Constructive Computer Architecture

Tutorial 1

BSV Types

Andy Wright
6.175 TA

Bit#(numeric type n)

- The most important type in BSV
  - We'll go into the details later

Bit#(numeric type n)

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

Bool

- Literal values:
  - True, False
- Common functions:
  - 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))

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

Constructing new types

- Renaming types:
  - typedef
- Enumeration types:
  - enum
- Compound types:
  - struct
  - vector
  - maybe
  - tagged union

typedef

- 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#(n);`
    - Can't be used *without* parameter: `Word`

enum

```
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 !=

struct

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

- Elements from `MemReq x` can be accessed with `x.address`, `x.data`, `x.write_en`

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
  - …

Vector

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

Maybe#(t)

- Type:
  - Maybe#(type t)
- Values:
  - tagged Invalid
  - tagged Valid x (where x is a value of type t)
- 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)

tagged union

- Maybe is a special type of tagged union

```
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

tagged union – Continued

- Values:
  - tagged <tag> value
- Pattern matching to get values:

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

- See BSV Reference Guide (on course website) for more examples of pattern matching

Reg#(t)
State element of BSV

- 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

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

// write to b_reg (needs to be done inside rule)
b_reg <= 7;
```

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 and write to the entire register 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 r – 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

- 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

Interfaces

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

```
interface MyInterface#(numeric type n);
    method ActionValue#(Bit#(b)) f();
    interface SubInterface s;
endinterface
```

Interface Methods

- Method
  - Returns value, doesn't change state
  - `method Bit#(32) peek_at_front();`
- Action
  - Changes state, doesn't return value
  - `method Action enqueue();`
- ActionValue
  - Changes state, returns value
  - `method ActionValue#(Bit#(32)) dequeue_front()`

Strong Typing

- The Bluespec Compiler throws errors if it can’t figure out a type
- Which of the following lines work?

```
Bit#(32) a = 7;  Bit#(8) small_b = 3;

let b = zeroExtend( small_b );

let a_plus_b = a + b;

Bit#(8) b_plus_fifty_truncated
    = truncate( b + 50 );
```

Quiz

Question 1

- What is the type of a?

```
Bit#(n) x = 1;
Bit#(m) y = 3;
let a = {x,y};
```

```
Bit#(TAdd#(n,m))
```

Question 2

- What is the type of b?

```
Bit#(n) x = 1;
Bit#(m) y = 3;
let a = {x,y};
let b = x + y;
```

Type Error! + expects inputs and outputs to all have the same type

Question 2 – BSC Error

```
Error: "File.bsv", line 10, column 9: …
  Type error at:
    y

  Expected type:
    Bit#(n)

  Inferred type:
    Bit#(m)
```

Question 3

- What is the type of c?

```
Bit#(8) x = 9;
let c = x[0];
```

```
Bit#(1)
```

Question 4

- What is the type of d?

```
Bit#(8) x = 9;
let d = zeroExtend(x);
```

Can’t tell, so the compiler gives a type error

Question 5

- What does this function do? How does it work?

```
function Bit#(m) resize(Bit#(n) x)
    Bit#(m) y = truncate(zeroExtend(x));
    return y;
endfunction
```

Produces a compiler error! zeroExtend(x) has an unknown type

Question 5 – Fixed

```
function Bit#(m) resize(Bit#(n) x)
    Bit#(TMax#(m,n)) x_ext;
    x_ext = zeroExtend(x);
    Bit#(m) y = truncate(x_ext);
    return y;
endfunction
```

Question 6

- What does this code do?

```
// mainQ, redQ, blueQ are FIFOs
// redC, blueC
let x = mainQ.first;
mainQ.deq;
if( isRed(x) )
    redQ.enq(x);
    redC <= redC + 1;
if( isBlue(x) )
    blueQ.enq(x);
    blueC <= blueC + 1;
```

Not what it looks like

Question 6 – Rewritten

```
let x = mainQ.first;
mainQ.deq;
if( isRed(x) )
    redQ.enq(x);
redC <= redC + 1;
if( isBlue(x) )
    blueQ.enq(x);
blueC <= blueC + 1;
```

Only the first action/expression after the if is done, that's why we have begin/end

Question 6 – Fixed

```
let x = mainQ.first;
mainQ.deq;
if( isRed(x) ) begin
    redQ.enq(x);
    redC <= redC + 1;
end
if( isBlue(x) ) begin
    blueQ.enq(x);
    blueC <= blueC + 1;
end
```