Constructive Computer Architecture
Tutorial 2

Advanced BSV

Sizhuo Zhang
6.175 TA

EHR and Scheduling

- Design example
  - Up/down counter

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

Up/Down Counter
Conflict Design

```
module mkCounter( Counter );
    Reg#(Bit#(8)) count <- mkReg(0);

    method Bit#(8) read;
        return count;
    endmethod
    method Action increment(Bit#(8) x);
        count <= count + x;
    endmethod
    method Action decrement;
        count <= count – 1;
    endmethod
endmodule
```

increment conflicts with decrement

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; // use port 0
        return count[0];
    endmethod
    method Action increment(Bit#(8) x);; // use port 1
        count[1] <= count[1] + x;
    endmethod
    method Action decrement; // use port 2
        count[2] <= count[2] – 1;
    endmethod
endmodule
```

We could use Ehr#(2, Bit#(8)), but not necessary.

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 request EHRs 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, Maybe#(Bit#(8))) inc_req <- mkEhr(Invalid);
   Ehr#(2, Bool) dec_req <- mkEhr(False);
   // canonicalize rule on next slide
   method Bit#(8) read = count;
   method Action increment(Bit#(8) x) if(!isValid(inc_req[0]));
       inc_req[0] <= Valid (x);
   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;
        let count_nxt = count;
        if(isValid(inc_req[1]))
            count_nxt = count_nxt + fromMaybe(?, inc_req[1]);
        if(dec_req[1])
            count_nxt = count_nxt - 1;

        count <= count_nxt
        inc_req[1] <= Invalid;
        dec_req[1] <= False;
    endrule
    // methods on previous slide
endmodule
```

This is basically the solution to Lab 4.

Synthesis Boundary

- Module without synthesis boundary
  - Methods are inlined
  - Pros: Aggressive guard lifting (if … else …)
  - Cons: long compile time if instantiated many times
- Module with synthesis boundary

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

  - The module will be compiled separately
  - The module becomes a black box
  - Pros: shorter compile time, no inlining
  - Cons: conservative guards

Synthesis Boundary

Guard Logic (Example)

- 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 || g(p)) && (x || g(q))
- Lifted guard with synthesis boundary: p && q
- Synthesis boundaries do not allow inputs to be in guards. There are other restriction…

Synthsis Boundary

- 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)
- Takeaway
  - Synthesis boundary may affect guard & scheduling
  - Faster compilation

Typeclass

- Motivating example: parametrized add

```
function t adder(t a, t b);
  return a + b;
endfunction
```

  - Some type t can be added: `Bit#(n)`, `Int#(n)`
  - Some cannot: enum, struct, tagged union
- Typeclass
  - If a type blongs to a typeclass, then certain operations can be performed on this type
  - Arith: +, -, *, /, %, negate
  - Bits: pack, unpack

Typeclass

- Define a typeclass

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

  - If types a and n are in typeclass Bits,
  - then we can do pack & unpack with them

Instance

- Types are added to typeclasses by creating instances of that typeclass

```
typedef enum { Red, Green, Blue } Color deriving (Eq);
// no deriving(Bits)
instance Bits#(Color, 2);
    function Bit#(2) pack(Color x);
        return x==Red ? 3 : x==Green ? 2 : 1;
    endfunction
    function Color unpack(Bit#(2) x);
        return x==3 ? Red : x==2 ? Green : Blue;
    endfunction
endinstance
```

- deriving: do above process in a default way

Provisos

- Tell compiler that type t can do “+”
  - Add provisos (compile error without provisos)

```
function t adder(t a, t b) provisos(Arith#(t));
  return a + b;
endfunction
```

- Provisos
  - Tell compiler additional information about the parametrized types
  - Compiler can type check based on the info

Typeclass Takeaway

- Typeclasses allow polymorphism across types
  - Provisos restrict modules type parameters to specified type classes
- Typeclass Examples (BSV Reference guide Section B.1)
  - Eq: ==, !=
  - Ord: <, >, <=, >=, min, max, compare
  - Bits: pack, unpack
  - Arith: +, -, *, /, %, negate
  - Literal: fromInteger, inLiteralRange
  - FShow: fshow (format values nicely as strings)

RISC-V Processor Code