Constructive Computer Architecture

Tutorial 2

Debugging BSV and Typeclasses.
Outline

- Debugging BSV code
- Typeclasses and functional style.

And maybe conflict-Freeness
Software Debugging

Print Statements

- See a bug, not sure what causes it
- Add print statements
- Recompile
- Run
- Still see bug, but you have narrowed it down to a smaller portion of code
- Repeat with more print statements...
- Find bug, fix bug, and remove print statements
BSV Debugging
Display Statements

- See a bug, not sure what causes it
- Add display statements
- Recompile
- Run
- Still see bug, but you have narrowed it down to a smaller portion of code
- Repeat with more display statements...
- Find bug, fix bug, and remove display statements
BSV Display Statements

The $display() command is an action that prints statements to the simulation console

Examples:

- $display("Hello World!");
- $display("The value of x is %d", x);
- $display("The value of y is ", fshow(y));
Ways to Display Values

Format Specifiers

- `%d` – decimal
- `%b` – binary
- `%o` – octal
- `%h` – hexadecimal
- `%0d, %0b, %0o, %0h`

- Show value without extra whitespace padding
Ways to Display Values

fshow

- fshow is a function in the FShow typeclass
- It can be derived for enumerations and structures

Example:

typedef enum {Red, Blue} Colors deriving(FShow);
Color c = Red;
$display("c is ", fshow(c));

Prints "c is Red"
Two big families of bugs

- Functional bug
  - E.g. "a*d+b*c" instead of "a*d-b*c"

- Liveness bug
  - Scheduling issue
Functional bug

module mkTest (Det);
    method ActionValue#(Data) det (Data a, Data b, Data c, Data d);
        let res = a * d + b * c;
        $display("%d %d %d %d %d", a, b, c, d, res);
        return res;
    endmethod
Endmodule
Method for debugging liveness

Add $display(“Name rule”) in every rule and method of your design.

- You get to see what is firing.
  - There are probably not firing when they should:
    - Think about the implicit and explicit guards that would prevent a rule/method to fire.
    - If thinking is not enough?
Method for debugging liveness

If thinking is not enough:

- You can add an extra rule that just print the explicit guards of all the methods
Method for debugging liveness

module mkTest(Det);

[...]

    rule problematic (complexExpression);
    $display(“Problematic fire”);

[...] //Other stuff (methods called etc...)

endrule

endmodule
Method for debugging liveness

module mkTest(Det);

  [...]

  rule debugRule;
    $display("Guard is %b", complexExpression);
  endrule;

  rule problematic (complexExpression);
    $display("Problematic fire");
  endrule

endmodule
Liveness

If the guard is false when you expected it to be true:
- Well you just found your problem

If the guard is true:
- Check the implicit guards with the same technique:
Method for debugging liveness

module mkTest(Det);

[...]

rule debugRule;

$display("Guard is \%b",complexExpression);
endrule;

rule problematic (complexExpression);

$display("Problematic fire");

[...]

submodule1.meth1();

endrule
endmodule
Method for debugging liveness

module mkSubmodule1(Submodule1);
 rule debugRule;
 $display("Guard is %b", complexExpression);
 endrule;
 method Action meth1() if (complexExpression);
 [...] 
 endmethod
endmodule
Method for debugging liveness

Repeat until you are confident that the problem does not come from a false guard:

- Reminder: registers can always be written and read so they don’t pose problem for guards.
- Usually you don’t have to do that recursively because you already know that your submodules are corrects.
All my guards are good, still it does not work
All my guards are good, still it does not work.

Scheduling problem: an other rule is preventing the one I want to fire.
All my guards are good, still it does not work

module mkTest();
[
]
rule r1;
[
]
myfifo.enq(1);
endrule
rule r2;
[
]
myfifo.enq(2);
endrule
endmodule
Final note: be careful

module mkTest();
    [...]
    rule r1;
        [...]
        x <= y;
    endrule
    rule r2;
        [...]
        $display("x is", x);
        y <= 2;
    endrule
endmodule
Typeclasses
Typeclasses

A typeclass is a group of functions that can be defined on multiple types

Examples:

```haskell
typeclass Arith#(type t);
  function t +(t x, t y);
  function t -(t x, t y);
  // ... more arithmetic functions
endtypeclass

typeclass Literal#(type t);
  function t fromInteger(Integer x);
  function Bool inLiteralRange(t target, Integer literal);
endtypeclass
```
Instances

Types are added to typeclasses by creating instances of that typeclass

```haskell
instance Arith#(Bit#(n));
    function Bit#(n) +(Bit#(n) a, Bit#(n) b);
        return truncate(csa(a,b));
    endfunction

    function Bit#(n) -(Bit#(n) a, Bit#(n) b);
        return truncate(csa(a, -b));
    endfunction

    // more functions...
endinstance
```
Provisos restrict type variables used in functions and modules through typeclasses. If a function or module doesn’t have the necessary provisos, the compiler will throw an error along with the required provisos to add. The add1 function with the proper provisos is shown below:

```plaintext
function t add1(t x) provisos(Arith #(t), Literal #(t));
    return x + 1;
endfunction
```
There are some Typeclasses defined on numeric types that are only for provisos:

- **Add#( n1, n2, n3 )**
  - asserts that \( n_1 + n_2 = n_3 \)

- **Mul#( n1, n2, n3 )**
  - asserts that \( n_1 \times n_2 = n_3 \)

An inequality constraint can be constructed using free type variables since all type variables are non-negative:

- **Add#( n1, _a, n2 )**
  - asserts that \( n_1 + _a = n_2 \)
  - equivalent to \( n_1 \leq n_2 \) if \(_a\) is a free type variable
The Bits Typeclasses

- The Bits typeclass is defined below

```haskell
class Bits#(type t, numeric type tSz);
    function Bit#(tSz) pack(t x);
    function t unpack(Bit#(tSz) x);
endclass
```

- This typeclass contains functions to go between t and Bit#(tSz)
- `mkReg(Reg#(t))` requires t to have an instance of Bits#(t, tSz)
typedef enum { red, green, blue } Color deriving (Eq); // not bits

instance Bits#(Color, 2);
  function Bit#(2) pack(a x);
    if ( x == red ) return 0;
    else if ( x == green ) return 1;
    else return 2;
  endfunction
  function Color unpack(Bit#(2) y);
    if ( x == 0 ) return red;
    else if ( x == 1 ) return green;
    else return blue;
  endfunction
endinstance
Typeclasses Summary

- Typeclasses allow polymorphism across types
  - Provisos restrict modules type parameters to specified type classes

- Typeclass Examples:
  - Eq: contains == and !=
  - Ord: contains <, >, <=, >=, etc.
  - Bits: contains pack and unpack
  - Arith: contains arithmetic functions
  - Bitwise: contains bitwise logic
  - FShow: contains the fshow function to format values nicely as strings
Conflict-freeness.

Or be careful for what you wish
module mkCounter( Counter );
    Reg#(Bit#(8)) count <- mkReg(0);

    method Bit#(8) read;
        return count;
    endmethod

    method Action increment;
        count <= count + 1;
    endmethod

    method Action decrement;
        count <= count - 1;
    endmethod
endmodule

Can’t fire in the same cycle
Concurrent Design
A general technique

- Replace conflicting registers with EHRs
- Choose an order for the methods
- Assign ports of the EHR sequentially to the methods depending on the desired schedule

Method described in paper that introduces EHRs: “The Ephemeral History Register: Flexible Scheduling for Rule-Based Designs” by Daniel Rosenband
Up/Down Counter
Concurrent design: read < inc < dec

module mkCounter( Counter );
    Ehr#(3, Bit#(8)) count <- mkEhr(0);

method Bit#(8) read;
    return count[0];
endmethod

method Action increment;
endmethod

method Action decrement;
endmethod
endmodule
module mkCounter( Counter );
    Ehr#(2, Bit#(8)) count <- mkEhr(0);

    method Bit#(8) read;
        return count[0];
    endmethod

    method Action increment;
        count[0] <= count[0] + 1;
    endmethod

    method Action decrement;
    endmethod
endmodule

This design only needs 2 EHR ports now
Conflict-Free Design
A more or less general technique

- Replace conflicting Action and ActionValue methods with writes to EHRs representing method call requests
  - If there are no arguments for the method call, the EHR should hold a value of `Bool`
  - If there are arguments for the method call, the EHR should hold a value of `Maybe#(Tuple2#(TypeArg1, TypeArg2))` or something similar

- Create a canonicalize rule to handle all of the method call requests at the same time

- Reset all the method call requests to `False` or tagged `invalid` at the end of the canonicalize rule

- Guard method calls with method call requests
  - If there is an outstanding request, don’t allow a second one to happen
Up/Down Counter
Conflict-Free design – methods

module mkCounter( Counter );
    Reg#(Bit#(8)) count <- mkReg(0);
    Ehr#(2, Bool) inc_req <- mkEhr(False);
    Ehr#(2, Bool) dec_req <- mkEhr(False);
    // canonicalize rule on next slide
method Bit#(8) read = count;
method Action increment if(!inc_req[0]);
    inc_req[0] <= True;
endmethod
method Action decrement if(!dec_req[0]);
    dec_req[0] <= True;
endmethod
endmodule
module mkCounter (Counter);

// Reg and EHR definitions on previous slide
rule canonicalize;
if (inc_req[1] && !dec_req[1]) begin
  count <= count + 1;
end else if (dec_req[1] && !inc_req[1]) begin
  count <= count - 1;
end
inc_req[1] <= False;
dec_req[1] <= False;
endrule
// methods on previous slide
endmodule
Well it’s morally broken

module mkTest();
    Reg#(Bit#(8)) r <- mkReg(0);
    let myCounter <- mkCounter();
    rule r1;
        $display("r");
        myCounter.increment();
    endrule
    rule r2;
        r <= myCounter.read();
    endrule
    rule display;
        $display(r);
    endrule
endmodule

We can schedule read after increment, but read will always see old Values because it is scheduled before canonicalize.
Fix: but read< {inc,dec}.

module mkCounter( Counter );
    Reg#(Bit#(8)) count <- mkReg(0);
    Ehr#(2, Bool) inc_req <- mkEhr(False);
    Ehr#(2, Bool) dec_req <- mkEhr(False);
    // canonicalize rule on next slide
method Bit#(8) read if(!inc_req[0] && !dec_req[0]) = count;
method Action increment if(!inc_req[0]);
    inc_req[0] <= True;
endmethod
method Action decrement if(!dec_req[0]);
    dec_req[0] <= True;
endmethod
Interesting questions

Is it possible to write a CF counter?

Is it possible to give an algorithm that will always make a module conflict free, but a non broken one.