Overview

- BSV reviews/notes
- Guard lifting
- EHRs
- Scheduling
- Lab 3
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

Variable vs. States

**Variables**
- Variables are used to name intermediate values
- Do not hold values over time
- Variable are **bound** to values
  - Statically elaborated

```vhdl
Bit#(32) firstElem = aQ.first();
rule process;
    aReg <= firstElem;
endrule
```
Scoping

- Any use of an identifier refers to its declaration in the nearest textually surrounding scope.

```verilog
module mkShift (Shift#(a));
    function Bit#(32) f();
        return fromInteger(valueOf(a))<<2;
    endfunction
    rule process;
        aReg <= f();
    endrule
endmodule
```

- Functions can take variables from surrounding scope.

Guard Lifting

- Last Time: implicit/explicit guards
  - But there is more to it when there are conditionals (if/else) within a rule.
- Compiler option `-aggressive-conditions` tells the compiler to peek into the rule to generate more aggressive enable signals.
  - Almost always used.
Guard Examples

```vhdl
rule process;
  if (aReg==True)
    aQ.deq();
  else
    bQ.deq();
  $display("fire");
endrule

rule process;
  if (aQ.notEmpty)
    aQ.deq();
  $display("fire");
endrule
```

(aReg==True && aQ.notEmpty) ||
(aReg==False && bQ.notEmpty) ||

(aQ.notEmpty && aQ.notEmpty) ||
(laQ.notEmpty) ➔ Always fires

---

Ephemeral History Register (EHR)

Encode a notion of “priority” when there are concurrent writes/reads
Design Example

An Up/Down Counter

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

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

Inherently conflicting
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
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

Up/Down Counter
Concurrent design: read < inc < dec

```verilog
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
```
Up/Down Counter
Concurrent design: read < inc < dec

Feb 19, 2016  http://csg.csail.mit.edu/6.375  T02-15
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 $r_i < r_j$, $r_j < r_k$, and $r_i < r_k$ are all valid.

Concurrently executable rules are scheduled to fire in the same cycle.
In HW, states change only at clock edges.
Rule Scheduling Intuitions

Can these rules fire in the same cycle?

```
rule r1 (a);
  x <= 1;
endrule

rule r2 (!a);
  x <= 2;
endrule
```

No, guards are mutually exclusive

---

Rule Scheduling Intuitions

Can these rules fire in the same cycle?

```
rule r1;
  y <= 1;
endrule

rule r2;
  x <= 1;
endrule
```

Yes, methods are unrelated (conflict free)
Rule Scheduling Intuitions

Is it legal? Can these rules fire in the same cycle?

\begin{verbatim}
rule increment;
    x <= x + 1;
endrule

rule decrement;
    x <= x - 1;
endrule
\end{verbatim}

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

Feb 19, 2016
http://csg.csail.mit.edu/6.375
T02-21

Rule Scheduling Intuitions

Can these rules fire in the same cycle?

\begin{verbatim}
rule r1;
    x <= y;
endrule

rule r2;
    y <= x;
endrule
\end{verbatim}
r1 C r2, so the two rules will never fire in parallel

Feb 19, 2016
http://csg.csail.mit.edu/6.375
T02-22
Lab 3: Overview

- Completing the audio pipeline:
  - PitchAdjust
  - FromMP, ToMP

Converting C to Hardware

- Think about what states you need to keep
- Loops in C are sequentially executed; loops in BSV are statically elaborated
  - Unrolled
Fixed Point

\[
10.01_2 = 1 \times 2^1 + 0 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2}
\]

```
typedef struct {
    Bit#(isize) i;
    Bit#(fsize) f;
} FixedPoint#(numeric type isize, numeric type fsize);
```

Fixed Point Arithmetic

- **Useful FixedPoint functions:**
  - `fxptGetInt`: extracts integer portion
  - `fxptMult`: full precision multiply
  - `*`: full multiply followed by rounding/saturation to the output size

- **Other useful bit-wise functions:**
  - `truncate`, `truncateLSB`
  - `zeroExtend`, `extend`
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

Feb 19, 2016
http://csg.csail.mit.edu/6.375
T02-27

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));

Feb 19, 2016
http://csg.csail.mit.edu/6.375
T02-28
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
- FixedPoint is also a FShow typeclass

Example:

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

Prints “c is Red”
Warning about $display

- $display is an Action within a rule
- Guarded methods called by $display will be part of implicit guard of rule

```plaintext
rule process;
    if (aQ.notEmpty)
        aQ.deq();
    $display("first elem is %x", aQ.first);
endrule
```

Extra Stuff
BSV Debugging
Waveform Viewer

- Simulation executables can dump VCD waveforms
  - ./simMyTest -V test.vcd
- Produces test.vcd containing the values of all the signals used in the simulator
  - Not the same as normal BSV signals
- VCD files can be viewed by a waveform viewer
  - Such as gtkwave
- The signal names and values in test.vcd can be hard to understand
  - Especially for structures and enumerations

Step 1
Generate VCD File

- Run ./simTestName -V test.vcd
Step 2
Open Bluespec GUI

Run “bluespec fifo.bspec”

Note, to run the GUI remotely, you need to SSH into the servers with the “ssh -X” command

For the fifo lab, fifo.bspec can be found in

Step 3
Set top module name

Open project options
Step 3
Set top module name

Set the top module name to match the compiled module in TestBench.bsv

Step 4
Open Module Viewer
Step 4
Open Module Viewer

Step 5
Open Wave Viewer
Step 5
Open Wave Viewer

Step 6
Open Wave Viewer
Step 6
Add Some Signals
Step 7
Look at the Waveforms

Types
Human readable
value names
Step 7
Look at the Waveforms

Step 8
Add Some More Signals
Step 8
Add Some More Signals

Step 9
Add Rules Too
Step 9
Add Rules Too

```plaintext
<table>
<thead>
<tr>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLK [Clock] = True</td>
</tr>
<tr>
<td>fifo_enq [Bit#2] = True</td>
</tr>
<tr>
<td>fifo_deq [Bit#2] = False</td>
</tr>
<tr>
<td>fifo_empty [Bool] = False</td>
</tr>
<tr>
<td>fifo_full [Bool] = True</td>
</tr>
<tr>
<td>WILL_FIRE_RL fifo_canonicalize [Bool] = True</td>
</tr>
<tr>
<td>CAN_FIRE_RL fifo_canonicalize [Bool] = True</td>
</tr>
</tbody>
</table>
```