Bluespec SystemVerilog Reference Card

Revision: 11/07

bold	as is
italic	user identifier being declared
{ }	repeated
[ ]	optional

Capitalization

Foo: Type names, Typeclass names, Interface names, Enum labels, Tagged union labels, Package names
foo: bit[..], int, module names, instance names, all variables, all type variables, rule names

Package

```
package Package_name ;
   typedef statements
   import statements
   interface declarations
   module declarations
endpackage [: Package_name]
```

Import Statement

```
import Package_name :: * ;
```

Predefined Data Types

```
Bit#(n)
Int#(n)          // signed
Uint#(n)         // unsigned
Integer          // static elaboration only
Bool
String
Action
ActionValue#(t)
Rules
Tuple2#(t1, t2) ... Tuple7#(t1,..., t7)
int              // Int#(32)
Nat              // Bit#(32)
Maybe#(t)
```

Type Definition

```
Type_name
Type_name#(type_variable)        // polymorphic type
```

Type Synonym

```
typedef type Type_name[#({type type_var})];
```
example:
```
typedef Bit#(8) Byte;
typedef Tuple3#(a, a, a) Triple#(type a);
```

Interface Declaration

```
interface ifc_name;
   method declarations
   subinterface declarations
endinterface[:ifc_name]
interface ifc_name #({type Type_name});
   method declarations
   subinterface declarations
endinterface[:ifc_name]
```
example:
```
   interface MyIfc#(t) ;
      method Action tick() ;
      interface FIFO#(t) inbuffer ;
   endinterface:MyIfc
```

Method Declaration

```
method Type method_name [(Type argument)] ;
```

Module Definition

```
module module_name [# ({parameter})]
   ({Ifc_type ifc_name*})[provisos] ;
         module instantiations
         variable declaration and initializations
         rules
         interface/method definitions
endmodule [:module_name]
* ifc_name optional if only one ifc
```

Module Instantiation

```
Ifc_type ifc_name <-  module_name({parameter});
Ifc_type ifc_name <-  module_name([{parameter,}
   clocked_by clock_name,
   reset_by reset_name]);
```
example:
```
   Reg#(Time32) state <- mkReg(0);
```

Rules

```
rule rule_name [rule_predicate] ;
   action statements
endrule[: rule_name]
rules [: rules_name]
   rule
   variable declaration or variable assignment
endrules[: rules_name]
```

Action Block

```
action [: action_name] ;
   action statements
endaction [:action_name]
```

Value Method Definition

```
method Type method_name ({parameter})
  [if  (method_predicate)];
         method body statements
         return statement
endmethod [:method_name]
```

Action Method Definition

```
method Action method_name ({parameter})
  [if (method_predicate)];
         method body statements
endmethod [:method_name]
```

ActionValue Method Definition

```
method ActionValue method_name({parameter})
  [if (method_predicate)];
         method body statements
         return statement
endmethod [:method_name]
```

Variable Declaration and Initialization

```
Type {variable_name [= expression ]};
```
example:
```
   Integer x = 16, y = 32;
   int a[20], b[40];
   Int#(5) xs[2][4] = {{1,2,3,4},
                       {5,6,7,8}};
```

Variable Assignment

```
variable_name = expression ;
```
example:
```
   x = 23 ;
   b = foo.bar(x);
```

ActionValue Assignment Statement

Special <- notation used to perform the action and return the value
```
type identifier <- expression ;
identifier <- expression ;
```

Implicit Type Declaration and Initialization

```
let identifier = expression ;
```
if expression is actionvalue method
```
let identifier <- expression ;
```
example:
```
   let n = valueof(Buffsize);
   let z <- rndm.get;
```

Register Read and Write

```
register_name  <= expression ;
```
example:
```
   state <= state + 1  ; // same as: state._write (state.read() + 1)
```

Enumeration

```
typedef enum {{Elements}} Type_name
   [deriving (Typeclass)];
```
example:
```
  typedef enum {Red, White, Blue} Color
          deriving (Eq, Bits);
```

Structure

(*struct value contains member1 and member2, etc.*)

```
typedef struct {Type member1;...;Type memberN}
   Type_name [#{[numeric] type type_variable}]
             [deriving (Typeclass)];
```

example:

```
   typedef struct {Int x; Int y;} Coord
        deriving (Eq, Bits);
```

Declaration and initialization of a structure variable

```
Type variable_name = Type{member:expression}
   Coord c1 =  Coord{x:1, y:foo};
```

Update of a structure variable

```
   c1.x = c1.x + 5 ;
```

Structure member selection

```
   xposition = c1.x ;
```

Tagged Union

(*union value contains member1 or member2)*

```
typedef union tagged {type Member1; ... ;
   type MemberN;} Type_name [#...[numeric]
   type type_variable];
```

example:

```
   typedef union tagged { void Invalid;
                  int  Valid; } MaybeInt;
```

Declaration and initialization of a tagged union

```
Type variable_name = Member expression ;
   MaybeInt x =  tagged Valid 5 ;
```

Pattern Matching

Tagged Union

```
tagged Member [ pattern ]
```

Structure

```
tagged Type [ member:pattern ]
```

Tuple

```
tagged {pattern, pattern}
```

Pattern Matching Examples

Pattern matching in a case statement

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

Pattern matching in an if statement

```
    if (x matches tagged Valid .n &&& n > 5...)
```

Pattern Matching Assignment Statement

```
match pattern = expression ;
```

example:

```
   Tuple2#(Bits(32) x, Bool y) a_tuple;
   match {.a, .b} = a_tuple ;
```

Function Definition

```
function type function_name ([{arguments}])
   [provisos];
```

function body statements
return statement

```
endfunction [: function_name]
```

Attributes

```
(* {attribute [= expression ]}*)
```

Module Attributes (top-level only)

```
synthesize
RST_N = "string"
CLK = "string"
always_ready [= "interface_method"]
always_enabled [= "interface_method"]
descending urgency = "{rule_names}"
preempts = "{rule_names, (list_rule_names)]}"
doc = "string"
```

Method Attributes

```
always_ready [= "interface_method"]
always_enabled [= "interface_method"]
ready = "string"
enable = "string"
result = "string"
prefix =  "string"
port = "string"
```

Interface Attributes

```
always_ready [= "interface_method"]
always_enabled [= "interface_method"]
```

Function Attributes (top-level only)

```
noinline
```

Rule Attributes

```
fire_when_enabled
no_implicit_conditions
descending_urgency = "{rule_names}"
preempts "{rule_names, [(list_rule_names)]}"
```

System Tasks and Functions

$display	**$finish**
$write	**$stop**
$fopen	**$dumpon**
$fdisplay	**$dumpoff**
$fwrite	**$dumpvars**
$fgetc	**$test$plusargs**
$fflush	**$time**
$fclose	**$stime**
$ungetc	

Importing C Functions

```
import "BDPI" [c_function_name =] function
  Return_type function_name [{argument}])
  [provisos] ;
```

Importing Verilog Modules

```
import "BVI" [verilog_module_name] =
  module [[Type]] module_name [# ({parameter})]
  ({Ifc_type ifc_name}) [provisos] ;
```

module statements
importBVI statements

```
endmodule [: module_name]
```

importBVI Statements

```
parameter parameter_name = expression ;
port port_name = expression ;
default_clock clock_name
   [(port_name, port_name)][= expression];
input_clock clock_name [(port_name,
   port_name)] = expression;
output_clock clock_name
   (port_name [,port_name]);
no_reset;
default_reset clock_name ([port_name])
   [= expression];
input_reset clock_name
   ([port_name]) = expression;
output_reset clock_name ( port_name );
ancestor ( clock1, clock2 );
same_family ( clock1, clock2 );
method [output_port] method_name
   ({input_ports}) [enable enable_port]
   [ready ready_port][clocked_by
clock_name]    [reset_by clock_name];
schedule ({method_name}) operator
   ({method_name});
       operators are CF, SB, SBR, and C
path (port_name1, port_name2) ;
```

Defined Interfaces

Reg

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

PulseWire

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

Wire

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

Defined Modules

Reg

```
module mkReg#(a_type resetval)
(Reg#(a_type));
module mkRegU(Reg#(a_type));
module mkRegA#(a_type
resetval)(Reg#(a_type));
```

Wire

```
module mkWire(Wire#(a_type));
```

BypassWire

```
module mkBypassWire(Wire#(a_type));
```

DWire

```
module mkDWire#(a_type defaultval)
       (Wire#(a_type));
```

PulseWire

```
module mkPulseWire(PulseWire);
```

Library Packages

FIFOFs (import FIFOF::*;)

see LRM for additional FIFOs

Interface

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

Modules

```
module mkFIFOF# (FIFO#(a_type)) ;
module mkFIFOF1#(FIFO#(a_type));
module mkSizedFIFOF#(Integer n)(FIFO#(a_type)) ;
module mkLFIFOF#(FIFO#(a_type));
```

Get/Put (import GetPut::*;)

Interfaces

```
interface Get#(type a_type);
    method ActionValue#(a_type) get();
endinterface: Get
interface Put#(type a_type);
    method Action put(a_type x1);
endinterface: Put
```

Type

```
typedef Tuple2#( Get#(a_type), Put#(a_type) )
                 GetPut#(type a_type);
```

Connectable (import Connectable::*;)

Typeclass

```
typeclass Connectable#(type a , type b) ;
```

Module

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

Client/Server (import ClientServer::*;)

Interfaces

```
interface Client#(type req_type, type
resp_type);
    interface Get#(req_type)  request;
    interface Put#(resp_type) response;
endinterface: Client
interface Server#(type req_type, type
resp_type);
    interface Put#(req_type)  request;
    interface Get#(resp_type) response;
endinterface: Server
```

Type

typedef Tuple2#(Client#(req_type, resp_type),
Server#(req_type,resp_type))
ClientServer#(type req_type, type resp_type);

BSV Example

```
package Counter ;

interface Counter#(type count_t);
    method count_t read();
    method Action load(count_t newval);
    method Action increment();
    method Action decrement();
endinterface

module mkCounter(Counter#(count_t))
        provisos(Arith#(count_t), Bits#(count_t,
count_t_sz));
    Reg#(count_t) value <- mkReg(0);

    PulseWire increment_called <- mkPulseWire();
    PulseWire decrement_called <- mkPulseWire();

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

    method count_t read();
        return value;
    endmethod
    method Action load(count_t newval);
        value <= newval;
    endmethod
    method Action increment();
        increment_called.send();
    endmethod
    method Action decrement();
        decrement_called.send();
    endmethod
endmodule
endpackage: Counter
```