# Constructive Computer Architecture

## Tutorial 1

# BSV

Sizhuo Zhang
6.175 TA

# 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 !=

# struct

```
typedef struct {
    Bit#(12) addr;
    Bit#(8) data;
    Bool wren;
} MemReq deriving (Bits, Eq);
```

- Elements from `MemReq x` can be accessed with `x.addr`, `x.data`, `x.wren`
- Struct Expression
  - `X = MemReq{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
```

# Vector

- Type:
  - `Vector#(numeric type size, type data_type)`
- Values:
  - newVector(), replicate(val)
- Functions:
  - Access an element: []
  - Rotate functions
  - Advanced functions: zip, map, fold
- Can contain registers or modules
- Must have 'import Vector::*;' in BSV file

# 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)

# tagged union

- Maybe is a special type of tagged union

```
typedef union tagged {
    void Invalid;
    t    Valid;
} Maybe#(type t) deriving (Eq, Bits);
```

- Tagged unions are collections of types and tags. The type contained in the union depends on the tag of the union.
  - If tagged Valid, this type contains a value of type t

# tagged union

- Values:
  - tagged <tag> value
- Pattern matching to get values:

```
case (x) matches
     tagged Valid .a : return a;
     tagged Invalid : return 0;
endcase
```

```
if(x matches tagged Valid .a &&& a > 1)
begin
     $display(“%d”, a);
end
```

# 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
```

# 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) );
- 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

# Modules

- Modules are building blocks for larger systems
  - Modules contain other modules and rules
  - Modules are accessed through their interface
- `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);
```

# Interfaces

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

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

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

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

# Interface Methods

- Method
  - 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)) deq;`

# Implement Interface of Module

- Instantiate methods at the end of module

```
interface MyIfc#(numeric type n);
    method ActionValue#(Bit#(n)) f();
    interface SubIfc#(n) s;
endinterface
module mkDut(MyIfc#(n));
    ……
    method ActionValue#(Bit#(n)) f();
        ……
    endmethod
    interface SubIfc s; // no param “n”
        // methods of SubIfc
    endinterface
endmodule
```

# Implement Interface of Module

- Return interface at the end of module
  - Interface expression

```
module mkDut(MyIfc#(n));
    ……
    MyIfc ret = (interface MyIfc;
        method ActionValue#(Bit#(n)) f();
            ……
        endmethod
        interface SubIfc s; // no param "n"
            // methods of SubIfc
        endinterface
    endinterface);
    return ret;
endmodule
```

# 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

# Best way to learn BSV

- BSV Reference guide
- Lab code
- Try it
  - Makefile in lab 1,2,3...

# Scheduling

- Compile flag (BSV user guide)
  - -aggressive-conditions (Section 7.12)
    - predicated implicit guards
  - -show-schedule (Section 8.2.2)
    - method/rule schedule information
    - Output file: info-dir/*.sched
  - -show-rule-rel r1 r2 (Section 8.2.2)
    - Print conflict information