Special-Purpose Arithmetic Circuits and Techniques

ARITHMETIC CIRCUITS

INTEGER/FX MULTIPLICATION

UNSIGNED MULTIPLICATION

- Sequential algorithm:

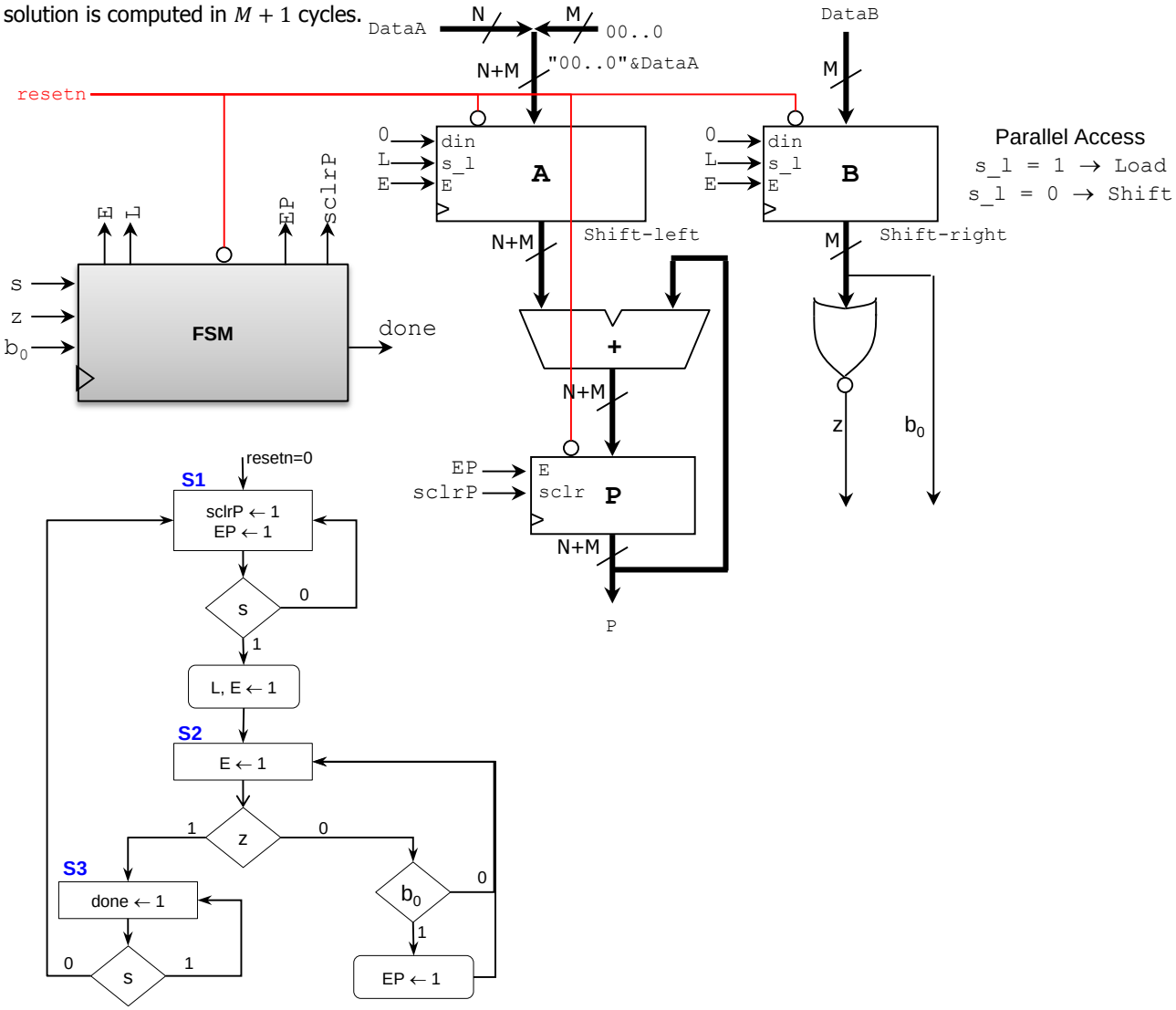
```
P ← 0, Load A,B
while B ≠ 0
   if b0 = 1 then
      P ← P + A
   end if
   left shift A
   right shift B
end while
```

Example:

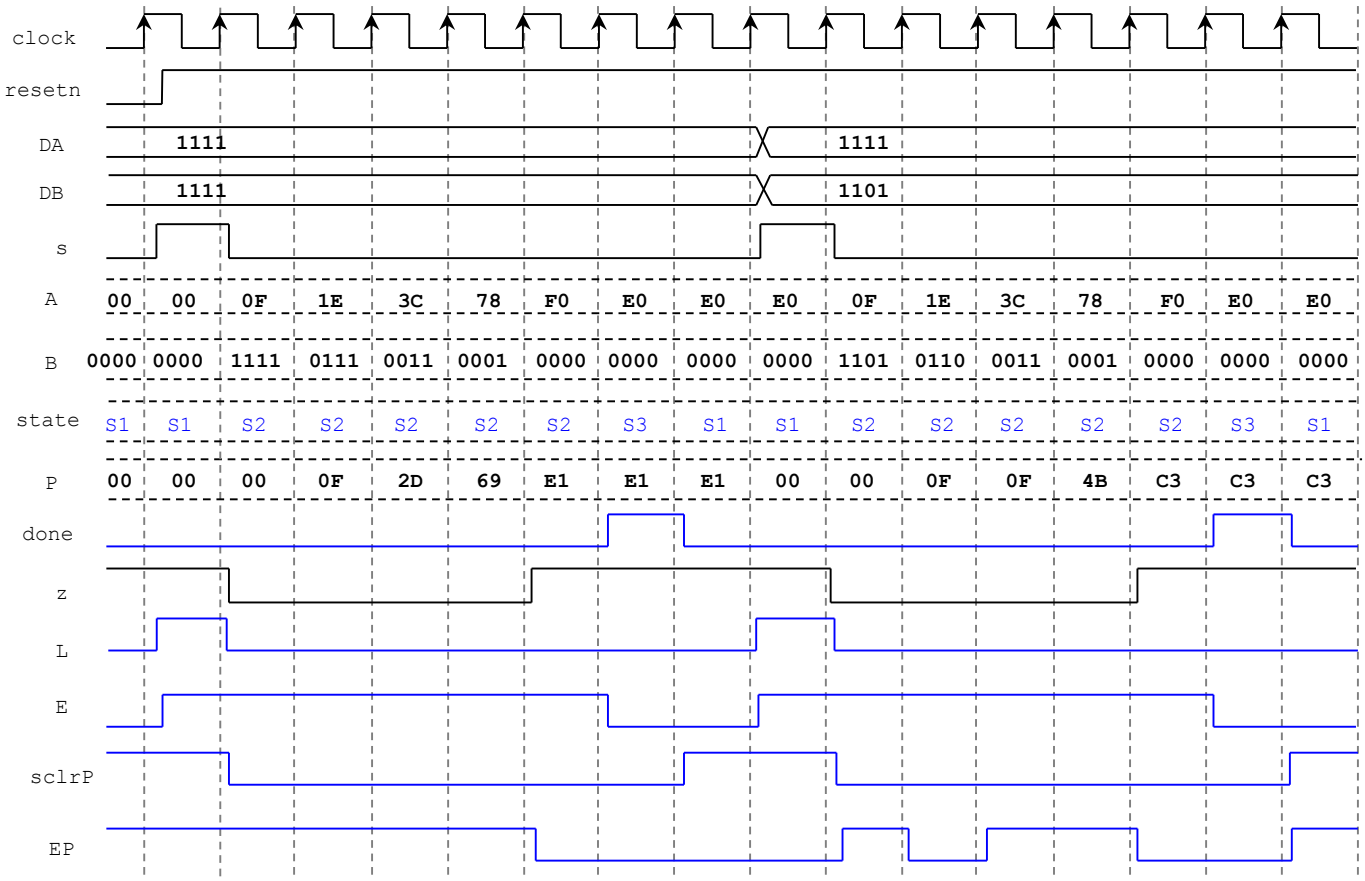
```
  1 1 1 1 x
  1 1 0 1
  ---------
  1 1 1 1   ⟶ P ← 0 + 1111
0 0 0 0     ⟶ P ← 1111
1 1 1 1     ⟶ P ← 1111 + 111100 = 1001011
1 1 1 1     ⟶ P ← 1001011 + 1111000 = 11000011
-----------
1 1 0 0 0 0 1 1
```

```
P ← 0, A ← 1111, B ← 1101
b0=1 ⇒ P ← P + A = 1111.          A ← 11110, B ← 110
b0=0 ⇒ P ← P = 1111.              A ← 111100, B ← 11
b0=1 ⇒ P ← P + A = 1111 + 111100 = 1001011.       A ← 1111000, B ← 1
b0=1 ⇒ P ← P + A = 1001011 + 1111000 = 11000011.  A ← 11110000, B ← 0
```

- Iterative Multiplier Architecture (N-bit by M-bit): 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 $M + 1$ cycles.

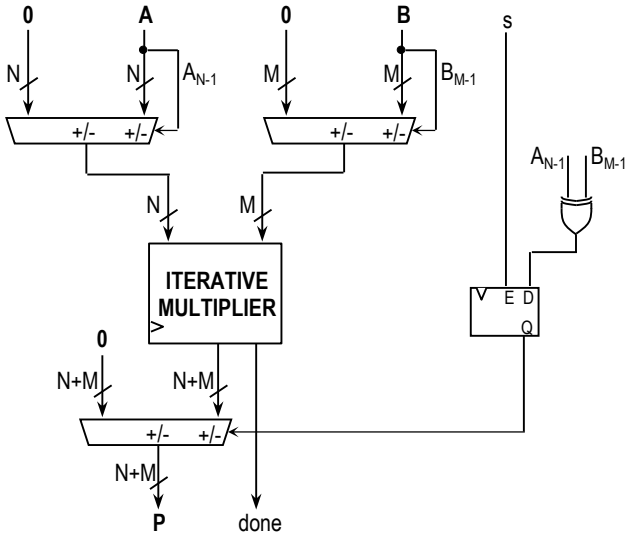


Example (timing diagram): N=M=4



SIGNED MULTIPLICATION

- Based on the iterative unsigned multiplier:



INTEGER/FX 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 $A = 10001100$; $B = 1001$.

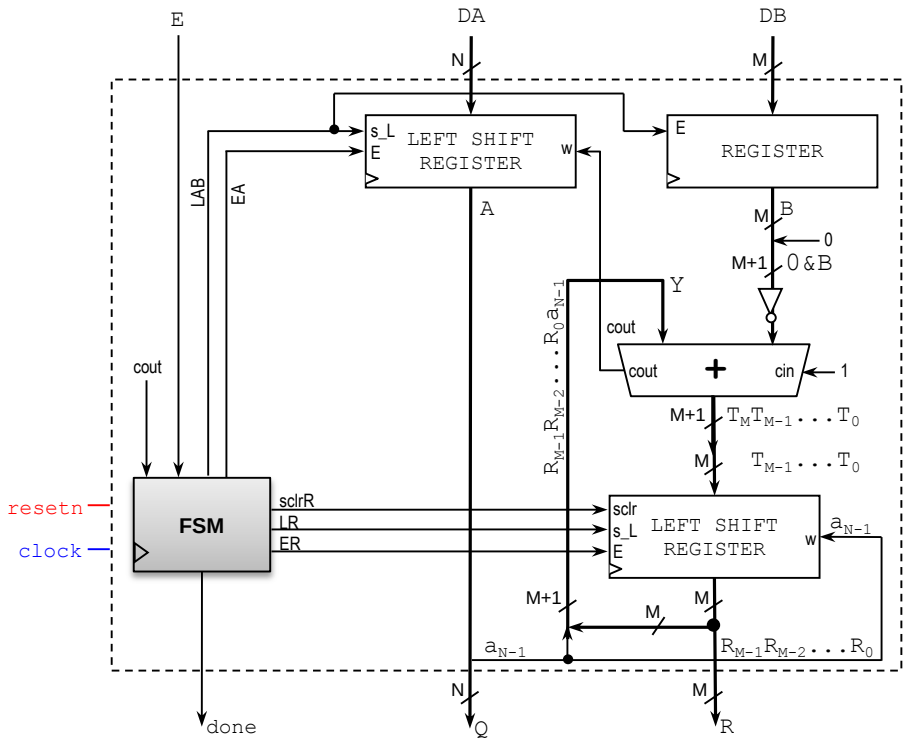
```
        15 ← Q                          00001111 ← Q
B → 9  ⟌ 140 ← A          B → 1001  ⟌ 10001100 ← A
         90                             1001
        ---                            -----
         50                            10001
         45                             1001
        ---                            -----
          5 ← R                        10000
                                        1001
                                       -----
                                        1110
                                        1001
                                       -----
                                         101 ← R
```

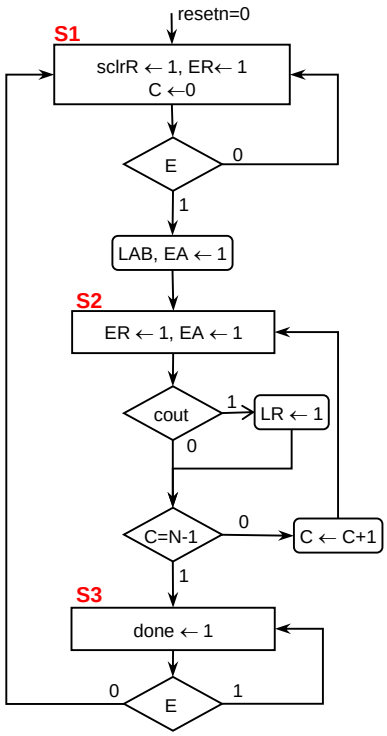
ALGORITHM

```
R = 0
for i = N-1 downto 0
    left shift R (input = a_i)
    if R ≥ B
       q_i = 1, R ← R-B
    else
       q_i = 0
    end
end
```

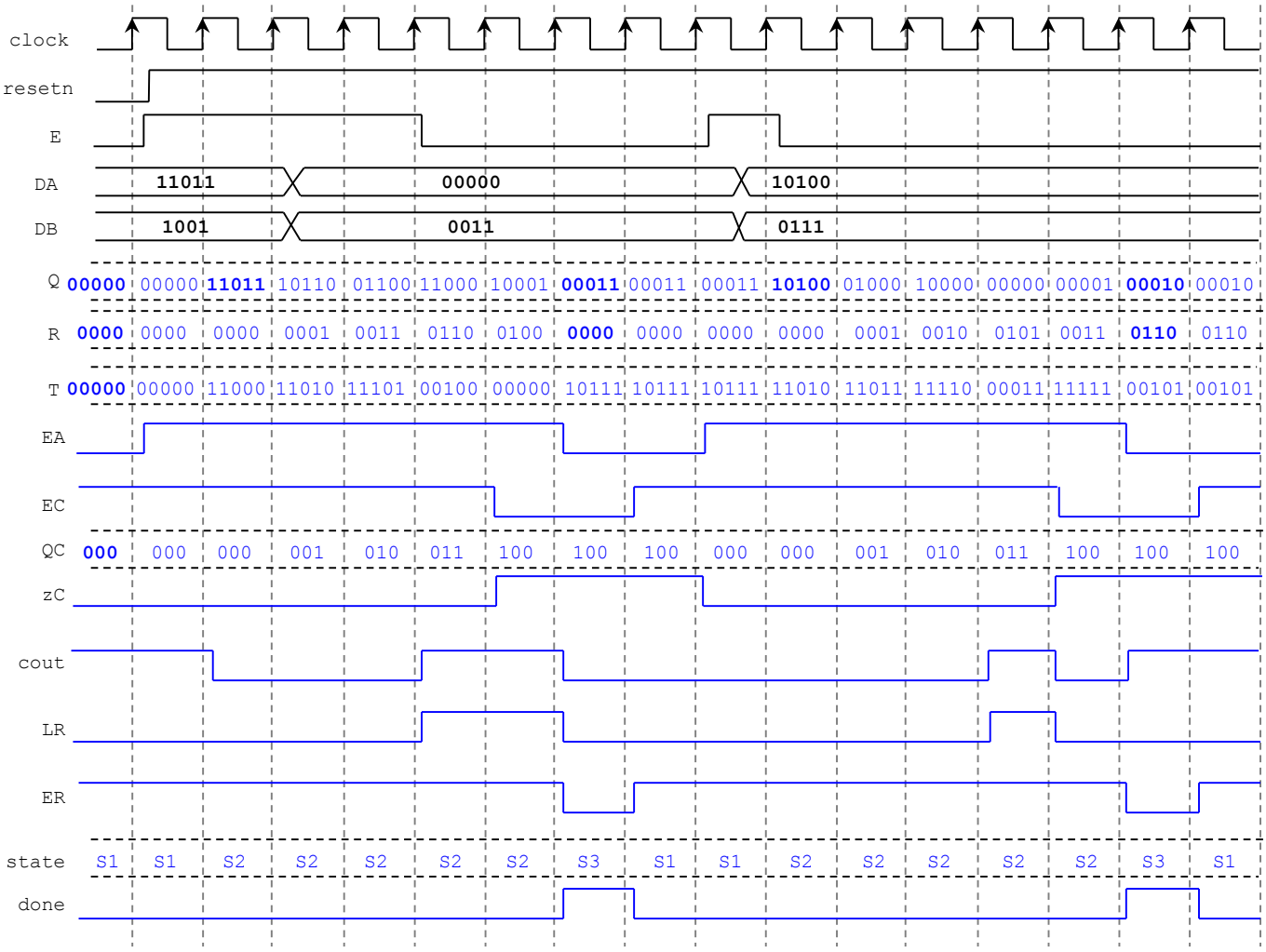
A: N=8 bits B: M=4 bits R: M=4 bits Intermediate subtraction requires M+1 bits Q: N=8 bits	A ← 10001100, B ← 1001, R ← 00000000 i = 7, a_7 = 1: R ← 00001 < 1001 ⇒ q_7 = 0 i = 6, a_6 = 0: R ← 00010 < 1001 ⇒ q_6 = 0 i = 5, a_5 = 0: R ← 00100 < 1001 ⇒ q_5 = 0 i = 4, a_4 = 0: R ← 01000 < 1001 ⇒ q_4 = 0 i = 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.



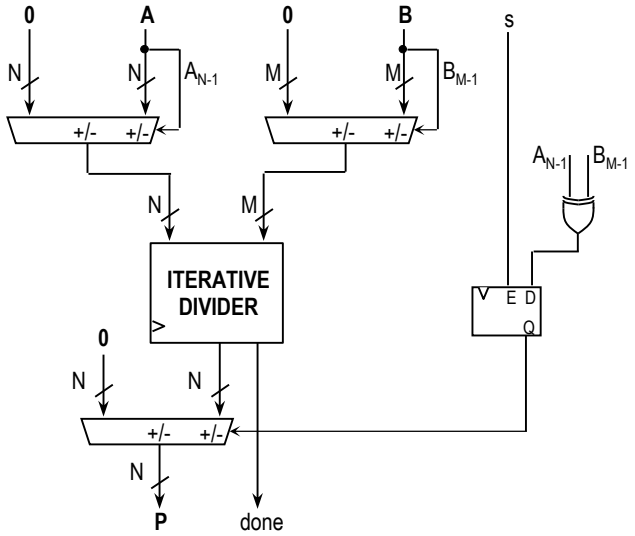


Example (timing diagram $N = 5, M = 4$). i) DA = 27, DB = 9, ii) DA = 20, DB = 7



SIGNED MULTIPLICATION

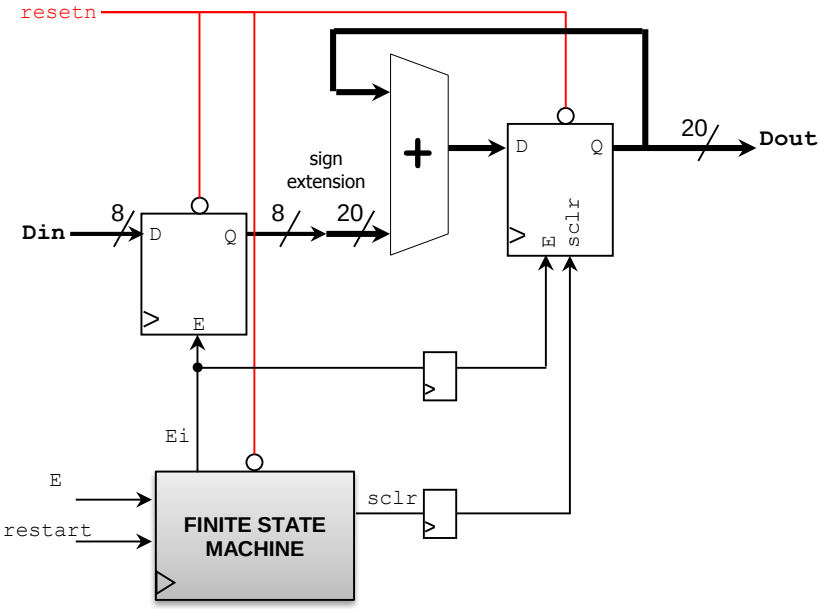
- Based on the iterative unsigned multiplier
  - ✓ Signed division: In this case, we first take the absolute value of the operators A and B. Depending on the sign of these operators, the division result (positive) of abs(A)/abs(B) might require a sign change.



INTEGER/FX ACCUMULATOR

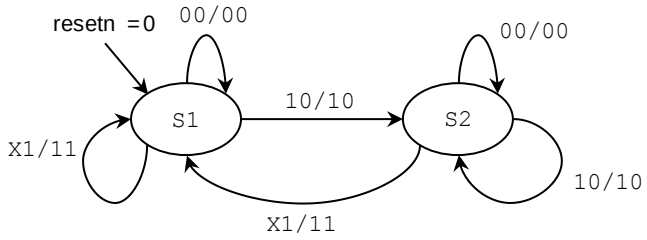
DIGITAL SYSTEM (FSM + Datapath circuit)

- `sclr`: Synchronous clear. If E = '1' and sclr = '1', then the output bits of the registers are set to zero.

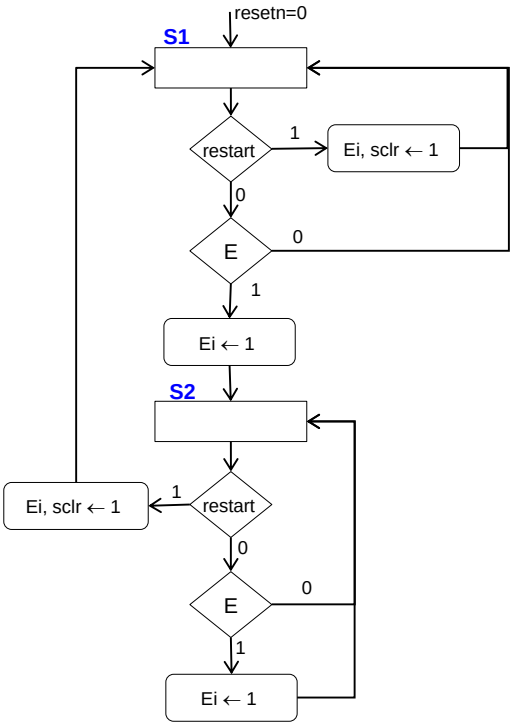


- **Finite State Machine (FSM)**:

E|restart/Ei|sclr



- **Algorithmic State Machine (ASM)**:



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	$\frac{s_2}{2^{\lvert e_1 - e_2 \rvert}}$
$e_1 < e_2$	1	e_2	$s_1 = b_1.f_1$	$s_2 = b_2.f_2$	$\frac{s_1}{2^{\lvert e_1 - e_2 \rvert}}$	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.

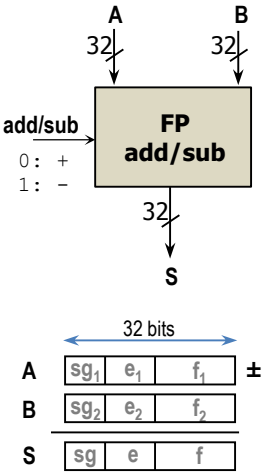
result	output	sign	Actions
$[0, p]$	$sh \in [0, p]$	0	The barrel shifter needs to shift to the left by sh bits. Exponent adder/subtractor needs to subtract sh from the exponent ep.
-1	$sh = 1$	1	The barrel shifter needs to shift to the right by 1 bit. Exponent adder/subtractor needs to add 1 to the exponent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

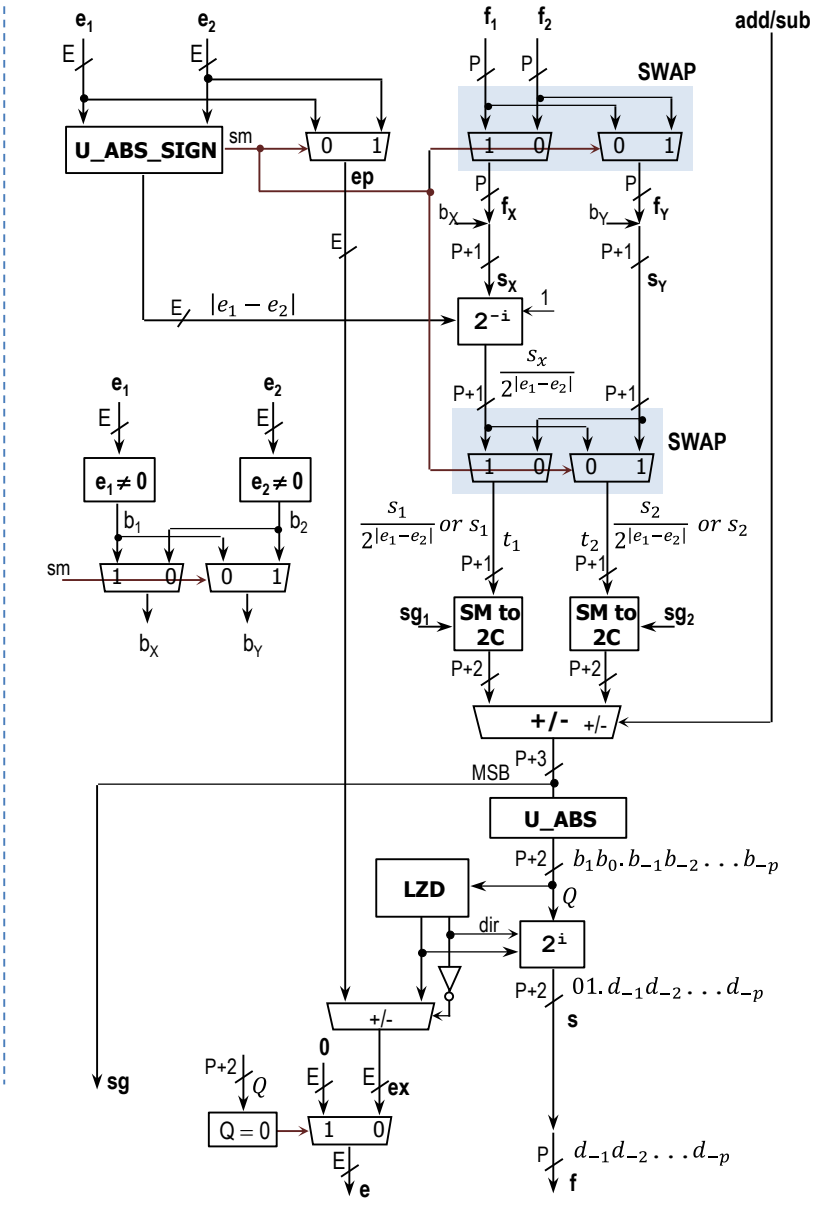
- **Exponent adder/subtractor**: The figure is not detailed. This circuit operates in 2C arithmetic; as the input operands are unsigned, we zero-extend to $E + 1$ bits. Note that for ordinary numbers, $ep \in [1, 2^E - 2]$. The $(E + 1)$-bit result (biased exponent) cannot be negative: at most, we subtract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

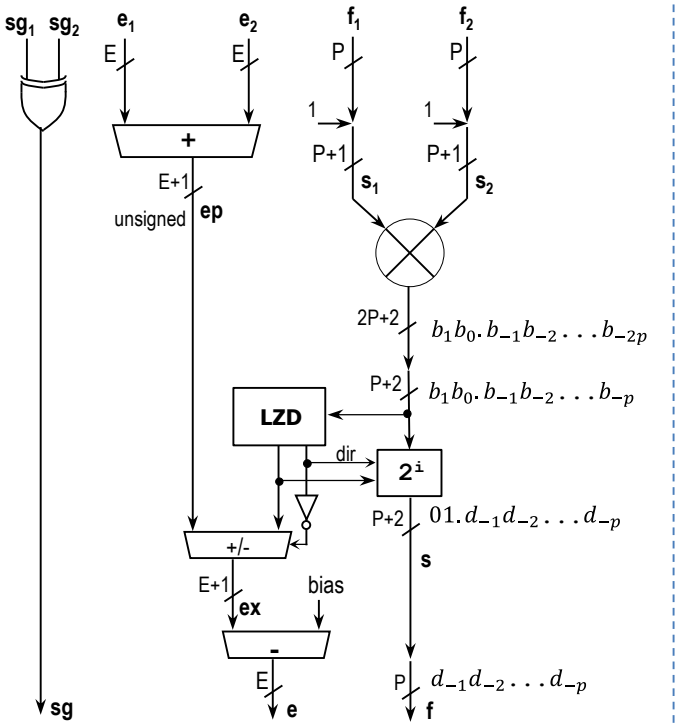




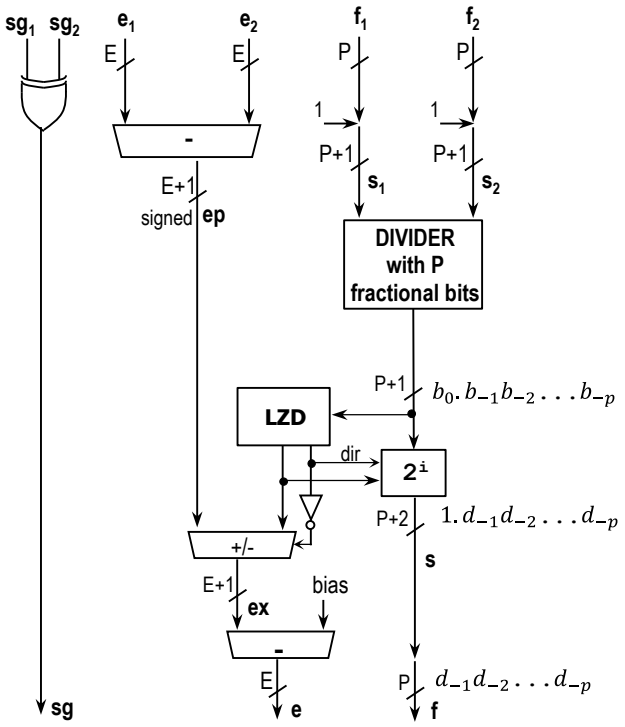
FLOATING POINT MULTIPLIER AND DIVIDER

- **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 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

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 ($i = 0, 1, 2, 3, \ldots$). 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 tend to the following values, for sufficiently large N:

Rotation Mode	Vectoring Mode
$x_n = A_n(x_0 cosz_0 - y_0 sinz_0)$ $y_n = A_n(y_0 cosz_0 + x_0 sinz_0)$ $z_n = 0$	$x_n = A_n\sqrt{x_0^2 + y_0^2}$ $y_n = 0$ $z_n = z_0 + tan^{-1}(y_0/x_0)$

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

- 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 $\rightarrow x_n = cosz_0$, $y_n = sinz_0$
  - ✓ $z_0 = 0, x_0 = 1$, vectoring mode $\rightarrow z_n = tan^{-1}(y_0)$
  - ✓ $x_0 = a, y_0 = b$, vectoring mode $\rightarrow x_n = A_n\sqrt{a^2 + b^2}$. 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 tend 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 $\rightarrow y_n = x_1 z_1$
  - ✓ $z_1 = 0$, vectoring mode $\rightarrow z_n = y_1/x_1$

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 tend to the following values, for sufficiently large N:

Rotation Mode	Vectoring Mode
$x_n = A_n(x_1 coshz_1 + y_1 sinhz_1)$ $y_n = A_n(y_1 coshz_1 + x_1 sinhz_1)$ $z_n = 0$	$x_n = A_n\sqrt{x_1^2 - y_1^2}$ $y_n = 0$ $z_n = z_1 + tanh^{-1}(y_1/x_1)$

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

- 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 $\rightarrow x_n = coshz_1$, $y_n = sinhz_1$
  - ✓ $z_1 = 0, x_1 = 1$, vectoring mode $\rightarrow z_n = tanh^{-1}(y_1)$
  - ✓ $x_1 = y_1 = 1/A_n$, rotation mode $\rightarrow x_n = y_n = coshz_1 + sinhz_1 = e^{z_1}$
  - ✓ $x_1 = \alpha + 1, y_1 = \alpha - 1, z_1 = 0$, vectoring mode $\rightarrow z_n = tanh^{-1}(\alpha - 1/\alpha + 1) = (\ln \alpha)/2$.
  - ✓ $x_1 = \alpha + 1/(4A_n^2), y_1 = \alpha - 1/(4A_n^2), z_1 = 0$, vectoring mode $\rightarrow x_n = \sqrt{\alpha}$

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:

Rotation Mode:	$\lvert z_{in}\rvert \le \theta_N + \sum_{i=i_{in}}^{N} \theta_i$	▫ Circular: $i_{in} = 0$, $z_{in} = z_0$, $\alpha_{in} = tan^{-1}(y_0/x_0)$ ▫ Linear: $i_{in} = 1$, $z_{in} = z_1$, $\alpha_{in} = y_1/x_1$ ▫ 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 $i = 4, 13, 40,$
Vectoring Mode:	$\lvert \alpha_{in}\rvert \le \theta_N + \sum_{i=i_{in}}^{N} \theta_i$	

- **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	$\lvert z_0\rvert \le 1.7433\ (99.9°)$	Input angle $\epsilon\ [-99.9°, 99.9°]$. Functions with angles outside this range can be computed by applying trigonometric identities.
Vectoring	$\lvert tan^{-1}(y_0/x_0)\rvert \le 1.7433\ (99.9°) \to y_0/x_0\ \epsilon\langle -\infty, \infty\rangle$	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	$\lvert z_1\rvert \le 1$	In both cases, there is a strict limitation on the input argument of the linear function (e.g. multiplication, division)
Vectoring	$\lvert y_1/x_1\rvert \le 1$	

- **Hyperbolic**:

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

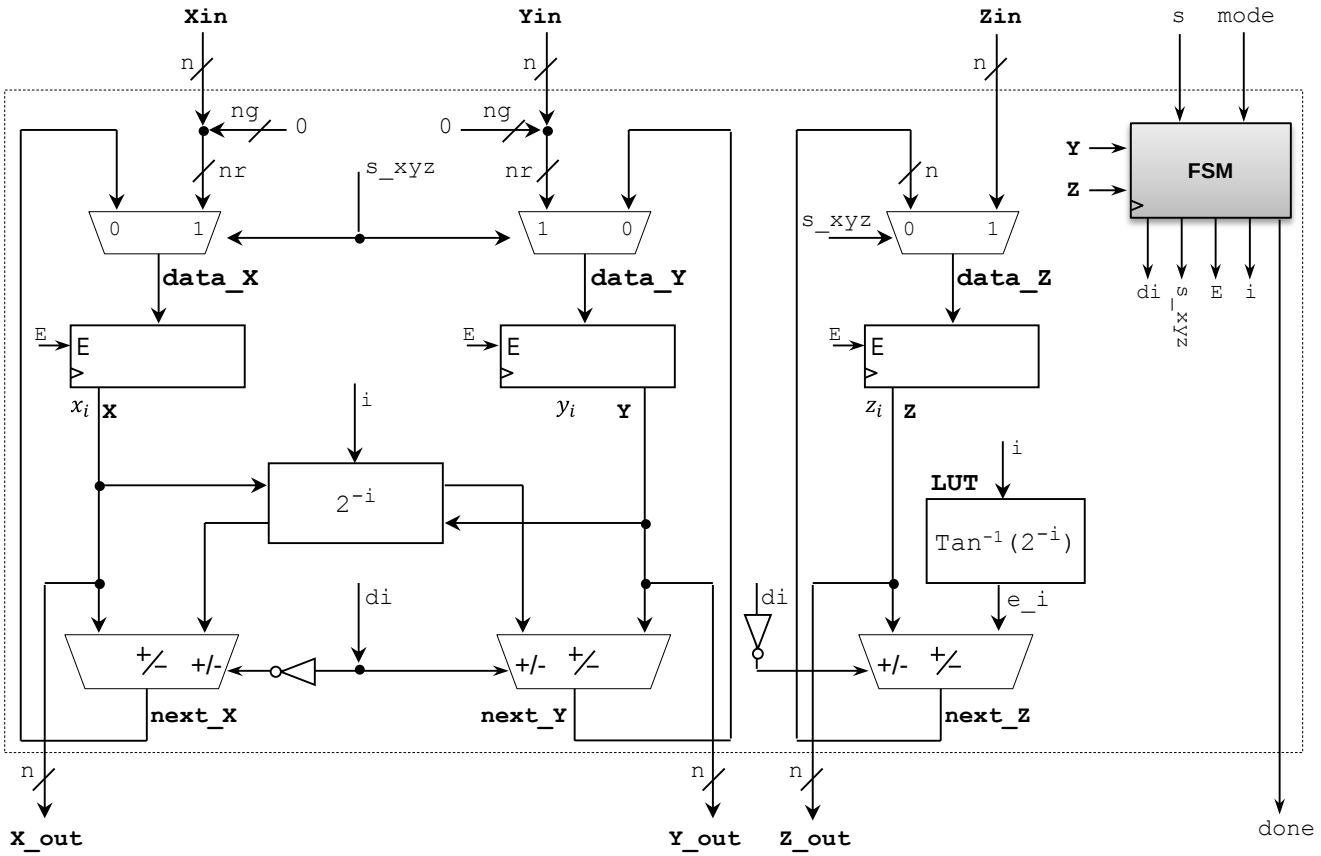
Rotation	$\lvert z_1\rvert \le 1.182$	This is the limitation imposed to the input argument of the hyperbolic functions. Note that the full domain of the functions $sinh$ and $cosh$ is $\langle -\propto, \propto\rangle$.
Vectoring	$\lvert tanh^{-1}(y_1/x_1)\rvert \le 1.182 \to \lvert y_1/x_1\rvert \le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

ITERATIVE ARCHITECTURE

- The architecture is 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 for a particular fixed-point representation.
  - n: input/output bit width
  - ng: additional guard bits on the LSB.
  - nr: $nr = ng + n$: bit width of the internal registers and operators
  - N: number of iterations ($i = 0,1, \dots, N$ for circular CORDIC, $i = 1, \dots, N$ for linear/hyperbolic CORDIC)
- x_i, y_i, z_i may require more bits than the final values. 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". A more accurate procedure is to perform software simulation for a given number of iterations and find out the number of bits required for proper representation of the x_i, y_i, z_i quantities.

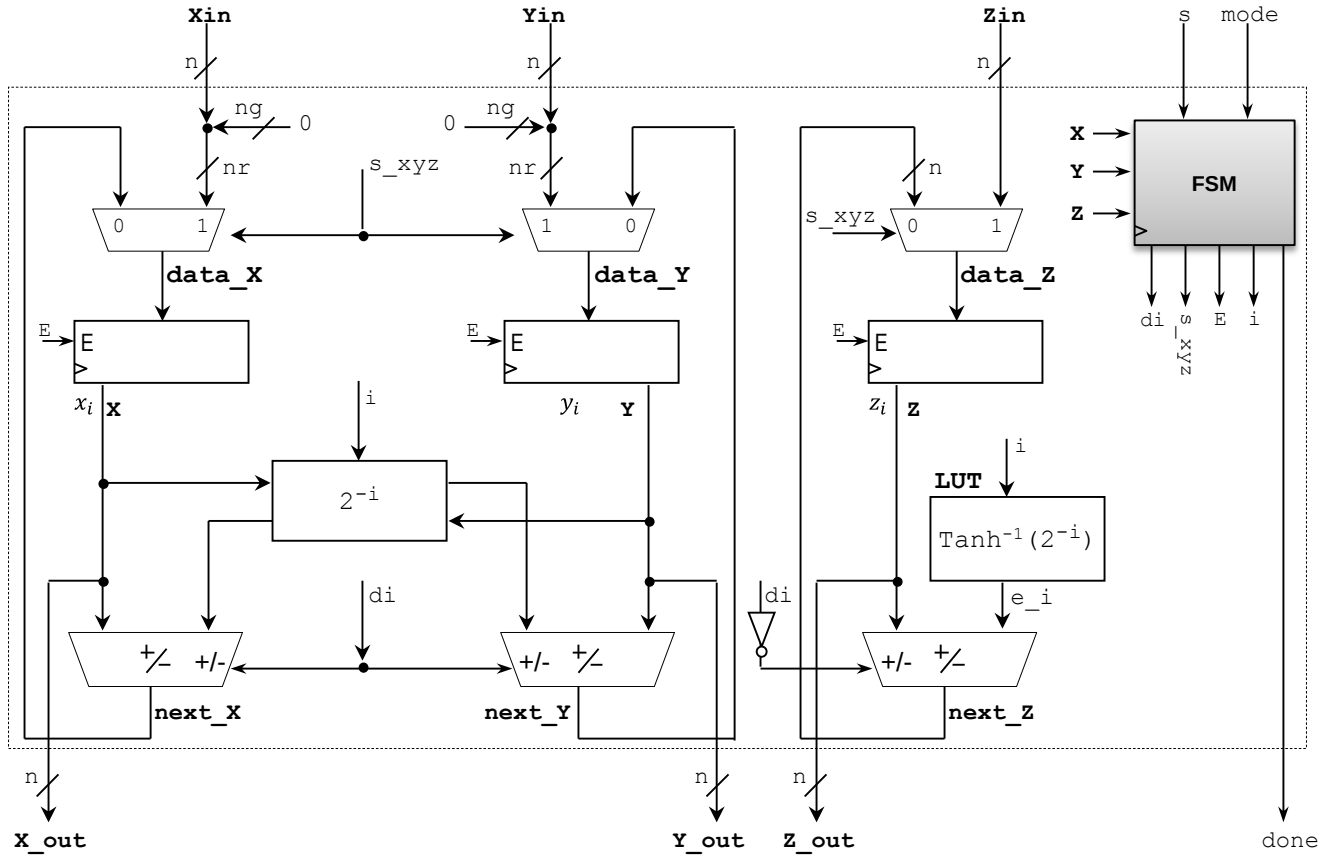
Circular CORDIC

- The figure below depicts the architecture that implements the circular CORDIC equations in an iterative fashion. The LUT (look-up table) is needed to store the sets of elementary angles $\theta_i = Tan^{-1}(2^{-i})$. The process begins when a start signal is asserted. After N clock cycles, the result is obtained in the registers X, Y and Z, and a new process can be started.
- A state machine, which 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.



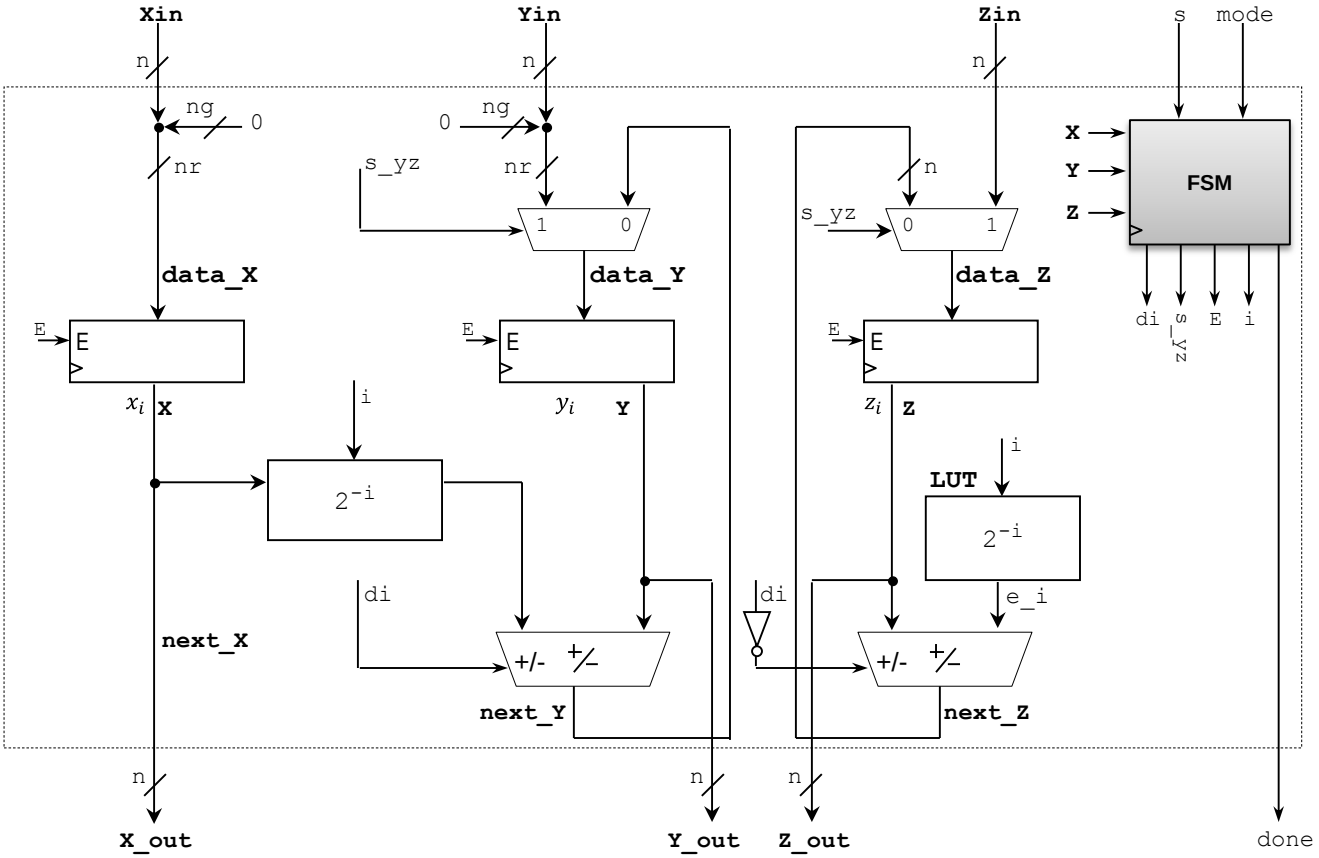
Hyperbolic CORDIC

- Here the LUT holds the $\theta_i = tanh^{-1}(2^{-i})$ values with $i = 1,2,\dots,N$. 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.



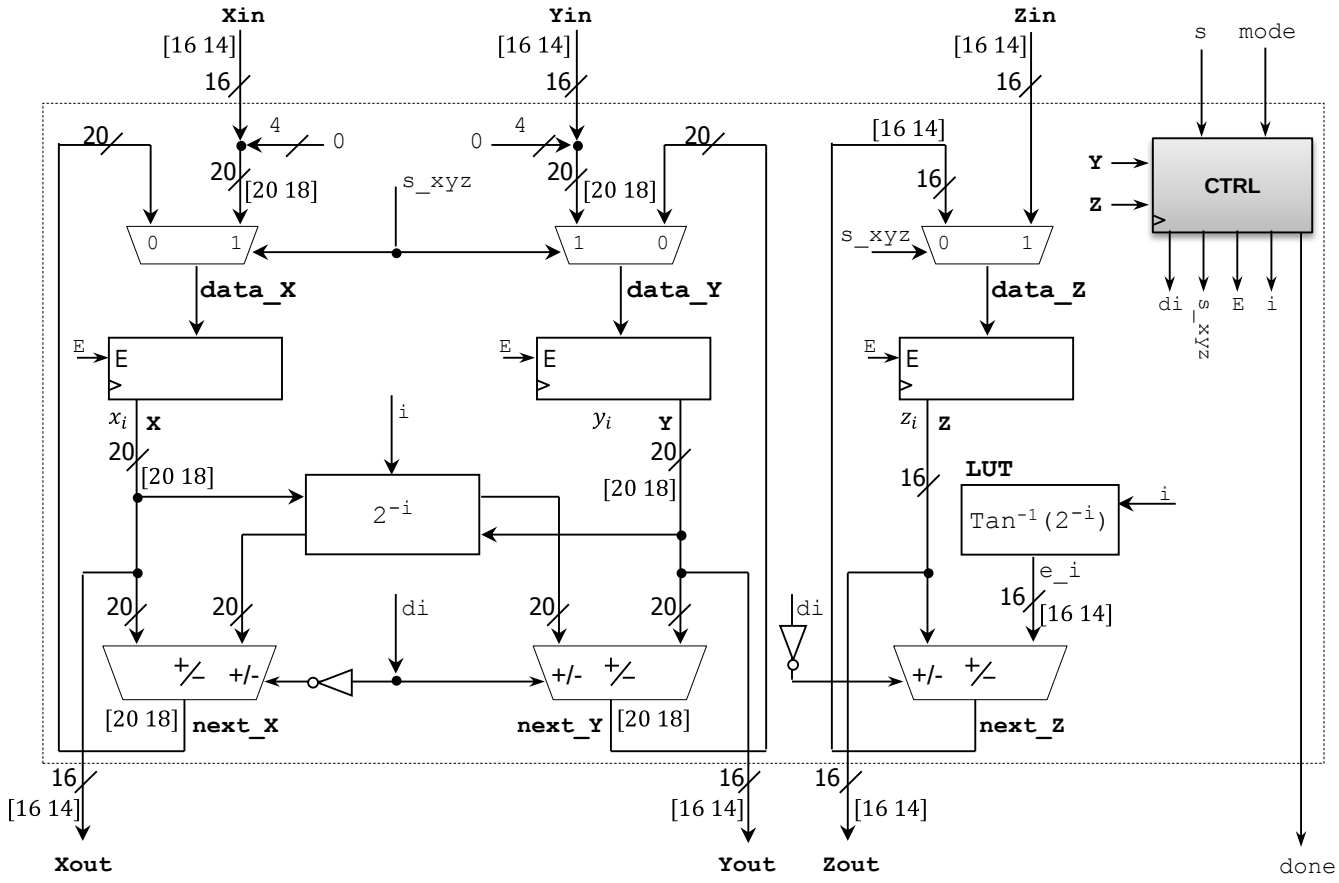
Linear CORDIC

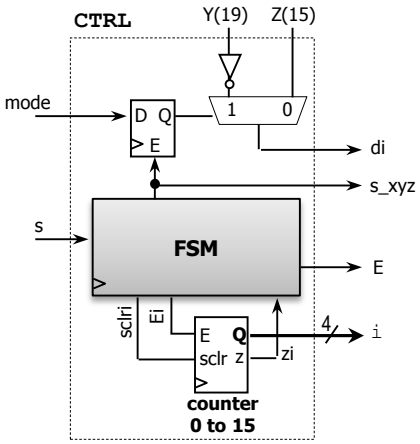
- Here the LUT holds the $\theta_i = 2^{-i}$ values with $i = 1,2,\dots,N$. After $N-1$ clock cycles, the result is obtained in the registers X, Y and Z, and a new process can be started. Note that we do not need an adder for x_i.

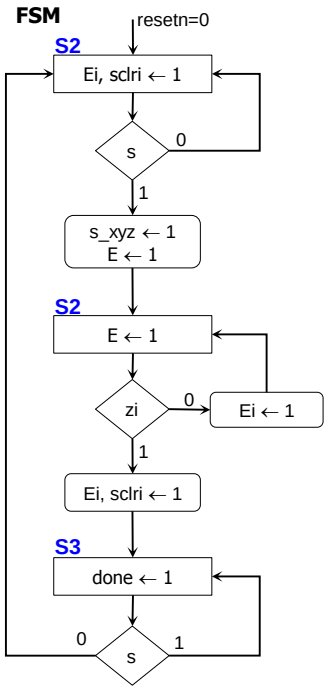


- Note that the architectures do not specify the numerical format we are using. We are free to use any format we desire (e.g.: fixed point, dual fixed point, floating point). The adders, barrel shifters, and LUT will change depending on the desired format. If an arithmetic unit requires more than one cycle to process its date, the FSM needs to account for this.

- **Example: FX CORDIC with [16 14]**







FIXED-POINT SQUARE ROOT

- Algorithms for hardware implementation amount to a 'binary search' and can be classified as Restoring and Non-Restoring. 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.

OPTIMIZED NON-RESTORING INTEGER SQRT 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 algorithms. A simple addition and subtraction is required based on the result bit generated in the previous iteration. No multipliers or multiplexors are needed. The result of the addition or subtraction is fed via registers to the next iteration directly even if it is negative.
- At the last iteration, if the remainder is non-negative, it is the precise remainder. Otherwise, we can get the precise remainder by an addition operation, but since it is rarely used, it is dismissed in order to reduce resource consumption.

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

We define: $Dk = d_{2n-1}d_{2n-2} \dots d_k, k = 0,1, \dots, n-1.$ D_{2k} has $2(n-k)$ bits.
$Q_k = q_{n-1}q_{n-2} \dots q_k, k = 0,1, \dots, n-1$ Q_k has $n-k$ bits.

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

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

- Estimated remainder: $R'_k = r'_n r'_{n-1} r'_{n-2} \dots r'_k$ (requires $n-k+1$ bits) The MSB (sign bit) determines the value of q_k (q_k is computed at each iteration). The R'_k value generated at each iteration is used in the next iteration even if it is negative (the 2C representation is used here). Note that the operands are always treated as unsigned numbers.
- Finally, in order to get the actual remainder $R = R_0$, only the $n+1$ LSBs of R'_0 are needed (the MSB determines q_0). In practice, the remainder is seldom needed.

Example:
$D = 0111111, n = 4, R = 00000, Q = 0000$
$k = n-1 = 3:\ R'_3 = 01 - 01 = 00, R'_3 = r'_4 r'_3 = 00, r'_3 \geq 0 \rightarrow q_3 = 1 \qquad Q = 1000$
$k = n-2 = 2:\ R'_2 = R'_3 11 - Q_3 01 = 0011 - 0101 = -10, R'_2 = r'_4 r'_3 r'_2 = 110, R'_2 < 0 \rightarrow q_2 = 0 \qquad Q = 1000$
When the subtraction result is < 0, we use the 2C representation with $n-k+1$ bits. The sign bit decides the value of q_k.
$k = 1:\ R'_1 = R'_2 11 + Q_2 11 = 11011 + 1011 = 100110, R'_1 = r'_4 r'_3 r'_2 r'_1 = 0110, R'_1 \geq 0 \rightarrow q_1 = 1 \qquad Q = 1010$
$k = 0:\ R'_0 = R'_1 11 - Q_1 01 = 011011 - 10101 = 00110, R'_0 = r'_4 r'_3 r'_2 r'_1 r'_0 = 00110, R'_0 \geq 0 \rightarrow q_0 = 1 \quad Q = 1011$
Also: $R = R'_0 = 00110$

ITERATIVE ARCHITECTURE

- The size of the elements (registers, adder/subtractor) will be:
  Register R: $n+1$ bits Register Q: n bits

  Adder/subtractor: $n+2$ bits. This is because the last iteration requires $R'_0 = \begin{cases} R'_1 d_1 d_0 - Q_1 01, if q_1 = 1 \\ R'_1 d_1 d_0 + Q_1 11, if q_1 = 0 \end{cases}$. R'_1 requires n bits, thus we need operands with $n+2$ bits. However, the result R'_0 only requires $n+1$ bits. Also, for the purposes of subtraction, the operands are treated as signed numbers.

  $(n+2)$-bit adder/subtractor: the 2 LSBs performs either $xy-01$ or $xy+11$, $xy = d_{2k+1}d_{2k}$. 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.

  ba depends only on xy, but c depends on the operation. However, a standard adder/subtractor with carry-in treats the carry-in as in positive logic when adding, and as in negative logic when subtracting. This allows us to re-define the truth table, where we invert c (for subtraction) in the truth table so that it works properly in the adder/subtractor with carry in:

  Now, c and ba depend only on 'xy': $c = x + y$, $b = \overline{x \oplus y}$, $a = y$

  This reduces the width of the adder/subtractor by 2 bits. ba, implemented with logic gates, is placed on the 2 LSBs of the register R', and the carry-in comes from the OR gate. Thus, we only need an adder/subtractor with n bits and a carry-in.

cba = xy + 11

xy	cba
00	011
01	100
00	101
01	110

cba = xy - 01

xy	cba
00	111
01	000
00	001
01	010

cba = xy + 11

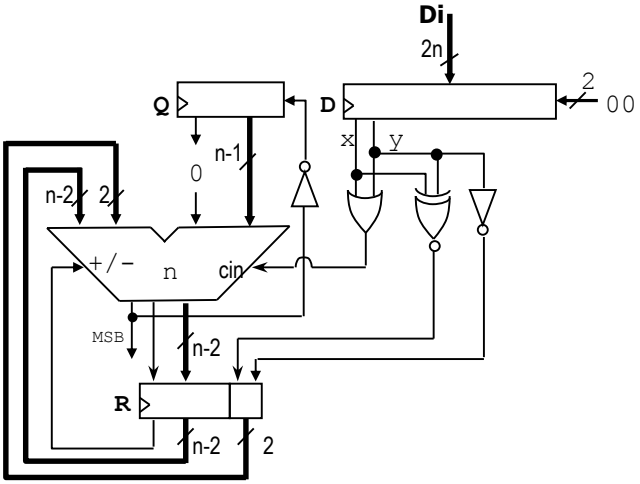
xy	cba
00	011
01	100
00	101
01	110

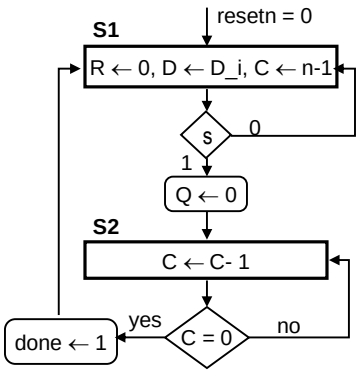
cba = xy - 01

xy	cba
00	011
01	100
00	101
01	110

Summation/subtraction operation

- $R'_{k+1}d_{2k+1}d_{2k}$. For $k=0$, this operator requires $n+2$ bits.
- $Q_{k+1}01$ or $Q_{k+1}11$. For $k=0$, this operator requires $n+1$ bits (Q_{k+1} requires $n-1$ bits) and we need to zero-extend it.
- Result R'_k: for $k=0$, it requires $n+1$ bits. So, for $k=0$, we only need $n-1$ bits (LSBs) out of the adder/subtractor (we get the other 2 bits from x and y). We use the MSB as the sign bit.





The process starts when s = '1'. After n clock cycles, the result appears in register 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

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

Hardware changes

- 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: $Q = 111.01 = 7.25 \approx \sqrt{55}$

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.
$\Rightarrow D = 1110111011 = 955$

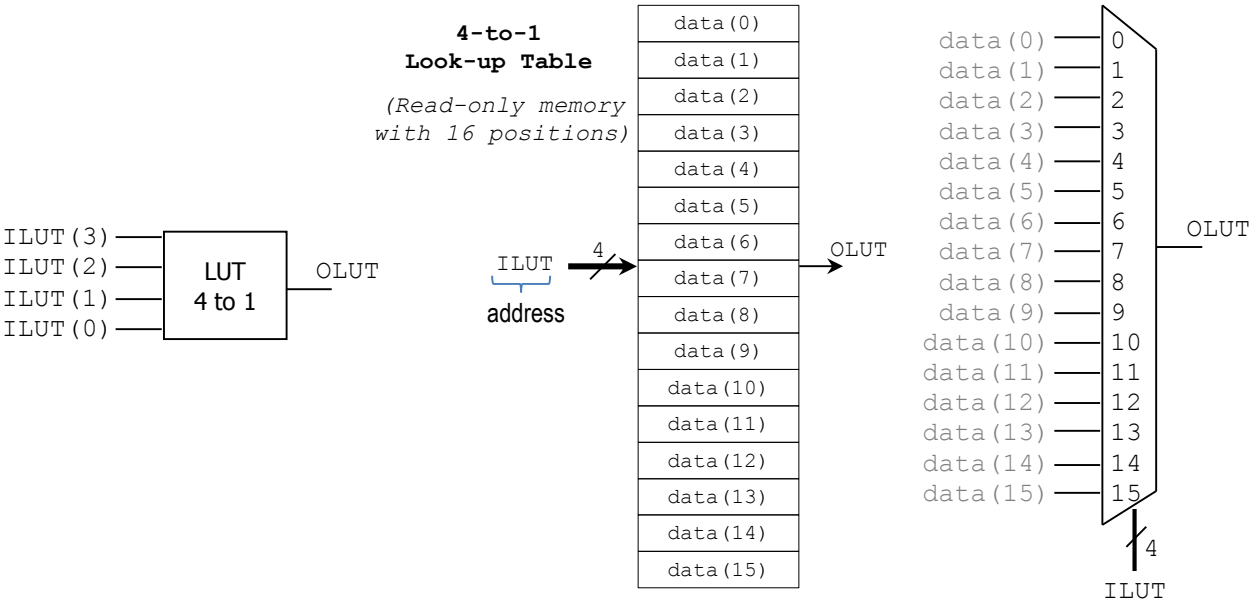
Step 2: Add (optionally) $2x = 4$ zeros
$\Rightarrow Dp = 11101110110000 = 15280$

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 ($p + x = 4$): $Qf = 111.1011 = 7.6875 \approx \sqrt{59.6875}$

SPECIAL TECHNIQUES

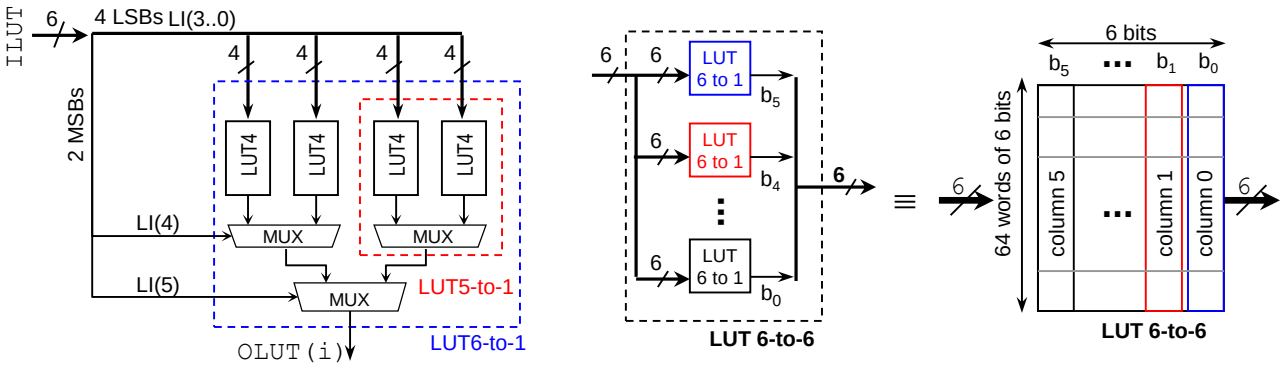
LUT (LOOK UP TABLE) APPROACH

- 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).