Introduction to Bluespec

Andy Wright
acwright@mit.edu

Updated: December 30, 2013

1 Combinational Logic

1.1 Primitive Types

definition: `Bool`

This is a basic type for expressing true-false values with a single bit. Literal values of this type are `True` and `False`.

definition: `Bit#(`*n*`)`

This is your basic bit-vector type in Bluespec. *n* is a number[1] that represent the number of bits in this bit-vector. `Bit#(5)` is the type definition for a 5-bit bit-vector. Literals values of this type look like the following:

- Decimal values - 0, 1, 2, ...
- Binary values - `5'b01101`, `5'b11100`, ...
- Hex values - `5'h07`, `5'h1f`, ...

Single bits or ranges of bits can be accessed using `a[1]` and `a[3:0]` notation where `a[0]` is the least significant bit of `a`. Multiple bits or bit-vectors can be combined by concatination using {`a,b`}. For arithmetic operations, values of this type are treated as unsigned.

definition: `Int#(`*n*`)`

This type is similar to `Bit#(`*n*`)`. For arithmetic operations, values of this type are treated as signed 2's complement numbers.

[1] Actually *n* needs to be a numberic type. This will be covered later.

definition: `UInt#(n)`

This type is also similar to `Bit#(n)`. For arithmetic operations, values of this type are treated as unsigned numbers.

1.2 Constructing New Types

definition: `typedef`

The `typedef` command can be used to create aliases for types. The following creates an alias called `OpCode` for `Bit#(8)`.

```
typedef Bit#(8) OpCode;
```

The `typedef` command can only appear at the top level of your bluespec source files; it cannot be used within modules.

definition: `enum`

The `enum` command can be used to create enumerations for lists. The following example shows an enumeration for colors.

```
typedef enum {red, blue, green, yellow} Color deriving (Bits, Eq);
```

The `Bits` portion of the `deriving` statement says that the values will be assigned bit values so they can be represented on a bus of wires or in a register. The `Eq` portion allows for values of the enumeration type to be compared for equality.

definition: `struct`

The `struct` command can be used to create structures containing fields of other types. The following example is a structure for a memory request.

```
typedef struct {
    Bit#(12) address;
    Bit#(8) data;
    Bool write_en;
} MemReq;
```

Fields of a structure can be modified one at a time:

```
MemReq req = ?;
req.address = 12'h100;
req.data = 8'h24;
req.write_en = True;
```

...or all at once:

```
MemReq req = { address: 12'h104, data: 8'h24, write_en: False };
```

definition: `Maybe#(t)`

The `Maybe#(t)` type effectively adds a valid bit to an existing type *t* to create a new type. Values of maybe types match one of two categories:

- `tagged Valid` *v* (where *v* is a value of type *t*)
- `tagged Invalid`

Values of maybe data types can be checked for valid values using the `isValid(x)` function. Values can be extracted from valid values of a maybe data type using the function `fromValid(default_val, x)`. `fromValid` returns *default_val* when `x` is not valid. The don't care value `?` is commonly used for the default value of `fromValid` when an `isValid` check is performed before using `fromValid`.

Trying to incrementing a maybe value by one looks like this:

```
Maybe#(Bit#(8)) x, y;
if( isValid(x) ) begin
    y = tagged Valid (fromMaybe(?, x) + 1);
end else begin
    y = tagged Invalid;
end
```

This is functionally equivalent to the following code:

```
Bit#(8) x, y;
Bool x_valid, y_valid;
if( x_valid ) begin
    y = x + 1;
    y_valid = True;
end else begin
    y_valid = False;
end
```

definition: `tagged union`

A tagged unions is a way to allow a variable to contain one of many data types while keeping a tag to signal what data type is currently stored. `Maybe#(t)` is a specific example of a tagged union as shown below.

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

Dexember 30th: This document is still a work in progress. There is a lot more to say about tagged unions. If you want to read about them before I finish this document, take a look at the Bluespec reference manual.

definition: `Vector#(n,t)`

`Vector#(n,t)` is a vector data type of `n` elements of type `t`. Single elements can be accessed using bracket notation. Vectors can be initialized using `newVector()` for an undefined vector or `replicate(v)` for a vector filled with the value `v`. To use vectors of types, you must have the following line at the top of your source file.

```
import Vector::*;
```

1.3 Logic Functions

definition: `&`, `|`, `^`, `~`

These are the *bitwise* logic operators for and, or, xor, and inverse. These functions work on data types that can be expressed in bits and return values of the same data type.

definition: `&&`, `||`, `!`

These are the *boolean* logic operators for and, or, and inverse. These functions work on `Bool` values and return a `Bool` value.

definition: `==`, `!=`, `<`, `<=`, `>`, `>=`

These are the comparison operators. These functions operate on two values of the same comparable data type and return a `Bool` value.

1.4 Bitwise Functions

definition: `zeroExtend`, `signExtend`

These functions can be used to produce longer bit vectors from shorter bit vectors. `zeroExtend` pads the left side of the bit vector with 0's. `signExtend` pads the left side of the bit vector with the MSB of the original bit vector.

definition: `truncate`, `truncateLSB`

These functions reduce the size of a bit vector. `truncate` trims off bits from the left side, and `truncateLSB` trims off bits from the right side.

definition: `<<`, `>>`

These are the binary bit-shift operators. `a << b` shifts the value `a` left by `b` positions, and 0's are shifted in. `a >> b` shifts the value `a` right by `b` positions, and either 0's or 1's are shifted in depending on the sign of `a`. If `a` is `Bit#(`*n*`)` or `UInt#(`*n*`)`, then a is always positive and 0's are shifted in. If `a` is `Int#(`*n*`)` and the most significant bit of `a` is

1.5 Arithmetic Functions

definition: `+`, `-`, `*`, `/`, `%`

These are the basic arithmetic functions add, subtract, multiply, divide, and modulo (remainder). The operands for these functions must be values of the same data type. These functions return a value of the same data type as the operands. To add two n-bit numbers to get an n+1-bit number, you must use signExtend on the operands and add two n+1-bit numbers instead. *Warning:* divide (`/`) and modulo (`%`) will simulate properly, but depending on the verilog compiler, they may not synthesize when using values that are not powers of 2 (for example `n % 2` will synthesize but `n % 3` may not).

1.6 Defining Functions

definition: `function`

Combinational functions can be defined using the `function` keyword like the following

```
function Bit#(9) eight_bit_signed_add(Bit#(8) a, Bit#(8) b);
    Bit#(9) a_ext = signExtend(a);
    Bit#(9) b_ext = signExtend(b);
    return a_ext + b_ext;
endfunction
```

1.7 Syntactic Sugar

definition: `if`

```
if( x < 0 ) begin
    y = -x;
end else if(x == -1) begin
    y = -1;
end else begin
```

```
    y = 0
end
```

definition: `case`

There are two ways to use the `case` keyword. First it can be used in a case expression like the below code.

```
out = case (sel)
          1'b0: in0;
          1'b1: in1;
      endcase
endcase
```

Second it can be used in a case statement like the below code.

```
case (sel)
    2b00: a <= 0;
    2b01: a <= 1;
    2b10: a <= 2;
    2b11: begin
              a <= 3;
              done <= True;
          end
endcase
```

definition: `for`

Bluespec contains support for loops to replace repetative sections of code. Its syntax is nearly identical to C. One big difference is that Bluespec does not have an increment operator like ++ in C, so loop indexes have to be incremented using addition by one and assignment.

When the Bluespec compiler sees a for loop at compile time, it performs static elaboration which unrolls the loop into individual lines of code. For example, the for loop here

```
for( Integer i = 0 ; i < 4 ; i = i + 1 ) begin
    y[i] = x[3 - i];
end
```

is elaborated at compilet time to the lines of code below.

```
y[0] = x[3-0];
y[1] = x[3-1];
y[2] = x[3-2];
y[3] = x[3-3];
```

1.8 Deducing Types

definition: `let`

The keyword `let` can be used to tell the compiler to deduce the type of a new variable. For example, adding two values of type `Bit#(5)` with the `+` operator always returns a value of type `Bit#(5)`, the `let` keyword can be used to set the type of the output as shown in the example below.

```
Bit#(5) a = 2;
Bit#(5) b = 7;
let x = a + b;
```

The Bluespec compiler also deduces types for intermediate values in an expression. For example, the Bluespec compiler deduces the type of the intermediate values `a + b` and `zeroExtend(a + b)` in the example below.

```
Bit#(5) a = 2;
Bit#(5) b = 7;
Bit#(8) c = 65;
Bit#(8) y = zeroExtend(a + b) + c;
```

1.9 Numeric Types

Bluespec requires type definitions like `Bit#(n)` to have the size (`n`) defined as a numeric type. Literals like `5` can be interpreted as numeric types, but values like `n + 1` (or even `5 + 1`) are not numeric types.

definition: `TAdd#()`, `TSub#()`, `TMul#()`, `TDiv#()`, `TLog#()`, `TExp#()`

These are numeric type constructors that simulate addition, subtraction, multiplication, division, logarithm base-2, and exponentials base-2 to create new numeric types from existing numeric type variables and literals. Here is an example of `TAdd#()` being used.

```
    Bit#(n) x, y;
    Bit#(TAdd#(n,1)) z = zeroExtend(x) + zeroExtend(y);
```

definition: `valueOf()`

`valueOf` is a pseudo function that takes a numeric type and returns a corresponding value of type `Integer`. This is especially useful for looping over bits in a `Bit#(n)` with a for loop.

1.10 Type Errors

The Bluespec compiler throws type errors for various reasons. If the Bluespec sees a type it was not expecting, it throws an error. Also when the Bluespec compiler has trouble deducing the type of a value (either a value defined with the `let` keyword or an intermediate value), the compiler also throws an error. Here are the main errors the compiler reports.

definition: "`Type error at: val, Expected type: typeA, Inferred type: typeB`"

```
Bit#(8) x = 3;
Bit#(8) y = 25;
Bit#(9) z = x + y;
```

With this code, the compiler would say there is a type error at `x` with an expected type of `Bit#(9)` and an inferred type of `Bit#(8)`. This type mismatch comes from looking at `z`. Since `z` is of type `Bit#(9)`, the operands of the addition that results in `z` need to both be of type `Bit#(9)`. Since `x` needs to be of type `Bit#(9)`, but it is actually `Bit#(8)`, the compiler produces an error. Depending on the desired functionality of this code it can be fixed in two ways:

```
Bit#(9) z = zeroExtend(x) + zeroExtend(y);
```

or

```
Bit#(8) z = x + y;
```

definition: "`Bit vector of unknown size introduced near this location.`"

```
Bit#(8) x = 25;
Bit#(8) y = truncate(300 + zeroExtend(x));
```

The compiler cannot deduce the type of the intermediate value `300 + zeroExtend( x )`. Fix this by assigning a type to the intermediate value like this:

```
Bit#(8) x = 25;
Bit#(9) tmp = 300 + zeroExtend(x);
Bit#(8) y = truncate(tmp);
```

definition: "`Unexpected '+'; expected type parameters, ',', or ')'`"

```
Bit#(n) x;
Bit#(n+1) y = signExtend(x);
```

The size of a bit vector needs to always be a numeric type. In this example, the compiler is complaining about `+` because the result is not a numeric type. Instead you need to use the type construcor `TAdd#(n,1)` to get the numeric type that one larger than n. The fixed code would look like this:

```
Bit#(n) x;
Bit#(TAdd#(n,1)) y = signExtend(x);
```