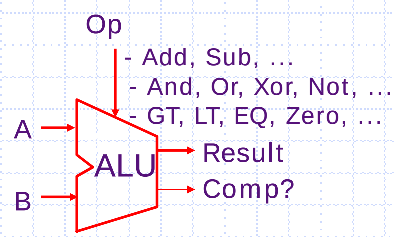
6.S078 - Computer Architecture:
A Constructive Approach

Combinational ALU

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

Arithmetic-Logic Unit (ALU)



ALU performs all the arithmetic and logical functions

We will first implement individual functions and then combine them to form an ALU

A w-bit Ripple-Carry Adder

```
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

Integer versus Int#(32)

- In mathematics integers are unbounded but in computer systems integers always have a fixed size
- Bluespec 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
```

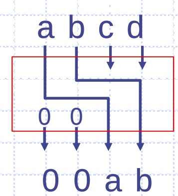
Static Elaboration phase

- When Bluespec program 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
```

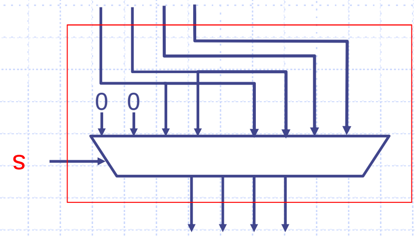
```
cs0 = fa(x[0], y[0], c[0]); c[1]=cs0[1]; s[0]=cs0[0];
cs1 = fa(x[1], y[1], c[1]); c[2]=cs1[1]; s[1]=cs1[0];
…
csw = fa(x[valw-1], y[valw-1], c[valw-1]);
      c[valw] = csw[1]; s[valw-1] = csw[0];
```


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

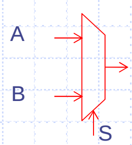
Conditional operation: shift versus no-shift

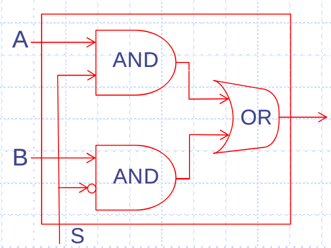


- We need a Mux to select the appropriate wires: if s is one the Mux will select the wires on the left otherwise it would select wires on the right

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


A 2-way multiplexer



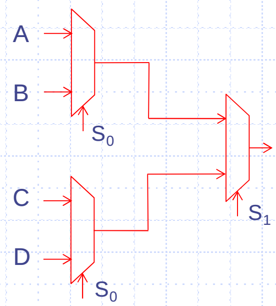


(s==0)?A:B

Gate-level implementation

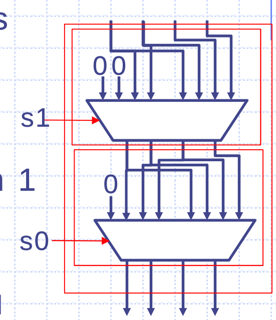
A 4-way multiplexer

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



Logical right shift by *n*

- Shift *n* can be broken down in several steps of fixed-length shifts of 1, 2, 4, ...
- Thus a shift of size 3 can be performed by first doing a shift of length 2 and then a shift of length 1
- We need a Mux to select whether we need to shift at all
  - The whole structure can be expressed an a bunch of nested conditional expressions



You will write a Blusepec program to produce a variable size shifter in Lab 1

Type synonyms

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

typedef Bit#(8) Byte;

typedef Bit [31:0] Word;

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

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

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


Enumerated types

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

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

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

Structure type

Example: Complex Addition

```
typedef struct{
  Int#(t) r;
  Int#(t) i;
} Complex#(numeric type t) deriving (Eq,Bits);

function Complex#(t) \+
  (Complex#(t) x, Complex#(t) y);
 Int#(t) real = x.r + y.r;
 Int#(t) imag = x.i + y.i;
 return(Complex{r:real, i:imag});
endfunction
```

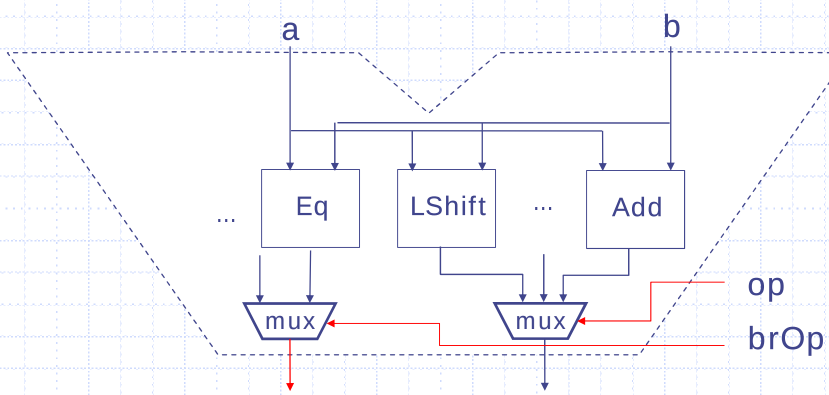

Combinational ALU

```
function Bit#(width) alu(Bit#(width) a,
                         Bit#(width) b, AluFunc op);
  Bit#(width) res = case(op)
        Add   : add(a, b);
        Sub   : subtract(a, b);
        And   : (a & b);
        Or    : (a | b);
        Xor   : (a ^ b);
        Nor   : ~(a | b);
        Slt   : setLessThan(a, b);
        Sltu  : setLessThanUnsigned(a, b);
        LShift: logicalShiftLeft(a, b[4:0]);
        RShift: logicalShiftRight(a, b[4:0]);
        Sra   : signedShiftRight(a, b[4:0]);
    endcase;
    return res;
endfunction
```

Comparison operators

```
function Bool aluBr(Bit#(width) a,
                    Bit#(width) b, BrType brOp);
  Bool brTaken = case(brOp)
      Eq   : (a == b);
      Neq  : (a != b);
      Le   : signedLE(a, 0);
      Lt   : signedLT(a, 0);
      Ge   : signedGE(a, 0);
      Gt   : signedGT(a, 0);
  endcase;
  return brTaken;
endfunction
```

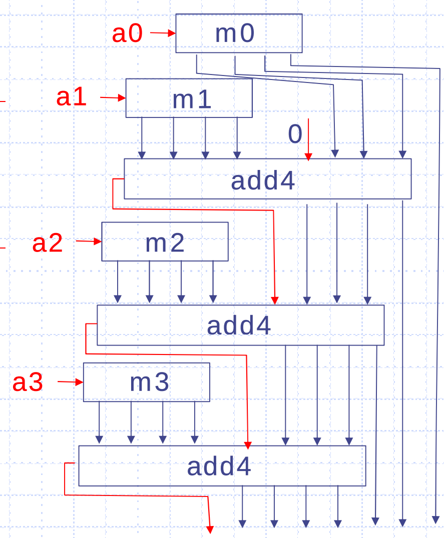

ALU including Comparison operators



Complex combinational circuits

Multiplication by repeated addition

```
b Multiplicand    1101   (13)
a Muliplier    *  1011   (11)
                  1101
             +   1101
             +  0000
             + 1101
               10001111  (143)
```



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

Combinational 32-bit multiply

```
function Bit#(64) mul32(Bit#(32) a, Bit#(32) b);
      Bit#(32) prod = 0;
      Bit#(32) tp = 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 = truncateLSB(sum);
  end
  return {tp,prod};
endfunction
```


Combinational n-bit multiply

```
function Bit#(TAdd#(w,w)) mulN(Bit#(w) a, Bit#(w) b);
      Bit#(w) prod = 0;
      Bit#(w) tp = 0;
  for(Integer i = 0; i < valueOf(w); i = i+1)
  begin
     Bit#(w) m = (a[i]==0)? 0 : b;
     Bit#(TAdd#(w,1)) sum = addN(m,tp,0);
     prod[i] = sum[0];
     tp = truncateLSB(sum);
  end
  return {tp,prod};
endfunction
```

Design issues with combinational multiply

- Lot of hardware
  - 32-bit multiply uses 31 addN 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!

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

next time: Sequential circuits