DO NOT SHARE
SLIDES AND CLASS MATERIALS
ON ONLINE SITES

Advanced Logic Design
Computer Arithmetic

Mingoo Seok
Columbia University

BV: Sec. 3.1-3.4, 3.6, 7.4-7.7

Unsigned Integer

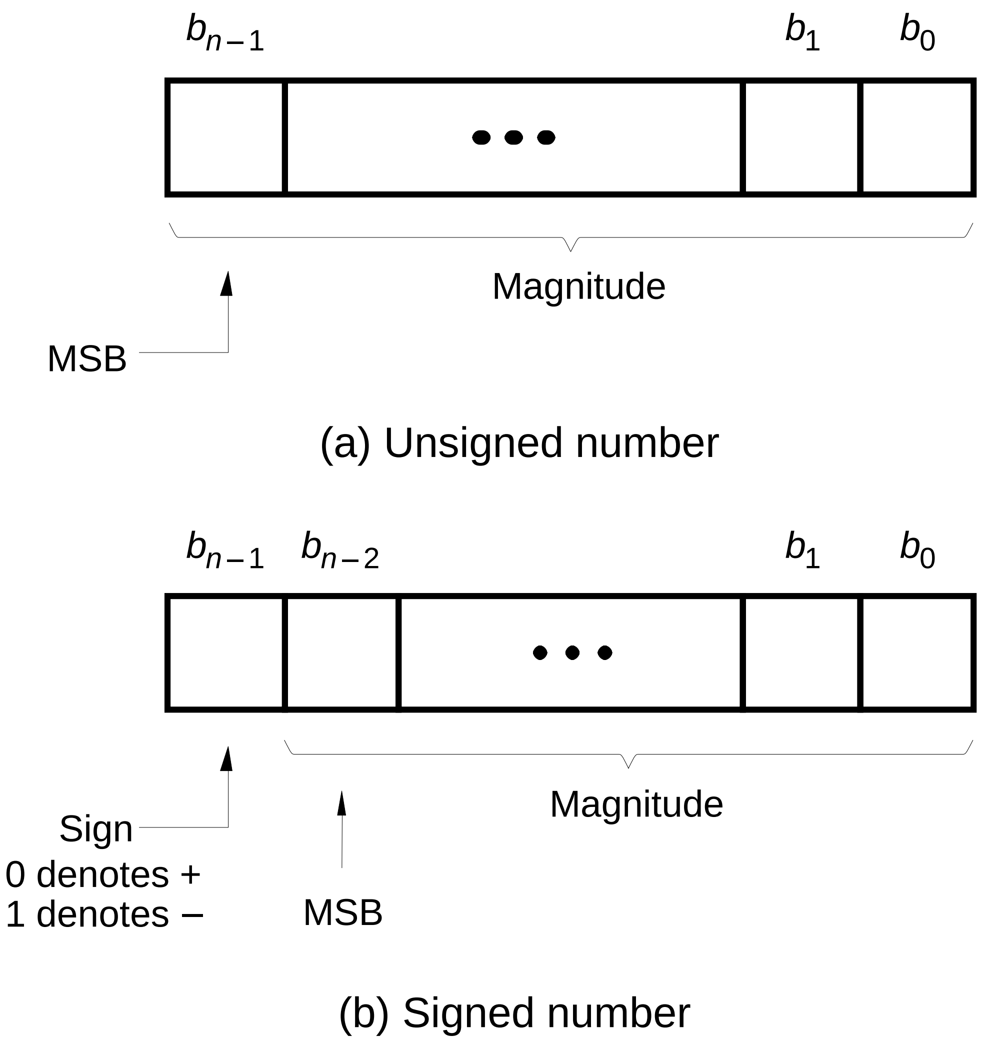
$$B = b_{n-1}b_{n-2}\cdots b_1 b_0$$

$$\begin{aligned} V(B) &= b_{n-1} \times 2^{n-1} + b_{n-2} \times 2^{n-2} + \cdots + b_1 \times 2^1 + b_0 \times 2^0 \\ &= \sum_{i=0}^{n-1} b_i \times 2^i \end{aligned}$$

- Above is for unsigned binary integer (base 2)
- Base 8: octal
- Base 16: hexadecimal
- Base 10: decimal

Decimal	Binary	Octal	Hexadecimal
00	00000	00	00
01	00001	01	01
02	00010	02	02
03	00011	03	03
04	00100	04	04
05	00101	05	05
06	00110	06	06
07	00111	07	07
08	01000	10	08
09	01001	11	09
10	01010	12	0A
11	01011	13	0B
12	01100	14	0C
13	01101	15	0D
14	01110	16	0E
15	01111	17	0F
16	10000	20	10
17	10001	21	11
18	10010	22	12

Signed Binary Integer



- MSB
- LSB
- Sign and Magnitude

2’s Complement

$$w = -a_{N-1}2^{N-1} + \sum_{i=0}^{N-2} a_i 2^i.$$

- MSB is negative-weighted
- In decimal, the negative counterpart of 3 is -3
- In 2’s complement, what is the negative counterpart of 0111?

2’s Complement

- Let K be the negative equivalent of an n-bit positive number P.
- Then, in 2’s complement representation K is obtained by subtracting P from 2^n , namely

$$K = 2^n - P$$

- For a positive n-bit number P, let K_1 and K_2 denote its 1’s and 2’s complements, respectively.

$$K_1 = (2^n - 1) - P$$

$$K_2 = 2^n - P$$

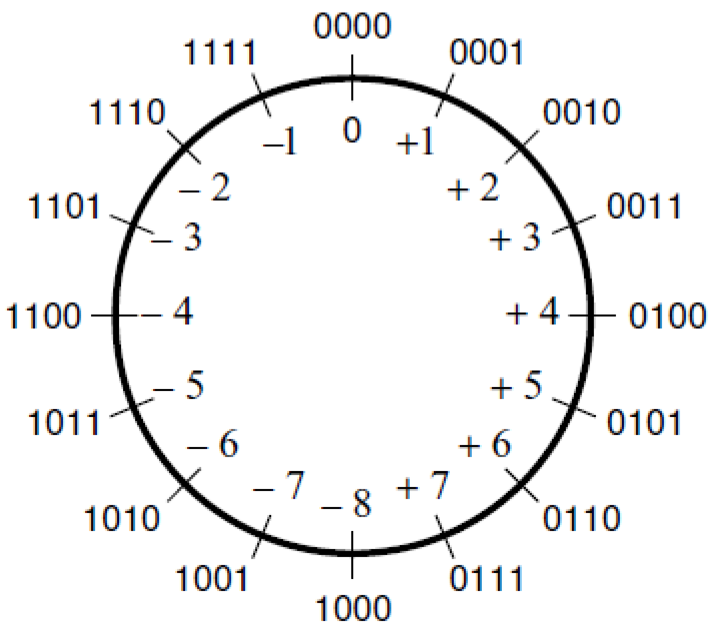
- Since $K_2 = K_1 + 1$, it is evident that in a logic circuit the 2’s complement can computed by inverting all bits of P and then adding 1 to the resulting 1’s-complement number.

2's Complement

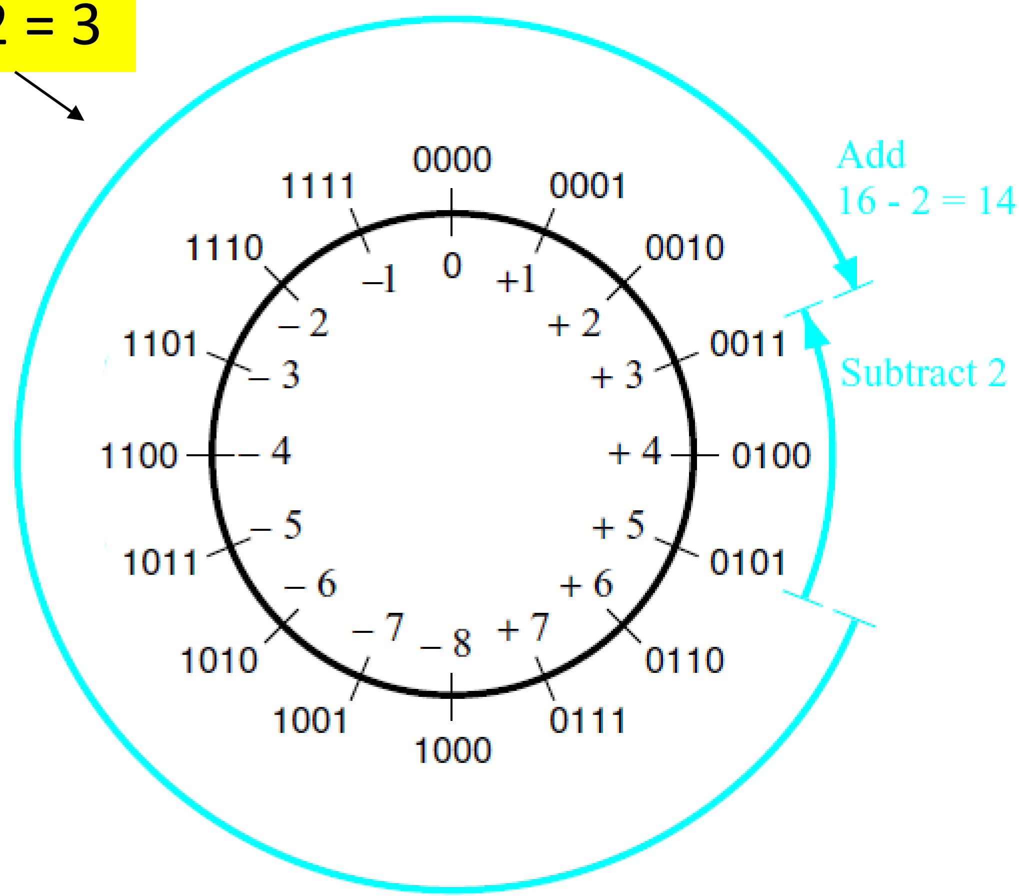
$b_3b_2b_1b_0$	Sign and magnitude	1's complement	2's complement
0111	+7	+7	+7
0110	+6	+6	+6
0101	+5	+5	+5
0100	+4	+4	+4
0011	+3	+3	+3
0010	+2	+2	+2
0001	+1	+1	+1
0000	+0	+0	+0
1000	−0	−7	−8
1001	−1	−6	−7
1010	−2	−5	−6
1011	−3	−4	−5
1100	−4	−3	−4
1101	−5	−2	−3
1110	−6	−1	−2
1111	−7	−0	−1

2's Complement

5-2 = 3



(a) The number circle



(b) Subtracting 2 by adding its 2's complement

2's Complement Add & Subtract

$$
\begin{array}{rr}
 & (+5) \\
+ & (+2) \\
\hline
 & (+7)
\end{array}
\qquad
\begin{array}{rr}
 & 0101 \\
+ & 0010 \\
\hline
 & 0111
\end{array}
\qquad\qquad
\begin{array}{rr}
 & (-5) \\
+ & (+2) \\
\hline
 & (-3)
\end{array}
\qquad
\begin{array}{rr}
 & 1011 \\
+ & 0010 \\
\hline
 & 1101
\end{array}
$$

$$
\begin{array}{rr}
 & (+5) \\
+ & (-2) \\
\hline
 & (+3)
\end{array}
\qquad
\begin{array}{rr}
 & 0101 \\
+ & 1110 \\
\hline
 & 10011
\end{array}
\qquad\qquad
\begin{array}{rr}
 & (-5) \\
+ & (-2) \\
\hline
 & (-7)
\end{array}
\qquad
\begin{array}{rr}
 & 1011 \\
+ & 1110 \\
\hline
 & 11001
\end{array}
$$

ignore

ignore

$$\begin{array}{r} (+\,5) \\ -\ (+\,2) \\ \hline (+\,3) \end{array} \qquad \begin{array}{r} 0\,1\,0\,1 \\ -\ 0\,0\,1\,0 \\ \hline \end{array} \qquad \Longrightarrow \qquad \begin{array}{r} 0\,1\,0\,1 \\ +\ 1\,1\,1\,0 \\ \hline 1\,0\,0\,1\,1 \end{array}$$

ignore

$$\begin{array}{r} (-5) \\ -\ (+\,2) \\ \hline (-7) \end{array} \qquad \begin{array}{r} 1\,0\,1\,1 \\ -\ 0\,0\,1\,0 \\ \hline \end{array} \qquad \Longrightarrow \qquad \begin{array}{r} 1\,0\,1\,1 \\ +\ 1\,1\,1\,0 \\ \hline 1\,1\,0\,0\,1 \end{array}$$

ignore

$$\begin{array}{r} (+\,5) \\ -\ (-2) \\ \hline (+\,7) \end{array} \qquad \begin{array}{r} 0\,1\,0\,1 \\ -\ 1\,1\,1\,0 \\ \hline \end{array} \qquad \Longrightarrow \qquad \begin{array}{r} 0\,1\,0\,1 \\ +\ 0\,0\,1\,0 \\ \hline 0\,1\,1\,1 \end{array}$$

$$\begin{array}{r} (-5) \\ -\ (-2) \\ \hline (-3) \end{array} \qquad \begin{array}{r} 1\,0\,1\,1 \\ -\ 1\,1\,1\,0 \\ \hline \end{array} \qquad \Longrightarrow \qquad \begin{array}{r} 1\,0\,1\,1 \\ +\ 0\,0\,1\,0 \\ \hline 1\,1\,0\,1 \end{array}$$

Overflow

$$\begin{array}{r} (+7) \\ +\ (+2) \\ \hline (+9) \end{array} \qquad \begin{array}{r} 0\,1\,1\,1 \\ +\ 0\,0\,1\,0 \\ \hline 1\,0\,0\,1 \end{array} \qquad \begin{array}{r} (-7) \\ +\ (+2) \\ \hline (-5) \end{array} \qquad \begin{array}{r} 1\,0\,0\,1 \\ +\ 0\,0\,1\,0 \\ \hline 1\,0\,1\,1 \end{array}$$

$$c_4 = 0 \qquad c_4 = 0$$
$$c_3 = 1 \qquad c_3 = 0$$

$$\begin{array}{r} (+7) \\ +\ (-2) \\ \hline (+5) \end{array} \qquad \begin{array}{r} 0\,1\,1\,1 \\ +\ 1\,1\,1\,0 \\ \hline 1\,0\,1\,0\,1 \end{array} \qquad \begin{array}{r} (-7) \\ +\ (-2) \\ \hline (-9) \end{array} \qquad \begin{array}{r} 1\,0\,0\,1 \\ +\ 1\,1\,1\,0 \\ \hline 1\,0\,1\,1\,1 \end{array}$$

$$c_4 = 1 \qquad c_4 = 1$$
$$c_3 = 1 \qquad c_3 = 0$$

- Carry out (c_3) in the MSB position and c_4 in the MSB+1 position are same? No overflow
- Different? Overflow.

Sign Extension

- In 2’s complement,
  - 00 0000 1010 is equal to 0000 0000 1010
  - 11 1111 0001 is equal to 1111 1111 0001

- Even if you extend the sign bit infinitely, the value remains the same.

Fixed-Point Number

- It has not only integer but also fractional parts
- A *W*-bit fixed-point number *A* is represented as

$$A = a_{W-1} \bullet a_{W-2} \ldots a_1 a_0$$

  - The range of this number is [-1, $1-2^{-(W-1)}$], or often shown as [-1, 1)
  - The value of this number is

$$A = -a_{W-1} + \sum_{i=1}^{W-1} a_{W-1-i} 2^{-i}$$

- The location of radix point is set by you

Q Notation

```
. 0011.1010        3.6250
+ 0100.0001      + 4.0625
= 0111.1011      = 7.6875
```

```
. 0011.1010        3.6250
+ 1110.1000      - 1.5000
= 0010.0010      = 2.1250
```

```
. 0011.0100        3.2500
x 0010.0001        2.0625
```

```
00000110.10110100 = 6.703125
```

```
0000[0110.1011]0100 = 0110.1011 = 6.6875
```

- Q*i.f* where *i* is the number of integer bits and *f* is the number of fractional bits
  - 0011_1010 in Q4.4 → 3.6250
- All math operations are done in the same way as 2's complement integer

https://timetoexplore.net/blog/fixed-point-numbers-in-verilog

Range and Precision

- You can get a larger <u>range</u> by using more *integer* bits and;
- better <u>precision</u> by using more *fraction* bits.
  – Q4.4: Range: -8 to 7.9375 (7+15/16); Precision: 0.0625 ($1/2^4$)
  – Q16.16: Range: -32768 to 32767.9999847…; Precision: 0.0000152… ($1/2^{16}$)

https://timetoexplore.net/blog/fixed-point-numbers-in-verilog

Overflow and Precision Loss

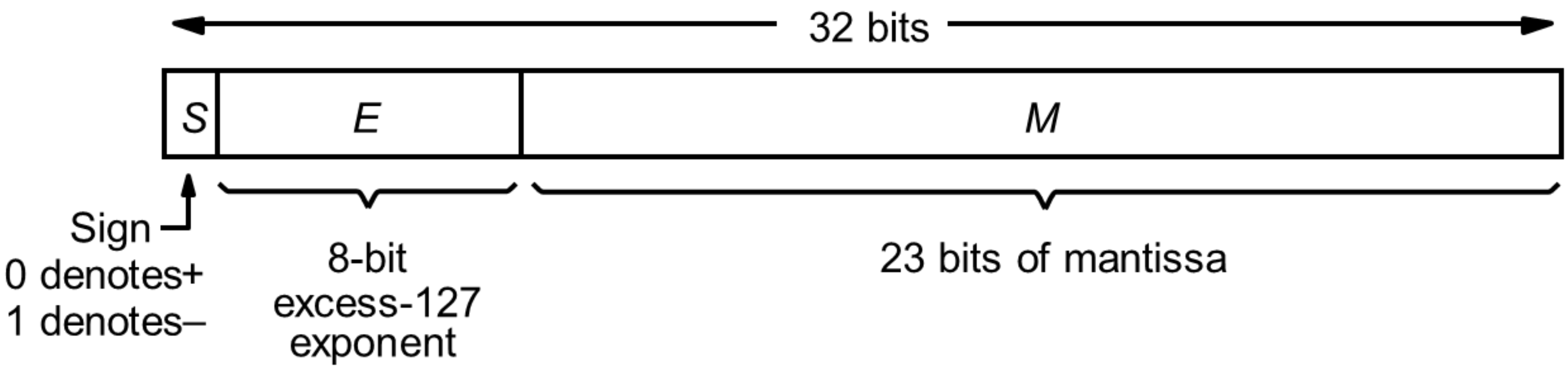
```
. 0110.1000          6.5000
x 0100.0000          4.0000
00011010.00000000   26.0000
```

```
          0110.1001
X         0100.0001
-------------------------
[00011010.1010]1001
```

- Computation result can go outside of the range of the inputs
  - Q4.4
  - In the above example, 1010.0000 is -6, not 26
  - In multiplication, if any of the first four bits contain 1 then we've overflowed
- Precision loss
  - In the above example (right), we lose the precision as we truncate LSBs
  - Loss amount: 0000.0000**1001** or $2^{-5} + 2^{-8}$ or 0.03515625

https://timetoexplore.net/blog/fixed-point-numbers-in-verilog

Floating-Point Number

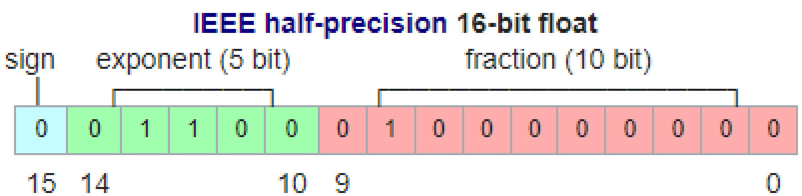


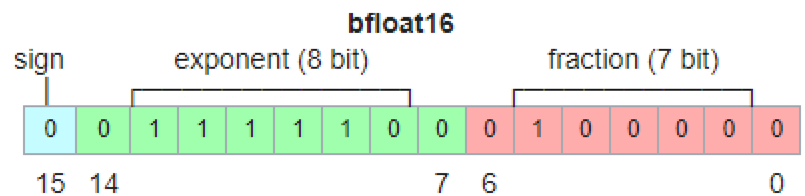
$$\mathbf{1.M} \times R^{Exponent} \qquad Exponent = E - 127$$

- 1.M
- **R** is 2
- **E** is a non-negative integer ranging from 0 to 255; thus **Exponent** ranges from -126 to 127
- Value = (+/-)1.M X 2^{E-127}
- Devised to handle a large range and a high precision

Floating-Point Number Types

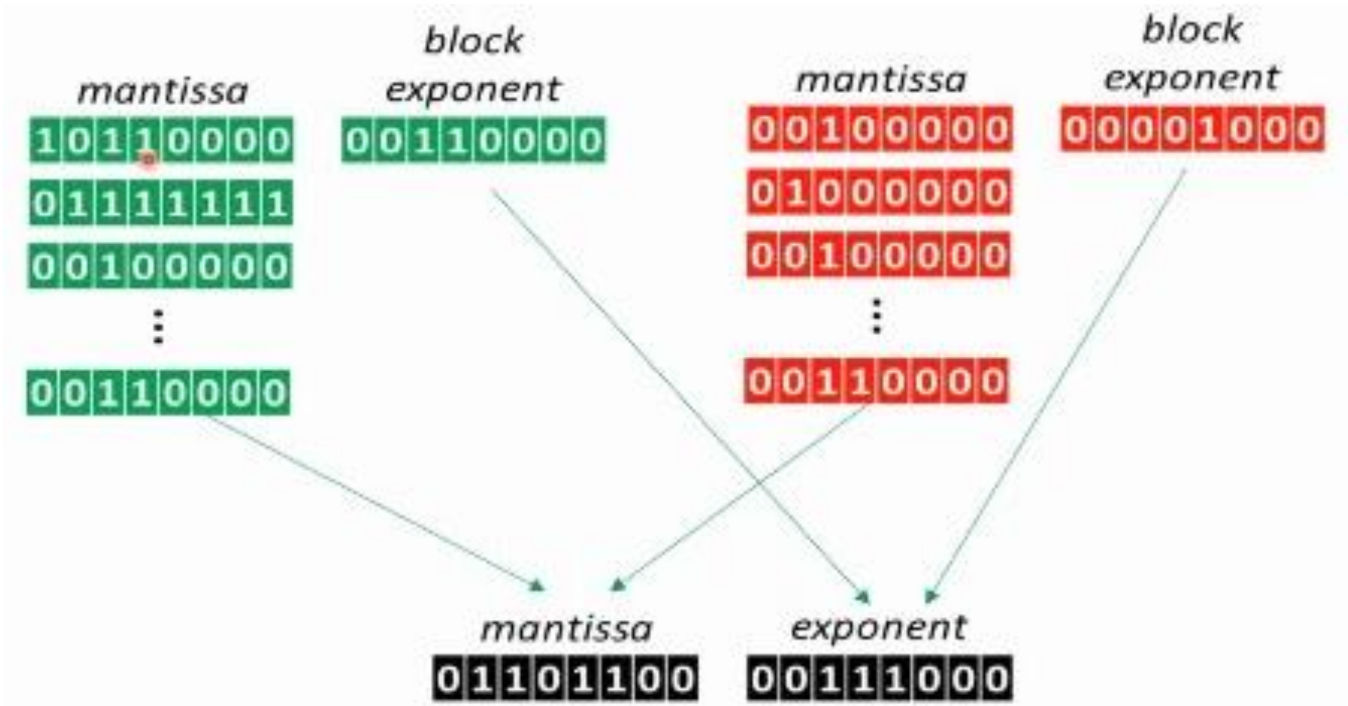
Name	Common name	Base	Significand bits[b] or digits	Decimal digits	Exponent bits	Decimal E max	Exponent bias[12]	E min	E max	Notes
binary16	Half precision	2	11	3.31	5	4.51	$2^4-1 = 15$	−14	+15	not basic
binary32	Single precision	2	24	7.22	8	38.23	$2^7-1 = 127$	−126	+127	
binary64	Double precision	2	53	15.95	11	307.95	$2^{10}-1 = 1023$	−1022	+1023	
binary128	Quadruple precision	2	113	34.02	15	4931.77	$2^{14}-1 = 16383$	−16382	+16383	
binary256	Octuple precision	2	237	71.34	19	78913.2	$2^{18}-1 = 262143$	−262142	+262143	not basic
decimal32		10	7	7	7.58	96	101	−95	+96	not basic
decimal64		10	16	16	9.58	384	398	−383	+384	
decimal128		10	34	34	13.58	6144	6176	−6143	+6144	





Source: Wikipedia

Block Floating Point

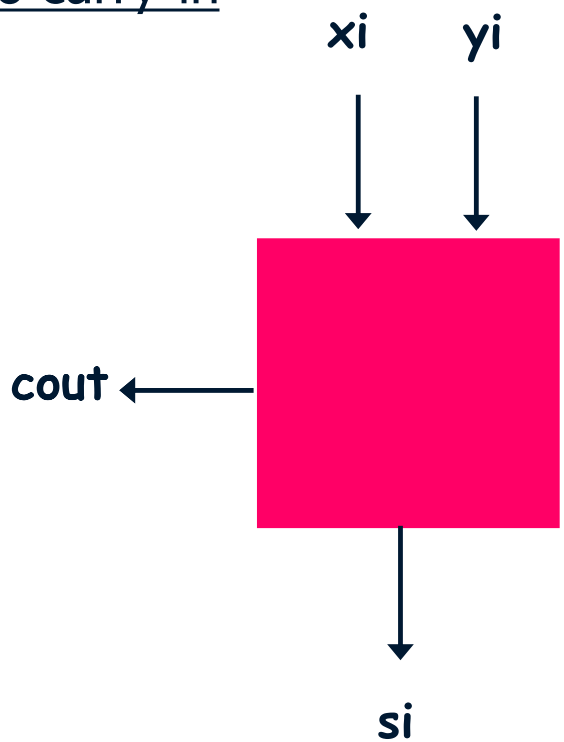


- Works well for vectors and matrixes
- Good for deep convolutional neural networks

https://semiwiki.com/efpga/283186-webinar-fpga-native-block-floating-point-for-optimizing-ai-ml-workloads/

Basic Building Blocks: Half Adder (HA)

Purpose: add two 1-bit operands with no carry-in

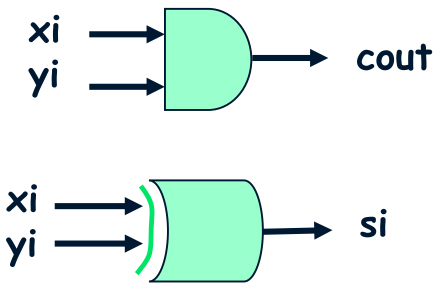


xi	yi	si	cout
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Truth Table

cout = xi yi

si = xi XOR yi

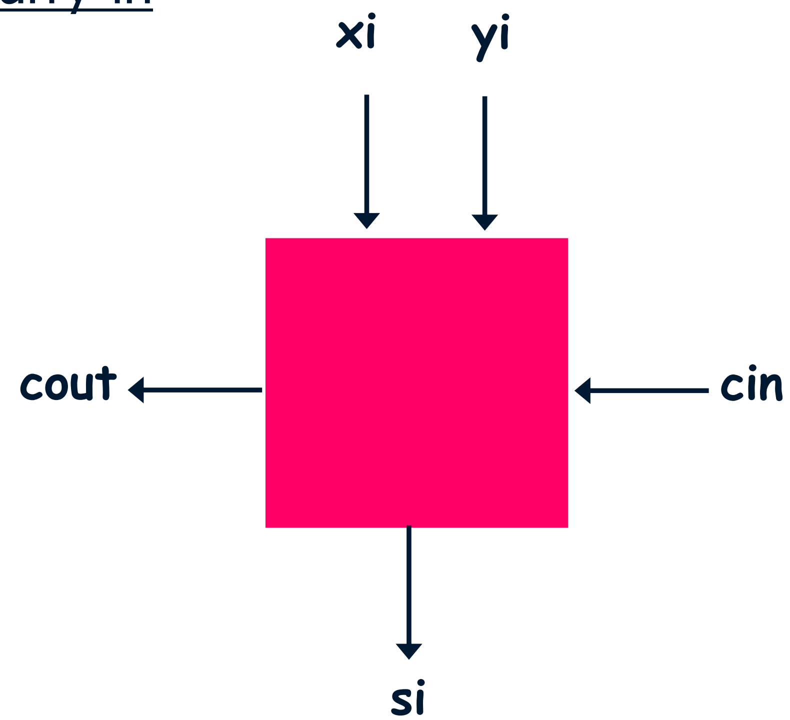


Basic Building Blocks: Full Adder (FA)

Purpose: add two 1-bit operands <u>with carry-in</u>

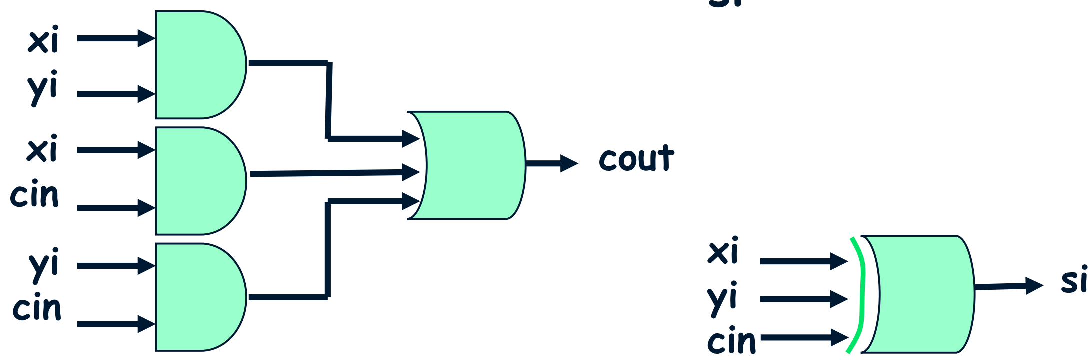
cin	xi	yi	si	cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Truth Table



cout = xi yi + xi cin + yi cin
= MAJORITY Function

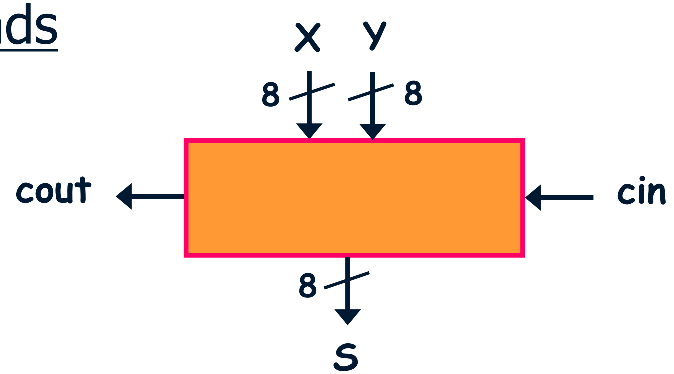
si = xi XOR yi XOR cin
= ODD PARITY Function

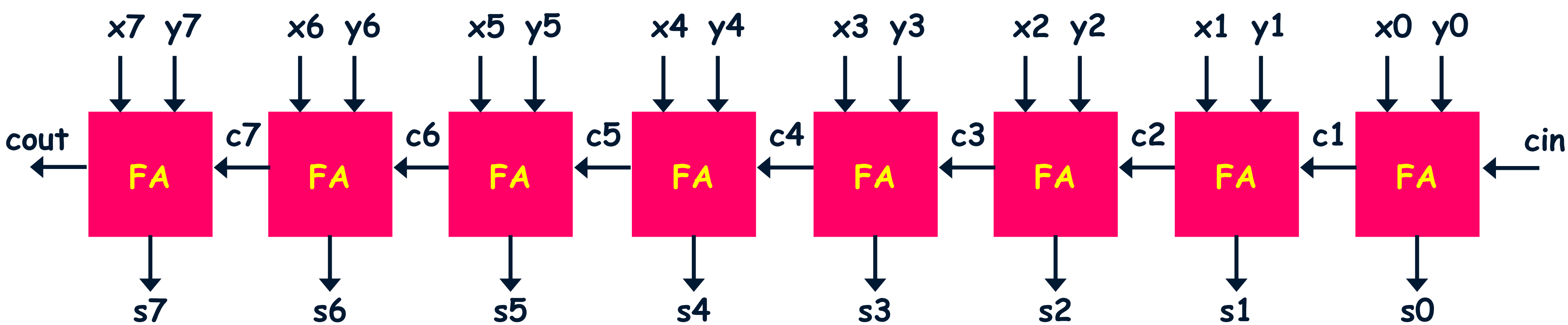


Ripple-Carry Adder (RCA): 8-Bit Example

Purpose: add two N-bit operands

Top-level block view:





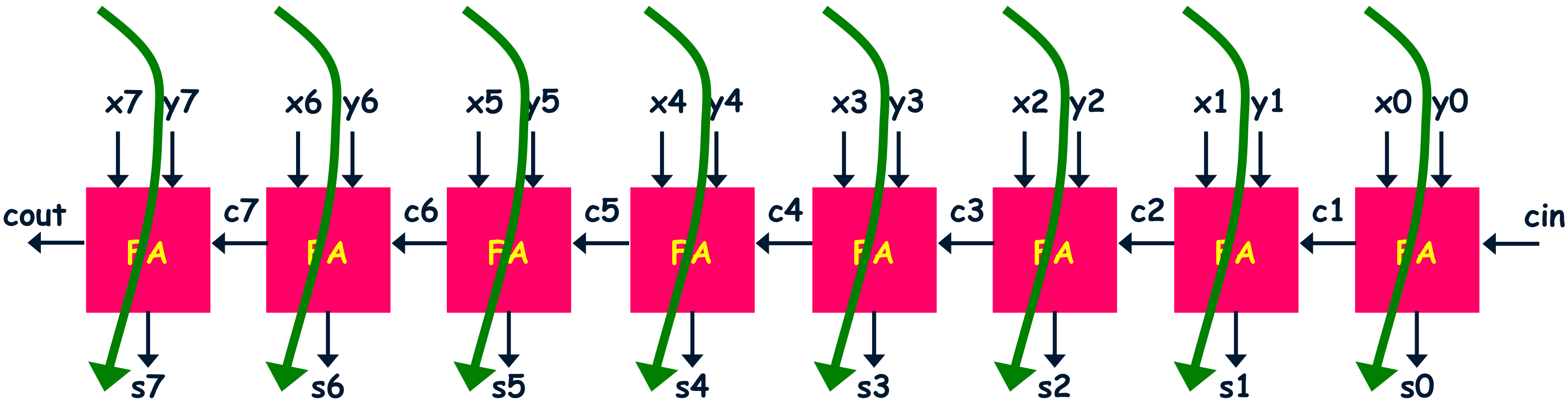
Detailed structural view:

Ripple-Carry Adder (RCA): Simulations

Example #1: best-case = no carry chain *(all carries are 0)*

Addition:

$$\begin{array}{rcr} 00110010 & = & 50 \\ +\ \underline{00000100} & = & \underline{4} \\ 00110110 & = & 54 \end{array}$$

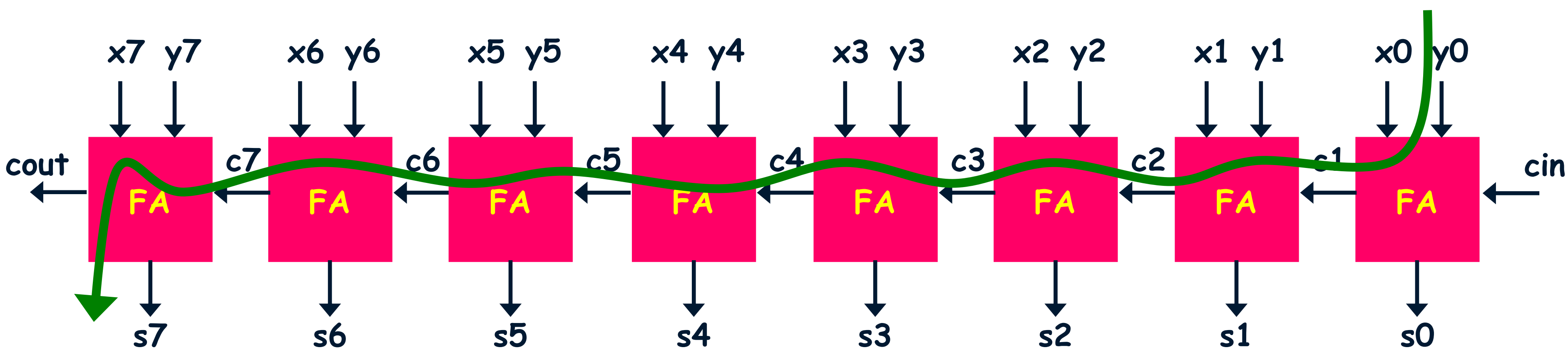


Ripple-Carry Adder (RCA): Simulations

Example #2: worst-case = longest carry chain *(all carries are 1)*

Addition:

00000001	=	1	
+ 11111111	=	-1	(2s Complement Representation)
00000000	=	0	



Carry Lookahead Adder (CLA)

$$c_{i+1} = x_i y_i + x_i c_i + y_i c_i$$

$$c_{i+1} = x_i y_i + (x_i + y_i) c_i$$

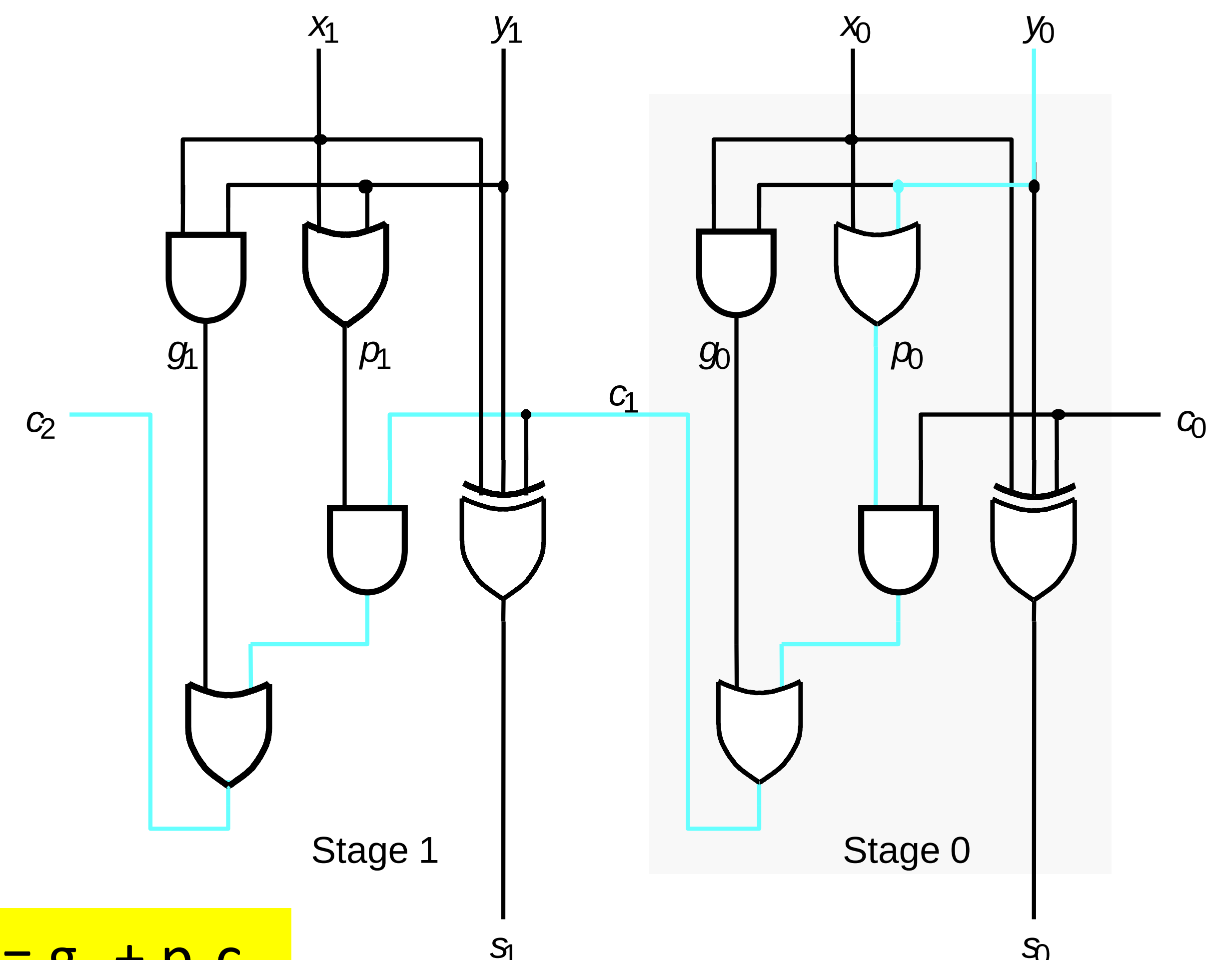
$$c_{i+1} = g_i + p_i c_i$$

$$g_i = x_i y_i$$

$$p_i = x_i + y_i$$

$$c_{i+1} = g_i + p_i g_{i-1} + p_i p_{i-1} g_{i-2} + \cdots + p_i p_{i-1} \cdots p_2 p_1 g_0 + p_i p_{i-1} \cdots p_1 p_0 c_0$$

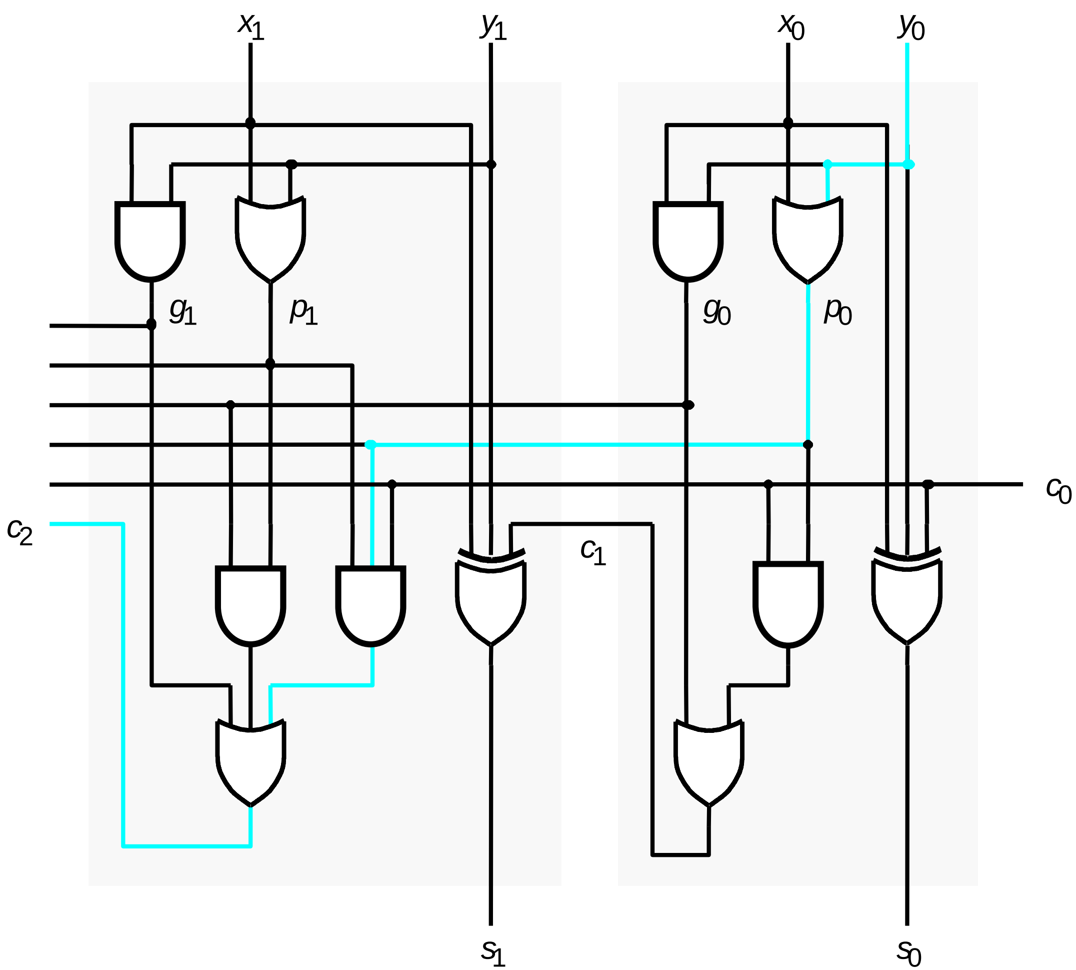
Ripple Carry Design



$c_1 = g_0 + p_0c_0$
$c_2 = g_1 + p_1c_1$

- Ripple carry design
- C2 delay is 5 gate delay
- For n bits, it becomes (2n + 1) gate delay

Carry Lookahead Adder (CLA)



- C2 is generated after three gate delays
- Need another xor gate to produce sum (output)
- ☹ The 2-level SOP becomes very large for a practical *n*
- Hierarchical CLA

$c_2 = g_1 + p_1g_0+p_1p_0c_0$

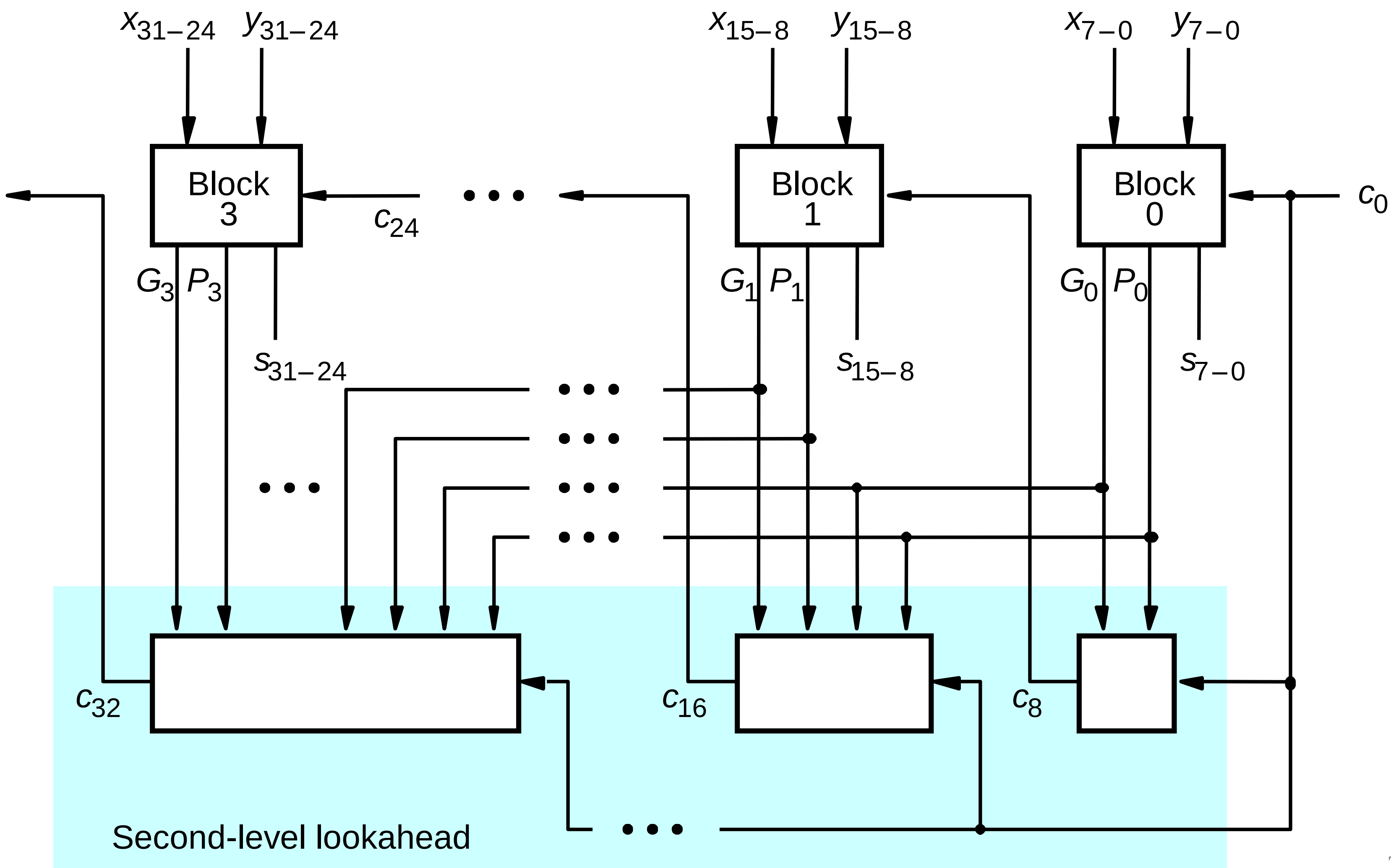
Hierarchical CLA

- Divide 32-b adder into 4 8-bit sections
- Generate P's and G's of each section

$$P_0 = p_7 p_6 p_5 p_4 p_3 p_2 p_1 p_0 \quad G_0 = g_7 + p_7 g_6 + p_7 p_6 g_5 + \cdots + p_7 p_6 p_5 p_4 p_3 p_2 p_1 g_0$$

- Produce the carry out of each section, i.e., c_8, c_{16}, c_{24}, $c_{32.}$ e.g.,
  - $c_8 = G_0 + P_0 c_0$
  - $c_{16} = G_8 + P_8 G_0 + P_8 P_0 c_0$
- Perform the additions of the sections using the 8-bit CLA method, all in parallel

Hierarchical CLA



Multiplication

$$A = a_{W-1} \bullet a_{W-2} \ldots a_1 a_0 = -a_{W-1} + \sum_{i=1}^{W-1} a_{W-1-i} 2^{-i}$$

$$B = b_{W-1} \bullet b_{W-2} \ldots b_1 b_0 = -b_{W-1} + \sum_{i=1}^{W-1} b_{W-1-i} 2^{-i}$$

– The full-precision product (2W-1 bits) is given by

$$P = -p_{2W-2} + \sum_{i=1}^{2W-2} p_{2W-2-i} 2^{-i}$$

– A $W \times W$ multiplication generates $2W$-1 bit product

– In constant word length multiplication, (W-1) LSBs in the product are ignored (truncated)

$$X = -x_{2W-2} + \sum_{i=1}^{W-1} x_{W-1-i} 2^{-i}$$

Tabular Form

- Horner's Rule

$$P = A \times (-b_{W-1} + \sum_{i=1}^{W-1} b_{W-1-i} 2^{-i})$$

$$= -Ab_{W-1} + (Ab_{W-2} + (Ab_{W-3} + (\ldots(Ab_1 + Ab_0 2^{-1})2^{-1})\ldots)2^{-1})2^{-1}$$

- 4-bit multiplication

$$P = -Ab_3 + \boxed{Ab_2 2^{-1} + Ab_1 2^{-2} + Ab_0 2^{-3}}$$

- Tabular form

$$\begin{array}{ccccccc}
 & & & a_3 & a_2 & a_1 & a_0 \\
 & & & b_3 & b_2 & b_1 & b_0 \\
\hline
 & & & -a_3b_0 & a_2b_0 & a_1b_0 & a_0b_0 \\
 & & -a_3b_1 & a_2b_1 & a_1b_1 & a_0b_1 & \\
 & -a_3b_2 & a_2b_2 & a_1b_2 & a_0b_2 & & \\
-\overline{a}_3b_3 & \overline{a}_2b_3 & \overline{a}_1b_3 & \overline{a}_0b_3 & & & \\
 & & & \mathbf{b_3} & & & \\
\hline
p_6 & p_5 & p_4 & p_3 & p_2 & p_1 & p_0
\end{array}$$

Tabular Form

- -A: In the 2’s complement system, negating a number is taking its one’s complement ($1-a_i$, or inversion) and then adding a 1 to the LSB.

$$-A = -\bar{a}_{W-1} + \Big(\sum_{i=1}^{W-1} \bar{a}_{W-1-i}\, 2^{-i}\Big) + 2^{-(W-1)}$$

- 4-bit multiplication

$$P = \boxed{-Ab_3} + Ab_2 2^{-1} + Ab_1 2^{-2} + Ab_0 2^{-3}$$

- Tabular form

			a_3	a_2	a_1	a_0
			b_3	b_2	b_1	b_0
			$-a_3b_0$	a_2b_0	a_1b_0	a_0b_0
		$-a_3b_1$	a_2b_1	a_1b_1	a_0b_1	
	$-a_3b_2$	a_2b_2	a_1b_2	a_0b_2		
$-\bar{a}_3b_3$	$\bar{a}_2b_3$	$\bar{a}_1b_3$	$\bar{a}_0b_3$			
			$\mathbf{b_3}$			
p_6	p_5	p_4	p_3	p_2	p_1	p_0

Tabular Form

			a_3	a_2	a_1	a_0
			b_3	b_2	b_1	b_0
			$-a_3b_0$	a_2b_0	a_1b_0	a_0b_0
		$-a_3b_1$	a_2b_1	a_1b_1	a_0b_1	
	$-a_3b_2$	a_2b_2	a_1b_2	a_0b_2		
$-\overline{a_3}b_3$	$\overline{a_2}b_3$	$\overline{a_1}b_3$	$\overline{a_0}b_3$			
			$\mathbf{b_3}$			
p_6	p_5	p_4	p_3	p_2	p_1	p_0

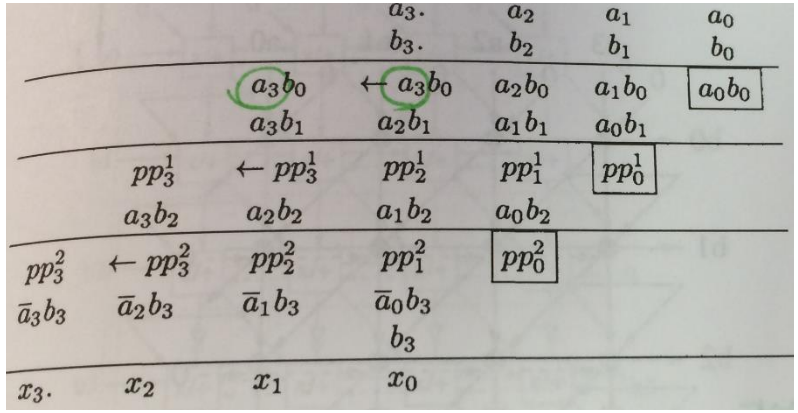
- Direct addition cannot be performed due to the negative weight
- Sign extension (extend the sign bit)

$$A = -a_3 + a_2 2^{-1} + a_1 2^{-2} + a_0 2^{-3}$$

$$= -a_3 2 + a_3 + a_2 2^{-1} + a_1 2^{-2} + a_0 2^{-3}$$ **(by 1 bit)**

$$= -a_3 2^2 + a_3 2 + a_3 + a_2 2^{-1} + a_1 2^{-2} + a_0 2^{-3}$$ **(by 2 bits)**

Sign Extension and Scaling



- 1-b sign extension
  - If the MSB of the multiplicand is defined as the guard bit and set equal to a_{w-2}, i.e., limit the multiplicand's range [-0.5, 0.5)
  - Otherwise, 2-bit sign extension is necessary (incurring more additions)
- Scaling by 2^{-1}

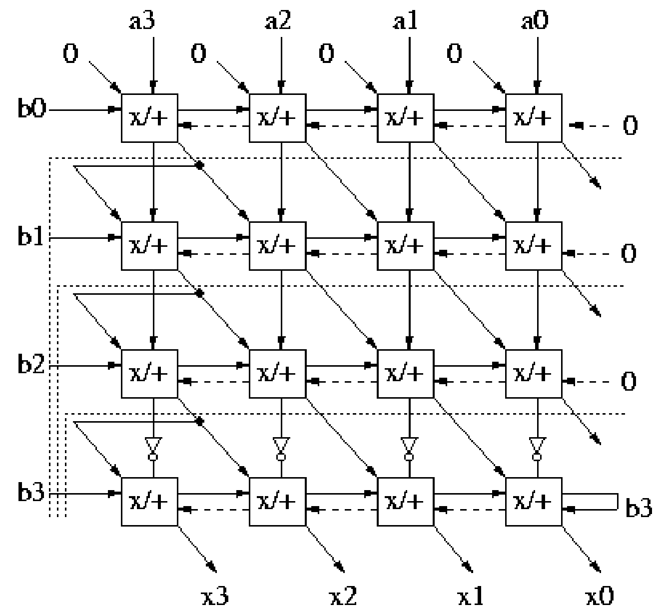
$$A \cdot 2^{-1} = (-a_3 + a_2 2^{-1} + a_1 2^{-2} + a_0 2^{-3})2^{-1}$$
$$= -a_3 2^{-1} + a_2 2^{-2} + a_1 2^{-3} + a_0 2^{-4}$$
$$= -a_3 + a_3 2^{-1} + a_2 2^{-2} + a_1 2^{-3} + a_0 2^{-4}$$

Often truncated

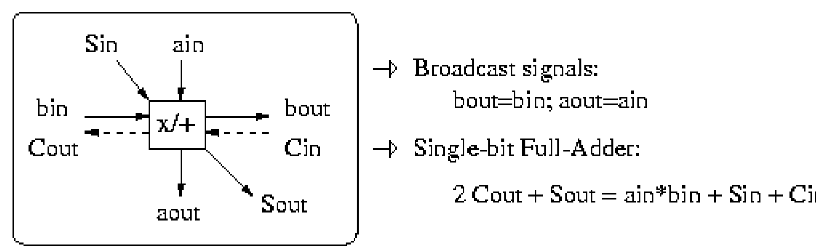
- Then, perform step-by-step accumulation w/ LSBs removed for constant-width multiplication

Parallel Carry Ripple Multiplier

- Critical path scales w/ 2*W*
  - It is a_1b_0 to a_0b_1 to a_3b_1 to a_1b_3 to a_3b_3
  - $1t_{and}+3t_c + 2t_s + 2t_c + 1t_s$
  - Note that a and b are broadcast signals

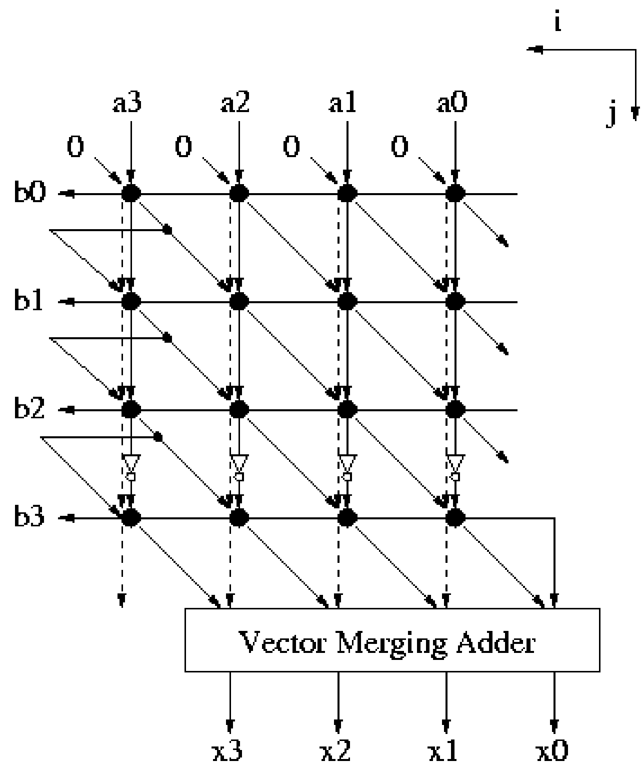


- Registers placed at the *cutsets* to reduce critical path to *W*
  - Still carries are propagated horizontally
  - Critical path = $3t_c+1t_s$



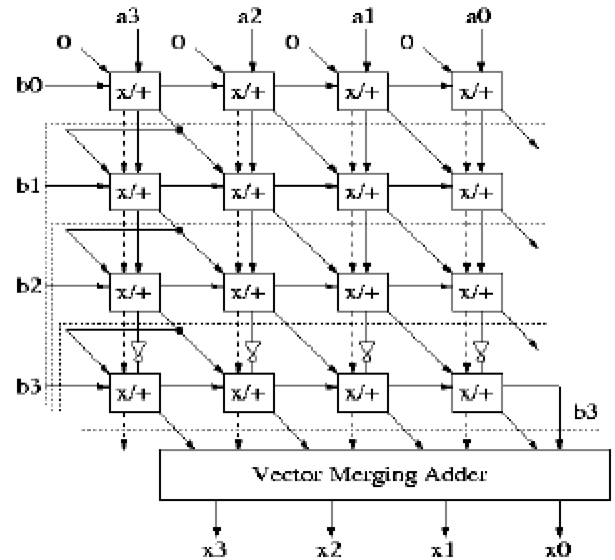
Parallel Carry-Save Array Multiplier

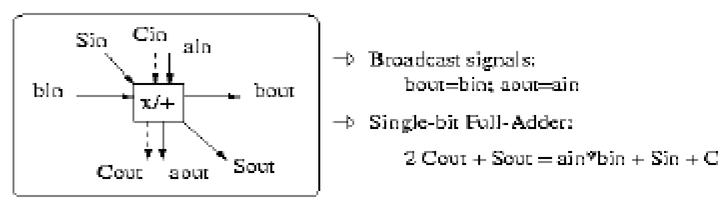
- Carry-save multiplier
  - Do not propagate carry
  - Save carry and add it, with proper alignment, to the next operand
  - *Additions at different bit positions in the same row are now independent of each other and can be carried out in parallel*
  - Carry-save addition can be applied to all but the last step
  - A vector merging adder (VMA) is used in the final step



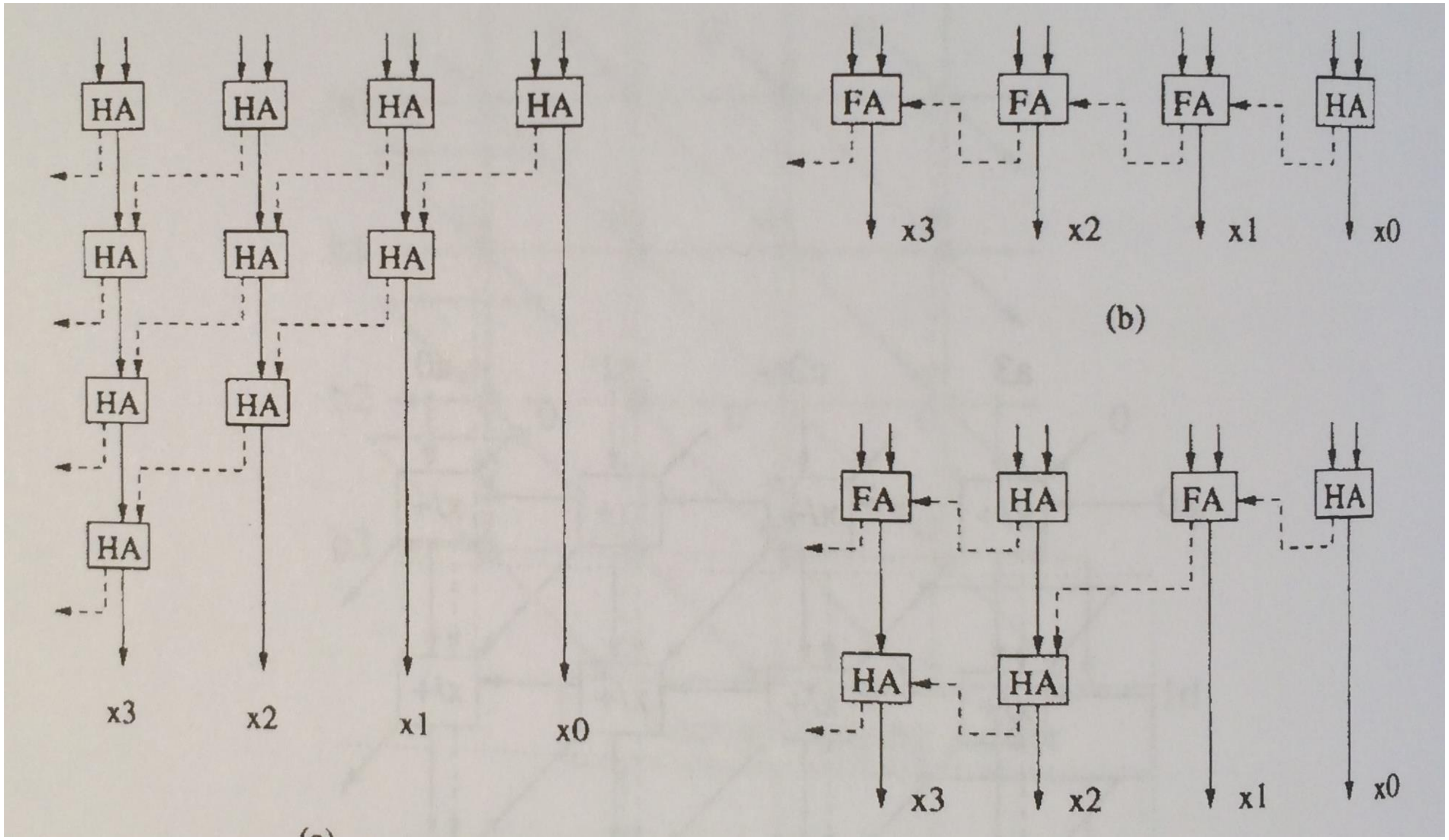
Parallel Carry-Save Array Multiplier

- Critical path: $W + T_{VMA}$
- Registers placed at the feed-forward cutsets to reduce critical path to ***1***
- We need to pipeline the VMA to the same level, which can be done easily





Vector Merging Units



- Vector merging unit choices
- Easy to pipeline

Baugh-Wooley Multiplier

$$A = -a_3 + a_2 2^{-1} + a_1 2^{-2} + a_0 2^{-3}$$

$$B = -b_3 + b_2 2^{-1} + b_1 2^{-2} + b_0 2^{-3}$$

$$\begin{aligned}
A \times B &= (a_3 b_3 + (a_2 2^{-1} + a_1 2^{-2} + a_0 2^{-3})(b_2 2^{-1} + b_1 2^{-2} + b_0 2^{-3})) \\
&-(a_3(b_2 2^{-1} + b_1 2^{-2} + b_0 2^{-3}) + (a_2 2^{-1} + a_1 2^{-2} + a_0 2^{-3}) b_3) \\
&= (a_3 b_3 + (a_2 2^{-1} + a_1 2^{-2} + a_0 2^{-3})(b_2 2^{-1} + b_1 2^{-2} + b_0 2^{-3})) \\
&-(a_3 b_2 + a_2 b_3) 2^{-1} - (a_3 b_1 + a_1 b_3) 2^{-2} - (a_3 b_0 + a_0 b_3) 2^{-3} \\
&= (a_3 b_3 + (a_2 2^{-1} + a_1 2^{-2} + a_0 2^{-3})(b_2 2^{-1} + b_1 2^{-2} + b_0 2^{-3})) \\
&+(1 - a_3 b_2 + 1 - a_2 b_3) 2^{-1} + (1 - a_3 b_1 + 1 - a_1 b_3) 2^{-2} + (1 - a_3 b_0 + 1 - a_0 b_3) 2^{-3} - 1 - 2^{-1} - 2^{-2} \\
&= (a_3 b_3 + (a_2 2^{-1} + a_1 2^{-2} + a_0 2^{-3})(b_2 2^{-1} + b_1 2^{-2} + b_0 2^{-3})) \\
&+(\overline{a_3 b_2} + \overline{a_2 b_3}) 2^{-1} + (\overline{a_3 b_1} + \overline{a_1 b_3} + 1) 2^{-2} + (\overline{a_3 b_0} + \overline{a_0 b_3}) 2^{-3}
\end{aligned}$$

Note that ($-1-2^{-1}-2^{-2}$) is outside the range of the 2's complement numbers [-1,1). It is $-7/4_{(10)}$ and can be represented as $10.010_{(2)} = -2+2^{-2} = -7/4$. So we need to add 1 in the (MSB+1) position and also add the 1 in the (MSB-2) position. In fact, we don't have to add the 1 at the MSB+1 since it is outside the range of 2's complement, and makes no impact on results.

Baugh-Wooley Multiplier

$$
\begin{array}{ccccccc}
 & & & a_3 & a_2 & a_1 & a_0 \\
 & & & b_3 & b_2 & b_1 & b_0 \\
\hline
 & & & \overline{a_3b_0} & a_2b_0 & a_1b_0 & a_0b_0 \\
 & & \overline{a_3b_1} & a_2b_1 & a_1b_1 & a_0b_1 & \\
 & \overline{a_3b_2} & a_2b_2 & a_1b_2 & a_0b_2 & & \\
a_3b_3 & \overline{a_2b_3} & \overline{a_1b_3} & \overline{a_0b_3} & & & \\
\hline
 & & \color{red}{1} & & & & \\
x_3 & x_2 & x_1 & x_0 & & &
\end{array}
$$

- Handles sign bits efficiently
  - Simply adding all the partial products will produce the result
  - Sign-bit extension is replaced with inversion!
  - Carry-save structure can be used to accumulate all the partial products

Booth Recoding

- Multiplication is done through two steps
  - Partial product accumulation: carry-save addition
  - Partial product generation: modified Booth recoding

- Fewer partial products need to be generated for groups of consecutive 0’s and 1’s
  - For a group of *m* consecutive 1’s in the multiplier
  - …0{11…1}0… =
    …1{00…0}0… – ….0{ 00…1}0 =
    …1{00…$\bar{1}$}0
  - Instead of *m* partial products, only 2 partial products are generated and signed-digit representation is used

- Multiplier bits are first recoded into signed-digit representation with fewer number of nonzero digits

Booth Recoding: Example

Sign extension

				0	0	1	1	0	6x
				0	1	1	1	0	14
				+1	0	0	-1	0	
				0	0	0	0	0	
1	1	1	1	1	0	1	0		(-6)
		0	0	0	0	0			
	0	0	0	0	0				
0	0	1	1	0					
0	0	1	0	1	0	1	0	0	84

Advantages and Disadvantages

- Advantage
  - It might reduce the # of 1's in multiplier
  - Good for constant multiplication
- Disadvantage
  - It doesn't save in speed
    (still have to wait for the critical path, e.g., the shift-add delay in sequential multiplier)
  - Increases area: recoding circuitry AND subtraction

Modified Booth

- Modified Booth: can produce *at most* n/2+1 partial products.
- **Algorithm: (for unsigned numbers)**
  1. Pad the LSB with one zero.
  2. Pad the MSB with 2 zeros if n is even and 1 zero if n is odd.
  3. Divide the multiplier into overlapping groups of 3-bits.
  4. Determine partial product scale factor from modified booth encoding table.
  5. Compute the Multiplicand Multiples
  6. Sum Partial Products

Modified Booth

- **Example: (n=4-bits unsigned)**

1. **Pad LSB with 1 zero**

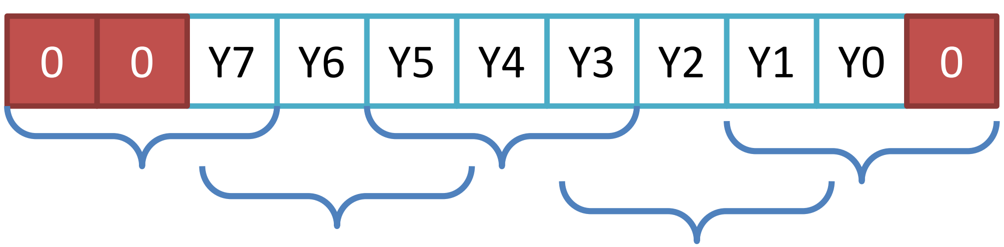
Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0	0

2. **If n is even, pad the MSB with two zeros**

0	0	Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0	0

3. **Form 3-bit overlapping groups for n=8 we have 5 groups**

0	0	0	0	0	1	0	1	0	0	0



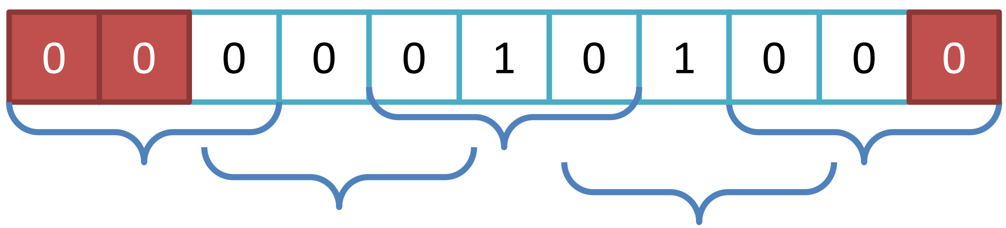
Modified Booth

- Record 3 bits using Booth recoding table
  - Add multiplicand times -2, -1, 0, 1, and 2
  - No 3, making generating partial product is easy

i+1	i	i-1	add	Explanation
0	0	0	0*M	No string of 1's in sight
0	0	1	1*M	End of a string of 1's
0	1	0	1*M	Isolated 1
0	1	1	2*M	End of a string of 1's
1	0	0	–2*M	Beginning of a string of 1's
1	0	1	–1*M	End one string, begin new one
1	1	0	–1*M	Beginning of a string of 1's
1	1	1	0*M	Continuation of string of 1's

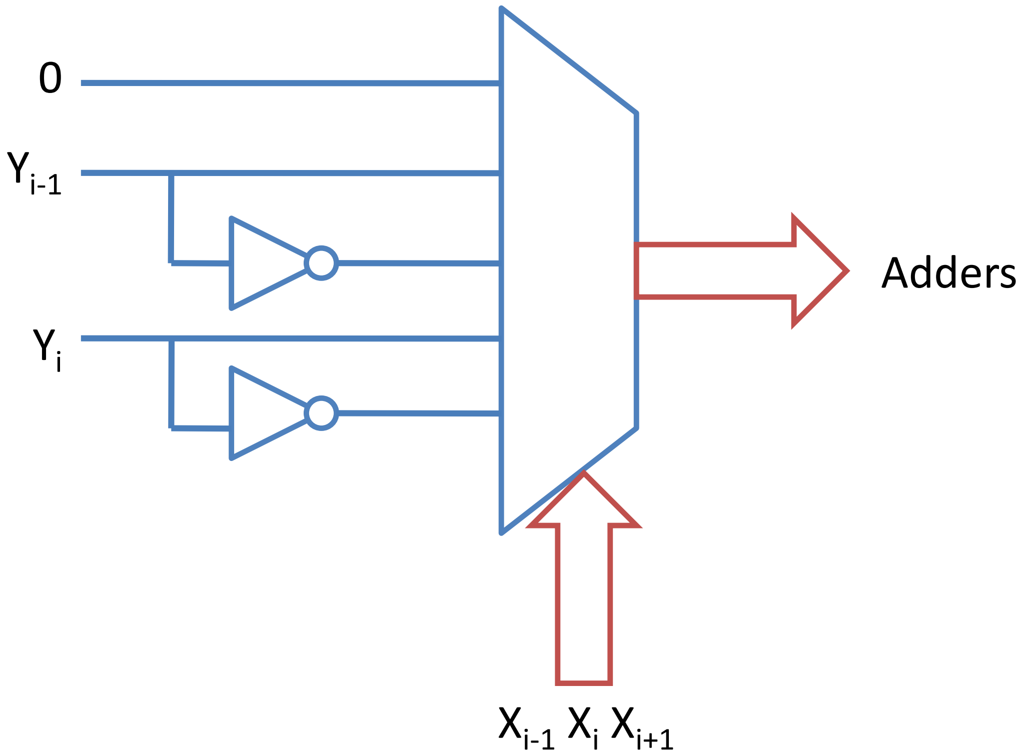
Modified Booth

4. Determine partial product scale factor from modified booth 2 encoding table.



Groups			Coding
0	0	0	0 × Y
0	1	0	1 × Y
0	1	0	1 × Y
0	0	0	0 × Y
0	0	0	0 × Y

Compute Partial Products



X_{i+1}	X_i	X_{i-1}	Action
0	0	0	$0 \times Y$
0	0	1	$1 \times Y$
0	1	0	$1 \times Y$
0	1	1	$2 \times Y$
1	0	0	$-2 \times Y$
1	0	1	$-1 \times Y$
1	1	0	$-1 \times Y$
1	1	1	$0 \times Y$

Modified Booth

5. Compute the Multiplicand Multiples

Groups			Coding
0	0	0	$0 \times Y$
0	1	0	$1 \times Y$
0	1	0	$1 \times Y$
0	0	0	$0 \times Y$
0	0	0	$0 \times Y$

							0	0	0	0	0	1	0	0	0	1
						×	0	0	0	0	1	0	1	0	0	20
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	$0 \times Y$
0	0	0	0	0	0	0	0	0	0	1	0	0	0			$1 \times Y$
0	0	0	0	0	0	0	0	1	0	0	0					$1 \times Y$
0	0	0	0	0	0	0	0	0	0							$0 \times Y$
0	0	0	0	0	0	0	0									$0 \times Y$

Modified Booth

6. Sum Partial Products

```
                  0 0 0 0 0 1 0 0 0     1
                × 0 0 0 0 1 0 1 0 0    20
  -------------------------------------
    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0   0 × Y
    0 0 0 0 0 0 0 0 0 0 1 0 0 0       1 × Y
  + 0 0 0 0 0 0 0 0 1 0 0 0           1 × Y
    0 0 0 0 0 0 0 0 0 0               0 × Y
    0 0 0 0 0 0 0 0                   0 × Y
  -------------------------------------
    0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0   160
```

Modified Booth Recoding: Summary

- Grouping multiplier bits into pairs
  - Reduces the num of partial products to half
  - If Booth recoding not used ➔ have to be able to multiply by 3 (hard: shift+add)
- Uses high-radix to reduce number of intermediate addition operands
  - Can go higher: radix-8, radix-16
  - Radix-8 should implement *3, *-3, *4, *-4
  - Recoding and partial product generation becomes more complex