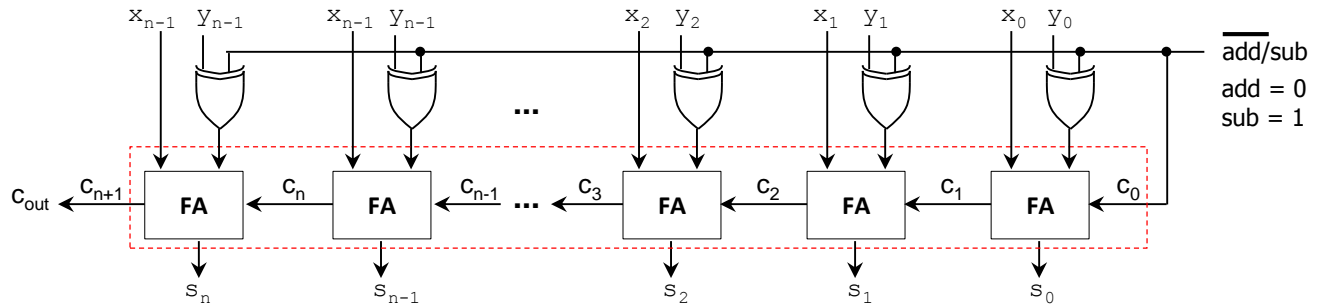# Unit 3 – Special-Purpose Arithmetic Circuits and Techniques

INTEGER/FIXED-POINT CIRCUITS

ADDITION/SUBTRACTION
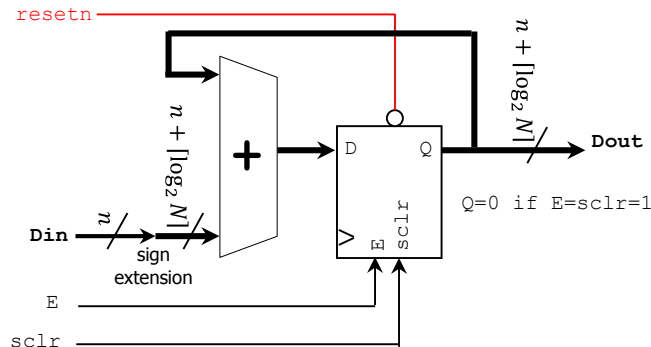
- Adder/Subtractor unit for two $n-$bit signed numbers.
  Notice the extra Full Adder. This takes care of the sign-extension to make sure that the circuit does not generate overflow.
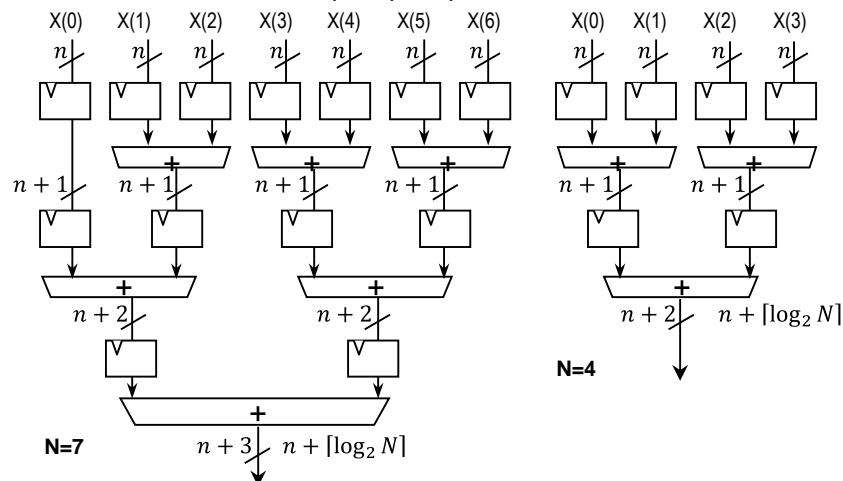

MULTI-OPERAND ADDITION

ACCUMULATOR
- Addition of N $n-$bit numbers (signed):
- Note how the required number of bits grow to $n + \lceil \log_2 N \rceil$

ADDER TREE
- Unsigned numbers: no need to zero extend numbers, just use the carry out as the MSB of the result.
- Signed numbers: at every stage, we need to sign extend the operands, so as to get the proper result.
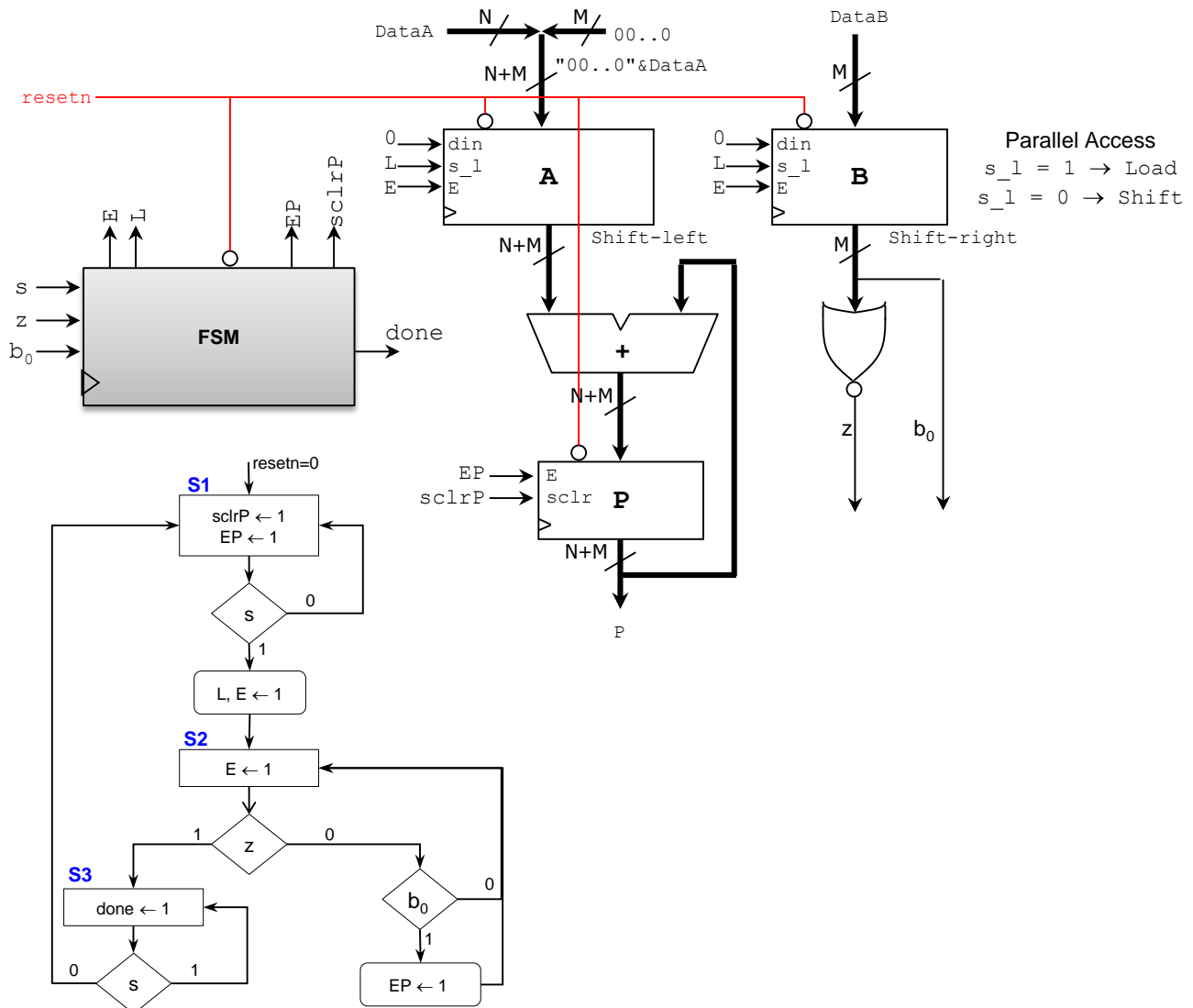- Pipelining: Registers are used to increase the frequency of operation.

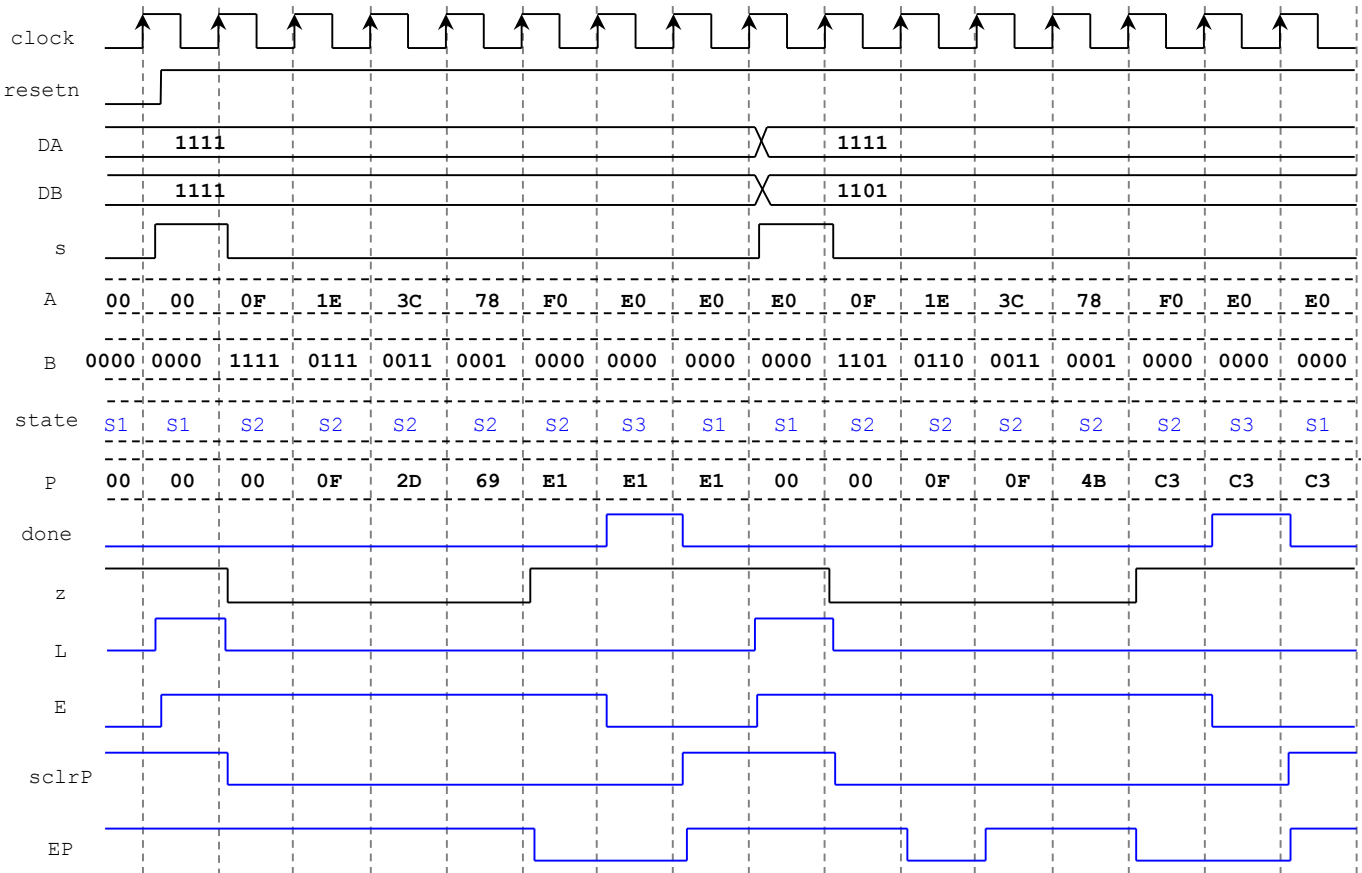MULTIPLICATION

UNSIGNED MULTIPLICATION
- Sequential algorithm:

	Example:

P ← 0, Load A,B
while B ≠ 0
    if b_0 = 1 then
        P ← P + A
    end if
    left shift A
    right shift B
end while
	`1 1 1 1 x`
	`  1 1 0 1`
	
	`  1 1 1 1` ⟶ P ← 0 + 1111
	`0 0 0 0` ⟶ P ← 1111
	`1 1 1 1` ⟶ P ← 1111 + 111100 = 1001011
	`1 1 1 1` ⟶ P ← 1001011 + 1111000 = 11000011
	`1 1 0 0 0 0 1 1`
	P ← 0, A ← 1111, B ← 1101
	b_0=1 ⇒ P ← P + A = 1111. A ← 11110, B ← 110
	b_0=0 ⇒ P ← P = 1111. A ← 111100, B ← 11
	b_0=1 ⇒ P ← P + A = 1111 + 111100 = 1001011. A ← 1111000, B ← 1
	b_0=1 ⇒ P ← P + A = 1001011 + 1111000 = **11000011.** A ← 11110000, B ← 0

- Iterative Multiplier Architecture: FSM + Datapath circuit.
  $sclr$: synchronous clear. In this case, if $sclr = 1$ and $E = 1$, the register contents are initialized to 0.
 The solution is computed in at most $M + 1$ cycles.

Example (timing diagram):

clock																
resetn																
DA	1111									1111						
DB	1111									1101						
s																
A	00	00	0F	1E	3C	78	F0	E0	E0	E0	0F	1E	3C	78	F0	E0	E0
B	0000	0000	1111	0111	0011	0001	0000	0000	0000	0000	1101	0110	0011	0001	0000	0000	0000
state	S1	S1	S2	S2	S2	S2	S2	S3	S1	S1	S2	S2	S2	S2	S2	S3	S1
P	00	00	00	0F	2D	69	E1	E1	E1	00	00	0F	0F	4B	C3	C3	C3
done																	
z																	
L																	
E																	
sclrP																	
EP																	

SIGNED MULTIPLICATION
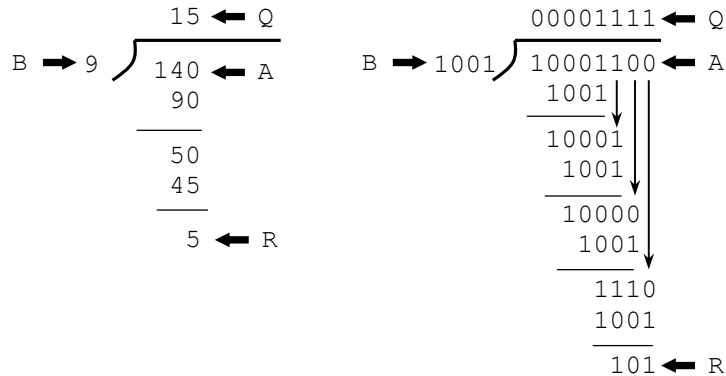
▪ Based on the iterative unsigned multiplier:
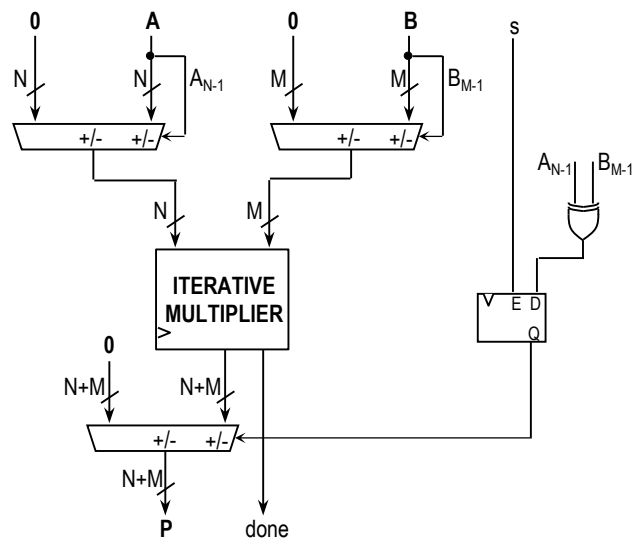
DIVISION

UNSIGNED DIVISION

- Unsigned division: Iterative case
  For the implementation, we follow the hand-division method. We grab bits of A one by one and compare it with the divisor. If the result is greater or equal than B, then we subtract B from it. On each iteration, we get one bit of Q. The example below shows the case where $A = 10001100; B = 1001$.

```
        15 ◄── Q                    00001111 ◄── Q              ALGORITHM
B ➡ 9 ⌐────                B ➡ 1001 ⌐────────
        140 ◄── A                   10001100 ◄── A              R = 0
        90                          1001                        for i = N-1 downto 0
        ────                        ─────                           left shift R (input = aᵢ)
        50                          10001                          if R ≥ B
        45                          1001                               qᵢ = 1, R ← R-B
        ────                        ─────                          else
        5 ◄── R                     10000                              qᵢ = 0
                                    1001                          end
                                    ─────                     end
                                    1110
                                    1001
                                    ─────
                                    101 ◄── R
```


A: N=8 bits
B: M=4 bits
R: M=4 bits
   Intermediate subtraction
   requires `M+1` bits
Q: N=8 bits
	A ← 10001100, B ← 1001, R ← 00000000
	i = 7, a_7 = 1: R ← 00001 < 1001 ⇒ q_7 = 0
	i = 6, a_6 = 0: R ← 00010 < 1001 ⇒ q_6 = 0
	i = 5, a_5 = 0: R ← 00100 < 1001 ⇒ q_5 = 0
	i = 4, a_4 = 0: R ← 01000 < 1001 ⇒ q_4 = 0
	i = 3, a_3 = 1: R ← 10001 ≥ 1001 ⇒ q_3 = 1, R ← 10001 − 1001 = 01000
	i = 2, a_2 = 1: R ← 10001 ≥ 1001 ⇒ q_2 = 1, R ← 10001 − 1001 = 01000
	i = 1, a_1 = 0: R ← 10000 ≥ 1001 ⇒ q_1 = 1, R ← 10000 − 1001 = 00111
	i = 0, a_0 = 0: R ← 01110 ≥ 1001 ⇒ q_0 = 1, R ← 01110 − 1001 = 00101
	⇒ Q ← 00001111, R ← 0101

- An iterative architecture is depicted in the figure for A with N bits and B with M bits, $N \geq M$. The register R stores the remainder. At every clock cycle, we either: i) shift in the next bit of A, or ii) shift in the next bit of A and subtract B.
- $(M + 1)$-bit unsigned subtractor: We can apply 2C operation to B. If the subtraction is negative, $cout = 0$. If the subtraction is positive, $cout = 1$ (here, we only need to capture R with M bits). This determines q_i, which is shifted into the register A, which after N cycles holds Q.
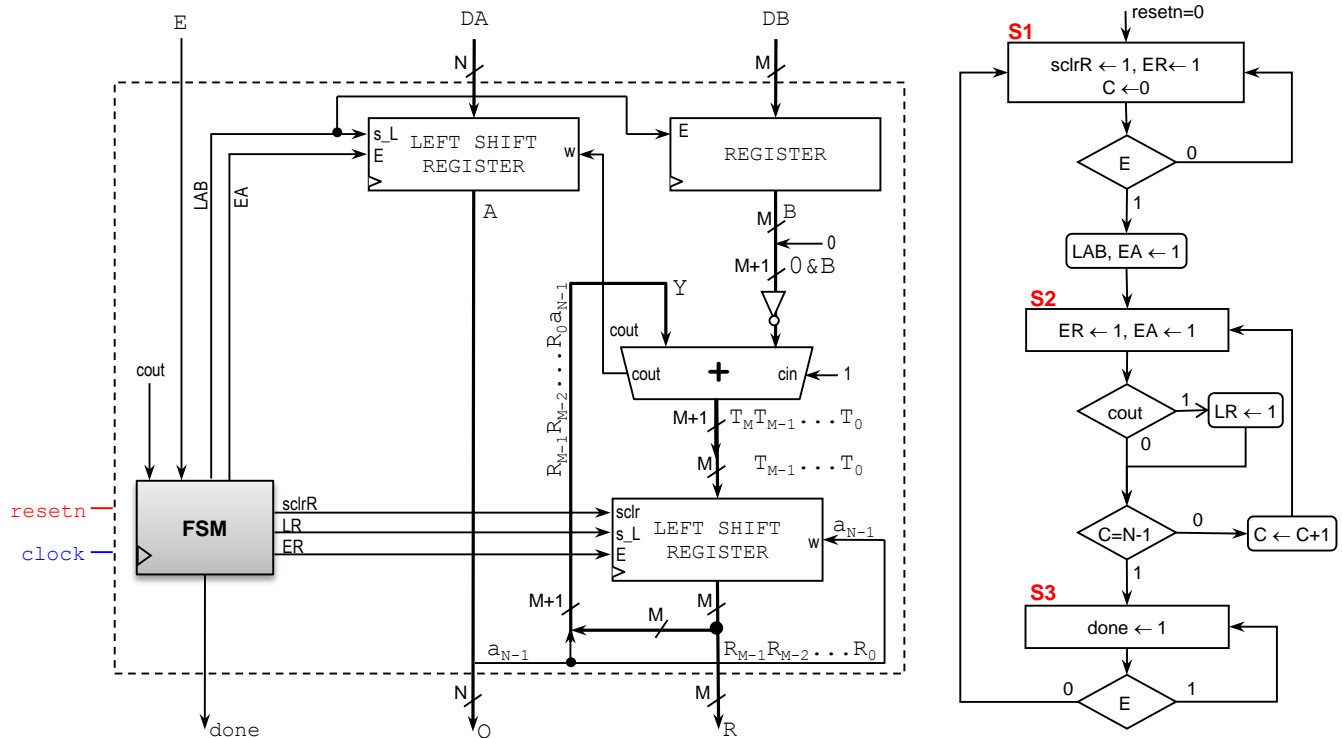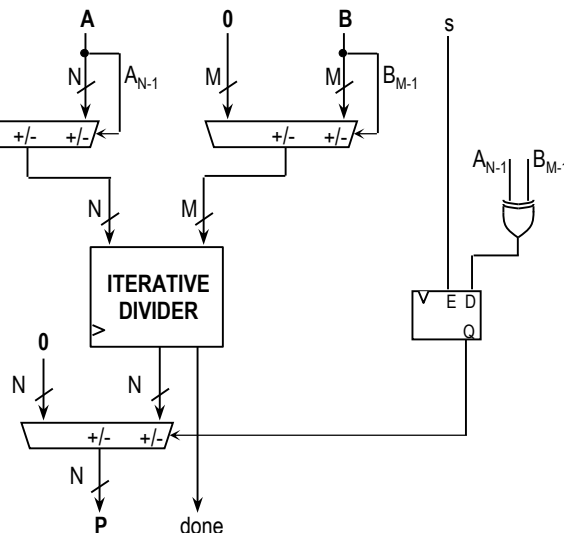
Example (timing diagram $N = 5, M = 4$). i) DA = 27, DB = 9, ii) DA = 20, DB = 7
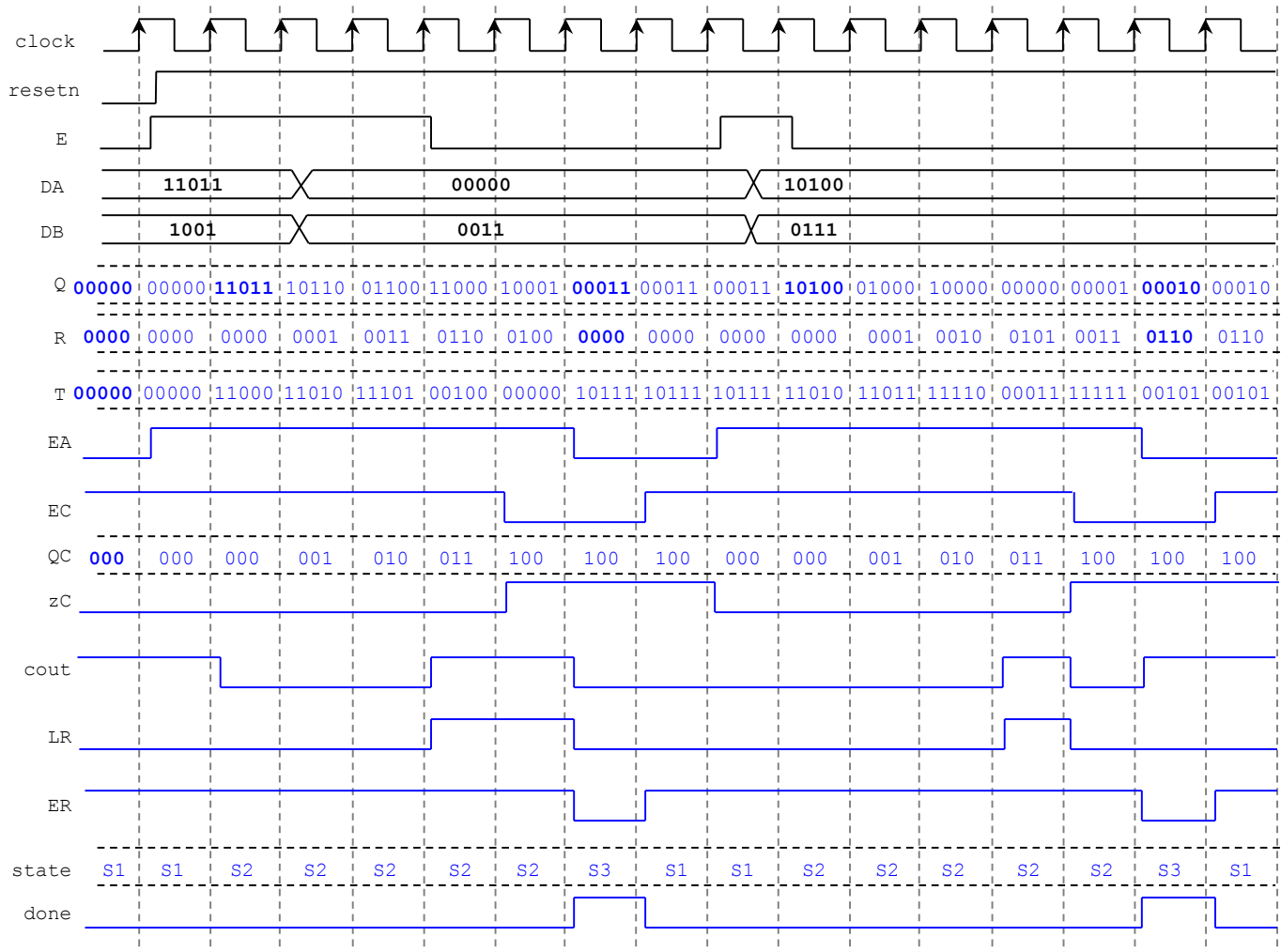
SIGNED DIVISION

- Based on the iterative unsigned divider

  ✓ Signed division: In this case, we first take the absolute value of the operators A and B. Depending on the sign of these operators, the division result (positive) of abs(A)/abs(B) might require a sign change.

FLOATING POINT CIRCUITS

FLOATING POINT ADDER/SUBTRACTOR

- e_1, e_2: biased exponents. Note that $|e_1 - e_2|$ is equal to the subtraction of the unbiased exponents.

- **U_ABS_SIGN**: This block computes $|e_1 - e_2|$. It also generates the signal sm.
  $e_1, e_2 \in [0, 2^E - 1] \rightarrow e_1 - e_2 \in [-(2^E - 1), 2^E - 1], |e_1 - e_2| \in [0, 2^E - 1]$.
  ✓ $e_1 \geq e_2 \rightarrow sm = 0, ep = e_1, f_x = f_2, f_y = f_1, b_x = b_2, b_y = b_1$
  ✓ $e_1 < e_2 \rightarrow sm = 1, ep = e_2, f_x = f_1, f_y = f_2, b_x = b_1, b_y = b_2$

- Denormal numbers: They occur if $e_1 = 0$ or $e_2 = 0$:
  ✓ $e_1 = 0 \rightarrow b_1 = 0.$ $e_1 \neq 0 \rightarrow b_1 = 1.$ ✓ $e_2 = 0 \rightarrow b_2 = 0.$ $e_2 \neq 0 \rightarrow b_2 = 1.$

- **SWAP blocks**: In floating point addition/subtraction, we usually require alignment shift: one operator (called s_x) is divided by $2^{|e_1 - e_2|}$, while the other (called s_y) is not divided.
  ○ First SWAP block: It generates s_x and s_y out of s_1 and s_2. That way we only feed s_x to the barrel shifter.
  ○ Second SWAP block: We execute $A \pm B$. For proper subtraction, we must have the minuend t_1 (either s_1 or $\frac{s_1}{2^{|e_1 - e_2|}}$) on the left hand side, and the subtrahend t_2 (either s_2 or $\frac{s_2}{2^{|e_1 - e_2|}}$) on the right hand side. This blocks generates t_1 and t_2.

|	sm	ep	s_x	s_y	t_1	t_2
$e_1 \geq e_2$	0	e_1	$s_2 = b_2.f_2$	$s_1 = b_1.f_1$	s_1	$\frac{s_2}{2^{|e_1 - e_2|}}$
$e_1 < e_2$	1	e_2	$s_1 = b_1.f_1$	$s_2 = b_2.f_2$	$\frac{s_1}{2^{|e_1 - e_2|}}$	s_2

- **Barrel shifter 2^{-i}**: This circuit performs alignment of s_x, where we always shift to the right by $|e_1 - e_2|$ bits.

- **SM to 2C**: Sign and magnitude to 2's complement converter. If the sign (sg$_1$, sg$_2$) is 0, then only a 0 is appended to the MSB. If the sign is 1, we get the negative number in 2C representation. Output bit-width: $P + 2$ bits.

- **Main adder/subtractor**: This circuit operates in 2C arithmetic. Note that we must sign-extend the $(P + 2)$-bit operands to $P + 3$ bits.
  Input operands $\in [-2^{P+1} + 1, 2^{P+1} - 1]$, Output result $\in [-2^{P+2} + 2, 2^{P+2} - 2]$.

- **U_ABS block**: It takes the absolute value of a number represented in 2C arithmetic. The output is provided as an unsigned number. The absolute value $\in [0, 2^{P+2} - 2]$, this only requires $P + 2$ bits in unsigned representation.

- **Leading Zero Detector (LZD)**: This circuit outputs a number that indicates the amount of shifting required to normalize the result of the main adder/subtractor. It is also used to adjust the exponent. This circuit is commonly implemented using a priority encoder. $result \in [-1, p]$. The result is provided as a sign and magnitude.

result	output	sign	Actions
$[0, p]$	$sh \in [0, p]$	0	The barrel shifter needs to shift to the left by sh bits. Exponent adder/subtractor needs to subtract sh from the exponent ep.
-1	$sh = 1$	1	The barrel shifter needs to shift to the right by 1 bit. Exponent adder/subtractor needs to add 1 to the exponent ep.

- **Exponent adder/subtractor**: The figure is not detailed. This circuit operates in 2C arithmetic; as the input operands are unsigned, we zero-extend to $E + 1$ bits. Note that for ordinary numbers, $ep \in [1, 2^E - 2]$. The $(E + 1)$-bit result (biased exponent) cannot be negative: at most, we subtract p from ep, or add 1. Thus, we use the unsigned portion: E bits (LSBs).

- **Barrel shifter 2^i**: This performs normalization of the final summation. We shift to the left (from 0 to P bits) or to the right (1 bit). The normalization step might incur in truncation of the LSBs.

- This circuit works for ordinary numbers.
  - NaN, $\pm\infty$: not considered.
  - Denormal numbers: not implemented: this would require $|e_1 - e_2| = |1 - e_2|$ when $e_1 = 0$, or $|e_1 - 1|$ when $e_2 = 0$. But we implement $A \pm B$ when $A = 0, B = 0, A = B = 0$.
    If $A = 0$ or $B = 0$, then $s_x = 0$ (barrel shifter input). So, the incorrect $|e_1 - e_2|$ does not matter; ep will also be correct. As for the biased exponent e, if $t_1 \pm t_2 = 0$, then $A \pm B = 0$, and we must make $e = 0$ (we use a multiplexer here).
  - After normalization, the unbiased e might be $2^E - 1$. This indicates overflow, but we would need to make $f = 0$. We do not implement this, so overflow is not detected.

- Typical cases:
  - ✓ Single Precision: E = 8, P = 23.
  - ✓ Double Precision: E = 8, P = 52.
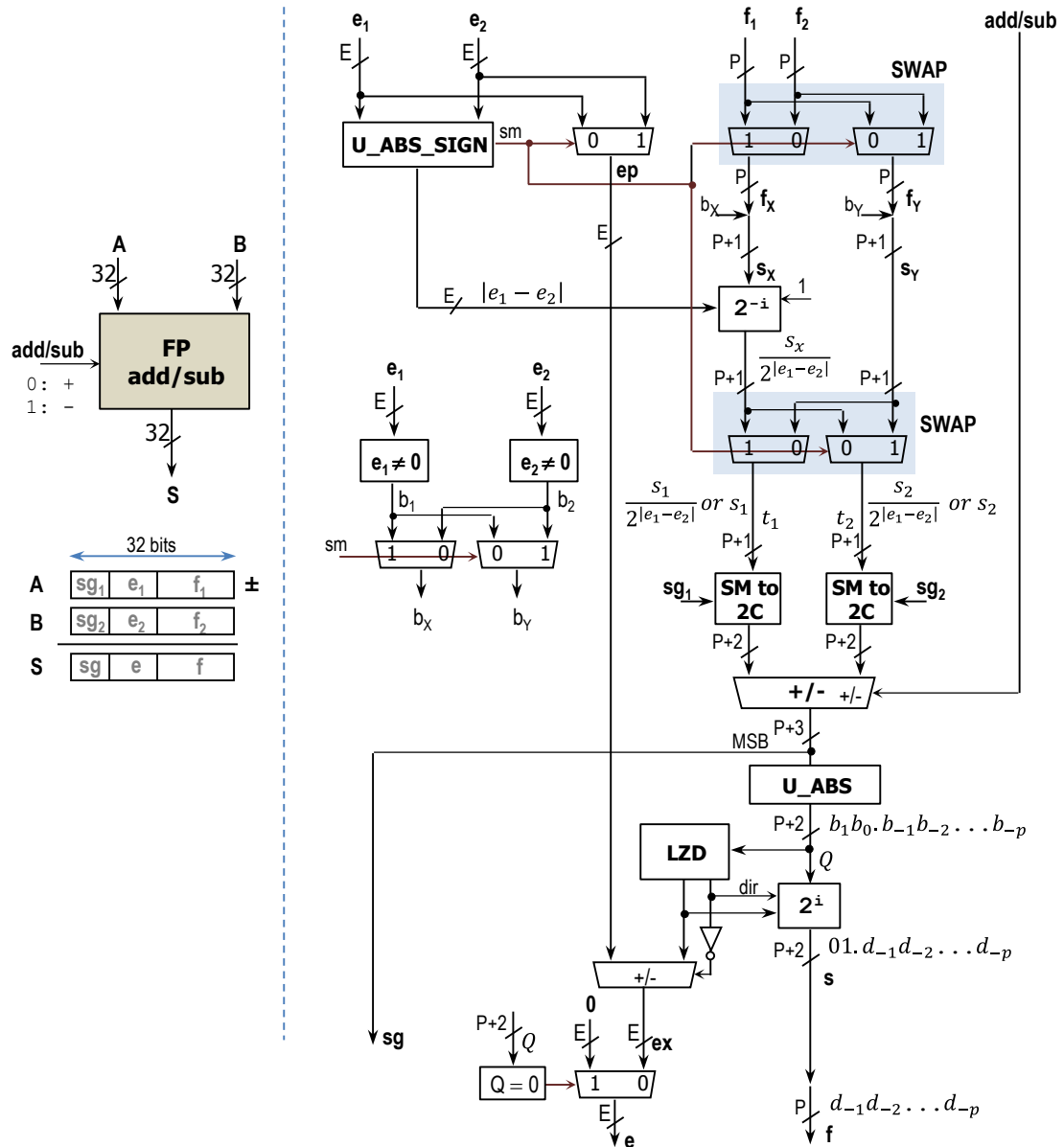
FLOATING POINT MULTIPLIER AND DIVIDER

- **Multiplier**: An unsigned multiplier is required. If we use a sequential multiplier, an FSM is required to control the dataflow.
  - ✓ We need to add the unbiased exponents: $ep = e_1 + e_2$. Here, a simple unsigned adder suffices. Since this operation adds $2 \times bias$ to ep, we subtract the $bias$ from the final adjusted exponent ex.
  - ✓ The multiplier will require $2P+2$ bits. Here, we need to truncate to $P+2$ bits.

- **Divider**: An unsigned divider is required. If we use a sequential divider, an FSM is required to control the dataflow.
  - ✓ We need to subtract the unbiased exponents: $ep = e_1 - e_2$. This requires us to operate in 2C arithmetic. Since this operation gets rid of the bias, we need to add the $bias = 2^{E-1} - 1$ to the final adjusted exponent ex.
  - ✓ The divider can include any number of extra fractional bits. We use P fractional bits of precision.

FP MULTIPLIER   **FP DIVIDER**
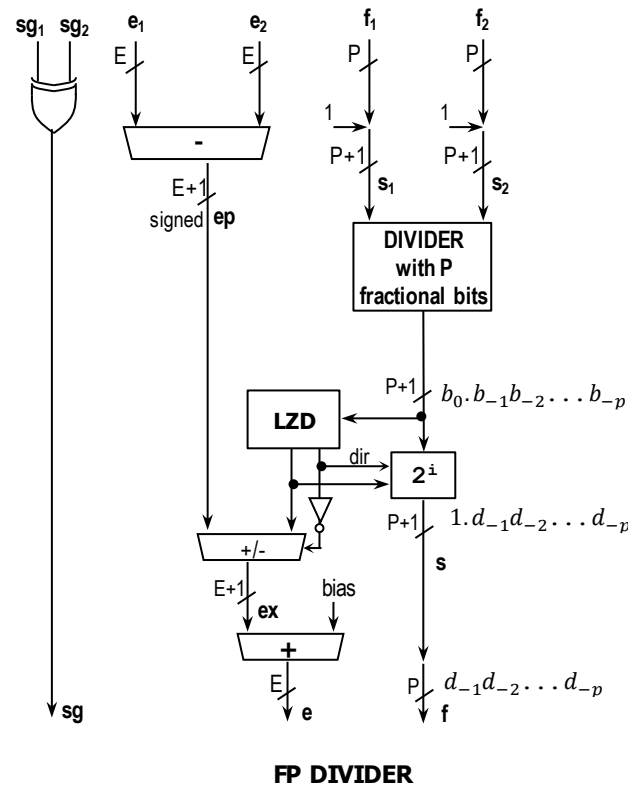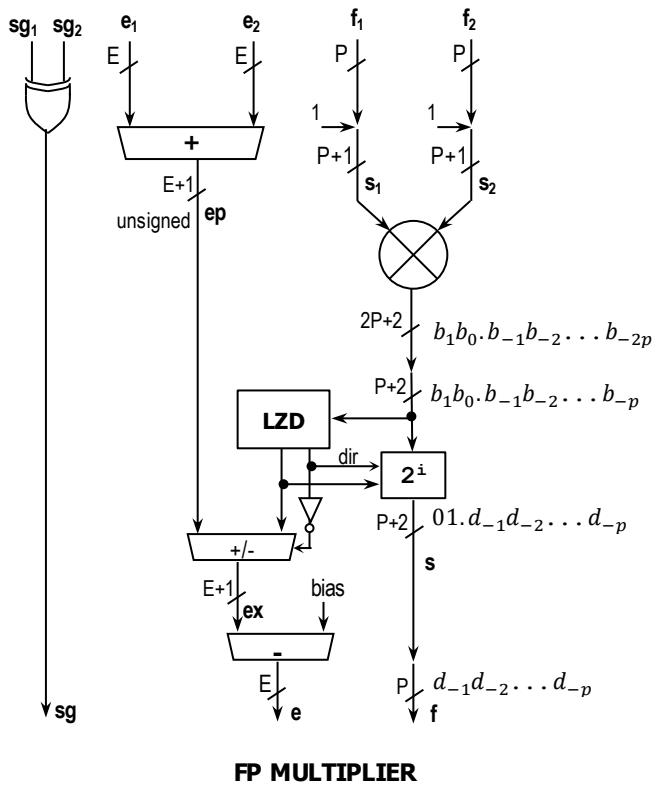
DUAL FIXED-POINT CIRCUITS

DFX ADDER/SUBTRACTOR

- Here, we add two DFX numbers A and B with n bits. To do this, we get rid of the exponent (E) bit, align the numbers, and then add two $(n-1)$-bit significands in fixed point arithmetic. Then, we convert the FX result into the DFX number.

PRE-SCALER

- It makes sure that the input operands (Asig, Bsig) are aligned. Four possibilities exist, based on the exponents of A and B:

A_{n-1}	B_{n-1}	Operation
0	0	$num0 + num0$. No need to align.
0	1	$num0 + num1$. Here, $num0$ is converted to $num1$. The $p_0 - p_1$ discarded bits are saved.
1	0	$num1 + num0$. Here, $num0$ is converted to $num1$. The $p_0 - p_1$ discarded bits are saved
1	1	$num1 + num1$. No need to align.

- If they both are either $num0$ or $num1$, addition is straightforward.
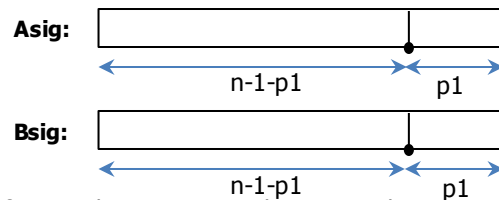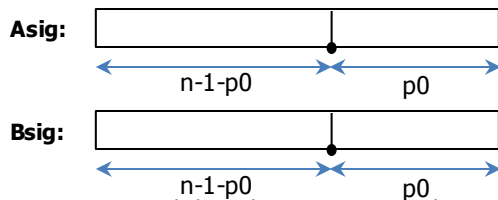
- If one is $num0$ and the other is $num1$, we have to align the fractional points to p_1. This means that we convert $[n-1 \ p_0]$ to $[n-1 \ p_1]$ by discarding $p_0 - p_1$ fractional bits and by sign-extending the extra $p_0 - p_1$ MSBs. This is not exactly the same as converting $num0$ to $num1$, because the $num0$ number fits with n bits, though the operation is very similar.
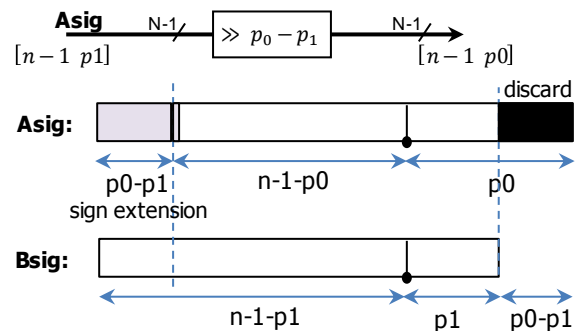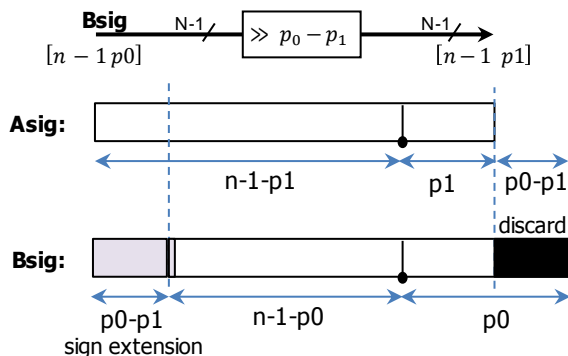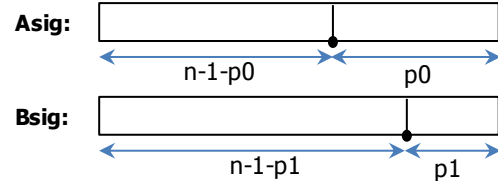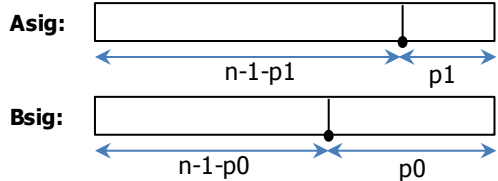
- Converting from $[n-1 \ p_0]$ to $[n-1 \ p_1]$: This operation consists of: arithmetic shift of $p_0 - p_1$ bits to the right, truncation of $p_0 - p_1$ LSBs, while keeping the fractional point where it is. This operation is not exactly $\gg p_0 - p_1$, but it is usually represented as such.
- **Improving DFX Adder accuracy**: We save the $p_0 - p_1$ truncated LSBs. In the post-scaler, we might need to convert $[n \ p_1]$ to $[n \ p_0]$. This operation requires shifting to the left, and we can shift in the truncated LSBs. This only happens when A and B have different exponents. If A and B are both $num0$, the sum S is $[n \ p_0]$: we cannot shift in any other bit. If A and B are both $num1$, the sum S is $[n \ p_1]$, and there were never truncated LSBs to begin with.

FIXED-POINT ADDITION

- Once the numbers are aligned, we perform the fixed-point addition of two $(n-1)$-bit FX numbers. This is done by sign-extending the operands to n bits; the result has n bits with either p_0 or p_1 fractional bits.
- DFX addition: We want the result to have the same number of bits as the inputs. We can always sign-extend the MSB of the significand to avoid overflow, but this defeats the purpose of DFX (we better just use FX).
- Overflow of FX addition: Here, we consider the overflow as if the addition were of two $(n-1)-$bit numbers (with no sign-extension), i.e., $overflow_{n-1} = c_{n-1} \oplus c_{n-2}$. We need this overflow since it tells us whether $n-1$ bits suffice for the addition

result. Note that the n-bit addition overflow is always zero (due to sign-extension). The FX adder performs n-bit addition (by sign-extending); however, note that the DFX format requires one exponent bit and $n - 1$ significand bits.

POST-SCALER

- If at least one input is $num1$, then the sum S will be in $[n\ p1]$. If A and B are $num0$, then the sum S will be $[n\ p0]$. Then, we need to determine whether the DFX n-bit number is a $num0$ or $num1$. If the sum $[n\ p0]$ has $overflow_{n-1} = 1$, then we convert the number to $num1$. If the sum $[n\ p1]$ has $overflow_{n-1} = 1$, then the DFX addition requires an overflow.
- From $[n\ p0]$ to $[n\ p1]$: This is the same circuit $\gg p_0 - p_1$ as in the pre-scaler, but here we use n bits as input.
- From $[n\ p1]$ to $[n\ p0]$: Left shift with zero pad (or we shift in the truncated bits that we saved).
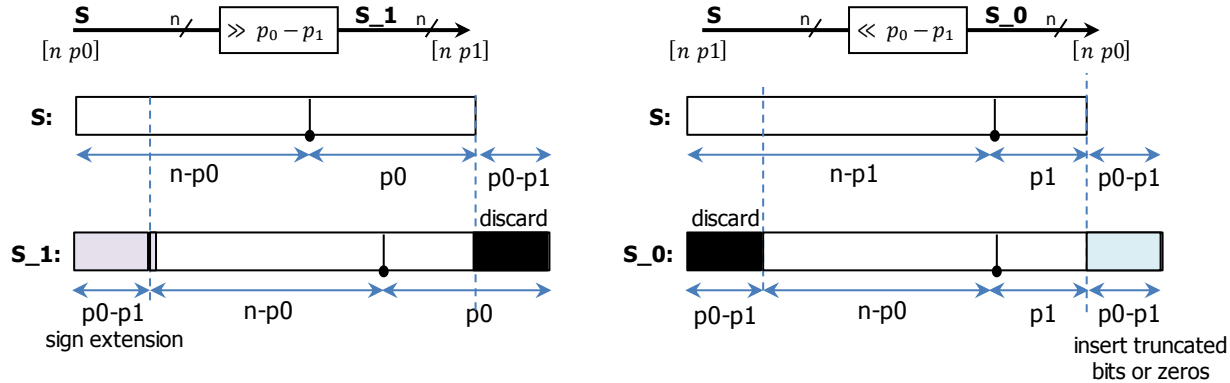
- **3-input Multiplexor**: it takes three $n-$bit FX inputs and outputs one $(n - 1)-$bit FX output (the MSB is discarded). Note how the saved $p_0 - p_1$ bits might be used when the final summation needs to be converted to $num0$.

- **Range Detector**
  ✓ It determines whether a fixed point (FX) number $[nin\ pin]$ can be represented as a DFX $num0$ number with n bits. Note that $E_{RD} = 1$ does not necessarily imply a DFX $num1$ number with n bits, because it may actually need more than n bits.
  ✓ For the DFX number to be $num0$ with n bits, the corresponding FX number has to be such that the $nin - pin - (n - 1 - p_0) + 1$ MSBs have be all 1 or 0 (due to sign extension). This means only one of those bits is needed.
  ✓ The figure assumes that: $nin - pin \geq n - 1 - p0$, $p0 \geq pin$. If $nin - pin < n - 1 - p0$ then the FX number is a $num0$ DFX number with n bits. If $p0 < pin$, we need to get rid of $p0 - pin$ LSBs (we lose precision here).

$$E_{RD} = \overline{b_{nin-pin-1} \cdots b_{n-1-p0-1}} + \overline{\overline{b_{nin-pin-1}} \cdots \overline{b_{n-1-p0-1}}}$$
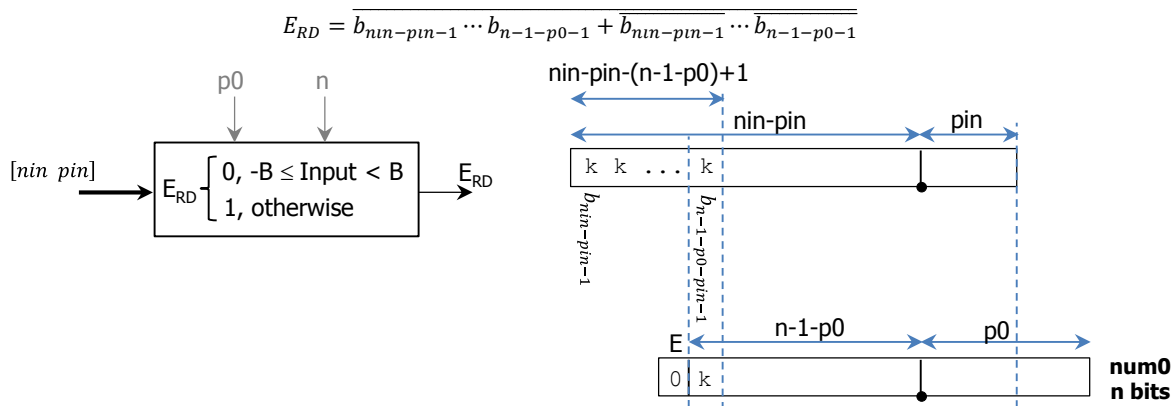
- The range detector needs to know FX format of the input signal (sum S), which could be $[n\ p_0]$ or $[n\ p_1]$. In the DFX adder/subtractor, we assume the input format to be $[n\ p_1]$. So, what happens if the input format is $[n\ p_0]$? Here, the Range Detector output will be invalid. This is why we need the signal f_num0 which indicates whether the format of S is $[n\ p_0]$.
  ✓ $f\_num0 = 0$: This means that the format of S is $[n\ p_0]$ and that E_{RD} is invalid. Here, $E = 0$. However, this does not mean that the number S can be represented in DFX as a $num0$ with n bits (since the result of the range detector is invalid). We need the $overflow_{n-1}$ bit to determine that. If this bit is 1, we need to convert S to $num1$ to avoid DFX overflow; if that bit is 0 the number S is a $num0$.
  ✓ $f\_num0 = 1$: This means that the format of S is $[n\ p_1]$. Here, $E = E_{RD}$. If $E = 0$, the sum S is a $num0$ with n bits. If $E = 1$, the sum S might be a $num1$ with n bits (we need to determine $overflow_{n-1}$ this).

- $overflow_{n-1}$: The adder/subtractor sign-extends the inputs of width $n - 1$ and the result is a n-bit number. The overflow of this circuit is always 0 (due to sign extension). $overflow_{n-1}$ refers to the overflow when only considering the $(n - 1)$-bit addition/subtraction. This useful signal determines whether the sum S requires more than $n - 1$ bits.
- Addition: We save the $p_0 - p_1$ bits that are discarded in the Pre-Scaling stage. If the final result is a $num0$, we can bring back those bits to increase precision. But if the final result is $num1$, we lose those bits for good.

- Subtraction: We would need a drastic change in the architecture to save those $p_0 - p_1$ bits discarded in the Pre-scaler. When $addsub = 1$, the carry in of the FX adder/subtractor is 1, thus the $p_0 - p_1$ bits of the subtrahend (even if flipped) are not useful. So, we do not save the $p_0 - p_1$ bits in the case of subtraction.

- **Overflow (DFX Adder/Subtractor)**: This occurs when the sum S cannot be represented as a $num1$ with n bits. One way to overcome this problem is to increase the DFX format to $n + 1$ bits, though this is not customary as the idea of DFX arithmetic is to keep the same number of bits throughout the operations.

- **Control Block**:

overflow$_{N-1}$	E_C	f_num0	overflow	ECTRL	sCTRL	Comments
0	0	0	0	0	00	Sum S is $[n\,p_0]$ and no overflow with $n-1$ bits: The sum S can be represented in DFX as a $num0$ with n bits.
0	0	1	0	0	10	Sum S is $[n\,p_1]$ and $E_C = 0$ means that the sum S can be represented in DFX as a $num0$ with n bits.
0	1	0	0	1	01	Impossible case: E_C should be 0 if $f\_num = 0$.
0	1	1	0	1	00	Sum S is $[n\,p_1]$, $E_C = 1$ means that S is not a $num0$ with n bits. As there is no overflow with $n-1$ bits, the sum S can be represented as a $num1$ with n bits.
1	0	0	0	1	01	Sum S is $[n\,p_0]$ and overflow with $n-1$ bits: The sum S needs to be first converted to $[n\,p1]$, where it can be represented as a $num1$ with n bits.
1	0	1	0	0	10	Impossible case: Sum S is $[n\,p_1]$ and $E_C = 0$ means that the sum S can be represented in DFX as a $num0$ with n bits. So, $overflow_{n-1}$ cannot be 1.
1	1	0	0	1	01	Impossible case: E_C should be 0 if $f\_num = 0$.
1	1	1	1	1	00	Sum S is $[n\,p_1]$, $E_C = 1$ means that S is not a $num0$ with n bits. As there is overflow with $n-1$ bits, the sum S cannot be represented as a $num1$ with n bits. Thus, we have DFX overflow.

Examples: n=16, p0=8, p1=4

Operation	Sum (FX)	overflow$_{N-1}$	E_{rng} E_C	Post-Scale	Answer
01.0A + 01.0B	01.0A+01.0B = 02.15	0	0 0	No need	02.15
800.3 + 00.CA	000.3+000.C = 000.F	0	0 1	To $[n\,p_0]$, append A	00.FA
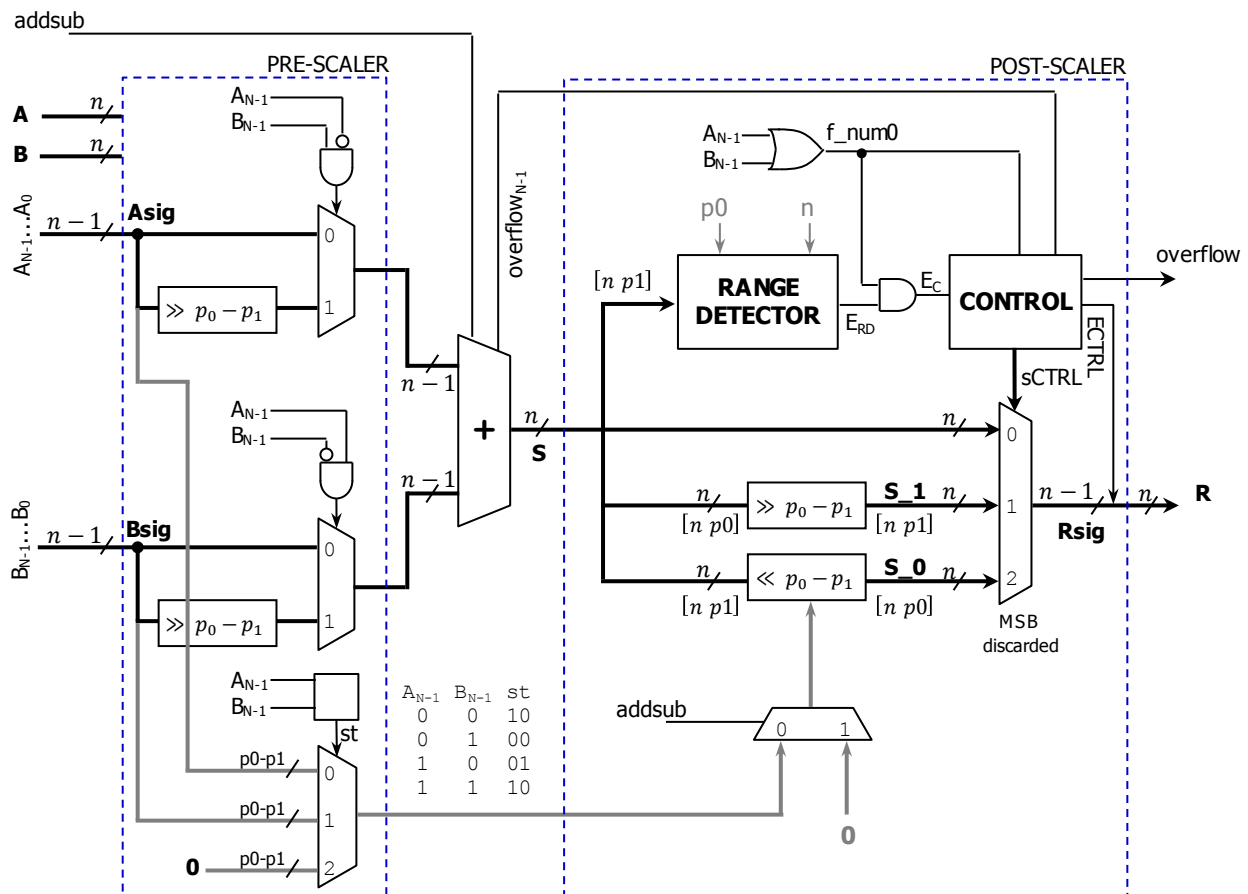
SPECIALIZED CIRCUITS
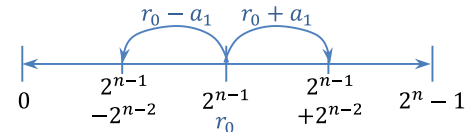
FIXED-POINT SQUARE ROOT

INTEGER SQUARE ROOT – BINARY SEARCH

- A common algorithm for hardware implementation is the 'binary search' method. There are Restoring and Non-Restoring versions. D (radical): $2n$ bits, Q (square root): n bits.
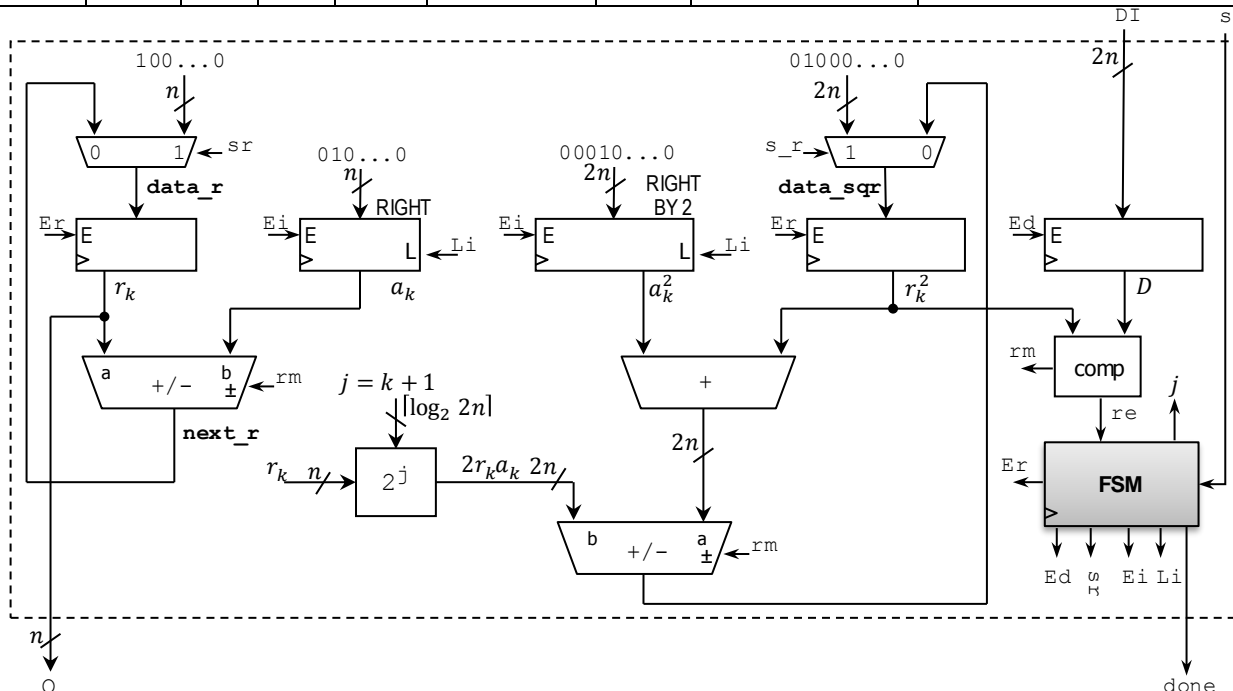
Restoring Algorithm	Non-Restoring Algorithm
$Q \leftarrow 0$ $for\ k = n - 1 \rightarrow 0$ $\quad q_k \leftarrow 1$ $\quad if\ D < Q^2\ then$ $\quad\quad q_k \leftarrow 0$ $\quad end$ end	$q_{n-1} \leftarrow 1$ $for\ k = n - 2 \rightarrow 0$ $\quad if\ D < Q^2\ then$ $\quad\quad Q \leftarrow Q - 2^k$ $\quad else$ $\quad\quad Q \leftarrow Q + 2^k$ $\quad end$ end
Example: $D = 40 = 101000, Q = 000, n = 3$ $k = 2: q_2 = 1\ (Q = 100)$ $\quad 40 < 4^2?\ No$ $k = 1: q_1 = 1\ (Q = 110)$ $\quad 40 < 6^2?\ No$ $k = 0: q_0 = 1\ (Q = 111)$ $\quad 40 < 7^2?\ Yes \rightarrow q_0 = 0\ (Q = 110)$ Result: $Q = 110, R = D - Q^2 = 0100$	Example: $D = 40 = 101000, n = 3$ $q_2 = 1\ (Q = 100)$ $k = 1: 40 < 4^2?\ No \Rightarrow Q \leftarrow Q + 2^1 = 110$ $k = 0: 40 < 6^2?\ No \Rightarrow Q \leftarrow Q + 2^0 = 111$ Result: $Q = 111, R = D - Q^2$? The LSB of the result might differ from that of the restoring case. Also, the remainder might be incorrect when using this algorithm.

Non-restoring binary search hardware implementation

- For hardware implementation, we will select the non-restoring version as it is a bit simpler to implement in hardware. We make the following definitions:
  - $a_k = 2^k$. This is the correction factor at iteration k.
  - $r_k = Q(k)$. Value of the square root at iteration k.
 - $r_k^2 = Q(k)^2 = (r_{k+1} \pm a_k)^2 = r_{k+1}^2 \pm 2a_k r_{k+1} + a_k^2$.

i	k	a_k	$2a_k$	a_k^2	r_k	r_k^2	$2a_k r_{k+1}$	Algorithm (re-defined)
0	$n-1$				2^{n-1}	2^{2n-2}		$r_{n-1} \leftarrow 2^{n-1}$
1	$n-2$	2^{n-2}	2^{n-1}	2^{2n-4}	$2^{n-1} \pm 2^{n-2}$		$2^{n-1}(2^{n-1})$	$for\ k = n - 2 \rightarrow 0$
2	$n-3$	2^{n-3}	2^{n-2}	2^{2n-6}				$\quad if\ D < r_k^2\ then$
...	...	...	...	...				$\quad\quad r_k \leftarrow r_{k+1} - a_k$
$n-3$	2	2^2	2^3	2^4				$\quad else$
$n-2$	1	2^1	2^2	2^2				$\quad\quad r_k \leftarrow r_{k+1} + a_k$
$n-1$	0	2^0	2^1	2^0				$\quad end$
								end

- For hardware implementation, a_k and r_k use n bits, while a_k^2 and r_k^2 use $2n$ bits. Also, $2a_k r_k$ use $2n$ bits for its representation.
- The representation used here is unsigned. However, we use a 2C adder/subtractor to implement $r_k - a_k$. Here, note that if $r_k \geq a_k$ (which is the case), there is no need to perform the operation in 2C using $n+1$ bits, since we won't be using the $(n+1)$-bit (which is equal to 0). The same is true for $r_k^2 + a_k^2 - 2a_k r_k$, where $2n$ bits suffice.
- Comparator: $rm = 1$ if $r_k^2 > D$, else 0. $re = 1$ if $r_k^2 = D$, else 0
- The FSM generates $j = k + 1$, because the barrel shifter multiplies by $2a_k = 2^{k+1} = 2^j$.
- a_k is shifted to the right by 1 bit every clock cycle, a_k^2 is shifted to the right by 2 bits.
- The following timing diagram is for $n = 8$. It also assumes that r_k^2 is never equal to D.
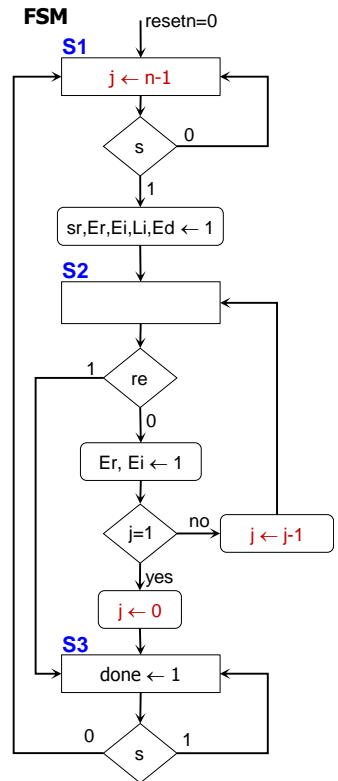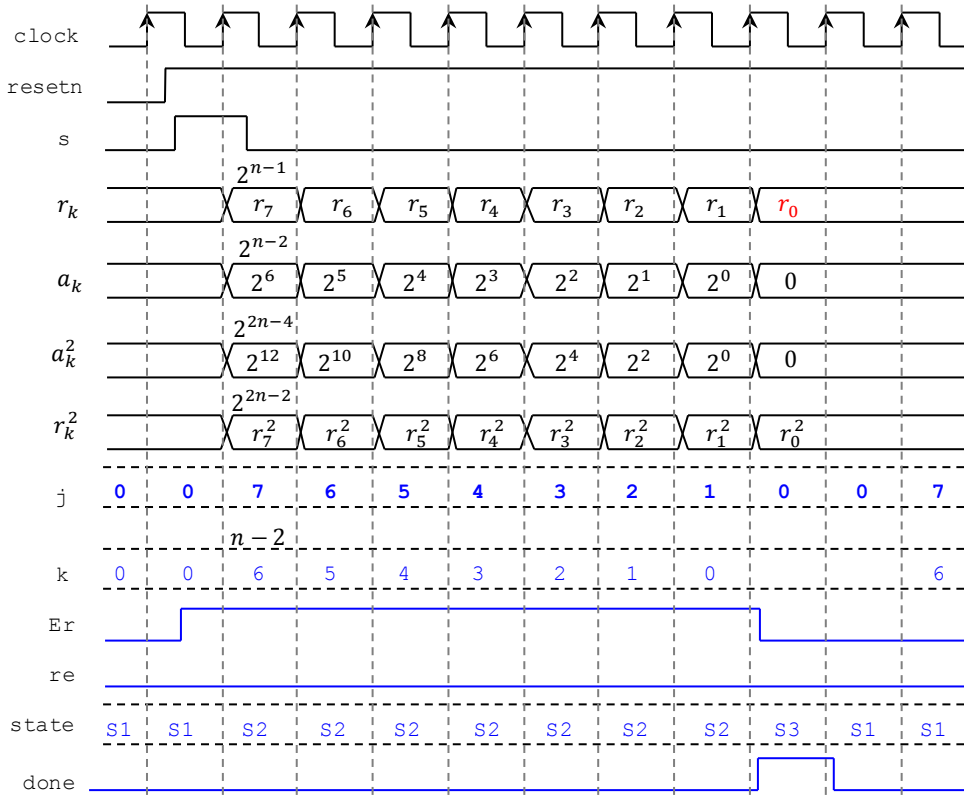
INTEGER SQUARE ROOT – OPTIMIZED NON-RESTORING ALGORITHM

- This algorithm for non-restoring square root VLSI implementation, described in *A New Non-Restoring Square Root Algorithm and its VLSI Implementation", Y. Li, W. Chu, 1996,* has proved to outperform most hardware implementations.
- A simple addition/subtraction is required based on the result bit from the previous iteration. No need for multiplexors or multipliers. The result of the addition/subtraction is fed via registers to the next iteration directly even if it is negative. At the last iteration, if the estimated remainder is positive, it is the actual remainder. Otherwise, the actual remainder is obtained via an extra addition operation. Since the remainder is rarely used, it is usually dismissed to reduce resource consumption.

Radical: $D = d_{2n-1}d_{2n-2}d_{2n-3}d_{2n-4} \ldots d_1 d_0$
Square Root: $Q = q_{n-1}q_{n-2} \ldots q_0$

We define:
$D_k = d_{2n-1}d_{2n-2} \ldots d_k$, $k = 0,1, \ldots, 2n - 1$ D_{2k} has $2n - k$ bits. Unsigned integer.
$Q_k = q_{n-1}q_{n-2} \ldots q_k$, $k = 0,1, \ldots, n - 1$ Q_k has $n - k$ bits. Unsigned integer.
$R'_k = r'_n r'_{n-1}r'_{n-2} \ldots r'_k$, $k = 0,1, \ldots, n - 1$ R'_k has $n - k + 1$ bits. Signed (2C) integer.

$for\ k = n - 1\ downto\ 0$
     $if\ k = n - 1\ then$
        $R'_k = d_{2k+1}d_{2k} - 01$ $(R'_{n-1} = d_{2n-1}d_{2n-2} - 01)$
     $else$
        $R'_k = \begin{cases} R'_{k+1}d_{2k+1}d_{2k} - Q_{k+1}01, & if\ q_{k+1} = 1 \\ R'_{k+1}d_{2k+1}d_{2k} + Q_{k+1}11, & if\ q_{k+1} = 0 \end{cases}$
     end
     $q_k = \begin{cases} 1, if\ R'_k \geq 0 \\ 0, if\ R'_k < 0 \end{cases}$
end

$$Remainder\ R = R_0 = \begin{cases} R'_0, & if\ R'_0 \geq 0 \\ R'_0 + Q_1 01 = R'_0 + Q_0 1, & if\ R'_0 < 0 \end{cases}$$

- At each iteration, we compute $R'_k = r'_n r'_{n-1} r'_{n-2} \ldots r'_k$ (estimated remainder).
  - ✓ R'_k: signed (2C) integer with at most $n - k + 1$ bits. Q_k: unsigned integer with at most $n - k$ bits.
  - ✓ R'_k computation. We need: two bits from D ($d_{2k+1} d_{2k}$) and Q_{k+1} (unsigned integer with $n - k - 1$ bits).
    - ▫ Left-hand side: $R'_{k+1} d_{2k+1} d_{2k}$. This is a signed number with $n - k + 2$ bits (R'_{k+1} requires $n - k$ bits).
    - ▫ Right-hand side: This is an unsigned integer with $n - k + 1$ bits (since Q_{k+1} is unsigned integer wit $n - k - 1$ bits). We zero-extend to $n - k + 2$ bits so that it is represented as a signed integer.
    - ▫ Once the result is ready, we only take the $n - k + 1$ LSBs for R'_k (it can be shown that R'_k only needs $n - k + 1$ bits).
  - ✓ Once R'_k is computed, we get q_k (square root k^{th} bit), thereby updating Q_k.
- $k = 0$: R'_0 has at most $n + 1$ bits, i.e., one more bit than the square root $Q = Q_0$. As for the actual remainder R, it needs at most $n + 1$ bits as an unsigned number (one more than the square root Q):
  a. $R = R'_0 + Q_0 1$: Since $R'_0 < 0$ and $Q_0 1 \geq 0$, we sign-extend R'_0 and zero-extend $Q_0 1$ to $n + 2$ bits. The result R is a positive signed $(n+2)$-bit number. Thus, the remainder R is a $(n+1)$-bit unsigned integer (we drop the MSB which is 0).

- Example: $n = 4$: $D = 01111111$, $Q = 0000$. Note that R'_k has one more bit than Q_k.

k	R'_k	R'_k width	q_k	$Q_k = q_{n-1} \ldots q_k$	Q
3	$R'_3 = 01 - 01 = 00 \geq 0\ (k = n - 1)$	2	$q_3 = 1$	1	1000
2	$R'_2 = R'_3 11 - Q_3 01 = 0011 - 0101 = 1110 = 110 < 0$	3	$q_2 = 0$	10	1000
1	$R'_1 = R'_2 11 + Q_2 11 = 11011 + 01011 = 00110 = 0110 < 0$	4	$q_1 = 1$	101	1010
0	$R'_0 = R'_1 11 - Q_1 01 = 011011 - 010101 = 000110 = 00110 < 0$	5	$q_0 = 1$	1011	1011

  - ✓ Also: $R = R'_0 = 00110$ (since $R'_0 \geq 0$).

Iterative Architecture

- We use a register **R** that holds the estimated reminder R'_k. **R** and **Q** are initialized with 0's.
  - ✓ To compute R'_k, we need an $(n+2)$-bit adder/subtractor, since on the last iteration (to compute R'_0), we use $n+2$ bits:
    - ▫ $R'_0 = \begin{cases} R'_1 d_1 d_0 - Q_1 01, if\ q_1 = 1 \\ R'_1 d_1 d_0 + Q_1 11, if\ q_1 = 0 \end{cases}$. After computation, R'_0 only requires $n + 1$ bits (the LSBs).
  - ✓ The $(n+2)$-bit result of the adder/subtractor is stored on register **R**. Only the n LSBs of the register **R** are fed back to the adder/subtractor. This is because, on the last iteration, we need R'_1 that requires at most n bits.
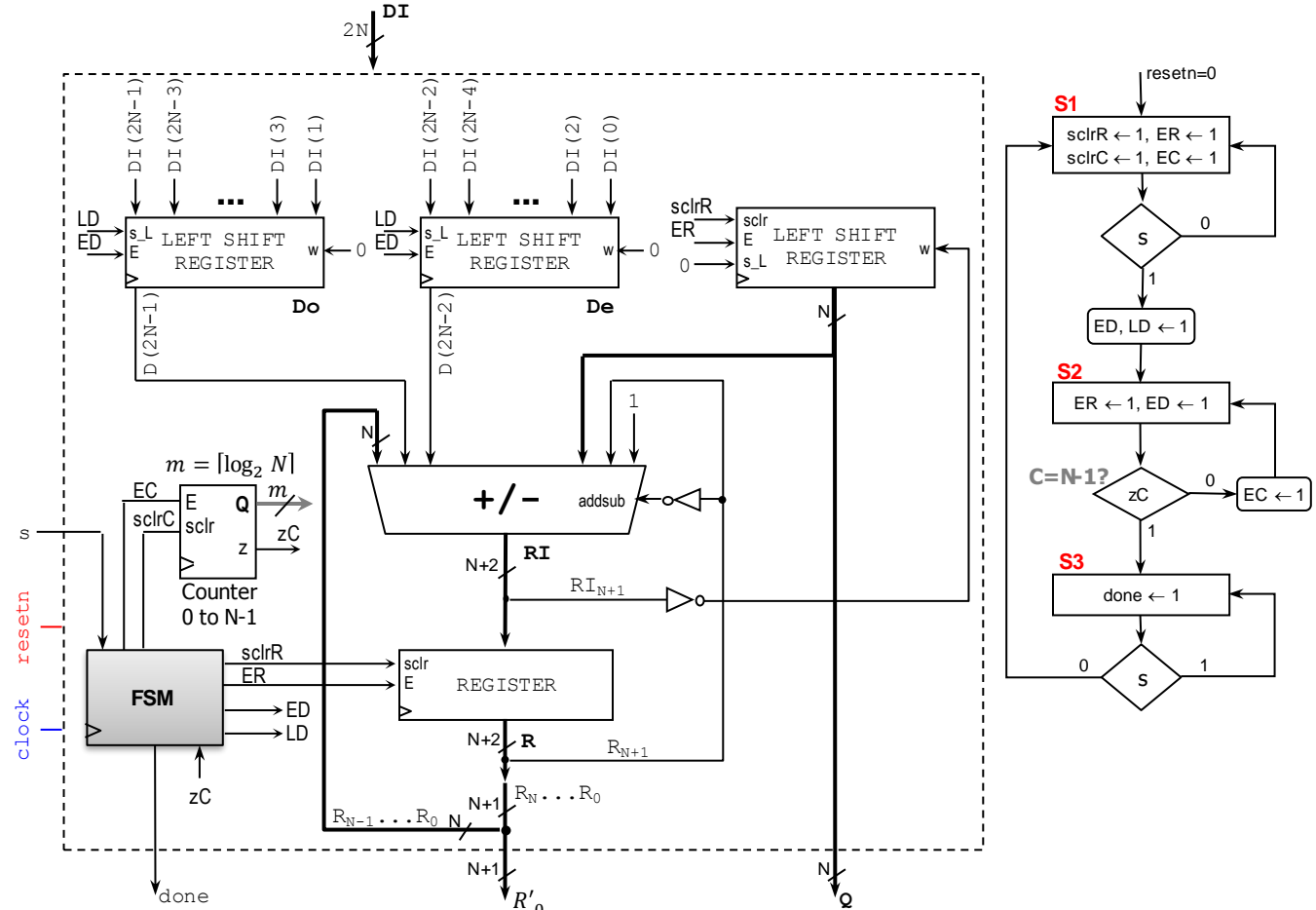
Iterative Architecture - Optimized

- The register **R** holds the estimated reminder R'_k. The register Q has n bits.

  Adder/subtractor: $n + 2$ bits. This is because of last iteration: $R'_0 = \begin{cases} R'_1 d_1 d_0 - Q_1 01, if\ q_1 = 1 \\ R'_1 d_1 d_0 + Q_1 11, if\ q_1 = 0 \end{cases}$

- $(n + 2)$-bit addition/subtraction of signed operands:

R'_{k+1}	xy +		R'_{k+1}	xy −	≡	R'_{k+1}	xy +
Q_{k+1}	11		Q_{k+1}	01		Q_{k+1}	11
ba			ba			ba	

  ✓ The 2 LSBs perform either $xy + 11$ or $xy - 01$, $xy = d_{2k+1} d_{2k}$. The operation yields: cba, where c is the carry-in of the next stage of the adder/subtractor, and ba the result of the operation.
    □ Note that $\overline{xy - 01} = xy + 11$. So, the result cba depends only on xy.
      $c = x + y$, $b = \overline{x \oplus y}$, $a = \bar{y}$.
    □ This reduces the width of the adder/subtractor by 2 bits.

cba = xy + 11		cba = xy - 01	
xy	cba	xy	cba
00	011	00	011
01	100	01	100
10	101	10	101
11	110	11	110

  ✓ The n MSBs perform $A \pm B \pm c$: an addition or subtraction where c is the carry-in (or borrow-in).
    □ For $xy + 11$: c is the carry-in to the n-bit addition.
    □ For $xy - 01$: c is the borrow-in to the n-bit subtraction $A - B$, $A = R'_{k+1}$, $B = Q_{k+1}$.
      · $c = 0$: The n MSBs implement $A + \bar{B} \equiv A - B - 1$), so this is a borrow-in.
      · $c = 1$: The n MSBs implement $A + \bar{B} + 1 = A - B$), so this is a no borrow-in.

    □ Thus, for the n-bit operation, we need a n-bit adder/subtractor with carry-in that treats the carry-in as active-high carry-in for addition and as <u>active-low borrow-in</u> for subtraction. This is a standard adder/subtractor with carry-in:
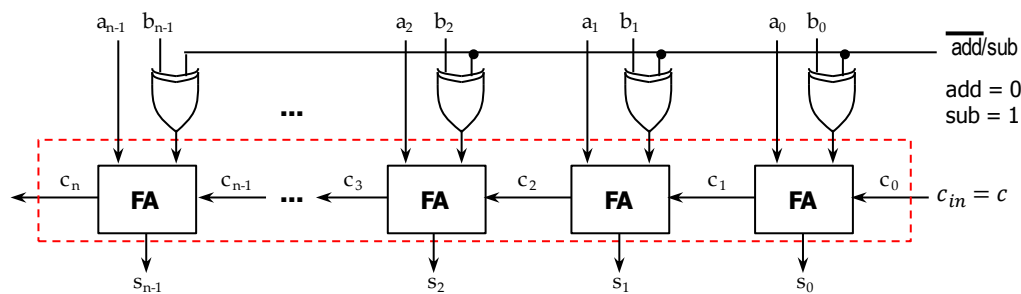
- Architecture:
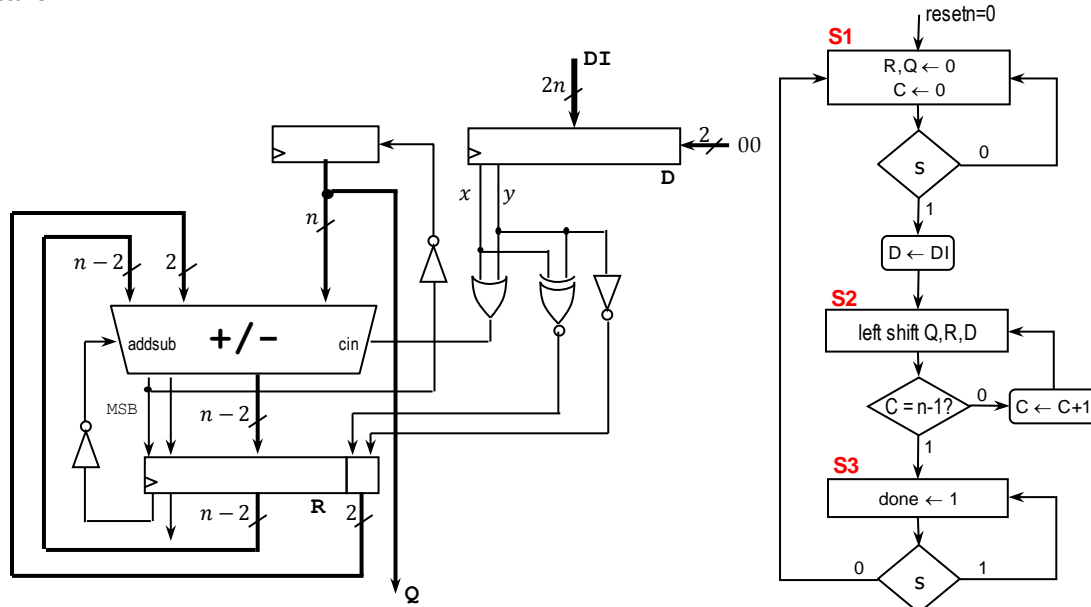
- There are some small further simplifications: the register R only needs $n + 1$ bits, thereby reducing the size of register R. Also, the MSB of Q does not need to be fed into the adder/subtractor, we can instead feed a '0' (the MSB of Q is always 0, except in the result of the last iteration, whose MSB is not fed into Q).

COMPUTING MORE PRECISION BITS

- If x more precision bits are needed, we can append $2x$ zeros to D. This implies that we need to add x extra bits to Q.
- $Dp = D \times 2^{2x}$, $Qp = \sqrt{Dp}$, $Q = \sqrt{D}$
- Dp: $2n + 2x$ bits, Qp: $n + x$ bits. x: number of precision bits

$$Qp = \sqrt{Dp} = \sqrt{D \times 2^{2x}} = \sqrt{D} \times 2^x \rightarrow Q = \sqrt{D} = \frac{Qp}{2^x}$$

Hardware changes – Optimized square root algorithm

- Let's define: $nq = n + x$. We use Q with nq bits, R with $nq + 1$ bits. The adder/subtractor uses nq bits.
- There is no need to increase the size of the register D. We can still use $2n$ bits, as '00' is always shifted in (this emulates the $2x$ zeros in the first x cycles). In the FSM, C starts with $nq - 1$, the result is obtained after nq cycles.

Example: (restoring algorithm)
Get $\sqrt{D}$ using $x = 2$ precision bits. $D = 110111 = 55$, $n = 3$
Then: $Dp = 1101110000 = 880$. Then $nq = n + x = 5$
$k = 4: q_4 = 1$ ($Q = 10000$). $880 < 16^2$? No
$k = 3: q_4 = 3$ ($Q = 11000$). $880 < 24^2$? No
$k = 2: q_2 = 1$ ($Q = 11100$). $880 < 28^2$? No
$k = 1: q_1 = 1$ ($Q = 11110$). $880 < 30^2$? $Yes \rightarrow q_2 = 0$ ($Q = 11100$)
$k = 0: q_0 = 1$ ($Q = 11101$). $880 < 29^2$? No
Result: $Qp = 11101, Rp = Dp - Qp^2 = 100111$
Final Result: $Q = 111.01 = 7.25 \approx \sqrt{55}$

FX SQUARE ROOT

What if the input (let's call it Df) is in fixed-point format $[2n\ 2p]$?

- The integer input (called D) is related to Df by: $Df = D \times 2^{-2p}$. $2n$ = number of total bits of Df.

$$Qf = \sqrt{Df} = \sqrt{D \times 2^{-2p}} = \sqrt{D} \times 2^{-p}$$

- So, we first compute the square root of D (i.e., Df without the fractional point), and then we place the fractional point so that the number has p fractional bits.

- If we need extra precision bits, we only need to add $2x$ zeros to D. Thus $Dp = D \times 2^{2x}$.

$$Qf = \sqrt{Df} = \sqrt{D} \times 2^{-p} = \sqrt{Dp \times 2^{-2x}} \times 2^{-p} = \sqrt{Dp} \times 2^{-p-x}$$

- Again, we first compute the square root of Dp, and then we place the fractional point so that the number Qf has $p + x$ fractional bits.

Example (restoring algorithm)
$Df = 111011.1011 = 59.6875, p = 2, n = 5$. Format [10 4].
Qf format: $[n + x\ p + x]$. x: extra precision bits.

Step 1: Get the integer D.
$\Rightarrow D = 1110111011 = 955$

Step 2: Add (optionally) $2x = 4$ zeros
$\Rightarrow Dp = 11101110110000 = 15280$

Step 3: Get $Qp = \sqrt{Dp}$
   Then: $Dp = 11101110110000 = 15280$. Then $nq = n + x = 5 + 2 = 7$
   $k = 6: q_6 = 1$ ($Q = 1000000$). $15280 < 64^2$? No
   $k = 5: q_5 = 1$ ($Q = 1100000$). $15280 < 96^2$? No
   $k = 4: q_4 = 1$ ($Q = 1110000$). $15280 < 112^2$? No
   $k = 3: q_3 = 1$ ($Q = 1111000$). $15280 < 120^2$? No
   $k = 2: q_2 = 1$ ($Q = 1111100$). $15280 < 124^2$? $Yes \rightarrow q_2 = 0$ ($Q = 1111000$)
   $k = 1: q_1 = 1$ ($Q = 1111010$). $15280 < 122^2$? No
   $k = 0: q_0 = 1$ ($Q = 1111011$). $15280 < 123^2$? No
   Result: $Qp = 1111011, Rp = Dp - Qp^2 = 10010111$
   Final Result ($p + x = 4$): $Qf = 111.1011 = 7.6875 \approx \sqrt{59.6875}$

CORDIC (COORDINATE ROTATION DIGITAL COMPUTER) ALGORITHM

CIRCULAR CORDIC

- The original circular CORDIC algorithm is described by the following iterative equations, where i is the index of the iteration ($i = 0, 1, 2, 3, \ldots, N - 1$). Depending on the mode of operation, the value of δ_i is either $+1$ or -1:

$$x_{i+1} = x_i + \delta_i y_i 2^{-i}$$
$$y_{i+1} = y_i - \delta_i x_i 2^{-i}$$
$$z_{i+1} = z_i + \delta_i \theta_i, \theta_i = Tan^{-1}(2^{-i})$$

$Rotation: \delta_i = +1 \ if \ z_i < 0; \ -1, otherwise$
$Vectoring: \delta_i = +1 \ if \ y_i \geq 0; \ -1, otherwise$

- Depending on the mode of operation, the quantities X, Y and Z converge to the following values, for sufficiently large N:

Rotation Mode	Vectoring Mode
$x_n = A_n(x_0 cos z_0 - y_0 sin z_0)$ $y_n = A_n(y_0 cos z_0 + x_0 sin z_0)$ $z_n = 0$	$x_n = A_n\sqrt{x_0^2 + y_0^2}$ $y_n = 0$ $z_n = z_0 + tan^{-1}(y_0/x_0)$

$A_n \leftarrow \prod_{i=0}^{N-1} \sqrt{1 + 2^{-2i}}$. For $N \to \propto$, $A_n = 1.647$. The tan^{-1} function here has a different definition (called $atan2$), as the values it computes lie in the range $[-180°, 180°]$, i.e., it indicates the quadrant where the point (x_0, y_0) lies.

- N iterations ($i = 0, 1, 2, 3, \ldots, N - 1$). x_0, y_0, z_0 are the initial values, and x_N, y_N, z_N are the final values. At iteration i, x_{i+1}, y_{i+1}, z_{i+1} are computed. Example ($N = 4$):

$i = 0$	x_0	y_0	z_0	$\theta_0 = Tan^{-1}(2^0)$	δ_0	Iteration 0 computes x_1, y_1, z_1
$i = 1$	x_1	y_1	z_1	$\theta_1 = Tan^{-1}(2^{-1})$	δ_1	Iteration 1 computes x_2, y_2, z_2
$i = 2$	x_2	y_2	z_2	$\theta_2 = Tan^{-1}(2^{-2})$	δ_2	Iteration 2 computes x_3, y_3, z_3
$i = 3$	x_3	y_3	z_3	$\theta_3 = Tan^{-1}(2^{-3})$	δ_3	Iteration 3 computes x_4, y_4, z_4
	x_4	y_4	z_4			Final Values

- With a proper choice of the initial values x_0, y_0, z_0 and the operation mode, the following functions can be directly computed:
  - ✓ $y_0 = 0, x_0 = 1/A_n$, rotation mode $\to x_n = cos z_0, y_n = sin z_0$
  - ✓ $z_0 = 0, x_0 = 1$, vectoring mode $\to z_n = tan^{-1}(y_0)$
  - ✓ $x_0 = a, y_0 = b$, vectoring mode $\to x_n = A_n\sqrt{a^2 + b^2}$. We need to post-scale the output.

LINEAR CORDIC

- This is an extension to the circular CORDIC. No scaling corrections are needed. ($i = 1, 2, 3, \ldots$).

$$x_{i+1} = x_i$$
$$y_{i+1} = y_i - \delta_i x_i 2^{-i}$$
$$z_{i+1} = z_i + \delta_i \theta_i, \theta_i = 2^{-i}$$

$Rotation: \delta_i = +1 \ if \ z_i < 0; \ -1, otherwise$
$Vectoring: \delta_i = +1 \ if \ x_i y_i \geq 0; \ -1, otherwise$

- Depending on the mode of operation, the quantities X, Y and Z converge to the following values, for sufficiently large N:

Rotation Mode	Vectoring Mode
$x_n = x_1$ $y_n = y_1 + x_1 z_1$ $z_n = 0$	$x_n = x_1$ $y_n = 0$ $z_n = z_1 + y_1/x_1$

- With a proper choice of the initial values x_0, y_0, z_0 and the operation mode, the following functions can be directly computed:
  - ✓ $y_1 = 0$, rotation mode $\to y_n = x_1 z_1$
  - ✓ $z_1 = 0$, vectoring mode $\to z_n = y_1/x_1$

HYPERBOLIC CORDIC

- This extension to the original CORDIC equations allows for the computation of hyperbolic functions, where i is the index of the iteration ($i = 1, 2, 3, \ldots$). The following iterations must be repeated to guarantee convergence: $i = 4, 13, 40, \ldots, k, 3k + 1$.

$$x_{i+1} = x_i - \delta_i x_i 2^{-i}$$
$$y_{i+1} = y_i - \delta_i x_i 2^{-i}$$
$$z_{i+1} = z_i + \delta_i \theta_i, \theta_i = tanh^{-1}(2^{-i})$$

$Rotation: \delta_i = +1 \ if \ z_i < 0; \ -1, otherwise$
$Vectoring: \delta_i = +1 \ if \ x_i y_i \geq 0; \ -1, otherwise$

- Depending on the mode of operation, the quantities X, Y and Z converge to the following values, for sufficiently large N:

Rotation Mode	Vectoring Mode
$x_n = A_n(x_1 cosh z_1 + y_1 sinh z_1)$ $y_n = A_n(y_1 cosh z_1 + x_1 sinh z_1)$ $z_n = 0$	$x_n = A_n\sqrt{x_1^2 - y_1^2}$ $y_n = 0$ $z_n = z_1 + tanh^{-1}(y_1/x_1)$

$A_n \leftarrow \prod_{i=1}^{N} \sqrt{1 - 2^{-2i}}$ (this includes the repeated iterations $i = 4, 13, 40, \ldots$). For $N \to \propto$, $A_n \cong 0.8$

- With a proper choice of the initial values x_1, y_1, z_1 and the operation mode, the following functions can be directly computed:
  - ✓ $y_1 = 0, x_1 = 1/A_n$, rotation mode → $x_n = cosh z_1, y_n = sinh z_1$
  - ✓ $z_1 = 0, x_1 = 1$, vectoring mode → $z_n = tanh^{-1}(y_1)$
  - ✓ $x_1 = y_1 = 1/A_n$, rotation mode → $x_n = y_n = cosh z_1 + sinh z_1 = e^{z_1}$
  - ✓ $x_1 = \alpha + 1, y_1 = \alpha - 1, z_1 = 0$, vectoring mode → $z_n = tanh^{-1}(\alpha - 1/\alpha + 1) = (\ln \alpha)/2$.
  - ✓ $x_1 = \alpha + 1/(4A_n^2), y_1 = \alpha - 1/(4A_n^2), z_1 = 0$, vectoring mode → $x_n = \sqrt{\alpha}$

RANGE OF CONVERGENCE

- The basic range of convergence, obtained by a method developed by X. Hu et al, *"Expanding the Range of Convergence of the CORDIC Algorithm"*, results in:

Rotation Mode:	$\|z_{in}\| \leq \theta_N + \sum_{i=i_{in}}^{N} \theta_i$	▫ Circular: $i_{in} = 0, z_{in} = z_0, \alpha_{in} = tan^{-1}(y_0/x_0)$ ▫ Linear: $i_{in} = 1, z_{in} = z_1, \alpha_{in} = y_1/x_1$
Vectoring Mode:	$\|\alpha_{in}\| \leq \theta_N + \sum_{i=i_{in}}^{N} \theta_i$	▫ Hyperbolic: $i_{in} = 1, z_{in} = z_1, \alpha_{in} = tanh^{-1}(y_1/x_1)$. Note that in the summation, we must repeat the terms $i = 4, 13, 40,$

- **Circular**: $\theta_N + \sum_{i=0}^{N} \theta_i = tan^{-1}(2^{-N}) + \sum_{i=0}^{N} tan^{-1}(2^{-i}) = 1.7433 \ (N \to \infty)$

Rotation	$\|z_0\| \leq 1.7433 \ (99.9°)$	Input angle $\epsilon [-99.9°, 99.9°]$. Functions with angles outside this range can be computed by applying trigonometric identities.
Vectoring	$\|tan^{-1}(y_0/x_0)\| \leq 1.7433 \ (99.9°) \to y_0/x_0 \ \epsilon \langle-\infty, \infty\rangle$	There are no restrictions on the ratio y_0/x_0. However, we cannot compute the angle for values outside the range $[-99.9°, 99.9°]$.

- **Linear**: $\theta_N + \sum_{i=1}^{N} \theta_i = 2^{-N} + \sum_{i=1}^{N} 2^{-i} = 1$

Rotation	$\|z_1\| \leq 1$	In both cases, there is a strict limitation on the input argument of the linear function (e.g. multiplication, division)
Vectoring	$\|y_1/x_1\| \leq 1$	

- **Hyperbolic**: $\theta_N + \sum_{i=1}^{N} \theta_i = tanh^{-1}(2^{-N}) + \sum_{i=1}^{N} tanh^{-1}(2^{-i}) = 1.182 \ (N \to \infty)$

Rotation	$\|z_1\| \leq 1.182$	This is the limitation imposed to the input argument of the hyperbolic functions. Note that the full domain of the functions $sinh$ and $cosh$ is $\langle-\infty, \infty\rangle$.
Vectoring	$\|tanh^{-1}(y_1/x_1)\| \leq 1.182 \to \|y_1/x_1\| \leq 0.807$	This is the limitation imposed to the ratio of the input arguments of the hyperbolic functions. Note that the domain of $tanh^{-1}$ is $\langle-1,1\rangle$.

EXPANDED CORDIC ALGORITHM

- The limited range of convergence of the original CORDIC algorithm can be expanded by including iterations with negative indices. We describe the expanded circular and hyperbolic CORDIC algorithms, and the functions that we will implement.

EXPANDED CIRCULAR CORDIC

$$\forall i: \begin{cases} x_{i+1} = x_i + \delta_i y_i 2^{-i} \\ y_{i+1} = y_i - \delta_i x_i 2^{-i} \\ z_{i+1} = z_i + \delta_i \theta_i, \theta_i = Tan^{-1}(2^{-i}) \end{cases}$$

$Rotation: \delta_i = +1 \ if \ z_i < 0; \ -1, otherwise$
$Vectoring: \delta_i = +1 \ if \ y_i \geq 0; \ -1, otherwise$

- There are M negative iterations ($i = -M, ..., -1$) and N positive iterations ($i = 0, 1, ..., N-1$). For sufficiently large N, the values of x_n, y_n, z_n converge to:

Rotation Mode	Vectoring Mode
$x_n = A_n(x_{in} cos z_{in} - y_{in} sin z_{in})$ $y_n = A_n(y_{in} cos z_{in} + x_{in} sin z_{in})$ $z_n = 0$	$x_n = A_n\sqrt{x_{in}^2 + y_{in}^2}, \quad y_n = 0$ $z_n = z_{in} + tan^{-1}(y_{in}/x_{in})$

$A_n = \prod_{i=-M}^{N-1} \sqrt{1 + 2^{-2i}}$. Here, the value of M affects A_n.

- We can cover the entire domain of cos/sin and range of tan^{-1} with $\theta_{max}(M) = \pi$, i.e. $M = 2$.

- $N + M$ iterations ($i = -M, -M + 1, \dots, 0, 1, 2, 3, \dots, N - 1$). x_{-M}, y_{-M}, z_{-M} are the initial values, and x_N, y_N, z_N are the final values. At iteration i, $x_{i+1}, y_{i+1}, z_{i+1}$ are computed. Example ($M = 2, N = 4$):

$i = -2$	x_{-2}	y_{-2}	z_{-2}	$\theta_{-2} = Tan^{-1}(2^2)$	δ_{-1}	Iteration -2 computes x_{-1}, y_{-1}, z_{-1}
$i = -1$	x_{-1}	y_{-1}	z_{-1}	$\theta_{-1} = Tan^{-1}(2^1)$	δ_{-2}	Iteration -1 computes x_0, y_0, z_0
$i = 0$	x_0	y_0	z_0	$\theta_0 = Tan^{-1}(2^0)$	δ_0	Iteration 0 computes x_1, y_1, z_1
$i = 1$	x_1	y_1	z_1	$\theta_1 = Tan^{-1}(2^{-1})$	δ_1	Iteration 1 computes x_2, y_2, z_2
$i = 2$	x_2	y_2	z_2	$\theta_2 = Tan^{-1}(2^{-2})$	δ_2	Iteration 2 computes x_3, y_3, z_3
$i = 3$	x_3	y_3	z_3	$\theta_3 = Tan^{-1}(2^{-3})$	δ_3	Iteration 3 computes x_4, y_4, z_4
	x_4	y_4	z_4			Final Values

- **Special Expanded Circular CORDIC**: Alternatively, we can repeat the iteration $i = 0$ two more times ($i = 0,0,0,1,2,,\dots,N - 1$) in order to get $\theta_{max}(M) = \pi$. This method optimizes hardware resources.
  ✓ $A_n = (1 + 2^0) \prod_{i=0}^{N-1} \sqrt{1 + 2^{-2i}}$. For $N \to \infty$, $A_n = 3.2935$
  ✓ $N + 2$ iterations ($i = 0,0,0, 1, 2, 3, \dots, N - 1$). x_0, y_0, z_0: initial values, and x_N, y_N, z_N are the final values. Example ($N = 4$):

$i = 0$	x_0	y_0	z_0	$\theta_0 = Tan^{-1}(2^0)$	δ_0	Iteration 0 computes x_0, y_0, z_0	x_0, y_0, z_0 is updated
$i = 0$	x_0	y_0	z_0	$\theta_0 = Tan^{-1}(2^0)$	δ_0	Iteration 0 computes x_0, y_0, z_0	x_0, y_0, z_0 is updated
$i = 0$	x_0	y_0	z_0	$\theta_0 = Tan^{-1}(2^0)$	δ_0	Iteration 0 computes x_1, y_1, z_1	
$i = 1$	x_1	y_1	z_1	$\theta_1 = Tan^{-1}(2^{-1})$	δ_1	Iteration 1 computes x_2, y_2, z_2	
$i = 2$	x_2	y_2	z_2	$\theta_2 = Tan^{-1}(2^{-2})$	δ_2	Iteration 2 computes x_3, y_3, z_3	
$i = 3$	x_3	y_3	z_3	$\theta_3 = Tan^{-1}(2^{-3})$	δ_3	Iteration 3 computes x_4, y_4, z_4	
	x_4	y_4	z_4			Final Values	

EXPANDED HYPERBOLIC CORDIC

- This extension to the original CORDIC equations allows for the computation of hyperbolic functions, where i is the index of the iteration ($i = 1, 2, 3, \dots$). The following iterations must be repeated to guarantee convergence: $i = 4, 13, 40, \dots, k, 3k + 1$.

$$i \leq 0: \begin{cases} x_{i+1} = x_i - \delta_i y_i (1 - 2^{i-2}) \\ y_{i+1} = y_i - \delta_i x_i (1 - 2^{i-2}) \\ z_{i+1} = z_i + \delta_i \theta_i, \theta_i = Tanh^{-1}(1 - 2^{i-2}) \end{cases}$$

$$i > 0: \begin{cases} x_{i+1} = x_i - \delta_i y_i 2^{-i} \\ y_{i+1} = y_i - \delta_i x_i 2^{-i} \\ z_{i+1} = z_i + \delta_i \theta_i, \theta_i = Tanh^{-1}(2^{-i}) \end{cases}$$

$Rotation: \delta_i = +1 \text{ if } z_i < 0; -1, otherwise$
$Vectoring: \delta_i = +1 \text{ if } x_i y_i \geq 0; -1, otherwise$

- There are $M + 1$ negative iterations ($i = -M, \dots, -1, 0$) and N positive iterations ($i = 1, 2, \dots, N$), with repeated iterations $4, 13, 40, \dots, k, 3k + 1$ to guarantee convergence. For sufficiently large N, the values of x_n, y_n, z_n converge to:

Rotation Mode	Vectoring Mode
$x_n = A_n(x_{in} \cosh z_{in} + y_{in} \sinh z_{in})$ $y_n = A_n(y_{in} \cosh z_{in} + x_{in} \sinh z_{in})$ $z_n = 0$	$x_n = A_n\sqrt{x_{in}^2 - y_{in}^2}, \quad y_n = 0$ $z_n = z_{in} + \tanh^{-1}(y_{in}/x_{in})$

$A_n = \left(\prod_{i=-M}^{0} \sqrt{1 - (1 - 2^{i-2})^2}\right) \prod_{i=1}^{N} \sqrt{1 - 2^{-2i}}$. Here, the value of M affects A_n.

- As M increases, the range of convergence $[-\theta_{max}(M), \theta_{max}(M)]$ can be greatly enlarged. However, this comes at the expense of a larger resource consumption.

M	$\cosh x, \sinh x, e^x$	$\ln x$
Basic CORDIC	$[-1.11820, 1.11820]$	$(0, 9.35958]$
0	$[-2.09113, 2.09113]$	$(0, 65.51375]$
1	$[-3.44515, 3.44515]$	$(0, 982.69618]$
2	$[-5.16215, 5, 16215]$	$(0, 3.04640 \times 10^4]$
3	$[-7.23371, 7.23371]$	$(0, 1.91920 \times 10^6]$
4	$[-9.65581, 9.65581]$	$(0, 2.43742 \times 10^8]$
5	$[-12.42644, 12.42644]$	$(0, 6.21539 \times 10^{10}]$
6	$[-15.54462, 15, 54462]$	$(0, 3.17604 \times 10^{13}]$
7	$[-19.00987, 19.00987]$	$(0, 3.24910 \times 10^{16}]$
8	$[-22.82194, 22.82194]$	$(0, 6.65097 \times 10^{19}]$
9	$[-26.98070, 26, 98070]$	$(0, 2.72357 \times 10^{23}]$
10	$[-31.48609, 31.48609]$	$(0, 2.23085 \times 10^{27}]$

COMPUTATION OF TRIGONOMETIC AND HYPERBOLIC FUNCTIONS

- The $cos/sin/tan^{-1}$ (circular) and $cosh/sinh/e^x/tanh^{-1}$ (hyperbolic) functions can be directly computed by proper selection of the operation mode and the initial values $x_{in} = x_{-M}, y_{in} = y_{-M}, z_{in} = z_{-M}$.

- ✓ For $e^{\alpha} = cosh\alpha + sinh\alpha$, we need $x_{in} = y_{in} = 1/A_n$, $z_{in} = 0$, *mode=rotation.*

- ▪ The functions $\sqrt{x}$, lnx, and x^y can be computed with the hyperbolic CORDIC:
  - ✓ For $\sqrt{x}$, we use $x_{in} = x + 1/(4A_n^2)$, $y_{in} = x - 1/(4A_n^2)$, $z_{in} = 0$, *mode=vectoring*.
  - ✓ For $lnx = 2tanh^{-1}(x - 1/x + 1)$, we use $x_{in} = x + 1$, $y_{in} = x - 1$, $z_{in} = 0$, *mode=vectoring*. A product by 2 is needed.

- ▪ Powering: $x^y = e^{y \ln x}$. We first get $z_n = (\ln x)/2$, followed by $z_n \times 2y = y \ln x$. Then, we use $x_{in} = y_{in} = 1/A_n$, $z_{in} = y \ln x$, *mode=rotation* to get $x_n = e^{y \ln x} = x^y$.
  - ✓ Argument bounds of x^y ((x, y) values for which x^y converges): $|y \ln x| \le \theta_{max}(M)$.

- ▪ The parameter M controls the range of convergence: $[-\theta_{max}(M), \theta_{max}(M)]$.
  - ✓ $[-\theta_{max}(M), \theta_{max}(M)]$: This is the bound on the domain of $cos/sin/cosh/sinh/e^x$ and the range of tan^{-1}, $tanh^{-1}$.
  - ✓ The domain of lnx is bounded by $\left(0, e^{\theta_{max}(M) \times 2}\right]$.
  - ✓ The domain of $\sqrt{x}$ is bounded by $\left(0, \frac{1}{4A_n^2}\left(\frac{1+tanh(\theta_{max})}{1-tanh(\theta_{max})}\right)\right]$.
- ▪ As M increases, the argument bounds of $cosh, \ sinh, e^x, tanh^{-1}, \sqrt{x}, lnx$ and x^y are greatly enlarged.

ITERATIVE FX ARCHITECTURE (BASIC CORDIC)
- ▪ The architectures shown here are such that the inputs and outputs have an identical bit width. We can reach an optimal number of iterations by noticing the iteration at which $\theta_i = Tan^{-1}(2^{-i})$ is equal to zero due to the given fixed-point format.
  - n: input/output bit width
  - ng: additional guard bits
  - nr: $nr = ng + n$: bit width of the internal registers and operators
  - N: # of iterations ($i = 0, 1, \dots, N-1$ for circular CORDIC, $i = 1, \dots, N$ for linear/hyperbolic CORDIC)

- ▪ x_i, y_i, z_i: make sure you can represent input, intermediate, and final values. For fractional bits, a common rule of thumb is "If n bits is the desired output precision, the internal registers should have $\lceil \log_2 n \rceil$ additional guard bits at the LSB position". In general, perform a through software simulation for a given number of iterations and find out the format required for proper representation of x_i, y_i, z_i.

Circular CORDIC
- ▪ The figure depicts the architecture that implements the circular CORDIC equations in an iterative fashion. The LUT (look-up table) stores the elementary angles $\theta_i = Tan^{-1}(2^{-i})$. The process begins when a start signal is asserted. After N clock cycles (i.e., N iterations), the result is obtained in the registers X, Y and Z, and a new process can be started.
- ▪ The state machine controls the load of the registers, the data that passes onto the multiplexers, the add/subtract decision for the adder/subtractors, and the count given to the barrel shifters and LUT.
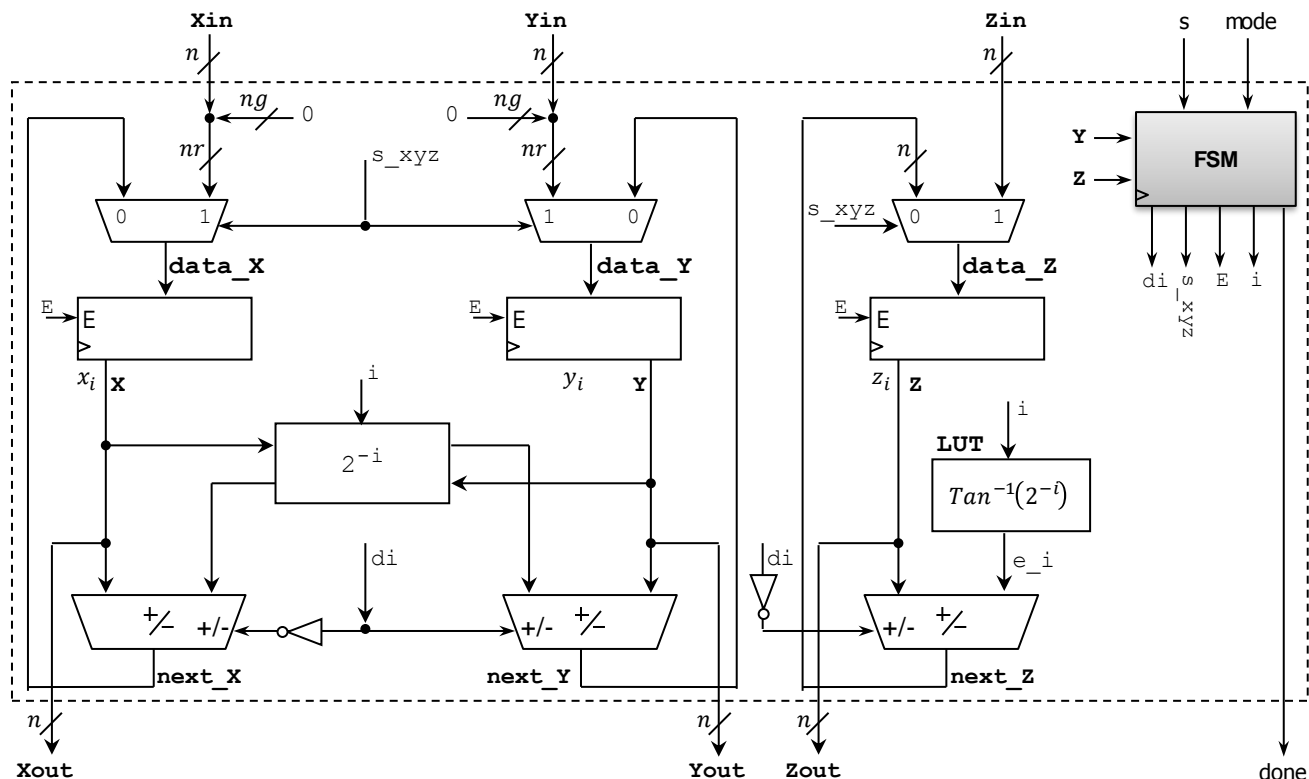
Hyperbolic CORDIC

- Here the LUT holds the $\theta_i = tanh^{-1}(2^{-i})$ values for $i = 1, 2, \ldots, N$. The FSM is more complex as it has to account for the repeated iterations. After $N - 1 + v$ (v: # of repeated iterations) clock cycles, the result is obtained in the registers X, Y and Z, and a new process can be started.
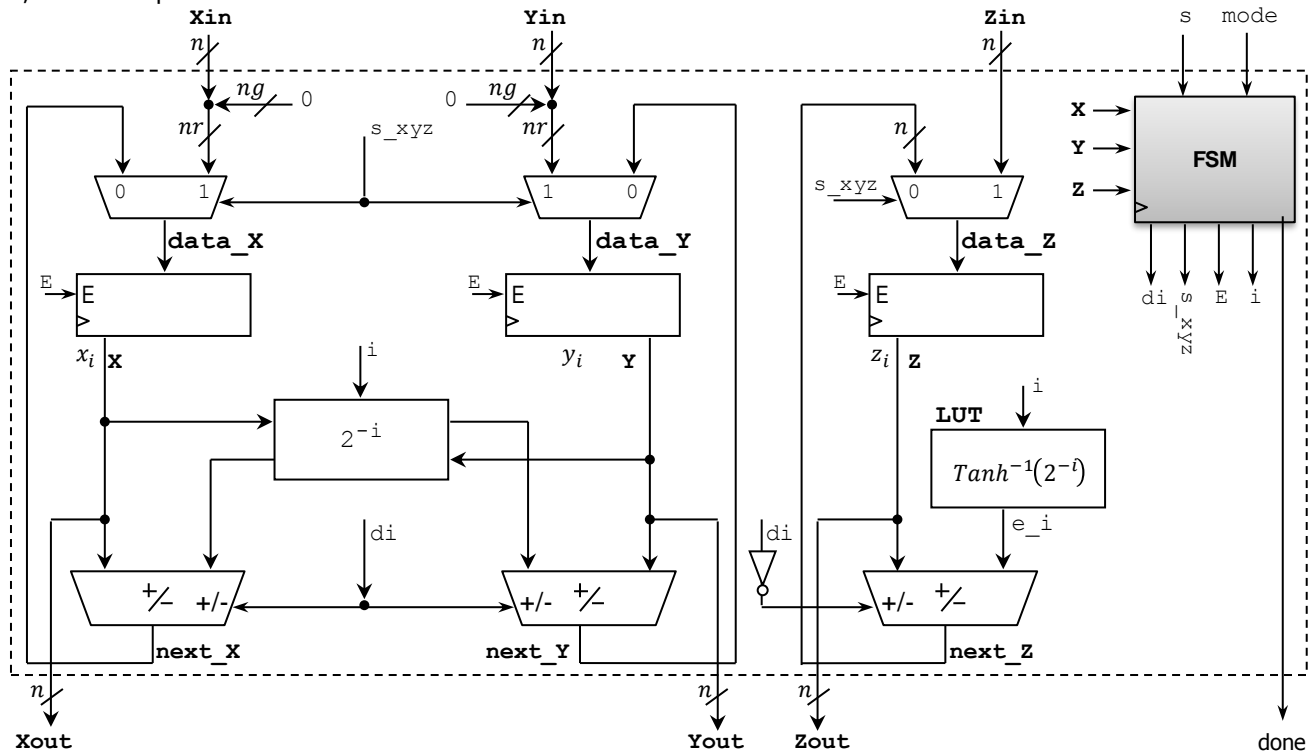
Linear CORDIC

- Here the LUT holds the $\theta_i = 2^{-i}$ values with $i = 1, 2, \ldots, N$. After $N - 1$ clock cycles, the result is obtained in the registers X, Y and Z, and a new process can be started. Note that we do not need an adder for x_i.
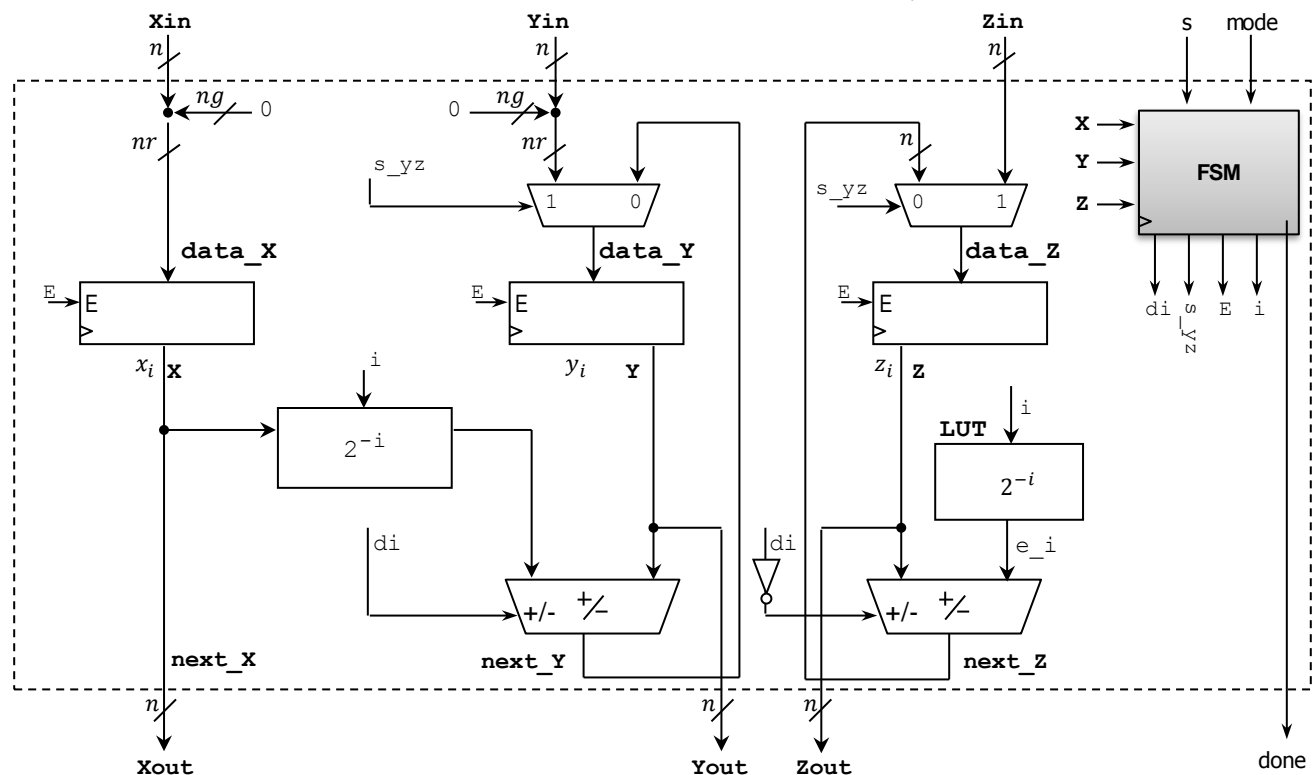
- Note that these architectures do not specify the numerical format we are using. We are free to use any format we desire (e.g.: fixed point, dual fixed point, floating point). The adders, barrel shifters, and LUT will change depending on the desired format. If an arithmetic unit requires more than one cycle to process its date, the FSM needs to account for this.

Example: FX Basic Circular CORDIC architecture. Format [16 14]

- $ng = 4$ guard bits. They improve accuracy, as the barrel shifters will get rid of many LSBs.
- mode = 0 → Rotation. mode = 1 → Vectoring.
- LUT: It holds the angles represented in [16 14] (signed) from $i = 0$ ($Tan^{-1}(2^0)$) to $i = N - 1$ ($Tan^{-1}(2^{-(N-1)})$.
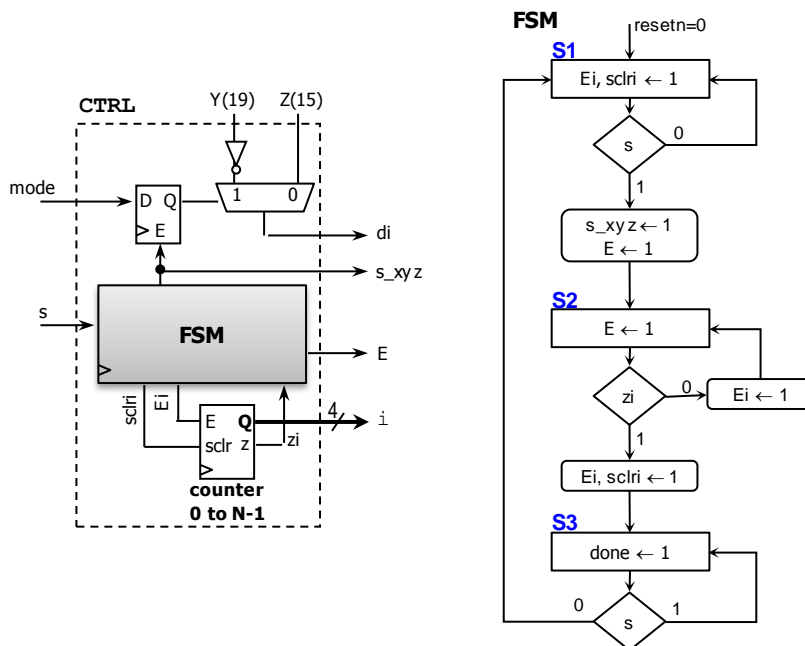- Format [16 14] applied to the LUT angles: We found that the optimal number of iterations is $N = 14$, since $Tan^{-1}(2^{-15}) = Tan^{-1}(2^{-14}) = 0$. If we use $N > 14$, Z will remain constant, and X, Y will update for a few more iterations (this depends on the guard bits). In the figure, we use 4 bits to represent the count from 0 to N-1.
- The format [16 14] was selected for X, Y, Z based on software simulations:
  - ✓ Rotation: Getting $sin(z_0)$ and $cos(z_0)$:
    - ▫ Inputs: $x_0 = y_0 = 1/An$, $z_0 \in [-\pi/2, \pi/2]$
    - ▫ Outputs: $x_N, y_N \in [-\sqrt{2}, \sqrt{2}]$, $z_N = 0$. Note: some intermediate values can be larger than outputs.
  - ✓ Vectoring: getting $atan2(1, y_0) = atan2(y_0/1)$
    - ▫ Inputs: $x_0 = 1, z_0 = 0$, $y_0 \in [-0.6, 0.6]$
    - ▫ Outputs: $x_N \in [0, 1.92]$, $z_N \in [-0.5404, 0.5404]$ $y_N = 0$. Note: some intermediate values can be larger than outputs

- Timing Diagram (N=14):
  - ✓ Input data: $xin = x_0$, $yin = y_0$, $zin = z_0$.
  - ✓ Output data: $xout = x_{14}$, $yout = y_{14}$, $zout = z_{14}$.
  - ✓ Counter goes from 0 to 13. Once input data is loaded, circuit needs N=14 cycles to produce the result

SPECIAL TECHNIQUES

LUT (LOOK UP TABLE) APPROACH

- In computer architecture, whenever a function is to be evaluated, we usually implement the algorithm that computes that function on hardware (e.g. $sqrt, ln, exp$). We can always take advantage of the specific properties of the algorithm to optimize both speed and resource utilization.
- Another option is not to compute the function values, but rather to store the values themselves in a LUT (ROM-like architecture). In this case, the value is taken directly from the memory rather than computed. For certain scenarios and under certain constraints, this idea can lead to more efficient architectures (both in speed and resource consumption).
- In a LUT, the LUT contents are hardwired. A 4-to-1 LUT can be seen as a ROM with 16 addresses, each address holding one bit. It can also be seen as a multiplexor with fixed inputs. A 4-to-1 LUT can implement any 4-input logic function.

LARGER LUTS

- $NI-to-NO$ LUT: NI input bits, NO output bits. This circuit can be thought of as a ROM with 2^{NI} addresses, each address holding NO bits.
- A larger LUT can be built by building a circuit that allows for more LUT positions.
- Efficient method: A larger LUT can also be built by combining LUTs with multiplexers as shown in the figure. We can build a $NI-to-1$ LUT with this method.
- We can build a $NI-to-NO$ LUT using NO $NI-to-1$ LUTs.

- You can implement any function using any desired format (e.g.: integer, fixed-point, dual fixed-point, floating point): $y = f(x)$, where y is represented with NO bits, and x with NI bits.

- The amount of resources increases linearly with the number of output bits (NO). However, the amount of resources grow exponentially with the number of input bits (NO). Thus, this approach is only efficient for small input data sizes (≤ 12 in modern FPGAs).

DISTRIBUTED ARITHMETIC

- This is a useful technique to implement inner product when one of the vectors is constant:

$$y = \sum_{k=0}^{N-1} h[k]x[k]$$

- If the coefficients $h[k]$ are known a priori, then the partial product term $h[k]x[k]$ becomes a multiplication with a constant. The Distributed Arithmetic Technique takes advantage of this fact:

$$y = \sum_{k=0}^{N-1} h[k]x[k] = h[0]x[0] + h[1]x[1] + h[2]x[2] + \cdots + h[N-1]x[N-1]$$

DISTRIBUTED ARITHMETIC – UNSIGNED INTEGER NUMBERS

- Each $x[k]$ value is an unsigned number with B bits: $x[k] = x_{B-1}[k]x_{B-1}[k] \dots x_0[k]$

$$x[k] = \sum_{b=0}^{B-1} x_b[k] \times 2^b \, , x_b[k] \in \{0,1\}$$

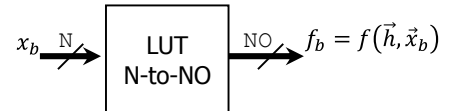where $x_b[k]$ denotes the b$^{\text{th}}$ bit of $x[k]$ (with B bits). Then:

$$y = \sum_{k=0}^{N-1} h[k]x[k] = \sum_{k=0}^{N-1}\left(h[k]\sum_{b=0}^{B-1} x_b[k] \times 2^b \right)$$

$$\begin{aligned}
y = \; & h[0](x_{B-1}[0]2^{B-1} + x_{B-2}[0]2^{B-2} + \cdots + x_0[0]2^0) + \\
& h[1](x_{B-1}[1]2^{B-1} + x_{B-2}[1]2^{B-2} + \cdots + x_0[1]2^0) + \\
& \dots + \\
& h[N-1](x_{B-1}[N-1]2^{B-1} + x_{B-2}[N-1]2^{B-2} + \cdots + x_0[N-1]2^0) +
\end{aligned}$$

- The summation can be rewritten as follows:

$$\begin{aligned}
y = \; & (h[0]x_{B-1}[0] + h[1]x_{B-1}[1] + \cdots + h[N-1]x_{B-1}[N-1]) \times 2^{B-1} + \\
& (h[0]x_{B-2}[0] + h[1]x_{B-2}[1] + \cdots + h[N-1]x_{B-2}[N-1]) \times 2^{B-2} + \\
& \dots + \\
& (h[0]x_0[0] + h[1]x_0[1] + \cdots + h[N-1]x_0[N-1]) \times 2^0
\end{aligned}$$

$$y = \sum_{b=0}^{B-1}\left(2^b \times \sum_{k=0}^{N-1} h[k]x_b[k] \right) = \sum_{b=0}^{B-1}\left(2^b \times f(\vec{h},\vec{x}_b) \right)$$

$$f(\vec{h},\vec{x}_b) = \sum_{k=0}^{N-1} h[k]x_b[k] \, , \vec{h} = [\, h[0]\, h[1] \dots h[N-1]\,], \vec{x}_b = [\, x_b[0]\, x_b[1] \dots x_b[N-1]\,]$$

- Preferred implementation of $f(\vec{h},\vec{x}_b)$: A 2^N-word LUT preprogrammed to accept an N-bit input vector $\vec{x}_b$ and output $f(\vec{h},\vec{x}_b)$.

- To get y, each $f(\vec{h},\vec{x}_b)$ is weighted by 2^b and all the resulting values are added up.

DISTRIBUTED ARITHMETIC – SIGNED INTEGER NUMBERS

- Each $x[k]$ value is an signed number with $B+1$ bits: $x[k] = x_B[k]x_{B-1}[k]x_{B-1}[k] \dots x_0[k]$

$$x[k] = -2^B x_B[k] + \sum_{b=0}^{B-1} x_b[k] \times 2^b \, , x_b[k] \in \{0,1\}$$

where $x_b[k]$ denotes the b$^{\text{th}}$ bit of $x[k]$ (with $B+1$ bits). Then:

$$y = \sum_{k=0}^{N-1} h[k]x[k] = \sum_{k=0}^{N-1}\left(h[k] \times \left(-2^B x_B[k] + \sum_{b=0}^{B-1} x_b[k] \times 2^b \right) \right)$$

Using a similar procedure as in the unsigned case, the inner product can be rewritten as:

$$y = -2^B \times \sum_{k=0}^{N-1} h[k]x_B[k] + \sum_{b=0}^{B-1}\left(2^b \times \sum_{k=0}^{N-1} h[k]x_b[k] \right) = -2^B \times f(\vec{h},\vec{x}_B) + \sum_{b=0}^{B-1}\left(2^b \times f(\vec{h},\vec{x}_b) \right)$$

$$f(\vec{h},\vec{x}_b) = \sum_{k=0}^{N-1} h[k]x_b[k] \, , \vec{h} = [\, h[0]\, h[1] \dots h[N-1]\,], \vec{x}_b = [\, x_b[0]\, x_b[1] \dots x_b[N-1]\,]$$

- Preferred implementation of $f(\vec{h},\vec{x}_b)$: A 2^N-word LUT preprogrammed to accept an N-bit input vector $\vec{x}_b$ and output $f(\vec{h},\vec{x}_b)$. To get y, each $f(\vec{h},\vec{x}_b)$ is weighted by 2^b and all of the resulting values are added up. Note that when $b = B$, we change the sign of the operand. Alternatively, we can modify the LUT for $b = B$, so that it outputs $-f(\vec{h},\vec{x}_B)$. To get y, each $f(\vec{h},\vec{x}_b)$ is weighted by 2^b and all of the resulting values are added up.

HARDWARE IMPLEMENTATION

i) <u>Iterative Implementation</u>: We make use of a shift-adder as shown in the figure.
   - The vector $\vec{x}_b, b = 0,1,\dots B$ is fed into the 2^N-word LUT at each clock cycle.
   - Instead of shifting each intermediate output value $f(\vec{h},\vec{x}_b)$ by b bits (which demands an expensive barrel shifter), it is more appropiate to shift the accumulator content itself in each iteration one bit to the right.
   - The adder unit includes a add/sub control so that when $b = B$, it will subtract the $f(\vec{h},\vec{x}_B)$ from the current result.
   - This shift-adder implementation requires the use of N shift registers of $B + 1$ length.
   - Notice that for $B = 1$, we have: $f(\vec{h},\vec{x}_0) \times 2^{-1} - f(\vec{h},\vec{x}_1)$. For $B = 2$, we have $f(\vec{h},\vec{x}_0) \times 2^{-2} + f(\vec{h},\vec{x}_1) \times 2^{-1} - f(\vec{h},\vec{x}_2)$. For $B = 2$, we adjust the result at the end by multiplying everything by 2^2: $f(\vec{h},\vec{x}_0) + f(\vec{h},\vec{x}_1) \times 2^1 - f(\vec{h},\vec{x}_2) \times 2^2$. This requires no extra hardware.
   - A simpler option is to input the vector $\vec{x}_b$ starting from $b = B, B - 1, \dots, 0$.
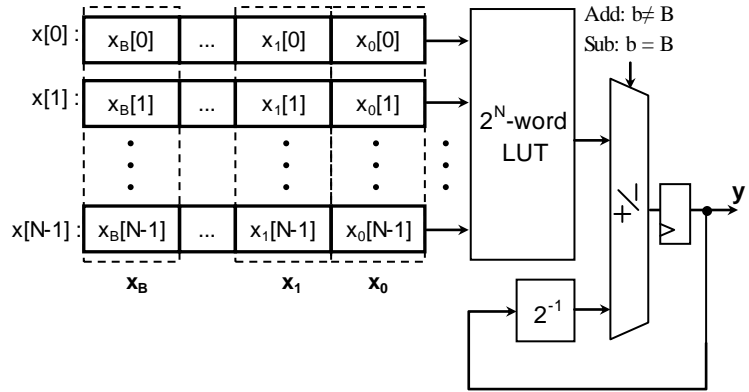
ii) <u>Fully parallel implementation</u>: We use an array of 2^N word LUTs as shown in the figure.
   - There are no shift registers here.
   - Each of the vectors $\vec{x}_b$ is fed to a 2^N-word LUT. As a result, we use $B + 1$ 2^N-word LUTs.
   - The output of each 2^N-word LUT is multiplied by its correspondent 2^b.
   - To account for the negative sign in $f(\vec{h},\vec{x}_B)$, we multiply it by -2^B. Another option is to modify the LUT so that when $b = B$ it outputs $- f(\vec{h},\vec{x}_B)$.
   - All the LUT outputs are weighted by 2^b and added into a final result.
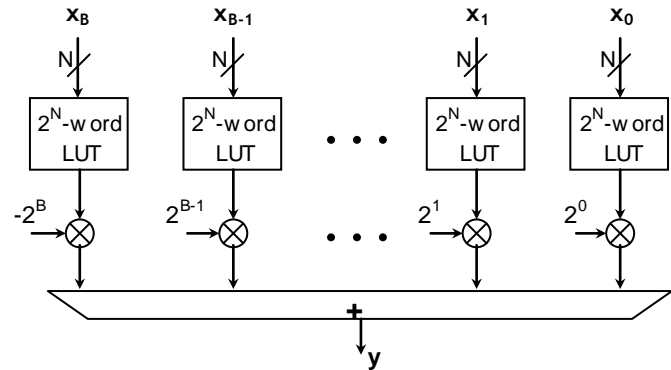
MODIFIED DA IMPLEMENTATION

- The LUT implementation becomes prohibitively expensive when N is large (if N = 32 $\rightarrow$ the LUT has 2^{32} words = 4G words!!!). A solution is to divide the inner product into inner product with L terms, i.e. we have N/L inner products of L terms, as follows:

$$y = \sum_{k=0}^{L-1} h[k]x[k] + \sum_{k=L}^{2L-1} h[k]x[k] + \sum_{k=2L}^{3L-1} h[k]x[k] + \cdots + \sum_{k=\left(\frac{N}{L}-1\right)L}^{N-1} h[k]x[k]$$
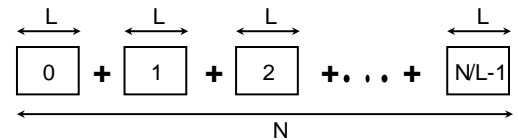
- Each of the N/L summations is transformed to DA form, and then computed in parallel. Finally, we add up all the resulting N/L values. With this in mind, we reformulate the 2 basic implementations:

i) <u>Iterative Implementation</u>: Here we use N/L 2^L-word LUTs. A vector $\vec{x}_b$ ($0 \le b \le B$) is fed into the LUT at each clock cycle. All LUTs outputs are accumulated; the final result goes through a shift-adder unit. The table illustrates the resource savings.

Iterative DA implementation	LUT Size	Total space required
No division in filter blocks	2^N words	2^N words
Division into N/L filter blocks	2^L words	$2^L \times (N/L)$ words

As an example, consider $N = 32, L = 4$. Then the original DA uses $2^{32} = 4G\ words$, while the Modified DA uses $2^4 \times \frac{32}{4} = 128\ words$. This is a vast improvement at the expense of one extra adder tree.

ii) <u>Fully Parallel Implementation</u>: The output of each of the N/L filter blocks is computed as in the case of Figure 6. The only diference is that the $\vec{x}_b$ vectors are of L bits; each of these vectors is fed into a 2^L-word LUT (we use $B + 1$ 2^L-word LUTs per filter block). Finally the N/L filter block outputs are added in parallel. The following table illustrates the resource savings.

LUT SPACE COMPARISONS – FULLY PARALLEL IMPLEMENTATION

Implementation	LUT Size	Total space required
No division in filter blocks	$2^N \times (B + 1)$ words	$2^N \times (B + 1)$ words
Division into N/L filter blocks	$2^L \times (B + 1)$ words	$2^L \times (B + 1) \times (N/L)$ words

As an example, consider N = 32, L = 4, B = 11. Then the original DA uses $2^{32} \times (11+1) = 48G\ words$, while the Modified

DA uses $2^4 \times (11+1) \times \dfrac{32}{4} = 1536\ words$. This is vast improvement.

Iterative Modified DA Implementation.
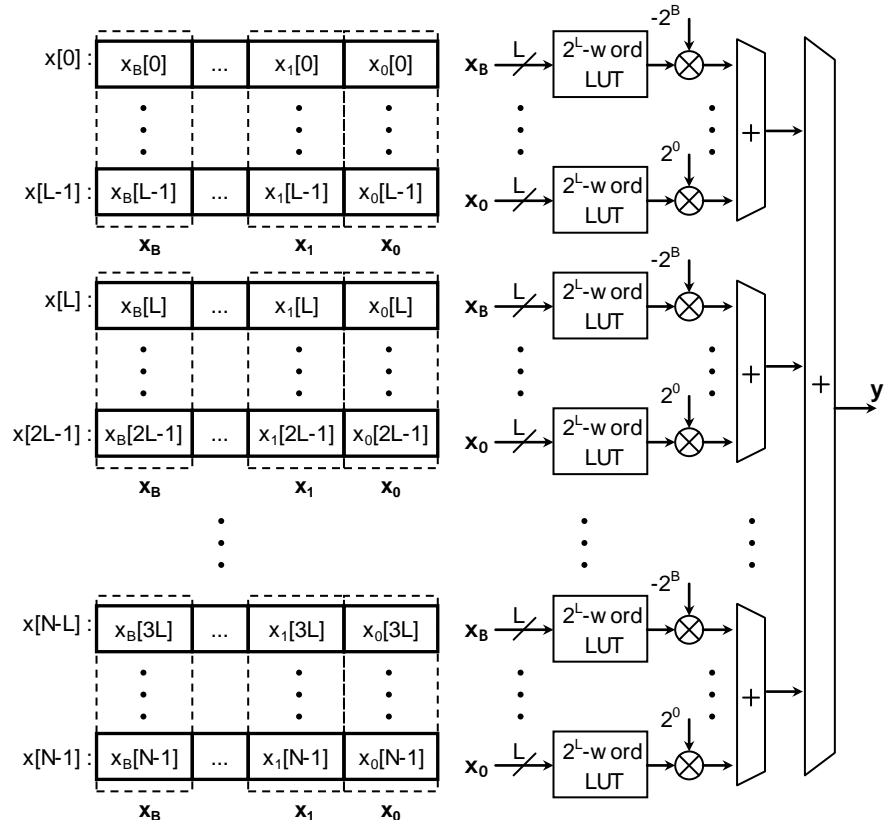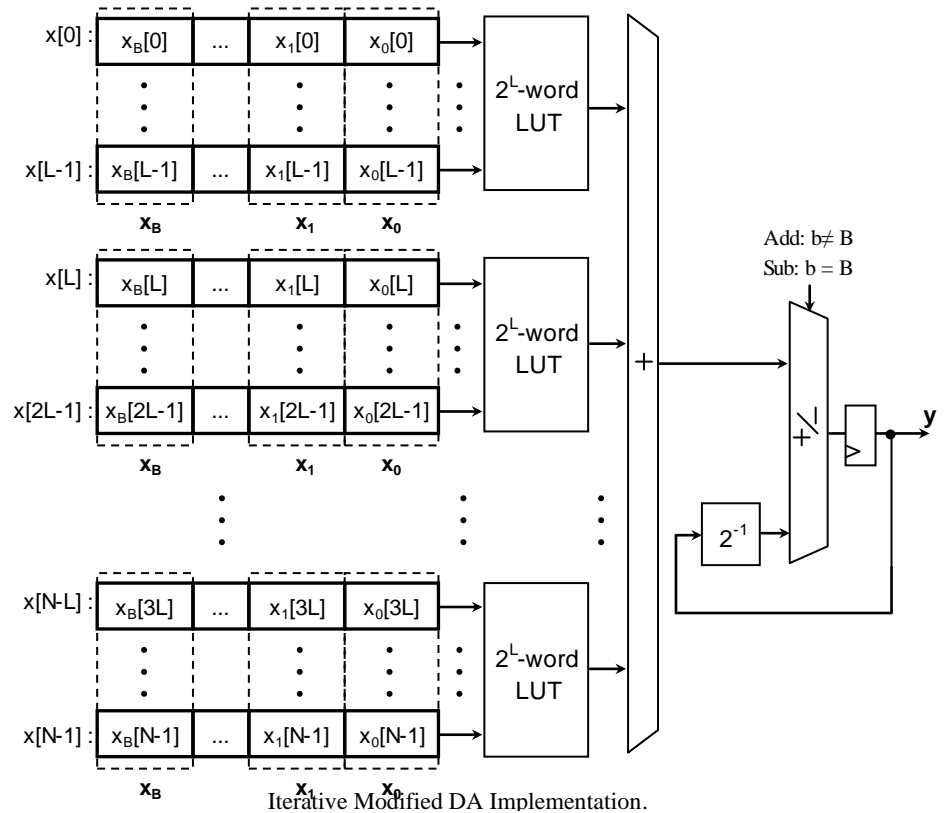
- Fixed-point considerations: The format of every stage differs from that of the input.

- Applications: non-symmetric, symmetric, anti-symmetric FIR filters, DCT, HEVC Transform.

Fully Parallel Modified DA Architecture