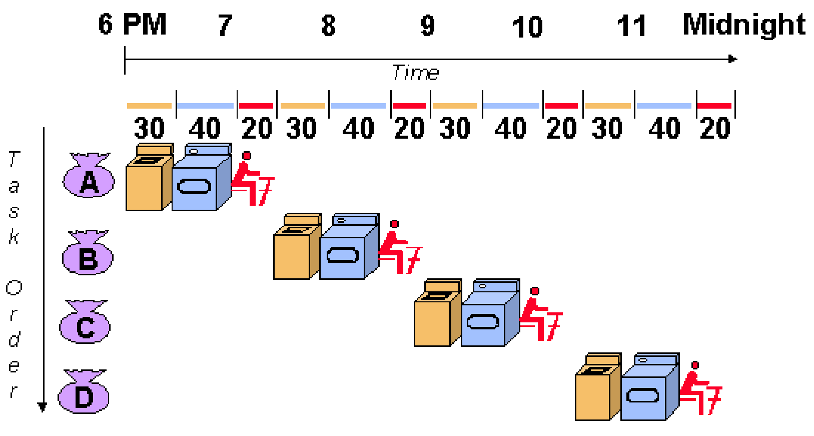
Unit 5- Pipelining and Unfolding

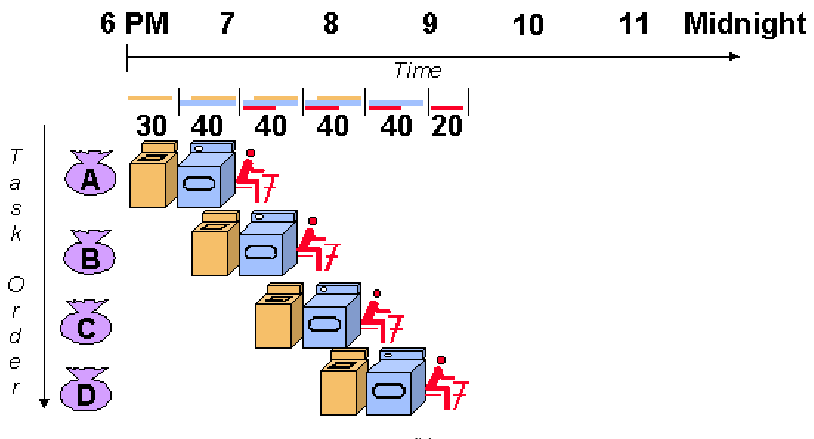
PIPELINING

- Common parallel pattern that mimics a traditional manufacturing assembly line.

  Laundry analogy explanation:
  "A useful method of demonstrating this is the laundry analogy. Let's say that there are four loads of dirty laundry that need to be washed, dried, and folded. We could put the first load in the washer for 30 minutes, dry it for 40 minutes, and then take 20 minutes to fold the clothes. Then pick up the second load and wash, dry, and fold, and repeat for the third and fourth loads. Supposing we started at 6 PM and worked as efficiently as possible, we would still be doing laundry until midnight.
  However, a smarter approach to the problem would be to put the second load of dirty laundry into the washer after the first was already clean and whirling happily in the dryer. Then, while the first load was being folded, the second load would dry, and a third load could be added to the pipeline of laundry. Using this method, the laundry would be finished by 9:30."



(a)



(b)

Figure 1. Pipeline explanation. (a) normal sequential operation. (b) pipeline approach.
Source: *https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/pipelining/index.html*

PIPELINE MODEL

- Pipeline: linear sequence of stages. Data flows through the pipeline, from the first stage to the last stage.
  - ✓ Stages of the pipeline can often be generated by using functional decomposition of tasks in an application.
  - ✓ Data is partitioned into pieces (also called items or data units).
  - ✓ Each stage performs a transformation on the data (this transformation is called a task).
  - ✓ A stage's transformation of items may be one-to-one or more complicated.
  - ✓ Stages in a pipeline can be balanced (uniform processing time) or non-balanced (non-uniform).
  - ✓ Type of pipeline stages:
    - Serial stage: It processes one item at a time, though different stages can run in parallel.
    - Parallel stage: It processes multiple items at once and can deliver output items out of order.

- Pipelines can be classified depending on the type of stages they contain:
  - ✓ Serial Pipeline: Pipeline with only serial stages. The throughput of the pipeline is limited to the throughput of the slowest serial stage because every item must pass through that stage at a time.
  - ✓ Parallel Pipeline: This pipeline includes parallel stages (it might include serial stages as well) to make it more scalable.

- Pipelines are found in:
  - ✓ Instruction pipelines: The processor breaks the execution of an instruction into stages. Results of one stage are fed onto the next stage. This allows multiple instructions to be in different stages of processing at the same time.
  - ✓ Hardware pipelines: A digital circuit is divided into stages, results of one stage are fed into the inputs of the next stage.
  - ✓ Software pipelines: A software routine can be thought of as a sequence of computing processes with the output stream of one process being fed as the input stream of the next one.

SERIAL PIPELINE

- Fig. 2 shows a pipeline with 4 stages. Data is fed to the pipeline in terms of data units (or items). For example, for data unit 'a', Stage 1 applies a transform like S1(a), while Stage 2 applies a transform like S2(S1(a)), and so on. We call this a serial pipeline, where each stage can only process one data unit at a time.

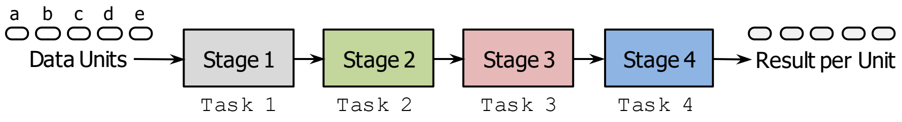


Figure 2. 4-stage serial pipeline. Each task is performed by a separate stage. The example shows 5 data units that go through the pipeline along with the final result per data unit.

Pipeline with Uniform Stages

- Here, each stage has a uniform processing time of T cycles. Fig. 3 depicts an example with 5 data units and 4 stages.
  - ✓ Sequential pipeline execution: Naïve approach depicted in Fig. 3(a). We feed the first data unit 'a' and wait until we get the result from Stage 4. Then, we feed data unit 'b' and wait until we get the result from Stage 4. This repeats until we feed the last data unit ('e') and get the associated result from Stage 4. Total computation time: $(5 \times 4) \times T$ cycles.
  - ✓ Concurrent pipeline execution: Depicted in Fig. 3(b). If we continuously feed a new data unit right after Stage 1 has processed a previous data unit, we can expose parallelism (all stages busy after a little while). Total computation time: $(4 + 5 - 1) \times T = 8T$ cycles. This large reduction in computation time is an advantageous feature of pipelining.

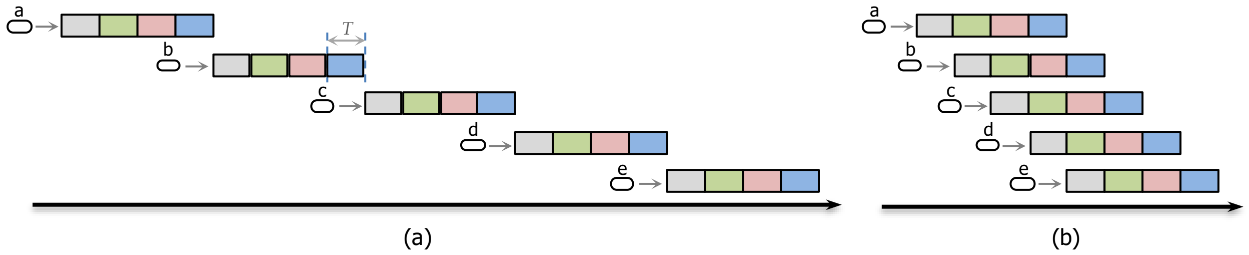


Figure 3. (a) Sequential pipeline execution for 5 data units: it takes 20×T cycles. (b) Concurrent parallel execution for 5 data unit: it takes 8×T cycles. Note how all stages are busy after some initial delay.

- For a pipeline with q stages (each with a processing time T) that is continuously fed n data units, we have that:
  - ✓ Latency (total time for one item to go through the whole system): $q \times T$. This is also called initial latency (number of cycles it takes to process the first data unit).
  - ✓ Total Processing Time: $(q + n - 1) \times T$ cycles.
  - ✓ Throughput (rate at which items are processed, in terms of data units per cycle): $\frac{n}{(q+n-1)\times T} = \frac{1}{\left(\frac{q-1}{n}+1\right)\times T}$
    - ▫ In practice, n can be very large and thus the throughput is given by: $\left.\frac{1}{\left(\frac{q-1}{n}+1\right)\times T}\right|_{n\to\infty} = \frac{1}{T}$ data units per cycle. This can be interpreted as the rate at which new items are processed after the first one (i.e., after the initial latency).

Pipeline with Non-Uniform Stages

- When the processing times of the stages are non-uniform, the slowest stage limits the throughput. Unlike the case with uniform stages, here we cannot guarantee that all stages will be necessarily operating at the same time.
  - ✓ Fig. 4(a) depicts the case where Stage 3 takes $1.5T$ cycles, while the other stages take T cycles each. The latency is $4.5T$ cycles. The pipeline must wait until Stage 3 computes its result before feeding a new data to Stage 3. Thus, the processing time is given by $(4.5 - 1) \times T + (n - 1) \times 1.5T + T = 4.5 \times T + (n - 1) \times 1.5T$. The throughput is given by: $\frac{n}{(4{,}5+(n-1)\times 1.5)\times T} = \frac{1}{\left(\frac{4.5-1.5}{n}+1.5\right)\times T}$. When $n \to \infty$, the throughput results in $\frac{1}{1.5T}$.
  - ✓ Fig. 4(b) depicts the case where Stage 2 takes $2T$ cycles, while the other stages take T cycles each. The latency is $5T$ cycles. The pipeline must wait until Stage 2 computes its result before feeding a new data to Stage 2. Thus, the total processing time is given by $(5 - 2) \times T + (n - 1) \times 2T + 2T = 5 \times T + (n - 1) \times 2T$. The throughput is given by: $\frac{n}{(5+(n-1)\times 2)\times T} = \frac{1}{\left(\frac{5-2}{n}+2\right)\times T}$. When $n \to \infty$, the throughput results in $\frac{1}{2T}$.

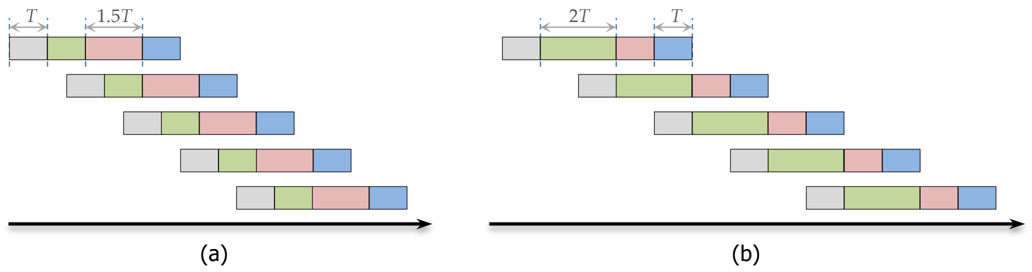


Figure 4. Pipelining for non-uniform stages. (a) Largest stage takes 1.5T cycles. Here, all the stages are busy at one point. (b) Largest stage takes 2T cycles. Here, at most only 3 stages are busy at a time.

- In general (for q stages and n data items), the total processing time is given by $L \times T + (n-1) \times f \times T$ cycles, where $f \times T$ is the processing time of the largest stage ($f > 1$) and $L \times T$ is the latency. Note that this formula holds even if the other stages are unbalanced. Also, a balanced pipeline (T cycles per stage) is a special case where $L = q$ and $f = 1$.
  - ✓ Throughput: $\frac{n}{(L+(n-1)\times f)\times T} = \frac{1}{\left(\frac{L-f}{n}+f\right)\times T}$ data units per cycle. When $n \to \infty$, the throughput results in $\frac{1}{fT}$, and as such it is determined by the slowest stage.

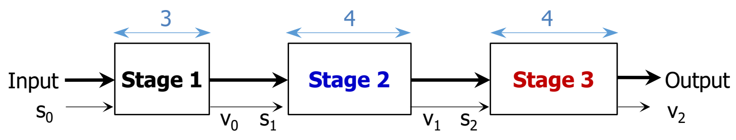
TABLE I. SERIAL PIPELINE: PROCESSING TIMES. n: NUMBER OF ITEMS.

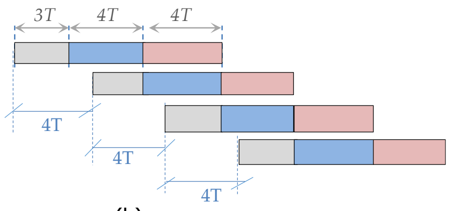
Serial Pipeline	Processing Time (cycles)	Throughput (data units per cycle)	Comments
Uniform (each stage takes T cycles)	$(q+n-1)\times T$	$\frac{n}{(q+n-1)\times T} = \frac{1}{T}, if\ n \to \infty$	q: Number of pipeline stages
Non-Uniform (at least one stage takes more than T cycles)	$L \times T + (n-1)\times f \times T$	$\frac{n}{(L+(n-1)\times f)\times T} = \frac{1}{fT}, if\ n \to \infty$	L: factor of the pipeline latency ($L \times T$) f: factor of the largest stage ($f > 1$)

APPLICATION TO HARDWARE WITH SEQUENTIAL (ITERATIVE) STAGES

- If your hardware design can be partitioned into a linear sequence of stages, we can apply pipelining in order to expose parallelism when feeding a stream of data. Pipelining allows us to optimize the rate at which we can feed new data.
- The following applies to iterative stages. In an iterative stage, if it takes P cycles to process an input sample, we have to wait P cycles before we can feed a new input sample to that stage.

- Fig. 5(a) depicts a hardware design with 3 stages. Each stage includes an 's' (start signal) and 'v' (done signal). When 's' is asserted, we feed a data sample, and when 'v' is asserted, the associated result is available.
  - ✓ Stage 1: It takes $p_1 = 3$ cycles to process data.
  - ✓ Stage 2: It takes $p_2 = 4$ cycles to process data.
  - ✓ Stage 3: It takes $p_3 = 4$ cycles to process data.
- Fig. 5(b) depicts a theoretical depiction of pipelining for the 3-stage circuit (assume $T = 1$).





(a) (b)

Figure 5. (a) Hardware design depicted as (iterative) stages along with their processing time. (b) Pipelining depiction: we can feed data samples every 4 cycles (at least)

- As per the pipelining formulas, we can issue a new sample every $F = \max(p_1, p_2, p_3) = \max(3,4,4) = 4$ cycles. This is shown in Fig. 6. Note how each sample is processed by the pipeline.

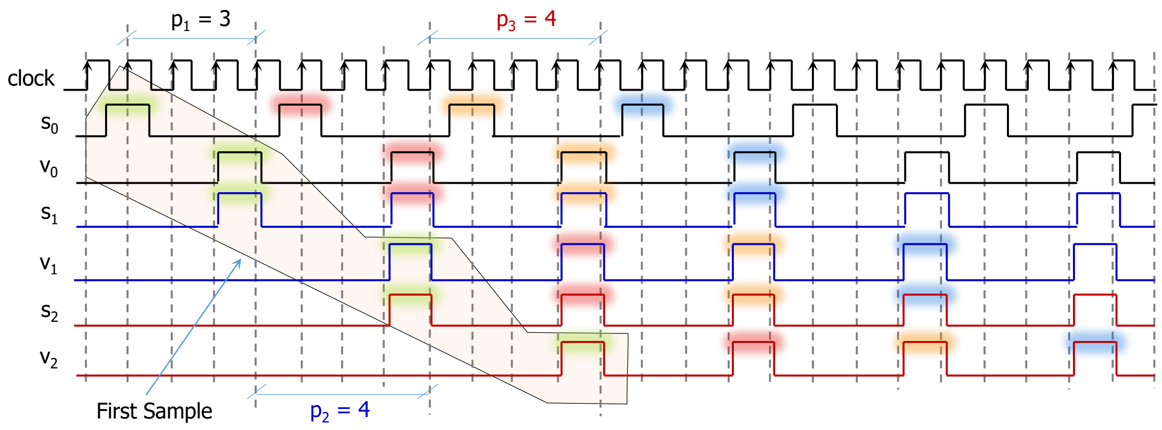


Figure 6. Pipelined Approach. We can feed a new data sample every 4 cycles

- Total processing time for n samples. Here, T is set to '1' (e.g.: $L \times T = 10$ cycles).
  - ✓ Pipelined approach: $L \times T + (n-1) \times f \times T = (4+3+3) + (n-1) \times 4$.
  - ✓ Non-pipelined approach: $L \times T \times n = 10 \times n$

- Ideally, we would like to be able to feed a new data sample every clock cycle (and retrieve output data at every clock cycle as well). These circuits are known as 'fully-pipelined' (see next section).

PIPELINING/UNFOLDING OF ITERATIVE ARCHITECTURES

- Here, we provide examples of how to convert iterative architectures/algorithms into ***fully-pipelined*** architectures. These architectures are such that we can feed a new data sample at every clock cycle.

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 $n + \lceil \log_2 N \rceil$.

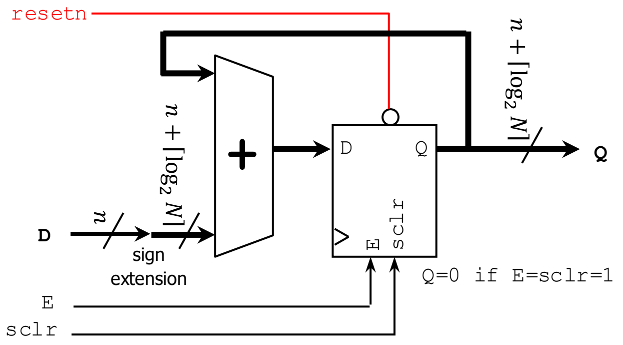


Figure 7. Accumulator Circuit

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.

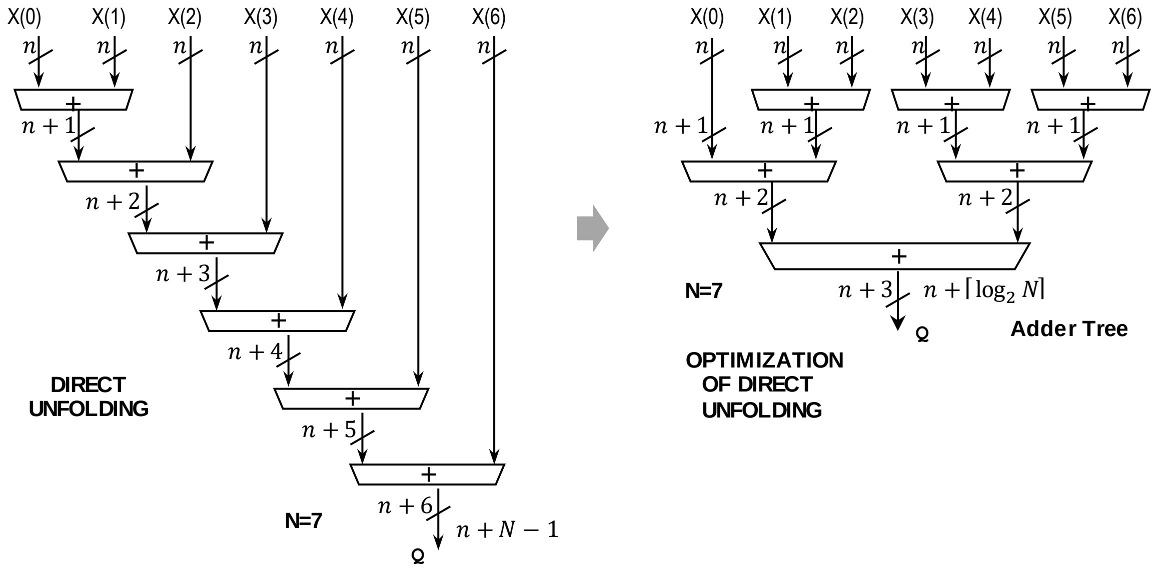

X(0) X(1) X(2) X(3) X(4) X(5) X(6)
n
$n+1$
$n+2$
$n+3$
$n+4$
$n+5$
$n+6$
$n+N-1$
Q
N=7
DIRECT UNFOLDING
X(0) X(1) X(2) X(3) X(4) X(5) X(6)
$n+1$
$n+2$
$n+3$
$n+\lceil \log_2 N \rceil$
Q
N=7
Adder Tree
OPTIMIZATION OF DIRECT UNFOLDING

Figure 8. Unfolding the Accumulator

- 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 this circuit assumes that you can feed a group of N numbers in one clock cycle.

- 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 so that we can partition the circuit into stages with a uniform processing time (one clock cycle).

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 we include depends on the architecture.
  - ✓ Initial Latency: Output data will be ready in a number of cycles (= register levels) after input data is loaded.
  - ✓ We can load new input data at every new cycle. After the initial latency, we get output results every clock cycle. Over time, the initial latency will become 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 <u>synchronization register</u> 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.

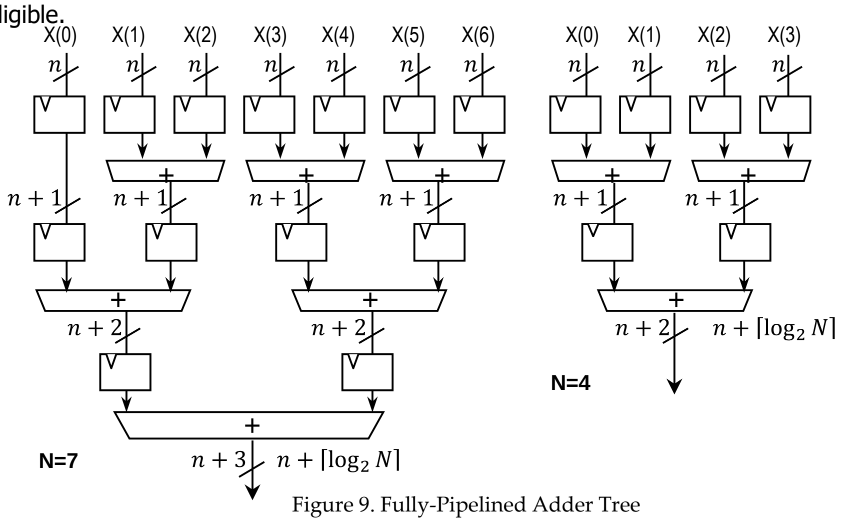


Figure 9. Fully-Pipelined Adder Tree

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.

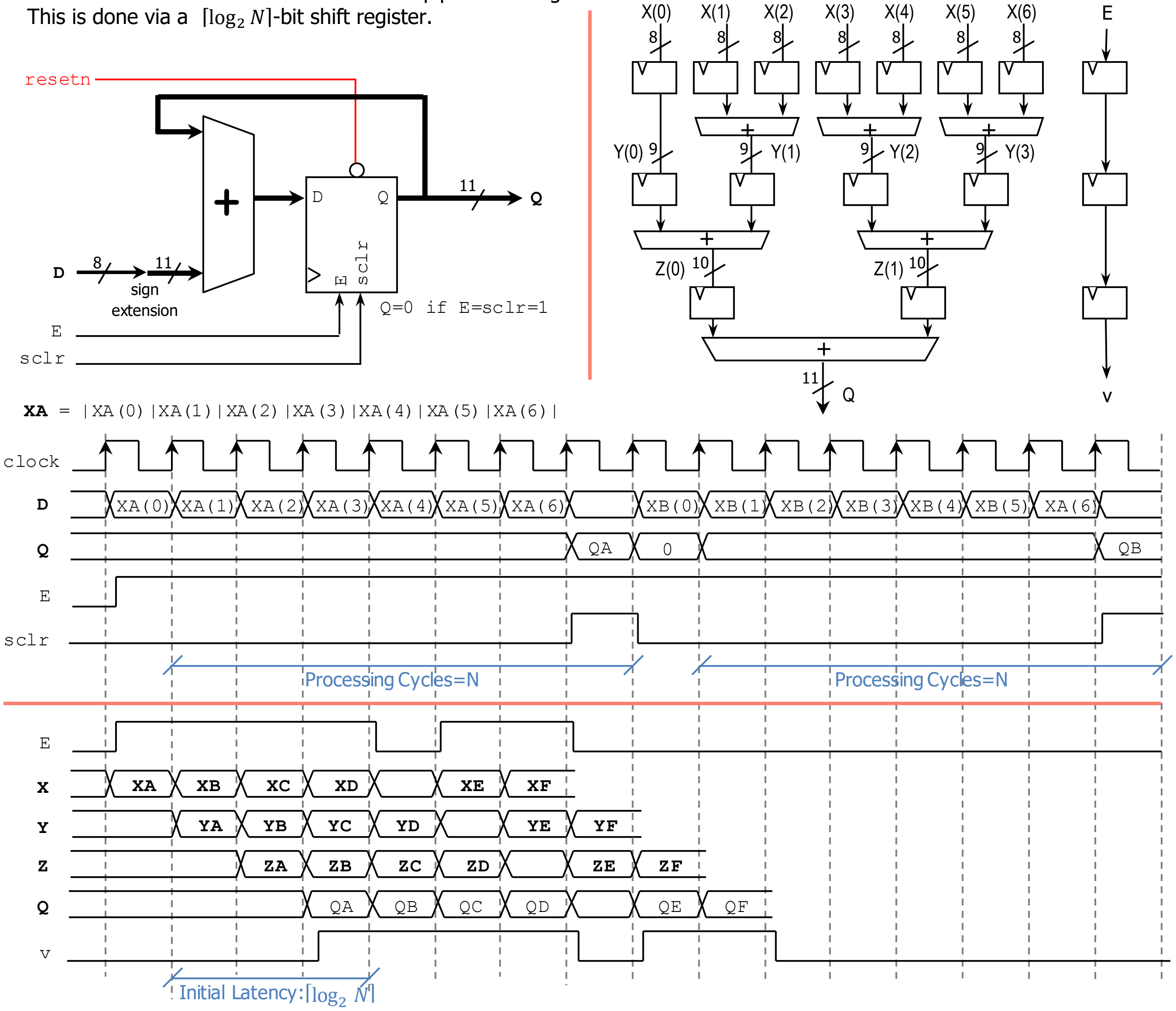


Figure 10. Processing Time comparison between a folded design and a pipelined design.

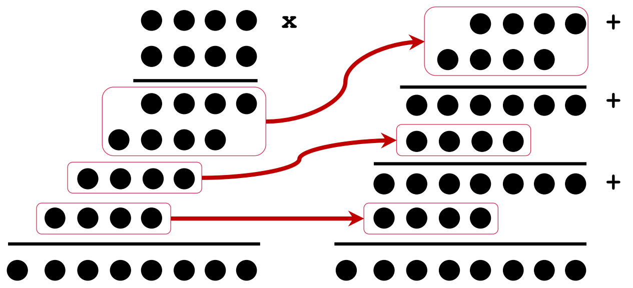
MULTIPLICATION

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.

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



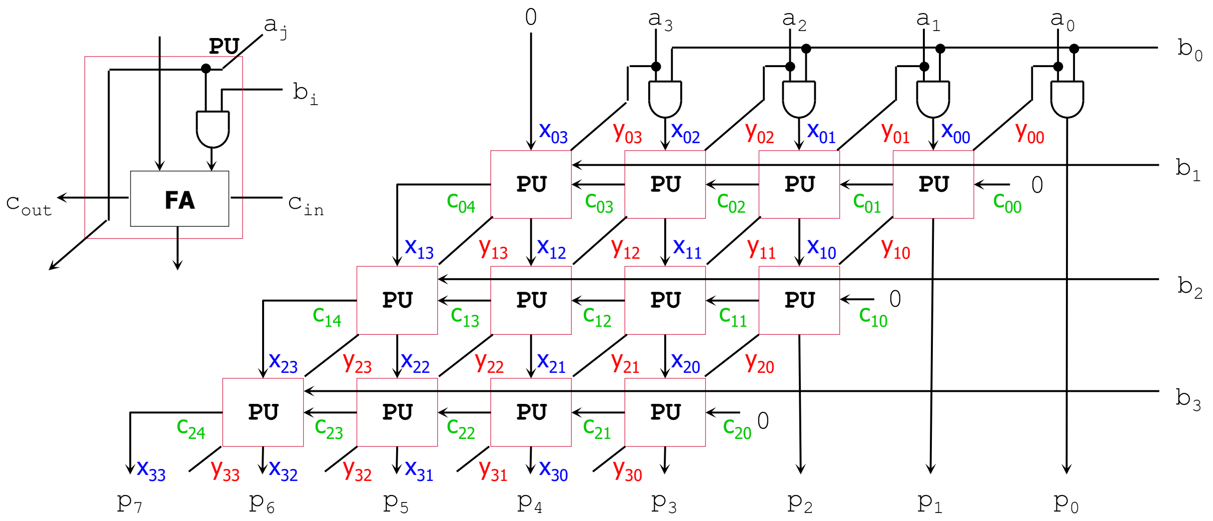

PU
a_j
b_i
FA
c_out
c_in
0
a_3
a_2
a_1
a_0
b_0
b_1
b_2
b_3
x_03 y_03 x_02 y_02 x_01 y_01 x_00 y_00
c_04 c_03 c_02 c_01 c_00
x_13 y_13 x_12 y_12 x_11 y_11 x_10 y_10
c_14 c_13 c_12 c_11 c_10
x_23 y_23 x_22 y_22 x_21 y_21 x_20 y_20
c_24 c_23 c_22 c_21 c_20
x_33 y_33 x_32 y_32 x_31 y_31 x_30 y_30
p_7 p_6 p_5 p_4 p_3 p_2 p_1 p_0

Figure 11. Multiplier: Unfolded (combinational) architecture.

- **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.

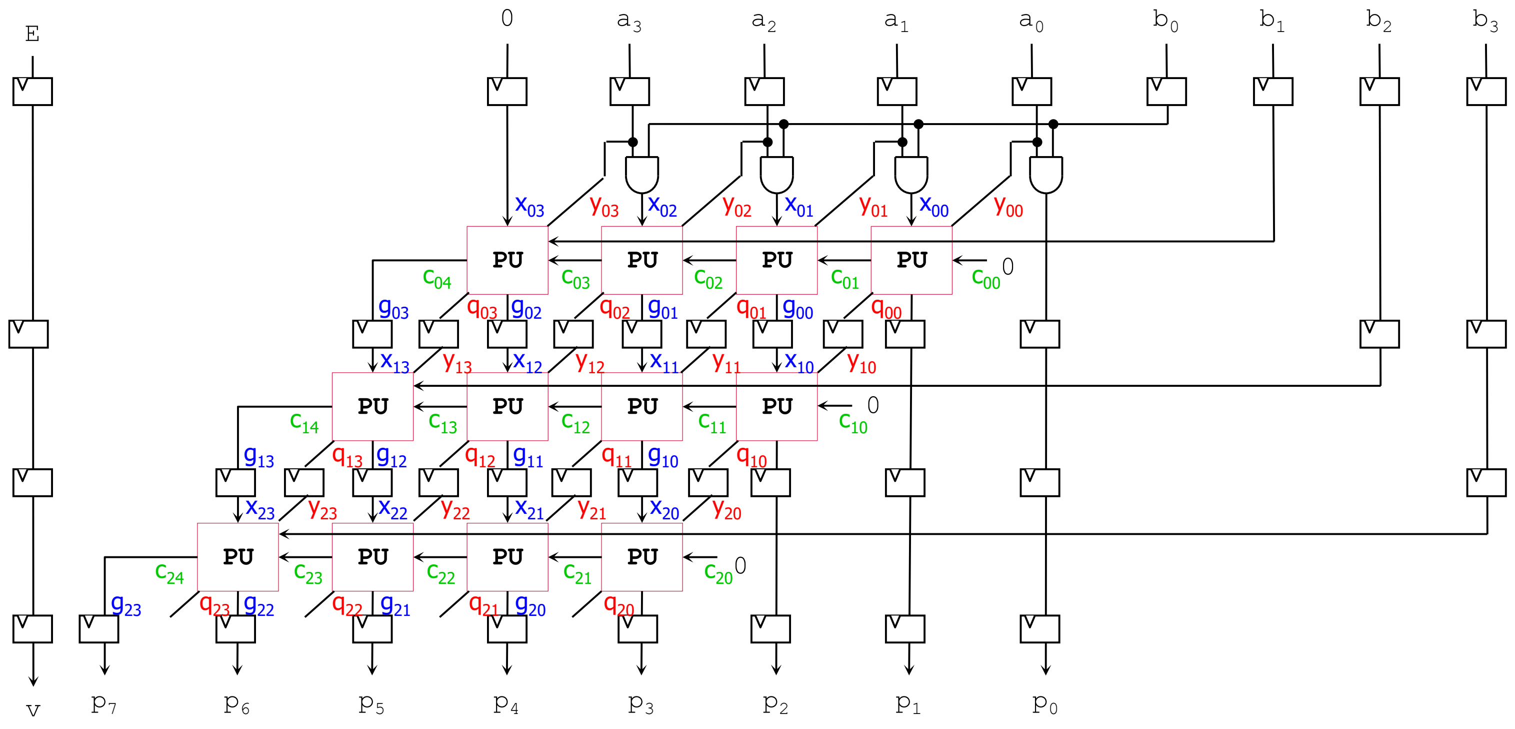

E
0
a3
a2
a1
a0
b0
b1
b2
b3
PU
x03 y03 x02 y02 x01 y01 x00 y00
c04 c03 c02 c01 c00 0
g03 q03 g02 q02 g01 q01 g00 q00
x13 y13 x12 y12 x11 y11 x10 y10
c14 c13 c12 c11 c10 0
g13 q13 g12 q12 g11 q11 g10 q10
x23 y23 x22 y22 x21 y21 x20 y20
c24 c23 c22 c21 c20 0
g23 q23 g22 q22 g21 q21 g20 q20
v p7 p6 p5 p4 p3 p2 p1 p0

Figure 12. Multiplier: Fully-pipelined architecture.

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.

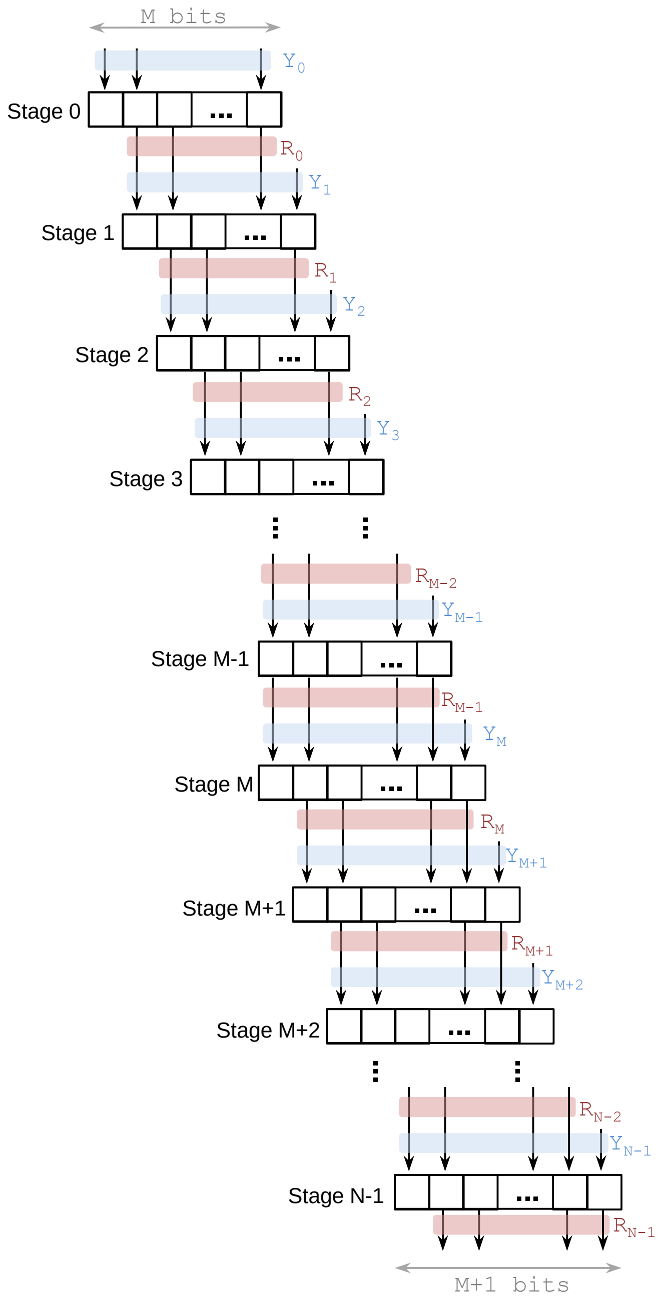


Figure 13. Divider: Parallel implementation algorithm

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

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.

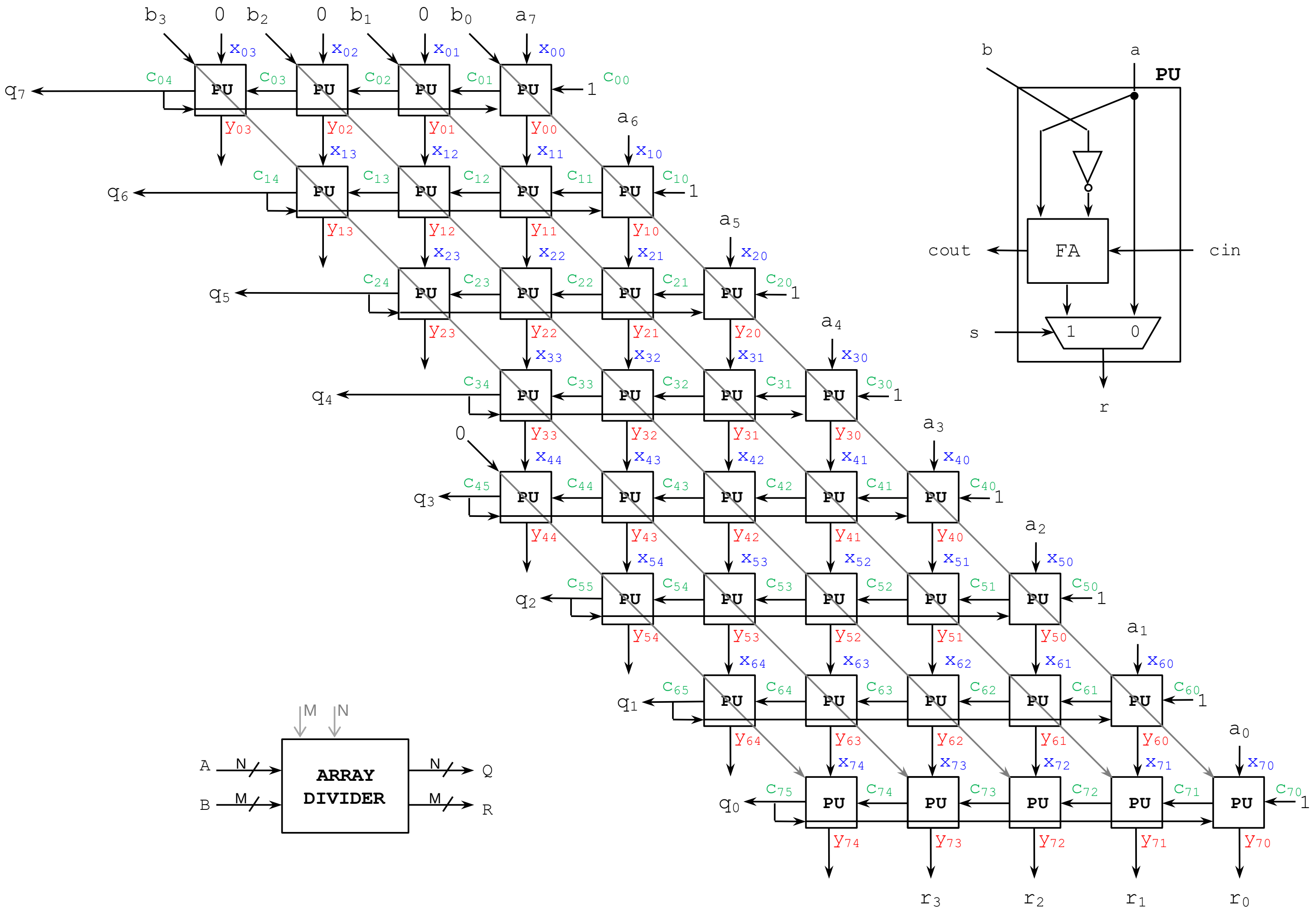


Figure 14. Fully Combinatorial Array Divider architecture for N=8, M=4

FULLY PIPELINED ARRAY DIVIDER

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

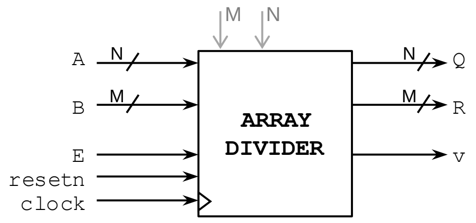


Figure 15. Fully pipelined IP core for the array divider

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).

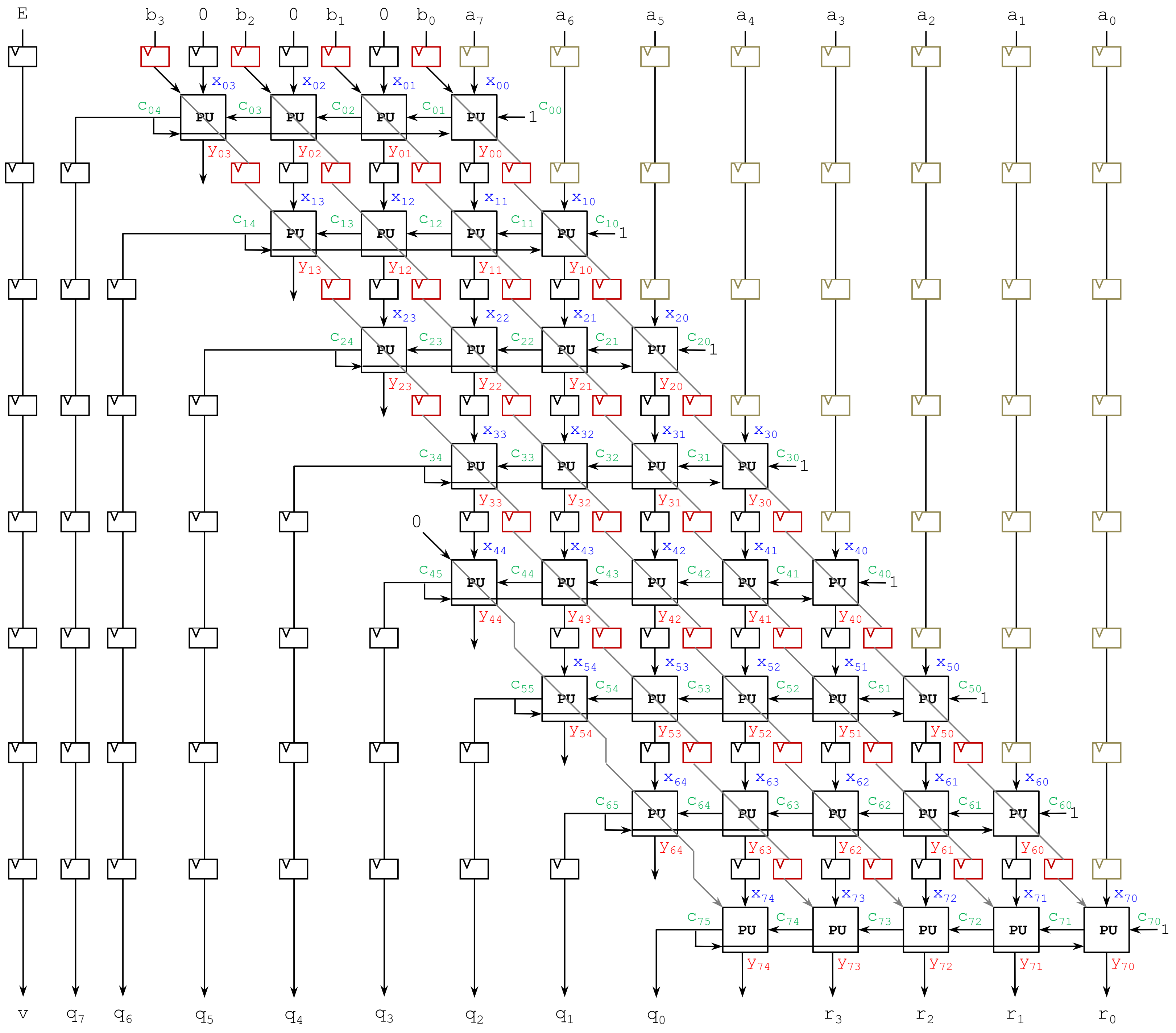


Figure 16. Fully Pipelined Array Divider architecture for N=8, M=4

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.

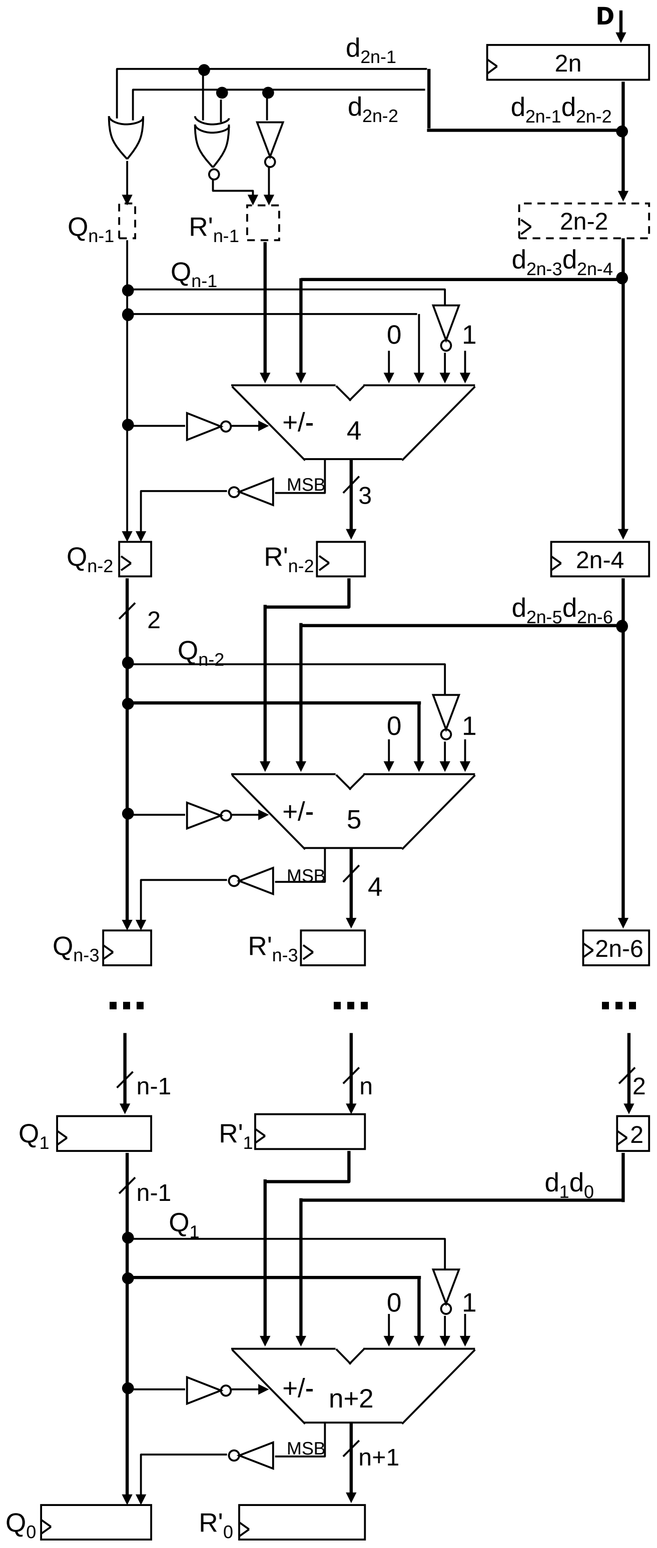


Figure 17. Fully Pipelined Architecture for Square Root Computation

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.

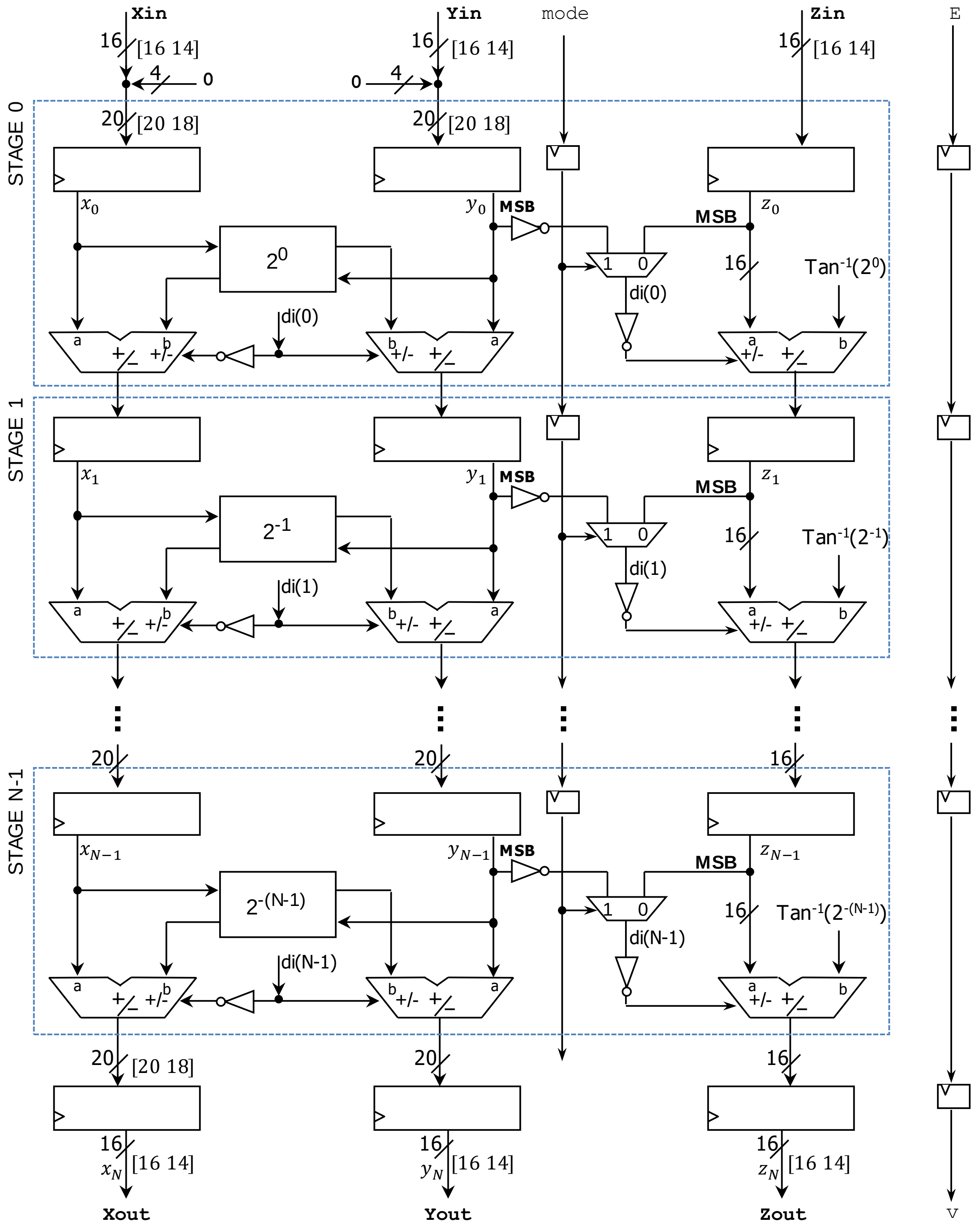


Figure 18. Fully Pipelined Architecture for Circular CORDIC Computation