Constructive Computer Architecture

Tutorial 2

Advanced BSV

Andy Wright
6.175 TA

BSV Review

Last Tutorial

- Types
  - Bit, UInt/Int, Bool
  - Vector, Tuple, Maybe
  - struct, enum, tagged union
- Register of Vector vs Vector of Registers
  - Partial writes lead to the double write error
- Modules and Interfaces
- Quiz

BSV Review

Expressions vs. Actions

- Expressions
  - Have no side effects (state changes)
  - Can be used outside of rules and modules in assignments
- Actions
  - Can have side effects
  - Can only take effect when used inside of rules
  - Can be found in other places intended to be called from rules
    - Action/ActionValue methods
    - functions that return actions

BSV Review

Valid Rule

- A Rule is valid if there is a valid total ordering of all the method calls in the rule that meets:
  - Syntax constraints
  - Reg constraints
  - EHR constraints
  - Module conflict matrix constraints
- Double write errors exist because register constraints prevent two calls to the _write method from happening in the same rule

BSV Review

Scheduling Circuitry

- Each rule has a CAN_FIRE and a WILL_FIRE signal in hardware
  - CAN_FIRE is true if the explicit and implicit guards are both true
  - WILL_FIRE is true if the rule is firing in the current cycle
- When does WILL_FIRE != CAN_FIRE?
  - When there are conflicts between rules
  - If CAN_FIRE is true and WILL_FIRE is false, there is a rule that has WILL_FIRE as true and it conflicts with the current rule
- If all rules are conflict free, WILL_FIRE = CAN_FIRE

The compiler will give a warning if WILL_FIRE != CAN_FIRE

BSV Review

Valid Concurrent Rules

- A set of rules r_i can fire concurrently if there exists a total order between the rules such that all the method calls within each of the rules can happen in that given order
    - Rules r_1, r_2, r_3 can fire concurrently if there is an order r_i, r_j, r_k such that $r_i < r_j$, $r_i < r_k$, and $r_i < r_k$ are all valid

Design Example

An Up/Down Counter

Up/Down Counter

Design example

- Some modules have inherently conflicting methods that need to be concurrent
    - This example will show a couple of ways to handle it

```
interface Counter;
    Bit#(8) read;
    Action increment;
    Action decrement;
endinterface
```

Inherently conflicting

Up/Down Counter

Conflicting design

```
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;
        count[1] <= count[1] + 1;
    endmethod
    method Action decrement;
        count[2] <= count[2] - 1;
    endmethod
endmodule
```

These two methods can use the same port

Up/Down Counter

Concurrent design: read < inc < dec

```
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;
        count[1] <= count[1] - 1;
    endmethod
endmodule
```

This design only needs 2 EHR ports now

Conflict-Free Design

A 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
```

Up/Down Counter

Conflict-Free design – canonicalize rule

```
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
```

Synthesis Boundary

Synthesis Boundary

- A synthesis boundary is an attribute on a module that causes the module to be compiled separately
- A synthesis boundary looks like this:

```
(* synthesize *)
Module mkMyModule( MyModuleIFC );
```

- A synthesis boundary can only be placed over a module with:
    - No type parameters in its interface
    - No parameters in the module’s constructor that can’t be converted to bits (no interfaces can be passed in)

Synthesis Boundary

Guard Logic

- Synthesis boundaries simplifies guard logic

```
method Action doAction( Bool x );
    if( x ) begin
        <a> when p;
    end else begin
        <a> when q;
    end
endmethod
```

Lifted guard without synthesis boundary: (!x || p) && (x || q)

Lifted guard with synthesis boundary: p && q

Synthesis boundaries do not allow inputs to be in guards

Synthesis Boundary

Guard Logic

- Synthesis boundaries simplifies guard logic

```
rule doStuff;
    let x <- m.getResult;
    if( isValid(x) ) begin
        <a> when p;
    end else begin
        <a> when q;
    end
endmethod
```

Lifted guard without synthesis boundary on m:
(!isValid(m.result) || p) && (isValid(m.result) || q)

Lifted guard with synthesis boundary: p && q

Synthesis boundaries do not allow outputs of ActionValue methods to be in guards

Synthesis Boundary

Why is it different?

- When a module has no synthesis boundary, its methods are effectively inlined in the rules
  - Long compilation times – A module used 64 times is compiled 64 times!
  - Aggressive guards – Rules will be able to fire more often
- When a module has a synthesis boundary, it is compiled separately and other modules instantiate copies of it
  - Shorter compilation times
  - Conservative guards
  - Separate hardware module
  - Required for top-level module

Advanced Types

Types used in function definitions

- Example from C/C++:

```
int add1( int x ) {
    return x + 1;
}
```

- The type definition of add1 says it takes values of type int and returns values of type int
  - Types are a collection of values
  - Defining a function to use a type restricts the values you can use in the function
  - banana is not a value in the collection int, so add1(banana) is not valid

Types *variables* used in function definitions

- Example from BSV:

```
function t add1( t x );
    return x + 1;
endfunction
```

- The type used for this function is a type variable (t)
  - It says the input and the output are values of the same type
  - Some types of values make sense for this function, but some types don't
    - do make sense: Integer, Int#(n), Bit#(n)
    - don't make sense: String, Fruit

How do you describe the collection of types that t can belong to?

Types *variables* used in function definitions

- Lets break down the example:

  ```
  x + 1
  ```

  is actually

  ```
  x + fromInteger(1)
  ```

- add1 uses the functions

  ```
  t \+(t x, t y)
  ```

  and

  ```
  t fromInteger(Integer x)
  ```

  - t needs to belong to the collection of types that has + defined on it, and it needs to belong to the collection of types that has fromInteger defined on it
- What is the collection of types that has + defined on it?
- What is the collection of types that has fromInteger defined on it?
- What are these collections of types?
  - *Typeclasses*!

Typeclasses

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

```
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

```
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

- 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:

```
function t add1(t x) provisos(Arith#(t), Literal#(t));
    return x + 1;
endfunction
```

Special Typeclasses for Provisos

- There are some Typeclasses defined on numeric types that are only for provisos:
- `Add#( n1, n2, n3 )`
  - asserts that n1 + n2 = n3
- `Mul#( n1, n2, n3 )`
  - asserts that n1 * n2 = n3
- An inequality constraint can be constructed using free type variables since all type variables are non-negative
  - `Add#( n1, _a, n2 )`
    - asserts that n1 + _a = n2
    - equivalent to n1 <= n2 if _a is a free type variable

The Bits Typeclasses

- The Bits typeclass is defined below

```
typeclass Bits#(type t, numeric type tSz);
    function Bit#(tSz) pack(t x);
    function t unpack(Bit#(tSz) x);
endtypeclass
```

- This typeclass contains functions to go between t and Bit#(tSz)
- mkReg(Reg#(t)) requires t to have an instance of Bits#(t, tSz)

Custom Bits#(a,n) instance

```
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

Quiz 2

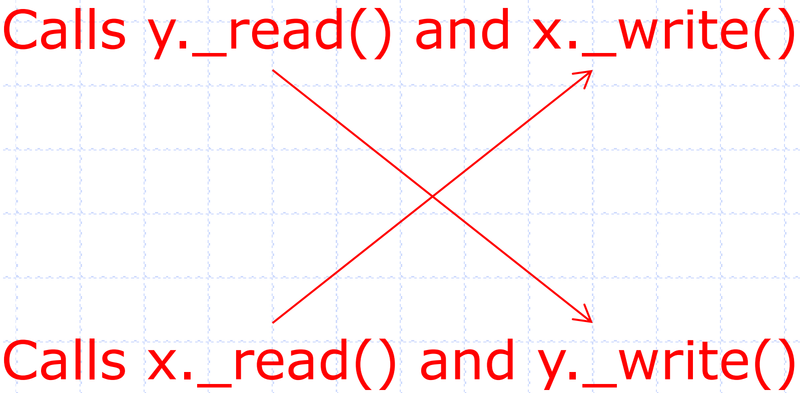
Fun with BSV’s scheduling

Question 1

- What is the schedule for the following rules?

```
rule r1;
    x <= y;
endrule

rule r2;
    y <= x;
endrule
```



Calls y._read() and x._write()

Calls x._read() and y._write()

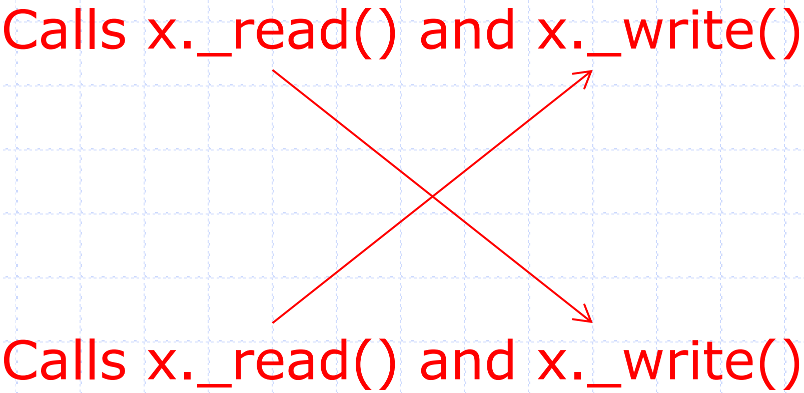
r1 C r2, so the two rules will never fire in parallel

Question 2

- What is the schedule for the following rules?

```
rule increment;
    x <= x + 1;
endrule

rule decrement;
    x <= x - 1;
endrule
```



Calls x._read() and x._write()

Calls x._read() and x._write()

increment C decrement, so the two rules will never fire in parallel

Question 3

- What is the schedule for the following rules?

```
rule increment;
    x <= x + 1;
endrule

rule reset;
    x <= 0;
endrule
```

Calls x._read() and x._write()

Calls x._write()

increment < reset

Question 3 – I'm so sorry

- It turns out the *double write error* does not exist between rules
  - Two rules can write to the same register in the same cycle assuming there are no other conflicts
- In this case, the compiler would warn about *Action shadowing*
  - The effects of increment are *shadowed* by reset
- Why is this so?
  - I have no idea
  - Idea: Its an easy optimization to run more rules in a single cycle

Conflict Matrix of Primitive modules: Registers and EHRs

Slide from lecture 6

Register

	reg.r	reg.w
reg.r	CF	<
reg.w	>	C*

*No conflict between rules

EHR

	EHR.r0	EHR.w0	EHR.r1	EHR.w1
EHR.r0	CF	<	CF	<
EHR.w0	>	C	<	<
EHR.r1	CF	>	CF	<
EHR.w1	>	>	>	C

EHR write ports still conflict between rules

Question 4

- What is the schedule for the following rules?

```
// q1, q2, and q3 are FIFOs
rule r1;
    q1.enq( f1(q3.first) ); q3.deq;
endrule
rule r2;
    q2.enq( f2(q1.first) ); q1.deq;
endrule
rule r3;
    q3.enq( f3(q2.first) ); q2.deq;
endrule
```

It depends on the type of FIFOs

Question 4.5

- What is the schedule for the following rules?

```
// q1, q2, and q3 are all pipeline FIFOs
rule r1;
    q1.enq( f1(q3.first) ); q3.deq;
endrule
rule r2;
    q2.enq( f2(q1.first) ); q1.deq;
endrule
rule r3;
    q3.enq( f3(q2.first) ); q2.deq;
endrule
```

r1 < r3 < r2 < r1

The compiler will introduce a conflict to break this cycle.

Question 5

- What type of FIFOs allow these rules to fire concurrently?

```
// q1, q2, and q3 are FIFOs
rule r1;
    q1.enq( f1(q3.first) ); q3.deq;
endrule
rule r2;
    q2.enq( f2(q1.first) ); q1.deq;
endrule
rule r3;
    q3.enq( f3(q2.first) ); q2.deq;
endrule
```

q1: pipeline
q2: pipeline
q3: bypass

q1: pipeline
q2: bypass
q3: bypass

q1: pipeline
q2: pipeline
q3: CF

Many different combinations