Constructive Computer Architecture

Combinational circuits-2

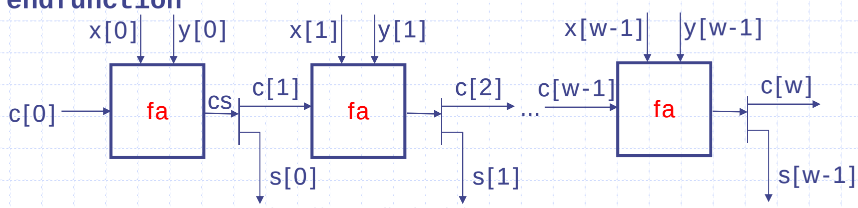
Arvind
Computer Science & Artificial Intelligence Lab.
Massachusetts Institute of Technology

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



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. The function valueOf allows us to do that
  - Thus
    - i<w is not type correct
    - i<valueOf(w)is 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
```

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

Static Elaboration phase

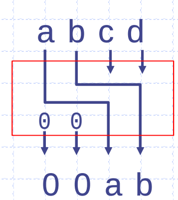
- When BSV programs are compiled, first type checking is done and then the compiler gets rid of many constructs which have no direct hardware meaning, like Integers, loops

```
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];
```

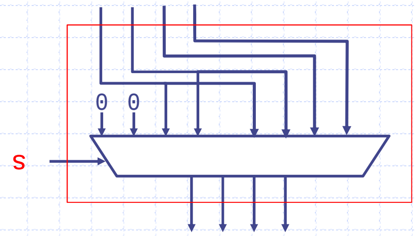
Shift operators

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

Conditional operation: shift versus no-shift

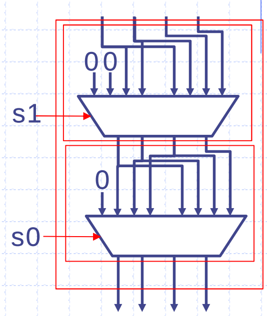


- We need a mux to select the appropriate wires: if s is one the mux will select the wires on the left otherwise it would select wires on the right

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


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



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

Enumerated types

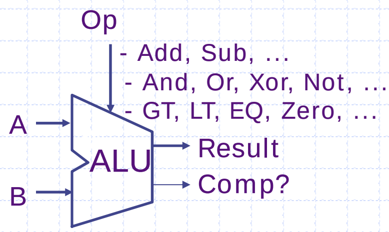
```
typedef enum {Red, Blue, Green}
Color deriving(Bits, Eq);

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

Arithmetic-Logic Unit (ALU)



ALU performs all the arithmetic and logical functions

Each individual function can be described as a combinational circuit

Combinational ALU

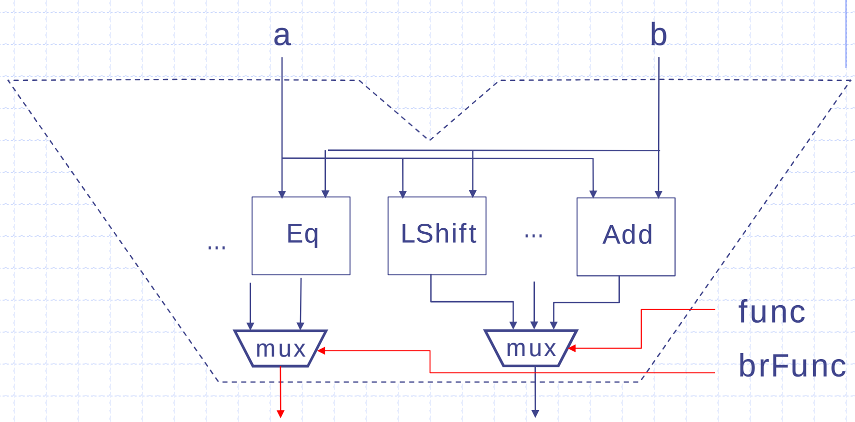
```
function Data alu(Data a, Data b, AluFunc func);
  Data res = case(func)
     Add    : (a + b);
     Sub    : (a - b);
     And    : (a & b);
     Or     : (a | b);
     Xor    : (a ^ b);
     Nor    : ~(a | b);
     Slt    : zeroExtend( pack( signedLT(a, b) ) );
     Sltu   : zeroExtend( pack( a < 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
```


ALU including Comparison operators



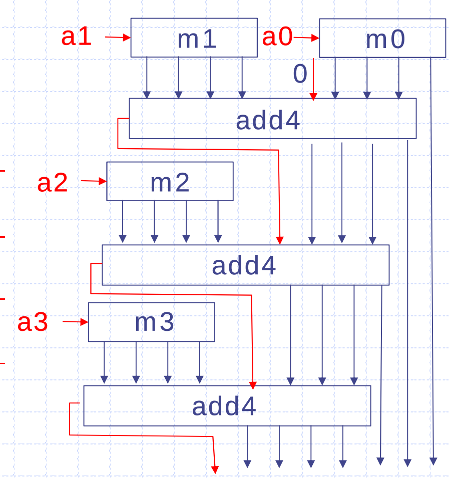
Complex Combinational Circuits

Multiplication by repeated addition

```
b Multiplicand   1101   (13)
a Mulitplier  *  1011   (11)

tp               0000
m0        +      1101
tp              01101
m1        +     1101
tp             100111
m2        +    0000
tp            0100111
m3        +   1101
tp           10001111   (143)
```

`mi = (a[i]==0)? 0 : b;`



Combinational 32-bit multiply

```
function Bit#(64) mul32(Bit#(32) a, Bit#(32) b);
  Bit#(32) tp = 0;
  Bit#(32) prod = 0;
  for(Integer i = 0; i < 32; i = i+1)
  begin
     Bit#(32) m   = (a[i]==0)? 0 : b;
     Bit#(33) sum = add32(m,tp,0);
     prod[i]      = sum[0];
     tp           = sum[32:1];
  end
  return {tp,prod};
endfunction
```

Combinational circuit uses 31 add32 circuits

We can reuse the same add32 circuit if we store the partial results in a *register*

Design issues with combinational multiply

- Lot of hardware
  - 32-bit multiply uses 31 add32 circuits
- Long chains of gates
  - 32-bit ripple carry adder has a 31-long chain of gates
  - 32-bit multiply has 31 ripple carry adders in sequence! Total delay ? 2(n-1) FAs?

The speed of a combinational circuit is determined by its longest input-to-output path

Can we do better?

Yes – Sequential Circuits; Circuits with state