Unit 4- Pipelining and Unfolding

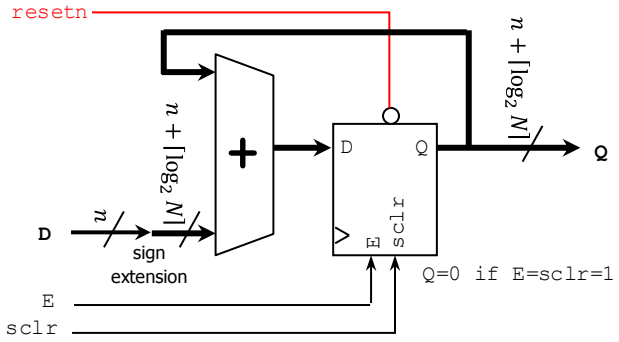
PIPELINING/UNFOLDING OF ITERATIVE ARCHITECTURES

MULTI-OPERAND ADDITION

- Addition of N n –bit numbers (signed, unsigned)

ITERATIVE DESIGN (FOLDED): ACCUMULATOR

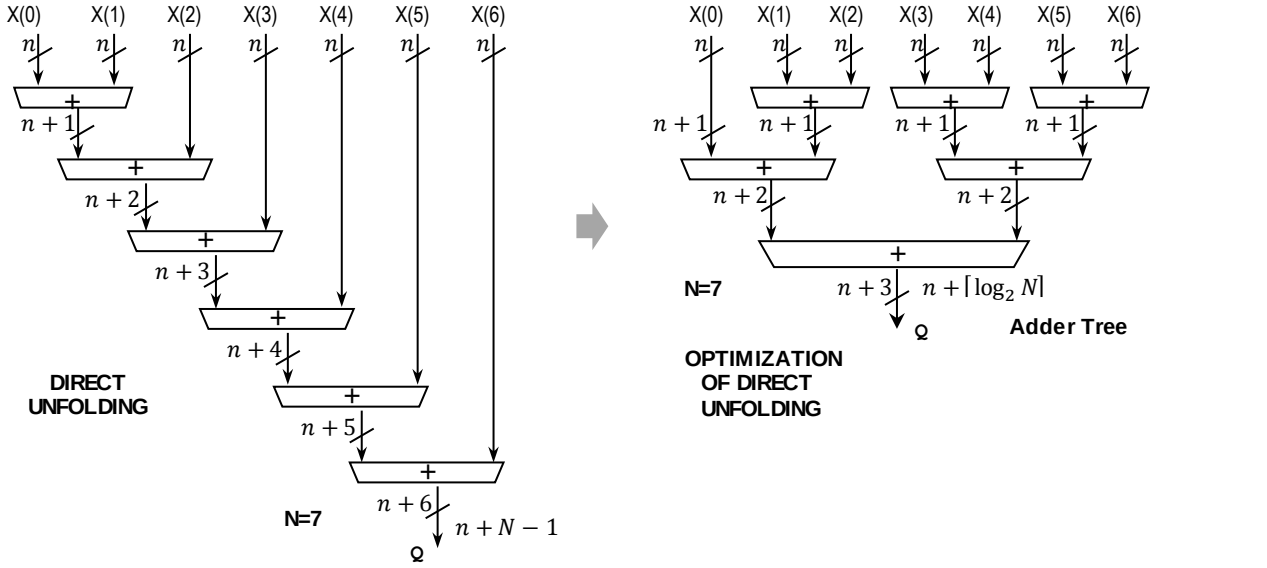
- Even if we have all the data (N numbers) ready, we can only feed one number at a time.
- We sign-extend (or zero-extend) the input D depending on whether we are adding signed or unsigned numbers.
- This architecture takes N cycles to add N numbers. It must wait one more cycle before loading the next batch of numbers.
- Computation time for T N-number groups: $T \times (N+1)$ cycles.
- Note how the required number of bits grow to $n + \lceil \log_2 N \rceil$.



UNFOLDED ACCUMULATOR:

- **Unfolding**: for each iteration, the architecture that computes that iteration is replicated. To add N numbers, we need to apply $N-1$ additions. For example, for $N=7$, the unfolded version of the iterative architecture is shown below. It is called 'Direct Unfolding' architecture.
- Note that we can optimize this 'Direct Unfolding' architecture by using an Adder Tree.

- **Adder Tree**: Structure that optimizes the number of two-input adders.
  - ✓ Adder Levels: This is given by $\lceil \log_2 N \rceil$. A level is a set of adders whose inputs have the same bit-width.
  - ✓ Number of output bits: $n + \lceil \log_2 N \rceil$.
  - ✓ If N is not a power of 2, some adder levels will have data inputs that are passed (sign-extended or zero-extended) to the next adder level. Within an adder, we increase the number of bits depending on the representation:
    - Signed numbers: at every level, we need to sign extend the operands, in order to get the proper result.
    - Unsigned numbers: you can zero-extend the operands, or just use the carry out as the MSB of the result.

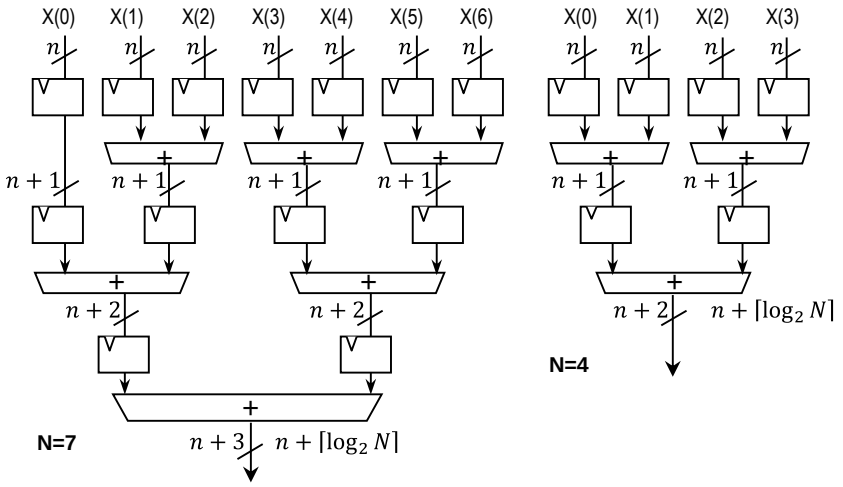


- This unfolded architecture can process a group of N numbers in one clock cycle at the expense of a large increase in hardware resources. Computation time for T groups of N numbers: T cycles.
- Note that if you can only produce one number per clock cycle, the iterative architecture is the best option.
- Even though data can be computed in clock cycle, the propagation delay is very large, and thus the clock cycle period will be large. To increase the frequency of operation, we need to apply pipelining.

PIPELINED DESIGN (UNFOLDED): ADDER TREE

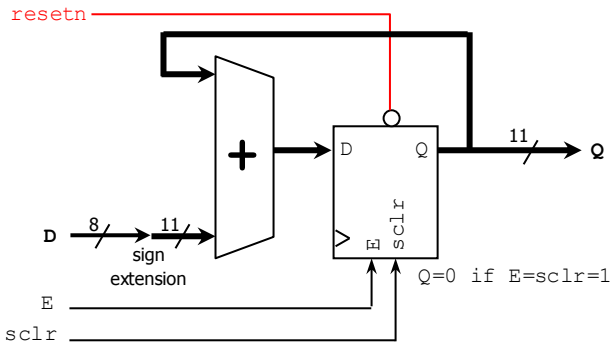
- **Pipelining**: Registers are inserted in between the architecture in order to increase the frequency of operation.
  - ✓ The number of register levels to include depends on the architecture.
  - ✓ Initial Latency: Output data will be ready a number of cycles (= register levels) after input data is loaded.
  - ✓ Note that we can load new input data at every new cycle. After the initial latency, we get output results every clock cycle. Overtime, the initial latency can be considered negligible.

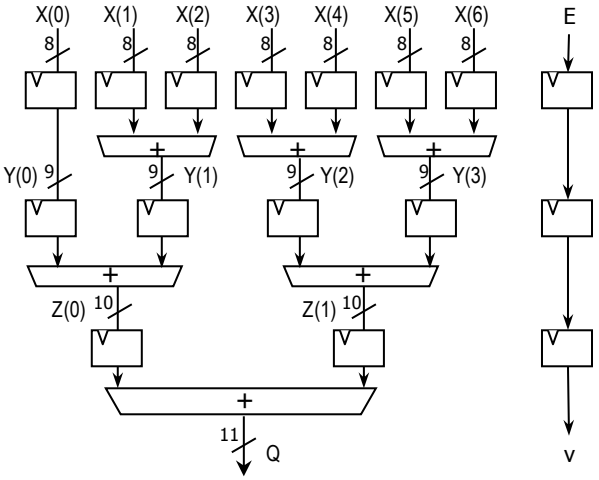
- **Adder Tree**: $\lceil \log_2 N \rceil$ register levels (or I/O delay). This is the same as the Initial latency.
- Note: For $N = 7$, we do not omit a register on the second register level when there is no adder. This is called a synchronization register and it makes sure that data arrives at the correct time.
- Computation time for T groups of N numbers: $T + \lceil \log_2 N \rceil$ cycles.

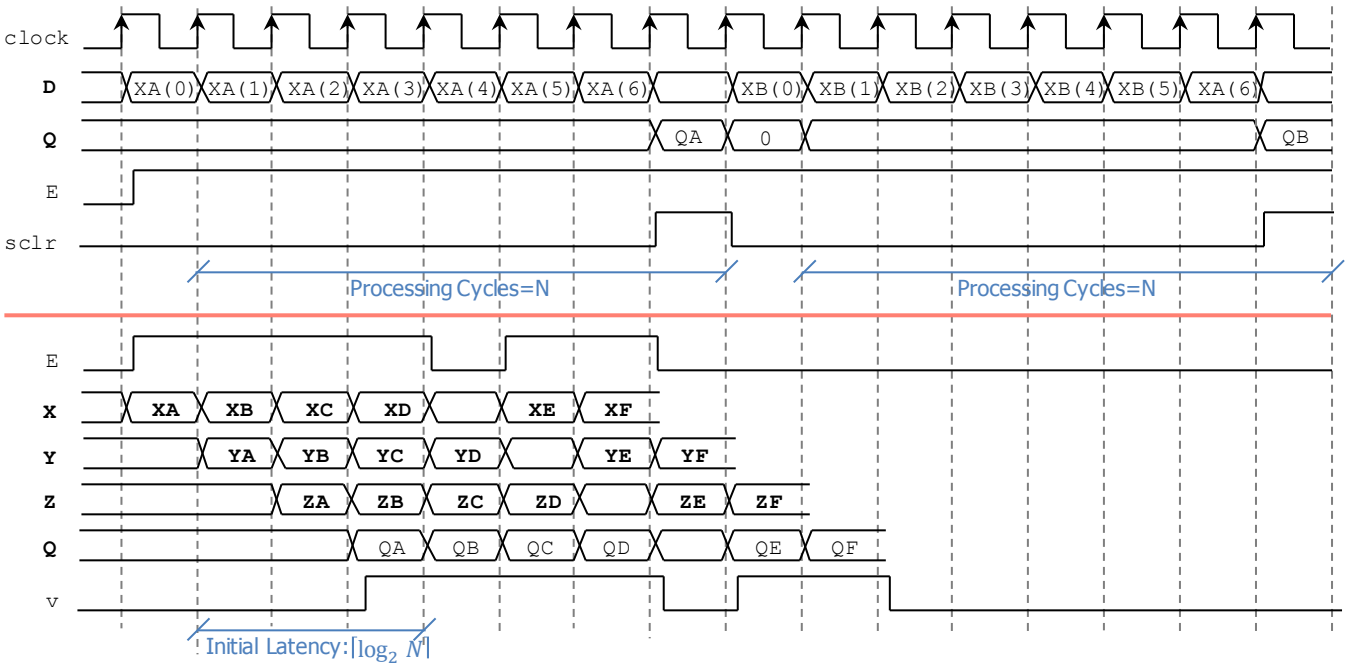


Timing Comparison

- An enable and a valid bit are added to the pipelined design. This is done via a $\lceil \log_2 N \rceil$-bit shift register.





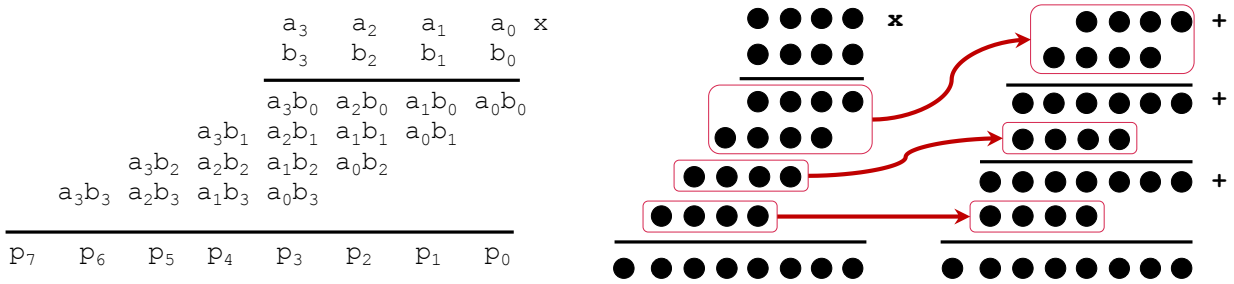


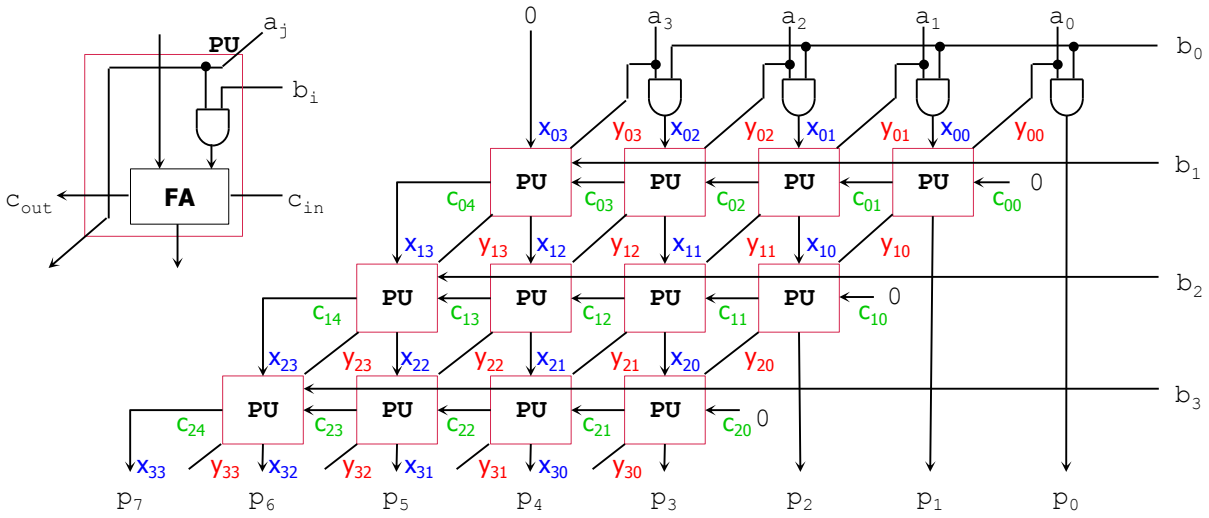
MULTIPLICATION

UNSIGNED MULTIPLICATION

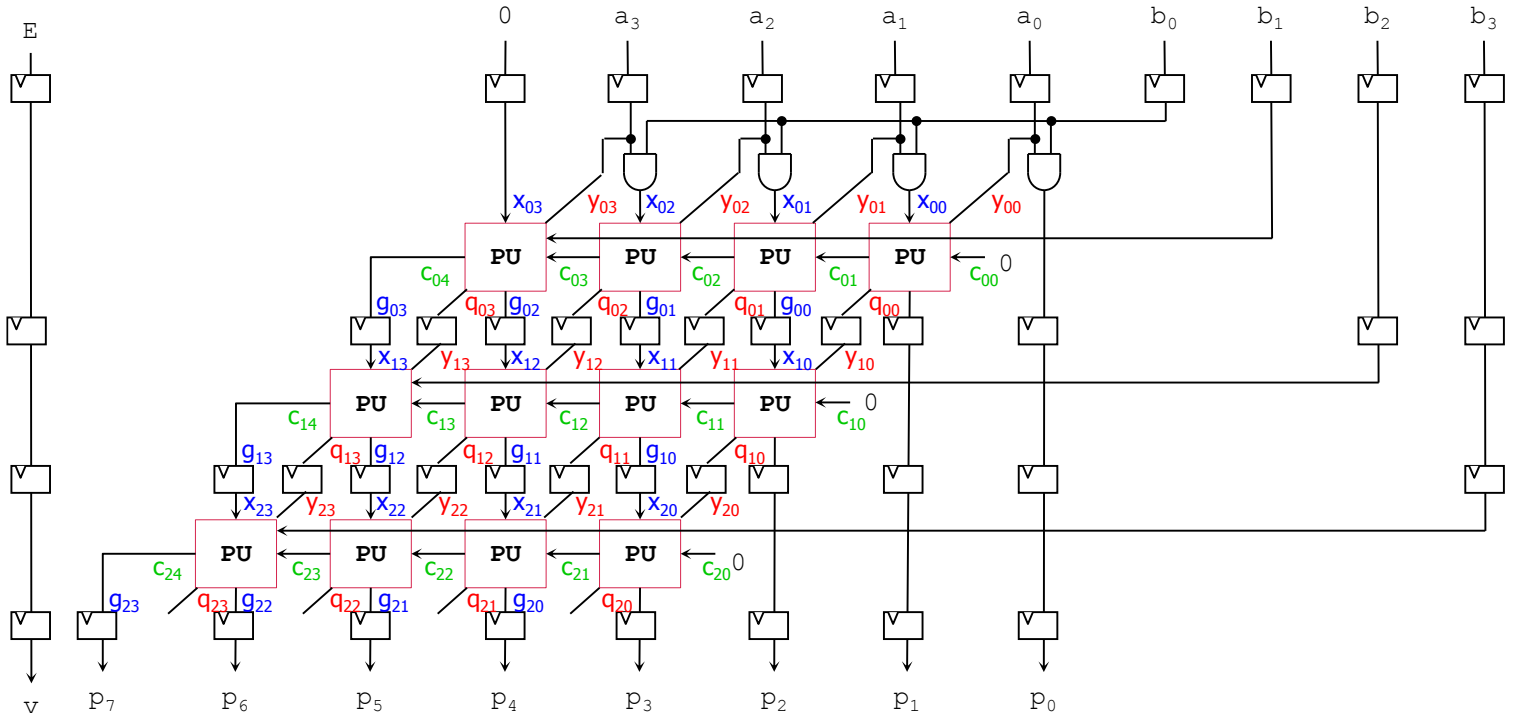
- We already know the iterative version of the multiplier. Here, we show how to implement the multiplication using an array multiplier. In this implementation, two rows are added up at each stage.
- We start from the iterative version (Unit 2) of the multiplier. As in the case of the Accumulator, if we directly unfold the iterative multiplier, the resulting architecture will not be optimal. Here we show an optimized architecture.

- **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.
- Note the synchronization registers included to make sure that data arrives at the right time. This applied to the input bits b_3-b_0 and output bits p_2, p_1, p_0 and p_7.



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

- This is based on the iterative algorithm for dividers presented in Unit 2. The architecture was unfolded and then optimized.

RESTORING ARRAY DIVIDER FOR UNSIGNED INTEGERS

- A, B: positive integers in unsigned representation. $A = a_{N-1}a_{N-2}\dots a_0$ with N bits, and $B = b_{M-1}b_{M-2}\dots b_0$ 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, $i = 0, \dots, N-1$, 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):

$$Y_{i+1} = R_i \& a_{N-1-i}, i = 0, \dots, N-1$$

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 $R_i = Y_i$.

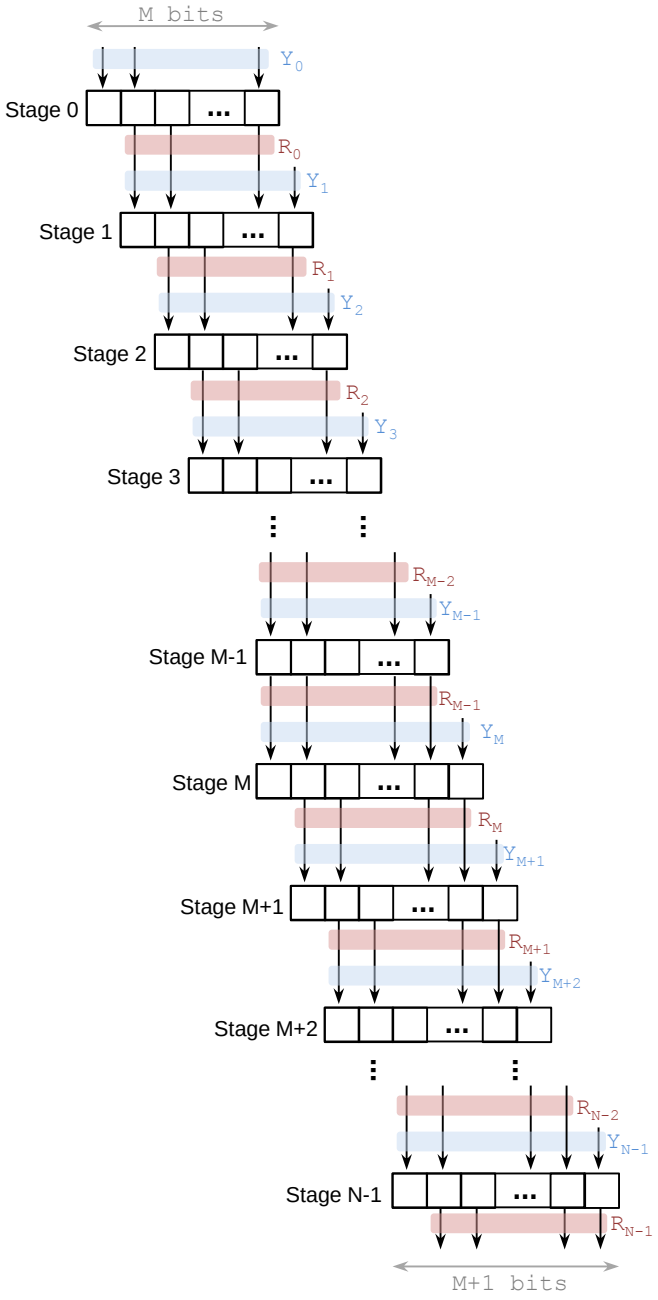
Stage	Y_i	Computation of R_i	# of R_i bits
0	$Y_0 = a_{N-1}$	$R_0 = Y_0 - B, if\ Y_0 \geq B$ $R_0 = Y_0, if\ Y_0 < B$	1
1	$Y_1 = R_0 \& a_{N-2}$	$R_1 = Y_1 - B, if\ Y_1 \geq B$ $R_1 = Y_1, if\ Y_1 < B$	2
2	$Y_2 = R_1 \& a_{N-3}$	$R_2 = Y_2 - B, if\ Y_2 \geq B$ $R_2 = Y_2, if\ Y_2 < B$	3
…	…	…	…
M-1	$Y_{M-1}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

<u>For stages 0 *to* $M-1$</u>:
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.

<u>Stages M *to* $N-1$:</u>
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.



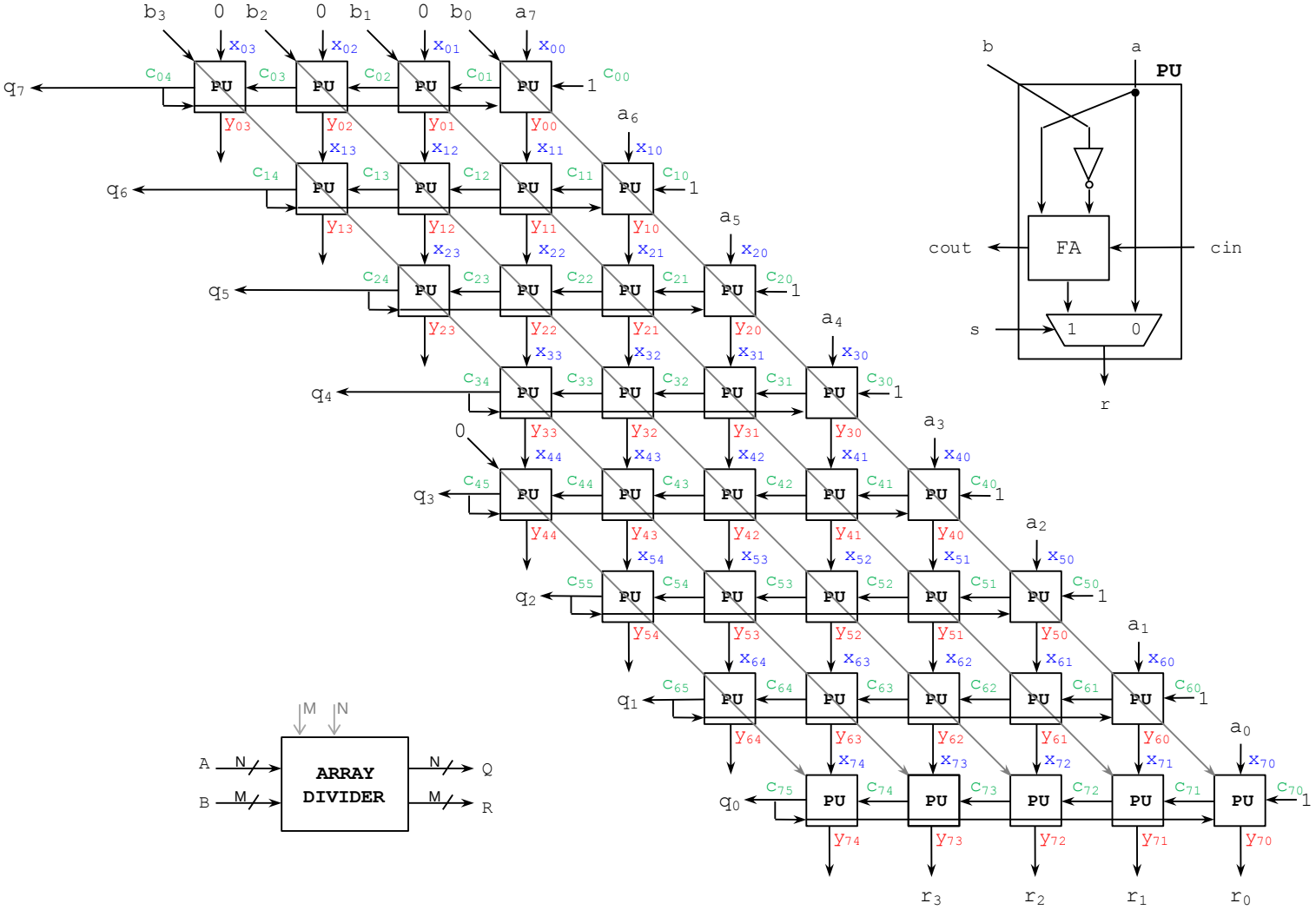
Parallel implementation algorithm

- 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 $\mathrm{R} = a_{M-1}a_{M-2}\dots a_0$.

COMBINATIONAL ARRAY DIVIDER (UNFOLDED)

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.

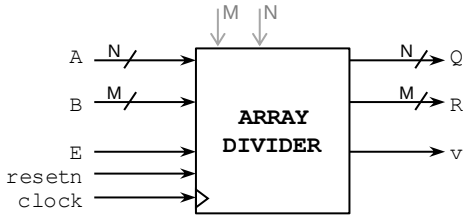
- Each level computes R_i. It first computes $Y_i - B$. When $Y_i \geq B \rightarrow cout_i = 1$, and when $Y_i < B \rightarrow cout_i = 0$. This $cout_i$ is used to determine whether the next R_i is $Y_i - B$ or Y_i.
- Each Processing Unit (PU) is used to process $Y_i - B$ one bit at a time, and to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

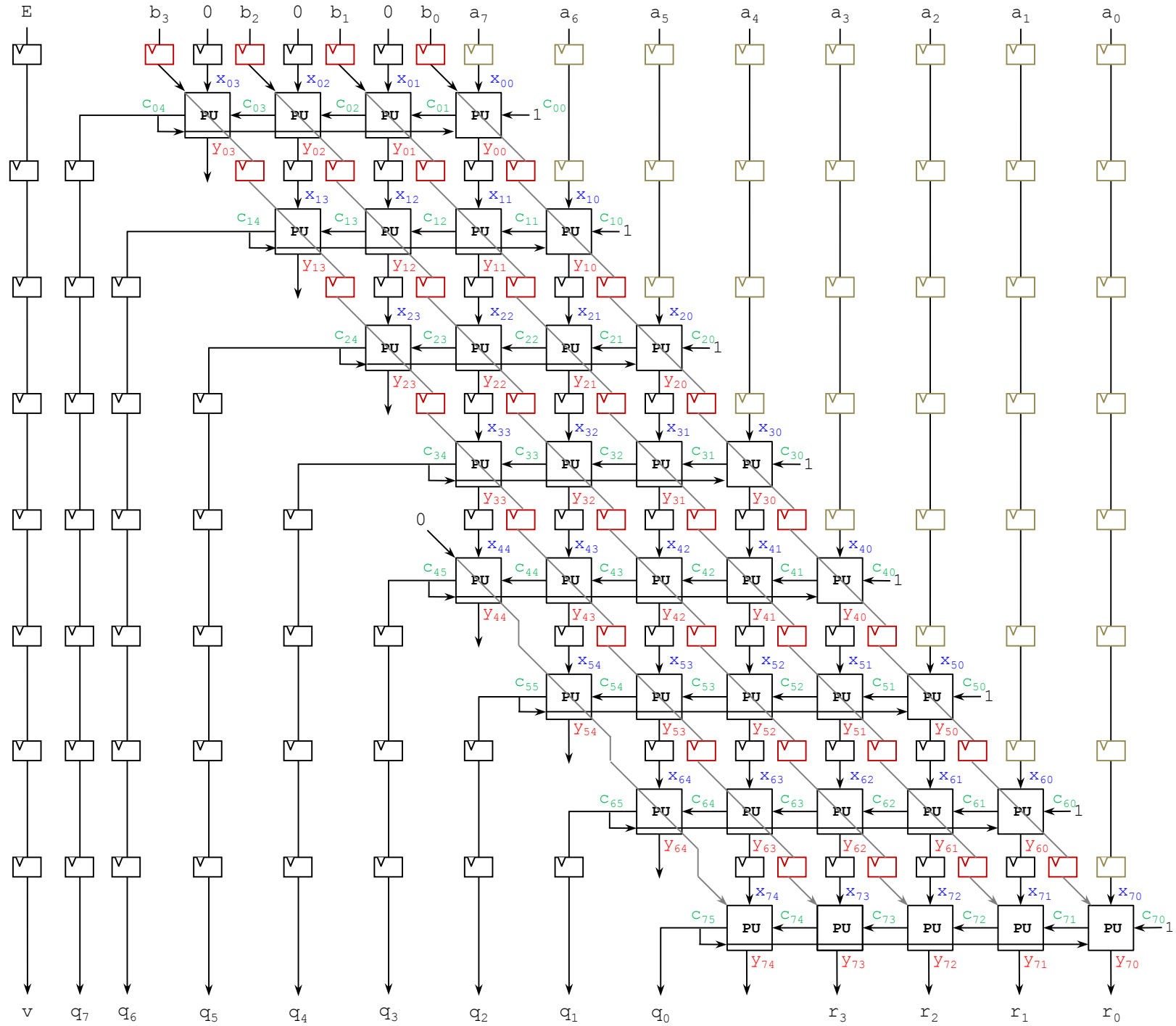
FULLY PIPELINED ARRAY DIVIDER

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 4.
- **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:

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

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

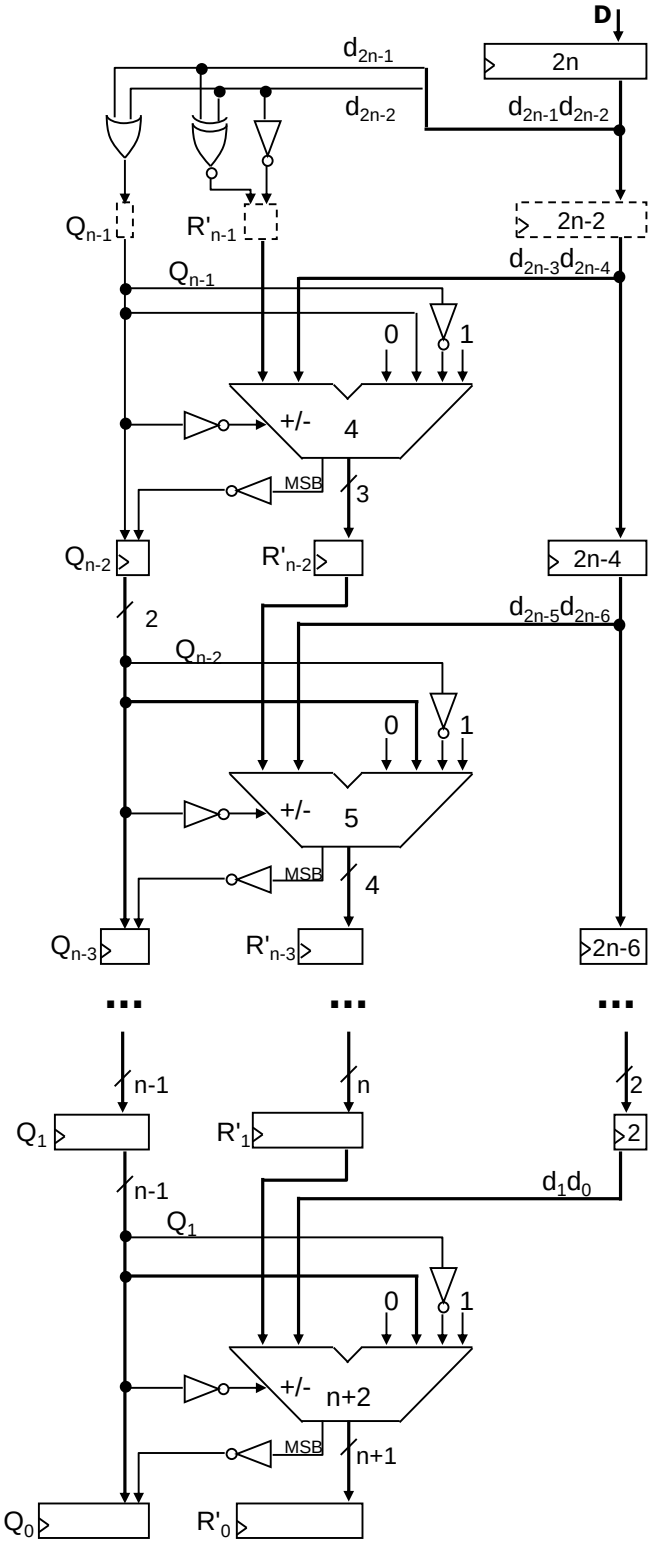
$\rightarrow q_{n-1} = d_{2n-1}d_{2n-2}$, $b = \overline{d_{2n-1} \oplus d_{2n-2}}$, $a = \overline{d_{2n-2}}$

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 for fixed point arithmetic.
- **Unfolding**: This is a very straightforward operation: we just repeat each iteration of the iterative CORDIC architecture. No optimization is applied. The output of each iteration becomes the input of the next iteration.
- **Pipelining**: It consists of adding registers between stages. The initial latency is N cycles, where N is the number of CORDIC iterations. We can feed new data (x_0, y_0, z_0, mode) at every clock cycle. N cycles after the first operation, this circuit can produce output data (x_N, y_N, z_N) every clock cycle.