

Constructive Computer Architecture

Combinational circuits

Arvind

Computer Science & Artificial Intelligence Lab.
Massachusetts Institute of Technology

September 11, 2015

<http://csg.csail.mit.edu/6.175>

L02-1

Content

- ◆ Design of a combinational ALU starting with primitive gates And, Or and Not
- ◆ Combinational circuits as acyclic wiring diagrams of primitive gates
- ◆ Introduction to BSV
 - Intro to types – enum, typedefs, numeric types, Int#(32) vs Integer, Bool vs Bit#(1), Vector
 - Simple operations: concatenation, conditionals, loops
 - Functions
 - Static elaboration and a structural interpretation of the textual code

September 11, 2015

<http://csg.csail.mit.edu/6.175>

L02-2

Combinational circuits

September 11, 2015

<http://csg.csail.mit.edu/6.175>

L02-3

Combinational circuits are acyclic interconnections of gates

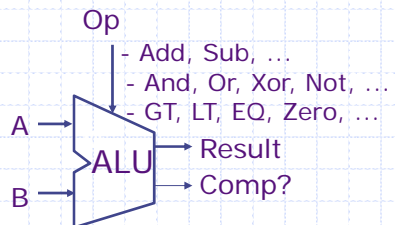
- ◆ And, Or, Not
- ◆ Nand, Nor, Xor
- ◆ ...

September 11, 2015

<http://csg.csail.mit.edu/6.175>

L02-4

Arithmetic-Logic Unit (ALU)



ALU performs all the arithmetic and logical functions

Each individual function can be described as a combinational circuit

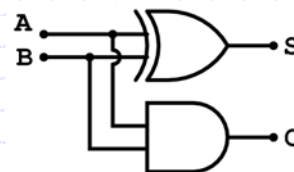
September 11, 2015

<http://csg.csail.mit.edu/6.175>

L02-5

Half Adder

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



Boolean equations

$$s = (\sim a \cdot b) + (a \cdot \sim b)$$

$$c = a \cdot b$$

"Optimized"

$$s = a \oplus b$$

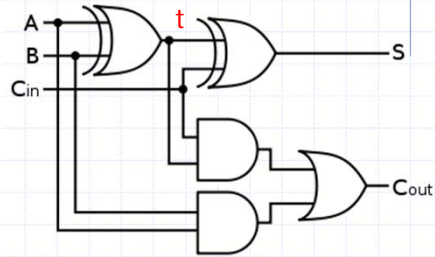
September 11, 2015

<http://csg.csail.mit.edu/6.175>

L02-6

Full Adder

A	B	C _{in}	S	C _{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



Boolean equations

$$s = (\sim a \cdot \sim b \cdot c_{in}) + (\sim a \cdot b \cdot \sim c_{in}) + (a \cdot \sim b \cdot \sim c_{in}) + (a \cdot b \cdot c_{in})$$

$$c_{out} = (\sim a \cdot b \cdot c_{in}) + (a \cdot \sim b \cdot c_{in}) + (a \cdot b \cdot \sim c_{in}) + (a \cdot b \cdot c_{in})$$

“Optimized”

$$t = a \oplus b$$

$$s = t \oplus c_{in}$$

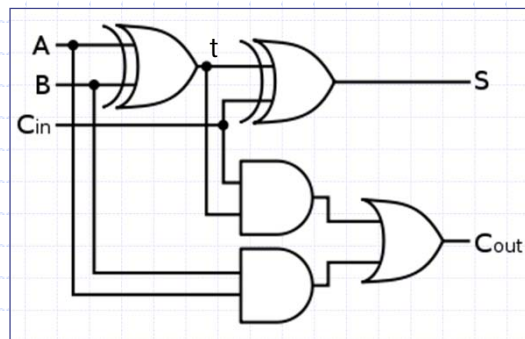
$$c_{out} = a \cdot b + c_{in} \cdot t$$

Full Adder: A one-bit adder

```
function fa(a, b, c_in);
    t = (a ^ b);
    s = t ^ c_in;
    c_out = (a & b) | (c_in & t);
    return {c_out, s};
endfunction
```

Structural code – only specifies interconnection between boxes

Not quite correct – needs type annotations



Full Adder: A one-bit adder

corrected

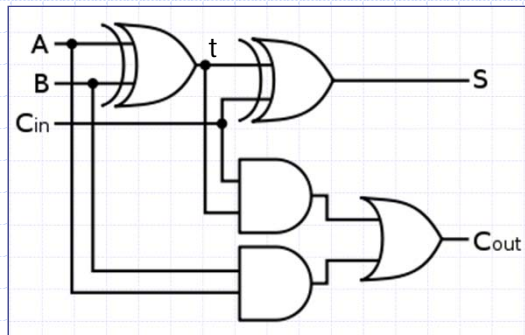
```
function Bit#(2) fa(Bit#(1) a, Bit#(1) b,  
                  Bit#(1) c_in);  
  Bit#(1) t = a ^ b;  
  Bit#(1) s = t ^ c_in;  
  Bit#(1) c_out = (a & b) | (c_in & t);  
  return {c_out, s};  
endfunction
```

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

{c_out, s} represents bit concatenation

How big is {c_out, s}?

2 bits



September 11, 2015

<http://csg.csail.mit.edu/6.175>

L02-9

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

September 11, 2015

<http://csg.csail.mit.edu/6.175>

L02-10

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#(type n);
```

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

The same

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#(1) t = a ^ b;
  Bit#(1) s = t ^ c_in;
  Bit#(2) c_out = (a & b) | (c_in & t);
  return {c_out, s};
endfunction
```

type error

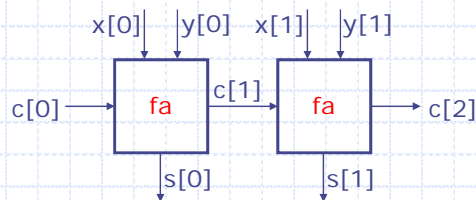
Type checking prevents lots of silly mistakes

September 11, 2015

<http://csg.csail.mit.edu/6.175>

L02-13

2-bit Ripple-Carry Adder



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

```
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
```

The "let" syntax avoids having to write down types explicitly

September 11, 2015

<http://csg.csail.mit.edu/6.175>

L02-14

"let" syntax

- ◆ The "let" syntax: No need to write the type if the compiler can deduce it:

- `let cs0 = fa(x[0], y[0], c[0]);`
 - `Bit#(2) cs0 = fa(x[0], y[0], c[0]);`
- The same

September 11, 2015

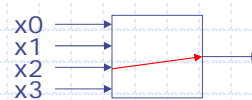
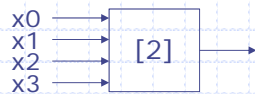
<http://csg.csail.mit.edu/6.175>

L02-15

Selecting a wire: $x[i]$

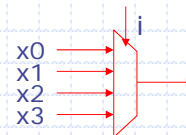
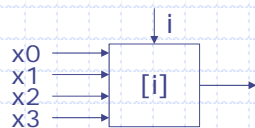
assume x is 4 bits wide

- ◆ Constant Selector: e.g., $x[2]$



no hardware;
 $x[2]$ is just
the name of
a wire

- ◆ Dynamic selector: $x[i]$



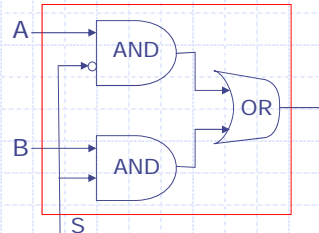
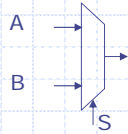
4-way mux

September 11, 2015

<http://csg.csail.mit.edu/6.175>

L02-16

A 2-way multiplexer



`(s==0)? A : B`

Gate-level implementation

Conditional expressions are also synthesized using muxes

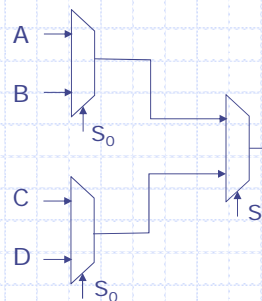
September 11, 2015

<http://csg.csail.mit.edu/6.175>

L02-17

A 4-way multiplexer

```
case {s1,s0} matches
  0: A;
  1: B;
  2: C;
  3: D;
endcase
```



September 11, 2015

<http://csg.csail.mit.edu/6.175>

L02-18

An w-bit Ripple-Carry Adder

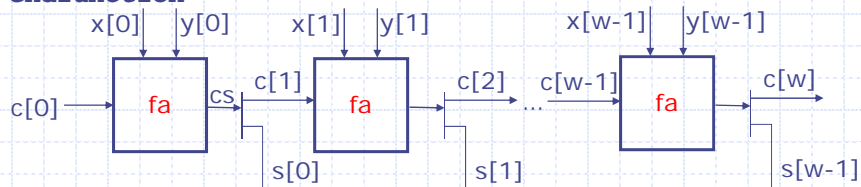
```

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

```

Not quite correct

Unfold the loop to get the wiring diagram



September 11, 2015

<http://csg.csail.mit.edu/6.175>

L02-19

Instantiating the parametric Adder

```

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

```

Now we define 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);

```

```

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

```

September 11, 2015

<http://csg.csail.mit.edu/6.175>

L02-20

valueOf(w) VERSUS w

- ◆ Each expression has a type and a value and these come from two entirely disjoint worlds
- ◆ w in $\text{Bit}\#(w)$ 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
 - $i < \text{valueOf}(w)$ is type correct

September 11, 2015

<http://csg.csail.mit.edu/6.175>

L02-21

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)`, ...

September 11, 2015

<http://csg.csail.mit.edu/6.175>

L02-22

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

September 11, 2015

<http://csg.csail.mit.edu/6.175>

L02-23

Static Elaboration phase

- ◆ When BSV programs are compiled, first type checking is done and then the compiler gets rid of many constructs which have no direct hardware meaning, like Integers, loops

```
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
```

```
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];
```

September 11, 2015

<http://csg.csail.mit.edu/6.175>

L02-24

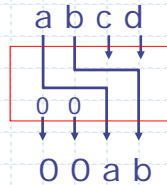
Integer versus Int# (32)

- ◆ 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

```
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
```

Shift operators

Logical right shift by 2



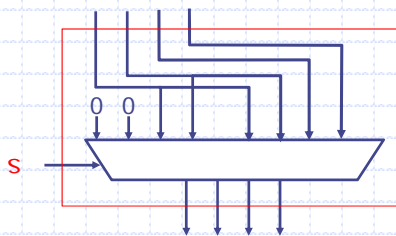
- ◆ Fixed size shift operation is cheap in hardware – just wire the circuit appropriately
- ◆ Rotate, sign-extended shifts – all are equally easy

September 11, 2015

<http://csg.csail.mit.edu/6.175>

L02-27

Conditional operation: shift versus no-shift



- ◆ We need a mux to select the appropriate wires: if **s** is one the mux will select the wires on the left otherwise it would select wires on the right

```
(s==0)?{a,b,c,d}:{0,0,a,b};
```

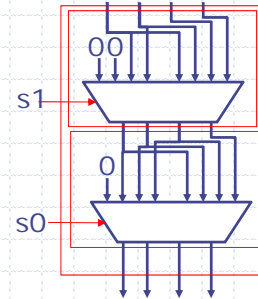
September 11, 2015

<http://csg.csail.mit.edu/6.175>

L02-28

Logical right shift by n

- ◆ Shift n can be broken down in $\log n$ steps of fixed-length shifts of size 1, 2, 4, ...
 - Shift 3 can be performed by doing a shift 2 and shift 1
- ◆ We need a mux to omit a particular size shift
- ◆ Shift circuit can be expressed as $\log n$ nested conditional expressions



September 11, 2015

<http://csg.csail.mit.edu/6.175>

L02-29

A digression on types

- ◆ Suppose we have a variable c whose values can represent three different colors
 - We can declare the type of c to be `Bit#(2)` and say that 00 represents Red, 01 Blue and 10 Green
- ◆ A better way is to create a new type called `Color` as follows:

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

Types prevent us from mixing bits that represent color from raw bits

The compiler will automatically assign some bit representation to the three colors and also provide a function to test if the two colors are equal. If you do not use "deriving" then you will have to specify the representation and equality

September 11, 2015

<http://csg.csail.mit.edu/6.175>

L02-30

Enumerated types

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

typedef enum {Eq, Neq, Le, Lt, Ge, Gt, AT, NT}
BrFunc deriving(Bits, Eq);

typedef enum {Add, Sub, And, Or, Xor, Nor, Slt, Sltu,
LShift, RShift, Sra}
AluFunc deriving(Bits, Eq);
```

Each enumerated type defines a new type

Combinational ALU

```
function Data alu(Data a, Data b, AluFunc func);
  Data res = case(func)
    Add   : (a + b);
    Sub   : (a - b);
    And   : (a & b);
    Or    : (a | b);
    Xor   : (a ^ b);
    Nor   : ~(a | b);
    Slt   : zeroExtend( pack( signedLT(a, b) ) );
    Sltu  : zeroExtend( pack( a < b ) );
    LShift: (a << b[4:0]);
    RShift: (a >> b[4:0]);
    Sra   : signedShiftRight(a, b[4:0]);
  endcase;
  return res;
endfunction
```

Given an implementation of the primitive operations like `addN`, `Shift`, etc. the ALU can be implemented simply by introducing a mux controlled by `op` to select the appropriate circuit

Comparison operators

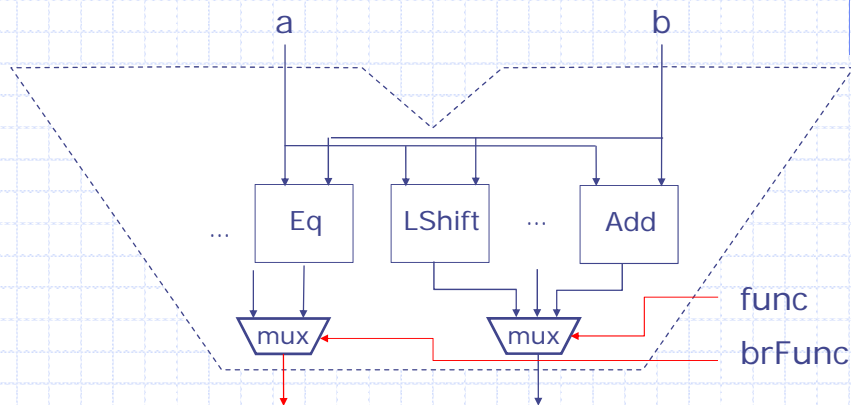
```
function Bool aluBr(Data a, Data b, BrFunc brFunc);
  Bool brTaken = case(brFunc)
    Eq : (a == b);
    Neq : (a != b);
    Le : signedLE(a, 0);
    Lt : signedLT(a, 0);
    Ge : signedGE(a, 0);
    Gt : signedGT(a, 0);
    AT : True;
    NT : False;
  endcase;
  return brTaken;
endfunction
```

September 11, 2015

<http://csg.csail.mit.edu/6.175>

L02-33

ALU including Comparison operators



September 11, 2015

<http://csg.csail.mit.edu/6.175>

L02-34