Bluespec SystemVerilog (BSV)

Quan Nguyen
(Only crashed PowerPoint three times)

What’s Bluespec?

• “A synthesizable subset of SystemVerilog”
• Rule-based execution
• Formal semantics, type safety, object-oriented programming, higher-order functions
• A way for you to express hardware designs
  ■ But you still have to know the syntax
BSV Lineage

“Bluespec is pretty much a port of Haskell”
– anonymous grad student

Outline

• Standard types
• User-defined types
• Modules, interfaces, and methods
• Writing and debugging BSV
Standard Types

Bit#(numeric type n)

- **Literal values:**
  - Decimal: 0, 1, 2, ... (each has type Bit#(n))
  - Binary: 5'b01101, 2'b11
  - Hex: 5'hD, 2'h3, 16'h1FF0

- **Common functions:**
  - Bitwise Logic: |, &, ^, ~, etc.
  - Arithmetic: +, -, *, %, etc.
  - Indexing: a[i], a[3:1]
  - Concatenation: {a, b}
  - truncate(), truncateLSB()
  - zeroExtend(), signExtend()
**Bool**

- Literal values: True, False
- Boolean Logic: |, &&, !, ==, !=, etc.
- All comparison operators (==, !=, >, <, >=, <=) return Bools

---

**Int#(n), UInt#(n)**

- Literal values:
  - Decimal:
    - 0, 1, 2, ... (Int#(n) and UInt#(n))
    - -1, -2, ... (Int#(n))
  - `int` a synonym for Int#(32)
- Common functions:
  - Arithmetic: +, -, *, %, etc.
  - Int#(n) performs signed operations
  - UInt#(n) performs unsigned operations
  - Comparison: >, <, >=, <=, ==, !=, etc.
Numeric types, Integers, Ints, oh my!

```
typedef 5 NumWidth;
```

```
for (Integer i = 0; i < 5; i = i + 1)
```

```
Bit#(NumWidth) a = 5;
```

```
Int#(NumWidth) a = 5;
```

User-defined types
Constructing new types

- "Renaming" types:
  - typedef

- Enumeration types:
  - enum

- Compound types:
  - struct
  - Vector
  - Maybe
  - tagged union

typedef keyword

- Syntax:
  - typedef <type> <new_type_name>;

- Basic:
  - typedef 8 BitsPerWord;
  - typedef Bit#(BitsPerWord) Word;
    - Can't be used with parameter: Word#(n)

- Parameterized:
  - typedef Bit#(TMul#(BitsPerWord, n))
    Word#(numeric type n);
    - Can't be used without parameter: Word
**enum keyword**

```haskell
typedef enum {Red, Blue} Color deriving (Bits, Eq);
```

- Creates the type Color with values Red and Blue
- Can create registers containing colors
  - `Reg#(Color)`
- Values can be compared with `==` and `!=`
  - (Why?)

---

**struct keyword**

```haskell
typedef struct {
    Bit#(12) addr;
    Bit#(8) data;
    Bool wren;
} MemReq deriving (Bits, Eq);
```

- Elements from `MemReq x` can be accessed with `x.addr`, `x.data`, `x.wren`
- Struct expression
  - `x = MemReq{addr: 0, data: 1, wren: True};`
**struct keyword**

```haskell
typedef struct {
    t a;
    Bit#(n) b;
} Req#(type t, numeric type n) deriving (Bits, Eq);
```

- Parameterized struct

**Tuple**

```haskell
Tuple2#(Bit#(2), Bool) tup = tuple2(2, True);
macht {.a, .b} = tup; // a = 2, b = True
```

- Types:
  - Tuple2#(type t1, type t2)
  - Tuple3#(type t1, type t2, type t3)
  - up to Tuple8
  - Consider using structs for larger aggregate types

- Construct tuple: `tuple2(x, y), tuple3(x, y, z)` ...

- Accessing an element:
  - `tpl_1(tuple2(x, y))` // x
  - `tpl_2(tuple3(x, y, z))` // y

- Pattern matching
Vector

- **Type:**
  - \texttt{Vector\#(numeric\_type\ size, type data\_type)}
- **Values:**
  - \texttt{newVector()}, \texttt{replicate(val)}
- **Functions:**
  - Access an element: []
  - Rotate functions
  - Advanced functions: \texttt{zip, map, fold}
- Can contain registers or modules
- Must have `'import Vector::*;' in BSV file

Maybe\#(t)

- **Type:**
  - \texttt{Maybe\#(type t)}
- **Values:**
  - tagged \texttt{Invalid}
  - tagged \texttt{Valid x} (where \texttt{x} is a value of type \texttt{t})
- **Functions:**
  - isValid(x)
    - Returns true if \texttt{x} is valid
  - fromMaybe\texttt{(default, m)}
    - If \texttt{m} is valid, returns the valid value of \texttt{m} if \texttt{m} is valid, otherwise returns default
    - Commonly used as `'fromMaybe(?, m)``
tagged union

- Maybe is a special type of tagged union

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

- Tagged unions are collections of types and tags. The type contained in the union depends on the tag of the union.
  - If tagged Valid, this type contains a value of type t

- Values:
  - tagged <tag> value

- Pattern matching to get values:

```cpp
case (x) matches
    tagged Valid .a : return a;
    tagged Invalid : return 0;
endcase
```
Pattern matching and `&&&`

- Use `.*` to skip a part of the struct

```plaintext
Tuple2#(Bit#(2), Bool) tup = tuple2(2, True);
match {.a, .*} = tup; // a = 2, b not assigned
```

- Use `&&&` to “filter” `match` expressions with ordinary conditional statements

```plaintext
function tup_even(Tuple2#(Bit#(2), Bool) tup) =
tup matches {.a, .*} &&
a [0] == 0 ? True : False;

if (a matches tagged Valid .v && v == 5) ...
```

Reg#(t)

- Main state element in BSV
- Type: Reg#(type data_type)
- Instantiated differently from normal variables
  - Uses <- notation
- Written to differently from normal variables
  - Uses <= notation
  - Can only be done inside of rules and methods

```plaintext
Reg#(Bit#(32)) a_reg <- mkReg(0); // value set to 0
Reg#(Bit#(32)) b_reg <- mkRegU(); // uninitialized
```
Reg and Vector

- Register of Vectors
  - Reg#(Vector#(32, Bit#(32))) rfile;
  - rfile <- mkReg(replicate(0));

- Vector of Registers
  - Vector#(32, Reg#(Bit#(32))) rfile;
  - rfile <- replicateM(mkReg(0));

- Each has its own advantages and disadvantages

Partial Writes

- Reg#(Bit#(8)) r;
  - r[0] <= 0 counts as a read & write to the entire reg r
    - let r_new = r; r_new[0] = 0; r <= r_new;
  - Reg#(Vector#(8, Bit#(1))) r;
    - Same problem, r[0] <= 0 counts as a read and write to the entire register
      - r[0] <= 0; r[1] <= 1 counts as two writes to register
        - double write problem
  - Vector#(8, Reg#(Bit#(1))) r;
    - r is 8 different registers
    - r[0] <= 0 is only a write to register r[0]
    - r[0] <= 0 ; r[1] <= 1 is not a double write problem
Modules, interfaces, and methods

- Modules are building blocks for larger systems
  - Modules contain other modules and rules
  - Modules are accessed through their interface
- module mkAdder(Adder#(32));
  - Adder#(32) is the interface
- Module can be parameterized
  - module name#(params)(args ..., interface);

module mkMul#(Bool signed)(Adder#(n) a, Mul#(n) x);
Interfaces

- Contain methods for other modules to interact with the given module
  - Interfaces can also contain sub-interfaces

```plaintext
interface MyIfc#(numeric type n);
    method ActionValue#(Bit#(n)) f();
    interface SubIfc#(n) s;
endinterface
```

- Special interface: Empty
  - No method, used in testbench

```plaintext
module mkTb(Empty);
module mkTb();       // () are necessary
```

Interface Methods

- **Method**
  - Returns value, doesn't change state
    - method `Bit#(32) first;`

- **Action**
  - Changes state, doesn't return value
    - method `Action enq(Bit#(32) x);`

- **ActionValue**
  - Changes state, returns value
    - method `ActionValue#(Bit#(32)) deq;`
  - Must use `<-` operator
Implement Interface of Module (variant 1)

- Instantiate methods at the end of module

```plaintext
interface MyIfc#(numeric type n);
  method ActionValue#(Bit#(n)) f();
interface SubIfc#(n) s;
endinterface
module mkDut(MyIfc#(n));
  ___
  method ActionValue#(Bit#(n)) f();
  ___
  endmethod
  interface SubIfc s; // no param “n”
  // methods of SubIfc
  endinterface
endmodule
```

Implement Interface of Module (variant 2)

- Return interface at the end of module

```plaintext
module mkDut(MyIfc#(n));
  ___
  MyIfc ret = (interface MyIfc;
    method ActionValue#(Bit#(n)) f();
    ___
    endmethod
    interface SubIfc s; // no param “n”
    // methods of SubIfc
    endinterface
endinterface);
  return ret;
endmodule
```
Vector Sub-interface

• Sub-interface can be vector

```vcs
interface VecIfc#(numeric type m, numeric type n);
    interface Vector#(m, SubIfc#(n)) s;
endinterface
```

```vcs
Vector#(m, SubIfc) vec = ?;
for (Integer i = 0; i < valueOf(m); i = i + 1) begin
    // implement vec[i]
end
VecIfc ifc = (interface VecIfc;
    interface Vector s = vec;   // interface s = vec;
endinterface);
```

• BSV Reference Guide Section 5
Best way to learn BSV

- BSV Reference Guide
- Lab code
- Try it
  - Makefile in lab 1,2,3...

Debugging

- $display()
  - Can only use where Actions allowed
  - Works like C printf() or Python str.format()
- FShow typeclass
  - Creates “pretty-printed” strings for user-defined types
  - Typeclasses covered in Tutorial 2
Scheduling

- Compile flag (BSV User Guide [not Reference])
  - `-aggressive-conditions` (Section 7.12)
    - predicated implicit guards
  - `-show-schedule` (Section 8.2.2)
    - method/rule schedule information
    - Output file: `buildDir/* sched`
  - `-show-rule-rel r1 r2` (Section 8.2.2)
    - Print conflict information

Credits

- Considerable previous material adapted from last year's tutorial by Sizhuo Zhang and Andy Wright