CS 250B: Modern Computer Systems
Introduction To Bluespec – Types

Sang-Woo Jun

Many slides adapted from
Arvind’s MIT “6.175: Constructive Computer Architecture”
and Hyoukjun Kwon’s Gatech “Designing CNN Accelerators”
Bluespec Types

- Primitive types
  - Bit, Int, UInt, Bool

- User-defined types
  - typedef, enum, struct

- More complex types
  - Tuple, Maybe, Vector, ...
Bit#(numeric type n) – The Most Basic

- **Literal values:**
  - Decimal: 0, 1, 2, ... (each have type Bit#(n))
  - Binary: 5’b01101 (13 with type Bit#(5)), 2’b11 (3 with type Bit#(2))
  - Hex: 5’hD, 2’h3, 16’h1FF0, 32’hdeadbeef

- **Supported functions:**
  - Bitwise Logic: |, &, ^, ~, etc.
  - Arithmetic: +, -, *, %, etc.
  - Indexing: a[i], a[i:j]
  - Concatenation: {a, b}

```text
Reg#(Bit#(32)) x <- mkReg(0);
rule rule1;
  Bit#(32) t = 32’hcaef00d;
  Bit#(64) m = zeroExtend(t)*zeroExtend(x);
  x <= truncate(m); // x <= m[31:0];
endrule
```
Int#(n), UInt#(n)

- **Literal values:**
  - Decimal: 0, 1, 2, ... (Int#(n) and UInt#(n))
  - -1, -2, ... (Int#(n))

- **Common functions:**
  - Arithmetic: +, -, *, %, etc. (Int#(n) performs signed operations, UInt#(n) performs unsigned operations)
  - Comparison: >, <, >=, <=, ==, !=, etc.

- **Bluespec provides some common shorthands**
  - int: Int#(32)
Bool

- Literal values:
  - True, False

- Common functions:
  - Boolean Logic: ||, &&, !, ==, !=, etc.

- All comparison operators (==, !=, >, <, >=, <=) return Bools

```verilog
Reg#(Bit#(32)) x <- mkReg(0);
Reg#(Bool) same_r <- mkReg(False);

rule rule1;
  Bit#(32) t = 32'hcafef00d;
  Bool same = (t==x);
  if ( same ) begin
    x <= 0;
    same_r <= True;
  end
endrule
```
Bluespec Types

- Primitive types
  - Bit, Int, UInt, Bool

- User-defined types
  - typedef, enum, struct

- More complex types
  - Tuple, Maybe, Vector, ...
**typedef**

- **Syntax:** `typedef <type> <new_type_name>;`
- **Basic examples**
  - `typedef 8 BitsPerWord;`
  - `typedef Bit#(BitsPerWord) Word;`
    - Can’t be used with parameter: `Word#(n) var; // Error!`
- **Parameterized example**
  - `typedef Bit#(TMul#(BitsPerWord,n)) Word#(n);`
    - Polymorphic type – Will be covered in detail later (BitsPerWord*n bits)
    - Can’t be used without parameter: `Word var; // Error!`
- **In global scope outside module boundaries**
**enum**

- **Syntax:** `enum {elements, ...}`
  - Typically used with `typedef`

```haskell
typedef enum{Mon, Tue, Wed, Thu, Fri, Sat, Sun} Days deriving (Bits, Eq);
module ...
    Reg#(Days) x <- mkReg(Sun);
    rule rule1;
        if ( day == Sun ) $display("yay");
    endrule
endmodule
```

Typeclasses will be covered later
**struct**

- **Syntax:** `struct {<type> <name>; ...}`
  - Typically used with `typedef`
  - Dot `.` notation to access sub elements

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

module ...
    Reg#(Memreq) x <- mkReg(MemReq{address:0, data: 0, write_en:False});
    Reg#(Memreq) y <- mkReg(?); //If you don’t care about init values
    rule rule1;
        if ( x.write_en == True ) $display("yay");
    endrule
endmodule
```
Bluespec Types

- **Primitive types**
  - Bit, Int, UInt, Bool

- **User-defined types**
  - typedef, enum, struct

- **More complex types**
  - Tuple, Maybe, Vector, ...
Tuples

- **Types:**
  - Tuple2#(type t1, type t2)
  - Tuple3#(type t1, type t2, type t3)
  - up to Tuple8

- **Values:**
  - tuple2( x, y ),
  - tuple3( x, y, z ), ...

- **Accessing an element:**
  - tpl_1( tuple2(x, y) ) = x
  - tpl_2( tuple3(x, y, z) ) = y
  - ...

```verilog
module ...
    FIFO#(Tuple3#(Bit#(32),Bool,Int#(32))) tQ <- mkFIFO;
    rule rule1;
        tQ.enq(tuple3(32'hc00d00d, False, 0));
    endrule
    rule rule2;
        tQ.deq;
        Tuple3#(Bit#(32),Bool,Int#(32)) v = tQ.first;
        $display( "%x", tpl_1(v) );
    endrule
endmodule
```
Vector

- **Type:** Vector#(numeric type size, type data_type)
- **Values:**
  - newVector()
  - replicate(val)
- **Functions:**
  - Access an element: []
  - Rotate functions
  - Advanced functions: zip, map, fold, ...
- **Provided as Bluespec library**
  - Must have ‘import Vector::*;’ in BSV file
import Vector::*; // required!

module ...
    Reg#(Vector#(8, Int#(32))) x <- mkReg(newVector());
    Reg#(Vector#(8, Int#(32))) y <- mkReg(replicate(1));
    Reg#(Vector#(2, Vector#(8, Bit#(32)))) zz <- mkReg(replicate(replicate(0)));
    Reg#(Bit#(3)) r <- mkReg(0);

    rule rule1;
        $display("%d", x[0] );
        x[r] <= zz[0][r];
        r <= r + 1; // wraps around
    endrule
endmodule
Array of Values Using Reg and Vector

- **Option 1: Register of Vectors**
  - `Reg#( Vector#(32, Bit#(32)) ) rfile;`
  - `rfile <- mkReg( replicate(0) ); // replicate creates a vector from values`

- **Option 2: Vector of Registers**
  - `Vector#( 32, Reg#(Bit#(32)) ) rfile;`
  - `rfile <- replicateM( mkReg(0) ); // replicateM creates vector from modules`

- Each has its own advantages and disadvantages
Partial Writes

- Reg#(Bit#(8)) r;
  - r[0] <= 0 counts as a read and write to the entire register r
  - Bit#(8) 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 r – write conflict error

- 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 does not cause a write conflict error
Maybe

- Type with a flag specifying whether it is valid or not
- Type: Maybe#(type t)
- Values:
  - tagged Invalid
  - tagged Valid x (where x is a value of type t)
- Helper Functions:
  - isValid(x)
    - Returns true if x is valid
  - fromMaybe(default, m)
    - If m is valid, returns the valid value of m if m is valid, otherwise returns default
    - Commonly used fromMaybe(? , m)
Maybe Example

```haskell
module ...

    Reg#(Maybe#(Int#(32))) x <- mkReg(tagged Invalid);

rule rule1;
    if (isValid(x)) begin
        Int#(32) value = fromMaybe(?,x);
        $display( "%d", value );
    end else begin
        x <= tagged Valid 32'hcafef00d;
    end
endrule
endmodule
```
Tagged Union

- Single value interpreted as different types, like C unions
- Syntax: `typedef union tagged { type Member1, ...} UnionName;`
  - Member names ("Member1", etc) are called tags
  - Member types can also be composite (struct, etc)

```c
typedef union tagged {
    Bit#(5) Register;
    Bit#(22) Literal;
    struct {
        Bit#(5) regAddr;
        Bit#(5) regIndex;
    } Indexed;
} InstrOperand;
```

Members share memory location
Tagged Union Usage

- Literal assignment syntax: tagged MemberName value;
- Use pattern matching ("case matches" syntax) to get values

```cpp
typedef union tagged {
    Bit#(5) Register;
    Bit#(22) Literal;
    struct {
        Bit#(5) regAddr;
        Bit#(5) regIndex;
    } Indexed;
} InstrOperand;

InstrOperand orand;
orand = tagged Indexed { regAddr:3, regIndex:4 };
orand = tagged Register 23;
...

case (orand) matches
    tagged Register .a : ...; //uses Register a
    tagged Literal .a : ...; //uses Literal a
...
endcase
```
This Had Been The Bluespec Types Catalog

- ...For now!
  - Real, Complex, FloatingPoint, ...
  - tagged unions, ...
Automatic Type Deduction Using “let”

- “let” statement enables users to declare a variable without providing an exact type
  - Compiler deduces the type using other information (e.g., assigned value)
  - Like “auto” in C++11, still statically typed

```plaintext
module ...
  Reg#(Maybe#(Int#(32))) x <- mkReg(tagged Invalid);

  rule rule1;
    if (isValid(x)) begin
      let value = fromMaybe(?,x);
      Int#(16) value2 = 0;
      if (value+value2 < 0) $display(“yay”); // error! Int#(32), Int#(16) mismatch
    end
  endrule
endmodule
```
Numeric Type And Numeric Literal

- Integer literal in a type context is a numeric type
  - This number does not exist in the generated circuit, and only affects circuit creation
  - e.g., `typedef 32 WordLength; Bit#(WordLength) val;`

- Integer literal as stored in state and processed by rules is a numeric literal
  - e.g., `Bit#(8) val = 32;`

- A third type, “Integer” represents unbounded integer values
  - Used to represent non-type numeric values for circuit creation
  - e.g., `Integer length = 16;`

- Numeric type and numeric literals are not interchangeable
  - `Bit#(8) len = WordLength; //Error!`
## Type Conversion Between Literals

- **Type conversion can only be done in one direction**

### Numeric Type

- **valueOf** from **Integer**

### Integer

- **fromInteger** to **Numeric Literal**

#### Module parameters
- Cannot be modified after defined
- Example: data bit-width, number of PEs, etc.

#### Conceptual numbers in a circuit (not a signal)
- Example: The index of a register array

#### Real Values in a circuit
- Represents values that exist either on a wire or memory element (register/FIFO)
Numeric Type And Numeric Literal Examples

typedef 32 WordLength;
module ...
    Reg#(Int#(WordLength)) x <- mkReg(0);
    Integer wordLength = valueOf(WordLength);
    Integer importantOffset = 2;
    FIFO#(WordLength) someQ <- mkSizedFIFO(wordLength);

    rule rule1;
        if (x[wordLength-1] == 1 && x[importantOffset] == 1) begin
            x <= fromInteger(wordLength);
        end
    endrule
endmodule
Typeclasses

☐ Bits: Types whose values can be converted to/from Bit vectors
  o Supports pack/unpack, SizeOf#(type), etc

☐ Eq: Types whose values can be checked for equality
  o Supports value1 == value2, value1 != value2

☐ Arith: Types whose values can be arithmetically computed
  o Supports +,-,*,/,%,-,abs,**,log (base e!),logb,log2,log10,etc...

☐ Ord: Types whose values can be compared
  o Supports <, >, >=, <=

☐ Bitwise: Types whose values can be modified bitwise
  o Supports bitwise &, |, ^, ~,...

☐ And more...
Typeclasses

- typedef’d types can derive typeclasses, but can only derive what its element allows
  - For example, we cannot derive Ord or Arith for an enum or struct
  - Bit# derives Bitwise, (among others), but Int# does not
  - Bool does not derive Ord, so True > False results in error

- We can add a new type to a typeclass by defining all supported functions
  - Definitely not a topic for “Bluespec Introduction”

```haskell
typedef enum{Mon, Tue, Wed, Thu, Fri, Sat, Sun} Days deriving (Bits, Eq);
```