1 Bluespec Quick Reference

A heavily abbreviated Bluespec reference accompanying the Intro Guide. Covers more basic syntax than the Bluespec Reference Card (which doesn't even have for loops?)

1.1 Comments

```
// single line comment
/* multiline
comment */
```

1.2 Types

```
Bit#(n)
Int#(n)  // signed
Uint#(n) // unsigned
Integer  // static elaboration only
Bool

Action
ActionValue#(t)
Rules
Tuple2#(t1, t2) ... Tuple7#(t1,..., t7)
```

1.3 Values

```
0       // constant zero
42      // decimal 42 of arbitrary size
'1      // Enough 1 bits to fill the needed width
4'b1010 // 4-bit value 1010 in binary

True    // Bool
False   // Bool
```

1.4 Operators and Built-In Functions

If `a` and `b` are variables of type `Bit#(n)`, expressions include:

```
a & b   a | b   a ^ b   ~a

a + b   a - b   a * b   a / b   a % b

a << b  a >> b

{a, b}  // Bit concatenation
a[0]    // Bit indexing (0 is the least significant bit, i.e. the right!)
a[7:0]  // Bit slicing (inclusive of both indices, so this has 8 bits)

// Add or remove bits from the left; you may need to put them into a variable
// with explicitly declared type
signExtend(a) zeroExtend(a) truncate(a)

// Comparisons give values of type Bool
a == b  a != b  a < b   a > b   a <= b  a >= b
```

If `p` and `q` are variables of type `Bool`, expressions include:

```
p && q  p || q  !p

p ? a : b
```

If `i` is an `Integer`, you can use `fromInteger(i)` to convert it to a `Bits#(n)`

`pack` converts from various types to `Bit#(n)`, `unpack` converts from `Bit#(n)` to various types.

1.5 Type-Level Operations

Type-level	Equivalent math
`TAdd#(a, b)`	`a + b`
`TSub#(a, b)`	`a - b`
`TMul#(a, b)`	`a * b`
`TDiv#(a, b)`	`ceiling(a / b)`
`TLog#(a)`	`ceiling(log_2 a)`
`TExp#(a)`	`2^a (2 to the power of a, not 2 xor a)`
`TMax#(a, b)`	`max(a, b)`
`TMin#(a, b)`	`min(a, b)`

If `a` is a numeric type, you can use `valueOf(n)` to convert it to an `Integer`.

1.6 Variable Declarations

```
Bit#(3) a = 7;  // a has explicit type Bit#(3)
let b = {a, a}; // type of b is inferred
```

1.7 Tuples

```
Tuple2#(Bit#(1), Bit#(2)) pair = tuple2(1, 0);

Bit#(1) first = tpl_1(pair);
Bit#(2) second = tpl_2(pair);
// or
match {.first, .second} = pair;
```

1.8 Structs

```
typedef struct {
    Bit#(1) foo;
    Bit#(2) bar;
} NewType;

NewType myNewVar = NewType{foo: 1, bar: 2};

let newFoo = myNewVar.foo;
let newBar = myNewVar.bar;
// or
match tagged NewType {foo: .newFoo, bar: .newBar} = myNewVar;
```

1.9 Enums

```
typedef enum Color { Red, Green, Yellow, Blue } deriving (Bits, Eq);
Color color = Red;
```

1.10 Control Flow

```
if (condition) begin
    x = 5;
end

for (Integer i = 0; i < max; i = i + 1) begin
    do_something();
end
```

1.11 Switch

```
case (someValue)
    1: do1();
    2: do2();
    default: do3();
endcase

let foo = case (someValue)
    1: bar;
    2: baz;
    default: quux;
endcase;
```

1.12 Functions

```
function ReturnType fnName(Type var1, Type var2); // semicolon!
    some_stuff();
    return other_stuff();
endfunction
```

1.13 Modules

```
interface MyInterface;
    method Action myAction(Bit#(1) flag);
    method ActionValue#(Bool) getMyResult;
endinterface

module mkMyModule(MyInterface);
    Reg#(Bool) myReg <- mkRegU;

    rule doSomething;
        do_some_stuff();
    endrule

    method Action myAction(Bit#(1) flag) if (myReg);
        do_some_other_stuff();
    endmethod

    method ActionValue#(Bool) getMyResult if (!myReg);
        return True;
    endmethod
endmodule
```

1.14 Registers

In a module, outside rules and methods, use `<-` to create registers and modules.

```
Reg#(Bit#(1)) myReg <- mkRegU();
Reg#(Bool) myRegFlag <- mkReg(False);
```

In a module rule or method, use `<-` to perform `ActionValue`s.

```
let result <- someModule.someMethod(someArg);
```

In a module rule or method, use `<=` to write into registers. Reading from registers is implicit.

```
myReg <= 1; // write 1 to x
```

1.15 Module Example

```
interface Tripler;
    method Action start(Bit#(32) n);
    method ActionValue#(Bit#(32)) getResult;
endinterface

module mkTripler(Tripler);
    Reg#(Bit#(32)) x <- mkRegU;
    Reg#(Bit#(32)) y <- mkRegU;
    Reg#(Bool) busy <- mkReg(False);
    rule tripleStep if (busy && x > 0);
        x <= x - 1;
        y <= y + 3;
    endrule
    method Action start(Bit#(32) n) if (!busy);
        x <= n;
        y <= 0;
        busy <= True;
    endmethod
    method ActionValue#(Bit#(32)) getResult if (busy && x == 0);
        busy <= False;
        return y;
    endmethod
endmodule
```