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

$$\begin{array}{l} a_0 a_0 a_0 a_0 . a_1 a_2 \; ... \; a_n \\ b_0 b_0 b_0 b_0 . b_1 b_2 \; ... \; b_n \\ c_0 c_0 c_0 c_0 . c_1 c_2 \; ... \; c_n \\ d_0 d_0 d_0 d_0 . d_1 d_2 \; ... \; d_n \\ \underbrace{e_0 e_0 e_0}_{\text{sign extension}} e_0 . e_1 e_2 \; ... \; e_n \end{array}$$

$m = 5$

$\lceil \log_2 5 \rceil = 3$

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:

represent x as $x_0.x_1x_2 \cdots x_n$

$$x = -x_0 + \sum_{i=1}^{n} x_i 2^{-i}$$

  - In other words: represent ***x*** as a fraction, and because of two's comp, x_0 has negative weight

Sign Extension Trick

- Apply identity

$$(-x_0) + 1 - 1 \quad = \quad (1 - x_0) - 1 \quad = \quad x_0' - 1$$

- So this transforms signed operand

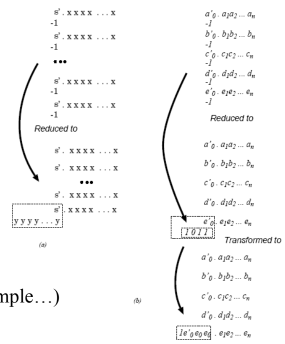
$$x_0 \,.\, x_1 \; x_2 \; x_3 \cdots x_n$$

to be replaced by

$$x_0' \,.\, x_1 \; x_2 \; x_3 \cdots x_n$$
$$-1$$

  - Now you can invert the x0's and add up number of times you did that

Sign Extension Trick



(Show Example…)

Example for m=5

- -5=1011
- What happens if we add this to x_0' ?

$$\begin{array}{cccc} 1 & 0 & 1 & 1 \\ & & & x_0' \\ \hline ? & ? & ? & ? \end{array}$$

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

Example for m=5

- -5=1011
- What happens if we add this to x_0' ?

$$\begin{array}{cccc} 1 & 0 & 1 & 1 \\ & & & x_0' \\ \hline ? & ? & ? & ? \end{array} \longrightarrow \begin{array}{cccc} 1 & 0 & 1 & 1 \\ & & & x_0' \\ \hline 1 & x_0' & x_0 & x_0 \end{array}$$

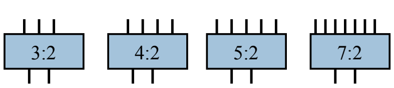
  - $1+x_0' = 1+(1 - x_0) = 2 - x_0$
    10-0=10, 10-1=01,
    $1+x_0'=x_0'\ x_0$
  - $0+x_0'=x_0'$

Reduction

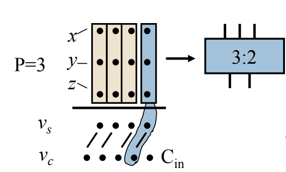
- Primitive operation in multioperand addition
  - Produces a smaller output bit-array by adding the inputs bits
  - Main approaches are:
    - Reduction by rows (using adders or compressors)
    - Reduction by columns (using counters)

[*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

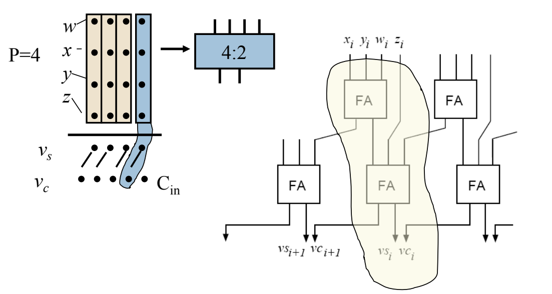


Example with [3:2] adders

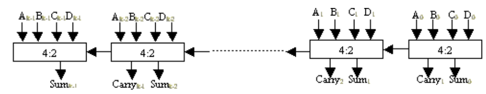


X		0	1	1	1	0	1	0	0
Y		0	0	1	1	1	0	1	1
Z		1	0	1	0	1	0	1	0
VS		1	1	1	0	0	1	0	1
(c_{out}, VC)	0	0	1	1	1	0	1	0	1
digit value	0	1	2	2	1	0	2	0	2

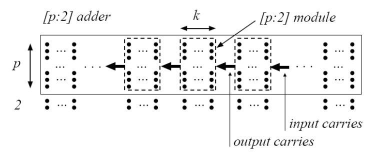
Example with [4:2] adders



[4:2] Compressor Adder



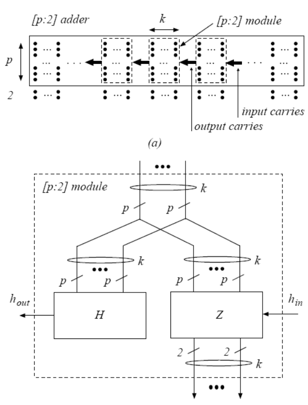
Note that even though it looks like carry is propagated, the Cout from each [4:2] cell is computed directly from the A and B inputs…



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_{out} doesn't wait for h_{in}



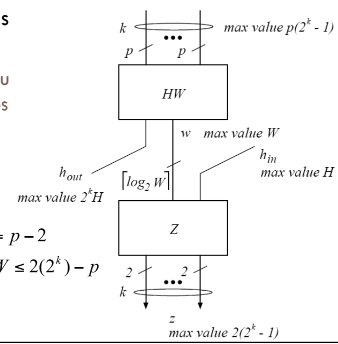
General [p:2] adder

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

min value of H when $H = p - 2$

$\max(2^k - 1, 2(2^k) - p) \le W \le 2(2^k) - p$

$2^k \ge p - 1$

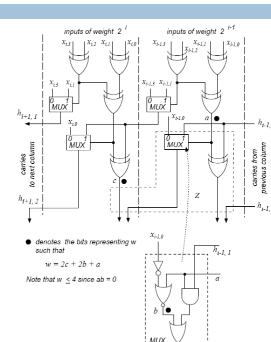


Typical [p:2] modules

p	H	k	W
3	1	1	1
4	2	2	4
5	3	2	3
6	4	3	10
7	5	3	9
9	7	3	7
11	9	4	21

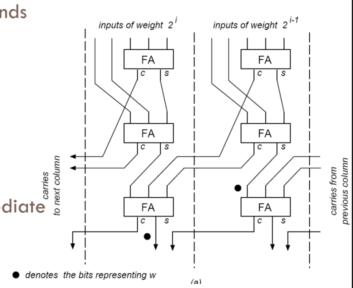
[4:2] Adder

- 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)



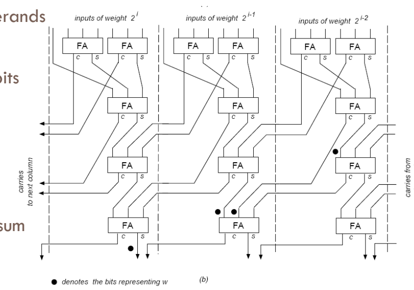
[5:2] Adder

- 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

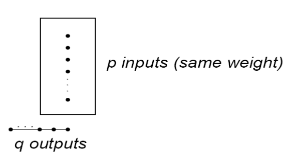
Add column of p bits of the same weight
Produce q bits of adjacent weights

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

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

$$2^q - 1 \geq p, \text{ i.e., } q = \lceil log_2(p+1) \rceil$$

$$\begin{array}{r} x_0 \\ x_1 \\ \vdots \\ + \quad x_{p-1} \\ \hline y_{q-1} \cdots y_0 \end{array}$$



(7:3] Counter

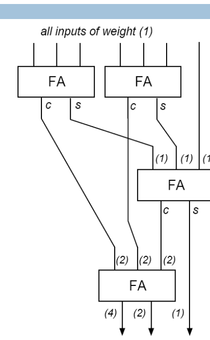


Figure 3.8: Implementation of (7:3] counter by an array of full adders.

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]

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

$$q_{B1} = [\text{any two bits}] \cdot (x_6 x_5 x_4 x_3)'$$
$$= aq'_{B2}$$
$$a = [x_6 x_5 + x_4 x_3 + (x_6 + x_5)(x_4 + x_3)]$$

$$q_{B2} = (x_6 x_5 x_4 x_3)'$$

X_6	X_5	X_4	X_3	q_{B2}	q_{B1}	q_{B0}
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	0	0	0	1
0	0	1	1	0	1	0
0	1	0	0	0	0	1
0	1	0	1	0	1	0
0	1	1	0	0	1	0
0	1	1	1	0	1	1
1	0	0	0	0	0	1
1	0	0	1	0	1	0
1	0	1	0	0	1	0
1	0	1	1	0	1	1
1	1	0	0	0	1	0
1	1	0	1	0	1	1
1	1	1	0	0	1	1
1	1	1	1	1	0	0

Gate Network for (7:3]

- Finally, $q = q_A + q_B$

$$\begin{array}{ccc} & q_{A1} & q_{A0} \\ q_{B2} & q_{B1} & q_{B0} \\ \hline q_2 & q_1 & q_0 \end{array}$$

$$q_0 = q_{A0} + q_{B0}$$

$$q_1 = \underbrace{(q_{A1} \oplus q_{B1})}_{\text{sum}} \oplus \underbrace{q_{A0} q_{B0}}_{\text{Carry from 0}}$$

$$q_2 = q_{B2} + \underbrace{q_{B1} q_{A1}}_{\text{G1}} + \underbrace{(q_{B1} \oplus q_{A1})(q_{B0} q_{A0})}_{\text{P1}\cdot\text{G0}}$$

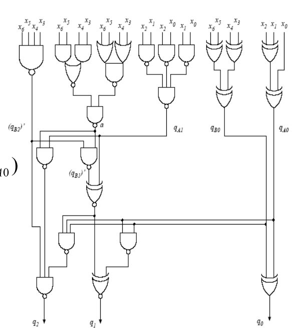
$$q_{B1} = a \cdot q'_{B2} \quad \Rightarrow \quad a \cdot q_{A1}$$

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...

$$(p_{k-1}, p_{k-2}, \ldots, p_0 : q]$$

$$v = \sum_{i=0}^{k-1} \sum_{j=1}^{p_i} a_{ij} 2^i \le 2^q - 1$$

$$v \le 1 \times 4 + 2 \times 2 + 3 \times 1 = 11 < 2^4 - 1$$

$$v \le 5 \times 2 + 5 \times 1 = 15 = 2^4 - 1$$

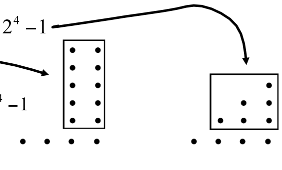
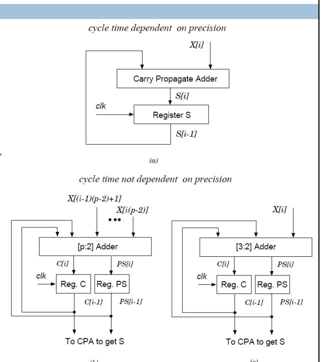


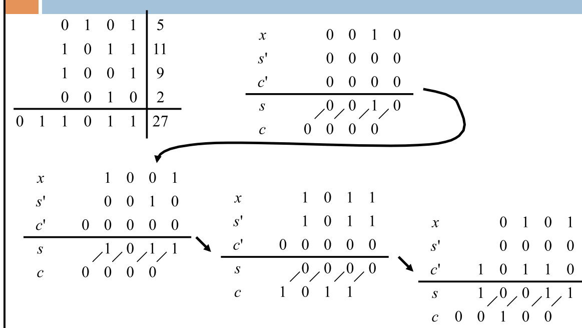
Figure 3.10: (a) (5,5:4] counter. (b) (1,2,3:4] counter.

Sequential Implementation

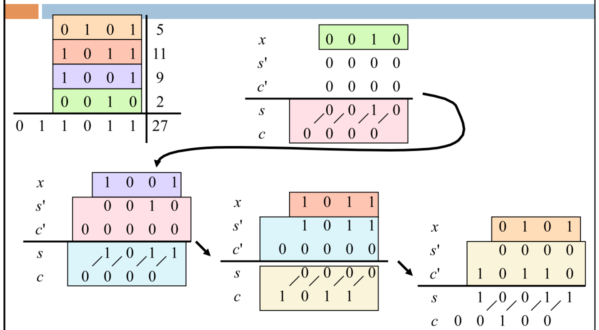
- If your input array is big
  - You can add rows sequentially
  - Uses only one adder and a register



Sequential Implementation



Sequential Implementation



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

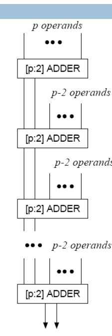
Linear Array

$\lceil (m-2)/(p-2) \rceil$ adders in the array

$n + p$ bits in final sum

where $p = \lceil \log_2(m) \rceil$

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 - the number of [p:2] CS adders for m operands:

$$pk = m + 2(k-1)$$

$$k = \left\lceil \frac{m-2}{p-2} \right\rceil \quad \text{[p:2] carry-save adders}$$

Adder Trees

• The number of adder levels

l	1	2	3	4	5	6	7	8	9
m_l	3	4	6	9	13	19	28	42	63

Reduction sequence

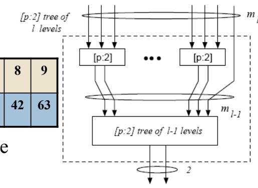


Figure 3.13: Construction of a [p:2] carry-save adder tree.

$$m_l = p\left\lfloor \frac{m_{l-1}}{2} \right\rfloor + m_{l-1} \bmod 2$$

Adder tree for 9 operands

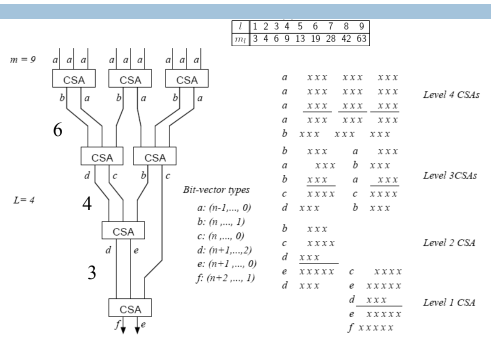
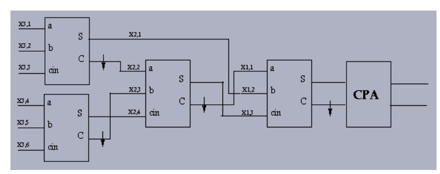


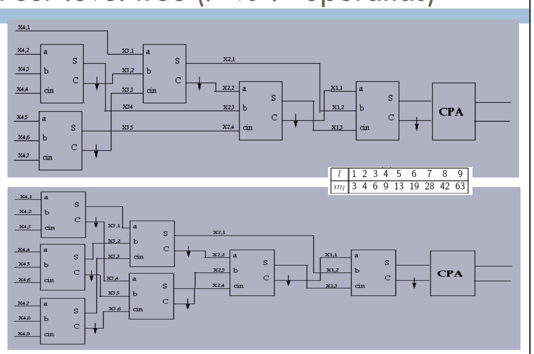
Figure 3.14: [3:2] adder tree for 9 operands (magnitudes with $n = 3$) .

Three-level tree (6 operands)

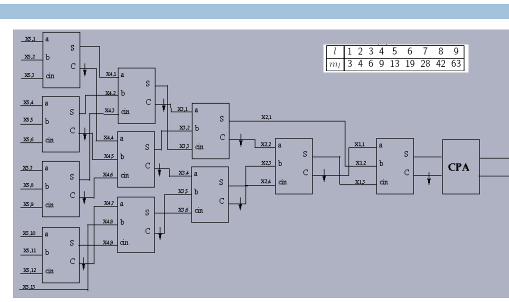


l	1	2	3	4	5	6	7	8	9
m_l	3	4	6	9	13	19	28	42	63

Four-level tree (7 vs 9 operands)



5-level Tree (13 inputs)



Tree of [4:2] adders for m=16

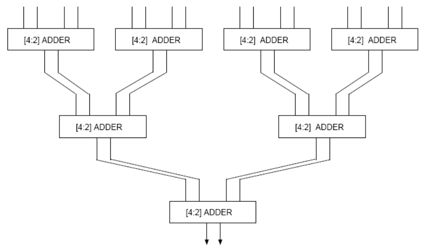


Figure 3.15: Tree of [4:2] adders for $m = 16$.

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

```
   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
  ---------
   0 1 0 1
 0 1 1 1
1 0 1 0
```

Reduction by Columns

- Why draw it that way?

```
    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
  ---------
    0 1 0 1
  0 1 1 1
1 0 1 0
```

```
     1 0 1 1  11
     0 0 1 0   2
     1 0 0 1   9
     0 1 1 0   6
     1 0 1 0  10
     1 1 1 1  15
     0 1 1 0   6
     ---------
       0 1 1
     1 1 0
   0 1 1
 1 0 0
 -------------
 1 1 1 0 1 1  59
```

Reduction by Columns

- Why draw it that way?

```
    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
  ---------
    0 1 0 1
  0 1 1 1
1 0 1 0
```

```
     1 0 1 1  11
     0 0 1 0   2
     1 0 0 1   9
     0 1 1 0   6
     1 0 1 0  10
     1 1 1 1  15
     0 1 1 0   6
     ---------
       0 1 1
     1 1 0
   0 1 1
 1 0 0
 -------------
 1 1 1 0 1 1  59
```

Number of Counter Levels

$$m_1 = p$$
$$m_l = p\left\lfloor \frac{m_{l-1}}{q} \right\rfloor + m_{l-1}\ mod\ q$$

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

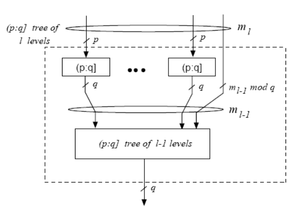


Figure 3.17: Construction of (p:q) reduction tree.

Sequences of Counters

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

Table 3.1: [3:2] Reduction sequence.

l	1	2	3	4	5	6	7	8	9
m_l	3	4	6	9	13	19	28	42	63

(3:2] counters

Table 3.2: Sequence for (7:3] counters

Number of levels	1	2	3	4	...
Max. number of rows	7	15	35	79	...

(7:3] counters

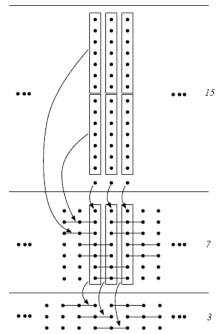
Example with (7:3] and 15 rows

Table says we can do it in 2 levels of counters

(example…)

Table 3.2: Sequence for (7:3] counters

Number of levels	1	2	3	4	…
Max. number of rows	7	15	35	79	…



Systemic Design Method

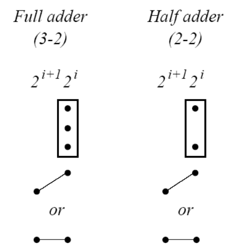


Table 3.1: [3:2] Reduction sequence.

l	1	2	3	4	5	6	7	8	9
m_l	3	4	6	9	13	19	28	42	63

• *denotes 0 or 1*

diagonal outputs when representing separately sum and carry bit-vectors is preferrable

horizontal outputs when interleaving sum and carry bits is acceptable

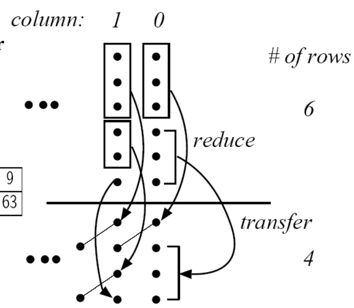
Reduction Process

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

Why not 4 FA?

Table 3.1: [3:2] Reduction sequence.

l	1	2	3	4	5	6	7	8	9
m_l	3	4	6	9	13	19	28	42	63



Relation at Level l

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

FA takes 3 bits, produces 1 sum (reduction of 2)

HA takes two bits, produces 1 sum

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

Plus carry bits from column to right

resulting in

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

Solution producing min number of carries:

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

Relation at Level l

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

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

resulting in

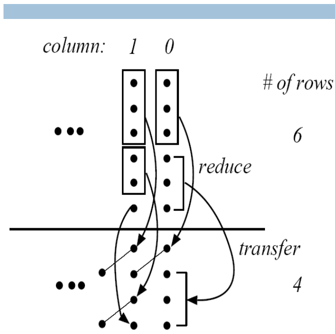
$$2f_i + h_i = e_i - m_{l-1} + f_{i-1} + h_{i-1} = p_i$$

Solution producing min number of carries:

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

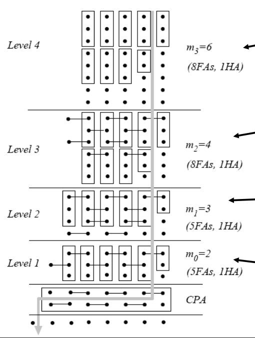
	i: 6	5	4	3	2	1	0
$l=4$							
e_i			8	8	8	8	8
m_3			6	6	6	6	6
h_i			0	0	0	1	0
f_i			2	2	2	1	1
$l=3$							
e_i		2	6	6	6	6	6
m_2		4	4	4	4	4	4
h_i		0	0	0	0	1	0
f_i		0	2	2	2	1	1
$l=2$							
e_i		4	4	4	4	4	4
m_1		3	3	3	3	3	3
h_i		0	0	0	0	0	1
f_i		1	1	1	1	1	0
$l=1$							
e_i	1	3	3	3	3	3	3
m_0	2	2	2	2	2	2	2
h_i	0	0	0	0	0	0	1
f_i	0	1	1	1	1	1	0

Relation at Level l



	i: 6	5	4	3	2	1	0
$l=4$							
e_i			8	8	8	8	8
m_3			6	6	6	6	6
h_i			0	0	0	1	0
f_i			2	2	2	1	1
$l=3$							
e_i		2	6	6	6	6	6
m_2		4	4	4	4	4	4
h_i		0	0	0	0	1	0
f_i		0	2	2	2	1	1
$l=2$							
e_i		4	4	4	4	4	4
m_1		3	3	3	3	3	3
h_i		0	0	0	0	0	1
f_i		1	1	1	1	1	0
$l=1$							
e_i	1	3	3	3	3	3	3
m_0	2	2	2	2	2	2	2
h_i	0	0	0	0	0	0	1
f_i	0	1	1	1	1	1	0

m=8, n=5 (26 FA, 4HA)



	i 6	5	4	3	2	1	0
$l=4$							
e_i			8	8	8	8	8
m_3			6	6	6	6	6
h_i			0	0	0	1	0
f_i			2	2	2	1	1
$l=3$							
e_i		2	6	6	6	6	6
m_2		4	4	4	4	4	4
h_i		0	0	0	0	1	0
f_i		0	2	2	2	1	1
$l=2$							
e_i		4	4	4	4	4	4
m_1		3	3	3	3	3	3
h_i		0	0	0	0	0	1
f_i		1	1	1	1	1	0
$l=1$							
e_i	1	3	3	3	3	3	3
m_0	2	2	2	2	2	2	2
h_i	0	0	0	0	0	0	1
f_i	0	1	1	1	1	1	0

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:

$$-4 + (-12) + (-12) - 4 = -32 \le f \le 3 + 9 + 9 + 3 = 24$$

  - Thus f requires 6 bits
  - Decompose 3b and 3c into 2b+b and 2c+c

Example: $f=a+3b+3c+d$

$$\begin{array}{c|cccccc}
a & a_2 & a_2 & a_2 & a_2 & a_1 & a_0 \\
b & b_2 & b_2 & b_2 & b_2 & b_1 & b_0 \\
2b & b_2 & b_2 & b_2 & b_1 & b_0 & 0 \\
c & c_2 & c_2 & c_2 & c_2 & c_1 & c_0 \\
2c & c_2 & c_2 & c_2 & c_1 & c_0 & 0 \\
d & d_2 & d_2 & d_2 & d_2 & d_1 & d_0
\end{array}$$

Note left shifts to compute $2b$ and $2c$…

Example: $f=a+3b+3c+d$

$$\begin{array}{c|cccccc}
a & a_2 & a_2 & a_2 & a_2 & a_1 & a_0 \\
b & b_2 & b_2 & b_2 & b_2 & b_1 & b_0 \\
2b & b_2 & b_2 & b_2 & b_1 & b_0 & 0 \\
c & c_2 & c_2 & c_2 & c_2 & c_1 & c_0 \\
2c & c_2 & c_2 & c_2 & c_1 & c_0 & 0 \\
d & d_2 & d_2 & d_2 & d_2 & d_1 & d_0
\end{array}
\longrightarrow
\begin{array}{c|cccc}
a & & a_2' & a_1 & a_0 \\
 & & \text{-1} & & \\
b & & b_2' & b_1 & b_0 \\
 & & \text{-1} & & \\
2b & b_2' & b_1 & b_0 & \\
 & \text{-1} & & & \\
c & & c_2' & c_1 & c_0 \\
 & & \text{-1} & & \\
2c & c_2' & c_1 & c_0 & \\
 & \text{-1} & & & \\
d & & d_2' & d_1 & d_0 \\
 & & \text{-1} & &
\end{array}$$

Use sign extension trick

Keep track of weight of sign extension bits!

-4x4 + -8x2 = -32

-32 = 100000

Example: $f=a+3b+3c+d$

Remember the 100000 (-32)

$$\begin{array}{c|cccc}
a & & a_2' & a_1 & a_0 \\
 & & \text{-1} & & \\
b & & b_2' & b_1 & b_0 \\
 & & \text{-1} & & \\
2b & b_2' & b_1 & b_0 & \\
 & \text{-1} & & & \\
c & & c_2' & c_1 & c_0 \\
 & & \text{-1} & & \\
2c & c_2' & c_1 & c_0 & \\
 & \text{-1} & & & \\
d & & d_2' & d_1 & d_0 \\
 & & \text{-1} & &
\end{array}
\longrightarrow
\begin{array}{|cccccc}
1 & 0 & b_2' & a_2' & a_1 & a_0 \\
 & & c_2' & b_2' & b_1 & b_0 \\
 & & & b_1 & b_0 & \\
 & & & c_2' & c_1 & c_0 \\
 & & & c_1 & c_0 & \\
 & & & d_2' & d_1 & d_0
\end{array}$$

Example: $f=a+3b+3c+d$

$$\begin{array}{|cccccc}
1 & 0 & b_2' & a_2' & a_1 & a_0 \\
 & & c_2' & b_2' & b_1 & b_0 \\
 & & & b_1 & b_0 & \\
 & & & c_2' & c_1 & c_0 \\
 & & & c_1 & c_0 & \\
 & & & d_2' & d_1 & d_0
\end{array}$$

	i 5	4	3	2	1	0
$l=3$						
e_i	1	0	2	6	6	4
m_2	4	4	4	4	4	4
h_i	0	0	0	1	0	0
f_i	0	0	0	1	1	0
$l=2$						
e_i	1	0	4	4	4	4
m_1	3	3	3	3	3	3
h_i	0	0	0	0	0	1
f_i	0	0	1	1	1	0
$l=1$						
e_i	1	1	3	3	3	3
m_0	2	2	2	2	2	1*
h_i	0	0	0	0	0	0
f_i	0	0	1	1	1	1

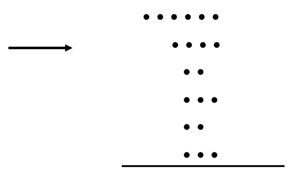
Determine adder types at each level

* Helps by reducing the width of the CPA

Example: $f=a+3b+3c+d$

$$\begin{array}{cccccc}
1 & 0 & b_2' & a_2' & a_1 & a_0 \\
 & & c_2' & b_2' & b_1 & b_0 \\
 & & & b_1 & b_0 & \\
 & & & c_2' & c_1 & c_0 \\
 & & & c_1 & c_0 & \\
 & & & d_2' & d_1 & d_0
\end{array}$$

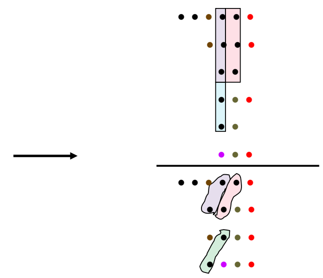
	i = 5	4	3	2	1	0
$l=3$						
e_i	1	0	2	6	6	4
m_2	4	4	4	4	4	4
h_i	0	0	0	1	0	0
f_i	0	0	0	1	1	0
$l=2$						
e_i	1	0	4	4	4	4
m_1	3	3	3	3	3	3
h_i	0	0	0	0	0	1
f_i	0	0	1	1	1	0
$l=1$						
e_i	1	1	3	3	3	3
m_0	2	2	2	2	2	1*
h_i	0	0	0	0	0	0
f_i	0	0	1	1	1	1



Example: $f=a+3b+3c+d$

$$\begin{array}{cccccc}
1 & 0 & b_2' & a_2' & a_1 & a_0 \\
 & & c_2' & b_2' & b_1 & b_0 \\
 & & & b_1 & b_0 & \\
 & & & c_2' & c_1 & c_0 \\
 & & & c_1 & c_0 & \\
 & & & d_2' & d_1 & d_0
\end{array}$$

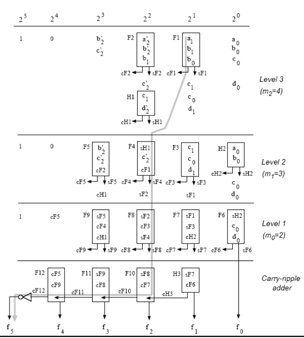
	i = 5	4	3	2	1	0
$l=3$						
e_i	1	0	2	6	6	4
m_2	4	4	4	4	4	4
h_i	0	0	0	1	0	0
f_i	0	0	0	1	1	0
$l=2$						
e_i	1	0	4	4	4	4
m_1	3	3	3	3	3	3
h_i	0	0	0	0	0	1
f_i	0	0	1	1	1	0
$l=1$						
e_i	1	1	3	3	3	3
m_0	2	2	2	2	2	1*
h_i	0	0	0	0	0	0
f_i	0	0	1	1	1	1



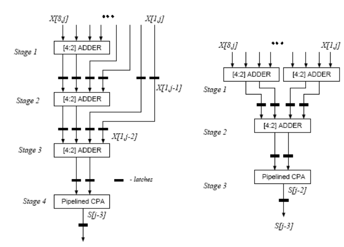
Example: $f=a+3b+3c+d$

$$\begin{array}{cccccc}
1 & 0 & b_2' & a_2' & a_1 & a_0 \\
 & & c_2' & b_2' & b_1 & b_0 \\
 & & & b_1 & b_0 & \\
 & & & c_2' & c_1 & c_0 \\
 & & & c_1 & c_0 & \\
 & & & d_2' & d_1 & d_0
\end{array}$$

	i = 5	4	3	2	1	0
$l=3$						
e_i	1	0	2	6	6	4
m_2	4	4	4	4	4	4
h_i	0	0	0	1	0	0
f_i	0	0	0	1	1	0
$l=2$						
e_i	1	0	4	4	4	4
m_1	3	3	3	3	3	3
h_i	0	0	0	0	0	1
f_i	0	0	1	1	1	0
$l=1$						
e_i	1	1	3	3	3	3
m_0	2	2	2	2	2	1*
h_i	0	0	0	0	0	0
f_i	0	0	1	1	1	1

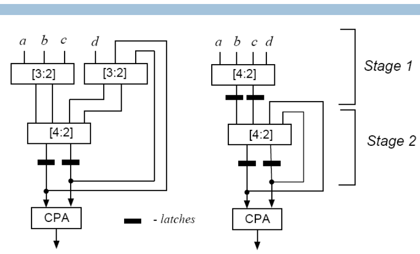


Pipelined Arrays



You can always add pipelining to increase throughput!

Partially Combinational Arrays

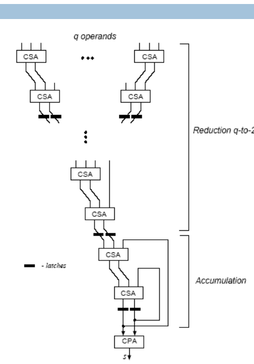


Non-pipelined

Pipelined

k operands are added per iteration (k = 4 in these examples)
m/k iterations required for complete result

Partially Combinational Arrays



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