A 32-bit ALU

- A ripple carry ALU
- Two bits decide operation
  - Add/Sub
  - AND
  - OR
  - LESS
- 1 bit decide add/sub operation
- A carry in bit
- Bit 31 generates overflow and set bit

Problem: ripple carry adder is slow

- Is a 32-bit ALU as fast as a 1-bit ALU?
- Is there more than one way to do addition?
  - two extremes: ripple carry and sum-of-products

Can you see the ripple? How could you get rid of it?

\[ c_1 = b_0c_0 + a_0c_0 + a_0b_0 \]
\[ c_2 = b_1c_1 + a_1c_1 + a_1b_1 \]
\[ c_3 = b_2c_2 + a_2c_2 + a_2b_2 \]
\[ c_4 = b_3c_3 + a_3c_3 + a_3b_3 \]

Not feasible! Why?

Carry look-ahead adder

- An approach in-between our two extremes
- Motivation:
  - If we didn’t know the value of carry-in, what could we do?
  - When would we always generate a carry? \( g_i = a_i \lor b_i \)
  - When would we propagate the carry? \( p_i = a_i + b_i \)
  - Did we get rid of the ripple?

\[
\begin{align*}
  c_1 &= g_0 + p_0c_0 \\
  c_2 &= g_1 + p_1c_1 + p_1p_0c_0 \\
  c_3 &= g_2 + p_2c_2 + p_2p_1g_1 + p_2p_1p_0c_0 \\
  c_4 &= g_3 + p_3c_3 + p_3p_2g_2 + p_3p_2p_1g_1 + p_3p_2p_1p_0c_0
\end{align*}
\]

Feasible! Why?

Use principle to build bigger adders

- A 16-bit adder uses four 4-bit adders
- It takes block g and p terms and cin to generate block carry bits out
- Block carries are used to generate bit carries
- could use ripple carry of 4-bit CLA adders
- Better: use the CLA principle again!

Delays in carry look-ahead adders

- 4-bit case
  - Generation of g and p: 1 gate delay
  - Generation of carries (and G and P): 2 more gate delay
  - Generation of sum: 1 more gate delay
- 16-bit case
  - Generation of g and p: 1 gate delay
  - Generation of block G and P: 2 more gate delay
  - Generation of block carries: 2 more gate delay
  - Generation of bit carries: 2 more gate delay
  - Generation of sum: 1 more gate delay
- 64-bit case
  - 12 gate delays
Multiplication

- More complicated than addition
  - accomplished via shifting and addition
- More time and more area
- Let's look at 3 versions based on grade school algorithm

### Multiplicand
01010010

### Multiplier
01101101

- Negative numbers: convert and multiply
- Use other better techniques like Booth’s encoding

### Multiplication: Implementation

1. Test
2. Shift the Multiplicand register left 1 bit
3. Shift the Multiplier register right 1 bit

#### Start

- **Multiplier0 = 0**
- **Multiplier0 = 1**

#### No:
- < 32 repetitions

#### Yes:
- 32 repetitions

#### 64-bit ALU

#### Control test

- **Multiplier**
- **Product**
- **Write**
- **Shift left**
- **64 bits**

### Second Version

1. Test
2. Shift the Product register right 1 bit
3. Shift the Multiplier register right 1 bit

#### Start

- **Product0 = 0**
- **Product0 = 1**

#### No:
- < 32 repetitions

#### Yes:
- 32 repetitions

#### 32-bit ALU

#### Control test

- **Multiplier**
- **Product**
- **Write**
- **Shift right**
- **32 bits**

### Final Version

1. Test
2. Shift the Product register right 1 bit

#### Start

- **Product0 = 0**
- **Product0 = 1**

#### No:
- < 32 repetitions

#### Yes:
- 32 repetitions

#### 32-bit ALU

#### Control test

- **Multiplier**
- **Product**
- **Write**
- **Shift right**
- **32 bits**

### Multiplication Example

<table>
<thead>
<tr>
<th>Iteration</th>
<th>multiplicand</th>
<th>Original algorithm</th>
<th>Step</th>
<th>Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0010</td>
<td>Initial values</td>
<td>0000</td>
<td>0110</td>
</tr>
<tr>
<td>1</td>
<td>0010</td>
<td>1: 0 = no operation</td>
<td>0000</td>
<td>0110</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2: Shift right</td>
<td>0000</td>
<td>0111</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>1a: 1 = prod = Prod + Mcand</td>
<td>0010</td>
<td>0011</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2: Shift right</td>
<td>0001</td>
<td>0011</td>
</tr>
<tr>
<td>3</td>
<td>0010</td>
<td>1a: 1 = prod = Prod + Mcand</td>
<td>0011</td>
<td>0001</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2: Shift right</td>
<td>0001</td>
<td>0000</td>
</tr>
<tr>
<td>4</td>
<td>0010</td>
<td>1: 0 = no operation</td>
<td>0001</td>
<td>0000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2: Shift right</td>
<td>0000</td>
<td>1100</td>
</tr>
</tbody>
</table>
Signed Multiplication

- Let Multiplier be Q[n-1:0], multiplicand be M[n-1:0]
- Let F = 0 (shift flag)
- Let result A[n-1:0] = 0…00
- For n-1 steps do
  - F <= F .or. (M[n-1] .and. Q[0]) /* determine shift bit */
  - Shift A and Q with F, i.e.,
- Do the correction step
  - Shift A and Q while retaining A[n-1]
- This works in all cases except when both operands are 10..00

Booth’s Encoding

- Numbers can be represented using three symbols, 1, 0, and -1
  - One representation is 111111
  - Another possible one 0000000
  - Another example +14
    - One representation is 0000111
    - Another possible one 0000100
  - We do not explicitly store the sequence
  - Look for transition from previous bit to next bit
  - 0 to 0 is 0; 0 to 1 is -1; 1 to 1 is 0; and 1 to 0 is 1
  - Multiplication by 1, 0, and -1 can be easily done
  - Add all partial results to get the final answer

Using Booth’s Encoding for Multiplication

- Convert a binary string in Booth’s encoded string
- For n bit by n-bit multiplication, n/2 partial product
- Partial products are signed and obtained by multiplying the multiplicand by 0, +1, +2, and -2 (all achieved by shift)
- Add partial products to obtain the final result
- Example, multiply 0111 (+7) by 1010 (-6)
- Booths encoding of 1010 is -1 +1 -1 0
- With 2-bit groupings, multiplication needs to be carried by -1 and -2
  - 1111 0010 (multiplication by -1)
  - 1111 0110 (multiplication by -2)
- Add the two partial products to get 11010110 (-42) as result

Booth’s algorithm (Neg. multiplier)

<table>
<thead>
<tr>
<th>Iteration</th>
<th>multiplicand</th>
<th>Booth’s algorithm</th>
<th>Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0010</td>
<td>initial values</td>
<td>0000 1101 0</td>
</tr>
<tr>
<td>1</td>
<td>0010</td>
<td>1c: 10 x prod = Prod - M cud</td>
<td>1110 1101 0</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>2: Shift right Prod</td>
<td>1111 0101 0</td>
</tr>
<tr>
<td>3</td>
<td>0010</td>
<td>1h: 01 x prod = Prod + M cud</td>
<td>0000 0110 1</td>
</tr>
<tr>
<td>4</td>
<td>0010</td>
<td>2: Shift right Prod</td>
<td>0090 0111 0</td>
</tr>
<tr>
<td>5</td>
<td>0010</td>
<td>1d: 11 x no operation</td>
<td>1111 0101 1</td>
</tr>
<tr>
<td>6</td>
<td>0010</td>
<td>2: Shift right Prod</td>
<td>1111 1010 1</td>
</tr>
</tbody>
</table>

Carry Save Addition

- Consider adding six set of numbers (4 bits each in the example)
- The numbers are 1001, 0110, 1111, 0111, 1010, 0110 (all positive)
- One way is to add them pair wise, getting three results, and then adding them again
  - 1001 1111 1010 0111 011001
  - 0110 0111 0110 0110 000000
  - 0111 0110 1010 0110 010101
- Other method is add them three at a time by saving carry
  - 1001 0111 00000 010101 001101
  - 0110 1010 1111 010000 201000
  - 1111 0110 0111 000101 310010
  - 00000 010110 001010 010000 000110
  - 11110 01100 010101 010000 CARRY

Carry Save Addition for Multiplication

- n-bit carry-save adder take 1FA time for any n
- For n x n bit multiplication, n or n/2 (for 2 bit at time Booth’s encoding) partial products can be generated
- For n partial products n3 n-bit carry save adders can be used
- This yields 2n/3 partial results
- Repeat this operation until only two partial results are remaining
- Add them using an appropriate size adder to obtain 2n bit result
- For n=32, you need 30 carry save adders in eight stages taking 8T time where T is time for one-bit full adder
- Then you need one carry-propagate or carry-look-ahead adder
Division

- Even more complicated
  - can be accomplished via shifting and addition/subtraction
- More time and more area
- We will look at 3 versions based on grade school algorithm

0011 / 0010 0010 (Dividend)

- Negative numbers: Even more difficult
- There are better techniques, we won’t look at them

Division, First Version

Division, Second Version

Division, Final Version

### Restoring Division

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Dividend</th>
<th>Divide algorithm</th>
<th>Step</th>
<th>Remainder</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0010</td>
<td>Initial values</td>
<td>0000 0110</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0010</td>
<td>Shift Rem left</td>
<td>1110 1110</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>2. Rem = Rem - Div</td>
<td>1111 1111</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0010</td>
<td>3b. Rem &lt; 0 ? all R, R0 = 0</td>
<td>0001 0001</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0010</td>
<td>2. Rem = Rem - Div</td>
<td>0001 0001</td>
<td></td>
</tr>
<tr>
<td>Done</td>
<td>0010</td>
<td>Shift left half of Rem right 1</td>
<td>0001 0011</td>
<td></td>
</tr>
</tbody>
</table>

### Non-Restoring Division

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Dividend</th>
<th>Divide algorithm</th>
<th>Step</th>
<th>Remainder</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0010</td>
<td>Initial values</td>
<td>0000 1110</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0010</td>
<td>1. Rem = Rem - Div</td>
<td>1110 1110</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>2b. Rem &lt; 0 ? all R, R0 = 0</td>
<td>1111 1100</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0010</td>
<td>3b. Rem = Rem + Div</td>
<td>0001 0001</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0010</td>
<td>2a. Rem &gt; 0 ? all R, R0 = 1</td>
<td>0010 0011</td>
<td></td>
</tr>
<tr>
<td>Done</td>
<td>0010</td>
<td>Shift left half of Rem right 1</td>
<td>0001 0011</td>
<td></td>
</tr>
</tbody>
</table>
Floating Point (a brief look)

- We need a way to represent
  - numbers with fractions, e.g., 3.1416
  - very small numbers, e.g., 0.00000001
  - very large numbers, e.g., 3.15576 \times 10^9
- Representation:
  - sign, exponent, significand: \((-1)^\text{sign} \times \text{significand} \times 2^{\text{exponent}}\)
  - more bits for significand gives more accuracy
  - more bits for exponent increases range
- IEEE 754 floating point standard:
  - single precision: 8 bit exponent, 23 bit significand
  - double precision: 11 bit exponent, 52 bit significand

IEEE 754 floating point standard

- Leading "1" bit of significand is implicit
- Exponent is "biased" to make sorting easier
  - all 0s is smallest exponent all 1s is largest
  - bias of 127 for single precision and 1023 for double precision
  - summary: \((-1)^\text{sign} \times (1 + \text{significand}) \times 2^{\text{exponent} - \text{bias}}\)
- Example:
  - decimal: -0.75 = -3/4 = -3/2^2
  - binary: -0.11 = -1.1 \times 2^{-1}
  - floating point: exponent = 126 = 01111110
  - IEEE single precision: 10111111010000000000000000000000

Floating Point Complexities

- Operations are somewhat more complicated (see text)
- In addition to overflow we can have "underflow"
- Accuracy can be a big problem
  - four rounding modes
    - positive divided by zero yields "infinity"
    - zero divide by zero yields "not a number"
  - other complexities
  - Implementing the standard can be tricky
  - Not using the standard can be even worse
    - see text for description of 80x86 and Pentium bug!

Floating Point Add/Sub

- To add/sub two numbers
  - We first compare the two exponents
  - Select the higher of the two as the exponent of result
  - Select the significand part of lower exponent number and shift it right by the amount equal to the difference of two exponents
  - Remember to keep two shifted out bit and a guard bit
  - Add/sub the signifand as required according to operation and signs of operands
  - Normalize significand of result adjusting exponent
  - Round the result (add one to the least significant bit to be retained if the first bit being thrown away is a 1
  - Re-normalize the result

Floating Point Multiply

- To multiply two numbers
  - Add the two exponent (remember access 127 notation)
  - Produce the result sign as exor of two signs
  - Multiple significand portions
  - Results will be 1.xxxxx or 0.1.xxxxx
  - In the first case shift result right and adjust exponent
  - Round off the result
  - This may require another normalization step

Floating Point Divide

- To divide two numbers
  - Subtract divisor's exponent from the dividend's exponent (remember access 127 notation)
  - Produce the result sign as exor of two signs
  - Divide dividend's significand by divisor's significand portions
  - Results will be 1.xxxxx or 0.1.xxxxx
  - In the second case shift result left and adjust exponent
  - Round off the result
  - This may require another normalization step