BSV - Rishiyur Nikhil

Lecture 1 - Introduction & Overview - No Notes

Lecture 2 - Combinational Structures and Basic Types

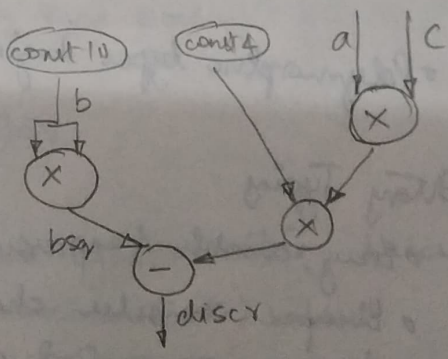
Existing HDLs are Simulation centric - BSV was introduced to make HDLs hardware centric.

→ Expressions are combinational circuits
  ◦ constants, variables, operators and function application
→ Basic Types
  ◦ Strong Typing
  ◦ Types describe values (independent of wires & storage elements)
→ Variables: declaration, assignment, control structures
  ◦ The "non-procedural" or "combinational" view of variable assignment
  ◦ Not simulation
→ Functions ⇒ parameterized combinational circuit
→ Variable scoping

⇒ Expressions are combinational Circuits

code (Verilog notation)

```
b = 10;
bsq = b*b;
discr = bsq - 4*a*c;
```



* variables are just names of wires in BSV.
* variables are never thought of as a storage element.

"Expressions describe the dataflow in a circuit".

⇒ Syntax details
→ comments //line; /* block */ → Whitespace
→ Identifiers → Case Sensitive → Standard scoping same like verilog.

⇒ Basic Operators ⇐ from Verilog/System Verilog

⇒ Basic Types

- Central Role in BSV
- described with "type Expressions"
  - * Simple type expressions are just identifiers
- In BSV all type names begin with a "Capital letter"
  Exceptions: 'int' and 'bit', for compatibility with verilog.

Ex: Integer → Unbound signed integers ⇒ Integer num1=3;
Static elaboration

int ⇒ 32b wide Signed integers ⇒ int num2 = 'h1;
bit [15:0] → 16b wide bit vector ⇒ bit [15:0] x = 23;
Bool → Possible values True/False ⇒ Bool condition=False;
String → as in Verilog/VHDL or C ⇒ String msg = "Hello";

⇒ Syntax Details

Convention:
- Type identifiers begin with capital letters
- Value identifiers begin with lowercase letters
- Polymorphic types begin with a lowercase letter

⇒ Strong Typing

- Every variable & expression has a Type
- Bluespec compiler checks if the constructs in the language are applied correctly according to Types
  - * Operator's/function's arguments are of the correct type
  - * assignment is to the correct type
  - * Module's parameters are the correct type
  - * Module's interface is of the correct type
- mismatch issues an error message
- More stringent than Verilog/System Verilog
  - * Registers are strongly typed
  - ** No automatic sign or zero extension; no automatic truncation.
    ⇒ But the compiler will calculate the extensions and truncations.

⇒ Extension / Truncation:

code:

```
bit [31:0] x;
x = signExtend (25'h9BEEF);
x = zeroExtend (25'h9BEEF);
x = {0, 25'h9BEEF}; // same as zeroExtend
x = zeroExtend {39'h9BEEF}; // error: Input too wide

x = truncate (37'h9BEEF);
x = truncate (25'h9BEEF); // error: Input too narrow
```

⇒ Types describe sets of values (Abstract entities)

* A type describes a set of value
* Types are independent entities that may carry values (such as wires, registers, ...)
    - No inherent connection with storage, or updating
* True for even of complex types

  Eg, struct {int ..., Bool ...}
    - This just represents a set of pairs of values, where the first member of pair is an int, the other element is of type Bool.

⇒ Strong Property of Types in BSV

- Any Expression
  * is guaranteed by BSV's type-checking rules to represent a pure (combinational) value
    → it cannot allocate any state
    → it cannot update any state
  * except if its type contains either of the following 2 types
    → Action → ActionValue

- Any expression can be freely shared or replicated without changing behaviour
  * The BSV compiler exploits this to perform aggressive "Common Subexpression elimination" optimization
  * This is very safe.

⇒ Variable declaration, initialization and assignment

- Every variable has a type
- We use standard Verilog notation for declaring its types, optional initialization and assignment

code:

```
type var[=init], var[=init], ...;
int x, y=23, z;
Bool b;
z = y+2;
x = z*2;
b = (x>=23);
```

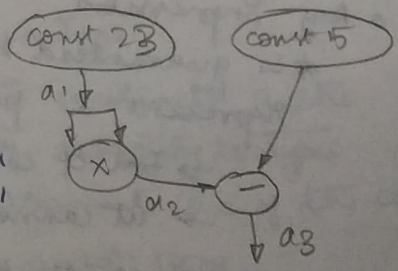
Same till now

⇒ Variable Assignment

- BSV does not use Verilog's "process" or "procedural" notation (to express behaviour, BSV uses rules)
  - * Variables are not an "updatable container".
  - * An assignment is not a "procedural" statement that updates a container
  - * Registers and state elements are modules.
- Variable is just a name for an expression
- Repeated assignment is just a notation for incrementally building up expressions.
  - * Think of it as a new variable from that point onwards
- Thinking in terms of hardware.

code:

```
Int a=23;          int a1=23
a = a * a;   (or)  a2 = a1 * a1
a = a - 5          a3 = a2 - 5;
```



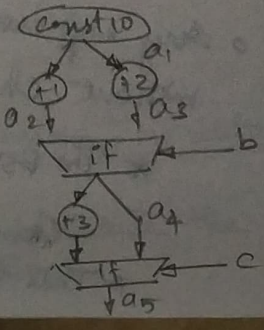
for understanding

every time 'a' is a new wire, with the same name.

⇒ Variable assignment (contd.)

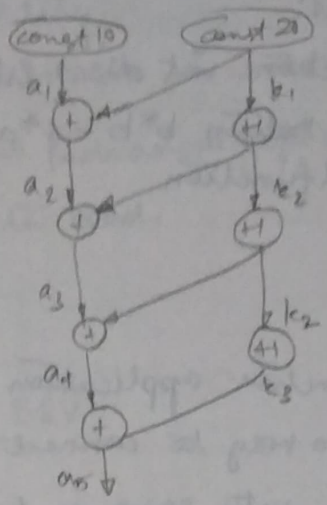
code

```
int a = 10;
if (b) a = a+1;
else   a = a+2;
if (c) a = a+3;
```



code:

```
int a = 10;
for (int k = 20; k < 24; k = k+1)
    a = a+k;
```



⇒ Expressions are combinational circuits

- Expressions describe dataflow
  - * Variables are not storage locations
- Everything is static elaboration, for variables & assignments, not processes or procedures
- No 'sensitivity lists', no 'always_comb';
- No "execute an assignment to drive this wire"

⇒ Variable declaration and initialization using "let".

- BSV has a "let" statement in which a variable can be declared and initialized, with the compiler deducing the type of the variable based on the type of the right-hand-side

code

```
// let var = init;
let x = 24'h9BEEF;   // compiler deduces type bit[23:0] for x
let y = x+3          // compiler deduces type bit[23:0] for y.
```

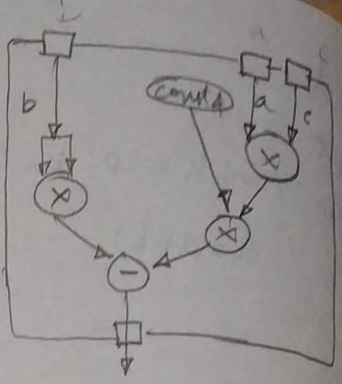
- Use it judiciously, The implicit type can make it less readable (and therefore less maintainable) but no hard and fast rules.
- Can be used when a large structure and things are used.

⇒ Functions are parameterized expressions

- A function is just an abstraction of a combinational expression
- Arguments are inputs to the circuit
- Results is the output of the circuit (output can be a struct carrying multiple values)

Code:

```
function int discr (int a, int b, int c);
    return b*b - 4*a*c;
endfunction
```

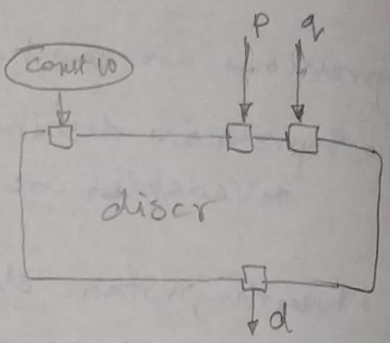


⇒ Function application

- way to connect (compose) a combinational expression into some content.

code

```
d = discr(10, p, q);
```



- instantiates combinational hardware

⇒ Advanced topics in Functions:

- functions can be used for both static elaboration and for describing combinational circuits.
- When used for static elaboration, function arguments and results can have any type, not just restricted types as in verilog / SystemVerilog.
  - * Rules, interfaces, modules and even functions
  - * very powerful abstraction mechanism
  - * powerful "generate" capabilities, where one can programmatically describe complex hardware
    - → loops / linear recursion for linear repetitive circuit structures
    - → Binary recursion for tree-structured circuits (eg: +'reduction)

⇒ Static Scoping:

- Traditional scoping rules.
  - * Each variable refers to the declaration in the nearest textually enclosing scope

- Recommendation for readability:
  - → Dont re-assign a variable in an inner scope
  - → Semantically meaningful (advanced topic), but can be confusing to read.

Lecture 3 – Types 1

Types play a very central role with BSV

- → Simple scalar types
- → Typedefs of enums and structs
  - Strong typing
  - Types describe sets of values, not wires/storage
- → Parameterized types : Type #(Type Param, ...)
- → Maybe Type
- → Overloading: deriving
- → Intro about Types II and Types III

⇒ BSV point of view:

- Types have played a central role in abstraction mechanisms
- BSV uses these ideas
- Basic types are similar to verilog. Sys Verilog extensions include typedefs, enums, structs, tagged unions, interface types, type parameterization/polymorphism.
- BSV extends this to systematic Overloading

⇒ Basic Types.

- Types are described with Type Expressions
  - * Simple type expressions are just identifiers
- In general upper case is used for the beginning of type identifiers
  - * Exceptions 'int' & 'bit'.

⇒ Type synonyms with "typedef"

"typedef" is often used to define a new, more readable or convient synonym for an existing type.

- * just a synonym, variables & expressions of either type can be mixed/assigned freely.

code:

Syntax → `typedef existingType NewType;`

```
typedef int Addr;
typedef bit[63:0] Data;
typedef bit[15:0] Halfword;
typedef Bool RoundRobinFlag;
```

* Advantages: updation can be easier

- Type names 'must' have an uppercase letter.

⇒ Defining a new 'enum' type (Best for FSMs)

- An enum type defines a new type (not a synonym) with a set of scalar values with symbolic names

  * Because it is a new type, it is more robust (type-safe) to define an enum, compared to using ints or bit-vectors to represent the set of values (type-checking ensures that you cannot accidentally use an unrelated bit/integer value)

code

⇒ Syntax ⇒ typedef enum {Identifier, ...} NewType deriving (Bits, Eq);

(deriving explained later with overloading)

typedef enum {Red, Green, Yellow} TrafficLight deriving (Bits, Eq);

typedef enum {Reset, Count, Decision} State deriving (Bits, Eq);

→ Even though State & TrafficLight are both 2 bits wide, the type-checking does not allow a TrafficLight value to a State and vice versa. TrafficLight & State are distinct.

⇒ Syntax notes for enum.

→ Labels must begin with uppercase letter

→ Enum labels can be repeated in different enum definitions unlike System Verilog

→ The default encoding of these labels is 0, 1, 2, ... using just enough bits to encode the full set.

- (Other encodings may be specified)

⇒ Using an enum.

- Defined enum type is used just like any other type, to declare variables, function/module parameters, etc.
- The enum labels are used as constant values of that type

code:

```
typedef enum {Reset, Count, Decision} State deriving (Bits, Eq);

State defaultState = Reset;

function State nextState (State s);
   case (s)
    Reset: s = Count;
    default: s = Decision;
   endcase
   return s;
endfunction
```

- Each defined enum type is a new type distinct from all other types

code:

```
typedef enum {Green, Yellow, Red} TrafficLight deriving (Bits, Eq);
typedef enum {Reset, Count, Decision} State deriving (Bits, Eq);

bit [1:0] x;
State s1 = x; // type checking error
State s2 = Yellow; // type-checking error.
```

⇒ Defining a new 'struct' type - "deriving (Bits)"

- A struct ("record") is a composite type. A struct value is a collection of values, each of which has a particular type and is identified by a 'member or field' name

code:

Syntax ⇒ `typedef struct {type identifier; ... ; } NewType deriving (Bits, Eq);`

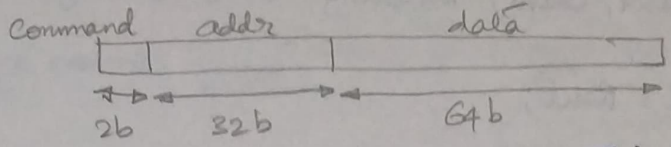
```
typedef enum {Load, Store, LoadLock, StoreCond} Command
                                                deriving (Bits, Eq);

typedef struct {
    Command command;
    bit [31:0] addr;
    bit [63:0] data;
  } BusRequest
    deriving (Bits, Eq);
```

⇒ Syntax notes
- NewType name begins with uppercase
- member names - lowercase
- member names can be repeated in other structs.

⇒ Struct Representation
- The default representation of a struct is a concatenation of the member representations, from MSB to LSB.

The BusRequest Struct looks like



⇒ This can be modified.

⇒ Using a struct type

- Just like any other type
- Member names are used to access members of that type

code:

```
BusRequest req;
req.command = Load;
req.addr = baseAddr + 32'h16;
req.data = 64'h9BEEF;
if (req.command == LoadLock)
    req.command = StoreCond;
```

⇒ Entire struct values.

- A struct type defines a set of struct values independent of entities that may carry such values (wires, registers, ---)
  - * A struct value is not, per se, associated with any storage.

code:

```
BusRequest req = BusRequest {
                    command : Load,
                    addr : baseAddr + 32'h16,
                    data : 64'h9BEEF};

function BusRequest incAddr (BusRequest req);
    req.addr = req.addr + 4;
    return req;
endfunction
```

Struct expression having Struct values

- Structs are maintainable.
- Structs can be arguments, expressions and values or even results.

⇒ Using 'let' for struct initialization

- It is often convenient to use "let" for declaring and initializing a struct value.

code

```
let req = BusRequest {
            command: Load, addr: baseAddr+32'h16,
            data: 64'h9BEEF};
// compiler reduces the type of req.
```

⇒ Each struct type is distinct.

- Each defined struct type is a new type distinct from all others (even though they may happen to have members with the same type)

code:

```
typedef struct {int a; Bool b;} Foo deriving (Bits, Eq);
typedef struct {int c; Bool d;} Baz deriving (Bits, Eq);

Foo x;
Baz y;

x = y;  // type checking error

x.a = y.c;  //ok
x.b = y.d;  //ok.
```

⇒ Parameterized types

- Many types have some other types associated with them in some (orthogonal or independent) way.
  - Array type, each item has its type
  - memory type, we associate the type of addresses & the type of data
  - register file type, we associate the type of register names and register data
- System Verilog introduces a notation for this:

  Type #(Type, ..., Type)

code:

```
Mem #(Addr, Data)
RegFile #(RegName, RegData)
Client #(Request, Response)    yields Requests, accepts Responses
Server #(Request, Response)    accepts Requests, yields Responses.
```

⇒ Numeric Types (Size)

- In BSV, some type parameters can be numerical and they indicate something about the size of each value of that type.

Type	Meaning	Example
Bit#(n) (just like bit[n-1:0], also int = Bit#(32))	Bit vector of width n	Bit#(132) vec = 132'd30;
UInt#(n)	unsigned int of width n	UInt#(4) vec2 = 4'01;
Int#(n)	signed int of width n	Int#(16) vec3 = 16'hFF00;

Type	Meaning	Example
Vector#(n, t)	Vector of n elements, each of type t	Vector#(3, Bool) vec1 4; Vector#(14, Int#(32)) vec1 5; Vector#(16, Tuple2#(Bool, Bit#(8))) vec1 6;

* Tuple2 is just a struct with 2 fields

- In numeric type positions, you can only supply constants

later { * later one, we can see that we can put type variables
* see how to express arithmetic constraints.

- You cannot mix ordinary numeric expressions with numeric types. There is no ambiguity:
  - Numeric types only occur in places where a type expression is expected (eg. type parameter to another type)
  - Ordinary numeric expressions only occur in value contents.

⇒ A useful type: Maybe#(t)

- In HW design we frequently has this situation
  - A value of some type t
  - an accompanying 'valid bit' which says if the value is meaningful or not
- BSV ~~produces~~ provides Maybe#(t) for this purpose (Maybe is defined as a tagged union type)

code: int + valid type

```
Maybe#(int) m1 = tagged Valid 23;   // Valid bit True, value 23
Maybe#(int) m2 = tagged Invalid;    // Valid bit False, value unspecified.

m2 = m1;                            // legal.
Bool b = isValid(m2);               // b == validbit of m2
int x = fromMaybe(34, m2)           // x = value of m2 if valid, else 34
```

⇒ First Brush with Overloading

The ability to use a common function name or operator on some repertoire of types.

Example:

- "+" is meaningful on bits, integers ... perhaps colors etc.
- "<=" is meaningful on bits, numeric types, vectors, perhaps Ethernet packets, etc.

- In most languages, overloading is ad hoc
  - * Usually, only a fixed set of operators
  - * Usually, not extensible by the designer/programmer
  - * Minor exception: SystemVerilog allows extensibility over a fixed set of operators.
  - * Major Exception: C++ has systematic extensibility (not allowed in synthesizable SystemC)

- BSV has a systematic way to extend overloading to any operator and function, and to any type.
  - * Terminology → typeclasses, typeclass instances, provisos & deriving
  - * Now we focus on deriving, later about others.

⇒ deriving (Bits) "Use it only when the type will be on HW"

- Rather than using ad hoc rules about representing a particular type in bits, BSV takes this systematic route
  - + There are 2 overloaded functions
    - function Bit#(n) pack (t x);
    - function t unpack (Bit#(n) y);
  - * The pack function encapsulates how to convert a value x of type t into a bit-vector of width n.
  - * The unpack is pack's dual.

- For an user-defined type, the user can define the functions, hence control the bit representations completely.
- With "deriving (Bits)" in the type definition, the user directs the compiler to define pack() & unpack() using "default packing".
- Without pack()/unpack(), the compiler assumes no bit representation! Hence if we have a type which will be present in hardware, either say "deriving (Bits)" or define pack() or unpack() explicitly.

⇒ deriving (Eq)

- Rather than use ad hoc rules about how to test equality between two values of type t, BSV takes the following systematic route.
  - There are 2 overloaded functions for equality & inequality
    - function Bool \== (t x, t y); // using verilog
    - function Bool \!= (t x, t y); // notation "\..." for escaped identifier
- for any user-defined type, the user can declare these operators and therefore control the meaning of equality/inequality
- By saying "deriving (Eq)" in the type definition, the user directs the compiler to define '==' and '!=' using "default rules"
- Without == and !=, the compiler assumes no way to test for equality! Hence whenever we expect to do such tests, either use "deriving (Eq)", or define == and != explicitly.

⇒ Future Topics on Types.

- Polymorphic types (generic types, "template classes"): when types contain type variables.
- Tagged Union types, and pattern-matching
- Constraints on numeric types
  - "Size of reg = log() of size of address space".
- More on overloading
  - Interpreting integer literals
  - Bitwise (ops: &, |, ^, ~^, ^~, invert, <<, >>)
  - Ord (ops: <, <=, >, >=)
  - Arith (ops: +, − negate, *)
  - Bounded (consts: minBound, maxBound)
  - Type classes, Provisos, Typeclass instances