# Notes - Unit 4

PIPELINING/UNFOLDING
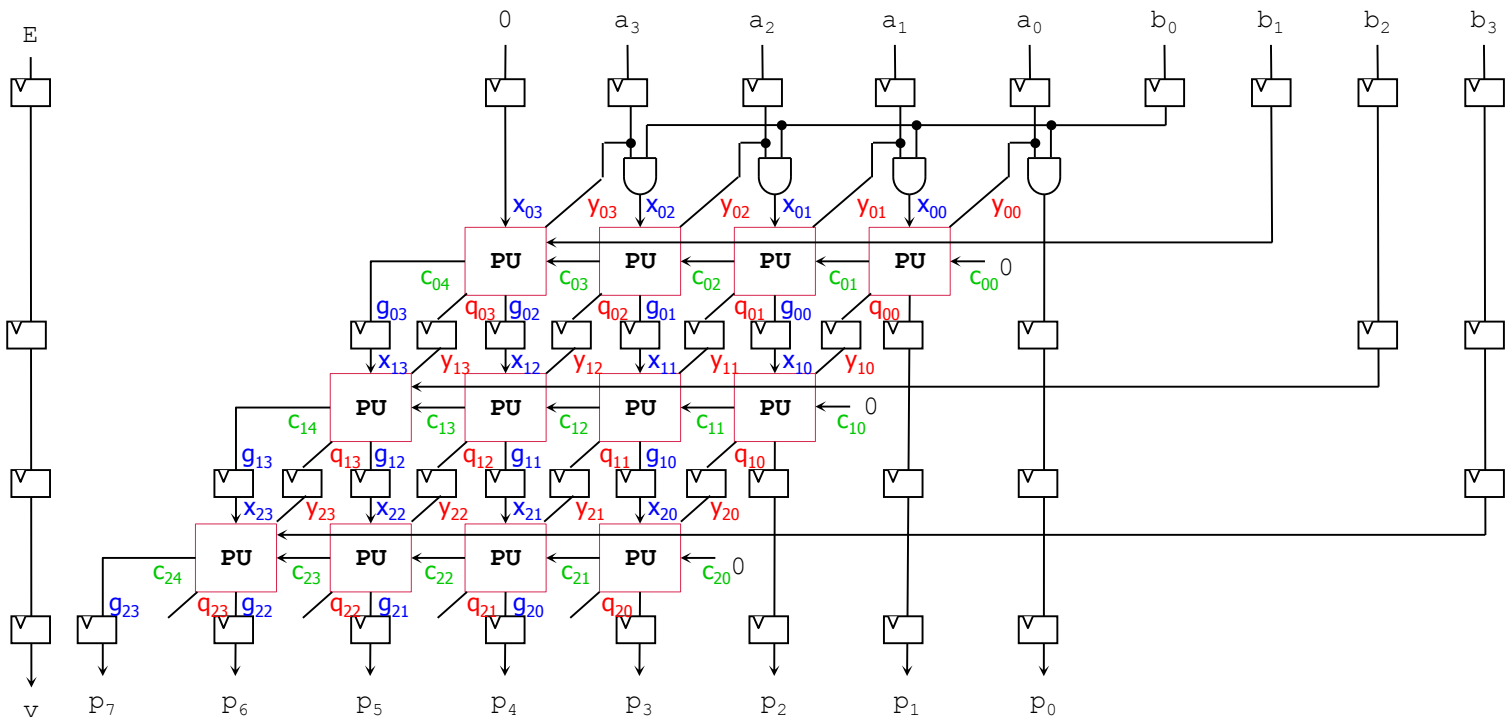
MULTIPLICATION

UNSIGNED MULTIPLICATION
- We already know the iterative version of the multiplier. Here, we show how to implement the multiplication using an array multiplier. In this implementation, two rows are added up at each stage.

- **Unfolded version** (purely combinational): Here, we have a different hardware for every summation of two rows.
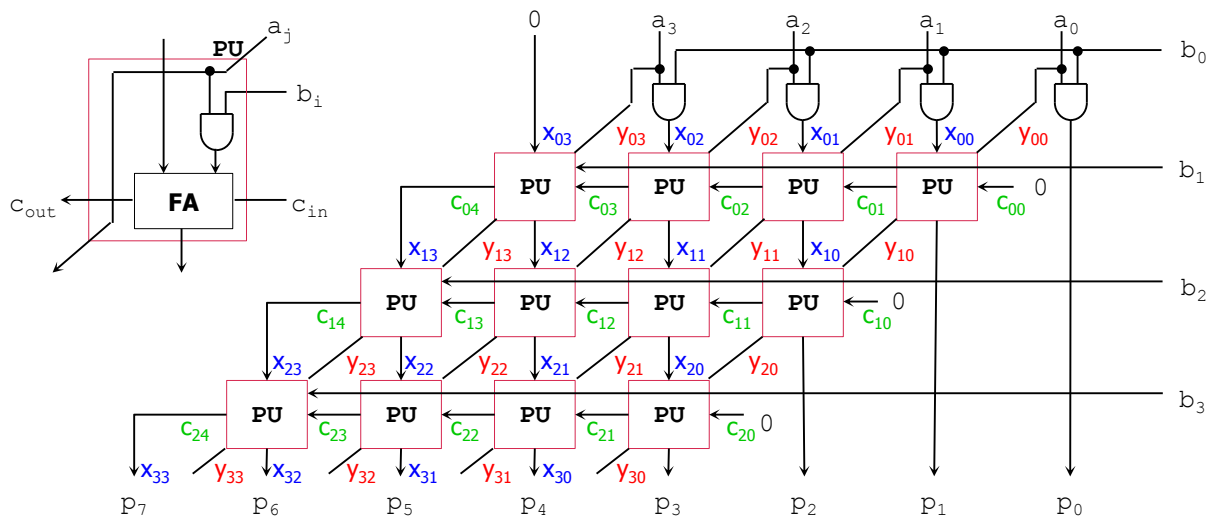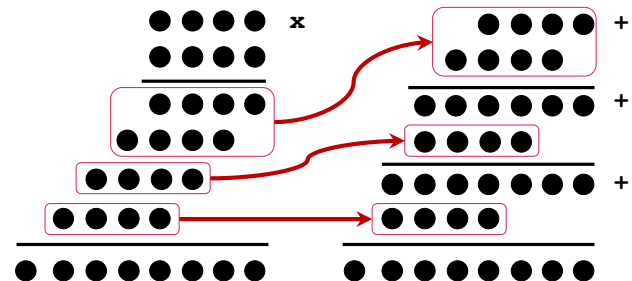
- **Pipelined version**: To increase frequency of operation, we place registers at every stage. Here, we are also including an enable input and a valid output.

SIGNED MULTIPLICATION

- We follow the same idea as in the iterative case. We need to add one pre-processing stage and one post-processing stage.

DIVISION

RESTORING ARRAY DIVIDER FOR UNSIGNED INTEGERS

- A, B: positive integers in unsigned representation. $A = a_{N-1}a_{N-2} \ldots a_0$ with N bits, and $B = b_{M-1}b_{M-2} \ldots b_0$ with M bits, with the condition that $N \geq M$. $Q = quotient$, $R = residue$. $A = B \times Q + R$.

In this parallel implementation, the result of every stage is called the remainder R_i.

The figure depicts the parallel algorithm with N stages. For each stage i, $i = 0, \ldots, N-1$, we have:
R_i: output of stage i. Remainder after every stage.
Y_i: input of stage i. It holds the minuend.

For the next stage, we append the next bit of A to R_i. This becomes Y_{i+1} (the minuend):
$$Y_{i+1} = R_i \& a_{N-1-i}, i = 0, \ldots, N-1$$

At each stage i, the subtraction $Y_i - B$ is performed. If $Y_i \geq B$ then $R_i = Y_i - B$. If $Y_i < B$, then $R_i = Y_i$.

Stage	Y_i	Computation of R_i	# of R_i bits
0	$Y_0 = a_{N-1}$	$R_0 = Y_0 - B, if\ Y_0 \geq B$ $R_0 = Y_0, if\ Y_0 < B$	1
1	$Y_1 = R_0 \& a_{N-2}$	$R_1 = Y_1 - B, if\ Y_1 \geq B$ $R_1 = Y_1, if\ Y_1 < B$	2
2	$Y_2 = R_1 \& a_{N-3}$	$R_2 = Y_2 - B, if\ Y_2 \geq B$ $R_2 = Y_2, if\ Y_2 < B$	3
...	...	...	...
M-1	$Y_{M-1} = R_{M-2} \& a_{M-N}$	$R_{M-1} = Y_{M-1} - B, if\ Y_{M-1} \geq B$ $R_{M-1} = Y_{M-1}, if\ Y_{M-1} < B$	M

Since B has M bits, the operation $Y_i - B$ requires M bits for both operands. To maintain consistency, we let Y_i be represented with M bits.

R_i: output of each stage. For the first M stages, R_i requires $i + 1$ bits. However, for consistency and clarity's sake, since R_i might be the result of a subtraction, we let R_i use M bits.

Figure (right side):

M bits

Stage 0 — Y_0, R_0
Stage 1 — Y_1, R_1
Stage 2 — Y_2, R_2
Stage 3 — Y_3
...
R_{M-2}, Y_{M-1}
Stage M-1 — R_{M-1}, Y_M
Stage M — R_M, Y_{M+1}
Stage M+1 — R_{M+1}, Y_{M+2}
Stage M+2
...
R_{N-2}, Y_{N-1}
Stage N-1 — R_{N-1}

M+1 bits

Parallel implementation algorithm

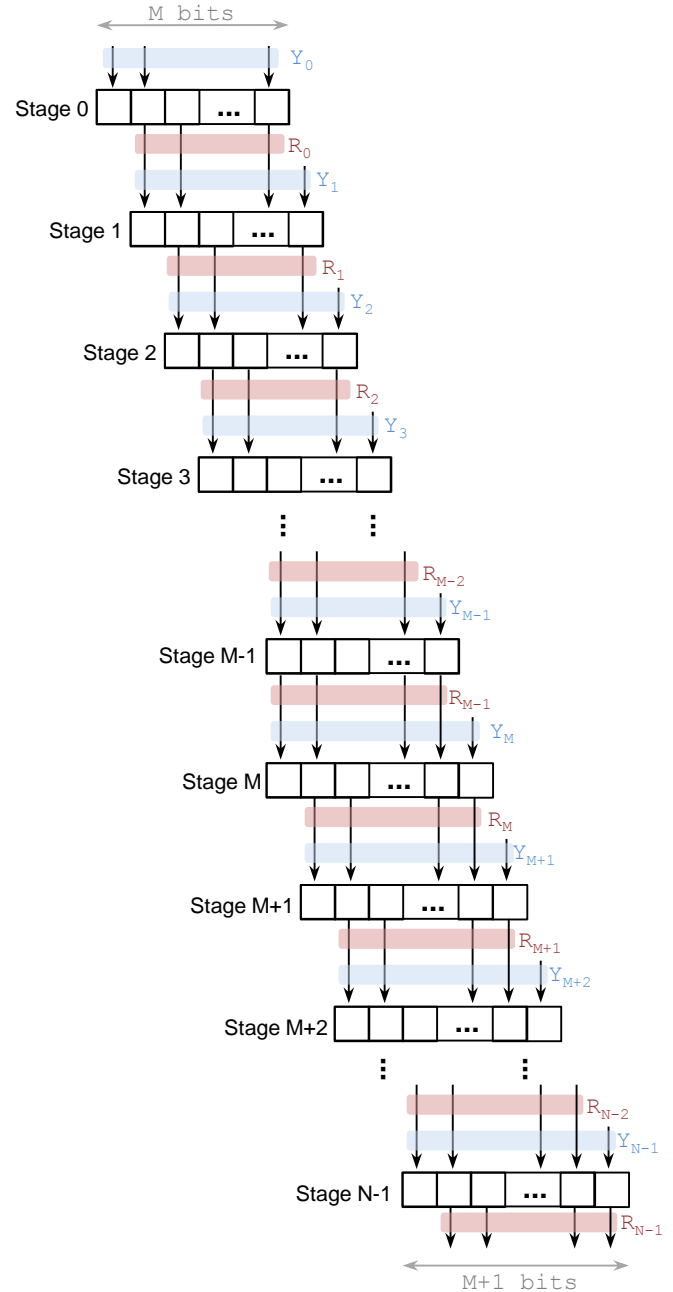For stages $0\ to\ M-1$:
R_i is always transferred onto the next stage. Note that we transfer R_i with $M-1$ least significant bits. There is no loss of accuracy here since R_i at most requires $M-1$ bits for stage $M-2$. We need R_i with M-1 bits since Y_{i+1} uses M bits.

Stages $M\ to\ N-1$:
Starting from stage $M-1$, R_i requires M bits. We also know that the remainder requires at most M bits (maximum value is $2^M - 2$).
So, starting from stage M-1 we need to transfer M bits.
As Y_{i+1} now requires $M+1$ bits, we need $M+1$ units starting from stage M.

- To implement the operation $Y_i - B$ we use a subtractor. When $Y_i \geq B \rightarrow cout_i = 1$, and when $Y_i < B \rightarrow cout_i = 0$. This $cout_i$ becomes a bit of the quotient: $Q_i = cout_{N-1-i}$. This quotient Q requires N bits at most.
- Also, the final remainder is the result of the last stage. The maximum theoretical value of the remainder is $2^M - 2$, thus the remainder R requires M bits. $R = R_{N-1}$.
- Also, note that we should avoid a division by 0. If $B = 0$, then, in our circuit: $Q = 2^N - 1$ and R $= a_{M-1}a_{M-2} \ldots a_0$.
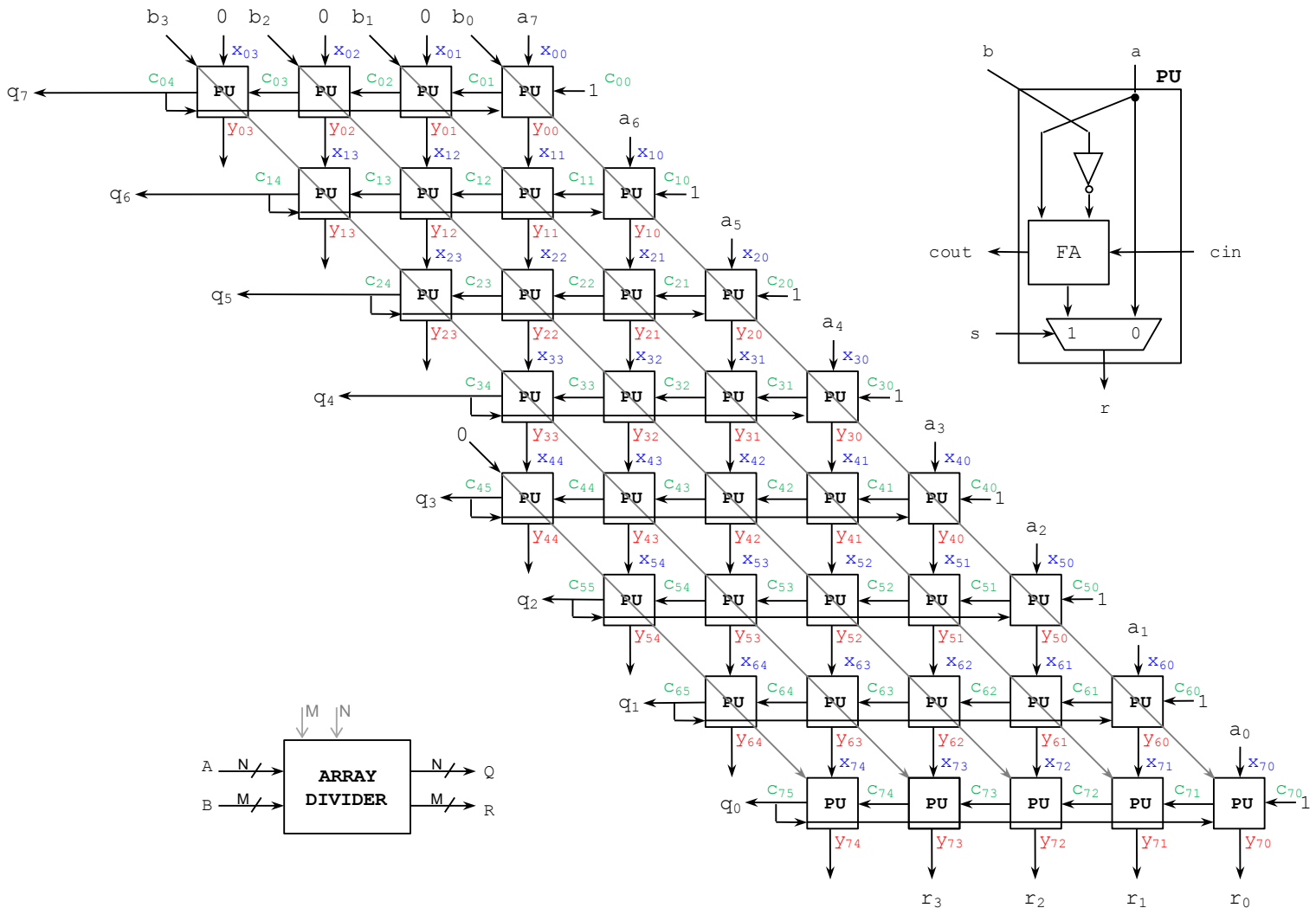
COMBINATIONAL ARRAY DIVIDER

The figure shows the hardware of this array divider for N=8, M=4. Note that the first $M = 4$ stages only require 4 units, while the next stages require 5 units. This is fully combinatorial implementation.

- Each level computes R_i. It first computes $Y_i - B$. When $Y_i \geq B \rightarrow cout_i = 1$, and when $Y_i < B \rightarrow cout_i = 0$. This $cout_i$ is used to determine whether the next R_i is $Y_i - B$ or Y_i.
- Each Processing Unit (PU) is used to process $Y_i - B$ one bit at a time, and to let a particular bit of either $Y_i - B$ or Y_i be transferred on to the next stage.

Fully Combinatorial Array Divider architecture for N=8, M=4

FULLY PIPELINED ARRAY DIVIDER

The figure shows the hardware core of the fully pipelined array divider with its inputs, outputs, and parameters.
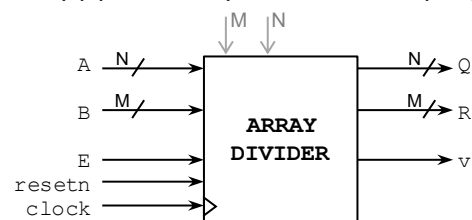
Fully pipelined IP core for the array divider

The figure shows the internal architecture of this pipelined array divider for N=8, M=4. Note that the first M=4 stages only require 4 units, while the next stages require 5 units. Note that the enable input 'E' is only an input to the shift register on the left, which is used to generate the valid output v. This way, valid outputs are readily signaled. If E='1', the output result is computed in N cycles (and v='1' after N cycles).
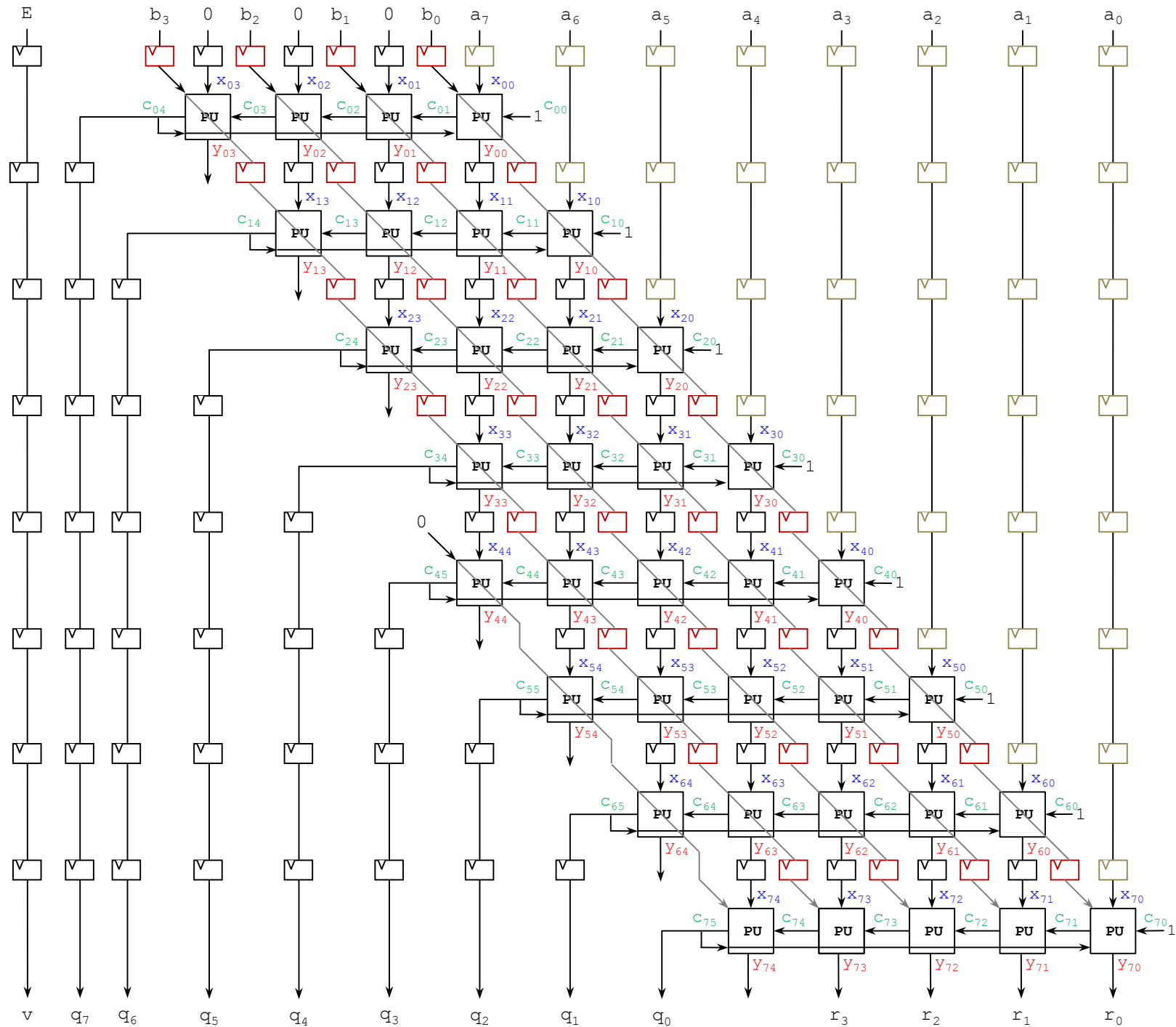
Fully Pipelined Array Divider architecture for N=8, M=4

SIGNED DIVISION
- We follow the same idea as in the iterative case. We need to add one pre-processing stage and one post-processing stage.

SQUARE ROOT

- We use the optimized algorithm of Unit 2.
- **Unfolding**: every single iteration is implemented by a particular hardware. By observing the algorithm, we need n stages with n adder/subtractors.
- As in the case of the iterative circuitry, there is a reduction in this case as well for the first iteration:

$$R'_{n-1} = d_{2n-1}d_{2n-2} - 01$$

$$q_{n-1} = \begin{cases} 1, if R'_{n-1} \geq 0 \\ 0, if R'_{n-1} < 0 \end{cases}$$

$$\rightarrow q_{n-1} = d_{2n-1}d_{2n-2}, \; b = \overline{d_{2n-1} \oplus d_{2n-2}}, \; a = \overline{d_{2n-2}}$$

d_{2n-1} d_{2n-2}	$R'_{n-1} = cba$	q_{n-1}
0 0	111	0
0 1	000	1
1 0	001	1
1 1	010	1

R'_{n-1} requires $n - (n-1) + 1 = 2$ bits, thus we only use the last 2 LSBs of the result.

Also, since these are few logic gates on the first iteration, we can embed the first and second stages into one stage. Finally, we include registers levels at every stage. We have $n - 1$ register stages.

In addition, you can always add a shift register for E and v.
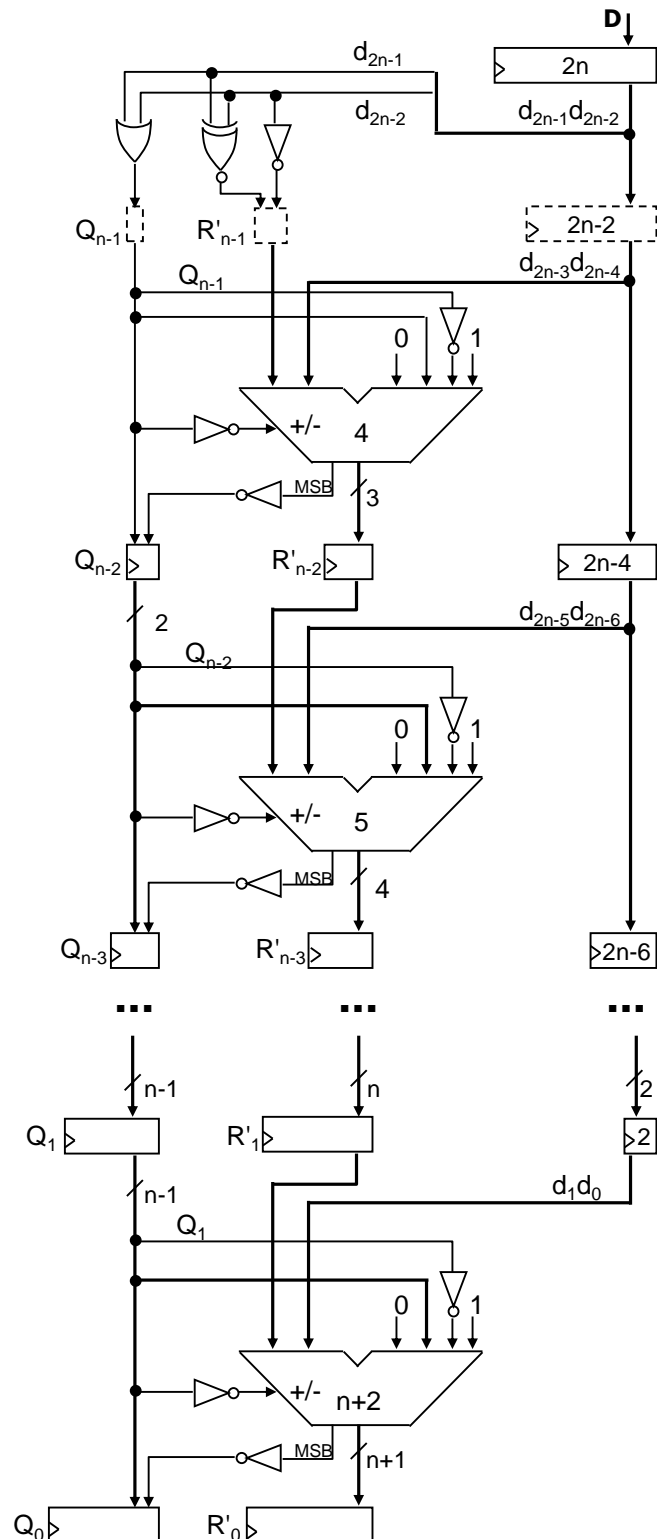
CORDIC

- Here, we just need to implement every iteration as a different hardware architecture. The figure shows the circular CORDIC (basic algorithm).
- **Unfolding**: every iteration is implemented by a specific hardware. For the circular CORDIC algorithm, where $i = 0,1,2,\ldots n$, there are n iterations where values are computed. So, we need n stages.
- To qualify input data xin, yin, zin, we use the enable input E. The valid output v is then generated by a simple shift register of length n.