6.375 Supplemental Resource

BSV and Labs 2-3

http://csg.csail.mit.edu/6.375
Overview

- Basic BSV
  - Slides 3-26
- BSV related to Labs 2 and 3
  - Slides 27-43
- More BSV types
  - Slides 44-55
Basic Bluespec
Modules

◆ Interfaces
  ■ Methods provide a way for the outside world to interact with the module

◆ State elements and sub-modules
  ■ Registers, FIFOs, BRAMs, FIR filters (Lab 1)

◆ Rules
  ■ *Guarded atomic actions to* describe how the state elements should change
Part 1: Declare Interfaces

- Contain methods for other modules to interact with the given module
  - Interfaces can also contain sub-interfaces

```vhdl
interface MyIfc;
  method ActionValue#(Bit#(32)) f();
  interface SubIfc s;
endinterface
```

- Special interface: Empty
  - No method, used in testbench

```vhdl
module mkTb (Empty);
module mkTb(); // () are necessary
```

http://csg.csail.mit.edu/6.375
Interface Methods

- **Value**
  - Returns value, doesn’t change state
  - method Bit#(32) first;

- **Action**
  - Changes state, doesn’t return value
  - method Action enq(Bit#(32) x);

- **ActionValue**
  - Changes state, returns value
  - method ActionValue#(Bit#(32)) deqAndGet;
Calling Interface Methods

- **Value**: Call inside or outside of a rule since it only returns a value
  - Bit#(32) a = aQ.first;
  - Bit#(32) sum = aQ.first + aQ.first + bQ.first;

- **Action**: Can call *once* within a rule
  - aQ.enq(sum);

- **ActionValue**: Can call *once* within a rule
  - Use "<-" operator *inside* a rule to apply the action and return the value
  - Bit#(32) prod <- multiplier.deqAndGet;
Part 2: Defining a Module

- module mkAdder( Adder#(32) );
  - Adder#(32) is the interface
- Module can be parametrized
  - module name#(params)(args ..., interface);

module mkMul#(Bool signed)(Adder#(n) a, Mul#(n) x);
Part 3: Instantiating sub-modules

- Examples: Registers, FIFOs, RAMs, FIR filter (from Lab 1)

- Instantiation:
  - The " <- " outside a rule is used to instantiate a module
  - MyIfc instOfModule <- mkModule();
  - Reg#(Bit#(32)) counter <- mkReg(0);
  - FIFO#(Uint#(32)) aQ <- mkFIFO();
Part 4: Rules

- Rules describe the actions to be applied atomically
  - Modifies state
- Rules have guards to determine when they can fire
  - Implicit or explicit
Rule Execution

One rule at a time:

- Choose an *enabled* rule
- Apply *all* of the *actions* of the rule
- Repeat

Conceptually rules execute one at a time in global order, but compiler aggressively *schedules* multiple rules to execute in the same clock cycle

- Scheduling will be covered in detail in upcoming lectures

http://csg.csail.mit.edu/6.375
Hello World

```plaintext
module mkHelloWorld (Empty);
    rule sayhello (True);
        $display("hello, world");
    endrule
endmodule
```

What does this do?

- Print “hello, world” infinitely
module mkHelloWorldOnce ();
    Reg#(Bool) said <- mkReg(False);
rule sayhello (!said);
    $display("hello, world");
    said <= True;
endrule

rule goodbye (said);
    $finish();
endrule
endmodule
When *can* a rule fire?

- Guard is true (explicit)
- *All* actions/methods in rule are ready (implicit)

```verilog
rule doCompute if (started);
    Bit#(32) a = aQ.first(); // aQ is a FIFO
    Bit#(32) b = bQ.first(); // bQ is a FIFO
    aQ.deq();
    bQ.deq();
    outQ.enq( {a, b} ); // outQ is a FIFO
endrule
```

*Will* it fire?

- That depends on scheduling
Part 5: Implement Interface

```verilog
interface MyIfc#(numeric type n);
    method ActionValue#(Bit#(n)) f();
interface SubIfc#(n) s;
endinterface

module mkDut(MyIfc#(n));
    // Methods and implementation
    method ActionValue#(Bit#(n)) f();
    ....
endmethod
interface SubIfc s; // no param “n”
    // Methods of SubIfc
endinterface
endmodule
```

Methods, just like rules, have can have implicit and explicit guards
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 are used to name intermediate values
- Do not hold values over time
- Variables are \textbf{bound} to values
  - Statically elaborated

```vhd
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.

Bit#(32) a = 1;
rule process;
    aReg <= a;
endrule

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.

http://csg.csail.mit.edu/6.375
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

```
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) ||
(!aQ.notEmpty) \Rightarrow Always fires
```
Vector Sub-interface

Sub-interface can be vector

```
interface VecIfc#(numeric type m, numeric type n);
    interface Vector#(m, SubIfc#(n)) s;
endinterface
```

```
Vector#(m, SubIfc) vec = ?;
for(Integer i=0; i<valueOf(m); i=i+1) begin
    // implement vec[i]
end
VecIfc ifc = (interface VecIfc;
    interface Vector s = vec; // interface s = vec;
Endinterface);
```

BSV reference guide Section 5

http://csg.csail.mit.edu/6.375
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

http://csg.csail.mit.edu/6.375
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
  - *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
```
Useful Labs 2 and 3 Topics
Vector

Type:
- `Vector#(numeric type size, type data_type)`

Values:
- `newVector()`, `replicate(val)`

Functions:
- Access an element: `[]`
- Range of vectors: `take`, `takeAt`
- Rotate functions
- Advanced functions: `zip`, `map`, `fold`

Can contain registers or modules

Must have `import Vector::*;` in BSV file

http://csg.csail.mit.edu/6.375
Vectors: Example

FIFO#(Vector#(FFT_POINTS, ComplexSample))
    inputFIFO <- mkFIFO();

Instantiating a single FIFO, holding vectors of samples

Vector#(TAdd#(1,FFT_LOG_POINTS),Vector#(FFT_POINTS, ComplexSample)) stage_data = newVector();

Declaring a vector of vectors

for (Integer i=0; i < 10; i=i+1) begin
    stage_data[i][0] = func(i);
end

Assigning values to a vector
Reg and Vector

Register of Vectors
- Reg#( Vector#(32, Bit#(32)) ) rfile;
- rfile <- mkReg( replicate(0) );

Vector of Registers
- Vector#( 32, Reg#(Bit#(32)) ) rfile;
- rfile <- replicateM( mkReg(0) );
- Similarly:
  fifoVec <- replicateM( mkFIFO() );

Each has its own advantages and disadvantages
Partial Writes

*Reg#(Bit#(8)) r;*
- r[0] <= 0 counts as a read & write to the entire reg r
  - let r_new = r; r_new[0] = 0; r <= r_new

*Reg#(Vector#(8, Bit#(1))) r*
- Same problem, r[0] <= 0 counts as a read and write to the entire register
- r[0] <= 0; r[1] <= 1 counts as two writes to register
  - double write problem

*Vector#(8,Reg#(Bit#(1))) r*
- r is 8 different registers
- r[0] <= 0 is only a write to register r[0]
- r[0] <= 0 ; r[1] <= 1 is not a double write problem
Polymorphic Interfaces

Declaring a polymorphic interface

```verilog
interface DSP#(
  numeric type w,
  type dType
);
  method Action putSample(Bit#(w) a, dType b);
  method Vector#(w, dType) newVal();
endinterface
```

Using polymorphic interfaces

```verilog
module mkDSP ( DSP#(w, dType) );
  Reg#(Bit#(w)) aReg <- mkReg(0);
  Reg#(dType) bReg <- mkRegU();
...
endmodule
```

Instantiating a module with polymorphic ifc

```verilog
module mkTb();
  DSP#(8, UInt#(32)) dspInst <- mkDSP();
endmodule
```
Get/Put Interfaces

- Pre-defined interface in BSV
- Provides a simple handshaking mechanism for getting data from a module or putting data into it

```haskell
import GetPut::*

interface Get#{type t};
  method ActionValue#{t} get();
endinterface

interface Put#{type t};
  method Action put(t x);
endinterface
```

http://csg.csail.mit.edu/6.375
Using Get/Put Interfaces

```verilog
import FIFO::*;
import GetPut::*;
interface FooIfc;
    interface Put#(Bit#(32)) request;
    interface Get#(Bit#(32)) response;
endinterface

module mkFoo (FooIfc);
    FIFO#(Bit#(32)) reqQ <- mkFIFO;
    FIFO#(Bit#(32)) respQ <- mkFIFO;
    interface Put request;
        method Action put (Bit#(32) req);
            reqQ.enq (req);
        endmethod
    endinterface

    interface Get response;
        method ActionValue#(Bit#(32)) get ();
            let resp = respQ.first;
            respQ.deq;
            return resp;
        endmethod
    endinterface
endmodule
```

http://csg.csail.mit.edu/6.375
Get/Put with FIFOs

```verilog
import FIFO::*;
import GetPut::*;
interface FooIfc;
    interface Put#(Bit#(32)) request;
    interface Get#(Bit#(32)) response;
endinterface
module mkFoo (FooIfc);
    FIFO#(Bit#(32)) reqQ <- mkFIFO;
    FIFO#(Bit#(32)) respQ <- mkFIFO;
    interface request = toPut(reqQ);
    interface response = toGet(respQ);
endmodule
```

http://csg.csail.mit.edu/6.375
Server Interfaces

Extension of Get/Put

```haskell
import ClientServer::*;

interface Server #(type req_t, type rsp_t);
    interface Put#(req_t) request;
    interface Get#(rsp_t) response;
endinterface
```
Server Interfaces

```verilog
import FIFO::*;
import GetPut::*;
import ClientServer::*;

typedef Server#(Bit#(32), Bit#(32)) FooIfc;

module mkFoo (FooIfc);

    FIFO#(Bit#(32)) reqQ <- mkFIFO;
    FIFO#(Bit#(32)) respQ <- mkFIFO;

    interface Put request = toPut(reqQ);
    interface Get response = toGet(respQ);

endmodule
```

http://csg.csail.mit.edu/6.375
Provisos

Tell compiler that type \( t \) can do “+”

- Add provisos (compile error without provisos)

```plaintext
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
Type Conversions

- **Numeric type: type parameters**
  - Often natural numbers
  - Bit#(w); Vector#(n, Uint#(w))

- **Integers**
  - Not synthesizable in hardware (vs Int#())
  - Often used in static elaboration (for loops)

- **Numeric type -> Integer:** `valueOf(w)`
- **Integer -> Numeric type: not possible**
- **Integer -> Bit#(), Int#() etc.:** `fromInteger(i)`
- **Numeric type -> Bit#(), Int#() etc:**
  `fromInteger(valueOf(w))`

http://csg.csail.mit.edu/6.375
Lab 3: Overview

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

http://csg.csail.mit.edu/6.375
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

http://csg.csail.mit.edu/6.375
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

http://csg.csail.mit.edu/6.375
More Types
Bit#(numeric type n)

- **Literal values:**
  - Decimal: 0, 1, 2, ... (each have type Bit#(n))
  - Binary: 5'b01101, 2'b11
  - Hex: 5’hD, 2’h3, 16’h1FF0

- **Common functions:**
  - Bitwise Logic: |, &, ^, ~, etc.
  - Arithmetic: +, -, *, %, etc.
  - Indexing: a[i], a[3:1]
  - Concatenation: {a, b}
  - truncate, truncateLSB
  - zeroExtend, signExtend
Bool

- Literal values:
  - True, False

- Common functions:
  - Boolean Logic: ||, &&, !, ==, !=, etc.

- All comparison operators (==, !=, >, <, >=, <=) return Bools
Int#(n), UInt#(n)

**Literal values:**
- Decimal:
  - 0, 1, 2, ... (Int#(n) and UInt#(n))
  - -1, -2, ... (Int#(n))

**Common functions:**
- Arithmetic: +, -, *, %, etc.
  - Int#(n) performs signed operations
  - UInt#(n) performs unsigned operations
- Comparison: >, <, >=, <=, ==, !=, etc.
Constructing new types

- Renaming types:
  - typedef

- Enumeration types:
  - enum

- Compound types:
  - struct
  - vector
  - maybe
  - tagged union
typedef

- **Syntax:**
  - `typedef <type> <new_type_name>;`

- **Basic:**
  - `typedef 8 BitsPerWord;`
  - `typedef Bit#(BitsPerWord) Word;`
    - Can’t be used with parameter: `Word#(n)`

- **Parameterized:**
  - `typedef Bit#(TMul#(BitsPerWord,n)) Word#(numeric type n);`
    - Can’t be used *without* parameter: `Word`
enum

typedef enum {Red, Blue} Color deriving (Bits, Eq);

- Creates the type Color with values Red and Blue
- Can create registers containing colors
  - Reg#(Color)
- Values can be compared with == and !=
Elements from MemReq $x$ can be accessed with $x$.addr, $x$.data, $x$.wren

Struct Expression
- $X = \text{MemReq}\{\text{addr: 0, data: 1, wren: True}\}$
struct

typedef struct {
    t a;
    Bit#(n) b;
} Req#(type t, numeric type n) deriving (Bits, Eq);

Parametrized struct
Tuple

Types:
- Tuple2#(type t1, type t2)
- Tuple3#(type t1, type t2, type t3)
- up to Tuple8

Construct tuple: tuple2( x, y ), tuple3( x, y, z ) ...

Accessing an element:
- tpl_1( tuple2(x, y) ) // x
- tpl_2( tuple3(x, y, z) ) // y

Pattern matching

Tuple2#(Bit#(2), Bool) tup = tuple2(2, True);
match {.a, .b} = tup;
// a = 2, b = True
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)

http://csg.csail.mit.edu/6.375
Reg#(t)

- Main state element in BSV
- Type: Reg#(type data_type)
- Instantiated differently from normal variables
  - Uses <- notation
- Written to differently from normal variables
  - Uses <= notation
  - Can only be done inside of rules and methods

```
Reg#(Bit#(32)) a_reg <- mkReg(0); // value set to 0
Reg#(Bit#(32)) b_reg <- mkRegU(); // uninitialized
```