Constructive Computer Architecture

Tutorial 1

BSV objects and a tour of known BSV problems

Thomas Bourgeat

From Andy Wright's tutorial

Outline

- Tour of BSV objects
- Double write – the truth
- A useful construction in BSV
- Tour of problems

Only modules

```
module makerName(nameInterface);
    // State instances
//(submodules)

    // Rules (mind of the module)

    // Implem. of the interface
    // (methods)
endmodule
```

The primitive module

- Register
- Methods:
  - `method Action _write(t newvalue);`
  - `method t _read();`

More usual

Bluespec add notations:

- `myreg <= newvalue;`
- `// equiv`
- `myreg._write(newvalue);`
- `let currentvalue = myreg;`
- `// equiv`
- `let currentvalue = myreg._read();`

Total saving: 13 characters!

Remark: `myreg._read;` is valid as well.

Interfaces

- They are the types of modules
  - `Gcd instanceGcd <- mkGcd();`
  - `Bit#(32) sum = add(a,b);`
- Differences:
  - The meaning of = and <-?
  - The parameters?

Meaning of =

- It is just naming (with type)!

```
function Bit#(5) add4( Bit#(4) a, Bit#(4) b,
Bit#(1) c_in );
    Bit#(4) sum = 0;
    Bit#(1) carry = c_in;
    for (Integer i = 0; i < 4; i = i + 1) begin
        sum[i] = fa_sum(a[i], b[i], carry);
        carry = fa_carry(a[i], b[i], carry);
    end
    return {carry, sum};
Endfunction
```

What is carry?

Meaning of =

We really defined 5 names, and carry is just a name, that evolved with the unfolding and that at the end is:

```
carry=fa_carry(a[3],b[3],fa_carry(a[2],b[2],fa_carry(…)
))
Sum[0]= fa_sum(a[0],b[0],c_in);
sum[1]= fa_sum(a[1],b[1],fa_carry(a[0],b[0],c_in));
```

Abuse in naming

- Reg#(Bool) myReg <- mkReg(1);
  - What is myReg?
  - What is mkReg?
  - What is mkReg(1)?
- Wait, what, mkReg(1)?

Parameters for module

```
module mkMyReg#(Bit#(32) init)(Reg#(Bit#(32));
        Reg#(Bit#(32)) internal <- mkReg(init+42);
        method Bit#(32) _read();
                return internal;
        endmethod
        method Action _write(Bit#(32) newvalue);
                internal <= newvalue;
        endmethod
Endmodule
```

More parameters!

```
module mkTestBench#(Fft fft)();
        let fft_reference <- mkFftCombinational;
// …
        rule feed;
                //… generate some data d
                fft_reference.enq(d);
                fft.enq(d);
        endrule
endmodule
```

Meaning of <-

- What happens if I do:
  - Bit#(32) one = 1;
  - let whatIsThat = mkReg(one);
- whatIsThat is a recipe, it contains all the information to build a register of 32 bits, initialized with a one.
- <- not just naming, instantiating

And function?

- A convenient way to do parametrized naming, they are just helpers.
- They are mostly used for combinational circuits.

- Technically you can write functions that operates on other functions or even modules (modules are first class values)

Adder

```
Interface Adder;
        method Bit#(33) add(Bit#(32) a,Bit#(32) b);
endinterface

module mkAdder(Adder);
        method Bit#(33) add(Bit#(32) a,Bit#(32) b);
                return( addFct(a,b));
        endmethod
endmodule
```

A bit of concurrency

Something to be careful about, in the future.

Double write – the truth

- In class we saw that:

```
rule a;
      myReg <= 2;
endrule
rule b;
      myReg <= 1;
endrule
```

Are conflicting rules and can't fire together.

Double write – the truth

- But not actually, they can, so be careful!

- Rule 'b' shadows the effect of 'a' when they execute in the same clock cycle. Affected method calls:
  - myReg.write

What is true

- You cannot write twice a register within a rule, or within a method.
- Hence: You cannot call twice the same action method within a rule or within an other method.

A useful construction

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

Tour of problems

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
```

Known Problem aNd Solutions

```
Bit#(n) out;

for(Integer i=0; i<valueOf(n);i=i+1)
begin
      out[i] = fnc(…);
End

return out;
```

Uninitialized values and KPNS

```
'out' uses uninitialized value […]. If
this error is unexpected, please consult
KPNS #32.[…]

//Solution:
Bit#(n) out = 0;
```

Question 7 – The beast

```
//Alright
Bit#(32) unshifted = input;
Bit#(32) shifted = {0, unshifted[31:12]};

//Boom
Bit#(32) unshifted = input;
Integer i = 12;
Bit#(32) shifted = {0, unshifted[31:i]};
```

Question 7 – The beast

```
An ambigious type was introduce at […]
This type resulted from:
      The proviso Add#(a__,b__,32) introduced in
or at the following locations: […]
```

What is going on?

Question 7 – The beast

- Typechecking is happening before elaboration:
  - That explain why the behavior differs.
- Then what is the type of unshifted[32:i]?

Question 7 – The beast

- We would like it to be of type
  - TSub#(32,” i”)
  - But i is a value and not a type!
    - We have already valueOf(): numeric type ->Integer
    - We want typeOf(): Integer-> numeric type

Question 7 – The beast

- typeOf does not exist ☹
- So really, the "honest" type for this value unshifted[31:i] is not expressible in BSV typesystem.
- But we want to be able to do selection based on integers!

Question 7 – The beast

- Solution: Don't type it.
- Bit#(n) unshifted[31:i];
- (Or Bit#(i) unshifted[31:i]; to be confusing)
- But the {0,unshifted[31:i]} is a concatenation of two things of unknown sizes.

Question 7 – The beast

- The simple complex workaround:
- Bit#(32) shifted = unshifted[31:i];

- What if I wanted to add ones instead of 0 in front?
  - See other programming pattern