MULTIPLE OPERAND ADDITION

Chapter 3

Multioperand Addition

- Add up a bunch of numbers
  \[ s = \sum_{i=1}^{m} x(i) \]

- Used in several algorithms
  - Multiplication, recurrences, transforms, and filters

- Signed (two’s comp) and unsigned
  - Don’t deal with overflow…
Bit Arrays

- Simplify things by assuming that all arguments have the same range of values
  - Bit arrays are rectangular
  - Means you may end up sign-extended operands
  - If you are adding \( m \) operands where each operand is an \( n \)-bit array: the sum has \( n+p \) bits

\[
p = \lceil \log_2 m \rceil
\]

- Extend operands (0-fill or sign-extend) to \( n+p \) bits

Sign Extension

\[
\begin{align*}
  a_0 & a_0 a_0 a_0 a_1 a_2 \ldots a_n \\
  b_0 & b_0 b_0 b_0 b_1 b_2 \ldots b_n \\
  c_0 & c_0 c_0 c_0 c_1 c_2 \ldots c_n \\
  d_0 & d_0 d_0 d_0 d_1 d_2 \ldots d_n \\
  e_0 & e_0 e_0 e_0 e_1 e_2 \ldots e_n \\
\end{align*}
\]

sign extension

\[ m = 5 \]

\[ \lceil \log_2 5 \rceil = 3 \]

Figure 3.1: SIGN-EXTENDED ARRAY FOR \( m = 5 \).
Sign Extension Trick

- Sign extension requires that all the adder bits for the sign extended bits be implemented
  - A trick for collecting all the sign extensions into one extra term is:
  - Recall:
    \[ x = -x_0 + \sum_{i=1}^{n} x_i 2^{-i} \]
    
  - In other words: represent \( x \) as a fraction, and because of two's comp, \( x_0 \) has negative weight

- Apply identity
  \[ (-x_0) + 1 - 1 = (1 - x_0) - 1 = x_0' - 1 \]

- So this transforms signed operand
  \[ x_0 \cdot x_1 \cdot x_2 \cdot x_3 \cdots x_n \]

  to be replaced by
  \[ x_0' \cdot x_1 \cdot x_2 \cdot x_3 \cdots x_n - 1 \]

- Now you can invert the \( x_0 \)'s and add up number of times you did that
Sign Extension Trick

Example for $m=5$

-5=1011

What happens if we add this to $x_0'$?

\[
\begin{array}{cccc}
1 & 0 & 1 & 1 \\
\hline
? & ? & ? & ?
\end{array}
\]

- $1+x_0' = 1+(1 - x_0) = 2 - x_0$
- $10-0=10$, $10-1=01$, $1+x_0' = x_0'$
- $0+x_0' = x_0'$
Example for m=5

-5=1011

What happens if we add this to $x_0'$?

\[ \begin{array}{cccc}
1 & 0 & 1 & 1 \\
\end{array} \quad \begin{array}{cccc}
1 & 0 & 1 & 1 \\
1 & x_0' & x_0 & x_0 \\
\end{array} \]

- $1 + x_0' = 1 + (1 - x_0) = 2 - x_0$
- $10 - 0 = 10$, $10 - 1 = 01$,
- $1 + x_0' = x_0' \cdot x_0$
- $0 + x_0' = x_0'$

Reduction

- Primitive operation in multioperand addition
  - Produces a smaller output bit-array by adding the inputs bits
  - Main approaches are:
    - Reduction by rows (using adders or compressors)
    - Reduction by columns (using counters)
[p:2] Adders

- Reduce p bit-vectors to 2 bit-vectors
- This means carry-save form at the output
  - [3:2] adder is a regular full adder
  - [4:2] adder adds two carry-save inputs
  - [5:2], [7:2] take even more rows as inputs and reduces them to two output rows

---

Example with [3:2] adders

- \( p = 3 \)
- \( x, y, z \)
- \( v_s, v_c, C_{in} \)

<table>
<thead>
<tr>
<th>( VS )</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>( c_{out} ), ( VC )</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- digit value | 0 | 1 | 2 | 2 | 1 | 0 | 2 | 0 | 2
Example with [4:2] adders

Note that even though it looks like carry is propagated, the Cout from each [4:2] cell is computed directly from the A and B inputs…

[4:2] Compressor Adder

General Case
k can be Greater than 1
In General

- Consider p vectors
- Break them up into k-bit chunks
- End up with k-wide [p:2] adder
- Imagine k=1, p=3
  - Then you can use regular [3:2] adders…
  - Note h\text{out} doesn’t wait for h\text{in}

General [p:2] adder

- Complexity depends of columns k
  - Minimize columns, but attention to max pos values
  - Also minimize carries H

\min H \text{ when } H = p - 2
\max(2^k - 1, 2(2^k) - p) \leq W \leq 2(2^k) - p
2^k \geq p - 1
Typical [p:2] modules

<table>
<thead>
<tr>
<th></th>
<th>H</th>
<th>k</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>9</td>
<td>7</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>11</td>
<td>9</td>
<td>4</td>
<td>21</td>
</tr>
</tbody>
</table>

[4:2] Adder

- **P=4**
  - 4 rows of bit operands
  - (or 2 carry save ops)
- **K=2**
  - Clumps of 2 bits
- **H=2**
  - 2 “carry” bits
- **W=4**
  - 3 bits of intermediate sum
  - W is computed from inputs only (no carries)
[5:2] Adder

- $P=5$
  - 5 rows of operands
- $K=2$
  - Clumps of 2 bits
- $H=3$
  - 3 “carry” bits
- $W=3$
  - 2 bits of intermediate sum

[7:2] Adder

- $P=7$
  - 7 rows of operands
- $K=3$
  - Clumps of 3 bits
- $H=5$
  - 5 “carry” bits
- $W=9$
  - 4 bits of intermediate sum
Rows vs. Columns

- Reduction by rows adds up \( p \) rows and produces a vector of 2 rows (carry save)
  - Different adders may take a different sized clump
  - Deals with carries from previous stage, and produces carries to next stage
    - No propagation further than one stage though!
- Reduction by columns adds a whole column
  - Produces a single-row output
  - As many bits as necessary for that size column

(p:q] Counters

- Add up a column of \( p \) bits
  - Result is \( q \) bits that represent the sum of those column bits
  - If \( p \) inputs, then max output value is \( p \) (all ones)
  - For example, ten rows (\( p=10 \))
    - You must be able to represent “ten” in \( q \) output bits
    - \( 2^q - 1 \geq p \)
    - \( q = \lceil \log_2(p + 1) \rceil \)
(p:q] Counters

Add column of \( p \) bits of the same weight
Produce \( q \) bits of adjacent weights

\[
\sum_{i=0}^{p-1} x_i = \sum_{j=0}^{q-1} y_j 2^j
\]

(3:2], (7:3], and (15:4] are examples

\[
2^q - 1 \geq p, \text{ i.e., } q = \lfloor \log_2(p + 1) \rfloor
\]

\[
\begin{array}{c}
\vdots \\
x_0 \\
x_1 \\
\vdots \\
x_{p-1} \\
+ \\
y_{q-1} \ldots y_0 \\
\end{array}
\]

\( p \) inputs (same weight)

\( q \) outputs

(7:3] Counter

Figure 3.8: Implementation of (7:3) counter by an array of full adders.
Gate Network for (7:3]

- Input is seven binary bit-vectors
  \[ X = (x_6, x_5, x_4, x_3, x_2, x_1, x_0) \]
- Output is three bits
  \[ q = \sum_{i=0}^{6} x_i = 4q_2 + 2q_1 + q_0 \]
- Partition the input array into two subvectors
  \[ X_A = (x_2, x_1, x_0) \quad X_B = (x_6, x_5, x_4, x_3) \]

Gate Network for (7:3]

- Partial sums of subvectors are
  \[ q_A = 2q_{A1} + q_{A0} \]
  \[ q_B = 4q_{B2} + 2q_{B1} + q_{B0} \]
- Sum \( q_A \) is like a full adder (three inputs)
  \[ q_{A0} = x_2 \oplus x_1 \oplus x_0 \]
  \[ q_{A1} = x_2 x_1 + x_2 x_0 + x_1 x_0 \]
### Gate Network for (7:3)

<table>
<thead>
<tr>
<th>$x_6$</th>
<th>$x_5$</th>
<th>$x_4$</th>
<th>$x_3$</th>
<th>$q_{B2}$</th>
<th>$q_{B1}$</th>
<th>$q_{B0}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

$q_{B0} = x_6 \oplus x_5 \oplus x_4 \oplus x_3$

$q_{B1} = \text{any two bits} \cdot (x_6x_5x_4x_3)'$

$a = [x_6x_5 + x_4x_3 + (x_6 + x_5)(x_4 + x_3)]$

$q_{B2} = (x_6x_5x_4x_3)'$

\[ q_{A0} = \overbrace{(q_{B1} \oplus q_{B0}) + (q_{B1} \oplus q_{A1})(q_{B0}q_{A0})}^{	ext{sum}} \]

\[ q_{A1} = q_{B2} + q_{B1}q_{A0} + (q_{B1} \oplus q_{A1})(q_{B0}q_{A0}) \]

Finally, $q = q_A + q_B$

\[ q_0 = q_{A0} + q_{B0} \]

\[ q_1 = (q_{A1} \oplus q_{B1}) \oplus q_{A0}q_{B0} \]

\[ q_2 = q_{B2} + q_{B1}q_{A1} + (q_{B1} \oplus q_{A1})(q_{B0}q_{A0}) \]

\[ G_1 \leadsto P_1 \cdot G_0 \]

$q_{B1} = a \cdot q_{B2} \Rightarrow a \cdot q_{A1}$
Gate Network for (7:3)

\[ q_0 = q_{A0} + q_{B0} \]

\[ q_1 = (q_{A1} \oplus q_{B1}) \oplus q_{A0}q_{B0} \]

\[ q_2 = q_{B2} + aq_{A1} + (q_{B1} \oplus q_{A1})(q_{B0}q_{A0}) \]

Multicolumn Counters...

\[ \left( p_{k-1}, p_{k-2}, \ldots, p_0 : q \right) \]

\[ v = \sum_{i=0}^{k-1} \sum_{j=1}^{p_i} a_{ij}2^i \leq 2^q - 1 \]

\[ v \leq 1 \times 4 + 2 \times 2 + 3 \times 1 = 11 < 2^4 - 1 \]

\[ v \leq 5 \times 2 + 5 \times 1 = 15 = 2^4 - 1 \]

Figure 3.10: (a) \( (5,5,4) \) counter. (b) \( (1,2,3,4) \) counter.
Sequential Implementation

- If your input array is big
  - You can add rows sequentially
  - Uses only one adder and a register

![Sequential Implementation Diagram]

### Example

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

\[
x \quad 0 \quad 0 \quad 1 \quad 0 \\
\]

\[
s' \quad 0 \quad 0 \quad 0 \quad 0 \\
\]

\[
c' \quad 0 \quad 0 \quad 0 \quad 0 \\
\]

\[
x \quad 0 \quad 1 \quad 0 \quad 1 \\
\]

\[
s' \quad 0 \quad 0 \quad 0 \quad 0 \\
\]

\[
c' \quad 0 \quad 0 \quad 0 \quad 0 \\
\]

\[
x \quad 0 \quad 1 \quad 0 \quad 1 \\
\]

\[
s' \quad 0 \quad 0 \quad 0 \quad 0 \\
\]

\[
c' \quad 1 \quad 0 \quad 1 \quad 1 \\
\]

\[
s \quad 1 \quad 0 \quad 0 \quad 1 \\
\]

\[
c \quad 0 \quad 0 \quad 1 \quad 0 \\
\]
Sequential Implementation

- **m operands** → **m/(p-2) iterations**
  - 4 operands = 4/(3-2) = 4 iterations with [3:2] adders
    - Each iteration adds 1 row plus s and c
  - Using [4:2] adders = 4/(4-2) = 2 iterations
    - In each iteration you add 2 rows plus s and c
Combinational Implementation

- **Reduction by rows**
  - Linear array of \([p:2]\) adders
  - Tree of \([p:2]\) adders

- **Reduction by columns**
  - Using \((p:q)\) counters

### Linear Array

\[
\left\lfloor \frac{m - 2}{p - 2} \right\rfloor \text{ adders in the array}
\]

\[n + p \text{ bits in final sum}
\]

where \(p = \left\lceil \log_2(m) \right\rceil\)

Last adder may have additional bits for the sign extension trick we saw at the beginning of the chapter.
Adder Trees

- Because addition is associative, you can organize as a tree
  - Number of adders is the same (same number of inputs...)

\[ k = \frac{m - 2}{p - 2} \]  \([p:2]\) carry-save adders

\[ pk = m + 2(k - 1) \]

Adder Trees

- The number of adder levels

\[ m_l = p \left\lfloor \frac{m_{l-1}}{2} \right\rfloor + m_{l-1} \mod 2 \]

Reduction sequence

\[
\begin{array}{cccccccccc}
1 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
m_l & 3 & 4 & 6 & 9 & 13 & 19 & 28 & 42 & 63
\end{array}
\]
Adder tree for 9 operands

Three-level tree (6 operands)
Four-level tree (7 vs 9 operands)

5-level Tree (13 inputs)
Tree of [4:2] adders for $m=16$

Reduction by Columns

- Use multiple levels of $(p:q)$ counters to reduce the columns to rows
  - This example uses 4 (7:3) counters
  - Each counter counts the 1's in one column
  - Then add the rows

Figure 3.15: Tree of [4:2] adders for $m=16$. 
Reduction by Columns

Why draw it that way?

\[
\begin{array}{cccc}
1 & 0 & 1 & 1 \\
0 & 0 & 1 & 0 \\
1 & 0 & 0 & 1 \\
0 & 1 & 1 & 0 \\
1 & 0 & 1 & 0 \\
1 & 1 & 1 & 1 \\
0 & 1 & 1 & 0 \\
\hline
0 & 1 & 0 & 1 \\
0 & 1 & 1 & 1 \\
1 & 0 & 1 & 0
\end{array}
\quad
\begin{array}{cccc}
1 & 0 & 1 & 1 \\
0 & 0 & 1 & 0 \\
1 & 0 & 0 & 1 \\
0 & 1 & 1 & 0 \\
1 & 0 & 1 & 0 \\
1 & 1 & 1 & 1 \\
0 & 1 & 1 & 0 \\
\hline
0 & 1 & 1 \\
0 & 1 & 1 \\
1 & 0 & 1 & 0 \\
1 & 1 & 1 & 1 \\
0 & 1 & 1 & 0 \\
1 & 1 & 0 & 1 & 1 & 59
\end{array}
\]

Reduction by Columns

Why draw it that way?

\[
\begin{array}{cccc}
1 & 0 & 1 & 1 \\
0 & 0 & 1 & 0 \\
1 & 0 & 0 & 1 \\
0 & 1 & 1 & 0 \\
1 & 0 & 1 & 0 \\
1 & 1 & 1 & 1 \\
0 & 1 & 1 & 0 \\
\hline
0 & 1 & 0 & 1 \\
0 & 1 & 1 & 1 \\
1 & 0 & 1 & 0
\end{array}
\quad
\begin{array}{cccc}
1 & 0 & 1 & 1 \\
0 & 0 & 1 & 0 \\
1 & 0 & 0 & 1 \\
0 & 1 & 1 & 0 \\
1 & 0 & 1 & 0 \\
1 & 1 & 1 & 1 \\
0 & 1 & 1 & 0 \\
\hline
0 & 1 & 1 \\
0 & 1 & 1 \\
1 & 0 & 1 & 0 \\
1 & 1 & 1 & 1 \\
0 & 1 & 1 & 0 \\
1 & 1 & 0 & 1 & 1 & 59
\end{array}
\]
Number of Counter Levels

\[ m_1 = p \]
\[ m_l = p \left( \left\lfloor \frac{m_{l-1}}{q} \right\rfloor + m_{l-1} \mod q \right) \]

\[ l \approx \log_{p/q}(m_l/q) \]

Sequences of Counters

How many rows can be reduced with \(L\) levels of counters?

<table>
<thead>
<tr>
<th>(l)</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>(m_l)</td>
<td>3</td>
<td>4</td>
<td>6</td>
<td>9</td>
<td>13</td>
<td>19</td>
<td>28</td>
<td>42</td>
<td>63</td>
</tr>
</tbody>
</table>

(3:2] counters

<table>
<thead>
<tr>
<th>Number of levels</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max. number of rows</td>
<td>7</td>
<td>15</td>
<td>35</td>
<td>79</td>
<td>...</td>
</tr>
</tbody>
</table>

(7:3] counters

Figure 3.17: Construction of [pq] reduction tree.
Example with (7:3) and 15 rows

Table says we can do it in 2 levels of counters

(example...)

Systemic Design Method

<table>
<thead>
<tr>
<th>Full adder (3-2)</th>
<th>Half adder (2-2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^{i+1} \cdot 2^l$</td>
<td>$2^{i+1} \cdot 2^l$</td>
</tr>
</tbody>
</table>

\[ \begin{array}{|c|c|c|c|c|c|} \hline \text{level} & 1 & 2 & 3 & 4 & \ldots \\ \hline \text{max. level} & 15 & 15 & 35 & 79 & \ldots \\ \hline \end{array} \]

- \textit{denotes 0 or 1}
- \textit{diagonal outputs when representing separately sum and carry bit-vectors is preferrable}
- \textit{horizontal outputs when interleaving sum and carry bits is acceptable}

\[ \begin{array}{|c|c|c|c|c|c|c|c|c|} \hline \text{level} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ \hline \text{index} & 3 & 4 & 6 & 9 & 13 & 19 & 28 & 42 & 63 \\ \hline \end{array} \]
Reduction Process

Use a mix of FA and HA cells. Transfer the other bits.

Why not 4 FA?

Table 3.1: [3,2] Reduction sequence.

<table>
<thead>
<tr>
<th>i</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>(m_i)</td>
<td>3</td>
<td>4</td>
<td>6</td>
<td>9</td>
<td>13</td>
<td>19</td>
<td>28</td>
<td>42</td>
<td>63</td>
</tr>
</tbody>
</table>

Relation at Level 1

\(e_i\) – number of bits in column \(i\)

\(f_i\) – number of full adders in column \(i\)

\(h_i\) – number of half adders in column \(i\)

\(e_i - 2f_i - h_i + f_{i-1} + h_{i-1} = m_{i-1}\)

resulting in

\(2f_i + h_i = e_i - m_{i-1} + f_{i-1} + h_{i-1} = p_i\)

Solution producing min number of carries:

\(f_i = \lfloor p_i / 2 \rfloor\)

\(h_i = p_i \mod 2\)
Relation at Level 1

- $e_i$ - number of bits in column $i$
- $f_i$ - number of full adders in column $i$
- $h_i$ - number of half adders in column $i$

$$e_i - 2f_i - h_i + f_{i-1} + h_{i-1} = m_{i-1}$$

resulting in

$$2f_i + h_i = e_i - m_{i-1} + f_{i-1} + h_{i-1} = p_i$$

Solution producing min number of carries:

$$f_i = \lfloor p_i/2 \rfloor \quad h_i = p_i \mod 2$$

---

Relation at Level 1

- $e_i$ - number of bits in column $i$
- $f_i$ - number of full adders in column $i$
- $h_i$ - number of half adders in column $i$
- $m_i$ - number of full adders in column $i$

<table>
<thead>
<tr>
<th>$i$</th>
<th>6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>$l = 4$</td>
<td>8 8 8 8 8</td>
</tr>
<tr>
<td>$e_i$</td>
<td>8 8 8 8 8</td>
</tr>
<tr>
<td>$m_3$</td>
<td>6 6 6 6 6</td>
</tr>
<tr>
<td>$h_i$</td>
<td>0 0 0 1 0</td>
</tr>
<tr>
<td>$f_i$</td>
<td>2 2 2 1 1</td>
</tr>
<tr>
<td>$l = 3$</td>
<td>2 6 6 6 6</td>
</tr>
<tr>
<td>$e_i$</td>
<td>2 6 6 6 6</td>
</tr>
<tr>
<td>$m_2$</td>
<td>4 4 4 4 4</td>
</tr>
<tr>
<td>$h_i$</td>
<td>0 0 0 0 1</td>
</tr>
<tr>
<td>$f_i$</td>
<td>0 2 2 2 1</td>
</tr>
<tr>
<td>$l = 2$</td>
<td>4 4 4 4 4</td>
</tr>
<tr>
<td>$e_i$</td>
<td>4 4 4 4 4</td>
</tr>
<tr>
<td>$m_1$</td>
<td>3 3 3 3 3</td>
</tr>
<tr>
<td>$h_i$</td>
<td>0 0 0 0 1</td>
</tr>
<tr>
<td>$f_i$</td>
<td>1 1 1 1 0</td>
</tr>
<tr>
<td>$l = 1$</td>
<td>3 3 3 3 3</td>
</tr>
<tr>
<td>$e_i$</td>
<td>1 3 3 3 3</td>
</tr>
<tr>
<td>$m_0$</td>
<td>2 2 2 2 2</td>
</tr>
<tr>
<td>$h_i$</td>
<td>0 0 0 0 1</td>
</tr>
<tr>
<td>$f_i$</td>
<td>0 1 1 1 1</td>
</tr>
</tbody>
</table>
Example

- **Build array to compute** \( f = a + 3b + 3c + d \)
  - Operands are integers in the range \(-4\) to \(3\)
  - Two’s comp
  - Compute range of result...

Operands in \([-4, 3]\). Result range:

\[-4 + (-12) + (-12) - 4 = -32 \leq f \leq 3 + 9 + 9 + 3 = 24\]

- Thus \( f \) requires 6 bits
- Decompose \(3b\) and \(3c\) into \(2b+b\) and \(2c+c\)
### Example: \( f = a + 3b + 3c + d \)

<table>
<thead>
<tr>
<th>( a )</th>
<th>( a_2 )</th>
<th>( a_2 )</th>
<th>( a_2 )</th>
<th>( a_1 )</th>
<th>( a_0 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( b )</td>
<td>( b_2 )</td>
<td>( b_2 )</td>
<td>( b_2 )</td>
<td>( b_1 )</td>
<td>( b_0 )</td>
</tr>
<tr>
<td>( 2b )</td>
<td>( b_2 )</td>
<td>( b_2 )</td>
<td>( b_2 )</td>
<td>( b_1 )</td>
<td>( b_0 )</td>
</tr>
<tr>
<td>( c )</td>
<td>( c_2 )</td>
<td>( c_2 )</td>
<td>( c_2 )</td>
<td>( c_1 )</td>
<td>( c_0 )</td>
</tr>
<tr>
<td>( 2c )</td>
<td>( c_2 )</td>
<td>( c_2 )</td>
<td>( c_2 )</td>
<td>( c_1 )</td>
<td>( c_0 )</td>
</tr>
<tr>
<td>( d )</td>
<td>( d_2 )</td>
<td>( d_2 )</td>
<td>( d_2 )</td>
<td>( d_1 )</td>
<td>( d_0 )</td>
</tr>
</tbody>
</table>

Note left shifts to compute \( 2b \) and \( 2c \)...

### Example: \( f = a + 3b + 3c + d \)

| \( a \) | \( a_2 \) | \( a_2 \) | \( a_2 \) | \( a_1 \) | \( a_0 \) |
| \( b \) | \( b_2 \) | \( b_2 \) | \( b_2 \) | \( b_1 \) | \( b_0 \) |
| \( 2b \) | \( b_2 \) | \( b_2 \) | \( b_2 \) | \( b_1 \) | \( b_0 \) |
| \( c \) | \( c_2 \) | \( c_2 \) | \( c_2 \) | \( c_1 \) | \( c_0 \) |
| \( 2c \) | \( c_2 \) | \( c_2 \) | \( c_2 \) | \( c_1 \) | \( c_0 \) |
| \( d \) | \( d_2 \) | \( d_2 \) | \( d_2 \) | \( d_1 \) | \( d_0 \) |

Use sign extension trick
Keep track of weight of sign extension bits!

\[-4 \times 4 + (-8) \times 2 = -32\]
\[-32 = 100000\]
Example: \( f = a + 3b + 3c + d \)

Remember the 100000 (-32)

<table>
<thead>
<tr>
<th>( a )</th>
<th>( a' _2 _1 _0 )</th>
<th>( b )</th>
<th>( b' _2 _1 _0 )</th>
<th>( c )</th>
<th>( c' _2 _1 _0 )</th>
<th>( 2c )</th>
<th>( c' _2 _1 _0 )</th>
<th>( d )</th>
<th>( d' _2 _1 _0 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a )</td>
<td>( a' _2 _1 _0 )</td>
<td>( b )</td>
<td>( b' _2 _1 _0 )</td>
<td>( c )</td>
<td>( c' _2 _1 _0 )</td>
<td>( 2c )</td>
<td>( c' _2 _1 _0 )</td>
<td>( d )</td>
<td>( d' _2 _1 _0 )</td>
</tr>
<tr>
<td>(-1)</td>
<td>(-1)</td>
<td>(-1)</td>
<td>(-1)</td>
<td>(-1)</td>
<td>(-1)</td>
<td>(-1)</td>
<td>(-1)</td>
<td>(-1)</td>
<td>(-1)</td>
</tr>
</tbody>
</table>

Example: \( f = a + 3b + 3c + d \)

\[
\begin{array}{cccc|cccc|cccc}
1 & 0 & b'_2 & a'_2 & a_1 & a_0 \\
& & c'_2 & b'_2 & b_1 & b_0 \\
& & & b_1 & b_0 \\
& & c'_2 & c_1 & c_0 \\
& & & c_1 & c_0 \\
& & d'_2 & d_1 & d_0 \\
\end{array}
\]

Determine adder types at each level
* Helps by reducing the width of the CPA
Example: \( f = a + 3b + 3c + d \)

You can always add pipelining to increase throughput!
Partially Combinational Arrays

Non-pipelined

\[ k \text{ operands are added per iteration (} k = 4 \text{ in these examples)} \]
\[ m/k \text{ iterations required for complete result} \]

Note that if the partial sum is added to the top of the array, you can’t pipeline.

Here’s a general technique for summing \( q \) operands per iteration.
Conclusion

- Lots of ways of adding up a bunch of numbers
  - Arguments are a bit-array
  - Reduce that bit array by reducing rows or columns
  - End result is typically in carry-save form
    - So you might need a CPA if you want the answer in conventional form
  - Pipelining is always an option