Constructive Computer Architecture

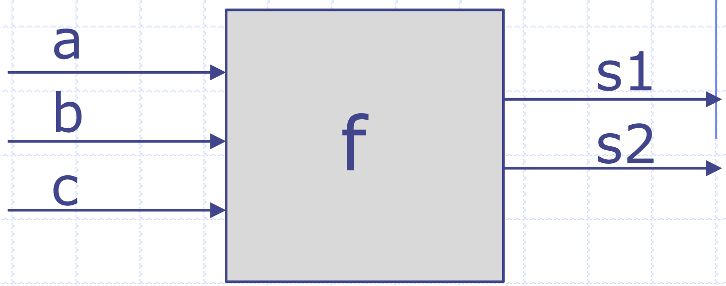
Combinational circuits

Arvind
Computer Science & Artificial Intelligence Lab.
Massachusetts Institute of Technology

Combinational circuit

- A combinational circuit is a pure function; given the same input, it produces the same output
- It can be described using a Truth Table though it is not practical to do so for large number of inputs
  - Size of truth table for a 32-bit adder? 2^{32+32} rows

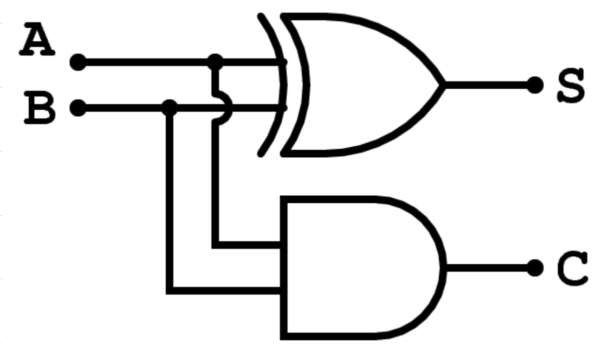
We will use a programming language called Bluespec System Verilog (BSV) to express all ckts



a	b	c	s1	s2
0	0	0	1	0
0	0	1	0	0
0	1	0	0	0
0	1	1	0	1
1	0	0	1	1
1	0	1	1	0
1	1	0	0	1
1	1	1	0	0

Half Adder

A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1




```
function ha(a, b);
    s = a ^ b;
    c = a & b;
    return {c,s};
endfunction
```

XOR

AND

Boolean equations

$s = a \oplus b$

$c = a \cdot b$

Not quite correct – needs type annotations

Half Adder *corrected*

```
function Bit#(2) ha(Bit#(1) a, Bit#(1) b);
 Bit#(1) s = a ^ b;
 Bit#(1) c = a & b;
   return {c,s};
endfunction
```

"`Bit#(1) a`" type declaration says that `a` is one bit wide

`{c,s}` represents bit concatenation

How big is `{c,s}`?

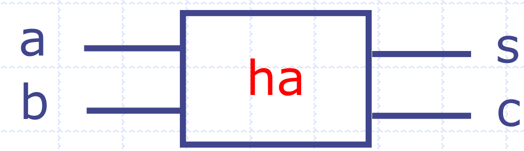
2 bits

BSV notes

```
function Bit#(2) ha(Bit#(1) a, Bit#(1) b);
 Bit#(1) s = a ^ b;
 Bit#(1) c = a & b;
   return {c,s};
endfunction
```

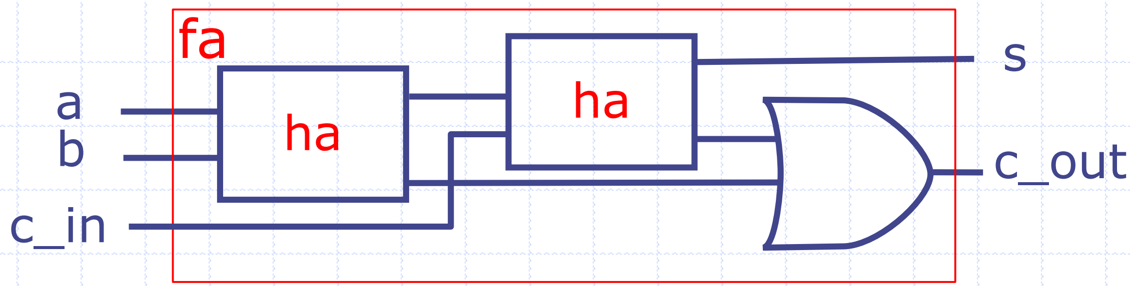
ha can be used as a black-box as long as we understand its type signature

- Suppose we write `t = ha(a,b)` then `t` is a two bit quantity representing `c` and `s` values
- We can recover `c` and `s` values from `t` by writing `t[1]` and `t[0]`, respectively



Full Adder

1-bit adder with a carry-in input



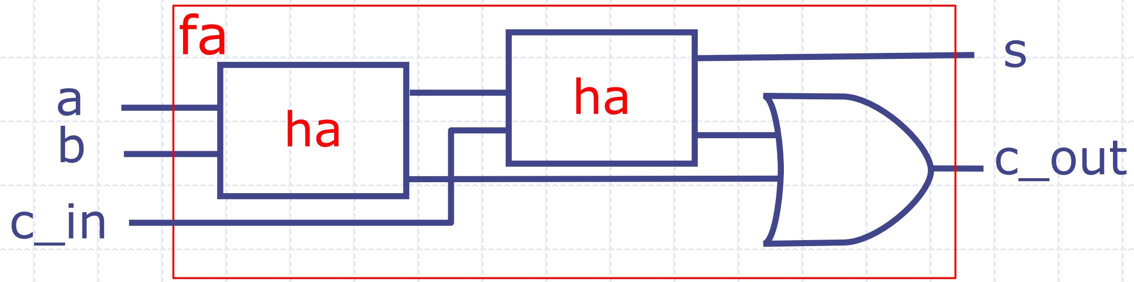

```
function Bit#(2) fa(Bit#(1) a, Bit#(1) b,
                                         Bit#(1) c_in);
   Bit#(2) ab  = ha(a, b);
   Bit#(2) abc = ha(ab[0], c_in);
   Bit#(1) c_out    = ab[1] | abc[1];
   return {c_out, abc[0]};
endfunction
```

Extracts the sum bit

Extracts the carry bit

ha is being used as a black-box; fa code is simply a wiring diagram

The "let" syntax




```
function Bit#(2) fa(Bit#(1) a, Bit#(1) b,
                                        Bit#(1) c_in);
   let ab  = ha(a, b);
   let abc = ha(ab[0], c_in);
   let c_out   = ab[1] | abc[1];
   return {c_out, abc[0]};
endfunction
```

No need to write the type if the compiler can deduce it

Types

- A type is a grouping of values:
  - Integer: `1, 2, 3, …`
  - Bool: `True, False`
  - Bit: `0,1`
  - A pair of Integers: `Tuple2#(Integer, Integer)`
  - A function `fname` from Integers to Integers:

```
function Integer fname (Integer arg)
```

- Every expression in a BSV program has a type; sometimes it is specified explicitly and sometimes it is deduced by the compiler
- Thus, we say an expression has a type or belongs to a type

The type of each expression is unique

Parameterized types:

- A type declaration itself can be parameterized by other types
- Parameters are indicated by using the syntax '#'
  - For example `Bit#(n)` represents n bits and can be instantiated by specifying a value of n

    `Bit#(1), Bit#(32), Bit#(8), …`

Type synonyms

```
typedef bit [7:0] Byte;

typedef Bit#(8) Byte;
```

The same

```
typedef Bit#(32) Word;

typedef Tuple2#(a,a) Pair#(type a);

typedef Int#(n) MyInt#(numeric type n);
```

In some special cases one can just write:

```
typedef Int#(n) MyInt#(type n);
```

Type declaration versus deduction

- The programmer writes down types of some expressions in a program and the compiler deduces the types of the rest of expressions
- If the type deduction cannot be performed or the type declarations are inconsistent then the compiler complains

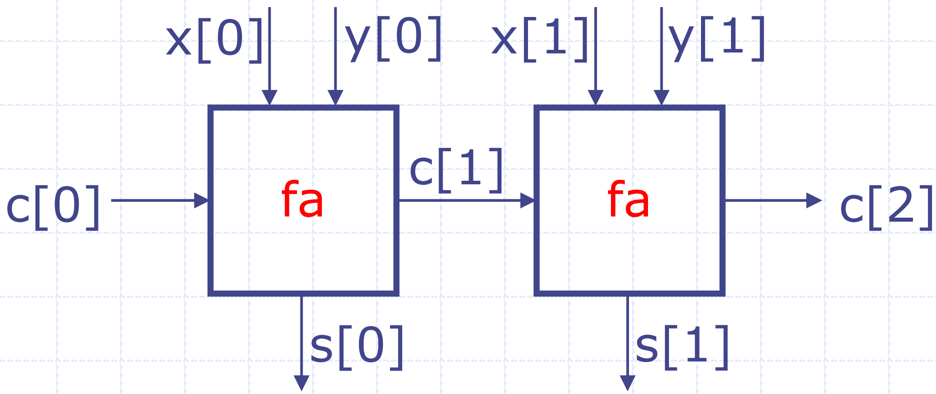
```
function Bit#(2) fa(Bit#(1) a, Bit#(1) b,
                                        Bit#(1) c_in);
   Bit#(2) ab  = ha(a, b);
   Bit#(2) abc = ha(ab[0], c_in);
   Bit#(2) c_out   = ab[1] | abc[1];
   return {c_out, abc[0]};
endfunction
```

type error?

Type checking prevents lots of silly mistakes

2-bit Ripple-Carry Adder

cascading full adders



Use fa as a black-box

```
function Bit#(3) add(Bit#(2) x, Bit#(2) y,
                                              Bit#(1) c0);
   Bit#(2) s = 0;    Bit#(3) c=0; c[0] = c0;
   let cs0 = fa(x[0], y[0], c[0]);
             c[1] = cs0[1];  s[0] = cs0[0];
   let cs1 = fa(x[1], y[1], c[1]);
             c[2] = cs1[1];  s[1] = cs1[0];
  return {c[2],s};
endfunction
```

Initially s wires are zero

wire s[0] is updated

wire s[1] is updated

Assigning to Vector elements

```
Bit#(3) c=0;
```

- Means `c` is three bits wide and each element is set to zero

```
c[0] = c0;
```

- Element `0` of `c` is connected to `c0` but the value of the rest of the elements is not affected

- Initial value of a vector must be set; we use "?" if we don't know the initial value

An w-bit Ripple-Carry Adder

- For a w-bit adder, unless we know the value of w, we cannot write a straight-line program as we did for 2-bit adder
- Use loops!

```
function Bit#(w+1) addN(Bit#(w) x, Bit#(w) y,
                                   Bit#(1) c0);
   Bit#(w) s; Bit#(w+1) c=0; c[0] = c0;
   for(Integer i=0; i<w; i=i+1)
   begin
      let cs = fa(x[i],y[i],c[i]);
      c[i+1] = cs[1]; s[i] = cs[0];
   end
return {c[w],s};
endfunction
```

There are some subtle type errors in this program but before we fix them, you may wonder what is the meaning of a loop in terms of gates?

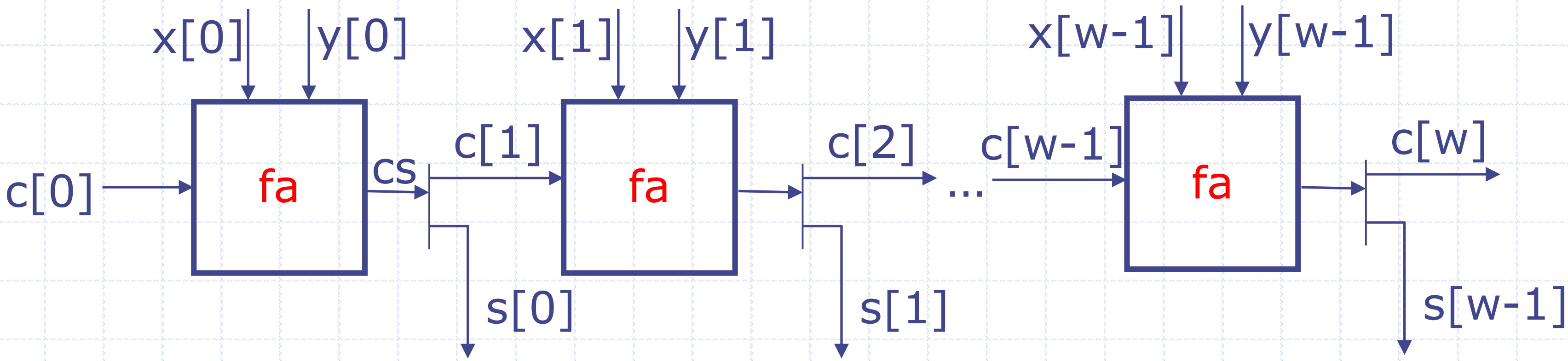
All loops are unfolded by the compiler!

```
for(Integer i=0; i<w; i=i+1)
   begin
      let cs = fa(x[i],y[i],c[i]);
      c[i+1] = cs[1]; s[i] = cs[0];
   end
```

Can be done only when the value of w is known

```
cs0 = fa(x[0], y[0], c[0]); c[1]=cs0[1]; s[0]=cs0[0];
cs1 = fa(x[1], y[1], c[1]); c[2]=cs1[1]; s[1]=cs1[0];
…
csw = fa(x[valw-1], y[valw-1], c[valw-1]);
      c[valw] = csw[1]; s[valw-1] = csw[0];
```

Loops to gates




```
cs0 = fa(x[0], y[0], c[0]); c[1]=cs0[1]; s[0]=cs0[0];
cs1 = fa(x[1], y[1], c[1]); c[2]=cs1[1]; s[1]=cs1[0];
…
csw = fa(x[valw-1], y[valw-1], c[valw-1]);
      c[valw] = csw[1]; s[valw-1] = csw[0];
```

Unfolded loop defines an acyclic wiring diagram

Instantiating the parametric Adder

```
function Bit#(w+1) addN(Bit#(w) x, Bit#(w) y,
                                     Bit#(1) c0);
```

How do we define a `add32`, `add3` … using `addN` ?

```
// concrete instances of addN!
function Bit#(33) add32(Bit#(32) x, Bit#(32) y,
                        Bit#(1) c0) = addN(x,y,c0);
```

The numeric type w on the RHS implicitly gets instantiated to 32 because of the LHS declaration

```
function Bit#(4) add3(Bit#(3) x, Bit#(3) y,
                      Bit#(1) c0) = addN(x,y,c0);
```

Fixing the type errors

`valueOf(w)` versus `w`

- Each expression has a type and a value, and these two come from entirely disjoint worlds
- `w` in `Bit#(w)` resides in the types world
- Sometimes we need to use values from the types world into actual computation. The function `valueOf` allows us to do that
  - Thus

    `i<w` is not type correct

    `i<valueOf(w)` is type correct

Fixing the type errors

`TAdd#(w,1)` versus `w+1`

- Sometimes we need to perform operations in the types world that are very similar to the operations in the value world
  - Examples: `Add`, `Mul`, `Log`
- We define a few special operators in the types world for such operations
  - Examples: `TAdd#(m,n)`, `TMul#(m,n)`, …

`Integer` versus `Int#(32)`

- In mathematics integers are unbounded but in computer systems integers always have a fixed size
- BSV allows us to express both types of integers, though unbounded integers are used only as a programming convenience

integer

```
for(Integer i=0; i<valw; i=i+1)
   begin
      let cs = fa(x[i],y[i],c[i]);
      c[i+1] = cs[1]; s[i] = cs[0];
   end
```

A w-bit Ripple-Carry Adder

corrected

```
function Bit#(TAdd#(w,1)) addN(Bit#(w) x, Bit#(w) y,
                                          Bit#(1) c0);
   Bit#(w) s; Bit#(TAdd#(w,1)) c; c[0] = c0;
   let valw = valueOf(w);
   for(Integer i=0; i<valw; i=i+1)
   begin
       let cs = fa(x[i],y[i],c[i]);
       c[i+1] = cs[1]; s[i] = cs[0];
   end
return {c[valw],s};
endfunction
```

types world equivalent of w+1

Lifting a type into the value world

Structural interpretation of a loop – unfold it to generate an acyclic graph

BSV Compiling phases

- Type checking: Ensures that type of each expression can be determined uniquely; Otherwise the program is rejected
- Static elaboration: Compiler eliminates all constructs which have no direct hardware meaning
  - Loops are unfolded
  - Functions are in-lined; even recursive functions can be used as long as all the recursion can be gotten rid of at compile time
  - After this stage the program does not contain any Integers because Integers are unbounded in BSV
- Gates are generated (actually Verilog)

Takeaway

- Once we define a combinational ckt, we can use it repeatedly to build larger ckts
- The BSV compiler, because of the type signatures of functions, prevents us from connecting them in obviously illegal ways
- We can write parameterized ckts in BSV, for example an n-bit adder. Once n is specified, the correct ckt is automatically generated
- Even though we use loop constructs and functions to express combinational ckts, all loops are unfolded and functions are inlined during the compilation phase