interface				
Method			Arguments	
Name	Type	Description	Name	Description
`_write`	Action	Writes a value `datain`	`datain`	Data to be written.
`_read`	`a_type`	Returns the value of the register in the source clock domain		
`crossed`	`a_type`	Returns the value of the register in the destination clock domain		

```
interface CrossingReg #( type a_type ) ;
   method Action _write(a datain) ;
   method a_type _read() ;
   method a_type crossed() ;
endinterface
```

Modules

`mkNullCrossingWire`	Defines a synchronizer that contains only a wire. It is left up to the designer to ensure the clock crossing is safe.
	`module mkNullCrossingWire #( Clock dClk, a_type dataIn ) ( ReadOnly#(a_type) ) provisos (Bits#(a_type, sa)) ;`

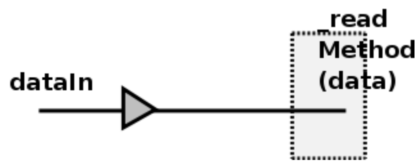

Figure 18: Wire synchronizer

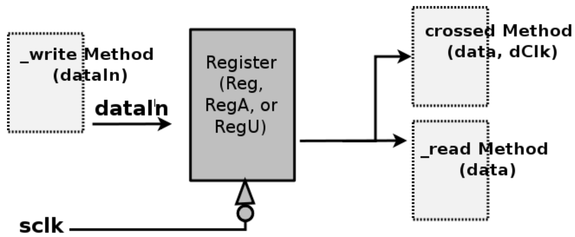

Figure 19: Register with wire synchronizer

`mkNullCrossingReg`	Defines a synchronizer that contains a register with a synchronous reset value, followed by a wire synchronizer. It is left up to the designer to ensure the clock crossing is safe.

	`module mkNullCrossingReg( Clock dClk, a_type resetval,`
	`                          CrossingReg#(a_type) ifc )`
	`   provisos (Bits#(a_type, sz_a)) ;`

`mkNullCrossingRegA`	Defines a synchronizer that contains a register with a given reset value where reset is asynchronous, followed by a wire synchronizer. It is left up to the designer to ensure the clock crossing is safe.

	`module mkNullCrossingRegA( Clock dClk, a_type resetval,`
	`                          CrossingReg#(a_type) ifc )`
	`   provisos (Bits#(a_type, sz_a)) ;`

`mkNullCrossingRegU`	Defines a synchronizer that contains a register without a reset, followed by a wire synchronizer. It is left up to the designer to ensure the clock crossing is safe.

	`module mkNullCrossingRegU( Clock dClk,`
	`                           CrossingReg#(a_type) ifc )`
	`   provisos (Bits#(a_type, sz_a)) ;`

Example: instantiating a null synchronizer

```
// domain2sig is domain1sig synchronized to clk0 with just a wire.
ReadOnly#(Bit#(2)) domain2sig <- mkNullCrossingWire (clk0, domain1sig);
```

Note: no synchronization is actually done. This is purely a way to tell BSC that it is safe to use the signal in the other domain. It is the responsibility of the designer to verify that this is correct.

There are some restrictions on the use of a `mkNullCrossingWire`. The expression used as the data argument must not have an implicit condition, and there cannot be another rule which is required to schedule before any method called in the expression.

`mkNullCrossingWire`s may not be used in sequence to pass a signal across multiple clock boundaries without synchronization. Once a signal has been crossed from one domain to a second domain without synchronization, it cannot be subsequently passed unsynchronized to a third domain (or back to the first domain).

Verilog Modules

The BSV modules correspond to the following Verilog modules, which are found in the BSC Verilog library, `$BLUESPECDIR/Verilog/`.

BSV Module Name	Verilog Module Name
`mkNullCrossingWire`	`BypassWire.v`

3.9.10 Reset Synchronization and Generation

Description

This section describes the interfaces and modules used to synchronize reset signals from one clock domain to another and to create reset signals. Reset generation converts a Boolean type to a Reset type, where the reset is associated with the default or `clocked_by` clock domain.

Interfaces and Methods

The `MakeResetIfc` interface is provided by the reset generators `mkReset` and `mkResetSync`.

`MakeResetIfc` Interface		
Method		
Name	Type	Description
`assertReset`	`Action`	Method used to assert the reset
`isAsserted`	`Bool`	Indicates whether the reset is asserted
`new_rst`	`Reset`	Generated output reset

```
interface MakeResetIfc;
   method Action assertReset();
   method Bool isAsserted();
   interface Reset new_rst;
endinterface
```

The interface `MuxRstIfc` is provided by the `mkResetMux` module.

`MuxRstIfc` Interface				
Method			Arguments	
Name	Type	Description	Name	Description
`select`	`Action`	Method used to select the reset based on the Boolean value `ab`	`ab`	Value determines which input reset to select
`reset_out`	`Reset`	Generated output reset		

```
interface MuxRstIfc;
   method Action select ( Bool ab );
   interface Reset reset_out;
endinterface
```

Modules

Reset Synchronization To synchronize resets from one clock domain to another, both synchronous and asynchronous modules are provided. The `stages` argument is the number of full clock cycles the output reset is held for after the input reset is deasserted. This is shown as the number of flops in figures 20 and 21. Specifying a 0 for the `stages` argument results in the creation of a simple wire between `sRst` and `dRstOut`.

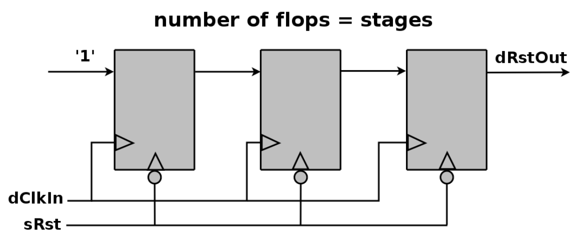

Figure 20: Module for asynchronous resets

mkAsyncReset	Provides synchronization of a source reset (`sRst`) to the destination domain. The output reset occurs immediately once the source reset is asserted.

	`module mkAsyncReset #( Integer stages,`
	`                       Reset sRst,`
	`                       Clock dClkIn )`
	`                     ( Reset ) ;`

mkAsyncResetFromCR	Provides synchronization of the current reset to the destination domain. There is no source reset `sRst` argument because it is taken from the current reset. The output reset occurs immediately once the current reset is asserted.

	`module mkAsyncResetFromCR #( Integer stages,`
	`                             Clock dClkIn )`
	`                           ( Reset ) ;`

The less common `mkSyncReset` modules are provided for convenience, but these modules *require* that `sRst` be held during a positive edge of `dClkIn` for the reset assertion to be detected. Both `mkSyncReset` and `mkSyncResetFromCR` use the model in figure 21.

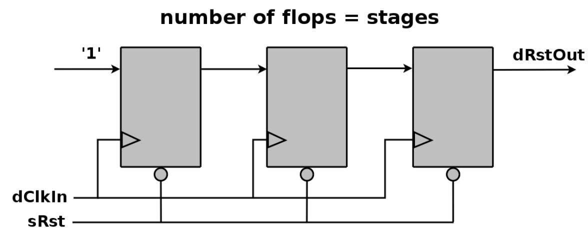

Figure 21: Module for synchronous resets

`mkSyncReset`	Provides synchronization of a source reset (`sRst`) to the destination domain. The reset is asserted at the next rising edge of the clock.

	`module mkSyncReset #( Integer stages`
	`Reset sRst,`
	`Clock dClkIn )`
	`( Reset ) ;`

`mkSyncResetFromCR`	Provides synchronization of the current reset to the destination domain. The reset is asserted at the next rising edge of the clock.

	`module mkSyncResetFromCR #( Integer stages`
	`Clock dClkIn )`
	`( Reset ) ;`

Example: instantiating a reset synchronizer

```
// 2 is the number of stages
Reset rstn2 <- mkAsyncResetFromCR (2, clk0);

// if stages = 0, the default reset is used directly
Reset rstn0 <- mkAsyncResetFromCR (0, clk0);
```

Reset Generation Two modules are provided for reset generation, `mkReset` and `mkResetSync`, where each module has one parameter, `stages`. The `stages` parameter is the number of full clock cycles the output reset is held after the `inRst`, as seen in figure 22, is deasserted. Specifying a 0 for the `stages` parameter results in the creation of a simple wire between the input register and the output reset. That is, the reset is asserted immediately and not held after the input reset is deasserted. It becomes the designer's responsibility to ensure that the input reset is asserted for sufficient time to allow the design to reset properly. The reset is controlled using the `assertReset` method of the `MakeResetIfc` interface.

The difference between `mkReset` and `mkResetSync` is that for the former, the assertion of reset is immediate, while the later asserts reset at the next rising edge of the clock. Note that use of `mkResetSync` is less common, since the reset requires clock edges to take effect; failure to assert reset for a clock edge will result in a reset not being seen at the output reset.

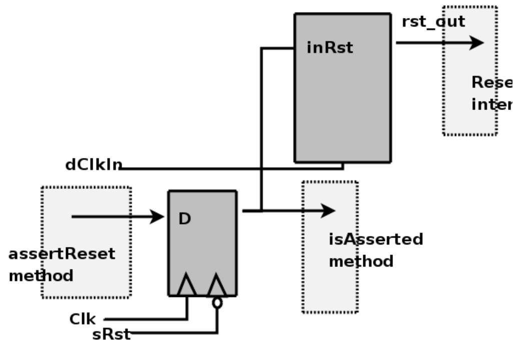

Figure 22: Module for generating resets

<table>
<tr><td rowspan="2"><code>mkReset</code></td><td>Provides conversion of a Boolean type to a Reset type, where the reset is associated with <code>dClkIn</code>. This module uses the model in figure 22. <code>startInRst</code> indicates the reset value of the register. If <code>startInRst</code> is True, the reset value of the register is 0, which means the output reset will be asserted whenever the currentReset (<code>sRst</code>) is asserted. <code>rst_out</code> will remain asserted for the number of clock cycles given by the stages parameter after <code>sRst</code> is deasserted. If <code>startInRst</code> is False, the output reset will not be asserted when <code>sRst</code> is asserted, but only when the <code>assert_reset</code> method is invoked. At the start of simulation <code>rst_out</code> will only be asserted if <code>startinRst</code> is True and <code>sRst</code> is initially asserted.</td></tr>
<tr><td><pre>module mkReset #(Integer stages,
                  Bool startInRst,
                  Clock dClkIn)
                (MakeResetIfc) ;</pre></td></tr>
</table>

<table>
<tr><td rowspan="2"><code>mkResetSync</code></td><td>Provides conversion of a Boolean type to a Reset type, where the reset is associated with <code>dClkIn</code> and the assertion of reset is at the next rising edge of the clock. This module uses the model in figure 22. <code>startInRst</code> indicates the reset value of the register. If <code>startInRst</code> is True, the reset value of the register is 0, which means the output reset will be asserted whenever the currentReset (<code>sRst</code>) is asserted. <code>rst_out</code> will remain asserted for the number of clock cycles given by the stages parameter after <code>sRst</code> is deasserted. If <code>startInRst</code> is False, the output reset will not be asserted when <code>sRst</code> is asserted, but only when the <code>assert_reset</code> method is invoked. At the start of simulation <code>rst_out</code> will only be asserted if <code>startinRst</code> is True and <code>sRst</code> is initially asserted.</td></tr>
<tr><td><pre>module mkResetSync #(Integer stages,
                      Bool startInRst,
                      Clock dClkIn)
                    (MakeResetIfc) ;</pre></td></tr>
</table>

A reset multiplexor `mkResetMux`, as seen in figure 23, creates one reset signal by selecting between two existing reset signals.

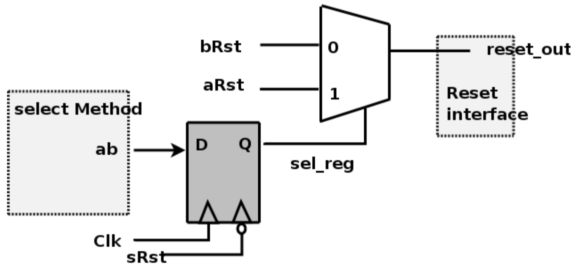

Figure 23: Reset Multiplexor

`mkResetMux`	Multiplexor which selects between two input resets, `aRst` and `bRst`, to create a single output reset `rst_out`. The reset is selected through a Boolean value provided to the `select` method where True selects `aRst`.
	`module mkResetMux #( Reset aRst, Reset bRst )` `( MuxRstIfc rst_out ) ;`

For testbenches, in which an absolute clock is being created, it is helpful to generate a reset for that clock. The module `mkInitialReset` is available for this purpose. It generates a reset which is asserted at the start of simulation. The reset is asserted for the number of cycles specified by the parameter `cycles`, counting the start of time as 1 cycle. Therefore, a `cycles` value of 1 will cause the reset to turn off at the first clock tick. This module is not synthesizable.

`mkInitialReset`	Generates a reset for `cycles` cycles, where the `cycles` parameter must be greater than zero. The `clocked_by` clause indicates the clock the reset is associated with. This module is not synthesizable.
	`module mkInitialReset #( Integer cycles )` `( Reset ) ;`

Example:

```
Clock c <- mkAbsoluteClock (10, 5);
// a reset associated with clock c:
Reset r <- mkInitialReset (2, clocked_by c);
```

When two reset signals need to be combined so that some logic can be reset when either input reset is asserted, the `mkResetEither` module can be used.

`mkResetEither`	Generates a reset which is asserted whenever either input reset is asserted.

	`module mkResetEither ( Reset aRst,`
	`Reset bRst)`
	`( Reset out_ifc );`

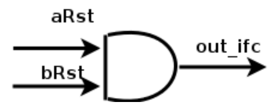

Figure 24: Reset Either

Example:

```
Reset r <- mkResetEither(rst1, rst2);
```

`mkResetInverter`	Generates an inverted Reset.
	`module mkResetInverter#(Reset in)` `                        (Reset);`

`isResetAsserted`	Tests whether a Reset is asserted, providing a Boolean value in the clock domain associated with the Reset.
	`module isResetAsserted( ReadOnly#(Bool) ifc ) ;`

Verilog Modules

The BSV modules correspond to the following Verilog modules, which are found in the BSC Verilog library, `$BLUESPECDIR/Verilog/`.

BSV Module Name	Verilog Module Name	Comments
`mkASyncReset`	`SyncReset0.v`	when stages==0
`mkASyncResetFromCR`	`SyncResetA.v`	
`mkSyncReset`	`SyncReset0.v`	when stages==0
`mkSyncResetFromCR`	`SyncReset.v`	
`mkReset`	`MakeReset0.v`	when stages==0
	`MakeResetA.v`	instantiates `SyncResetA`
`mkResetSync`	`MakeReset0.v`	when stages==0
	`MakeReset.v`	instantiates `SyncReset`
`mkResetMux`	`ResetMux.v`	
`mkResetEither`	`ResetEither.v`	
`mkResetInverter`	`ResetInverter.v`	
`isResetAsserted`	`ResetToBool.v`	

3.10 Special Collections

3.10.1 ModuleContext

Package

```
import ModuleContext :: * ;
```

Description

An ordinary Bluespec module, when instantiated, adds state elements and rules to the growing accumulation of elements and rules already in the design. In some designs, items other than state elements and rules must be accumulated as well. While there is a need to add these items, it is also desirable to keep these additional design details separate from the main design, keeping the natural structure of the design intact.

The `ModuleContext` package provides the capability of accumulating items and maintaining the compile-time state of additional items, in such a way that it doesn't change the structure of the original design.

The `ModuleContext` mechanism allows the designer to *hide* the details of the additional interfaces. Before the module can be synthesized, it must be converted (or *exposed*) into a module containing only rules and state elements, as the compiler does not know how to handle the other items. The `ModuleContext` package provides the mechanisms to allow additional items to be collected, processed, and exposed.

Types and Type Classes

The default BSV module type is `Module`, but you can define other BSV module types as well. The `ModuleContext` type is a variation on the `Module` type that allows additional items, other than states and rules, to be collected while elaborating the module structure.

The `ModuleContext` package defines the typeclass `Context`, which includes functions `getContext` and `putContext`. A `Context` typeclass has two type parameters: a module type (`mc1`) and the context (`c2`).

```
typeclass Context#(type mc1, type c2);
   module [mc1] getContext(c2) provisos (IsModule#(mc1, a));
   module [mc1] putContext#(c2 s)(Empty) provisos (IsModule#(mc1, a));
endtypeclass
```

A regular module type (`Module`) will have a context of `void`:

```
instance Context#(Module, void);
```

A module type of `ModuleContext` will return the context of the module:

```
instance Context#(ModuleContext#(st1), st1);
```

An instance is defined where the context type `st1` of the `ModuleContext` and the context type `st2` are different, but `Gettable` (as defined in `Hlist` Section 3.10.4):

```
instance Context#(ModuleContext#(st1), st2)
   provisos (Gettable#(st1, st2));
```

The modules `applyToContext` and `applyToContextM` are used to apply a function over a context. The `applyToContextM` modules is used for monadic functions.

<table>
<tr><td rowspan="2"><code>applyToContext</code></td><td>Applies a function over a context.</td></tr>
<tr><td><pre>module [mc1] applyToContext#(function c2 f(c2 c))(Empty)
   provisos (IsModule#(mc1, a), Context#(mc1, c2));</pre></td></tr>
</table>

`applyToContextM`	Applies a monadic function over a context.

	`module [mc1] applyToContextM#(function module#(c2) m(c2 c))`
	`(Empty)`
	`provisos (IsModule#(mc1, a), Context#(mc1, c2));`

ClockContext

The structure `ClockContext` is defined to be comprised of two clocks: `clk1` and `clk2` and two resets: `rst1` and `rst2`.

```
typedef struct {
                Clock clk1;
                Clock clk2;
                Reset rst1;
                Reset rst2;
               } ClockContext;
```

An `initClockContext` is defined with the values of both clocks set to `noClock` and both resets set to `noReset`:

```
ClockContext initClockContext = ClockContext {
   clk1: noClock, clk2: noClock, rst1: noReset, rst2: noReset };
```

Expose

The `Expose` typeclass converts a context to an interface for a synthesis boundary, converting it to a module type of `Module`. The `Expose` typeclass provides the modules `unburyContext` and `unburyContextWithClocks`.

```
typeclass Expose#(type c, type ifc)
   dependencies (c determines ifc);
```

An `HList` of contexts is convertible if its elements are, and results in a `Tuple` of subinterfaces.

```
instance Expose#(HList1#(ct1), ifc1)
   provisos (Expose#(ct1,ifc1));

instance Expose#(HCons#(c1,c2), Tuple2#(ifc1,ifc2))
   provisos (Expose#(c1,ifc1), Expose#(c2,ifc2));

instance Expose#(ClockContext, Empty);
```

The `unburyContext` module is for use at the top level of a module to be separately synthesized. It takes as an argument a module which is to be instantiated in a particular context, and an initial state for that context. The module is instantiated, and the final context converted into an extra interface, returned in pair with the intantiated module's own interface.

`unburyContext`	Converts a context to an interface for a synthesis boundary. An `HList` of contexts is convertible if its elements are, and results in a tuple of subinterfaces.

	`module unburyContext#(c x)(ifc);`
	
	`module unburyContext#(HList1#(ct1) c1)(ifc1);`
	
	`module unburyContext#(HCons#(c1,c2) c12)(Tuple2#(ifc1,ifc2));`
	
	`module unburyContext#(ClockContext x)();`

The `unburyContextWithClocks` takes a `ClockContext` along with the `Context` it is specifically handling

`unburyContextWithClocks`	Converts a context to an interface for a synthesis boundary and takes a ClockContext as a second argument.

	`module unburyContextWithClocks#(c x, ClockContext cc)`
	`(ifc);`
	
	`module unburyContextWithClocks#(HList1#(ct1) c1,`
	`ClockContext cc)(ifc1);`
	
	`module unburyContextWithClocks#(HCons#(c1,c2) c12,`
	`ClockContext cc)`
	`(Tuple2#(ifc1,ifc2));`
	
	`module unburyContextWithClocks#(ClockContext x,`
	`ClockContext cc)();`

Hide

The `Hide` typeclass provides the module `reburyContext`, which takes an interface as an argument (and provides an Empty interface). It is intended to be run in a context which can absorb the information from the interface. As with `Expose`, a `Tuple` of interfaces can be hidden if each element can be hidden.

`reburyContext`	Connects the provided interface with the surrounding context.

	`module [mc] reburyContext#(ifc i)(Empty);`
	
	`module [mc] reburyContext#(Empty i)(Empty);`
	
	`module [mc] reburyContext#(Tuple2#(ifc1,ifc2) i12)(Empty);`

ContextRun

The `ContextRun` and `ContextsRun` typeclasses provides modules to run modules in contexts. The module `runWithContext` runs a module with an entirely new context.

```
typeclass ContextRun#(type m, type c1, type ctx2)
   dependencies ((m, c1) determines ctx2);
```

```
typeclass ContextsRun#(type m, type c1, type ctx2)
   dependencies ((m, c1) determines ctx2);
```

<table>
<tr><td rowspan="2"><code>runWithContext</code></td><td>Runs a module with an entirely new context.</td></tr>
<tr><td><pre>module [m] runWithContext #(c1 initState,
                            ModuleContext#(ctx2, ifcType) mkI)
                           (Tuple2#(c1, ifcType));

module [ModuleContext#(ctx)] runWithContext#(c1 initState,
     ModuleContext#(HCons#(c1, ctx), ifcType) mkI)
                            (Tuple2#(c1, ifcType));

module [Module] runWithContext#(c1 initState,
                                ModuleContext#(c1,ifcType) mkI)
                               (Tuple2#(c1, ifcType));</pre></td></tr>
</table>

<table>
<tr><td rowspan="2"><code>runWithContexts</code></td><td>Runs a module with an entirely new context.</td></tr>
<tr><td><pre>module [m] runWithContexts#(c1 initState,
                            ModuleContext#(ctx2, ifcType) mkI)
                           (Tuple2#(c1, ifcType));

module [ModuleContext#(ctx)] runWithContexts#(c1 initState,
     ModuleContext#(ctx2, ifcType) mkI)
                            (Tuple2#(c1, ifcType));

module [Module] runWithContexts#(c1 initState,
                                 ModuleContext#(c1, ifcType) mkI)
                                (Tuple2#(c1, ifcType));</pre></td></tr>
</table>

Contexts.defines

BSC provides macros in the `Context.defines` file to handle the treatment of the module contexts at synthesis boundaries.

1. The designer defines a leaf or intermediate node module, with module type `[ErrorReporter]` or `[ErrorReporterA]`, appending a `0` to its name (e.g. `mkM0`). Elsewhere in the package the appropriate macro is chosen from the macros `SynthBoundary` and `SynthBoundaryWithClocks`.
2. The macro defines a synthesizable version of the module, `mkMV`, which provides the original interface together with an error-reporting subinterface. It also defines a module with the original name `mkM` to be used for instantiating the original module. It uses the Context mechanism to re-bury the error-reporting plumbing and returns the original interface of the original `mkM`) module

These macros assume that the complete module context (such as an `HList` of individual contexts) is named `CompleteContext` and that its initial value may be obtained from either `mkInitialCompleteContext` or `mkInitialCompleteContextWithClocks`.

Example Without Clocks

```
SynthBoundary(mkM,IM)
```

Becomes

```
(*synthesize*)
module [Module] mkMV(Tuple2#(CompleteContextIfc,IM));
   let init <- mkInitialCompleteContext;
   let _ifc <- unbury(init, mkM0);
   return _ifc;
endmodule

module [ModuleContext#(CompleteContext)] mkM(IM);
   let _ifc <- rebury(mkMV);
   return _ifc;
endmodule
```

Example With Clocks

```
SynthBoundaryWithClocks(mkM,IM)
```

Becomes

```
(*synthesize*)
module [Module] mkMV#(Clock c1,Reset r1,Clock c2,Reset r2)(Tuple2#(CompleteContextIfc,IM));
   let init <- mkInitialCompleteContextWithClocks(c1, r1, c2, r2);
   let _ifc <- unburyWithClocks(initialCompleteContext, c1, r1, c2, r2, mkM0);
   return _ifc;
endmodule

module [ModuleContext#(CompleteContext)] mkM(IM);
   let _ifc <- reburyWithClocks(mkMV);
   return _ifc;
endmodule
```

3.10.2 ModuleCollect

Package

```
import ModuleCollect :: * ;
```

Description

The `ModuleCollect` package provides the capability of adding additional items, such as configuration bus connections, to a design in such a way that it does not change the structure of the design. This section provides a brief overview of the package. For a more detailed description of its usage, see the `CBus` package (3.10.3), which utilizes `ModuleCollect`. There is also a detailed example and more complete discussion of the `CBus` package in the configbus tutorial in the BSV/tutorials directory.

An ordinary Bluespec module, when instantiated, adds its own state elements and rules to the growing accumulation of state elements and rules defined in the design. In some designs, for example a configuration bus, additional items, such as the logic for the bus address decoding must be accumulated as well. While there is a need to add these items, it is also desirable to keep these additional design details separate from the main design, keeping the natural structure of the design intact.

The `ModuleCollect` mechanism allows the designer to *hide* the details of the additional interfaces. A module which is going to be synthesized must contain only rules and state elements, as the compiler does not know how to handle the additional items. Therefore, the collection must be brought into the open, or exposed, before the module can be synthesized. The ModuleCollect package provides the mechanisms to allow these additional items to be collected, processed and exposed.

Types and Type Classes

The ModuleCollect type is a variation on the Module type that allows additional items, other than states and rules, to be collected while elaborating the module structure. A module defining the accumulation of a special collection will have the type of ModuleCollect which is defined as a type of ModuleContext (Section 3.10.1):

```
typedef ModuleContext#(HList1#(UAList#(a))) ModuleCollect#(type a_type);
```

where a_type defines the type of the items being collected. The collection is kept as an HList, therefore each item in the collection does not have the same type.

Your new type of module is a ModuleCollect defined to collect a specific type. It is often convenient to give a name to your new type of module using the typedef keyword.

For example:

```
typedef ModuleCollect#(element_type, ifc_device)
        MyModuleType#(type ifc_device)
```

specifies a type named MyModuleType.

An ordinary module, one defined with the keyword module without a type in square brackets immediately after it, can be of any module type. It is polymorphic, and when instantiated takes the type of the surrounding module context. Only modules of type Module can be synthesized, so the *synthesize* attribute forces the type to be Module. This is equivalent to writing:

```
module [Module]...
```

Normally, all the modules instantiated inside a synthesized module take the type Module.

A module which is accumulating a collection must have the appropriate type, specified in square brackets immedately after the keyword, as shown in the following example:

```
module [AssertModule] mkAssertionReg...
```

The complete example is found later in this section. This implies that any module instantiating mkAssertionReg is no longer polymorphic, its type is constrained by the inner module, so it will have to be explicitly given the AssertModule type too. Note, however, that you can continue to instantiate other modules not concerned with the collection (for example, mkReg, mkFIFO, etc.) alongside mkAssertReg just as before. But now they will take the type AssertModule from the context instead of the type Module.

Since only modules of type Module can be synthesized, before this group of AssertModule instantiations can be synthesized, you must use exposeCollection to contain the collection in a top-level module of type Module.

Interfaces

The IWithCollection interface couples the normal module interface (the device interface) with the collection of collected items (the collection interface). This is the interface provided by the exposeCollection function. It separates the collection list and the device module interface, to allow the module to be synthesized.

```
interface IWithCollection #(type a, type i);
    method i device();
    method List#(a) collection();
endinterface: IWithCollection
```

OLD:

```
interface IWithCollection #(type collection_type, type item_type);
   interface item_type device();
   interface List#(collection_type) collection();
endinterface: IWithCollection
```

Modules and Functions

In the course of evaluating a module body during its instantiation, an item may be added to the current collection by using the function `addToCollection`.

`addToCollection`	Adds an item to the collection.
	`function ModuleCollect#(a_type, ifc) addToCollection(a_type item);`

Once a set of items has been collected, those items must be exposed before synthesis. The `exposeCollection` module constructor is used to bring the collection out into the open. The `exposeCollection` module takes as an argument a `ModuleCollect` module (`m`) with interface `ifc`, and provides an `IWithCollection` interface.

`exposeCollection`	Expose the collection to allow the module to be synthesized.
	`module exposeCollection#(ModuleCollect#(a_type, ifc) m) (IWithCollection#(a_type, ifc));`

Finally, the `ModuleCollect` package provides a function, `mapCollection`, to apply a function to each item in the current collection.

`mapCollection`	Apply a function to each item added to the collection within the second argument.
	`function ModuleCollect#(a_type, ifc) mapCollection(function a_type x1(a_type x1), ModuleCollect#(a_type, ifc) x2);`

Example - Assertion Wires

```
// This example shows excerpts of a design which places various
// test conditions (Boolean expressions) at random places in a design,
// and lights an LED (setting an external wire to 1), if the condition
// is ever satisfied.

import ModuleCollect::*;
import List::*;
import Vector::*;
import Assert::*;
```

```
// The desired interface at the top level is:
interface AssertionWires#(type n);
   method Bit#(n) wires;
   method Action clear;
endinterface

// The "wires" method tells which conditions have been set, and the
// "clear" method resets them all to 0.
// The items in our extra collection will be interfaces of the
// following type:

interface AssertionWire;
   method Integer index;   //Indicates which wire is to be set if
   method Bool fail;       // fail method ever returns true.
   method Action clear;
endinterface

// We next define the "AssertModule" type.  This is to behave like an
// ordinary module providing an interface of type "i", except that it
// also can collect items of type "AssertionWire":

typedef ModuleCollect#(AssertionWire, i) AssertModule#(type i);

typedef Tuple2#(AssertionWires#(n), i) AssertIfc#(type i, type n);

...

// The next definition shows how items are added to the collection.
// This is the module which will be instantiated at various places in
// the design, to test various conditions.  It takes one static
// parameter, "ix", to specify which wire is to carry this condition,
// and one dynamic parameter (one varying at run-time) "c", giving the
// value of the condition itself.

interface AssertionReg;
   method Action set;
   method Action clear;
endinterface

module [AssertModule] mkAssertionReg#(Integer ix)(AssertionReg);

   Reg#(Bool) cond <- mkReg(False);

   // an item is defined and added to the collection
   let item = (interface AssertionWire;
                 method index;
                    return (ix);
                 endmethod
                 method fail;
                    return(cond);
                 endmethod
                 method Action clear;
                     cond <= False;
                 endmethod
```

```
              endinterface);
   addToCollection(item);
   ...
endmodule

// the collection must be exposed before synthesis
module [Module] exposeAssertionWires#(AssertModule#(i) mkI)(AssertIfc#(i, n));

   IWithCollection#(AssertionWire, i) ecs <- exposeCollection(mkI);

   ...(c_ifc is created from the list ecs.collection)

   // deliver the array of values in the registers
   let dut_ifc = ecs.device;

   // return the values in the collection, and the ifc of the device
   return(tuple2(c_ifc, dut_ifc));
endmodule
```

3.10.3 CBus

Package

```
import CBus :: * ;
```

Description

The `CBus` package provides the interface, types and modules to implement a configuration bus capability providing access to the control and status registers in a given module hierarchy. This package utilizes the `ModuleCollect` package and functionality, as described in section 3.10.2. The `ModuleCollect` package allows items in addition to usual state elements and rules to be accumulated. This is required to collect up the interfaces of the control status registers included in a module and to add the associated logic and ports required to allow them to be accessed via a configuration bus.

Types and Type Classes

The type `CBusItem` defines the type of item to be collected by `ModuleCollect`. The items to be collected are the same as the ifc which we will later expose, so we use a type alias:

```
typedef CBus#(size_address, size_data)
        CBusItem #(type size_address, type size_data);
```

The type `ModWithCBus` defines the type of module which is collecting `CBusItem`s. An ordinary module, one not collecting anything other than state elements and rules, has the type `Module`. Since `CBusItem`s are being collected, a module type `ModWithCBus` is defined. When the module type is not `Module`, the type must be specified in square brackets immediately after the `module` keyword in the module definition.

```
typedef ModuleCollect#(CBusItem#(size_address, size_data), item)
        ModWithCBus#(type size_address, type size_data, type item);
```

Interface and Methods

The `CBus` interface provides `read` and `write` methods to access control status registers. It is polymorphic in terms of the size of the address bus (`size_address`) and size of the data bus (`size_data`).

CBus Interface	
Name	Description
write	Writes the data value to the register if and only if the value of addr matches the address of the register.
read	Returns the value of the associated register if and only if addr matches the register address. In all other cases the read method returns an Invalid value.

```
interface CBus#(type size_address, type size_data);
   method Action write(Bit#(size_address) addr, Bit#(size_data) data);
   (* always_ready *)
   method ActionValue#(Bit#(size_data)) read(Bit#(size_address) addr);
endinterface
```

The IWithCBus interface combines the CBus interface with a normal module interface. It is defined as a structured interface with two subinterfaces: cbus_ifc (the associated configuration bus interface) and device_ifc (the associated device interface). It is polymorphic in terms of the type of the configuation bus interface and the type of the device interface.

```
interface IWithCBus#(type cbus_IFC, type device_IFC);
   interface cbus_IFC cbus_ifc;
   interface device_IFC device_ifc;
endinterface
```

Modules

The collectCBusIFC module takes as an argument a module with an IWithCBus interface, adds the associated CBus interface to the current collection (using addToCollection from the ModuleCollect package), and returns a module with the normal interface. Note that collectCBusIFC is of module type ModWithCBus.

collectCBusIFC	Adds the CBus to the collection and returns a module with just the device interface.
	`module [ModWithCBus#(size_address, size_data)] collectCBusIFC#(Module#(IWithCBus#( CBus#(size_address,size_data),i)) m)(i);`

The exposeCBusIFC module is used to create an IWithCBus interface given a module with a normal interface and an associated collection of CBusItems. This module takes as an argument a module of type ModWithCBus and provides an interface of type IWithCBus. The exposeCBusIFC module exposes the collected CBusItems, processes them, and provides a new combined interface. This module is synthesizable, because it is of type Module.

exposeCBusIFC	A module wrapper that takes a module with a normal interface, processes the collected CBusItems and provides an IWithCBus interface.
	`module [Module] exposeCBusIFC#(ModWithCBus#( size_address, size_data, item) sm) (IWithCBus#(CBus#(size_address, size_data), item));`

The CBus package provides a set of module primitives each of which adds a CBus interface to the collection and provides a normal Reg interface from the local block point of view. These modules are used in designs where a normal register would be used, and can be read and written to as registers from within the design.

<table>
<tr><td rowspan="2">mkCBRegR</td><td>A wrapper to provide a read only CBus interface to the collection and a normal Reg interface to the local block.</td></tr>
<tr><td><pre>module [ModWithCBus#(size_address, size_data)]
       mkCBRegR#(CRAddr#(size_address2) addr, r x)
                 (Reg#(r))
   provisos (Bits#(r, sr), Add#(k, sr, size_data),
             Add#(ignore, size_address2, size_address));</pre></td></tr>
</table>

<table>
<tr><td rowspan="2">mkCBRegRW</td><td>A wrapper to provide a read/write CBus interface to the collection and a normal Reg interface to the local block.</td></tr>
<tr><td><pre>module [ModWithCBus#(size_address, size_data)]
       mkCBRegRW#(CRAddr#(size_address2) addr, r x)
                 (Reg#(r))
   provisos (Bits#(r, sr), Add#(k, sr, size_data),
             Add#(ignore, size_address2, size_address));</pre></td></tr>
</table>

<table>
<tr><td rowspan="2">mkCBRegW</td><td>A wrapper to provide a write only CBus interface to the collection and a normal Reg interface to the local block.</td></tr>
<tr><td><pre>module [ModWithCBus#(size_address, size_data)]
       mkCBRegW#(CRAddr#(size_address2) addr, r x)
                 (Reg#(r))
   provisos (Bits#(r, sr), Add#(k, sr, size_data),
             Add#(ignore, size_address2, size_address));</pre></td></tr>
</table>

<table>
<tr><td rowspan="2">mkCBRegRC</td><td>A wrapper to provide a read/clear CBus interface to the collection and a normal Reg interface to the local block. This register can read from the config bus but the write is clear mode; for each write bit a 1 means clear, while a 0 means don't clear.</td></tr>
<tr><td><pre>module [ModWithCBus#(size_address, size_data)]
       mkCBRegRC#(CRAddr#(size_address2) addr, r x)
                 (Reg#(r))
   provisos (Bits#(r, sr), Add#(k, sr, size_data),
             Add#(ignore, size_address2, size_address));</pre></td></tr>
</table>

The mkCBRegFile module wrapper adds a CBus interface to the collection and provides a RegFile interface to the design. This module is used in designs as a normal RegFile would be used.

`mkCBRegFile`	A wrapper to provide a normal `RegFile` interface and automatically add the `CBus` interface to the collection.

	`module [ModWithCBus#(size_address, size_data)]`
	`      mkCBRegFile#(Bit#(size_address) reg_addr,`
	`                   Bit#(size_address) size)`
	`                   (RegFile#(Bit#(size_address), r))`
	`   provisos (Bits#(r, sr), Add#(k, sr, size_data));`

Example

Provided here is a simple example of a CBus implementation. The example is comprised of three packages: `CfgDefines`, `Block`, and `Tb`. The `CfgDefines` package contains the definition for the configuration bus, `Block` is the design block, and `Tb` is the testbench which executes the block.

The `Block` package contains the local design. As seen in Figure 25, the configuration bus registers look like a single field from the CBus (`cfgResetAddr, cfgStateAddr, cfgStatusAddr`), while each field (`reset, init, cnt`, etc.) in the configuration bus registers looks like a regular register from from the local block point of view.

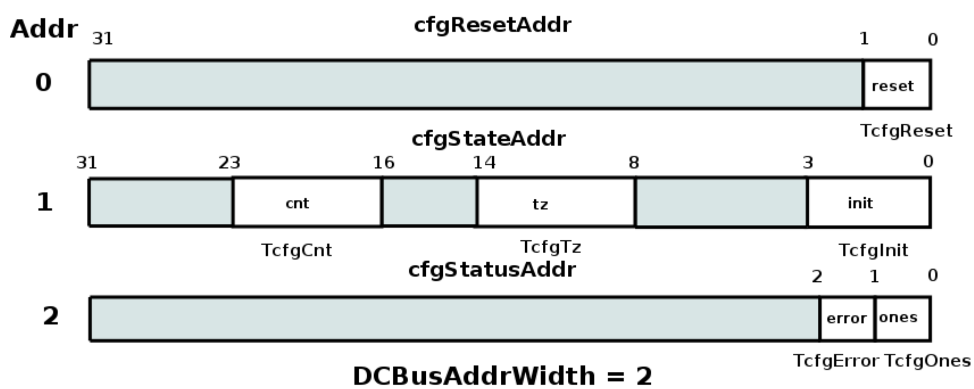

Figure 25: CBus Registers used in Block example

```
import CBus::*;       // this is a BSC library
import CfgDefines::*; // user defines - address,registers, etc

interface Block;
   // TODO: normally this block would have at least a few methods
   // Cbus interface is hidden, but it is there
endinterface

// In order to access the CBus at this parent, we need to expose the bus.
// Only modules of type [Module] can be synthesized.
module [Module] mkBlock(IWithCBus#(DCBus, Block));
   let ifc <- exposeCBusIFC( mkBlockInternal );
   return ifc;
endmodule

// Within this module the CBus looks like normal Registers.
// This module can't be synthesized directly.
```

```
// How these registers are combined into CBus registers is
// defined in the CfgDefines package.

module [DModWithCBus] mkBlockInternal( Block );
   // all registers are read/write from the local block point of view
   // config register interface types can be
   //   mkCBRegR  -> read only from config bus
   //   mkCBRegRW -> read/write from config bus
   //   mkCBRegW  -> write only from config bus
   //   mkCBRegRC -> read from config bus, write is clear mode
   //                i.e. for each bit a 1 means clear, 0 means don't clear
   // reset bit is write only from config bus
   // we presume that you use this bit to fire some local rules, etc
   Reg#(TCfgReset)  reg_reset_reset    <- mkCBRegW(cfg_reset_reset,    0 /* init val */);

   Reg#(TCfgInit)   reg_setup_init     <- mkCBRegRW(cfg_setup_init,    0 /* init val */);
   Reg#(TCfgTz)     reg_setup_tz       <- mkCBRegRW(cfg_setup_tz,      0 /* init val */);
   Reg#(TCfgCnt)    reg_setup_cnt      <- mkCBRegRW(cfg_setup_cnt,     1 /* init val */);

   Reg#(TCfgOnes)   reg_status_ones    <- mkCBRegRC(cfg_status_ones,   0 /* init val */);
   Reg#(TCfgError)  reg_status_error   <- mkCBRegRC(cfg_status_error,  0 /* init val */);

   // USER: you know have registers, so do whatever it is you do with registers :)
   // for instance
   rule bumpCounter ( reg_setup_cnt != unpack('1) );
      reg_setup_cnt <= reg_setup_cnt + 1;
   endrule

   rule watch4ones ( reg_setup_cnt == unpack('1) );
      reg_status_ones <= 1;
   endrule
endmodule
```

The `CfgDefines` package contains the user defines describing how the local registers are combined into the configuration bus.

```
package CfgDefines;
import CBus::*;

////////////////////////////////////////////////////////////////////////////////
/// basic defines
////////////////////////////////////////////////////////////////////////////////
// width of the address bus, it's easiest to use only the width of the bits needed
// but you may have other reasons for passing more bits around (even if some address
// bits are always 0)
typedef  2 DCBusAddrWidth;  // roof( log2( number_of_config_registers ) )

// the data bus width is probably defined in your spec
typedef 32 DCBusDataWidth;  // how wide is the data bus

////////////////////////////////////////////////////////////////////////////////
// Define the CBus
////////////////////////////////////////////////////////////////////////////////
typedef CBus#(  DCBusAddrWidth,DCBusDataWidth)          DCBus;
```

```
typedef CRAddr#(DCBusAddrWidth,DCBusDataWidth)          DCAddr;
typedef ModWithCBus#(DCBusAddrWidth, DCBusDataWidth, i) DModWithCBus#(type i);

////////////////////////////////////////////////////////////////////////////////
/// Configuration Register Types
////////////////////////////////////////////////////////////////////////////////
// these are configuration register from your design.  The basic
// idea is that you want to define types for each individual field
// and later on we specify which address and what offset bits these
// go to.  This means that config register address fields can
// actually be split across modules if need be.
//
typedef bit      TCfgReset;

typedef Bit#(4)  TCfgInit;
typedef Bit#(6)  TCfgTz;
typedef UInt#(8) TCfgCnt;

typedef bit      TCfgOnes;
typedef bit      TCfgError;

////////////////////////////////////////////////////////////////////////////////
/// configuration bus addresses
////////////////////////////////////////////////////////////////////////////////
Bit#(DCBusAddrWidth) cfgResetAddr  = 0; //
Bit#(DCBusAddrWidth) cfgStateAddr  = 1; //
Bit#(DCBusAddrWidth) cfgStatusAddr = 2; // maybe you really want this to be 0,4,8 ???

////////////////////////////////////////////////////////////////////////////////
/// Configuration Register Locations
////////////////////////////////////////////////////////////////////////////////
// DCAddr is a structure with two fields
//     DCBusAddrWidth a ; // this is the address
//                        // this does a pure comparison
//     Bit#(n)        o ; // this is the offset that this register
//                        // starts reading and writting at

DCAddr cfg_reset_reset  = DCAddr {a: cfgResetAddr, o:  0};  // bits 0:0

DCAddr cfg_setup_init   = DCAddr {a: cfgStateAddr, o:  0};  // bits 0:0
DCAddr cfg_setup_tz     = DCAddr {a: cfgStateAddr, o:  4};  // bits 9:4
DCAddr cfg_setup_cnt    = DCAddr {a: cfgStateAddr, o: 16};  // bits 24:16

DCAddr cfg_status_ones  = DCAddr {a: cfgStatusAddr, o:  0};  // bits 0:0
DCAddr cfg_status_error = DCAddr {a: cfgStatusAddr, o:  1};  // bits 1:1

////////////////////////////////////////////////////////////////////////////////
///
////////////////////////////////////////////////////////////////////////////////
endpackage
```

The `Tb` package executes the block.

```
import CBus::*;        // bsc library
```

```
import CfgDefines::*;  // address defines, etc
import Block::*;       // test block with cfg bus
import StmtFSM::*;     // just for creating a test sequence

(* synthesize *)
module mkTb ();
   // In order to access this cfg bus we need to use IWithCBus type
   IWithCBus#(DCBus,Block) dut <- mkBlock;

   Stmt test =
   seq
      // write the bits need to the proper address
      // generally this comes from software or some other packing scheme
      // you can, of course, create functions to pack up several fields
      // and drive that to bits of the correct width
      // For that matter, you could have your own shadow config registers
      // up here in the testbench to do the packing and unpacking for you
      dut.cbus_ifc.write( cfgResetAddr, unpack('1) );

      // put some ones in the status bits
      dut.cbus_ifc.write( cfgStateAddr, unpack('1) );

      // show that only the valid bits get written
      $display("TOP: state = %x at ", dut.cbus_ifc.read( cfgStateAddr ), $time);

      // clear out the bits
      dut.cbus_ifc.write( cfgStateAddr, 0 );

      // but the 'ones' bit was set when it saw all ones on the count
      // so read it to see that...
      $display("TOP: status = %x at ", dut.cbus_ifc.read( cfgStatusAddr ), $time);

      // now clear it
      dut.cbus_ifc.write( cfgStatusAddr, 1 );

      // see that it's clear
      $display("TOP: status = %x at ", dut.cbus_ifc.read( cfgStatusAddr ), $time);

      // and if we had other interface methods, that where not part of CBUS
      // we would access them via dut.device_ifc
   endseq;
   mkAutoFSM( test );
endmodule
```

3.10.4 HList

Package

```
import HList :: * ;
```

Description

The `HList` package defines a datatype `HList` which stores a list of data of different types. The package also provides typeclasses and functions to perform various list operations on the `HList` type.

The primitive data structures for an `HList` are `HNil` and the polymorphic `HCons`. The various functions are provided by typeclasses, one for each function.

The package defines a typeclass `Gettable` for finding (`getIt`) and replacing (`putIt`) items in an `HList`. This requires that all the items in the `HList` are different types. If two types are the same, they must be disambiguated by encapsulating at least one of them (but preferably each of them) in a new struct type. The functions of the `Gettable` typeclass require that the `HList` be flat (no nested `HLists`) and well-formed (terminating in `HNil`). That is, the target of a recursive search must be either the complete `hHead` or found within the `hTail`.

Types and type classes

The `HList` packages defines a typeclass `HList`:

```
typeclass HList#(type l);
```

The `HNil` datatype defines a nil instance, the empty set. An `HList` is usually terminated by a `HNil`.

```
typedef struct {} HNil deriving (Eq);
```

The `HCons` datatype is a structure with two members, a head of datatype `e` and a tail of datatype `l`.

```
typedef struct {
   e hd;
   l tl;
   } HCons#(type e, type l) deriving (Eq);
```

Functions

The various functions for heterogenous lists are provided by typeclasses, one for each functions.

<table>
<tr><td rowspan="2">HHead</td><td>Returns the first element of the list.</td></tr>
<tr><td><pre>typeclass HHead#(type l, type h)
   dependencies (l determines h);
   function h hHead(l x);
endtypeclass

instance HHead#(HCons#(e, l), e);</pre></td></tr>
</table>

<table>
<tr><td rowspan="2">HTail</td><td>Returns the tail element from the list.</td></tr>
<tr><td><pre>typeclass HTail#(type l, type lt)
   dependencies (l determines lt);
   function lt hTail(l xs);
endtypeclass

instance HTail#(HCons#(e, l), l);</pre></td></tr>
</table>

`HLength`	Returns a numeric value with the length of the list. For a `HNil`, will return 0.	

```
typeclass HLength#(type l, numeric type n);
endtypeclass

instance HLength#(HNil, 0);

instance HLength#(HCons#(e, l), nPlus1)
   provisos (HLength#(l, n), Add#(n,1,nPlus1));
```

`HAppend`	Appends two lists, returning the combined list. The elements do not have to be of the same data type. The combined list will be of type `l2`, and will contain all the elements of `xs` followed in order by all the elements of `ys`.

```
typeclass HAppend#(type l, type l1, type l2)
   dependencies ((l, l1) determines l2);
   function l2 hAppend(l xs, l1 ys);

instance HAppend#(HNil, l, l);

instance HAppend#(HCons#(e, l), l1, HCons#(e, l2))
   provisos (HList#(l), HAppend#(l, l1, l2));
```

`HSplit`	The `hSplit` function takes an `HList` of type `l` and returns a `Tuple2` of two `HLists`. This function is the inverse of `hAppend`.

```
typeclass HSplit#(type l, type l1, type l2);
   function Tuple2#(l1,l2) hSplit(l xs);
endtypeclass

instance HSplit#(HNil, HNil, HNil);

instance HSplit#(l, HNil, l);

instance HSplit#(HCons#(hd,tl), HCons#(hd,l3), l2)
   provisos (HSplit#(tl,l3,l2));
```

`Gettable`	This typeclass is for finding (`getIt`) and replacing (`putIt`) a particular element in an `HList`. All items in the `HList` must be of different types. If two types are the same, they should be disambiguated by encapsulating at least one of them (and preferably both of them) in a new struct type.

	`typeclass Gettable#(type c1, type c2);`
	`   function c2 getIt(c1 x);`
	`   function c1 putIt(c1 x, c2 y);`
	`endtypeclass`
	
	`instance Gettable#(HCons#(t1, t2), t1);`
	
	`instance Gettable#(HCons#(t1, t2), t3)`
	`   provisos (Gettable#(t2, t3));`

Small Lists

The `HList` packcage provides type definitions for small lists, ranging from 1 element to 8 elements, along with constructor functions to build the lists.

HList1

```
typedef HCons#(t, HNil)
        HList1#(type t);

function HList1#(t1) hList1(t1 x1) = hCons(x1, hNil);
```

HList2

```
typedef HCons#(t1, HCons#(t2, HNil))
        HList2#(type t1, type t2);

function HList2#(t1, t2) hList2(t1 x1, t2 x2) = hCons(x1, hCons(x2, hNil));
```

HList3

```
typedef HCons#(t1, HCons#(t2, HCons#(t3, HNil)))
        HList3#(type t1, type t2, type t3);

function HList3#(t1, t2, t3) hList3(t1 x1, t2 x2, t3 x3)
      = hCons(x1, hCons(x2, hCons(x3, hNil)));
```

HList4

```
typedef HCons#(t1, HCons#(t2, HCons#(t3, HCons#(t4, HNil))))
        HList4#(type t1, type t2, type t3, type t4);

function HList4#(t1, t2, t3, t4) hList4(t1 x1, t2 x2, t3 x3, t4 x4)
      = hCons(x1, hCons(x2, hCons(x3, hCons(x4, hNil))));
```

HList5

```
typedef HCons#(t1, HCons#(t2, HCons#(t3, HCons#(t4, HCons#(t5, HNil)))))
        HList5#(type t1, type t2, type t3, type t4, type t5);

function HList5#(t1, t2, t3, t4, t5) hList5(t1 x1, t2 x2, t3 x3, t4 x4, t5 x5)
      = hCons(x1, hCons(x2, hCons(x3, hCons(x4, hCons(x5, hNil)))));
```

HList6

```
typedef HCons#(t1, HCons#(t2, HCons#(t3, HCons#(t4, HCons#(t5, HCons#(t6, HNil))))))
        HList6#(type t1, type t2, type t3, type t4, type t5, type t6);

function HList6#(t1, t2, t3, t4, t5, t6)
   hList6(t1 x1, t2 x2, t3 x3, t4 x4, t5 x5, t6 x6)
      = hCons(x1, hCons(x2, hCons(x3, hCons(x4, hCons(x5, hCons(x6, hNil))))));
```

HList7

```
typedef HCons#(t1, HCons#(t2, HCons#(t3, HCons#(t4, HCons#(t5,
               HCons#(t6, HCons#(t7, HNil)))))))
        HList7#(type t1, type t2, type t3, type t4, type t5, type t6, type t7);

function HList7#(t1, t2, t3, t4, t5, t6, t7)
   hList7(t1 x1, t2 x2, t3 x3, t4 x4, t5 x5, t6 x6, t7 x7)
      = hCons(x1, hCons(x2, hCons(x3, hCons(x4, hCons(x5, hCons(x6, hCons(x7, hNil)))))));
```

HList8

```
typedef HCons#(t1, HCons#(t2, HCons#(t3, HCons#(t4, HCons#(t5,
               HCons#(t6, HCons#(t7, HCons#(t8, HNil))))))))
        HList8#(type t1, type t2, type t3, type t4, type t5, type t6, type t7, type t8);

function HList8#(t1, t2, t3, t4, t5, t6, t7, t8)
   hList8(t1 x1, t2 x2, t3 x3, t4 x4, t5 x5, t6 x6, t7 x7, t8 x8)
      = hCons(x1, hCons(x2, hCons(x3, hCons(x4, hCons(x5, hCons(x6,
              hCons(x7, hCons(x8, hNil))))))));
```

3.10.5 UnitAppendList

Package

```
import UnitAppendList :: * ;
```

Description

This provides a representation of lists for which append(x,y) is O(1), rather than O(length(x)) as in the normal representation; the downside is that there is no longer a unique representation for a given list. These lists are useful for situations in which the list is constructed by recursively amalgamating lists from sub-computations, and then subsequently processed. Functions for `map` and `mapM` are provided for processing sublists during construction. For final processing it is almost always preferable first to flatten the list (by a function also provided) into the conventional representation, thus eliminating empty subtrees.

Types and type classes

The `UnitAppendList` package defines the structure `UAList`:

```
typedef union tagged {
    void NoItems;
    a One;
    Tuple2#(UAList#(a),UAList#(a)) Append;
} UAList#(type a);
```

`UAList` is a member of the `DefaultValue` typeclass, which defines a default value for user defined structures. The default value for `UAList` is defined as:

```
instance DefaultValue#(UAList#(a));
   defaultValue = NoItems;
endinstance
```

Functions

`flatten0`	Given a `UAList#(a)` and a `List#(a)`, returns a conventional list of type `List`.
	`function List#(a) flatten0(UAList#(a) c, List#(a) xs);`

`flatten`	Converts a list of type `UAList` into a conventional list of type `List`.
	`function List#(a) flatten(UAList#(a) c) = flatten0(c, Nil);`

`uaMap`	Maps a function of a list of type `UAList`, returning a `UAList`.
	`function UAList#(b) uaMap(function b f(a x), UAList#(a) c);`

`uaMapM`	Maps a monadic function of a list of type `UAList`, returning a `UAList`.
	`module uaMapM#(function module#(b) f(a x), UAList#(a) c)(UAList#(b));`

Index

<< (Bitwise class method), 18
>> (Bitwise class method), 18
| (BitReduction class method), 18
| (Bitwise class method), 17
π (Real constant), 160
&& (Bool operator), 27
* (Arith class method), 12
** (Arith class method), 12
+ (String concatenation operator), 29
+ (Arith class method), 12
- (Arith class method), 12
/ (Arith class div method), 12
/= (Eq class method), 10
< (Ord class method), 14
<= (Ord class method), 14
== (Eq class method), 10
> (Ord class method), 14
>= (Ord class method), 14
$bitstoreal (Real system function), 29
$realtobits (Real system function), 29
%(Arith class mod method), 12
& (BitReduction class method), 18
& (Bitwise class method), 17
_read (PulseWire interface method), 54
_read (Reg interface method), 45
_write (Reg interface method), 45
{} (Bit concatenation operator), 25
^~ (BitReduction class method), 18
^ (BitReduction class method), 18
^ (Bitwise class method), 17
^& (BitReduction class method), 18
^~ (Bitwise class method), 17
~^ (BitReduction class method), 18
~ (Bitwise class method), 17
~| (BitReduction class method), 18
~^ (Bitwise class method), 17

abs (function), 59
abs (Arith class method), 12
acosh (Real function), 162
Action (type), 41
ActionValue (type), 41
Add (type provisos), 43
addBIUInt (function), 177
addRules (Rules function), 42
addToCollection (ModuleCollect function), 284
addUInt (function), 177
Alias, 22
AlignedFIFOs (package), 109
all (List function), 150
all (Vector function), 126
and (List function), 151
and (Vector function), 127
any (List function), 150
any (Vector function), 126
append (List function), 143
append (Vector function), 119
applyToContext (module), 278
applyToContextM (module), 278
Arbiter (package), 207
Arith (type class), 12
Array (type), 37
  example, 49
array
  named type, 37
    example, 49
arrayToVector (Vector function), 141
asinh (Real function), 162
asReg (Reg function), 45
Assert (package), 222
AssertFifoTest_IFC (interface), 234
AssertQuiescentTest_IFC (interface), 233
AssertSampleTest_IFC (interface), 232
AssertStartStopTest_IFC (interface), 233
AssertStartTest_IFC (interface), 232
AssertTest_IFC (interface), 232
AssertTransitionTest_IFC (interface), 233
atan2 (Real function), 162
atanh (Real function), 162
await (StmtFSM function), 182

Bit (type), 24
bit (type), 25
bitconcat (Bit concatenation operator), 25
BitExtend (type class), 20
BitReduction (type class), 18
Bits (type class), 9
Bitwise (type class), 17
Bool (type), 27
Bounded (type class), 16
BRAM (interface type), 81
BRAM (package), 79
BRAM_Configure (type), 79
BRAM_DUAL_PORT (interface type), 85
BRAM_PORT (interface type), 85
BRAMCore (package, 85
BRAMFIFO (package), 105
BRAMRequest (type), 80
bsv_assert_always (module), 236

bsv_assert_always_on_edge (module), 236
bsv_assert_change (module), 236
bsv_assert_cycle_sequence (module), 236
bsv_assert_decrement (module), 237
bsv_assert_delta (module), 237
bsv_assert_even_parity (module), 237
bsv_assert_fifo_index (module), 237
bsv_assert_frame (module), 237
bsv_assert_handshake (module), 238
bsv_assert_implication (module), 238
bsv_assert_increment (module), 238
bsv_assert_never (module), 238
bsv_assert_never_unknown (module), 239
bsv_assert_never_unknown_async (module), 239
bsv_assert_next (module), 239
bsv_assert_no_overflow (module), 239
bsv_assert_no_transition (module), 240
bsv_assert_no_underflow (module), 240
bsv_assert_odd_parity (module), 240
bsv_assert_one_cold (module), 240
bsv_assert_one_hot (module), 241
bsv_assert_proposition (module), 241
bsv_assert_quiescent_state (module), 241
bsv_assert_range (module), 241
bsv_assert_time (module), 242
bsv_assert_transition (module), 242
bsv_assert_unchange (module), 242
bsv_assert_width (module), 242
bsv_assert_win_change (module), 243
bsv_assert_win_unchange (module), 243
bsv_assert_window (module), 243
bsv_assert_zero_one_hot (module), 243
BufferMode (type), 66
BuildVector (package), 246
buildVersion, 64
BypassWire (interface), 53

CBus (interface), 286
CBus (package), 286
ceil (Real function), 163
CGetPut (package), **197**
Char (type), 31
Client (interface), 194
ClientServer (package), 194
Clock (type), 39, 247
ClockDividerIfc (interface), 255
clockOf (function), 248
Clocks (package), 247
cmplx (complex function), 166
cmplxConj (complex function), 167
cmplxMap (complex function), 167
cmplxSwap (complex function), 167
cmplxWrite (complex function), 167
Cntrs (package), 209
collectCBusIFC (module), 287
CommitIfc (package), 199
compare (Ord class method), 14
compilerVersion, 63
CompletionBuffer (interface), 214
CompletionBuffer (package), 213
Complex (package), 165
compose (function), 60
composeM (function), 60
concat (List function), 143
concat (Vector function), 119
ConfigReg (package,interface), 76
Connectable (class), 193
Connectable (package), 192
Cons (List constructor), 142
cons (List function), 143
cons (Vector function), 118
constFn (function), 60
continuousAssert, 223
Control (interface), 206
cos (Real function), 161
cosh (Real function), 161
Count (interface), 209
countElem (Vector function), 127
countIf (Vector function), 127
countOnes (function), 61
countOnesAlt (bit-vector function), 129
countZerosLSB (function), 61
countZerosMSB (function), 61
CRC (package), 230
curry (function), 60

date, 64
decodeReal (Real function), 164
DefaultValue (package), 220
delay (StmtFSM function), 182
Div (type provisos), 43
div (Integer function), 26
DReg (package,interface), 77
drop (List function), 145
drop (Vector function), 122
dropWhile (List function), 146
dropWhileRev (List function), 146
DualPortRamIfc (interface), 269
DWire (interface), 53
dynamicAssert, 222

elem (List function), 150
elem (Vector function), 126
emptyRules (Rules variable), 42
epochTime, 64
epsilon (FixedPoint function), 169
Eq (type class), 10

error (forced error), 58
errorM (forced error), 58
exp_e (Arith class method), 12
exposeCBusIFC (module), 287
exposeCollection (ModuleCollect function), 284
exposeCurrentClock (function), 248
exposeCurrentReset (function), 248
extend (BitExtend class method), 20

False (Bool constant), 27
FIFO (package), 90
FIFOCountIfc (interface), 99
FIFOF (package), 90
fifofToFifo (function), 95
FIFOLevel (package), 98
FIFOLevelIfc (interface), 98
fifoToGet (GetPut function), 191
fifoToPut (GetPut function), 191
File (type), 38
select (filter function), 145
find (List function), 146
find (Vector function), 127
findElem (Vector function), 127
findIndex (Vector function), 128
FixedPoint (package), 168
flatten (function), 297
flatten0 (function), 297
flip (function), 60
floor (Real function), 163
Fmt (type), 33
fold (List function), 155
fold (Vector function), 133
foldl (List function), 155
foldl (Vector function), 133
foldl1 (List function), 155
foldl1 (Vector function), 133
foldr (List function), 154
foldr (Vector function), 133
foldr1 (List function), 155
foldr1 (Vector function), 133
fromInt (FixedPoint function), 170
fromInteger (Literal class method), 11, 27
fromMaybe (Maybe function), 34
fromReal (RealLiteral class method), 11
fromSizedInteger (SizedLiteral class method), 12
fromUInt (FixedPoint function), 170
FShow (type class), 22
fxptAdd (FixedPoint function), 171
fxptGetFrac (FixedPoint function), 170
fxptGetInt (FixedPoint function), 170
fxptMult (FixedPoint function), 171
fxptQuot (FixedPoint function), 172
fxptSignExtend (FixedPoint function), 174
fxptSub (FixedPoint function), 171
fxptTruncate (FixedPoint function), 172
fxptTruncateRoundSat (FixedPoint function), 172
fxptTruncateSat (FixedPoint function), 172
fxptWrite (FixedPoint function), 175
fxptZeroExtend (FixedPoint function), 174

GatedClockIfc (interface), 250
gcd (function), 62
Gearbox (interface), 113
Gearbox (package), 112
genC, 63
Generic, 67
genModuleName, 63
genPackageName, 63
genVector (Vector function), 118
genVerilog, 63
genWith (Vector function), 118
genWithM (Vector function), 139
Get (interface), 187
GetPut (package), 187
Gettable (typeclass), 294
Gray (package), 212
GrayCounter (package), 211
grayDecode (function), 213
grayDecr (function), 213
grayEncode (function), 213
grayIncr (function), 213
grayIncrDecr (function), 213
group (List function), 149
groupBy (List function), 149

HAppend (typeclass), 294
hClose, 65
head (List function), 144
head (Vector function), 121
hFlush, 66
hGetBuffering, 66
hGetChar, 66
hGetLine, 67
HHead (typeclass), 293
hIsClosed, 65
hIsEOF, 65
hIsOpen, 65
hIsReadable, 65
hIsWriteable, 65
HLength (typeclass), 293
HList (package), 292
hPutChar, 66
hPutStr, 66
hPutStrLn, 66
hSetBuffering, 66

HSplit (typeclass), 294
HTail (typeclass), 293

id (function), 60
init (List function), 145
init (Vector function), 121
Inout (type), 39
Int (type), 26
int (type), 26
Integer (type), 26
Invalid (type constructor), 34
invert (Bitwise class method), 17
invertCurrentClock (function), 249
invertCurrentReset (function), 249
isAncestor (function), 248
isInfinite (Real function), 163
isNegativeZero (Real function), 163
isResetAsserted (module), 277
isValid (Maybe function), 34
IWithCBus (interface), 286

joinActions (List function), 155
joinActions (Vector function), 134
joinRules (List function), 156
joinRules (Vector function), 134

last (List function), 145
last (Vector function), 121
lcm (function), 62
length (List function), 150
LevelFIFO, *see* FIFOLevel
LFSR (package), 203
List (type), 142
ListN (type), 141
Literal (type class), 11
Log (type provisos), 43
log (Arith class method), 12
log10 (Arith class method), 12
log2 (Arith class method), 12
logb (Arith class method), 12
lookup (List function), 146
loop statements
    temporal, in FSMs, 179
lsb (Bitwise class method), 18
LUInt (type), 114

MakeClockIfc (interface), 249
MakeResetIfc (interface), 272
map (List function), 152
map (Vector function), 131
mapAccumL (List function), 158
mapAccumL (Vector function), 136
mapAccumR (List function), 158
mapAccumR (Vector function), 137
mapCollection (ModuleCollect function), 284
mapM (Monad function on List), 159
mapM (Monad function on Vector), 137
mapM_ (List function), 159
mapM_ (Vector function), 138
mapPairs (List function), 156
mapPairs (Vector function), 134
Max (type provisos), 43
max (Ord class method), 14
max (function), 59
maxBound (Bounded class method), 16
Maybe (type), 34
Memory (package), 196
message (compilation message), 58
messageM (compilation message), 59
MIMO (interface), 115
MIMO (package), 114
MIMOConfiguration (type), 114
Min (type provisos), 43
min (Ord class method), 14
min (function), 59
minBound (Bounded class method), 16
mk1toNGearbox (module), 114
mkAbsoluteClock (module), 252
mkAbsoluteClockFull (module), 252
mkAlignedFIFO (module), 112
mkArbiter (module), 208
mkAsyncReset (module), 273
mkAsyncResetFromCR (module), 273
mkAutoFSM, 181
mkBRAM1Server (module), 83
mkBRAM1ServerBE (module), 83
mkBRAM2Server (module), 83
mkBRAMCore1 (module), 87
mkBRAMCore1BE (module), 87
mkBRAMCore1BELoad (module), 87
mkBRAMCore1Load (module), 87
mkBRAMCore2 (module), 88
mkBRAMCore2BE (module), 88
mkBRAMCore2BELoad (module), 89
mkBRAMCore2Load (module), 88
mkBRAMStore1W2R (module), 111
mkBRAMStore2W1R (module), 111
mkBypassFIFO (module), 108
mkBypassFIFOF (module), 108
mkBypassFIFOLevel (module), 109
mkBypassWire (module), 53
mkCBRegFile (module), 288
mkCBRegR (module), 288
mkCBRegRC (module), 288
mkCBRegRW (module), 288
mkCBRegW (module), 288
mkCClientServer (function), 199
mkCGetPut (function), 198
mkClientCServer (function), 199

mkClock (module), 250
mkClockDivider (module), 255
mkClockDividerOffset (module), 255
mkClockInverter (module), 255
mkClockMux (module), 254
mkClockSelect (module), 254
mkCompletionBuffer (module), 214
mkConfigReg (module), 76
mkConfigRegA (module), 76
mkConfigRegU (module), 76
mkConstrainedRandomizer (module), 206
mkCount (module), 210
mkCRC(module), 230
mkCRC16(module), 231
mkCRC32(module), 231
mkCRC_CCIT(module), 231
mkCReg (module), 47
mkCRegA (module), 47
mkCRegU (module), 47
mkDepthParamFIFO (module), 93
mkDepthParamFIFOF (module), 93
mkDFIFOF (module), 109
mkDReg (module), 77
mkDRegA (module), 77
mkDRegU (module), 77
mkDualRam (module), 269
mkDWire (module), 53
mkFeedLFSR(module), 204
mkFIFO (module), 92
mkFIFO1 (module), 93
mkFIFOCount (module), 102
mkFIFOF (module), 92
mkFIFOF1 (module), 93
mkFIFOLevel (module), 101
mkFSM, 181
mkFSMServer, 185
mkFSMWithPred, 181
mkGatedClock (module), 251
mkGatedClockDivider (module), 255
mkGateClockFromCC (module), 251
mkGatedClockInverter (module), 255
mkGDepthParamFIFOF (module), 94
mkGenericRandomizer (module), 206
mkGetCPut (function), 198
mkGFIFOCount (module), 102
mkGFIFOF (module), 94
mkGFIFOF1 (module), 94
mkGFIFOLevel (module), 102
mkGLFIFOF (module), 95
mkGPFIFO (GetPut module), 190
mkGPFIFO1 (GetPut module), 191
mkGPSizedFIFO (GetPut module), 191
mkGrayCounter (module), 211
mkGSizedFIFOF (module), 94
mkInitialReset (module), 274
mkLFIFO (module), 94
mkLFIFOF (module), 94
mkMIMO (module), 116
mkMIMOBRAM (module), 116
mkMIMOReg (module), 116
mkMIMOV (module), 116
mkNto1Gearbox (module), 113
mkNullCrossingReg (module), 271
mkNullCrossingRegA (module), 271
mkNullCrossingRegU (module), 271
mkNullCrossingWire (module), 270
mkOnce, 181
mkPipelineFIFO (module), 108
mkPipelineFIFOF (module), 108
mkPulseWire (module), 54
mkPulseWireOR (module), 54
mkReg (module), 45
mkRegA (module), 45
mkRegFile (RegFile module), 73
mkRegFileFull (RegFile module), 73
mkRegFileFullFile (RegFileLoad function), 74
mkRegFileLoad (RegFileLoad function), 74
mkRegStore (module), 111
mkRegU (module), 45
mkRegVectorStore (module), 111
mkReset (module), 274
mkResetEither (module), 276
mkResetInverter (module), 277
mkResetMux (module), 275
mkResetSync (module), 274
mkRevertingVirtualReg (module), 77
mkRWire (RWire module), 50
mkRWireSBR (RWire module), 50
mkSizedBRAMFIFO (module), 106
mkSizedBRAMFIFOF (module), 106
mkSizedBypassFIFOF (module), 108
mkSizedFIFO (module), 93
mkSizedFIFOF (module), 93
mkStickyArbiter (module), 208
mkSyncBit (module), 258
mkSyncBit05 (module), 261
mkSyncBit05FromCC (module), 261
mkSyncBit05ToCC (module), 261
mkSyncBit1 (module), 260
mkSyncBit15 (module), 259
mkSyncBit15FromCC (module), 259
mkSyncBit15ToCC (module), 260
mkSyncBit1FromCC (module), 260
mkSyncBit1ToCC (module), 260
mkSyncBitFromCC (module), 259
mkSyncBitToCC (module), 259
mkSyncBRAM2Server (module), 84

mkSyncBRAM2ServerBE (module), 84
mkSyncBRAMCore2 (module), 88
mkSyncBRAMCore2BE (module), 88
mkSyncBRAMCore2BELoad (module), 89
mkSyncBRAMCore2Load (module), 89
mkSyncBRAMFIFO (module), 106
mkSyncBRAMFIFOFromCC (module), 106
mkSyncBRAMFIFOToCC (module), 106
mkSyncFIFO (module), 267
mkSyncFIFO1 (module), 268
mkSyncFIFOCount (module), 103
mkSyncFIFOFromCC (module), 268
mkSyncFIFOLevel (module), 102
mkSyncFIFOToCC (module), 268
mkSyncHandshake (module), 264
mkSyncHandshakeFromCC (module), 264
mkSyncHandshakeToCC (module), 264
mkSyncPulse (module), 263
mkSyncPulseFromCC (module), 263
mkSyncPulseToCC (module), 263
mkSyncReg (module), 265
mkSyncRegFromCC (module), 266
mkSyncRegToCC (module), 266
mkSyncReset (module), 273
mkSyncResetFromCR (module), 273
mkTriState, 227
mkUCount (module), 210
mkUGDepthParamFIFOF (module), 94
mkUGFIFOF (module), 93
mkUGFIFOF1 (module), 93
mkUGLFIFOF (module), 94
mkUGSizedFIFOF (module), 93
mkUngatedClock (module), 251
mkUngatedClockMux (module), 254
mkUngatedClockSelect (module), 254
mkUniqueWrappers (UniqueWrappers module), 216
mkUnsafeDWire (module), 53
mkUnsafePulseWire (module), 54
mkUnsafePulseWireOR (module), 54
mkUnsafeRWire (RWire module), 50
mkUnsafeWire (module), 52
mkWire (module), 52
mkZBus (function), 229
mkZBusBuffer (function), 229
mod (Integer function), 26
ModuleCollect (package), 282
ModuleCollect (type), 283
ModuleContext (package), 277
msb (Bitwise class method), 18
Mul (type provisos), 43
MuxClockIfc (interface), 253
MuxRstIfc (interface), 272

negate (Arith class method), 12
newVector (Vector function), 118
Nil (List constructor), 142
nil (Vector function), 118
noAction (empty action), 41
noClock (function), 249
noReset (function), 249
not (Bool function), 27
NumAlias, 22
NumberTypes (package), 175

OInt (package), 164
OInt (type), 164
oneHotSelect (List function), 144
openFile, 64
or (List function), 150
or (Vector function), 127
Ord (type class), 14
Ordering (type), 38
OVLAssertions (package), 232

pack (Bits type class overloaded function), 9
parity (function), 61
pi (Real constant), 160
pow (Real function), 162
Printf (package), 245
Probe (package), 223
provisos, 43
PulseWire (interface), 54
pulseWireToReadOnly (function), 57
Put (interface), 187

quot (Integer function), 26

Randomizable (package), 205
Randomize (interface), 206
ReadOnly (interface), 56
readReadOnly (function), 57
readReg (Reg function), 45
readVReg (Vector function), 129
Real (package), 160
Real (type), 28
real (type), 28
RealLiteral (type class), 11
realToDigits (Real function), 164
reburyContext (module), 280
reduceAnd (BitReduction class method), 18
reduceNand (BitReduction class method), 18
reduceNor (BitReduction class method), 18
reduceOr (BitReduction class method), 18
reduceXnor (BitReduction class method), 18
reduceXor (BitReduction class method), 18
reflect(CRC function), 231
Reg (interface), 265
Reg (type), 45

RegFile (interface type), 73
RegFileLoad (package), 74
regToReadOnly (function), 56
rem (Integer function), 26
replicate (List function), 143
replicate (Vector function), 118
replicateM (List function), 160
replicateM (Vector function), 139
Reserved (type), 224
Reserved (package), 224
ReservedOne (type), 224
ReservedZero (type), 224
Reset (type), 39, 247
clear, 177
resetOf (function), 249
reverse (List function), 148
reverse (Vector function), 125
reverseBits (function), 61
RevertingVirtualReg (package), 77
rJoin (Rules operator), 42
rJoinConflictFree (Rules operator), 42
rJoinDescendingUrgency (Rules operator), 42
rJoinExecutionOrder (Rules operator), 42
rJoinMutuallyExclusive (Rules operator), 42
rJoinPreempts (Rules operator), 42
rotate (List function), 148
rotate (Vector function), 123
rotateBitsBy (bit-vector function), 129
rotateBy (Vector function), 124
rotateR (List function), 148
rotateR (Vector function), 124
round (Real function), 163
Rules (type), 42
runWithContext (function), 281
runWithContexts (function), 281
RWire, 50

sameFamily (function), 248
satMinus (SaturatingArith class method), 21
satPlus (SaturatingArith class method), 21
SaturatingArith (type class), 21
sbtrctBIUInt (function), 177
scanl (List function), 158
scanl (Vector function), 136
scanr (List function), 157
scanr (Vector function), 135
select (List function), 144
select (Vector function), 120
SelectClockIfc (interface), 253
send (PulseWire interface method), 54
Server (interface), 195
shiftInAt0 (Vector function), 124
shiftInAtN (Vector function), 124
shiftOutFrom0 (Vector function), 124
shiftOutFromN (Vector function), 125
signedMul (function), 59
signedQuot (function), 59
signExtend (BitExtend class method), 20
signum (Arith class method), 12
sin (Real function), 161
sinh (Real function), 161
SizedLiteral (type class), 12
SizeOf (pseudo-function on types), 44
sort (List function), 148
sortBy (List function), 148
SpecialFIFOs (package), 107
split (Bit function), 25
splitReal (Real function), 163
sprintf (function), 245
sqrt (Real function), 162
sscanl (List function), 158
sscanl (Vector function), 136
sscanr (List function), 157
sscanr (Vector function), 135
Standard Prelude, 9
start, 177
staticAssert, 222
StmtFSM (package), 177
strConcat (String concatenation operator), 29
String (type), 29
StringLiteral (type class), 24
sub (RegFile interface method), 73
SyncBitIfc (interface), 258
SyncFIFOCountIfc (interface), 101
SyncFIFOIfc (interface), 266
SyncFIFOLevelIfc (interface), 100
SyncPulseIfc (interface), 262

TAdd (type functions), 44
tail (List function), 145
tail (Vector function), 121
take (List function), 145
take (Vector function), 121
takeAt (Vector function), 122
takeWhile (List function), 146
takeWhileRev (List function), 146
tan (Real function), 161
tanh (Real function), 161
TDiv (type functions), 44
TExp (type functions), 44
TieOff (package), 222
TLog (type functions), 44
TMax (type functions), 44
TMin (type functions), 44

TMul (type functions), 44
toChunks (Vector function), 141
toGet (function), 187
toGPClient (function), 195
toGPServer (function), 196
toList (Vector function), 141
toPut (function), 188
toVector (Vector function), 141
transpose (List function), 148
transpose (Vector function), 125
transposeLN (Vector function), 125
TriState (interface), 226
TriState (package), 226
True (Bool constant), 27
trunc (Real function), 162
truncate (BitExtend class method), 20
truncateLSB (function), 62
TSub (type functions), 44
tuples
    expressions, 36
    selecting components, 37
    type definition, 35
type classes, 9

uaMap (function), 297
uaMapM (function), 297
UCount (interface), 209
UInt (type), 26
unburyContext (module), 279
unburyContextWithClocks (module), 280
uncurry (function), 61
UnitAppendList (package), 296
unpack (Bits type class overloaded function), 9
unsignedMul (function), 59
unsignedQuot (function), 60
unwrap (function), 177
unwrapBI (function), 176
unzip (List function), 152
unzip (Vector function), 130
upd (RegFile interface method), 73
update (List function), 144
update (Vector function), 120
updateDataWithMask (module), 197
upto (List function), 143

Valid (type constructor), 34
valueOf (pseudo-function of size types), 44
valueof (pseudo-function of size types), 44
Vector, 116
vectorToArray (Vector function), 141
Void (type), 34

warning (forced warning), 58
warningM (forced warning), 58
wget (RWire interface method), 50
when (function), 62
while (function), 62
Wire (interface), 51
wrap (function), 177
Wrapper (interface type), 216
WriteOnly (interface), 57
writeReg (Reg function), 45
writeVReg (Vector function), 129
wset (RWire interface method), 50

ZBus (package), 227
ZBusBusIFC (interface), 228
ZBusClientIFC (interface), 228
ZBusDualIFC (interface), 228
zeroExtend (BitExtend class method), 20
zip (List function), 151
zip (Vector function), 129
zip3 (List function), 151
zip3 (Vector function), 130
zip4 (List function), 152
zip4 (Vector function), 130
zipAny (Vector function), 130
zipWith (List function), 153
zipWith (Vector function), 131
zipWith3 (List function), 153
zipWith3 (Vector function), 132
zipWith3M (List function), 160
zipWith3M (Vector function), 138
zipWith4 (List function), 153
zipWithAny (Vector function), 131
zipWithAny3 (Vector function), 132
zipWithM (List function), 159
zipWithM (Vector function), 138

Function and Module by Package

AlignedFIFOs
- mkAlignedFIFO, 112
- mkBRAMStore1W2R, 111
- mkBRAMStore2W1R, 111
- mkRegStore, 111
- mkRegVectorStore, 111

BRAM
- mkBRAM1Server, 83
- mkBRAM1ServerBE, 83
- mkBRAM2Server, 83
- mkBRAMCore1, 87
- mkBRAMCore1BE, 87
- mkBRAMCore1BELoad, 87
- mkBRAMCore1Load, 87
- mkBRAMCore2, 88
- mkBRAMCore2BE, 88
- mkBRAMCore2BELoad, 89
- mkBRAMCore2Load, 88
- mkSyncBRAM2Server, 84
- mkSyncBRAM2ServerBE, 84
- mkSyncBRAMCore2, 88
- mkSyncBRAMCore2BE, 88
- mkSyncBRAMCore2BELoad, 89
- mkSyncBRAMCore2Load, 89

BRAMFIFO
- mkSizedBRAMFIFO, 106
- mkSizedBRAMFIFOF, 106
- mkSyncBRAMFIFO, 106
- mkSyncBRAMFIFOFromCC, 106
- mkSyncBRAMFIFOToCC, 106

CBus
- collectCBusIFC, 287
- exposeCBusIFC, 287
- mkCBRegFile, 288
- mkCBRegR, 288
- mkCBRegRC, 288
- mkCBRegRW, 288
- mkCBRegW, 288

ClientSefiforver
- toGPServer, 196

ClientServer
- toGPClient, 195

Clocks
- clockOf, 248
- exposeCurrentClock, 248
- exposeCurrentReset, 248
- invertCurrentClock, 249
- invertCurrentReset, 249
- isAncestor, 248
- isResetAsserted, 277
- mkAbsoluteClock, 252
- mkAbsoluteClockFull, 252
- mkAsyncReset, 273
- mkAsyncResetFromCR, 273
- mkClock, 250
- mkClockDivider, 255
- mkClockDividerOffset, 255
- mkClockInverter, 255
- mkClockMux, 254
- mkClockSelect, 254
- mkDualRam, 269
- mkGatedClock, 251
- mkGatedClockDivider, 255
- mkGatedClockFromCC, 251
- mkGatedClockInverter, 255
- mkInitialReset, 274
- mkNullCrossingReg, 271
- mkNullCrossingRegA, 271
- mkNullCrossingRegU, 271
- mkNullCrossingWire, 270
- mkReset, 274
- mkResetEither, 276
- mkResetInverter, 277
- mkResetMux, 275
- mkResetSync, 274
- mkSyncBit, 258
- mkSyncBit05, 261
- mkSyncBit05FromCC, 261
- mkSyncBit05ToCC, 261
- mkSyncBit1, 260
- mkSyncBit15, 259
- mkSyncBit15FromCC, 259
- mkSyncBit15ToCC, 260
- mkSyncBit1FromCC, 260
- mkSyncBit1ToCC, 260
- mkSyncBitFromCC, 259
- mkSyncBitToCC, 259
- mkSyncFIFO, 267
- mkSyncFIFO1, 268
- mkSyncFIFOFromCC, 268
- mkSyncFIFOToCC, 268
- mkSyncHandshake, 264
- mkSyncHandshakeFromCC, 264
- mkSyncHandshakeToCC, 264
- mkSyncPulse, 263
- mkSyncPulseFromCC, 263
- mkSyncPulseToCC, 263
- mkSyncReg, 265

mkSyncRegFromCC, 266
mkSyncRegToCC, 266
mkSyncReset, 273
mkSyncResetFromCR, 273
mkUngatedClock, 251
mkUngatedClockMux, 254
mkUngatedClockSelect, 254
noClock, 249
noReset, 249
resetOf, 249
sameFamily, 248

Cntrs
mkCount, 210
mkUCount, 210

FIFO
fifofToFifo, 95
mkDepthParamFIFO, 93
mkFIFO, 92
mkFIFO1, 93
mkLFIFO, 94
mkSizedFIFO, 93

FIFOF
mkDepthParamFIFOF, 93
mkFIFOF, 92
mkFIFOF1, 93
mkGDepthParamFIFOF, 94
mkGFIFOF, 94
mkGFIFOF1, 94
mkGLFIFOF, 95
mkGSizedFIFOF, 94
mkLFIFOF, 94
mkSizedFIFOF, 93
mkUGDepthParamFIFOF, 94
mkUGFIFO1, 93
mkUGFIFOF, 93
mkUGLFIFOF, 94
mkUGSizedFIFOF, 93

FIFOLevel
mkFIFOCount, 102
mkFIFOLevel, 101
mkGFIFOCount, 102
mkGFIFOLevel, 102
mkSyncFIFOCount, 103
mkSyncFIFOLevel, 102

FixedPoint
fxptTruncateSat, 172

Gearbox
mk1toNGearbox, 114
mkNto1Gearbox, 113

GetPut
fifoToGet, 191
fifoToPut, 191
mkGPFIFO, 190
mkGPFIFO1, 191
mkGPSizedFIFO, 191
toGet, 187
toPut, 188

Gray
grayDecode, 213
grayDecr, 213
grayEncode, 213
grayIncr, 213
grayIncrDecr, 213

GrayCounter
mkGrayCounter, 211

HList
Gettable, 294
HAppend, 294
HHead, 293
HLength, 293
hSplit, 294
HTail, 293

List
all, 150
and, 151
any, 150
append, 143
concat, 143
cons, 143
drop, 145
dropWhile, 146
dropWhileRev, 146
elem, 150
filter, 145
find, 146
fold, 155
foldl, 155
foldl1, 155
foldr, 154
foldr1, 155
group, 149
groupBy, 149
head, 144
init, 145
joinActions, 155
joinRules, 156
last, 145
length, 150
lookup, 146
map, 152
mapAccumL, 158
mapAccumR, 158
mapM, 159
mapM_, 159

mapPairs, 156
oneHotSelect, 144
or, 150
replicate, 143
replicateM, 160
reverse, 148
rotate, 148
rotateR, 148
scanl, 158
scanr, 157
select, 144
sort, 148
sortBy, 148
sscanl, 158
sscanr, 157
tail, 145
take, 145
takeWhile, 146
takeWhileRev, 146
transpose, 148
unzip, 152
update, 144
upto, 143
zip, 151
zip3, 151
zip4, 152
zipWith, 153
zipWith3, 153
zipWith3M, 160
zipWith4, 153
zipWithM, 159

Memory
updateDataWithMask, 197
MIMO
mkMIMO, 116
mkMIMOBRAM, 116
mkMIMOReg, 116
mkMIMOV, 116
ModuleContext
applyToContext, 278
applyToContextM, 278
reburyContext, 280
runWithContext, 281
unburyContext, 279
unburyContextWithClocks, 280

NumberTypes
addBIUInt, 177
addUInt, 177
sbtrctBIUInt, 177
unwrap, 177
unwrapBI, 176
wrap, 177

OVLAssertions
bsv_assert_always, 236
bsv_assert_always_on_edge, 236
bsv_assert_change, 236
bsv_assert_cycle_sequence, 236
bsv_assert_decrement, 237
bsv_assert_delta, 237
bsv_assert_even_parity, 237
bsv_assert_fifo_index, 237
bsv_assert_frame, 237
bsv_assert_handshake, 238
bsv_assert_implication, 238
bsv_assert_increment, 238
bsv_assert_never, 238
bsv_assert_never_unknown, 239
bsv_assert_never_unknown_async, 239
bsv_assert_next, 239
bsv_assert_no_overflow, 239
bsv_assert_no_transition, 240
bsv_assert_no_underflow, 240
bsv_assert_odd_parity, 240
bsv_assert_one_cold, 240
bsv_assert_one_hot, 241
bsv_assert_proposition, 241
bsv_assert_quiescent_state, 241
bsv_assert_range, 241
bsv_assert_time, 242
bsv_assert_transition, 242
bsv_assert_unchange, 242
bsv_assert_width, 242
bsv_assert_win_change, 243
bsv_assert_win_unchange, 243
bsv_assert_window, 243
bsv_assert_zero_one_hot, 243

Prelude
!=, 10
<<, 18
>>, 18
|, 17, 18
*, 12
**, 12
+, 12, 29
-, 12
/, 12
<, 14
<=, 14
==, 10
>, 14
>=, 14
$bitstoreal, 29
$realtobits, 29
%, 12
&, 17, 18

^, 17, 18
^~, 17, 18
~, 17, 18
~|, 18
~^, 17, 18
abs, 12, 59
addRules, 42
asReg, 45
bitsToDigit, 31
bitsToHexDigit, 31
buildVersion, 64
charToInteger, 31
charToString, 31
compare, 14
compilerVersion, 63
compose, 60
composeM, 60
constFn, 60
countOnes, 61
countZerosLSB, 61
countZerosMSB, 61
curry, 60, 61
date, 64
digitToBits, 31
digitToInteger, 31
div, 26
doubleQuote, 29
epochTime, 64
error, 58
errorM, 58
exp, 12
extend, 20
flip, 60
from, 67
fromInteger, 11, 27
fromMaybe, 34
fromReal, 11
fromSizedInteger, 12
fromString, 24
fshow, 22
gcd, 62
genC, 63
genModuleName, 63
genPackageName, 63
genVerilog, 63
hClose, 65
hexDigitToBits, 31
hexDigitToInteger, 31
hFlush, 66
hGetBuffering, 66
hGetChar, 66
hGetLine, 67
hIsClosed, 65
hIsEOF, 65
hIsOpen, 65
hIsReadable, 65
hIsWriteable, 65
hPutChar, 66
hPutStr, 66
hPutStrLn, 66
hSetBuffering, 66
id, 60
integerToChar, 31
integerToDigit, 31
integerToHexDigit, 31
invert, 17
isAlpha, 31
isAlphaNum, 31
isDigit, 31
isHexDigit, 31
isLower, 31
isOctDigit, 31
isSpace, 31
isUpper, 31
isValid, 34
lcm, 62
log, 12
log10, 12
log2, 12
logb, 12
lsb, 18
max, 14, 59
maxBound, 16
message, 58
messageM, 59
min, 14, 59
minBound, 16
mkBypassWire, 53
mkCReg, 47
mkCRegA, 47
mkCRegU, 47
mkDWire, 53
mkPulseWire, 54
mkPulseWireOR, 54
mkReg, 45
mkRegA, 45
mkRegU, 45
mkRWire, 50
mkRWireSBR, 50
mkUnsafeDWire, 53
mkUnsafePulseWire, 54
mkUnsafePulseWireOR, 54
mkUnsafeRWire, 50
mkUnsafeWire, 52
mkWire, 52
mod, 26
msb, 18
negate, 12

not, 27
openFile, 64
pack, 9
parity, 61
pulseWireToReadOnly, 57
quot, 26
quote, 29
readReadOnly, 57
readReg, 45
reduceAnd, 18
reduceNand, 18
reduceNor, 18
reduceOr, 18
reduceXNor, 18
reduceXor, 18
regToReadOnly, 56
rem, 26
reverseBits, 61
rJoin, 42
rJoinConflictFree, 42
rJoinDescendingUrgency, 42
rJoinExecutionOrder, 42
rJoinMutuallyExclusive, 42
rJoinPreempts, 42
satMinus, 21
satPlus, 21
sharListToString, 29
signedMul, 59
signedQuot, 59
signExtend, 20
signum, 12
SizeOf, 44
split, 25
strConcat, 29
stringCons, 29
stringHead, 29
stringLength, 29
stringSplit, 29
stringTail, 29
stringToCharList, 29
TAdd, 44
TDiv, 44
TExp, 44
TLog, 44
TMax, 44
TMin, 44
TMul, 44
to, 67
toLower, 31
toUpper, 31
truncate, 20
truncateLSB, 62
TSub, 44
unpack, 9
unsignedMul, 59
unsignedQuot, 60
valueOf, 44
warning, 58
warningM, 58
when, 62
while, 62
writeReg, 45
zeroExtend, 20

Printf
sprintf, 245

Real
acosh, 162
asinh, 162
atan2, 162
atanh, 162
ceil, 163
cos, 161
cosh, 161
decodeReal, 164
floor, 163
isInfinite, 163
isNegativeZero, 163
pow, 162
realToDigits, 164
round, 163
sin, 161
sinh, 161
splitReal, 163
sqrt, 162
tan, 161
tanh, 161
trunc, 162

SpecialFIFOs
mkBypassFIFO, 108
mkBypassFIFOF, 108
mkBypassFIFOLevel, 109
mkDFIFOF, 109
mkPipelineFIFO, 108
mkPipelineFIFOF, 108
mkSizedBypassFIFOF, 108

StmtFSM
await, 182
callServer, 185
delay, 182
mkAutoFSM, 181
mkFSM, 181
mkFSMServer, 185
mkFSMwithPred, 181
mkOnce, 181

UnitAppendList
flatten, 297

flatten0, 297
uaMap, 297
uaMapM, 297

Vector, 141
all, 126
and, 127
any, 126
append, 119
arrayToVector, 141
concat, 119
cons, 118
countElem, 127
countIf, 127
countOnesAlt, 129
drop, 122
elem, 126
find, 127
findElem, 127
findIndex, 128
fold, 133
foldl, 133
foldl1, 133
foldr, 133
foldr1, 133
genVector, 118
genWith, 118
genWithM, 139
head, 121
init, 121
joinActions, 134
joinRules, 134
last, 121
map, 131
mapAccumL, 136
mapAccumR, 137
mapM, 137
mapM_, 138
mapPairs, 134
newVector, 118
nil, 118
or, 127
readVReg, 129
replicate, 118
replicateM, 139
reverse, 125
rotate, 123
rotateBitsBy, 129
rotateBy, 124
rotateR, 124
scanl, 136
scanr, 135
select, 120
shiftInAt0, 124
shiftInAtN, 124
shiftOutFrom0, 124
shiftOutFromN, 125
sscanl, 136
sscanr, 135
tail, 121
take, 121
takeAt, 122
toChunks, 141
toList, 141
transpose, 125
transposeLN, 125
unzip, 130
update, 120
vectorToArray, 141
writeVReg, 129
zip, 129
zip3, 130
zip4, 130
zipAny, 130
zipWith, 131
zipWith3, 132
zipWith3M, 138
zipWithAny, 131
zipWithAny3, 132
zipWithM, 138

Typeclasses

Alias, 22
Arith, 12

BitExtend, 20
BitReduction, 18
Bits, 9
Bitwise, 17
Bounded, 16

Connectable, 193

DefaultValue, 220

Eq, 10

FShow, 22

Literal, 11

NumAlias, 22

Ord, 14

Randomizable, 206
RealLiteral, 11

SaturatingArith, 21
SizedLiteral, 12
StringLiteral, 24

TieOff, 222
ToGet, 187
ToPut, 187