Topic 7: Carry Lookahead & Multipliers

- Propagate & Generate
- Carry Lookahead, version 1
- Super-Propagate & Super-Generate
- Carry Lookahead, version 2
- Multiplication
- Optimizing Multiplication
The Problem

- Ripple Carry is Slow
  - 32 bits
  - Gets worse on 64-bit!

- Max CPU speed:
  - Speed of slowest circuit
An Inspiration

- Sum-of-Products is fast!
  - Cost of each AND:
    log in the # of inputs
  - Cost of OR:
    log in the # of ANDs

An 8-input AND gate has a path length of 3, in our simplified model.
Terminology

- $a_i$, $b_i$  
  Our inputs, at bit $i$

- $c_i$ 
  The carry bit $i$
  $\text{carryIn for column } i$
  $\text{carryOut for column } (i-1)$

- $c_0$ 
  $\text{carryIn to the whole adder (bNegate)}$
Group Exercise:

Suppose that you wanted to write a sum-of-products to calculate \( c_2 \), given the inputs \( c_0, a_0, b_0, a_1, b_1 \). (Carry-in, plus 2 columns of input bits.)

A truth table would be too long to do by hand (32 rows). But can you find some ways to simplify the problem? For instance, what combinations of \( a_1 \), \( b_1 \) will force a carry, no matter what else is true?
This sum-of-products calculates the $c_2$, which is the carryOut value from bit 1 in an adder.
Column 1 **forces** a carry, no matter what happens in Column 0.
Column 0 **generates** a carry. Column 1 keeps it going because of $a_1$.

Column 1 **forces** a carry, no matter what happens in Column 0.

Column 0 **generates** a carry. Column 1 keeps it going because of $a_1$.

Column 0 **generates** a carry. Column 1 keeps it going because of $a_1$. 

---

**Diagram Description:**

The diagram consists of logic gates and input variables $a_0, b_0, a_1, b_1, c_0, a_1 \cdot a_0 \cdot b_0, a_1 \cdot b_1, b_1 \cdot a_0 \cdot b_0, a_1 \cdot a_0 \cdot c_0, a_1 \cdot b_0 \cdot c_0, b_1 \cdot a_0 \cdot c_0, b_1 \cdot b_0 \cdot c_0$, and the output variable $c_2$.
Column 0 generates a carry. Column 1 keeps it going because of $a_1$.

Column 1 forces a carry, no matter what happens in Column 0.

Column 0 generates a carry. Column 1 keeps it going because of $b_1$.

Column 0 generates a carry. Column 1 keeps it going because of $a_1$.

Column 0 generates a carry. Column 1 keeps it going because of $a_1$.

Column 0 generates a carry. Column 1 keeps it going because of $b_1$.

Column 0 generates a carry. Column 1 keeps it going because of $b_1$. 
An Inspiration

• What if?
  – Build sum-of-products for each of the 32 sum values

**Group Exercise:**

How many input bits to a 32-bit adder?

How large is each AND gate? What is the path length for each?
An Inspiration

- What if?
  - Build sum-of-products for each of the 32 sum values

Group Exercise:

65 bits = 2 x 32 + carryIn[0]

Each AND gate has 65 inputs, path length 6 or 7
An Inspiration

• What if?
  - Build sum-of-products for each of the 32 sum values
• Why doesn't it work?

Group Exercise:

How many rows are there in the truth table?
How many true values = how many AND gates?
An Inspiration

• What if?
  - Build sum-of-products for each of the 32 values
• Why doesn't it work?

Group Exercise:

$2^{65}$ rows in the truth table

$2^{64}$ AND gates, if half the rows are true
A Compromise

• Carry Lookahead Adders
  - Build summaries of how bits respond to carry bits
  - Use summaries to calculate carry bits rapidly
  - Use carry bits to generate answers rapidly

• 3 basic stages (each is fast)
  - Calculate “propagate” and “generate”
  - Calculate carry
  - Calculate result
Calculating Carry Bits

Group Exercise:

Build a sum-of-products expression for $c_{i+1}$ in terms of $a_i, b_i, c_i$

Bonus:
If you can, simplify it down to three terms instead of 4.
Calculating Carry Bits

\[ c_{i+1} = a_i b_i + a_i c_i + b_i c_i \]

Intuition:

We have a **carryOut** if any 2 of the three inputs \((a, b, \text{carryIn})\) are true.
Calculating Carry Bits

\[ c_{i+1} = a_i b_i + a_i c_i + b_i c_i \]

\[ c_{i+1} = a_i b_i + (a_i + b_i) c_i \]

Generate \quad Propagate

This simplification will make algebra easier in the next few slides.

Plus, it show us how to calculate the **generate** and **propagate** bits!
Calculating Carry Bits

Group Exercise:

Use substitution to build a sum-of-products expression for $c_1, c_2$, in terms of $a_0, b_0, a_1, b_1, c_0$.

Remember:

$$c_{i+1} = a_i b_i + (a_i + b_i) c_i$$
Calculating Carry Bits

\[ c_1 = a_0 b_0 + (a_0 + b_0) c_0 \]
\[ c_2 = a_1 b_1 + (a_1 + b_1) c_1 \]
\[ c_2 = a_1 b_1 + (a_1 + b_1) (a_0 b_0 + (a_0 + b_0) c_0) \]
\[ c_3 = a_2 b_2 + (a_2 + b_2) c_2 \]
\[ c_3 = a_2 b_2 + (a_2 + b_2) (\ldots) \]
Generate and Propagate

Let's think about this again...

- When would we **generate** a carryOut (no matter what the carryIn value was)?
- When would we **propagate** a carry bit forward, if one were given to us?

\[ c_{i+1} = a_i b_i + (a_i + b_i) c_i \]

*Generate  Propagate*
Generate and Propagate

- $g_i$ \textbf{generate} bit for column $i$
- $p_i$ \textbf{propagate} bit for column $i$

\[
g_i = a_i b_i \\
p_i = a_i + b_i \\
c_{i+1} = g_i + p_i c_i
\]

\textbf{Intuition:}
We have a \texttt{carryOut} if we generated it locally, or if we propagated a \texttt{carryIn}. 
Calculating Carry Bits

Group Exercise:

Use substitution to build a sum-of-products expression for \( c_1, c_2, c_3, c_4 \), in terms of \( g_0, p_0, g_1, p_1, g_2, p_2, g_3, p_3, c_0 \).

Remember:

\[
    c_{i+1} = g_i + p_i c_i
\]
\[ c_1 = g_0 + p_0 c_0 \]

\[ c_2 = g_1 + p_1 c_1 \]

\[ c_2 = g_1 + p_1 (g_0 + p_0 c_0) \]

\[ c_2 = g_1 + p_1 g_0 + p_1 p_0 c_0 \]

\[ c_3 = g_2 + p_2 c_2 \]

\[ c_3 = g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_0 \]
\[ c_1 = g_0 + p_0 c_0 \]

This is a sum-of-products expression of \( g_i, p_i \). It is \textbf{not} a sum-of-products of \( a_i, b_i \). To get that, we would need to do substitution – and this would get exponentially large.

Our solution is simple: \textbf{don't do the substitution.}

\[ c_3 = g_2 + p_2 c_2 \]

\[ c_3 = g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_0 \]
How It Fits Together

- We can calculate the generate and propagate bits *in parallel!*
  - No need for ripples

- Each bit of the adder can **generate its own carryIn**, in parallel with all other bits

- Each carryIn is a sum-of-products, with no more than 32 AND gates
  - But each bit needs all of the previous generate and propagate bits
How It Fits Together

Calculate Generate / Propagate

Adder

out

a

b

g

p
How It Fits Together

All of the g / p bits are calculated in parallel.

And we already have AND / OR in the ALU, so they don't require any new hardware!
How It Fits Together

Ripple is eliminated because each Adder element can calculate its own carryIn from the $g / p$ bits.
We Can Do Even Better

- The *carryIn* calculation can get very large!
  - High bits need to consider generate / propagate at all lower bits

- But it's cheap for the first few bits

- Can we use what we've learned to improve this more?
We Can Do Even Better

- What if we organized our adders into nibbles?
We Can Do Even Better

- Each nibble is a 4-bit adder
- Use Carry-Lookahead inside the nibble
  - Size is reasonable because only 4 bits
- Chain nibbles together
  - We have a Ripple Adder again (though slightly better)
  - Can we use Carry-Lookahead again to help?
Super-Propagate, Super-Generate

- $G_i$ generate for nibble $i$
- $P_i$ propagate for nibble $i$

NOTE:
Capital letters are used for the super values. Lowercase for the original values.
Super-Propagate, Super-Generate

Group Exercise:

Figure out the formula for $c_4$ in terms of $g_0, p_0, g_1, p_1, g_2, p_2, g_3, p_3, c_0$.

Come up with formulae for $G_0, P_0$ (the super-generate and super-propagate bits for nibble 0) in terms of $g_0, p_0, g_1, p_1, g_2, p_2, g_3, p_3$. 
Super-Propagate, Super-Generate

\[ P_0 = p_3 p_2 p_1 p_0 \]
\[ G_0 = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0 \]

**Intuition:**

In order to propagate, we must propagate at every bit.

In order to generate, we generate somewhere and then carry it through, all the way to carryOut.
Recursion

Calc

g / p

Calc

G / P

Calc

C

Calc

c

Add
Adding up the Delay

- **Total Delay:**
  - AND / OR to generate $g / p$
  - Sum-of-products to generate $G / P$
  - Sum-of-products to generate nibble $\text{carryIn}$
  - Sum-of-products to generate bit $\text{carryIn}$
  - Cost of adding 1 bit

- Since each sum-of-products is small, total delay is quite small!
Multiplication

Group Exercise:
Add up the values to find the final answer.

Question:
Is this problem in decimal or binary?

Then:
In general, how did we find each row to add up?
A Different Sort of Sum-of-Products

Group Exercise:

Write a sequence of instructions which calculates

\[ s_1 = s_2 \times 13 \]

using only add and shift instructions. You may use \( t_0 \) as a temporary.
A Different Sort of Sum-of-Products

```
sll $s1, $s2, 3    # s1 = s2 * 8
sll $t0, $s2, 2    # t0 = s2 * 4
add $s1, $s1, $t0  # s1 = s2 * 12
add $s1, $s2, $s2  # s1 = s2 * 13
```

Group Exercise:

Write a sequence of instructions which calculates

\[ s1 = s2 \times 13 \]

using only add and shift instructions. You may use \( t0 \) as a temporary.
A Multiplication Algorithm

```c
int64 multiply(int multiplicand_32,
              int multiplier)
{
    int64 retval = 0;
    int64 multiplicand = multiplicand_32;

    for (int i=0; i<32; i++)
    {
        int bit = (multiplier >> i) & 0x1;
        if (bit)
            retval += (multiplicand << i);
    }
    return retval;
}
```
Group Exercise:

Write a sequence of instructions which calculates
\[ s_1 = s_2 \times 13 \]
using only add and shift instructions. **Do not use any temporary registers!**

**Hint:**
\[ 13 = 1 + 4\times3 \]
A Different Sort of Sum-of-Products

\[
\text{sl}l \quad \text{\$s1, \$s2, 1} \quad \# \quad \text{s1} = \text{s2} \times 2 \\
\text{add} \quad \text{\$s1, \$s1, \$s2} \quad \# \quad \text{s1} = \text{s2} \times 3 \\
\text{sl}l \quad \text{\$s1, \$s1, 2} \quad \# \quad \text{s1} = \text{s2} \times 12 \\
\text{add} \quad \text{\$s1, \$s1, \$s2} \quad \# \quad \text{s1} = \text{s2} \times 13
\]

**Group Exercise:**

Write a sequence of instructions which calculates $s_1 = s_2 \times 13$ using only add and shift instructions. Do not use any temporary registers.
In General...

- Multiplication can be decomposed into add/shift-1 operations:
  - If lowest bit of multiplier is 1 (odd), then add multiplicand to output
  - Shift multiplicand left by 1, and multiplier right by 1
  - Repeat
An Iterative Algorithm

<table>
<thead>
<tr>
<th>Multiplicand</th>
<th>Multiplier</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>13</td>
<td>7</td>
</tr>
<tr>
<td>14</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>28</td>
<td>3</td>
<td>35</td>
</tr>
<tr>
<td>56</td>
<td>1</td>
<td>91</td>
</tr>
<tr>
<td>112</td>
<td>0</td>
<td>91</td>
</tr>
</tbody>
</table>
A Multiplication Algorithm

```c
int64 multiply(int multiplicand_32, int multiplier)
{
    int64 retval = 0;
    int64 multiplicand = multiplicand_32;

    for (int i=0; i<32; i++)
    {
        int bit = multiplier & 0x1;
        if (bit)
            retval += multiplicand;
        multiplier >>= 1;
        multiplicand <<= 1;
    }

    return retval;
}
```
Seeing It In Hardware
Version 2

- 32 bit multiplicand register
- 32 bit adder instead of 64 bit
Version 3

- Store multiplier in output register
- Some bits not used yet!
mul / div (Review)

• **mul** / **div** only have **two** operands
  – Both ordinary registers

• **Answers always go to hi and lo registers**
  – Special registers (**NOT** part of the normal 32)
  – **Use** **mfhi** / **mflo** to read them
Summary

- Multiply performed by shifts and adds

- A lot harder than add
  - 1 clock cycle per bit
  - Can be accelerated with more hardware – but it's expensive!

- 64-bit result