Constructive Computer Architecture

Combinational ALU

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

Outline

- Building complex combinational circuits in pieces
- Parameterization that goes beyond data path widths
Arithmetic-Logic Unit (ALU)

ALU performs all the arithmetic and logical functions

```
function Data alu(Data a, Data b, AluFunc func);
```

Each individual function can be described as a combinational circuit and these can be combined together to produce a combinational ALU

---

ALU for comparison operators

Like ALU but returns a Bool

```
function Bool aluBr(Data a, Data b, BrFunc func);
```

what does func look like?
Enumerated types

- Suppose we have a variable \( c \) whose values can represent three different colors.
  - Declare the type of \( c \) to be \( \text{Bit#(2)} \) and adopt the convention that 00 represents Red, 01 Blue and 10 Green.
- A better way is to create a new type called \( \text{Color} \):

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

  BSV compiler automatically assigns a bit representation to the three colors and provides a function to test if the two colors are equal.
  
  If you do not use "deriving" then you will have to specify your own encoding and equality function.

Enumerated types

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

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

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

```plaintext
function Data alu(Data a, Data b, AluFunc func);
    Data res = case(func)
        Add : addN(a,b);
        Sub : subN(a,b);
        And : andN(a,b);
        Or  : orN(a,b);
        Xor : xorN(a,b);
        Nor : norN(a,b);
        Slt : zeroExtend(pack(signedLT(a,b)));
        Sltu : zeroExtend(pack(lt(a,b)));
        LShift: shiftLeft(a,b[4:0]);
        RShift: shiftRight(a,b[4:0]);
        Sra : signedShiftRight(a,b[4:0]);
    endcase;
    return res;
endfunction
```

Comparison operators

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

Selectors and Multiplexers
Selecting a wire: $x[i]$

- **Constant Selector**: e.g., $x[2]$
  - No hardware; $x[2]$ is just the name of a wire

- **Dynamic selector**: $x[i]$
  - 4-way mux

Assume $x$ is 4 bits wide

A 2-way multiplexer

(s==0)? A : B ;

Gate-level implementation
A 4-way multiplexer

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

Shift operators
Logical right shift by 2

Fixed size shift operation is cheap in hardware
- just wire the circuit appropriately

Other types of shifts are similar

Logical right shift by \( n \)

Suppose we want to build a shifter which shift a value \( x \) by \( n \) where \( n \) is between 0 and 31

One way to do this is by connecting 31 different shifters via a mux
Logical right shift by $n$

- Shift $n$ can be broken down in $\log n$ steps of fixed-length shifts of size 1, 2, 4, ...
  - For example, we can perform Shift 3 (=2+1) by doing shifts of size 2 and 1
  - Shift 5 (=4+1) by doing shifts of size
  - Shift 21 (=16+4+1) by doing shifts of size
- For a 32-bit number, a 5-bit $n$ can specify all the needed shifts
  - $3_{10} = 00011_{2}$, $5_{10} = 00101_{2}$, $21_{10} = 10101_{2}$
- The bit encoding of $n$ tells us which shifters are needed; if the value of the $i^{th}$ (least significant) bit is 1 then we need to shift by $2^i$ bits

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 ckt

- Define log $n$ shifters of sizes 1, 2, 4, ...
- Define log $n$ muxes to perform a particular size shift
- Shift circuit can be expressed as log $n$ nested conditional expressions where s0, s1 ...
  Represent the bits of $n$
### Multiplication by repeated addition

<table>
<thead>
<tr>
<th>b Multiplicand</th>
<th>1101</th>
<th>(13)</th>
</tr>
</thead>
<tbody>
<tr>
<td>a Multiplier</td>
<td>1011</td>
<td>(11)</td>
</tr>
<tr>
<td>0000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>1101</td>
<td></td>
</tr>
<tr>
<td>m0</td>
<td></td>
<td>1101</td>
</tr>
<tr>
<td>m1</td>
<td>+</td>
<td>1101</td>
</tr>
<tr>
<td>m2</td>
<td>+</td>
<td>0000</td>
</tr>
<tr>
<td>m3</td>
<td>+</td>
<td>1101</td>
</tr>
<tr>
<td>m</td>
<td></td>
<td>10001111</td>
</tr>
</tbody>
</table>

At each step we add either 1101 or 0 to the result depending upon a bit in the multiplier:

\[ m_i = (a[i]==0)? 0 : b; \]

We also shift the result by one position at every step.

However, our addN circuit adds only two numbers at a time!

---

### Multiplication by repeated addition cont.

<table>
<thead>
<tr>
<th>b Multiplicand</th>
<th>1101</th>
<th>(13)</th>
</tr>
</thead>
<tbody>
<tr>
<td>a Multiplier</td>
<td>1011</td>
<td>(11)</td>
</tr>
<tr>
<td>tp</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>m0</td>
<td>+</td>
<td>1101</td>
</tr>
<tr>
<td>tp</td>
<td>01101</td>
<td></td>
</tr>
<tr>
<td>m1</td>
<td>+</td>
<td>1101</td>
</tr>
<tr>
<td>tp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>m2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>m3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tp</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

At each step we add either 1101 or 0 to the result depending upon a bit in the multiplier:

\[ m_i = (a[i]==0)? 0 : b; \]

We also shift the result by one position at every step.
Multiplication by repeated addition ckt

b Multiplicand 1101 \text{(13)}
a Multiplier * 1011 \text{(11)}

\[\begin{array}{c}
tp & 0000 \\
m0 + & 1101 \\
tp & 01101 \\
m1 + & 1101 \\
tp & 100111 \\
m2 + & 0000 \\
tp & 0100111 \\
m3 + & 1101 \\
tp & 1000111 \text{(143)} \\
m0 = (a[i]==0)? 0 : b;
\end{array}\]

Combinational 32-bit multiply

```markdown
function \text{Bit\#(64) mul32(\text{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
```

- Long chains of gates
  - 32-bit multiply has 32 ripple carry adders in sequence!
  - 32-bit ripple carry adder has a 32-long chain of gates
  - Total delay ?