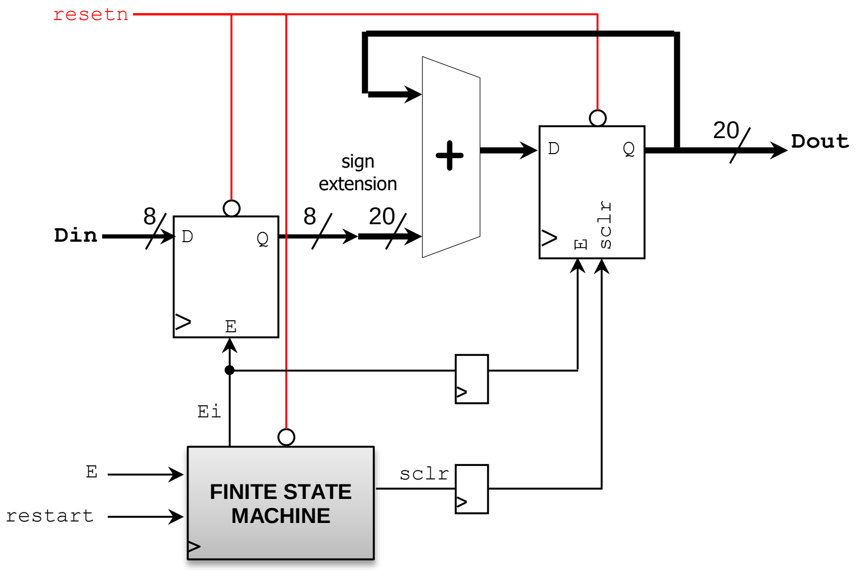
Unit 4- Special-Purpose Circuits and Techniques

ARITHMETIC CIRCUITS

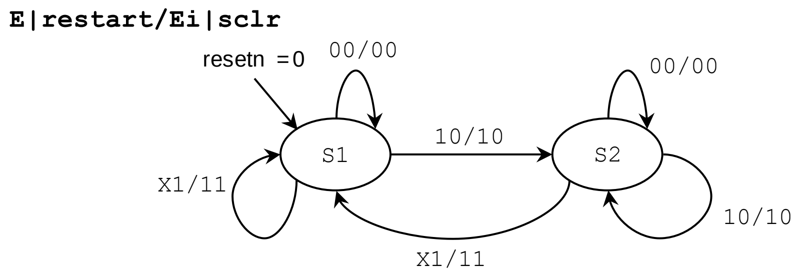
INTEGER/FX ACCUMULATOR

DIGITAL SYSTEM (FSM + Datapath circuit)

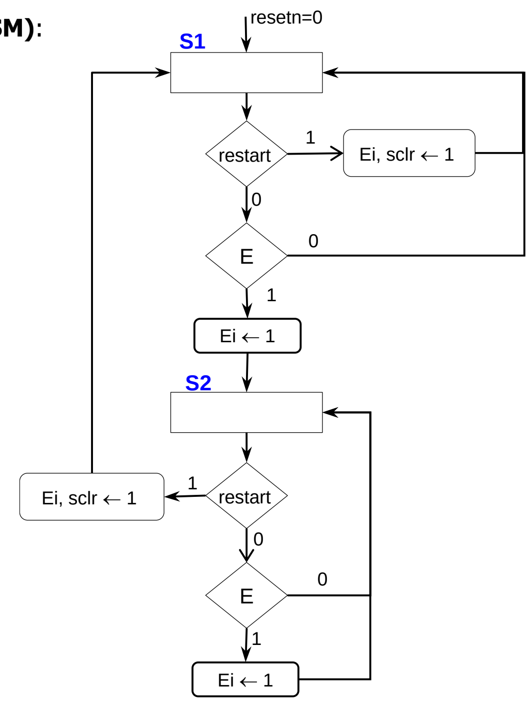
- `sclr`: Synchronous cllear. If E = '1' and sclr = '1', then the output bits of the registers are set to zero.



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



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



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 ($i = 0,1,2,3,\dots,N-1$). 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
$x_n = A_n(x_0 cosz_0 - y_0 sinz_0)$ $y_n = A_n(y_0 cosz_0 + x_0 sinz_0)$ $z_n = 0$	$x_n = A_n\sqrt{x_0^2 + y_0^2}$ $y_n = 0$ $z_n = z_0 + tan^{-1}(y_0/x_0)$

$A_n \leftarrow \prod_{i=0}^{N-1}\sqrt{1+2^{-2i}}$. For $N \to \propto$, $A_n = 1.647$. 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 ($i = 0,1,2,3,\dots,N-1$). 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 $\rightarrow x_n = cosz_0$, $y_n = sinz_0$
  - ✓ $z_0 = 0, x_0 = 1$, 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. ($i = 1,2,3,\dots$).

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

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

- 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 $\rightarrow y_n = x_1 z_1$
  - ✓ $z_1 = 0$, 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 ($i = 1,2,3,\dots$). The following iterations must be repeated to guarantee convergence: $i = 4, 13, 40, \dots, k, 3k+1$.

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

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

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

$A_n \leftarrow \prod_{i=1}^{N}\sqrt{1-2^{-2i}}$ (this includes the repeated iterations $i = 4, 13, 40, \dots$). 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 $\rightarrow x_n = coshz_1$, $y_n = sinhz_1$
  - ✓ $z_1 = 0, x_1 = 1$, 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 in:

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

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

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

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

Rotation	$\lvert z_1\rvert \leq 1.182$	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 $\langle -\propto, \propto\rangle$.
Vectoring	$\lvert tanh^{-1}(y_1/x_1)\rvert \leq 1.182 \rightarrow \lvert y_1/x_1\rvert \leq 0.807$	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 ($i = -M, \dots, -1$) and N positive iterations ($i = 0,1, \dots, N-1$). For sufficiently large N, the values of x_n, y_n, z_n converge to:

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

$A_n = \prod_{i=-M}^{N-1} \sqrt{1 + 2^{-2i}}$. 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 $N\to\propto$, $A_n=3.2935$
  - ✓ $N+2$ 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: $i=4,13,40,\dots,k,3k+1$.

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

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

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

- 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
$x_n=A_n(x_{in}coshz_{in}+y_{in}sinhz_{in})$ $y_n=A_n(y_{in}coshz_{in}+x_{in}sinhz_{in})$ $z_n=0$	$x_n=A_n\sqrt{x_{in}^2-y_{in}^2},\quad y_n=0$ $z_n=z_{in}+tanh^{-1}(y_{in}/x_{in})$

$A_n=\left(\prod_{i=-M}^{0}\sqrt{1-(1-2^{i-2})^2}\right)\prod_{i=1}^{N}\sqrt{1-2^{-2i}}$. 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	$coshx, sinhx, e^x$	$\ln x$
$Basic\ CORDIC$	$[-1.11820, 1.11820]$	$(0, 9.35958]$
0	$[-2.09113, 2.09113]$	$(0, 65.51375]$
1	$[-3.44515, 3.44515]$	$(0, 982.69618]$
2	$[-5.16215, 5, 16215]$	$(0, 3.04640\times10^4]$
3	$[-7.23371, 7.23371]$	$(0, 1.91920\times10^6]$
4	$[-9.65581, 9.65581]$	$(0, 2.43742\times10^8]$
5	$[-12.42644, 12.42644]$	$(0, 6.21539\times10^{10}]$
6	$[-15.54462, 15,54462]$	$(0,3.17604\times10^{13}]$
7	$[-19.00987, 19.00987]$	$(0, 3.24910\times10^{16}]$
8	$[-22.82194, 22.82194]$	$(0, 6.65097\times10^{19}]$
9	$[-26.98070, 26,98070]$	$(0, 2.72357\times10^{23}]$
10	$[-31.48609, 31.48609]$	$(0, 2.23085\times10^{27}]$

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): $|y \ln x| \leq \theta_{max}(M)$.

- 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)

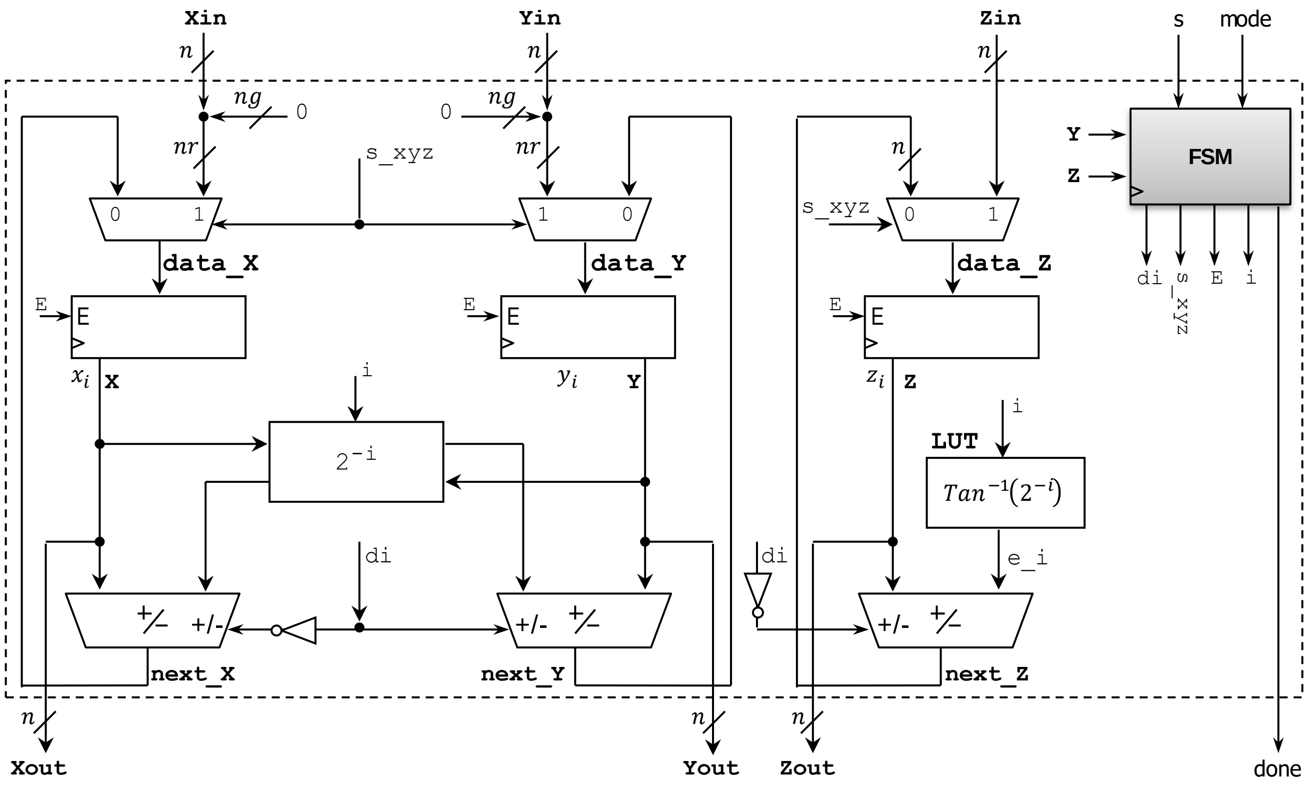
- The architectures shown here are such that the inputs and outputs have an identical bit width. We can reach an optimal number of iterations by noticing the iteration at which $\theta_i = Tan^{-1}(2^{-i})$ is equal to zero due to for a given fixed-point format.

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:	# 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 thorough software simulation for a given number of iterations and find out the format required for proper representation of the x_i, y_i, z_i.

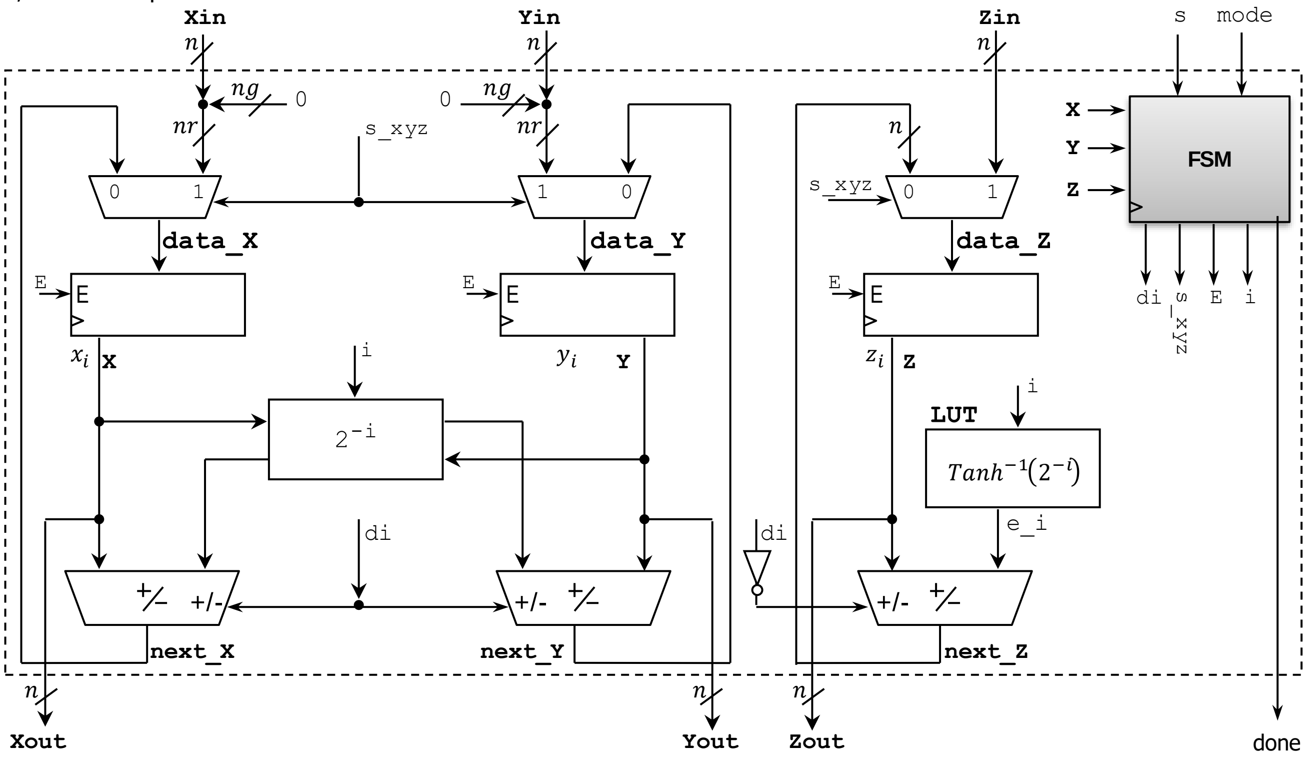
Circular CORDIC

- 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.



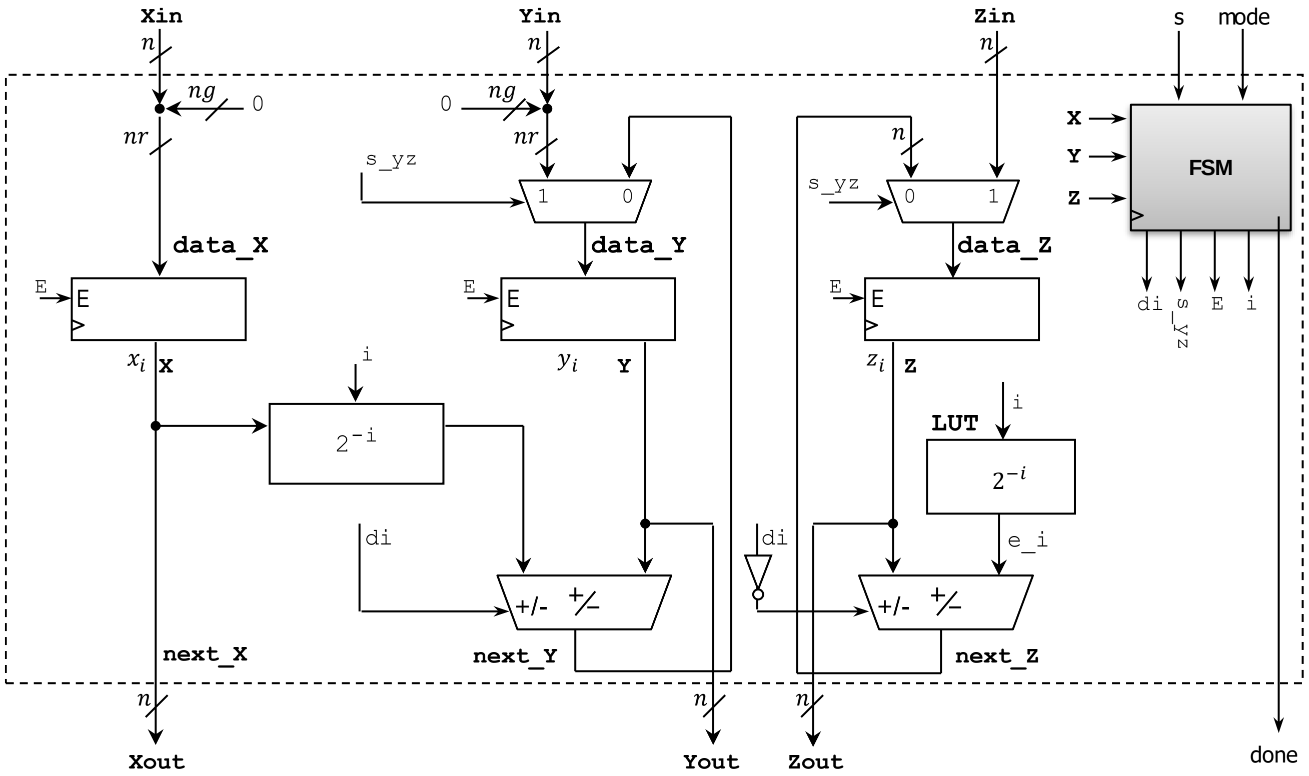
Hyperbolic CORDIC

- Here the LUT holds the $\theta_i = tanh^{-1}(2^{-i})$ values 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

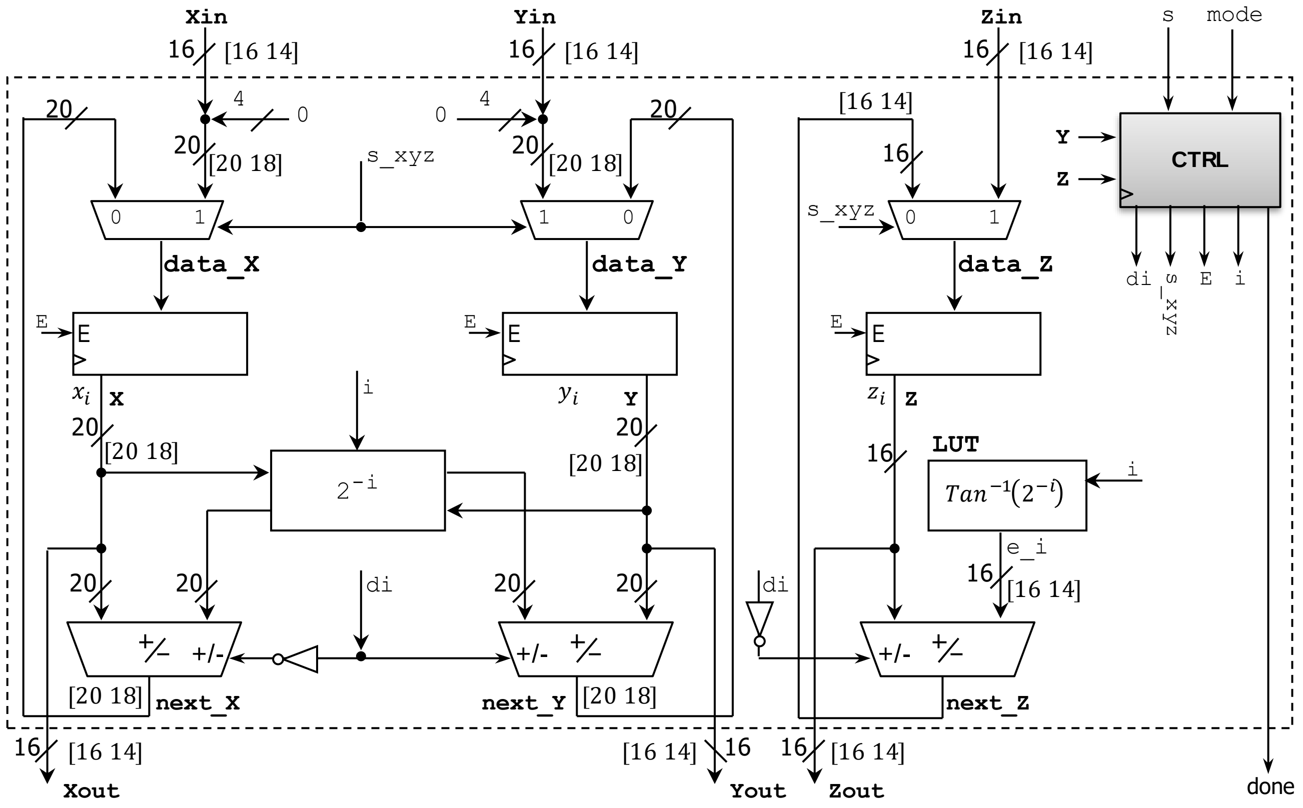
- Here the LUT holds the $\theta_i = 2^{-i}$ values with $i = 1,2,\dots,N$. After $N-1$ clock cycles, the result is obtained in the registers X, Y and Z, and a new process can be started. Note that we do not need an adder for x_i.

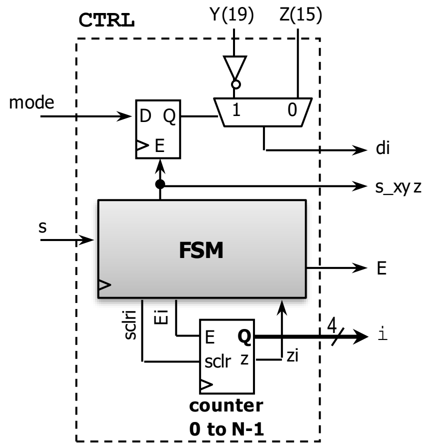


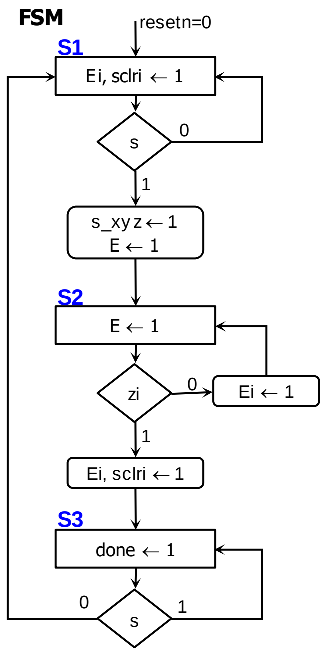
- Note: These architectures do not specify the numerical representation we are using. We are free to use any representation we see fit (e.g.: fixed point, dual fixed point, floating point). The adders, barrel shifters, and LUT will change depending on the desired format. If an arithmetic unit requires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

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

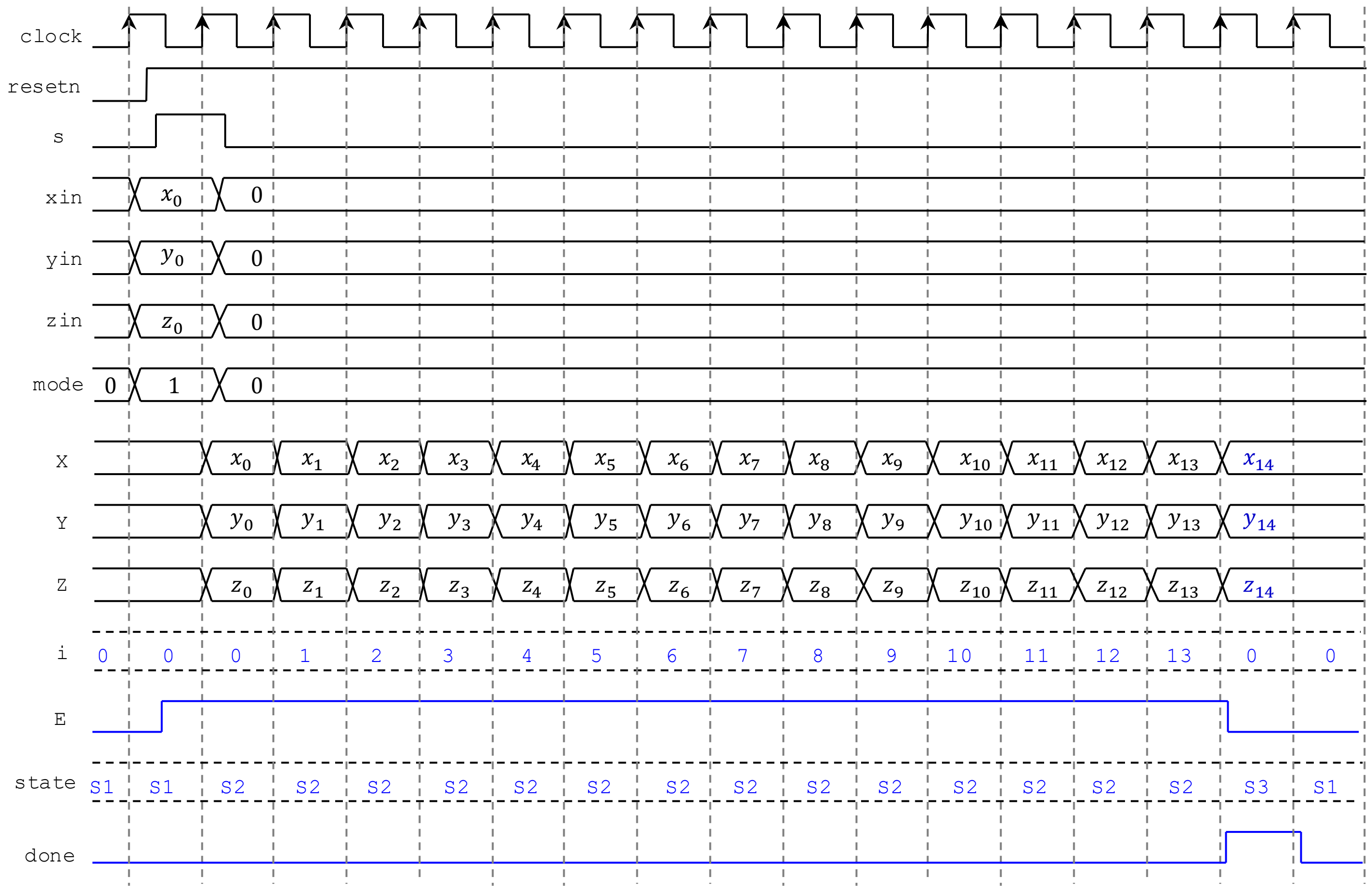
- $ng = 4$ guard bits. They improve accuracy, as the barrel shifters will get rid of many LSBs.
- mode = 0 → Rotation. mode = 1 → Vectoring.
- LUT: It holds the angles represented in [16 14] (signed) from $i = 0$ ($Tan^{-1}(2^0)$) to $i = N-1$ ($Tan^{-1}(2^{-(N-1)})$).
- Format [16 14] applied to the 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.


clock
resetn
s
xin
x_0
0
yin
y_0
0
zin
z_0
0
mode
0
1
0
X
x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 x_9 x_{10} x_{11} x_{12} x_{13} x_{14}
Y
y_0 y_1 y_2 y_3 y_4 y_5 y_6 y_7 y_8 y_9 y_{10} y_{11} y_{12} y_{13} y_{14}
Z
z_0 z_1 z_2 z_3 z_4 z_5 z_6 z_7 z_8 z_9 z_{10} z_{11} z_{12} z_{13} z_{14}
i
0 0 0 1 2 3 4 5 6 7 8 9 10 11 12 13 0 0
E
state
S1 S1 S2 S2 S2 S2 S2 S2 S2 S2 S2 S2 S2 S2 S2 S2 S3 S1
done

FIXED-POINT SQUARE ROOT

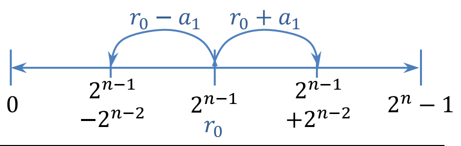
INTEGER SQUARE ROOT – BINARY SEARCH

- 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**
$Q \leftarrow 0$ $for\ k = n-1 \rightarrow 0$ $q_k \leftarrow 1$ $if\ D < Q^2\ then$ $q_k \leftarrow 0$ end end	$q_{n-1} \leftarrow 1$ $for\ k = n-2 \rightarrow 0$ $if\ D < Q^2\ then$ $Q \leftarrow Q - 2^k$ $else$ $Q \leftarrow Q + 2^k$ end end
Example: $D = 40 = 101000, Q = 000, n = 3$ $k = 2: q_2 = 1\ (Q = 100)$ $40 < 4^2?\ No$ $k = 1: q_1 = 1\ (Q = 110)$ $40 < 6^2?\ No$ $k = 0: q_0 = 1\ (Q = 111)$ $40 < 7^2?\ Yes \rightarrow\ q_0 = 0\ (Q = 110)$ 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

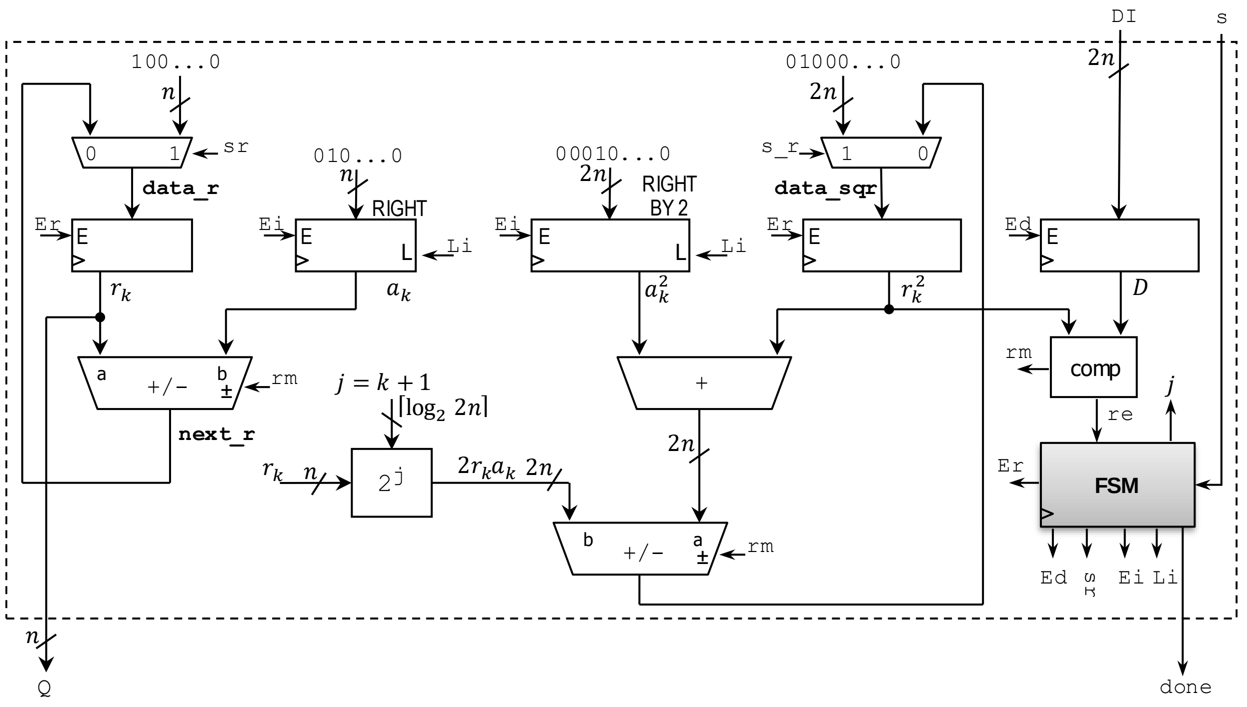
- For hardware implementation, we will select the non-restoring version as it is a bit simpler to implement in hardware. We make the following definitions:
  - ✓ $a_k = 2^k$. This is the correction factor at iteration k.
  - ✓ $r_k = Q(k)$. Value of the square root at iteration k.
  - ✓ $r_k^2 = Q(k)^2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



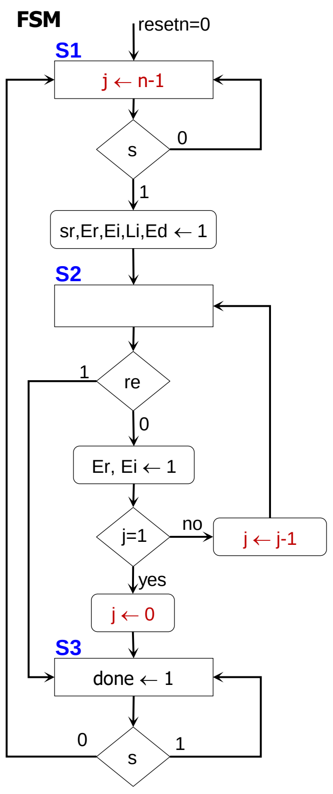
i	k	a_k	$2a_k$	a_k^2	r_k	r_k^2	$2a_k r_{k+1}$
0	$n-1$				2^{n-1}	2^{2n-2}	
1	$n-2$	2^{n-2}	2^{n-1}	2^{2n-4}	$2^{n-1} \pm 2^{n-2}$		$2^{n-1}(2^{n-1})$
2	$n-3$	2^{n-3}	2^{n-2}	2^{2n-6}			
...	...	...	...	...			
$n-3$	2	2^2	2^3	2^4			
$n-2$	1	2^1	2^2	2^2			
$n-1$	0	2^0	2^1	2^0			

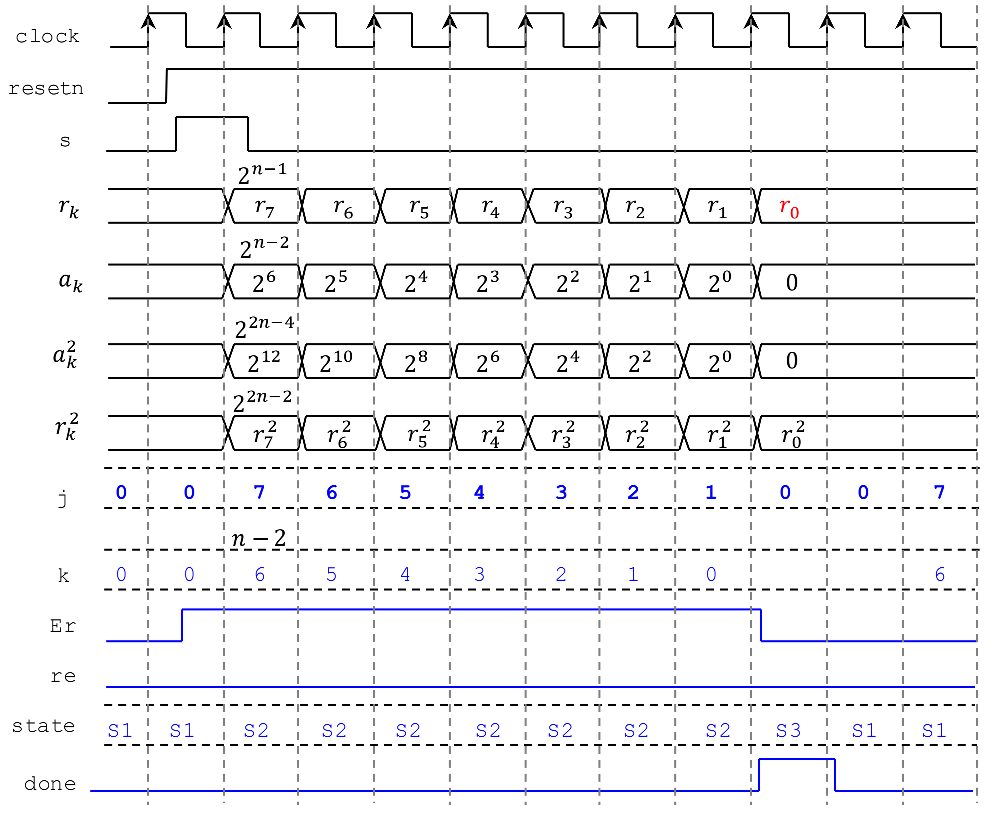
Algorithm (re-defined)

```
$r_{n-1} \leftarrow 2^{n-1}$
$for\ k = n-2 \rightarrow 0$
    $if\ D < r_k^2\ then$
        $r_k \leftarrow r_{k+1} - a_k$
    $else$
        $r_k \leftarrow r_{k+1} + a_k$
    $end$
$end$
```



- For hardware implementation, a_k and r_k use n bits, while a_k^2 and r_k^2 use $2n$ bits. Also, $2a_kr_k$ 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 $r_k^2 + a_k^2 - 2a_kr_k$, 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.

Radical: $D = d_{2n-1}d_{2n-2}d_{2n-3}d_{2n-4} \dots d_1d_0$

Square Root: $Q = q_{n-1}q_{n-2} \dots q_0$

We define:

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

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

- This is non-restoring algorithm, meaning that at the las iteration we might not have the correct remainder R. To get the correct value of R, an extra operation might be required.

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

- In practice, the remainder is rarely used, and the operation is usually not implemented (to reduce resource consumption).

- At each iteration, we compute $R'_k = r'_n r'_{n-1} r'_{n-2} \dots r'_k$ (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):
  - ✓ $R = R'_0 + Q_0 1$: 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).

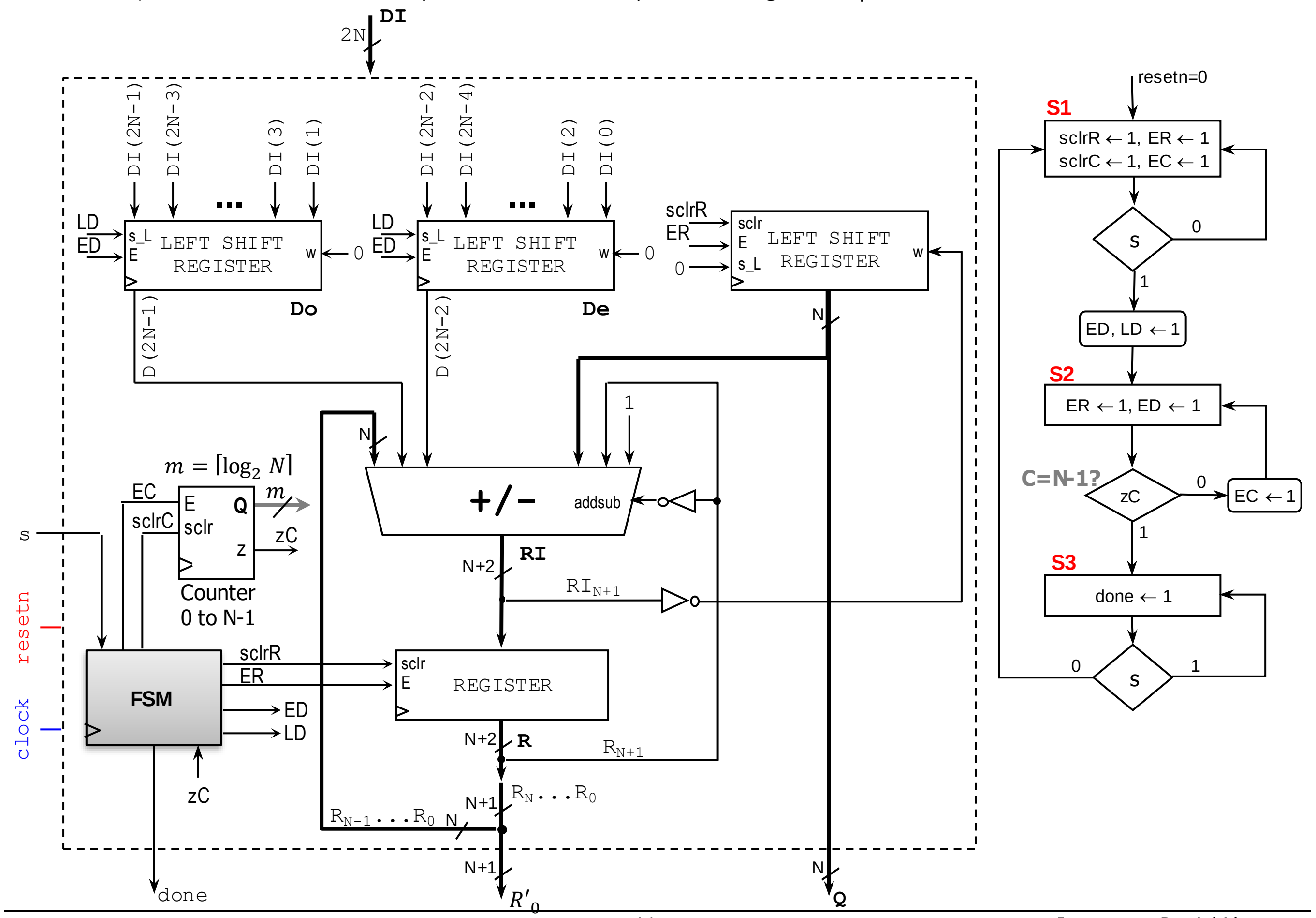
- Example: $n=4$: $D = 01111111$, $Q = 0000$. Note that R'_k has one more bit than Q_k.

k	R'_k	R'_k width	q_k	$Q_k = q_{n-1} \dots q_k$	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

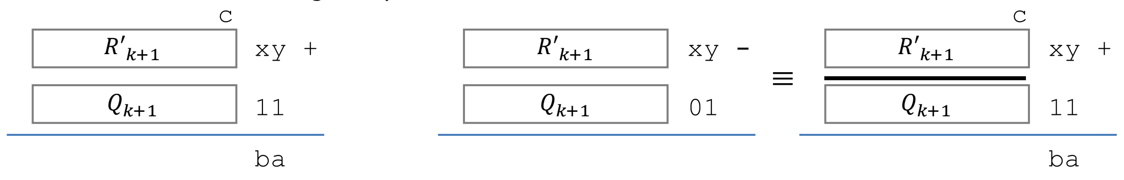
- The input data `DI` is captured into shift register `D`. The bits $d_{2k+1}d_{2k}$ correspond to the 2 MSBs of the register `D`. At every iteration, the register `D` shifts two bits to the left.
  - ✓ Register `D`: implemented by two parallel access shift registers: `Do` (for odd bits of `D`) and `De` (for even bits of `D`).
- 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:



  - ✓ 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 $xy - 01 \;=\; \overline{xy + 11}$. 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

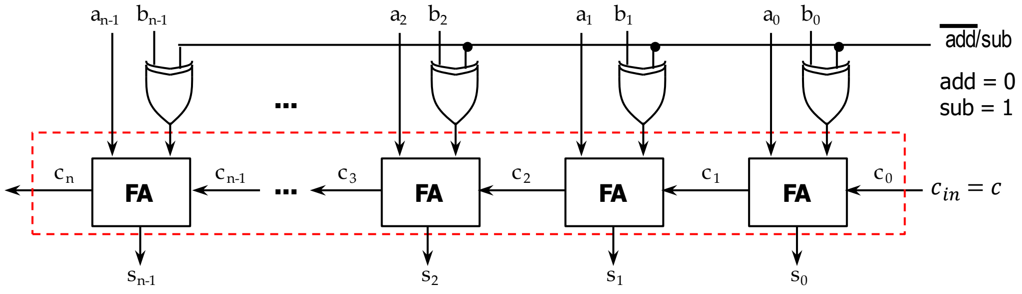
xy	cba
00	011
01	100
10	101
11	110

cba = xy - 01

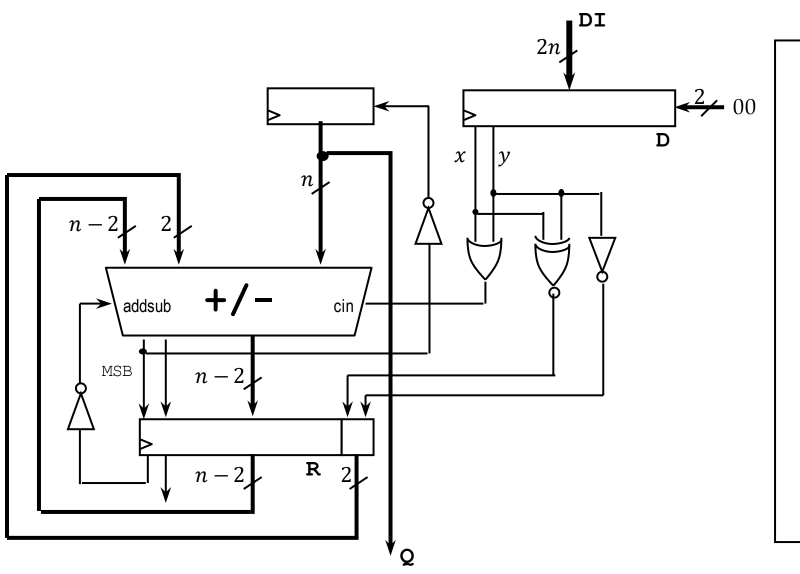
xy	cba
00	011
01	100
10	101
11	110

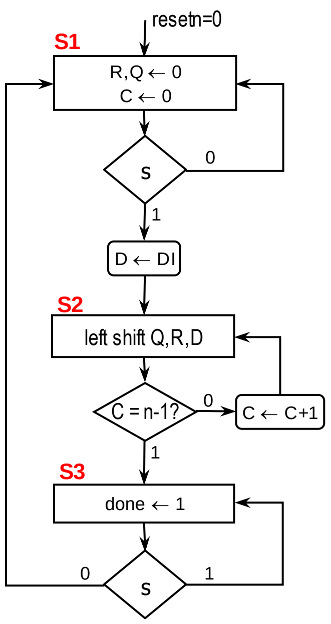
  - ✓ The n MSBs perform $A \pm B \pm c$: an addition or subtraction where c is the carry-in (or borrow-in).
    - For $xy + 11$: c is the carry-in to the n-bit addition.
    - For $xy - 01$: c is the borrow-in to the n-bit subtraction $A - B$, $A = R'_{k+1}$, $B = Q_{k+1}$.
      - $c = 0$: The n MSBs 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

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

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

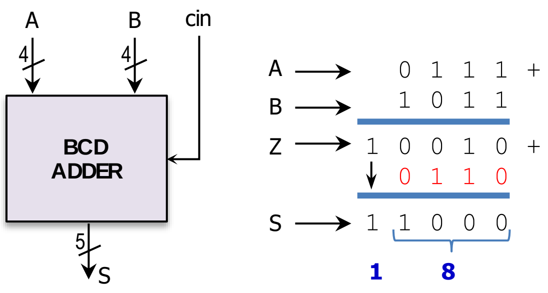
BCD ADDITION

- A number of input interfaces and output interfaces use BCD to represent their data as it provides a convenient human-readable format. The drawback of the BCD representation is arithmetic BCD circuits are complex and only 10 of the possible 16-bit patterns are used. As a result, it is common to include BCD to binary and binary to BCD converters when dealing with human-readable interfaces. We can then perform the operations in binary arithmetic.
- The importance of BCD has diminished somewhat, though it is still encountered. To illustrate the complexity of BCD arithmetic circuits, we consider the implementation of a BCD adder.

BCD ADDER

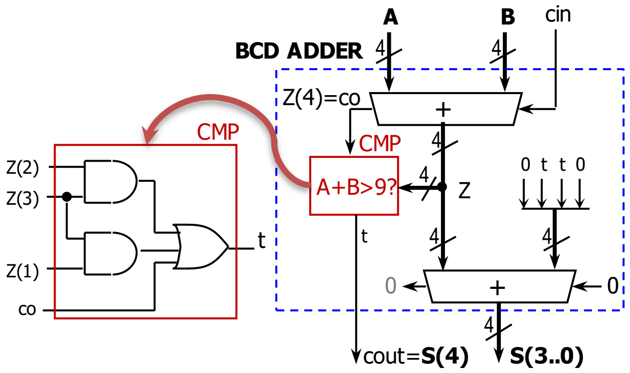
- This circuit adds two 4-bit unsigned numbers A and B (and a carry in). The binary addition is $Z = A + B + cin$.
- $Z(4..0)$ includes the carry out of the binary addition. $S(4..0)$ includes the carry out of the BCD addition. Some implementations use $S(7..0)$, where $S(7..5) = $ "000", so that the result is in BCD digits.
  - ✓ If $A + B + cin \leq 9$: The BCD result matches the binary result: $S = Z$. Here, $S(4) = 0$.
  - ✓ If $A + B + cin > 9$: The BCD result does not match the binary result. The table below shows all these cases. If we treat Z and S as 5-bit binary numbers (unsigned integers), we have that: $S = Z + 6$. Here, $S(4)=1$.

A+B+cin	Z	S	
10	01010	1 0000	
11	01011	1 0001	
12	01100	1 0010	
13	01101	1 0011	
14	01110	1 0100	
15	01111	1 0101	
16	10000	1 0110	
17	10001	1 0111	
18	10010	1 1000	
19	10011	1 1001	19 = 9 + 9 + 1



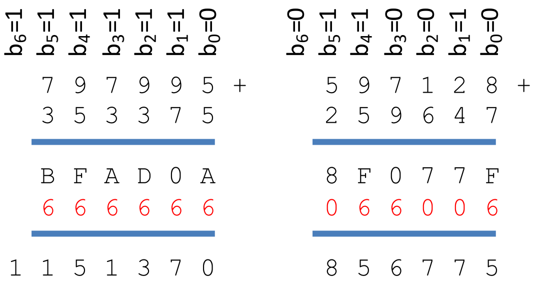
HARDWARE IMPLEMENTATION

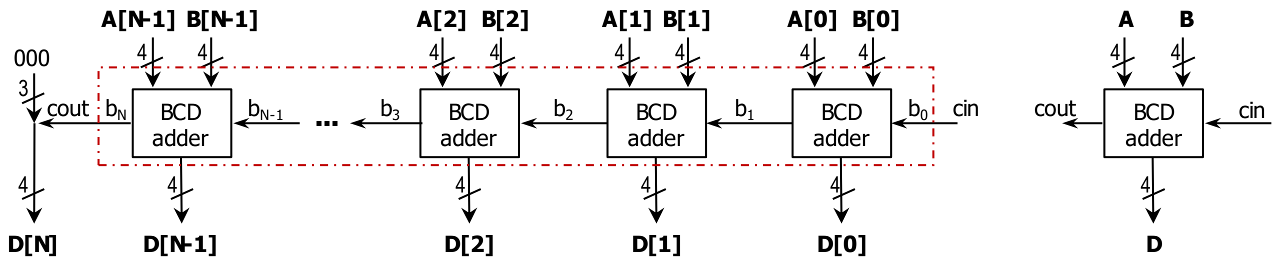
- A and B are 4-bit wide. Here, $Z(4)$: carry out of the 4-bit binary addition `A+B+cin`.
- $S(4) = 1$ if $A+B+cin > 9$. $S(4)$ can be thought as the carry out of the 4-bit binary addition $S=A+B+cin+P$ ($P=0000$ or 0110). $S(4)$ can be interpreted as the carry out of the BCD addition.
- Note that $Z(3..0) + 0110 \leq 1001$. Here, the carry out is always 0. Thus, $S(3..0) = Z(3...0) + P$.
- Note that if A or B is greater than 9, the result is invalid.
- Some implementations add 3 more 0's to the MSB, so as to have 2 BCD digits.



MULTI DIGIT BCD ADDITION

- To add multiple BCD digits, we need to propagate the carry from one digit to the other. Thus, we implement a carry-ripple BCD adder.
- Here, we prefer to represent a BCD adder with an output D (4 bits) and a carry out cout.
- In the examples shown, b_i are the nibble-level carries generated by each BCD summation. They can also be interpreted as the carries resulting from a typical decimal summation by hand.
- It is common to add "000" to the carry out of the multi-digit BCD addition. This is so that every BCD digit has 4 bits.
- VHDL code: Multi-Digit BCD Adder.



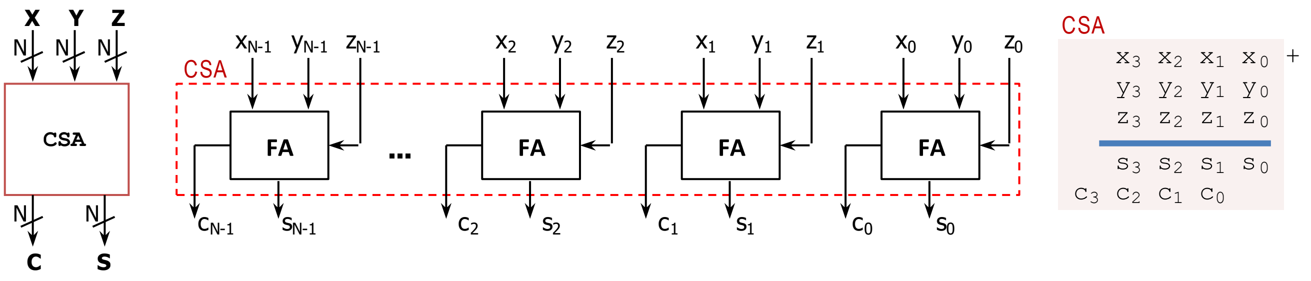


CARRY SAVE ADDITION (CSA)

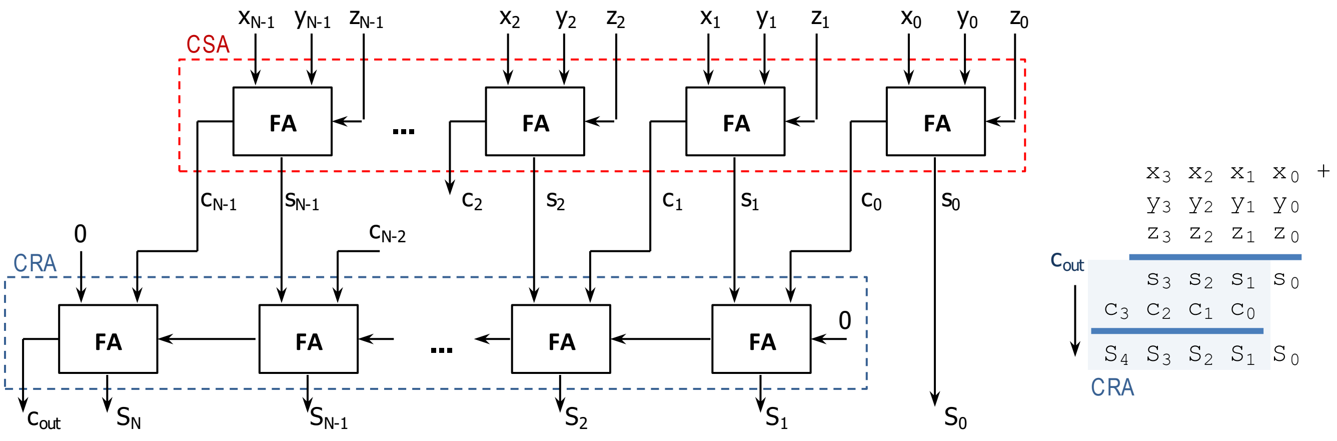
- Here, the carries are not propagated. Instead, the carries are generated as outputs.
- This technique allows for optimized implementation of 3-input adders and 2-input multipliers.

CSA ADDER

- The technique can be applied very effectively for 3-input adders. Inputs: three n-bit numbers X, Y, Z. Outputs: two n-bit numbers S and C. An n-bit CSA consists of n disjoint full adders.
- Unlike a normal adder (e.g.: carry-ripple adder, carry-lookahead adder), a CSA has the propagation delay of one FA.
- This is a powerful mechanism to improve timing with little, if any area penalty (even reduced area).

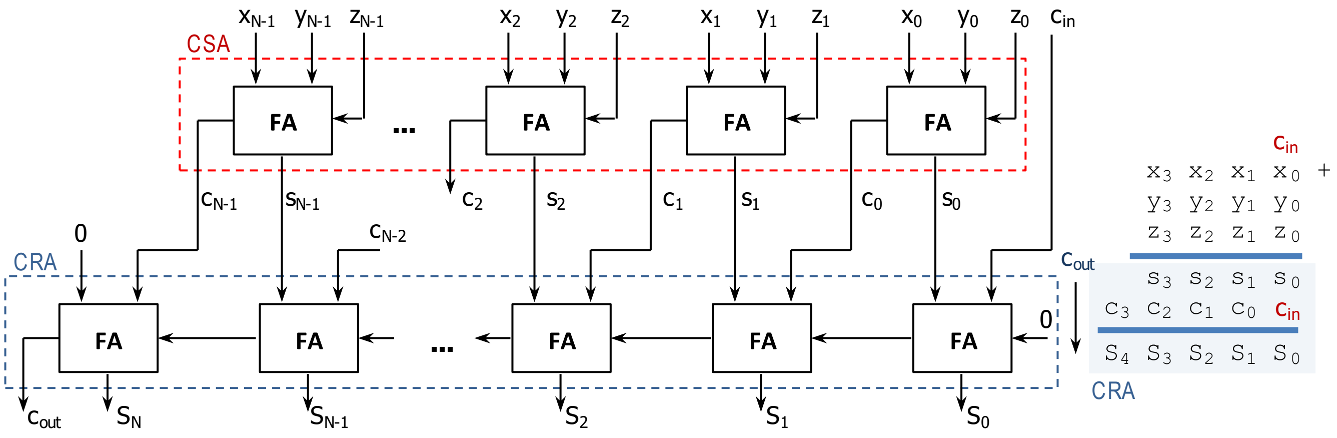

X Y Z
N N N
CSA
N N
C S
x_{N-1} y_{N-1} z_{N-1} x_2 y_2 z_2 x_1 y_1 z_1 x_0 y_0 z_0
CSA
FA ... FA FA FA
c_{N-1} s_{N-1} c_2 s_2 c_1 s_1 c_0 s_0
CSA
x_3 x_2 x_1 x_0 +
y_3 y_2 y_1 y_0
z_3 z_2 z_1 z_0
s_3 s_2 s_1 s_0
c_3 c_2 c_1 c_0

- The final result of adding 3 n-bit numbers requires at most $n+2$ bits. In order to get this result we need to add $s_{N-1} \dots s_1 s_0$ and $c_{N-1} \dots c_1 c_0 0$. This can be done by including a carry-ripple adder (CRA).
- The circuit below works for <u>unsigned numbers</u>, as we sum 0 and c_{N-1}. For unsigned numbers, c_{out} is the MSB of the final result. For signed numbers, sign-extend s_{N-1}. Note that for adding 3 numbers, this technique results in faster circuitry with fewer resources (2N FAs). If we were two carry-ripple adders to add 3 numbers, we would need ($2\text{N}+1$ FAs).


x_{N-1} y_{N-1} z_{N-1} x_2 y_2 z_2 x_1 y_1 z_1 x_0 y_0 z_0
CSA
FA ... FA FA FA
c_{N-1} s_{N-1} c_2 s_2 c_1 s_1 c_0 s_0
0
c_{N-2}
CRA
FA FA ... FA FA
0
c_{out} s_N s_{N-1} s_2 s_1 s_0
x_3 x_2 x_1 x_0 +
y_3 y_2 y_1 y_0
z_3 z_2 z_1 z_0
c_{out}
s_3 s_2 s_1 s_0
c_3 c_2 c_1 c_0
s_4 s_3 s_2 s_1 s_0
CRA

Adding 'carry in'

- If we want to cascade CSA-based blocks together, we need a carry in. This is useful for multi-precision addition.
- The figure shows how we updated the previous circuit (for <u>unsigned numbers</u>) to consider the c_{in}. Note that the CRA adder now is $(n+1)$-bits wide. We sent down c_{in} to the final ripple-carry adder. This circuit requires $2\text{N}+1$ FAs.

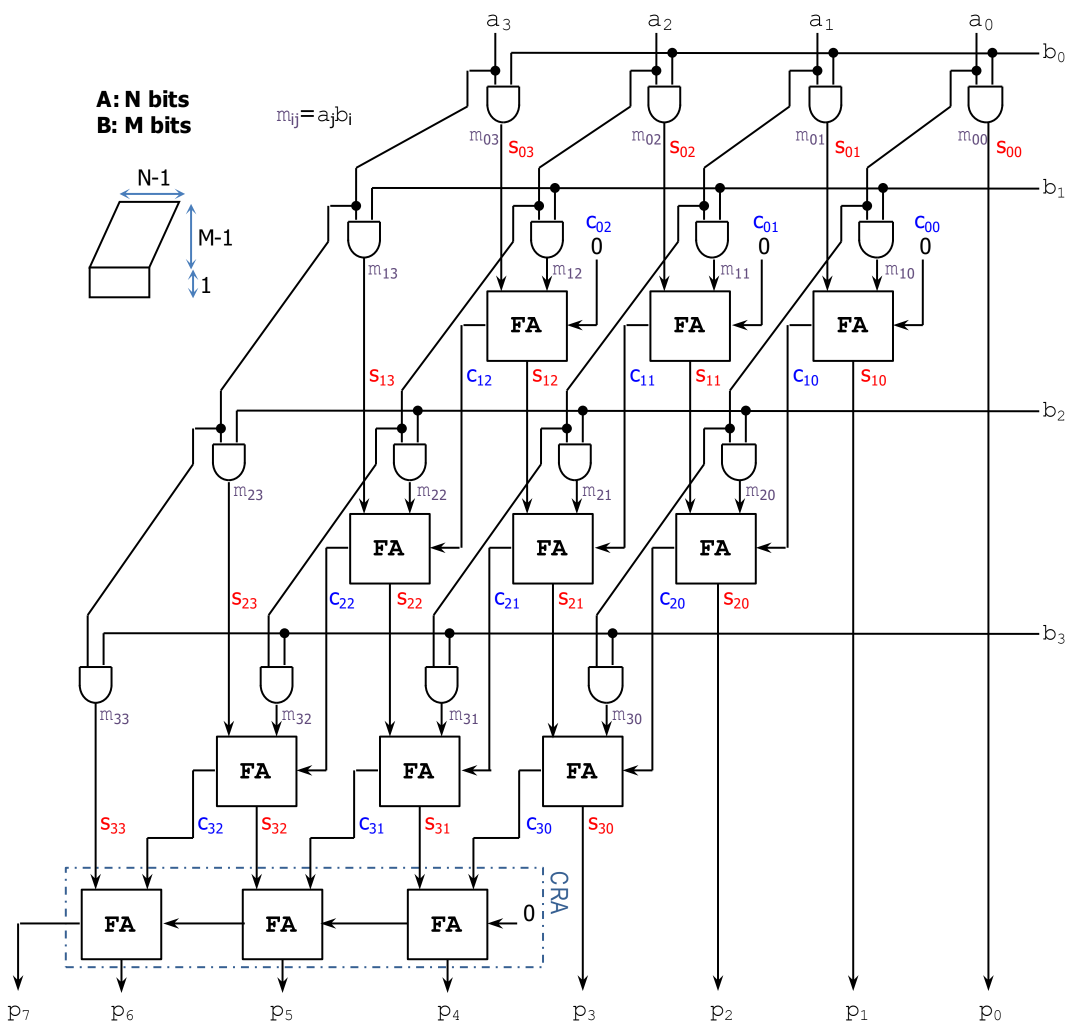

x_{N-1} y_{N-1} z_{N-1} x_2 y_2 z_2 x_1 y_1 z_1 x_0 y_0 z_0 c_{in}
CSA
FA ... FA FA FA
c_{N-1} s_{N-1} c_2 s_2 c_1 s_1 c_0 s_0
0
c_{N-2}
CRA
FA FA ... FA FA FA
0
c_{out} s_N s_{N-1} s_2 s_1 s_0
c_{in}
x_3 x_2 x_1 x_0 +
y_3 y_2 y_1 y_0
z_3 z_2 z_1 z_0
c_{out}
s_3 s_2 s_1 s_0
c_3 c_2 c_1 c_0 c_{in}
s_4 s_3 s_2 s_1 s_0
CRA

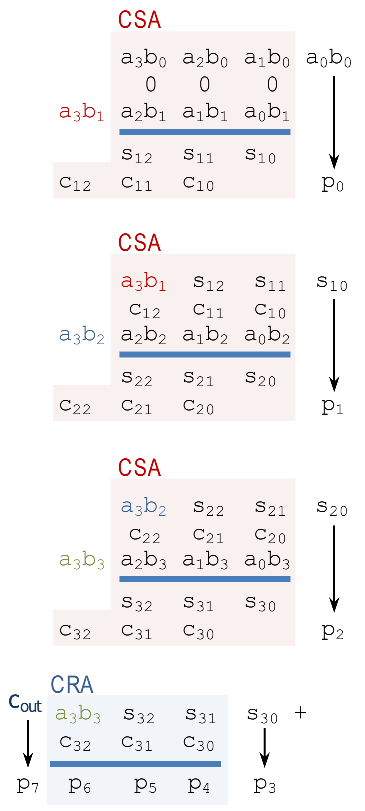
CSA UNSIGNED MULTIPLIER

- The concept of carry-save addition can also be used to implement efficient multipliers. An optimal implementation is shown in the figure. Delay is greatly reduced as compared to a typical array multiplier.
- Every stage (row) does not propagate the carries to the left; instead, the carries are sent down to the next stage. This facilitates pipelining. Only the last stage (CRA) propagates carries to the left. Since the last stage has the longest delay, it can be replaced by a carry look-ahead adder for speed improvement.
  - ✓ Inputs: N-bit A, M-bit B where A and B are unsigned numbers.
  - ✓ Output: $N+M$ bits.

$$\begin{array}{cccccccc}
 & & & & a_3 & a_2 & a_1 & a_0 \ \times \\
 & & & & b_3 & b_2 & b_1 & b_0 \\
\hline
 & & & & a_3b_0 & a_2b_0 & a_1b_0 & a_0b_0 \\
 & & & a_3b_1 & a_2b_1 & a_1b_1 & a_0b_1 & \\
 & & a_3b_2 & a_2b_2 & a_1b_2 & a_0b_2 & & \\
 & a_3b_3 & a_2b_3 & a_1b_3 & a_0b_3 & & & \\
\hline
p_7 & p_6 & p_5 & p_4 & p_3 & p_2 & p_1 & p_0
\end{array}$$

- The circuit uses M-1 CSA adders, each of N-1 bits. At the last stage, it uses a $(N$-$1)$-bit carry-ripple adder (CRA).
- Hardware resources: $(N\text{-}1)\times M$ FAs, and $N\times M$ AND gates.

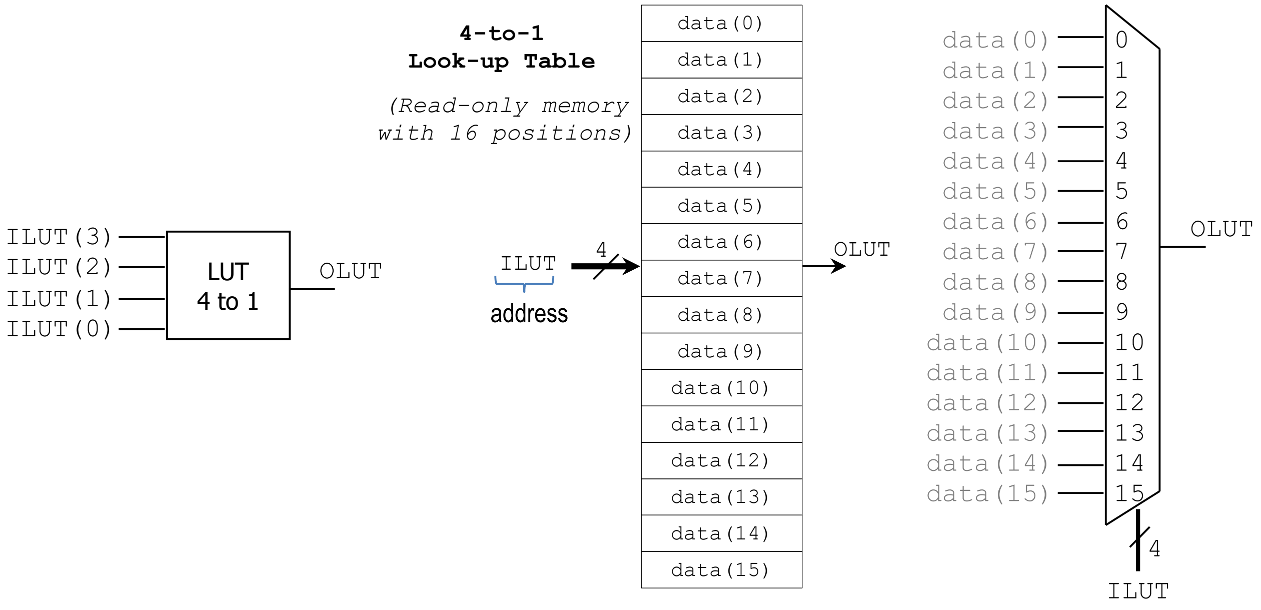



CSA
a_3b_0 a_2b_0 a_1b_0 a_0b_0
0 0 0
a_3b_1 a_2b_1 a_1b_1 a_0b_1
s_{12} s_{11} s_{10}
c_{12} c_{11} c_{10} p_0
CSA
a_3b_1 s_{12} s_{11} s_{10}
c_{12} c_{11} c_{10}
a_3b_2 a_2b_2 a_1b_2 a_0b_2
s_{22} s_{21} s_{20}
c_{22} c_{21} c_{20} p_1
CSA
a_3b_2 s_{22} s_{21} s_{20}
c_{22} c_{21} c_{20}
a_3b_3 a_2b_3 a_1b_3 a_0b_3
s_{32} s_{31} s_{30}
c_{32} c_{31} c_{30} p_2
CRA
c_{out}
a_3b_3 s_{32} s_{31} s_{30} +
c_{32} c_{31} c_{30}
p_7 p_6 p_5 p_4 p_3

SPECIAL TECHNIQUES

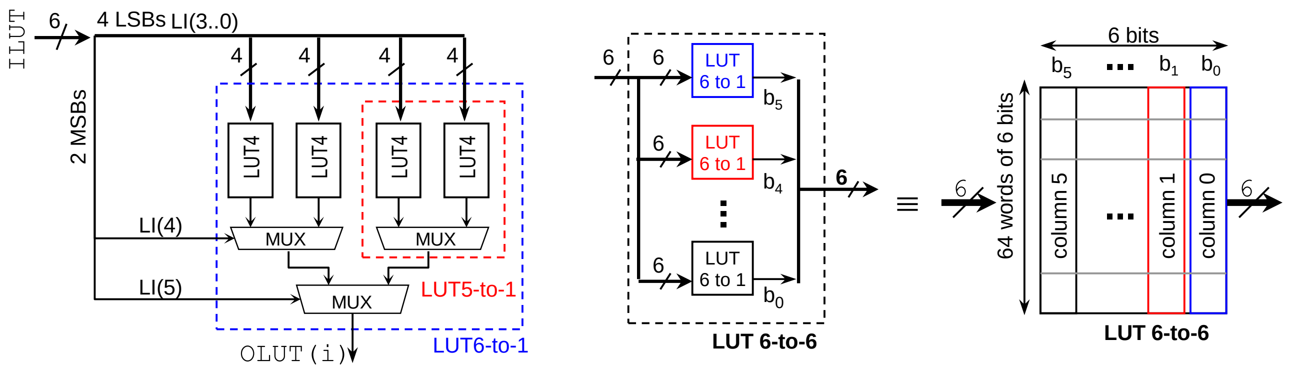
LUT (LOOK UP TABLE) APPROACH

- 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 (NI). Thus, this approach is only efficient for small input data sizes (≤ 12 in modern FPGAs).

FPGA RESOURCES

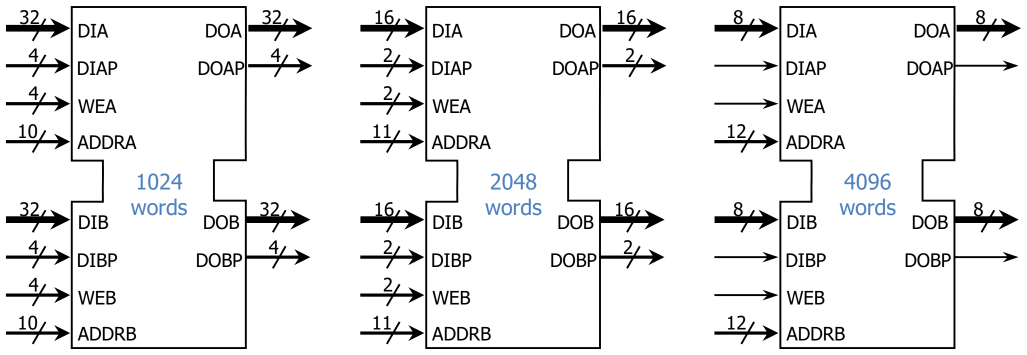
- When designing digital circuits using VHDL, we usually describe circuits at the Register Transfer Level (RTL). This allows us to implement digital systems that include registers and combinational components.
- When implementing digital circuits on FPGAs, we can include a variety of hard-wired components: Memories (BRAMs), FIFOs, Multiply-and-accumulate circuits (DSPs), Digital Clock Managers (DCMs), Analog-to-Digital Converters (ADCs). The quantity, speed, and configuration mechanism of these components depend on the specific FPGA vendor and FPGA family.
- Here, we deal with the Xilinx® 7-Series devices such as the Artix-7 FPGA and Zynq-7000 PSoC.

BLOCK RAMS (BRAMS)

- In theory, we can design memory elements using registers. However, these resources are very expensive and they will not satisfy simple memory requirements. For example, the XC7A100T FPGA (inside the Nexys-4 DDR Board) contains 15850×8 flip flops, i.e., we can only have 15850 bytes. This hardly satisfies simple memory demands in a digital system.
- Because of this, modern FPGAs include hard-wired memory elements called Block RAMs (BRAMs). For example, the XC7A100T (inside the Nexys-4 DDR Board) has a maximum of 540 KB of BRAM.
- Moreover, these BRAMs can be configured as true dual-port RAMs. This would not be possible to do with registers.
- A comprehensive documentation of BRAMs for 7-series devices can be found in: *UG473: 7 Series FPGAs Memory Resources*.
- Among the 7 series FPGA BRAM features, we can mention:
  - ✓ Two block RAM primitives (basic building blocks): RAMB36E1 (32,786 bits) and RAMB18E1 (16,384 bits). The port aspect ratio (data width and depth) can be configured as follows:

RAMB18E1			RAMB36E1		
Data Width	**Address Width**	**Depth**	**Data Width**	**Address Width**	**Depth**
1	14	16,384	1	15	32,768
2	13	8,192	2	14	16,384
4	12	4,096	4	13	8,192
8	11	2,048	8	12	4,096
16	10	1,024	16	11	2,048
			32	10	1,024

  - ✓ For Data Widths of 8, 16, and 32, there are byte-wide parity bits (1, 2, and 4), though they can also be used for additional data inputs, effectively increasing the memory capacity.
  - ✓ Two modes: Single dual-port (one read-only port and one write-only port) and True dual-port (both ports can access any memory location at any time).
  - ✓ Latency: 1 clock cycle. An output register can be added, which would increase the latency to 2 clock cycles.
  - ✓ Contents can be initialized using a RAM initialization file or by specifying a constant matrix as a parameter.
  - ✓ A byte-wide write enable is also included. The figure shows the True Dual-Port Data Flows for a RAMB36E1 primitive:



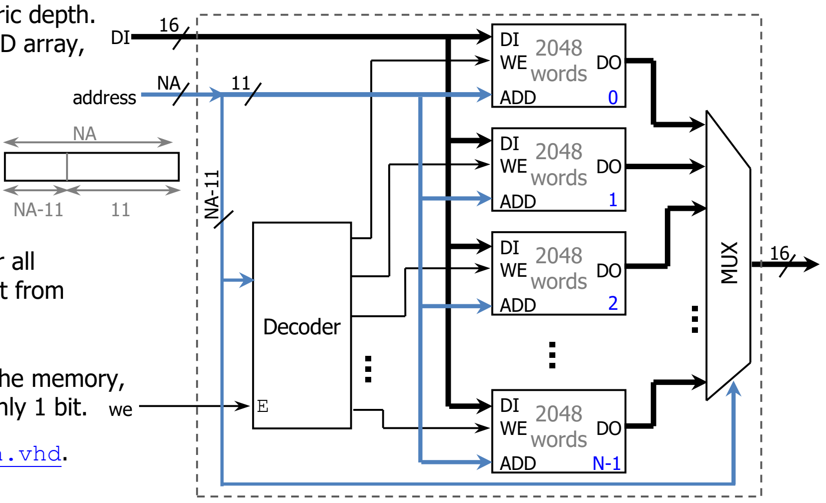
- **Example**: Memory with 16-bit data and generic depth. Memory size: $nr \times nc$ 16-bit words. This is a 2D array, where data is stored in a raster scan fashion.

  RAMB36E1 block: 2048 16-bit words.
  Thus, we need $N = \left\lceil \frac{nr \times nc}{2048} \right\rceil$ RAMB36E1 blocks.
  Address width: $NA = \lceil \log_2(nr \times nc) \rceil$.

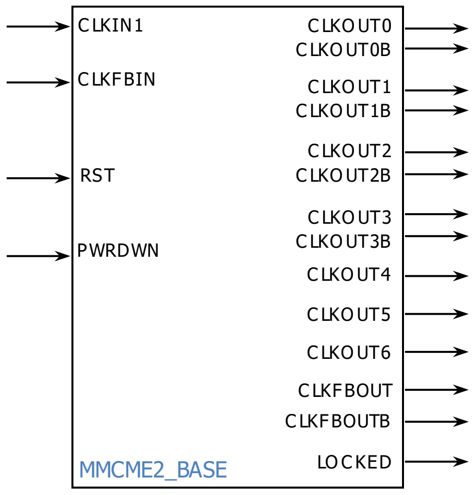
  Memory decoding: the 11 LSBs of 'address' for all the RAMB36E1 blocks. The $N-11$ MSBs select from which RAMB36E1 block we write or read.

  Here, we do not use the dual-port feature of the memory, only 1 port is used. Also, the write enable is only 1 bit.

  File (Artix-7 FPGA or 7-series PL): in_RAMgen.vhd.



DIGITAL CLOCK MANAGERS



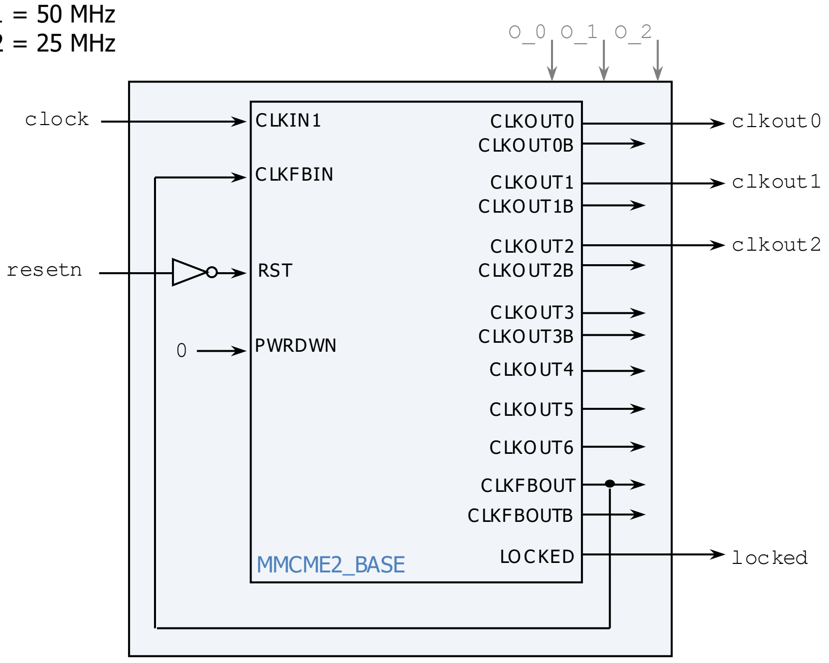
- Advanced FPGAs (including the 7-series) include mixed-mode clock managers (MMCM) as well as phase-locked loops (PLL, a subset of MMCM functions). They allow for frequency synthesis, jitter filtering, clock deskew.
- The Artix-7 FPGA includes a basic primitive: MMCME2_BASE.

- MMCME2_BASE: This primitive provides access to the most frequently used features of a MMCM: clock deskew, frequency synthesis, coarse phase shifting, and duty cycle programming.
- The number of output counters (dividers) is eight with some of them capable of driving out an inverted clock signal (180° phase shift).
- The output ports (shows in the figure) are described as follows:
  - ✓ Clock Input: `CLKIN1, CLKFBIN`
  - ✓ Control Inputs: `RST` (high-level asynchronous reset, required if input clock conditions change)
  - ✓ Clock Outputs:
    - `CLKOUT0 – CLKOUT6` (programmable clock outputs)
    - `CLKOUT0B – CLOUKOUT3B` (inverted clock outputs)
    - `CLKFBOUT, CLKFBOUTB` (MMCM feedback output)
  - ✓ Status Output: `LOCKED` (it asserts to indicate the MMCM has achieved frequency matching and phase alignment)
  - ✓ Power Control: `PWRDWN` (powers down the MMMCM)

- Formulas for output frequency:
  - ✓ $F_{CLKOUT\#} = F_{CLKIN} \times \frac{M}{D \times O\#}, \#: 0,1,2,3,4,5,6$
  - ✓ $F_{CLKOUTB} = F_{CLKIN} \times \frac{M}{D \times M} = \frac{F_{CLKIN}}{D}$
- There are restrictions on the values `M`, `D`, and `O#` (as well as on $F_{VCO} = F_{CLKIN} \times \frac{M}{D}$, which is the internal voltage controlled oscillator). For example, for the Artix-7 FPGAs, we have that:
  - ✓ `M`: 2 – 64 (fractions are allowed in increments of 1/8)
  - ✓ `D`: 1 -106
  - ✓ `O0-O6`: 1 – 128 (fractions are allowed for `O0` in increments of 1/8)

- Besides the frequency of operation, the duty cycle and phase can also be modified. A comprehensive documentation of MMCMs for 7-series devices can be found in: *UG472: 7 Series FPGAs Clocking Resources User Guide*.

- **MMCM wrapper**: This circuit instantiates the MMCME2_BASE primitive and provides a simplified version. This circuit includes only three output clocks, whose frequency can be programmable (via parameters `O_1, O_2, O_3`).
  - ✓ File (Artix-7 FPGA or 7-series PL): `MMCM_wrapper.vhd`.
  - ✓ Example: input frequency: 100 MHz, `O_0=1, O_1=2, O_2=4`
    - clockout0 = 100 MHz
    - clockout1 = 50 MHz
    - clockout2 = 25 MHz


O_0
O_1
O_2
clock
resetn
0
CLKIN1
CLKFBIN
RST
PWRDWN
CLKOUT0
CLKOUT0B
CLKOUT1
CLKOUT1B
CLKOUT2
CLKOUT2B
CLKOUT3
CLKOUT3B
CLKOUT4
CLKOUT5
CLKOUT6
CLKFBOUT
CLKFBOUTB
LOCKED
MMCME2_BASE
clkout0
clkout1
clkout2
locked