Number Systems – Integers

Unsigned

0 to 255

1	0	1	1	1	0	1	0	= 186
128	64	32	16	8	4	2	1	

Sign Magnitude

-127 to +127 (+/- zero)

1	0	1	1	1	0	1	0	= -58
+/-	64	32	16	8	4	2	1	

'1's complement

-127 to +127 (+/- zero)

1	0	1	1	1	0	1	0	= -69
-127	64	32	16	8	4	2	1	

'2's complement

-128 to +127

1	0	1	1	1	0	1	0	= -70
-128	64	32	16	8	4	2	1	

Biased (bias =128)

-128 to +127

1	0	1	1	1	0	1	0	-128 = 58
128	64	32	16	8	4	2	1	

Arithmetic – Integer Addition

										Unsigned	'2's complement
	1	0	1	1	1	0	1	0	+	186 +	-70 +
	0	0	0	1	1	1	0	0		28	+28
0	0	1	1	1	0	0	0				
	1	1	0	1	0	1	1	0		214	-42

- Simple Binary Arithmetic
  - works for unsigned and '2's complement[1] integers
  - doesn't work for other integer number systems
  - most integer systems support unsigned and '2's complement only

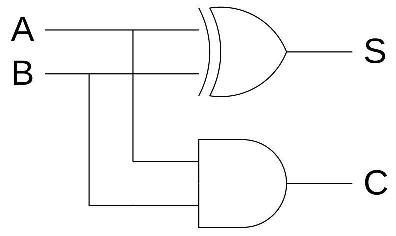
[1] addition of two negative numbers will generate a *carry out* which must be ignored — the result has the same number of bits as the operands

Half Adder & Incrementer

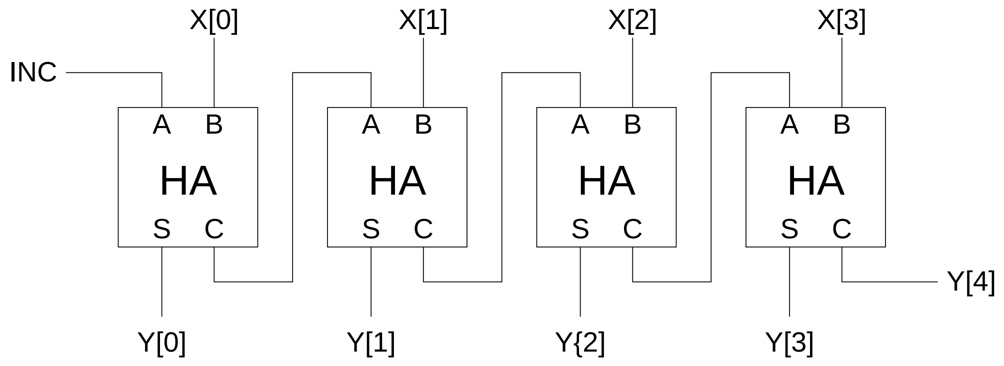
- Half Adder

$$S = A \oplus B$$

$$C = A\ B$$



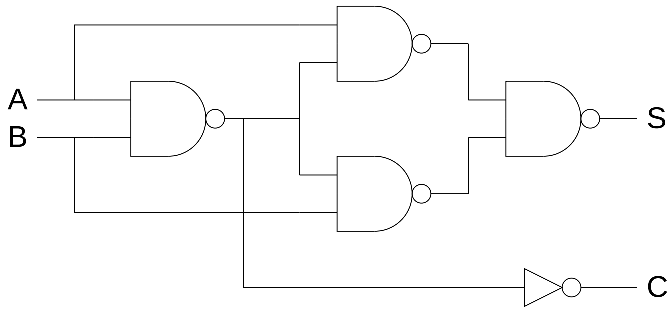
- Incrementer Unit



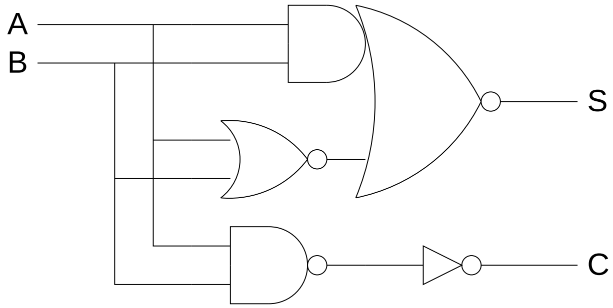
Y[ ] = X[ ] + INC

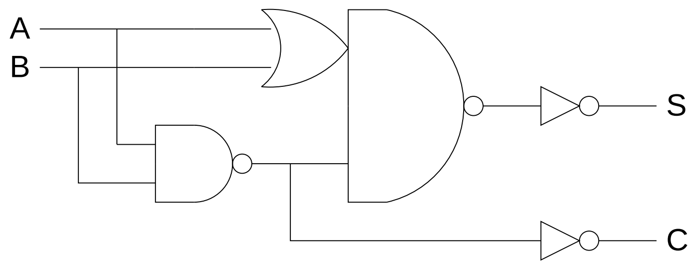
Half Adders

- Implementation
  - the simplest NAND based implementation re-uses $\overline{Cout}$ in the calculation of S.



  - more advanced implementations may be technology dependent; the following use CMOS compound gates:



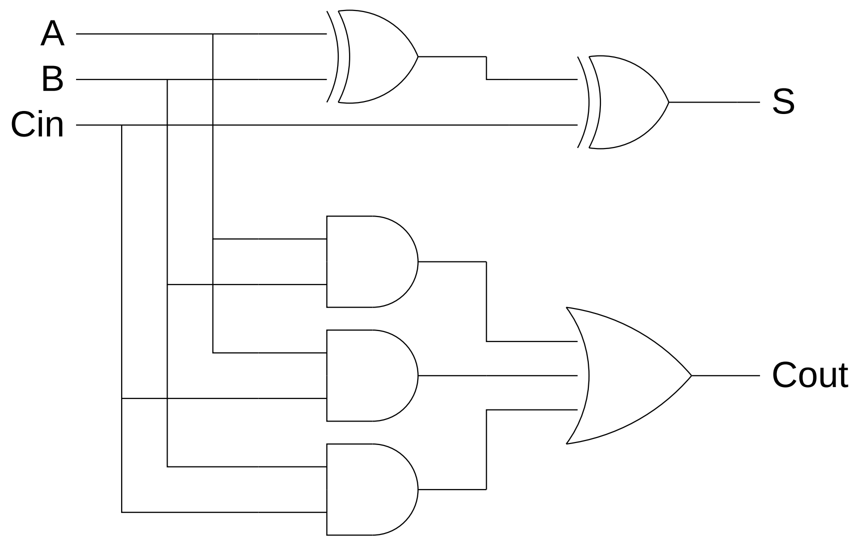


Full Adder

- Full Adder

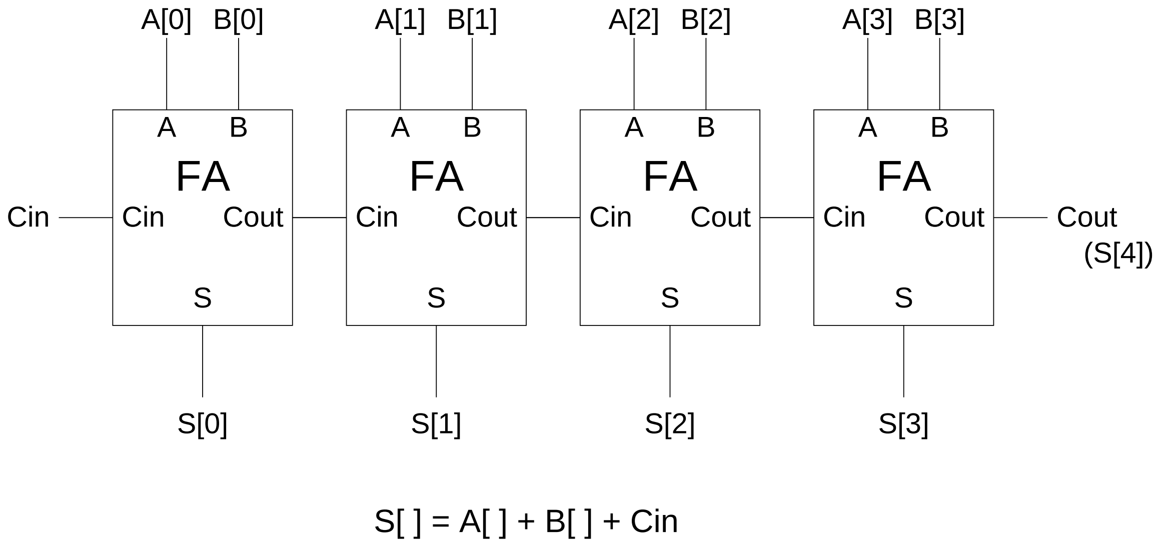
$S = A \oplus B \oplus Cin$

Cout = A.B + A.Cin + B.Cin



Multi-bit Adders

- Ripple Carry Adder

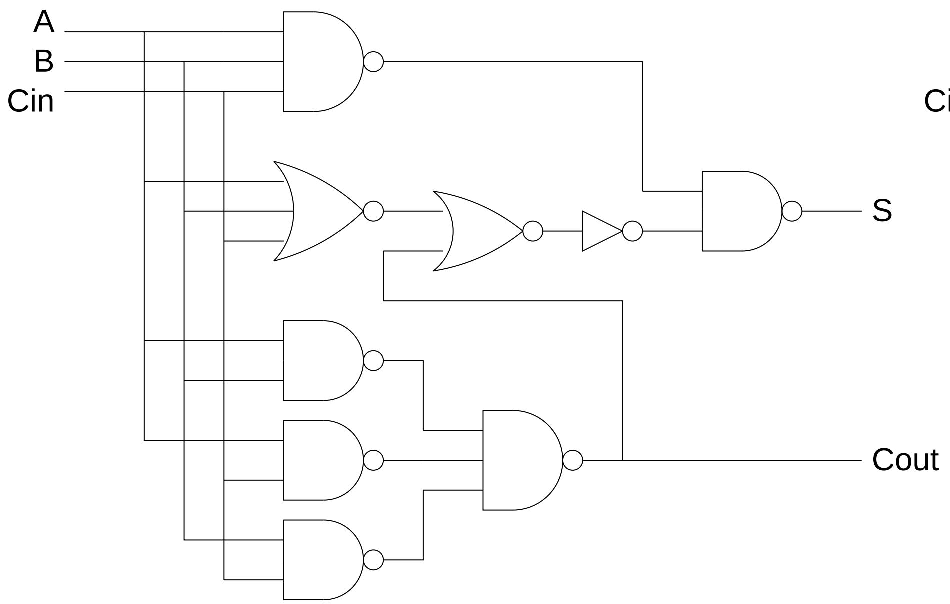


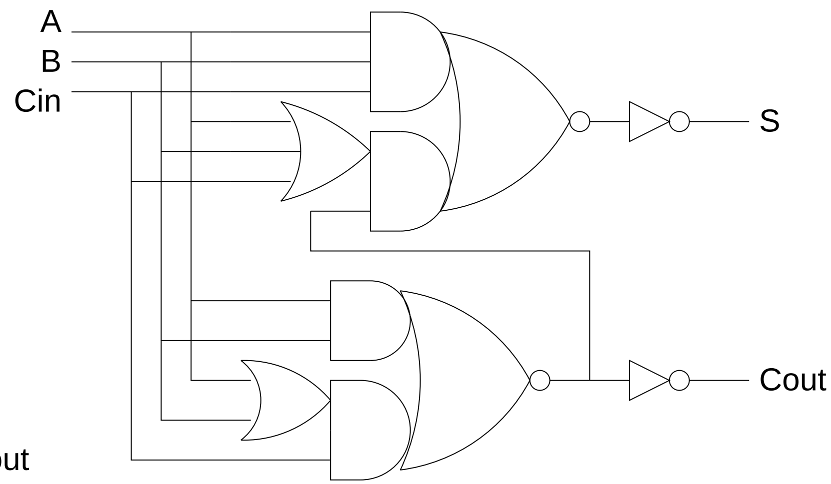
  – note that the delay of the complete adder is primarily determined by the addition of delays in the carry path[2].

[2]calculation of S[3] may also be in the critical path since Cout is likely to become available before S[3]

Full Adder

- Implementation
  - various implementations exist which use a minimum number of gates/transistors dependent on technology:





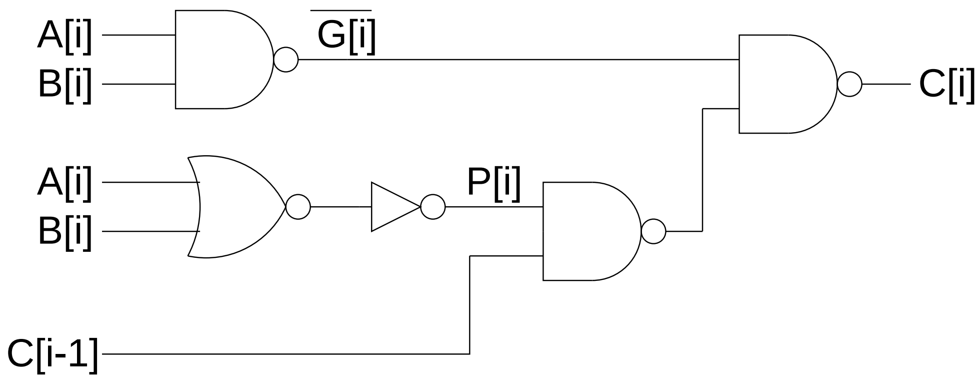
Multi-bit Adders

- Fast Ripple Carry Adder
  - in order to optimize performance it is necessary to minimize the carry propagation delay. Generate, $G[i]$, and propagate, $P[i]$, signals are pre-calculated allowing fast response to changes on carry in, $C[i-1]$.

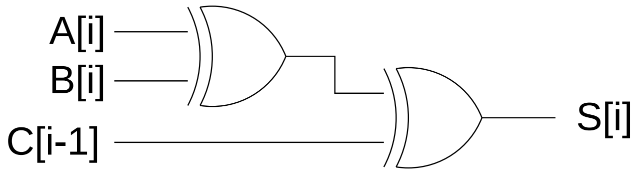
G[i] = A[i].B[i]

P[i] = A[i] + B[i]

C[i] = G[i] + P[i].C[i-1]



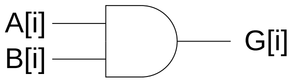
S[i] = A[i] $\oplus$ B[i] $\oplus$ C[i-1]



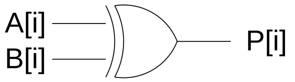
Multi-bit Adders

- Manchester Carry Adder
  - the Manchester carry system uses a different definition of propagate, this propagate signal is used in the calculation of the sum output.

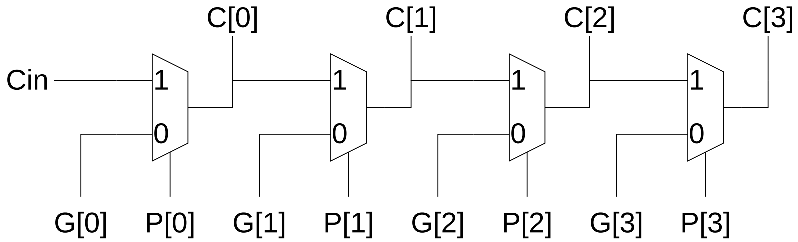
$$G[i] = A[i]\;B[i]$$



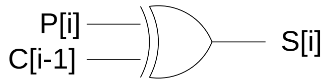
$$P[i] = A[i] \oplus B[i]$$



$$C[i] = C[i-1]\;P[i] + G[i]\;\overline{P[i]}$$

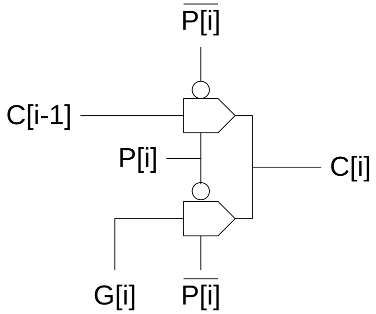


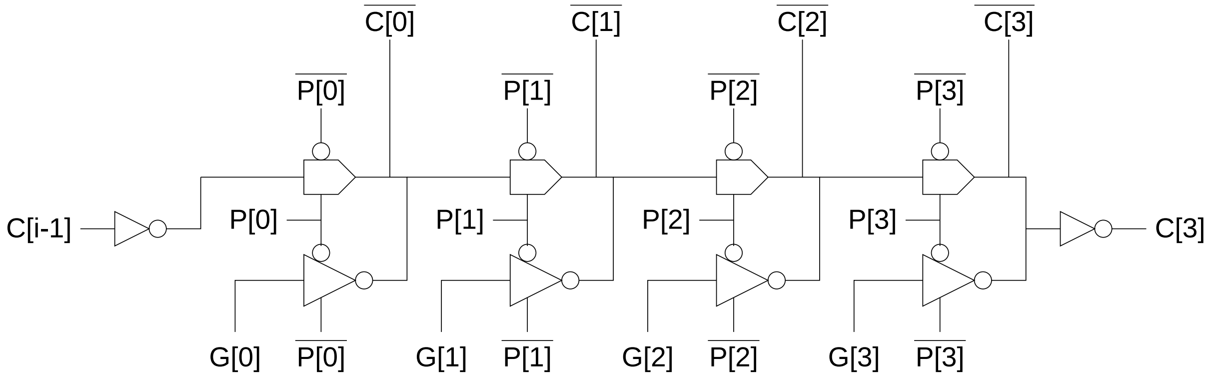
$$S[i] = P[i] \oplus C[i-1]$$



Multi-bit Adders

- Manchester Carry Adder
  - various implementations exist making use of fast multiplexors based around pass transistors or transmission gates.
  - a long run of these gates requires buffering to maintain performance.

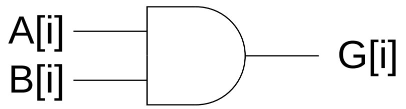




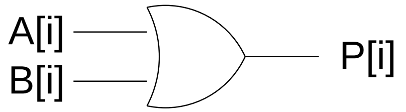
Adders

- Carry Lookahead Adder

$G[i] = A[i].B[i]$

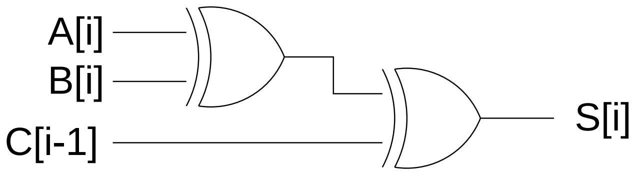


$P[i] = A[i] + B[i]$



$C[i] = G[i] + P[i].C[i-1]$

$S[i] = A[i] \oplus B[i] \oplus C[i-1]$



  – uses generate and propagate as before – but expands recursive carry expression

Adders

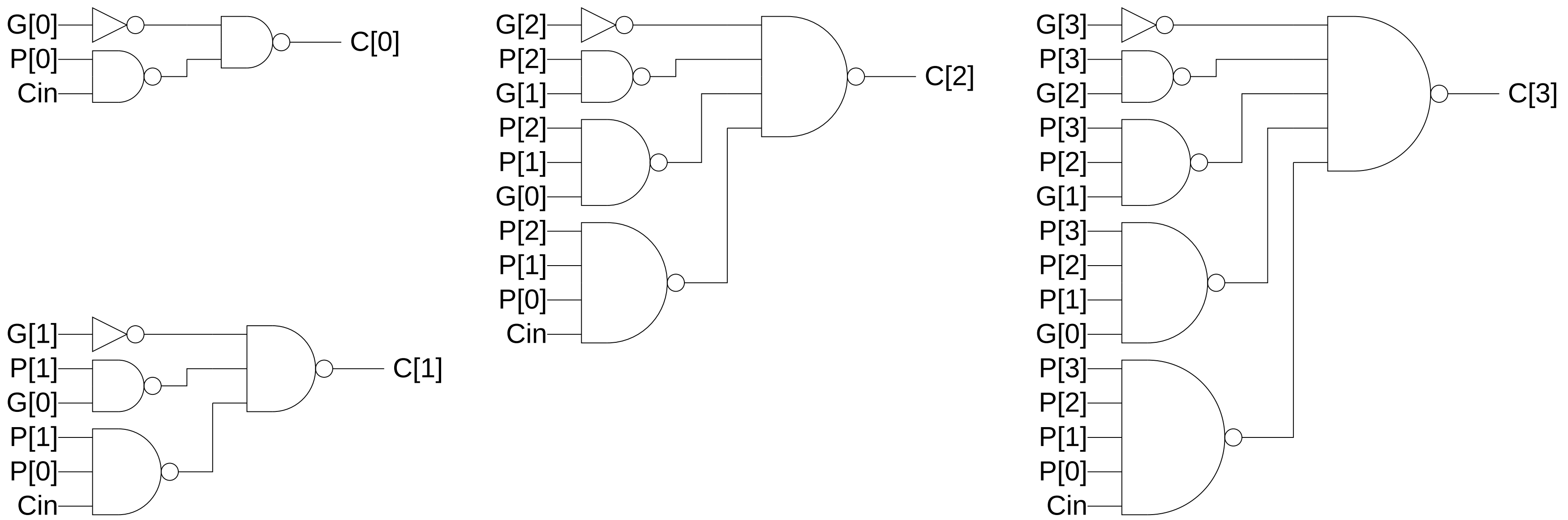
- Carry Lookahead Adder
  - using 2 stage NAND NAND logic

C[0] = G[0] + P[0].Cin

C[1] = G[1] + P[1].G[0] + P[1].P[0].Cin

C[2] = G[2] + P[2].G[1] + P[2].P[1].G[0] + P[2].P[1].P[0].Cin

C[3] = G[3] + P[3].G[2] + P[3].P[2].G[1] + P[3].P[2].P[1].G[0] + P[3].P[2].P[1].P[0].Cin



Adders

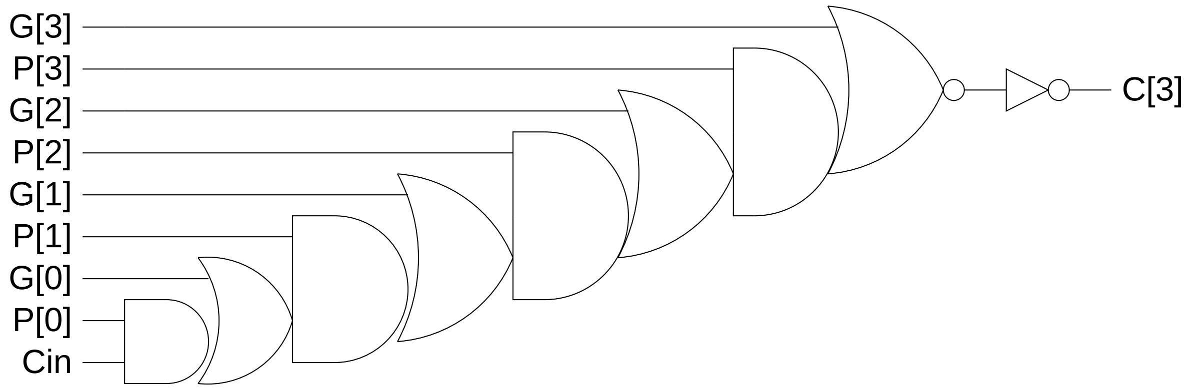
- Carry Lookahead Adder
  - using compound CMOS gates
    - - max. 5 transistors in series in compound gate!

C[0] = G[0] + P[0].Cin

C[1] = G[1] + P[1].(G[0] + P[0].Cin)

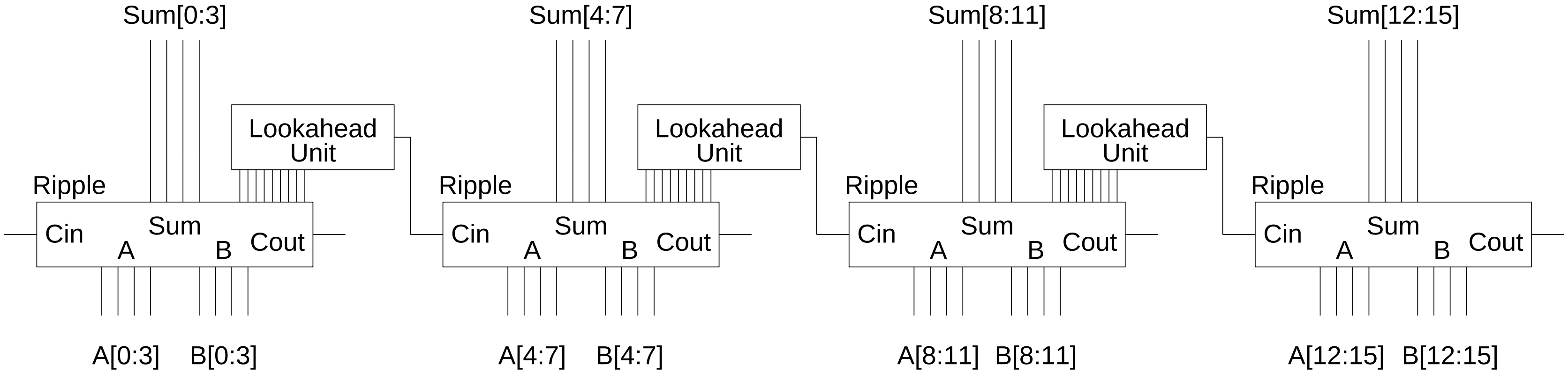
C[2] = G[2] + P[2].(G[1] + P[1].(G[0] + P[0].Cin))

C[3] = G[3] + P[3].(G[2] + P[2].(G[1] + P[1].(G[0] + P[0].Cin)))



Hybrid Adders

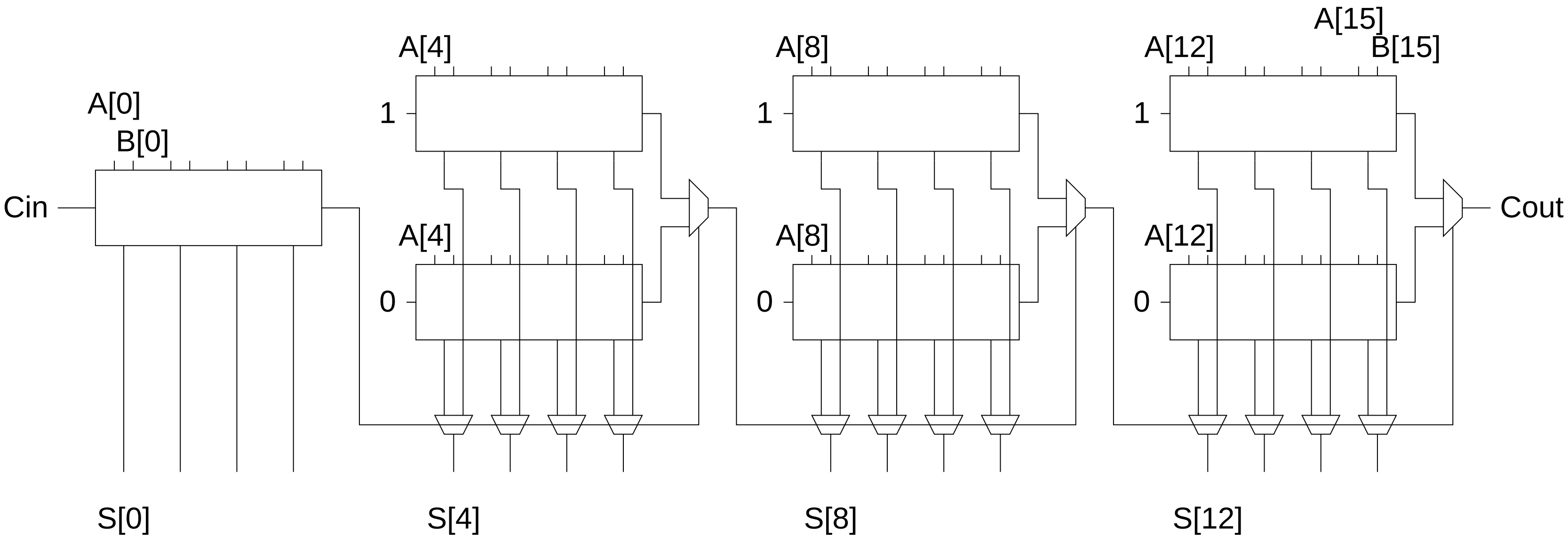
- Carry Lookahead Adder
  - multi-bit carry lookahead is limited by large fanin and fanout requirements
  - carry lookahead is frequently used to accelerate ripple carry adders



  - 4 bit lookahead unit shares propagate and generate with 4 bit ripple carry adder
  - lookahead is used to calculate every fourth carry

Hybrid Adders

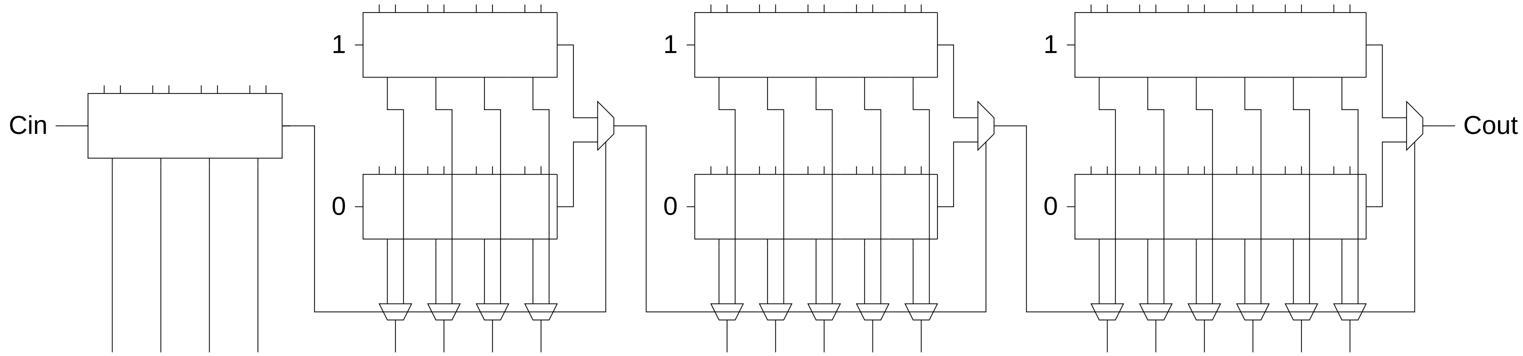
- Carry Select Adder
  - carry select offers another technique for carry acceleration



  - all possible values are calculated in the time taken for a 4 bit adder, carry signals are then used to select correct results
  - pairs of 4 bit adders act as macro carry propagate and carry generate units

Hybrid Adders

- Carry Select Adder
  - any adder may be used as the building block for a carry select adder – overall delay is only partially proportional to adder delay.
  - variable length adders can be arranged such that the carry in signal is valid at the same time as the results for selection, giving a better overall performance.



Arithmetic Overflow

- Unsigned Addition

										Unsigned
	1	0	1	1	0	0	1	0		178
									+	+
	1	1	0	0	0	1	1	0		198
1	0	0	0	0	1	1	0			Overflow!
	0	1	1	1	1	0	0	0		120

  – the carry out indicates an overflow

Arithmetic Overflow

- '2's Complement Addition

										'2's complement
	1	1	1	1	0	0	1	0		-14
									+	+
	1	1	0	0	0	1	1	0		-58
1	1	0	0	0	1	1	0			
	1	0	1	1	1	0	0	0		-72

  – the carry out is ignored – no overflow here!

Arithmetic Overflow

- '2's Complement Addition

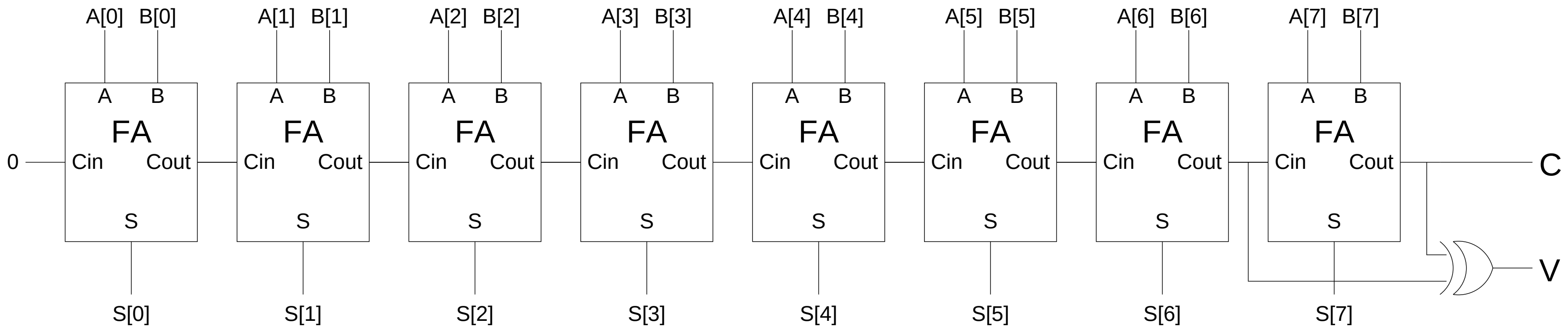
										'2's complement
	1	0	1	1	0	0	1	0	+	-78 +
	1	1	0	0	0	1	1	0		-58
1	0	0	0	0	1	1	0			Overflow!
	0	1	1	1	1	0	0	0		120

										'2's complement
	0	1	1	1	0	0	1	0	+	114 +
	0	1	0	0	0	1	1	0		70
0	1	0	0	0	1	1	0			Overflow!
	1	0	1	1	1	0	0	0		-72

  - when overflow occurs the sign of the result is different from the sign of either operand
  - we can detect '2's complement overflow as $V = C[8] \oplus C[7]$

Arithmetic Overflow

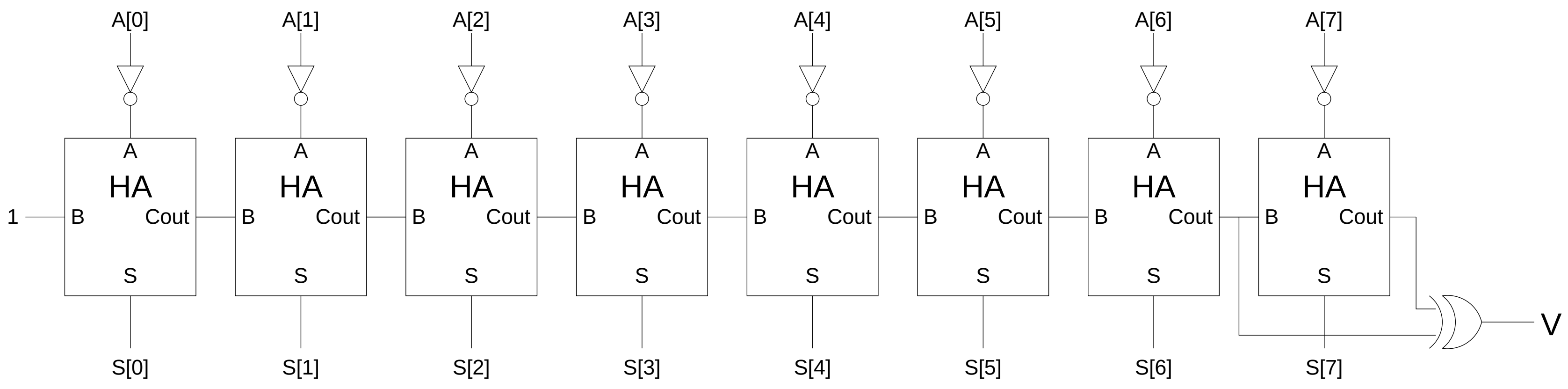
- Dual purpose overflow detecting adder



- Use the same adder for unsigned and '2's complement addition
  - C indicates an overflow for unsigned addition
  - V indicates an overflow for '2's complement addition

'2's Complement Arithmetic

- '2's Complement Negation



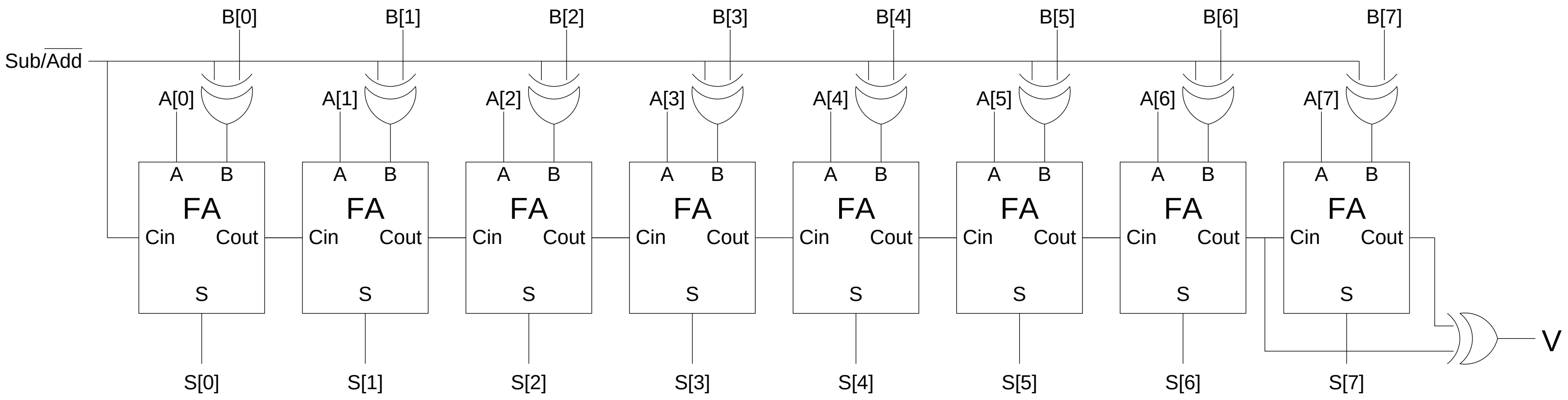
$$S[\,] = -A[\,] = \overline{A[\,]} + 1$$

– overflow occurs for -(-128) since the number system cannot represent +128

- - overflow detection uses carry monitoring, as for addition

‘2’s Complement Arithmetic

- ‘2’s Complement Adder/Subtractor



  – Merging the addition and negation units gives a single adder/subtractor unit
    - - $Sub/\overline{Add}$ signal is used to select between subtraction and addition

Unsigned Arithmetic

- Unsigned Subtraction

										Unsigned
	1	0	1	1	0	0	1	0	-	178 -
	1	0	0	0	0	1	1	0		134
0	0	0	0	1	1	0	0			
	0	0	1	0	1	1	0	0		44

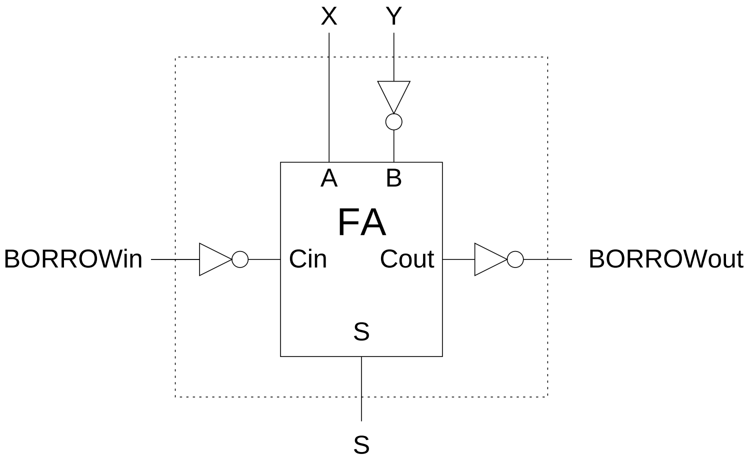
										Unsigned
	1	0	1	1	0	0	1	0	-	178 -
	1	1	0	0	0	1	1	0		198
1	1	0	0	1	1	0	0			Overflow!
	1	1	1	0	1	1	0	0		236

  – the final borrow indicates an overflow since we cannot represent negative numbers

Unsigned Arithmetic

- Full Subtractor

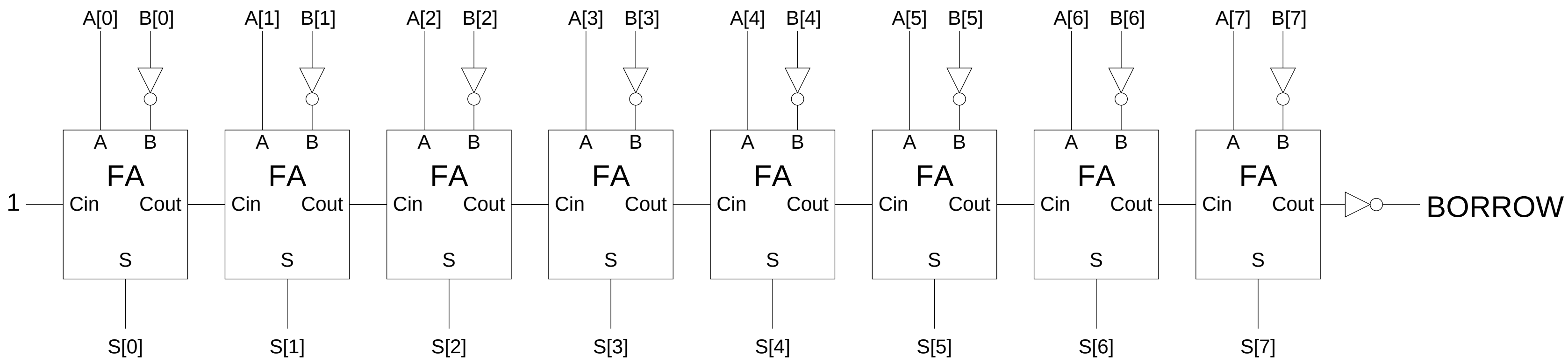
X	Y	Bin	S	Bout
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1



  – a full subtractor unit may be constructed from an existing full adder unit

Multi-bit Subtractor

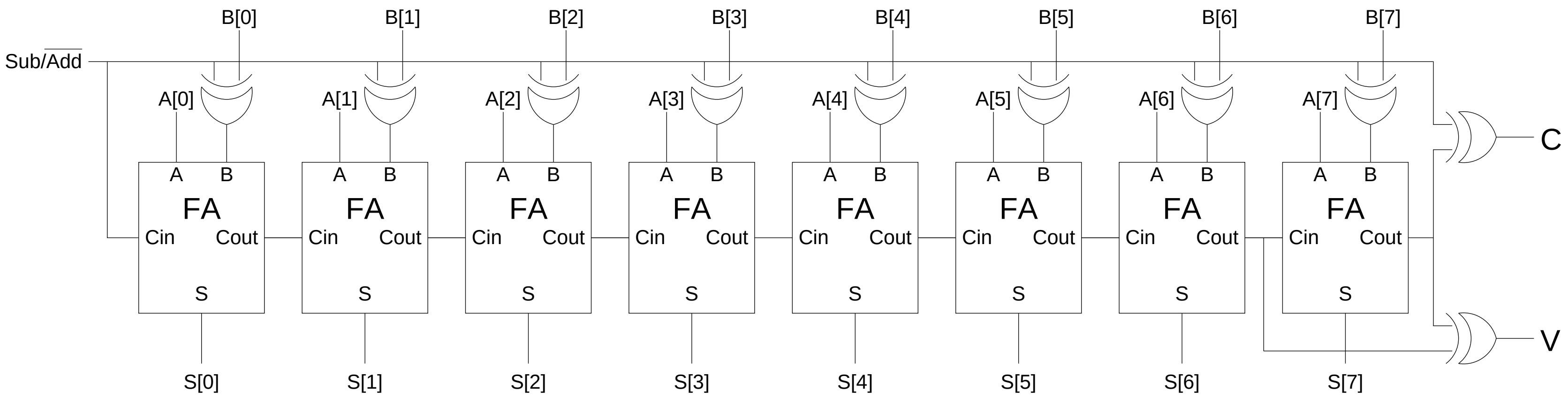
- Multi-bit Subtractor for Unsigned Numbers



  – in a multi-bit subtractor we need not invert the carry signals between full adders

Multi-Purpose Adder/Subtractor

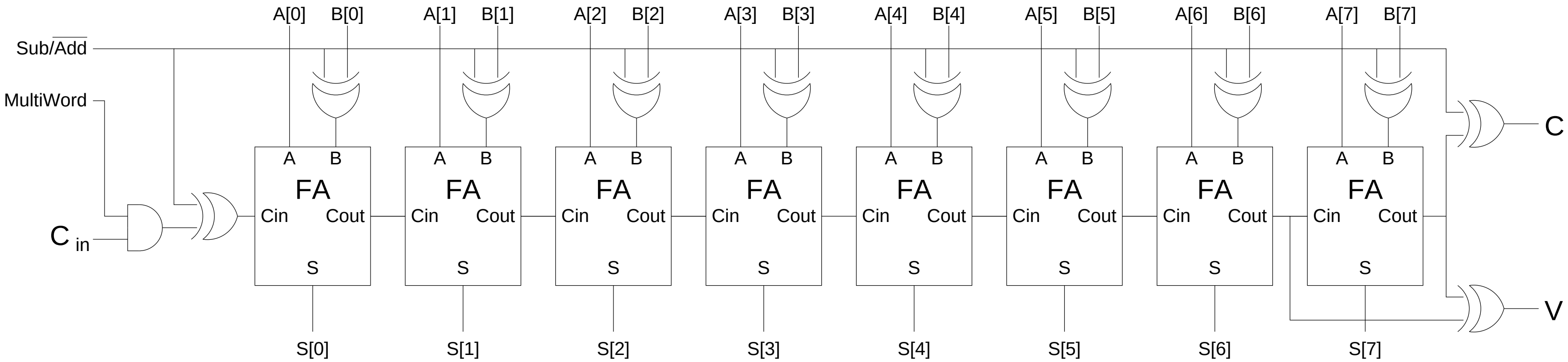
- Overflow detecting adder/subtractor for unsigned and '2's complement integers



- Overflow:
  - C indicates an overflow for unsigned arithmetic
  - V indicates an overflow for '2's complement arithmetic

ALU adder unit

- ALU integer adder unit.



  – most ALUs offer the capability of multi-word addition{subtraction} the carry{borrow} out signal is stored after each operation such that it may fed back into the next operation as carry{borrow} in.

    - - $Sub/\overline{Add}$ signal controls operation as before
    - - $MultiWord$ signal enables C_{in} for all but the first, least significant, word of a multi-word operation

Number Systems – Reals

Fixed point

0 to 31.875

1	0	1	1	1	•	0	1	0	$= {}^{186}/_{8}$
16	8	4	2	1		1/2	1/4	1/8	

Floating Point

0 to 3968

1	0	1	1	1		0	1	0	$= 23 \text{ x}2^{2}$
16	8	4	2	1	x2	4	2	1	

- Fixed Point
  - arithmetic as easy as integer arithmetic
  - limited application due to small range (even with 32 bits)
- Floating Point
  - arithmetic difficult
  - potentially very large range
  - many different format possibilities

Floating Point Formats

- Format choices

1	1	0	1	1		0	1	0	$= -11 \times 2^{+2}$
+/-	8	4	2	1	x2	+/-	2	1	

Range: -120 ... +120

1	1	0	1	1		0	1	0	$= -5 \times 2^{+2}$
-16	8	4	2	1	x2	-4	2	1	

Range: -128 ... +120

1	1	0	1	1		0	1	0	$= -11 \times 4^{+2}$
+/-	8	4	2	1	x4	+/-	2	1	

Range: -960 ... +960

1	1	0	1		1	0	1	0	$= -5 \times 2^{-2}$
+/-	4	2	1	x2	+/-	4	2	1	

Range: -896 ... +896

  - Radix
    - - 2, 2^n, 10
  - Sign formats for mantissa and exponent
    - - Sign magnitude, '2's complement, '1's complement, biased
  - Bits per field
    - - Range *vs* accuracy

Floating Point Formats

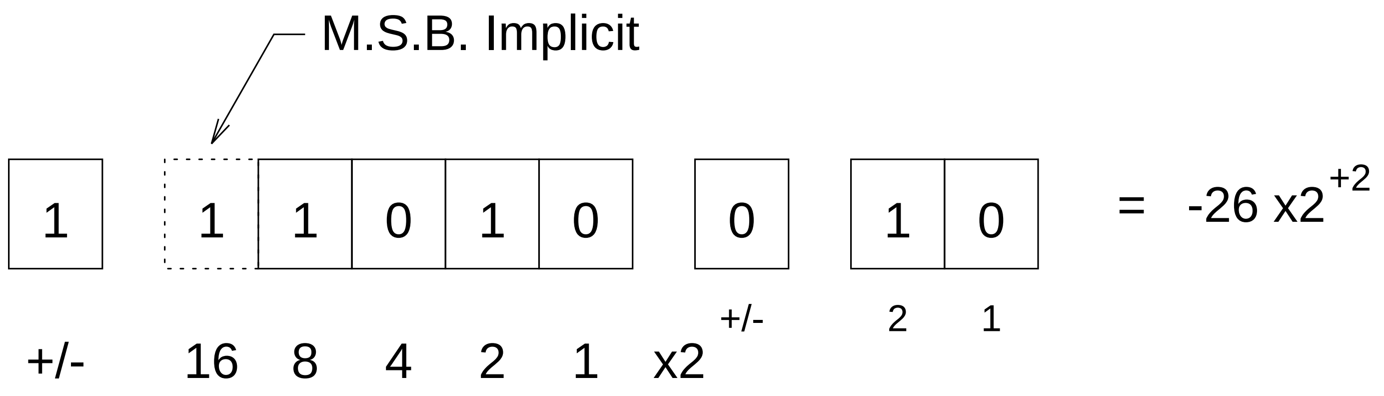
- Multiple representations for the same number

1	1	0	0	0	0	0	0	=	$-8 \text{ x}2^{+0}$ =	-8
+/-	8	4	2	1	$\text{x}2^{+/-}$	2	1			
1	0	1	0	0	0	0	1	=	$-4 \text{ x}2^{+1}$ =	-8
+/-	8	4	2	1	$\text{x}2^{+/-}$	2	1			
1	0	0	0	1	0	1	1	=	$-1 \text{ x}2^{+3}$ =	-8
+/-	8	4	2	1	$\text{x}2^{+/-}$	2	1			

  - wastes almost 1 bit of information

Floating Point Formats

- Normalized numbers



– arrange for most significant bit to be '1'

– since m.s.b. is almost always '1' it need not be stored

– zero must be treated as an exception

IEEE 754 Standard

- Single precision

31	30 – 23	22 – 0
1	1 1 0 1 0 0 0 1	1 0 1 0 1 0 0 0 1 0 1 1 1 1 0 1 0 0 0 1 0 0 1
Sign	Biased Exponent Field	Fraction Field

$$- \; 1 \bullet 10101000101111010001001_2 \times 2^{11010001_2 - 127}$$

  - 32 bits; Sign (1-bit), Biased Exponent (8-bit), Fraction (23-bit).
  - Radix = 2
  - Normalized, giving 24 bits of precision
  - Implicit m.s.b. and binary point
  - $\{BiasedExponent = Fraction = 0\}$ represents ZERO
  - Sign magnitude (hence +/- zero)
  - Biased exponent field; bias = 127

IEEE 754 Standard – Exceptions

- Zero {+/-} & denormals

Sign (31)	Biased Exponent Field (30–23)	Fraction Field (22–0)
0	00000000	00001000101111010001001

$$+\ 0 \bullet 00001000101111010001001_2 \times 2^{00000000_2 + 1 - 127}$$

  - zero is a special case of a denormal number; i.e. a number that is too small to be normalized.
  - denormal numbers have an implicit leading zero.
  - a biased exponent of zero indicates a denormal number.
  - the actual exponent is '1' minus the exponent bias!

IEEE 754 Standard – Exceptions

- Infinity {+/-}
  - infinity may be taken to indicate any number too large to be represented.
    c.f. zero represents all numbers too small to be represented – hence the usefulness of +/-zero
  - infinity is indicated by a zero fraction field and a maximal biased exponent field.

31	30 23	22 0
0/1	1 1 1 1 1 1 1 1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Sign	Biased Exponent Field	Fraction Field

- NaN *Not a Number*

  Any number represented by a maximal biased exponent field and a non zero fraction field is a *Not a Number*, such as may result from $\sqrt{-1}$ or $0 \div 0$.

Arithmetic with Normalized Numbers

Before

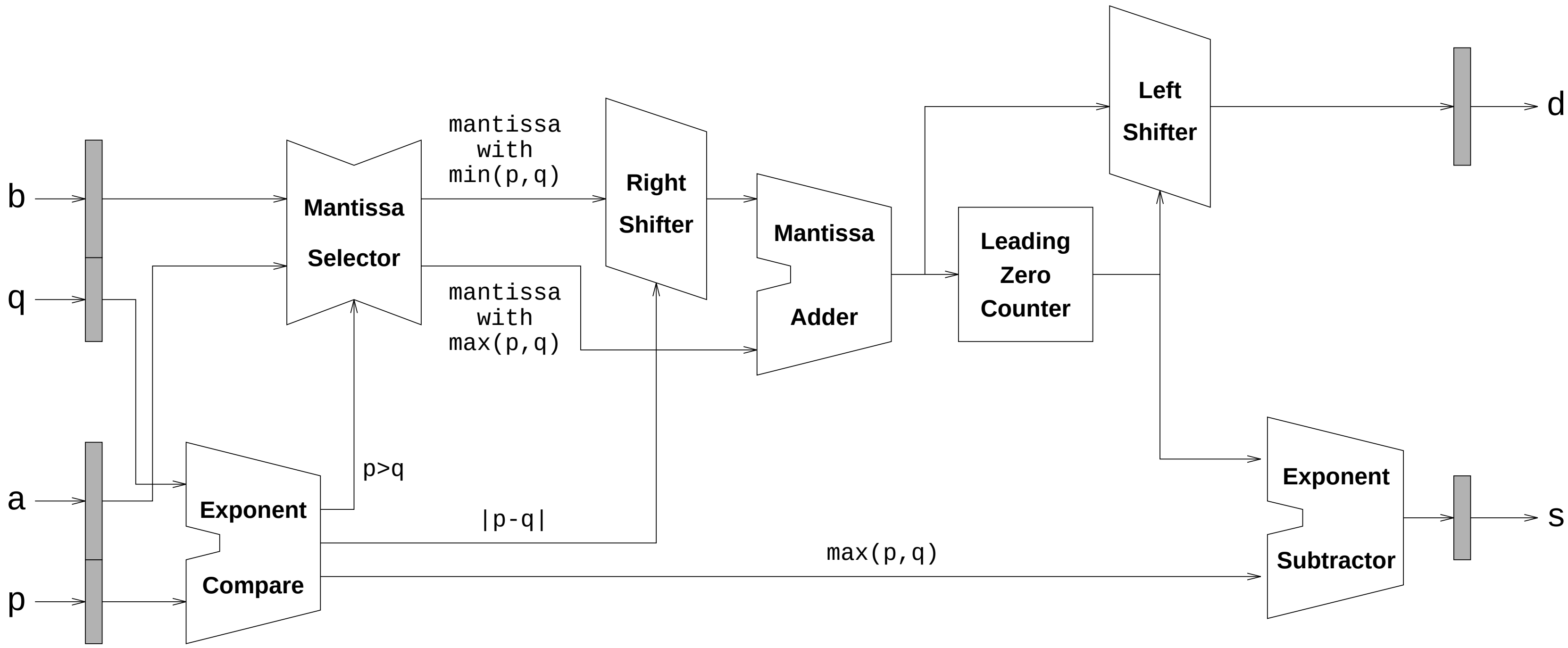
- Unpack operands
  - make most significant bit explicit; '1' for normal numbers, '0' for denormals
  - set biased exponent to '1' for denormals

After

- Normalize
  - shift left mantissa and decrement exponent up to the limit imposed by the exponent range
- Pack result
  - if m.s.b. = zero set biased exponent to zero
  - discard m.s.b.

Floating Point Addition

$$d \times 2^s = a \times 2^p + b \times 2^q$$



Careful control is required to avoid loss of accuracy[3]

[3] IEEE standard defines that the result should be as if calculated exactly and rounded in one of a number of ways

Floating Point Multiplication

$$d \times 2^s = (a \times 2^p) \times (b \times 2^q) = ab \times 2^{(p+q)}$$

- Simple Algorithm
  - Multiply mantissas
    $$d = a \times b$$
    sign magnitude integer multiplication[4]
  - Add exponents
    $$s = p + q$$
    biased addition $[s + bias] = [p + bias] + [q + bias] - bias$

[4]note that allowance must be made for the result which has twice the original precision with two bits before the binary point