Constructive Computer Architecture

Combinational circuits

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

Content

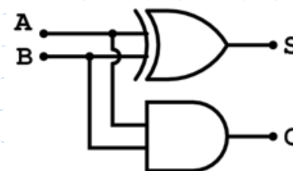
- Design of a combinational ALU starting with primitive gates And, Or and Not
- Combinational circuits as acyclic wiring diagrams of primitive gates
- Introduction to BSV
  - Intro to types – enum, typedefs, numeric types, Int#(32) vs Integer, Bool vs Bit#(1), Vector
  - Simple operations: concatenation,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

Combinational circuits are acyclic interconnections of gates

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

Half Adder

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



Boolean equations

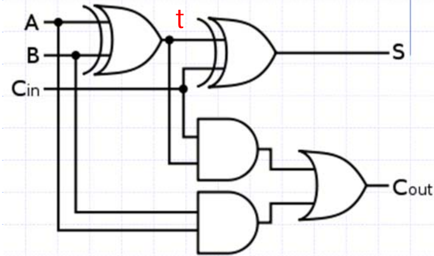
```
s = (~a·b) + (a·~b)
c = a·b
```

"Optimized"

```
s = a ⊕ b
```

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

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

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

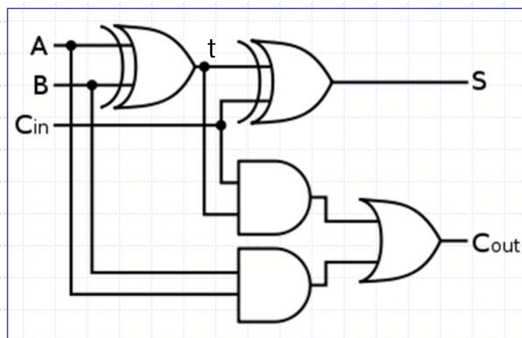
"Optimized"

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

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
```

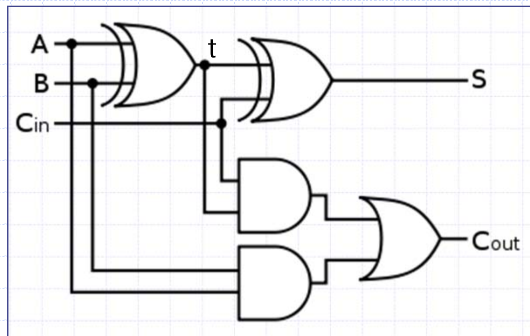
Structural code – only specifies interconnection between boxes



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}`?

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;

typedef Bit#(32) Word;

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

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

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

Type declaration versus deduction

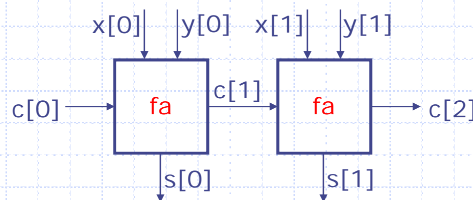
- The programmer writes down types of some expressions in a program and the compiler deduces the types of the rest of expressions
- If the type deduction cannot be performed or the type declarations are inconsistent then the compiler complains

```
function Bit#(2) fa(Bit#(1) a, Bit#(1) b,
                              Bit#(1) c_in);
   Bit#(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

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

"let" syntax

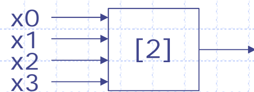
- The "let" syntax: No need to write the type if the compiler can deduce it:
  - `let cs0 = fa(x[0], y[0], c[0]);`
  - `Bit#(2) cs0 = fa(x[0], y[0], c[0]);`

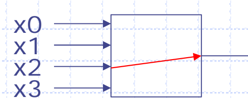
The same

Selecting a wire: x[i]

assume x is 4 bits wide

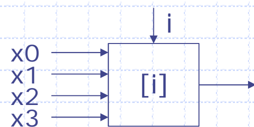
- Constant Selector: e.g., x[2]

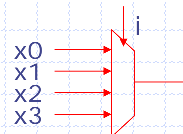




no hardware; x[2] is just the name of a wire

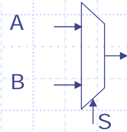
- Dynamic selector: x[i]

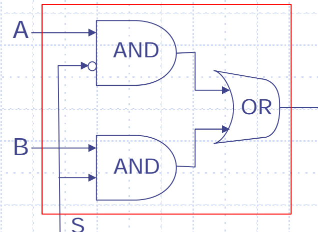




4-way mux

A 2-way multiplexer





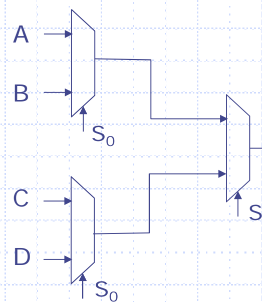

```
(s==0)? A : B
```

Gate-level implementation

Conditional expressions are also synthesized using muxes

A 4-way multiplexer

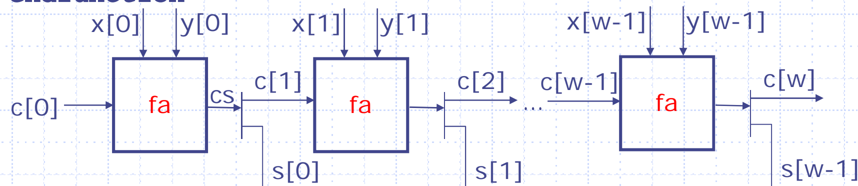
```
case {s1,s0} matches
    0:  A;
    1:  B;
    2:  C;
    3:  D;
endcase
```



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
```

Unfold the loop to get the 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);
```

`valueOf(w)` versus `w`

- Each expression has a type and a value and these come from two 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, e.g., `i<w`
  - But `i<w` is not type correct
- The function `valueOf` allows us to lift a numeric type to a value
  - Making `i<valueOf(w)` type correct

`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

```
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
```

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

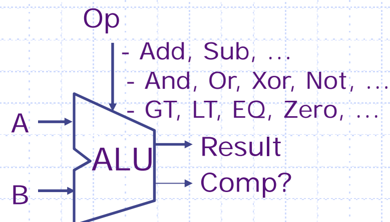
Static Elaboration phase

- 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];
```


Arithmetic-Logic Unit (ALU)



ALU performs all the arithmetic and logical functions

Each individual function can be described as a combinational circuit