Multiplication

```
     1101    multiplicand

x    0101    multiplier
-------------
     1101
    0000
   1101
  0000
-------------
  1000001
```

Multiplication

	1101		multiplicand
Shift by 0	1101	1	multiplier
Shift by 1	~~11010~~	0	
Shift by 2	110100	1	Include term
Shift by 3	~~1101000~~	0	MSB
	1000001		

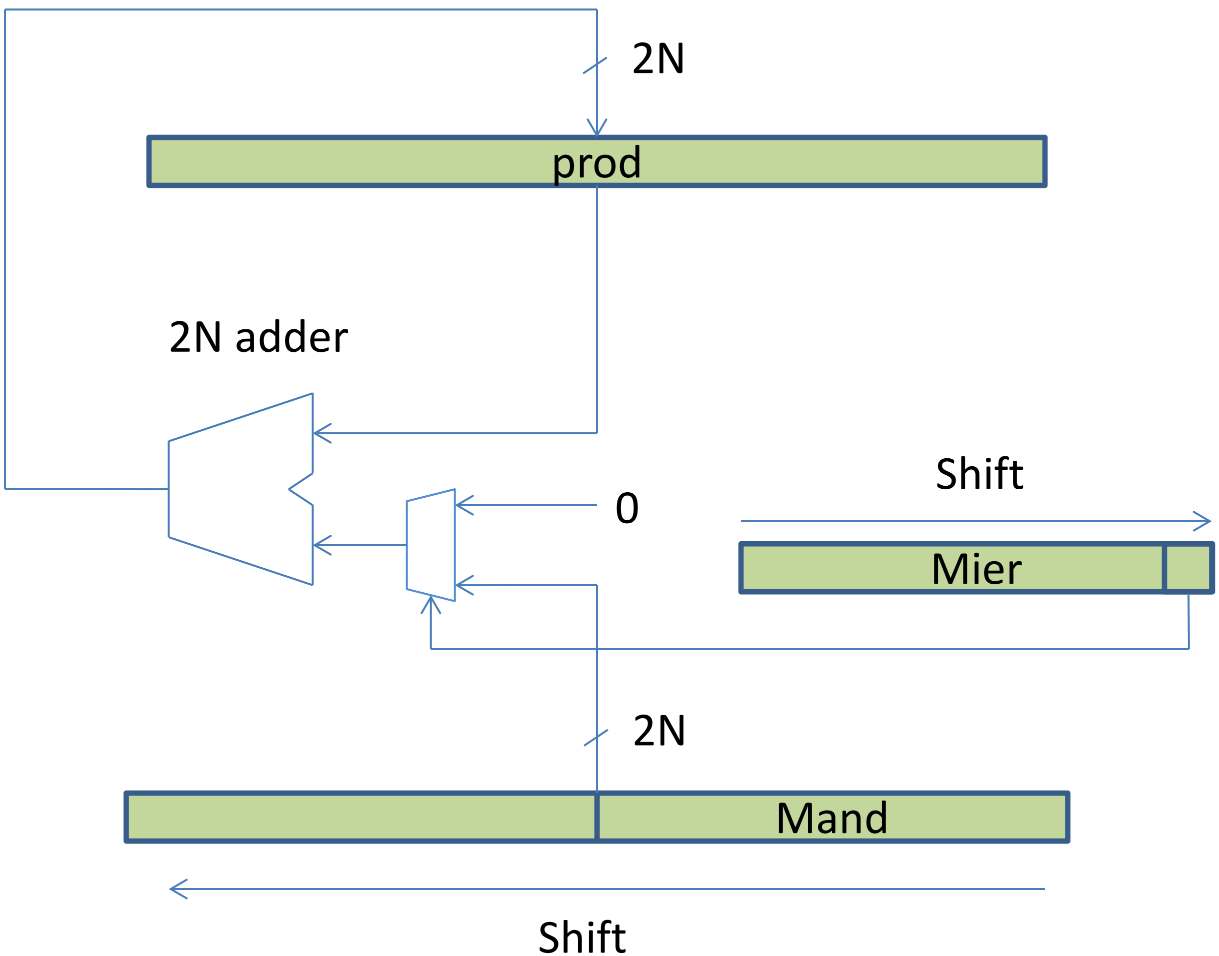
Multiplication

multiplicand		1101	RUNNING PRODUCT
			00000000
1	+	1101	=00001101
0	+	00000	=00001101
1	+	110100	=1000001
0	+	0000000	=01000001
multiplier			

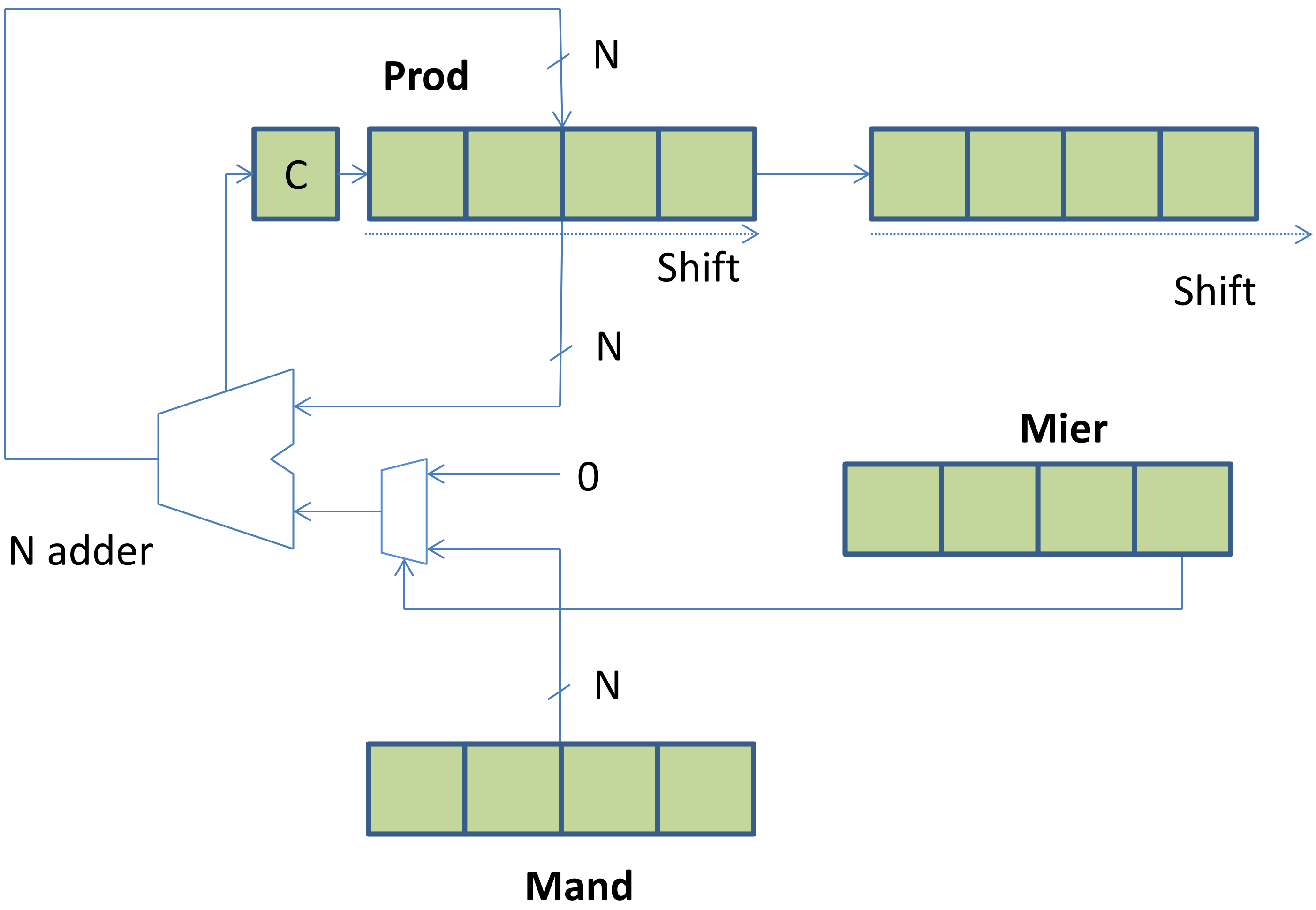
Algorithm

```
prod = 0
N = 32
while (N > 0)
{
    if (Mier & 1) prod += Mand;
    Mier >>= 1;
    Mand <<= 1;
    N--;
}
```

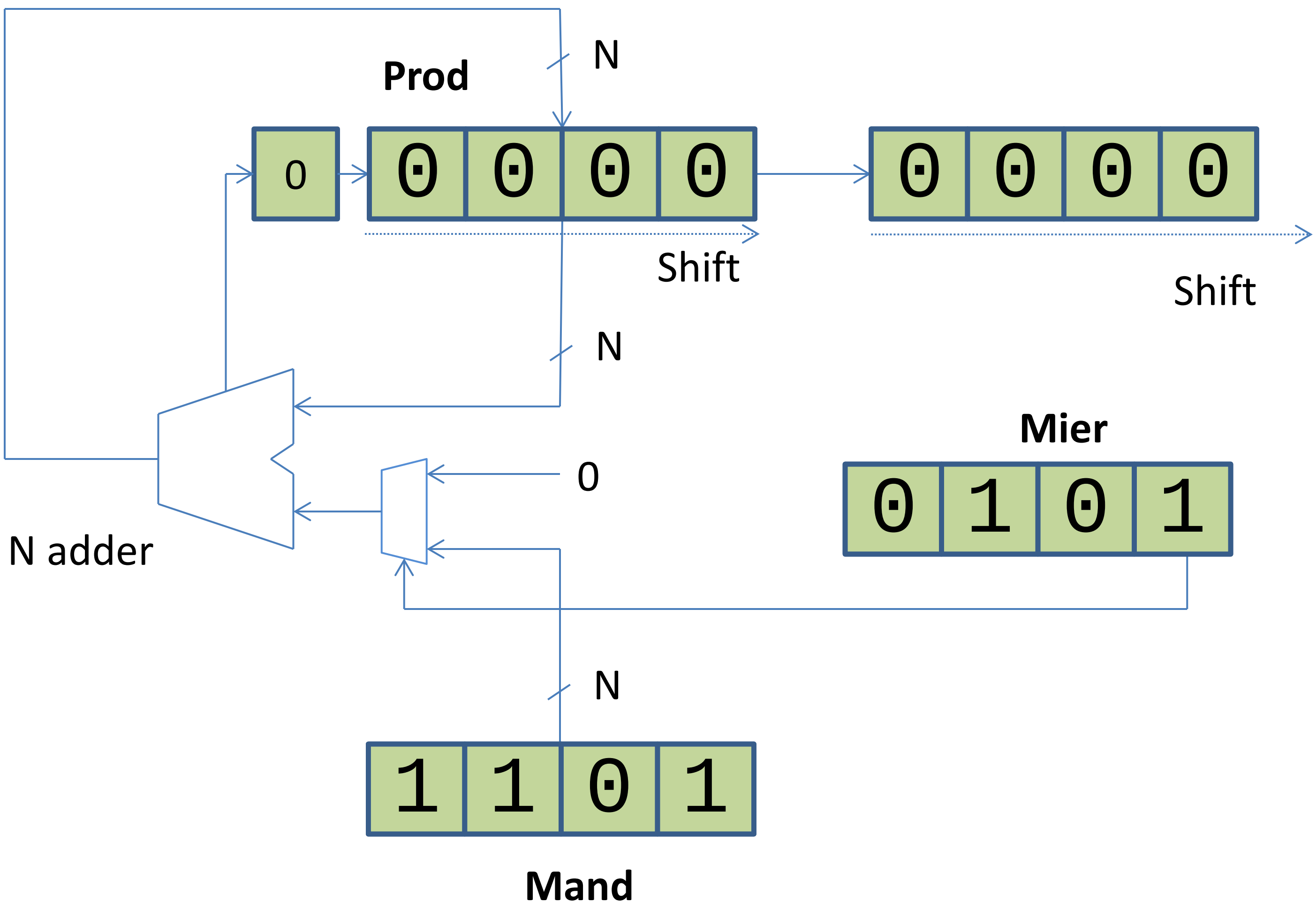
Multiplier: Take #1



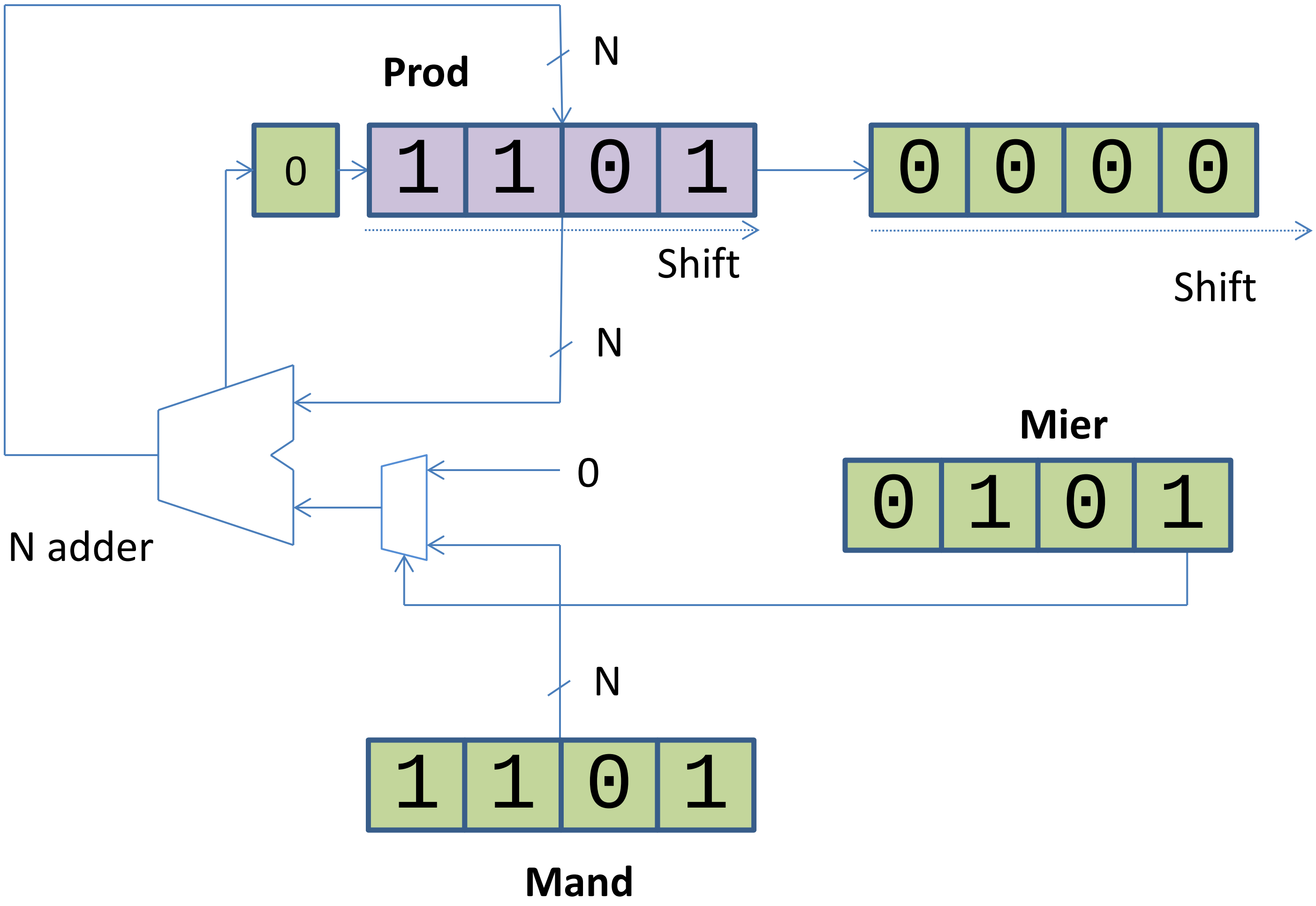
Multiplier: Take #2



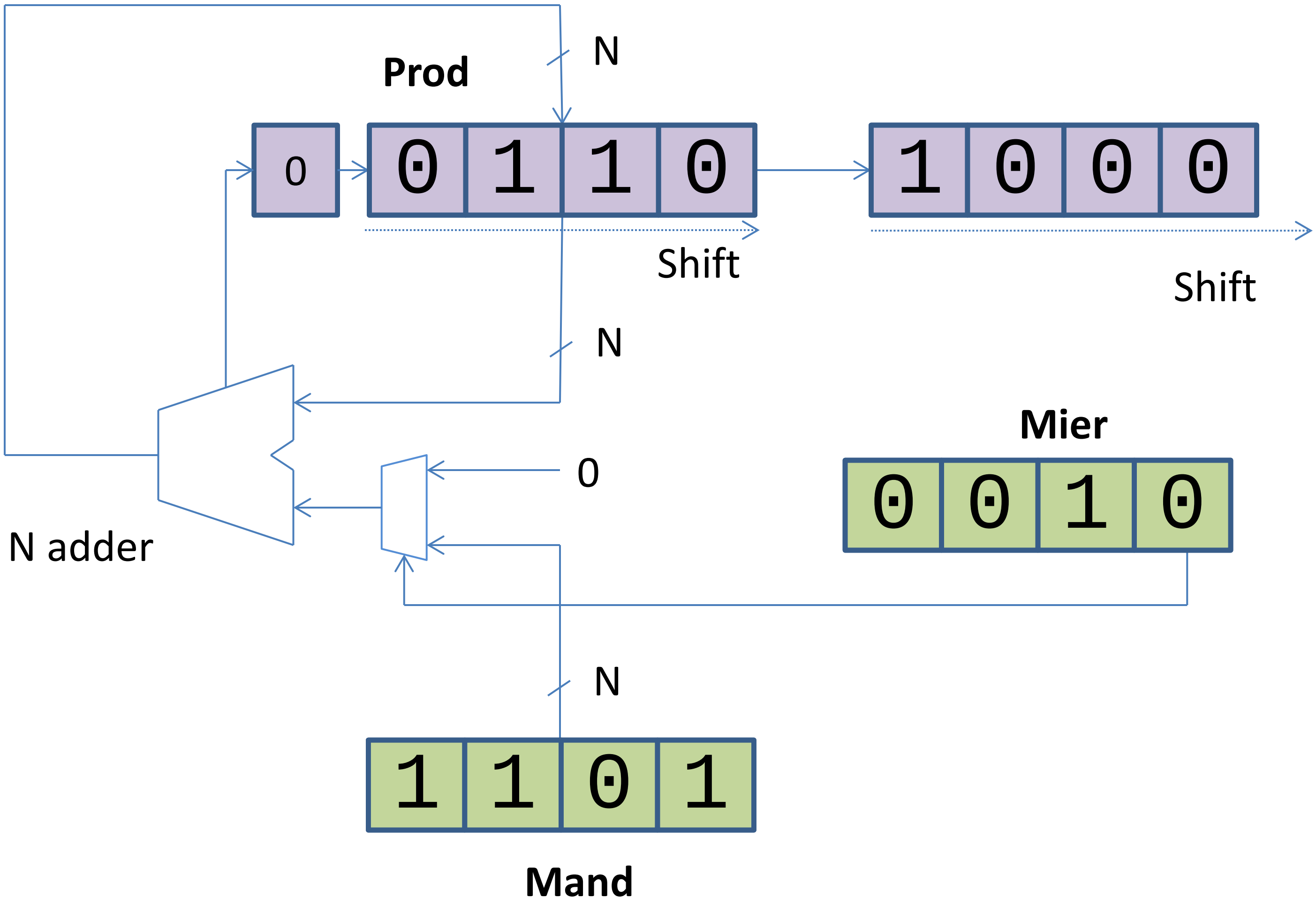
Multiplier: Initial State



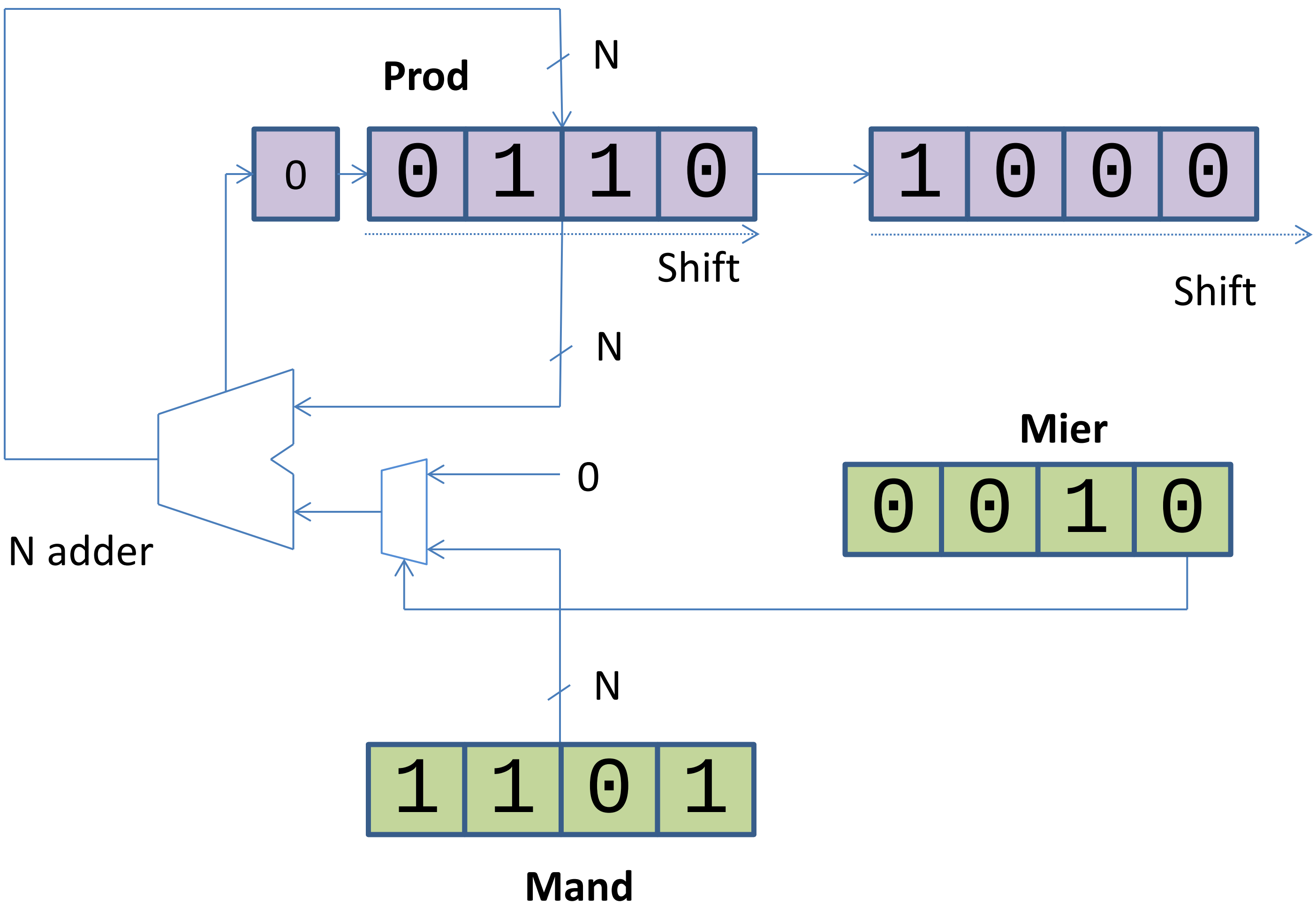
Step 1: Add Mand to Prod



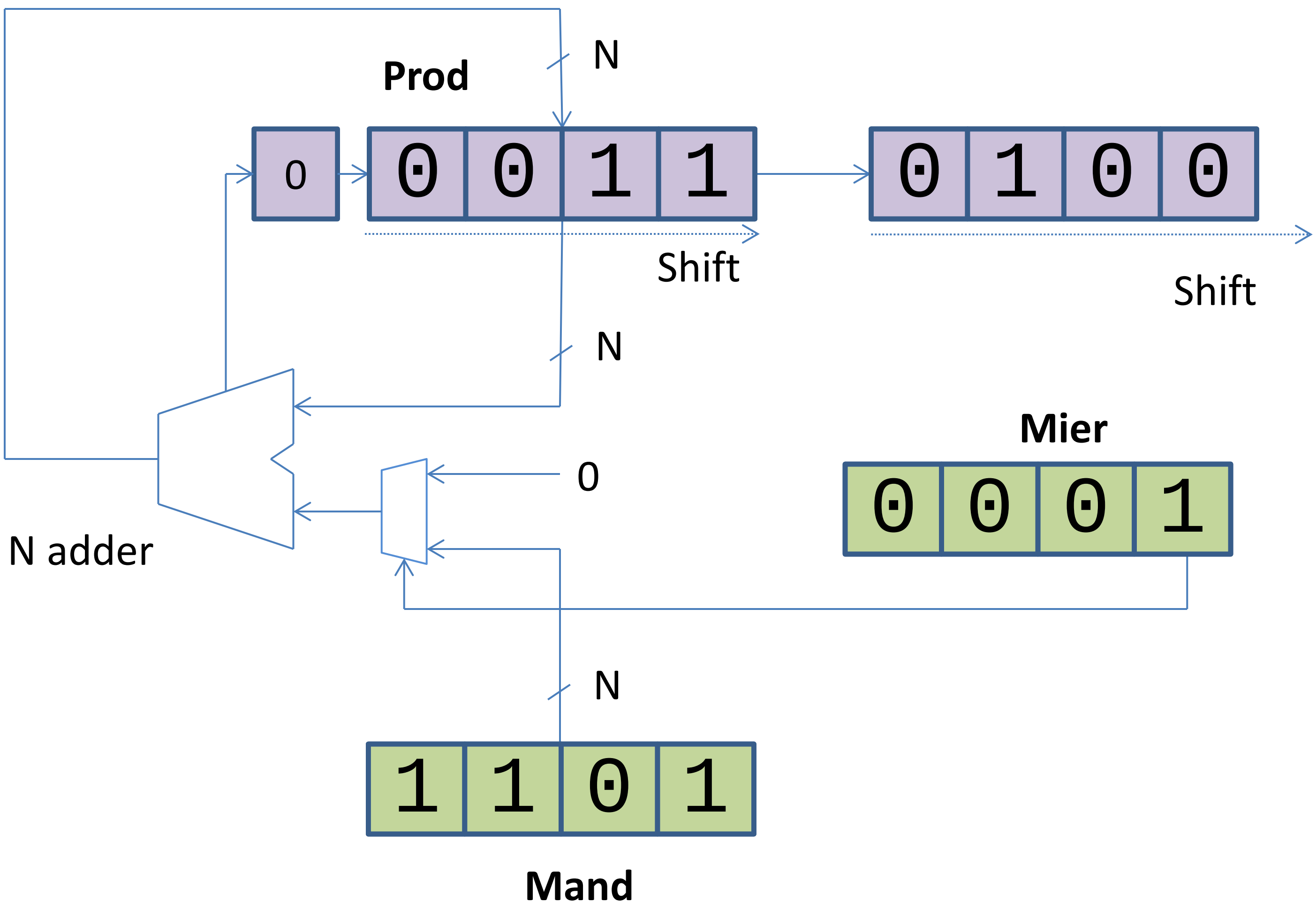
Step 1: Shift Prod and Mier



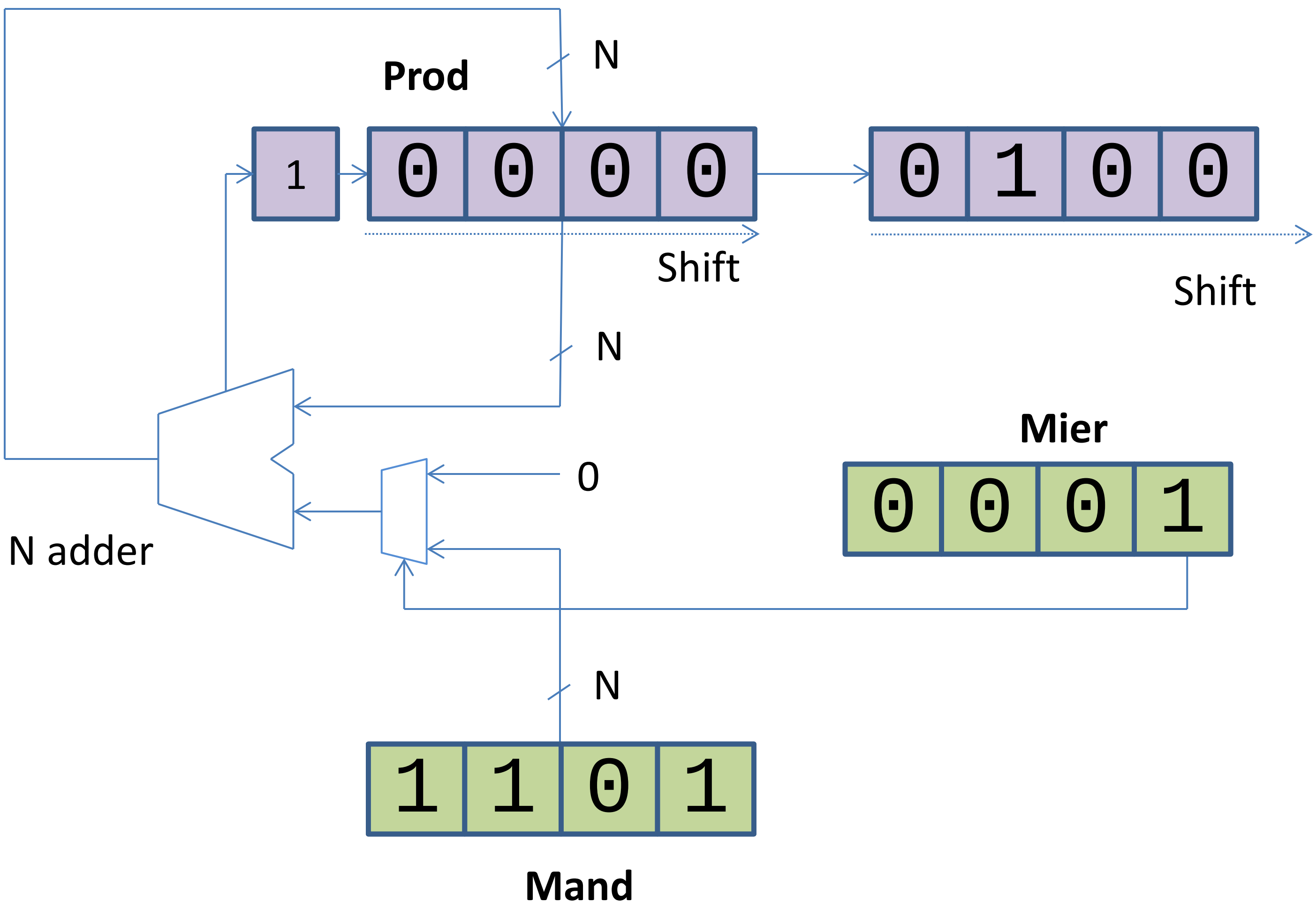
Step 2: Add Mand to Prod



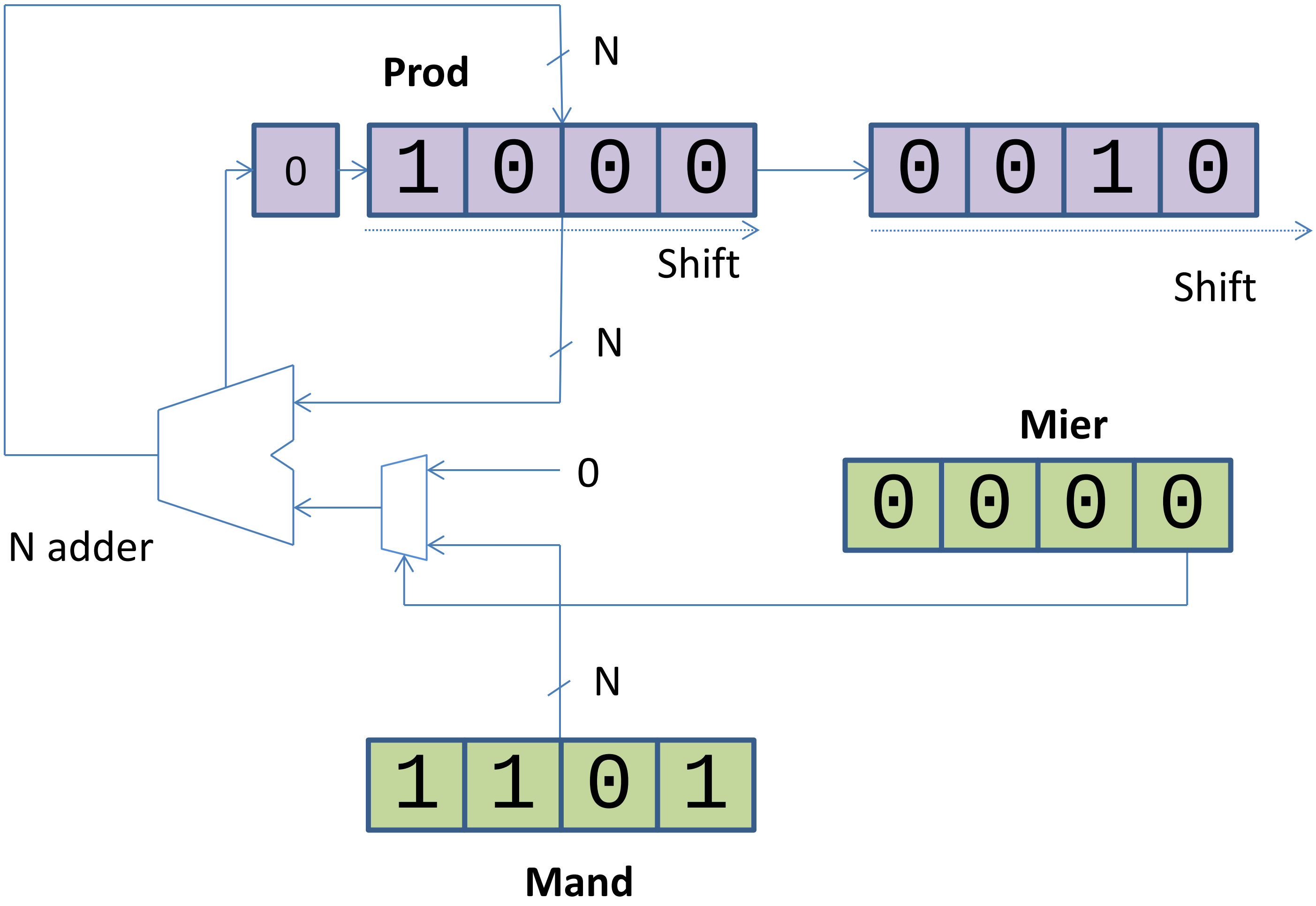
Step 2: Shift Prod and Mier



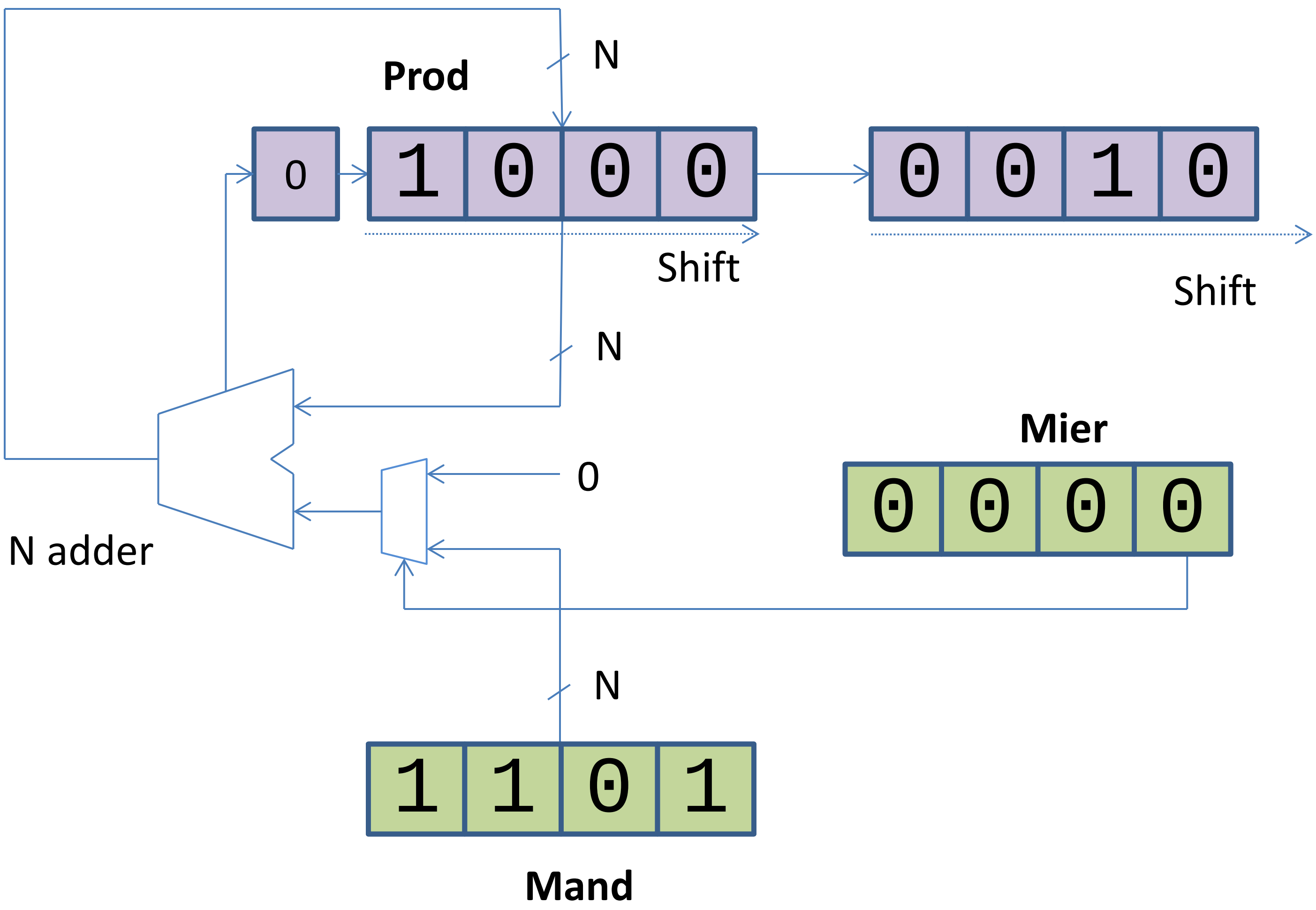
Step 3: Add Mand to Prod



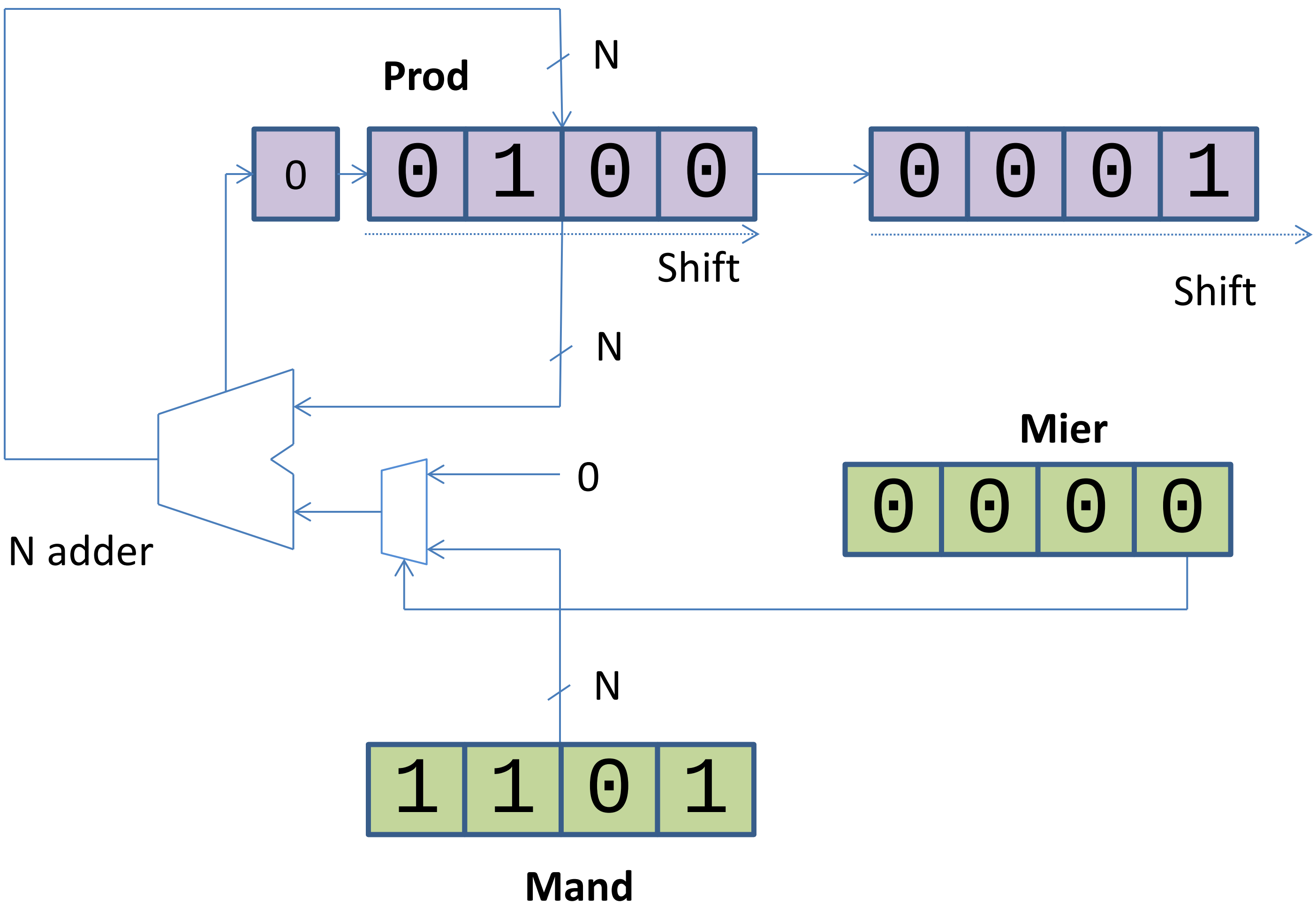
Step 3: Shift Prod and Mier



Step 4: Add Mand to Prod



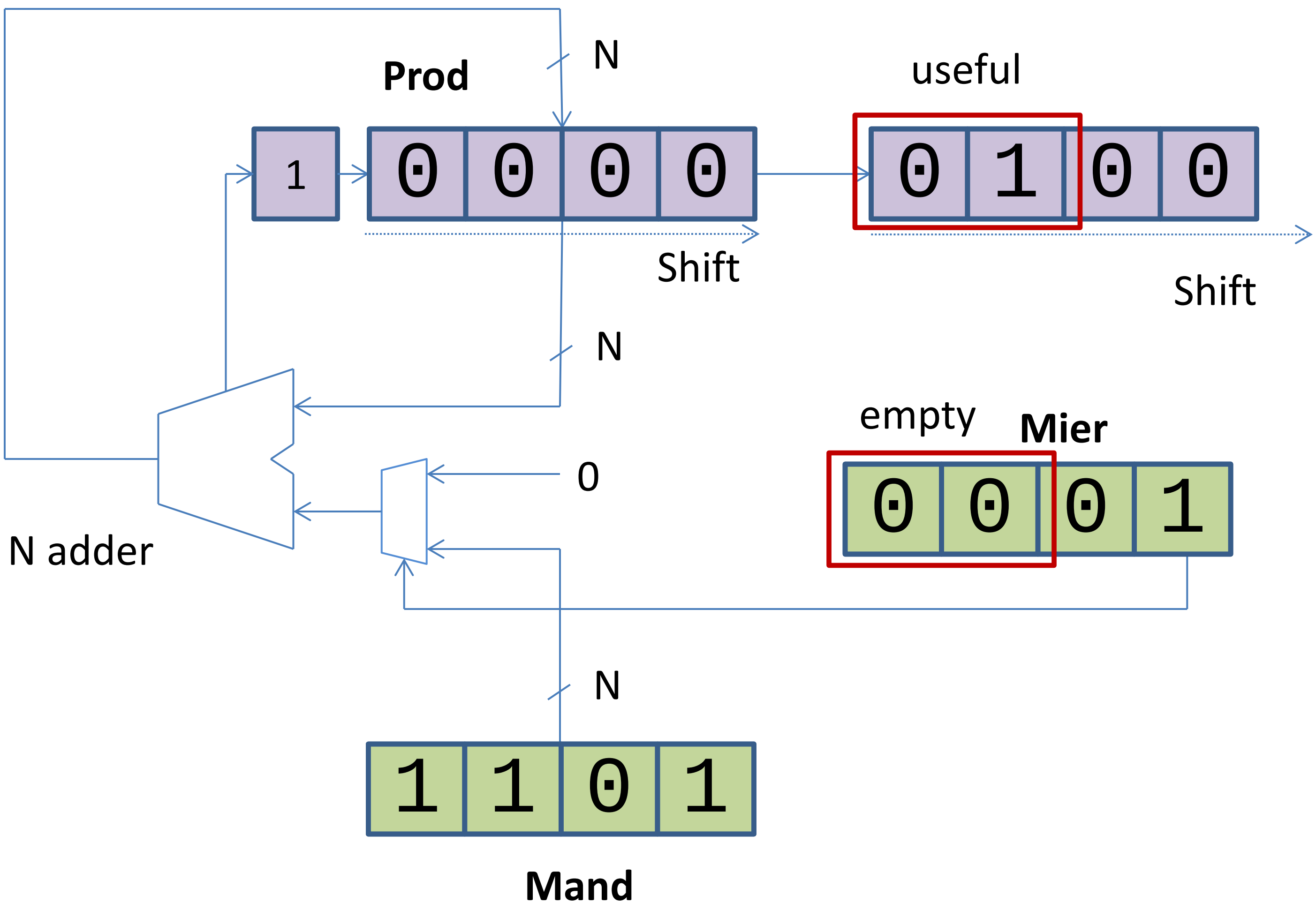
Step 4: Shift Prod and Mier



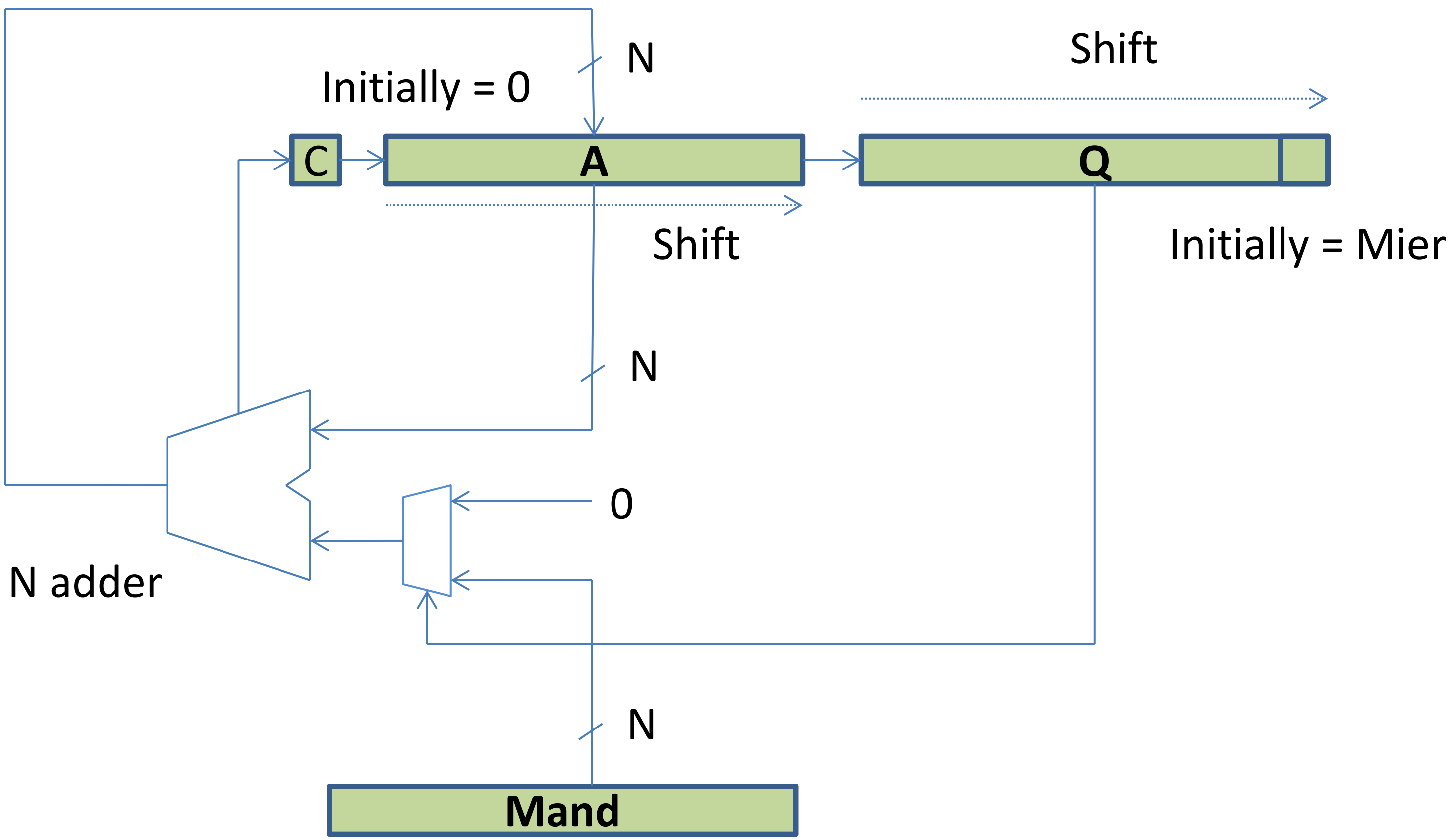
Multiplier Take #2: How it works

Iteration	Step	Carry	Prod	Mier
0	Initial Value	0	0000 0000	0101
1	Add + Mult	0	1101 0000	0101
	Shift	0	0110 1000	0010
2	Add+Mult	0	0110 1000	0010
	Shift	0	0011 0100	0001
3	Add+Mult	1	0000 0100	0001
	Shift	0	1000 0010	0000
4	Add+Mult	0	1000 0010	0000
	Shift	0	0100 0001	0000

Low Prod and Mier can be merged



Multiplier: Take #2



Multiplier Array

1101 multiplicand

1101 1 multiplier

~~11010~~ 0

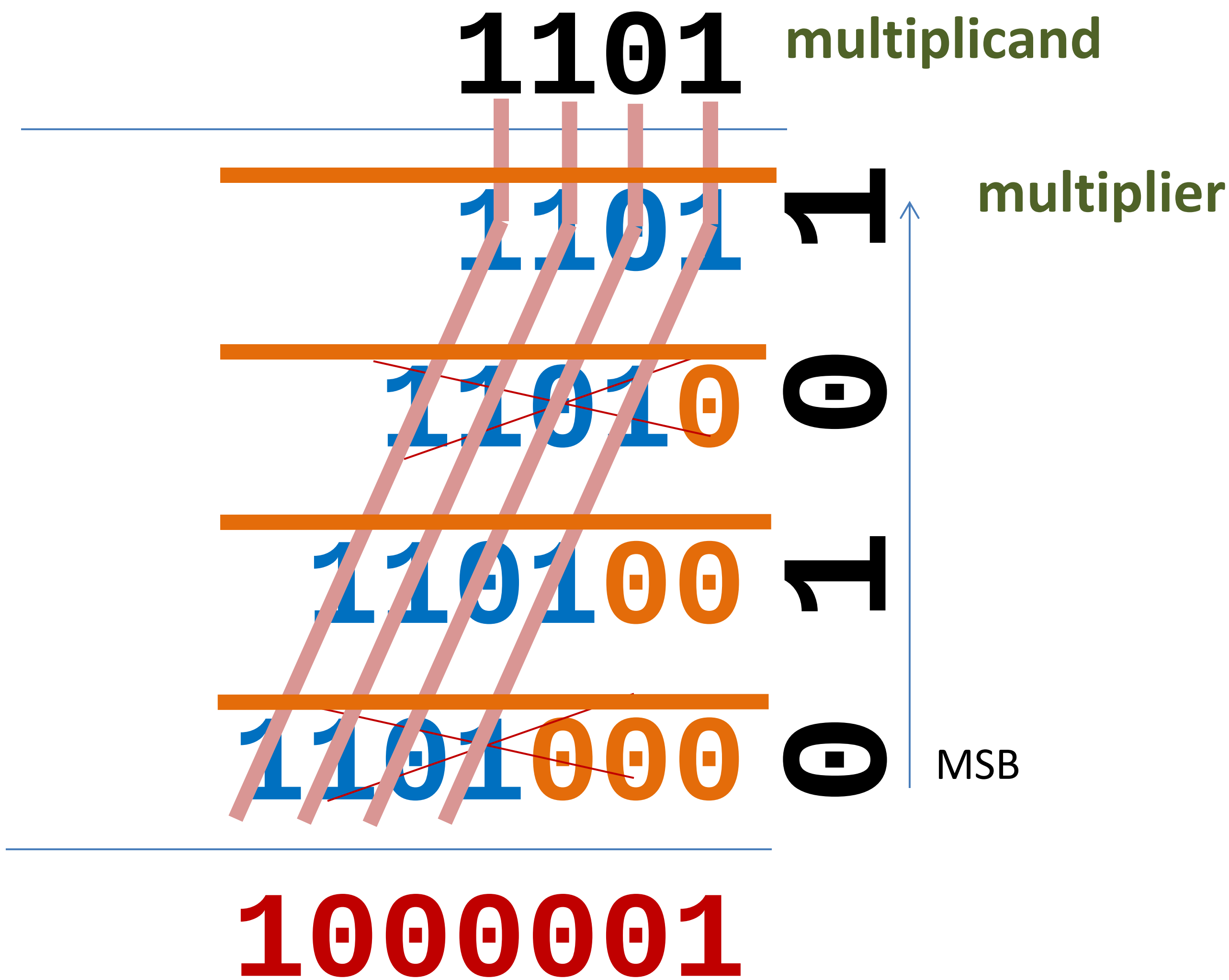
Include term

110100 1

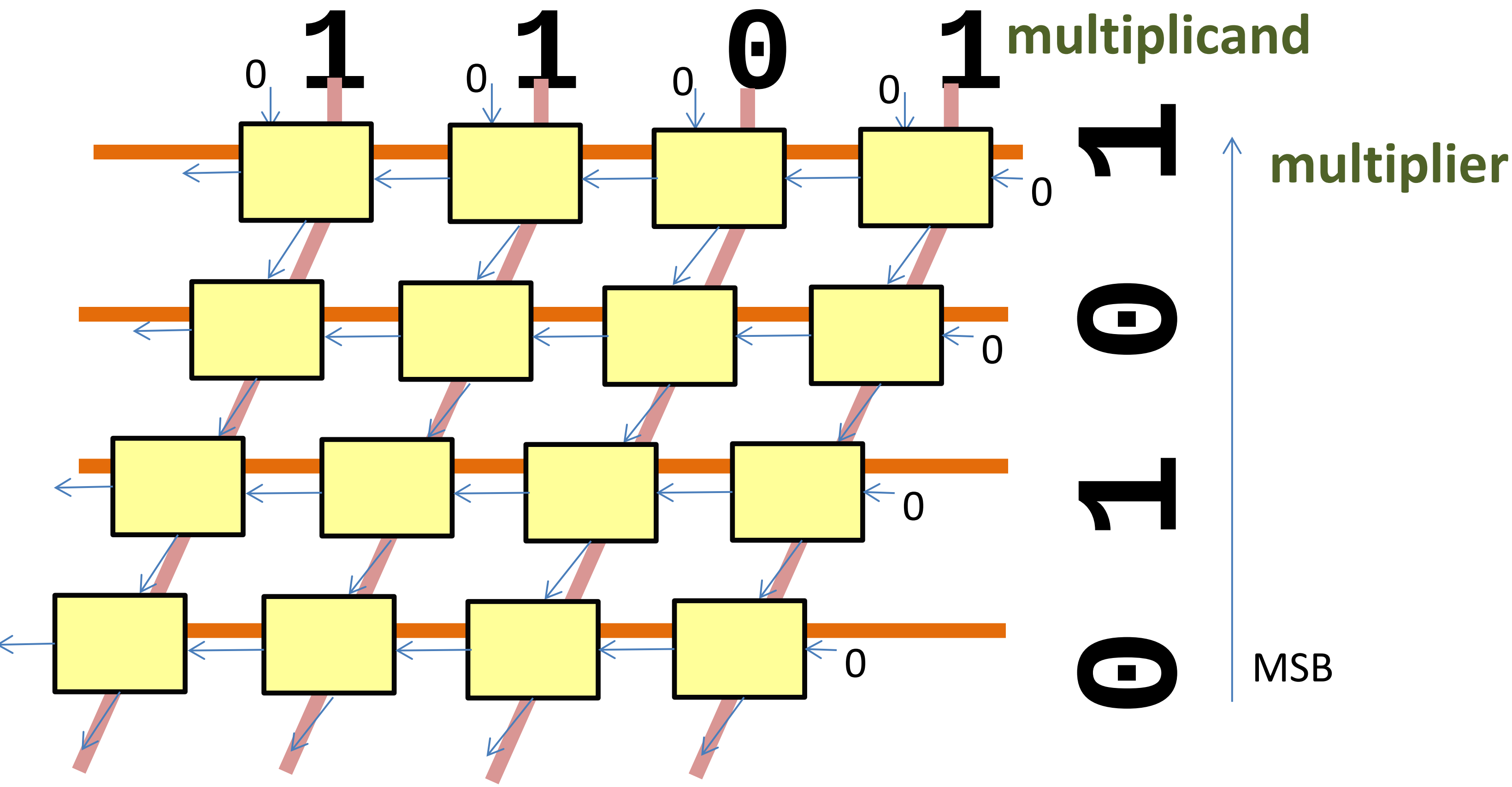
~~1101000~~ 0 MSB

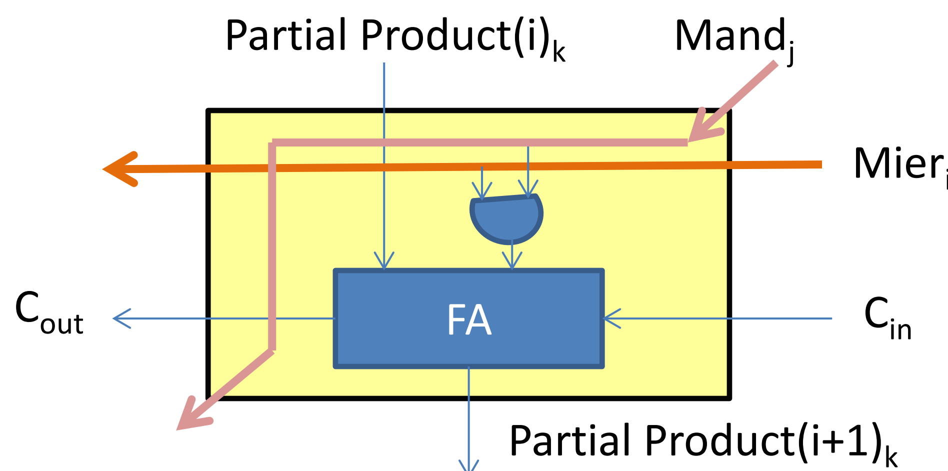
1000001

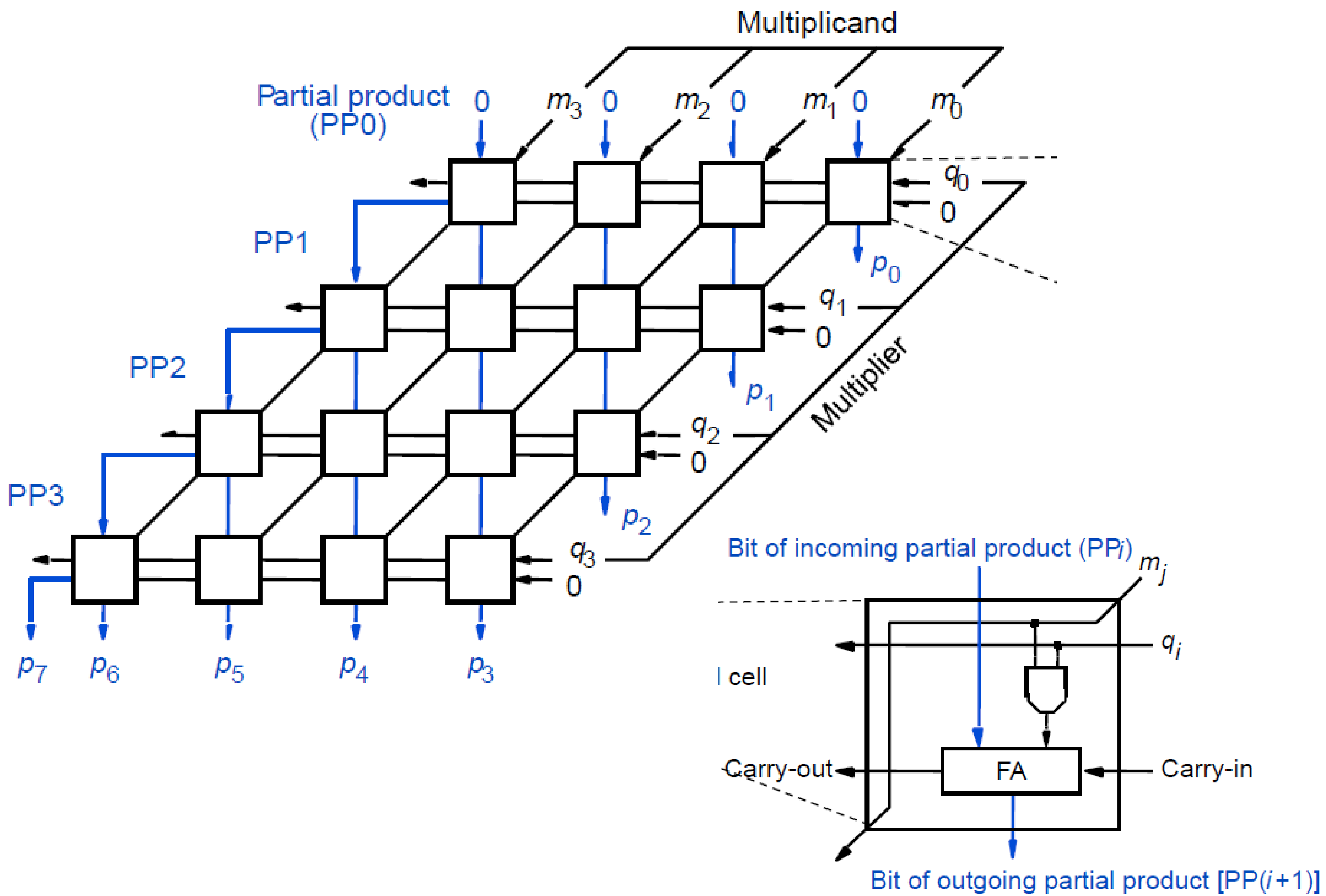
Multiplier Array



Multiplier Array

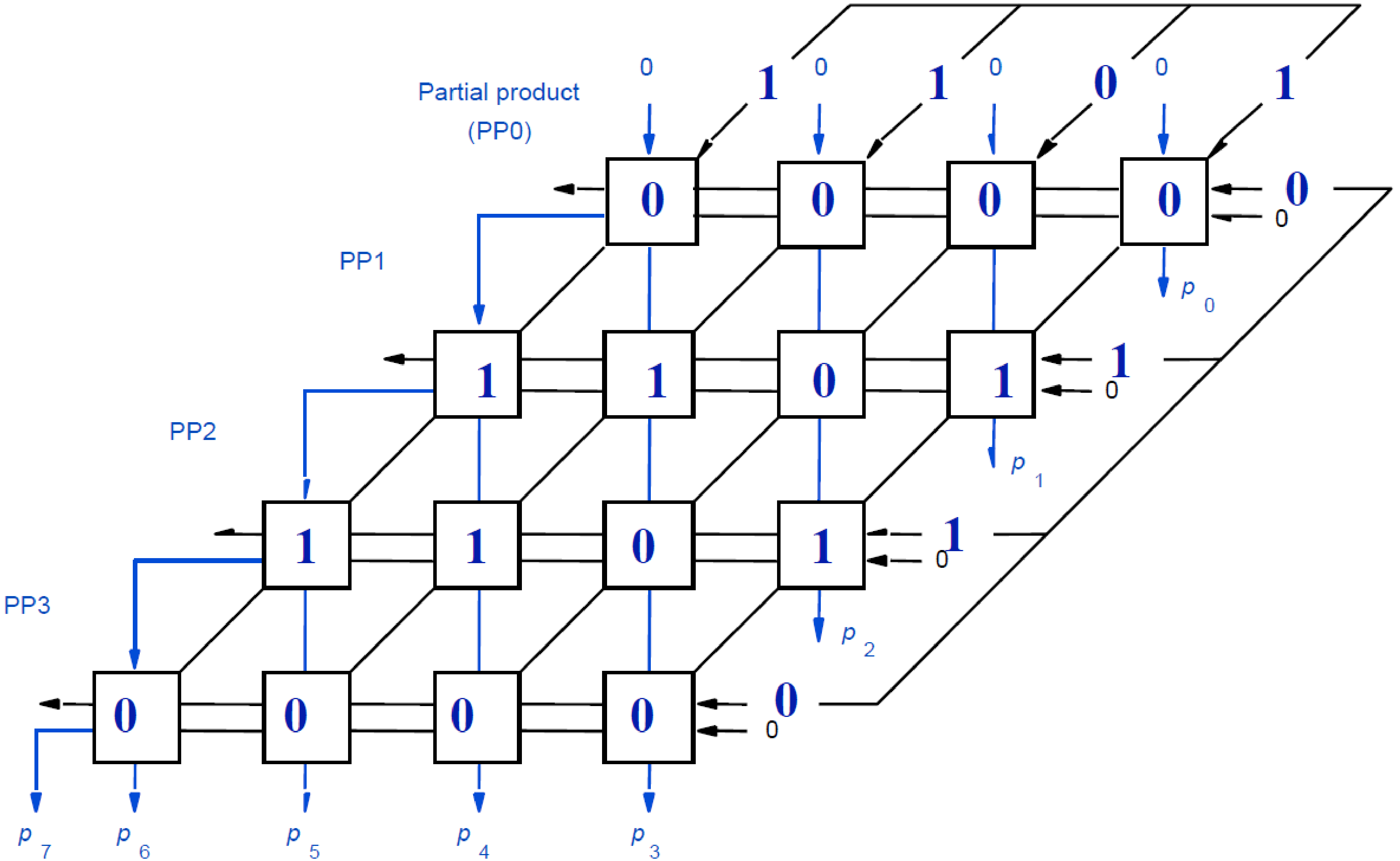




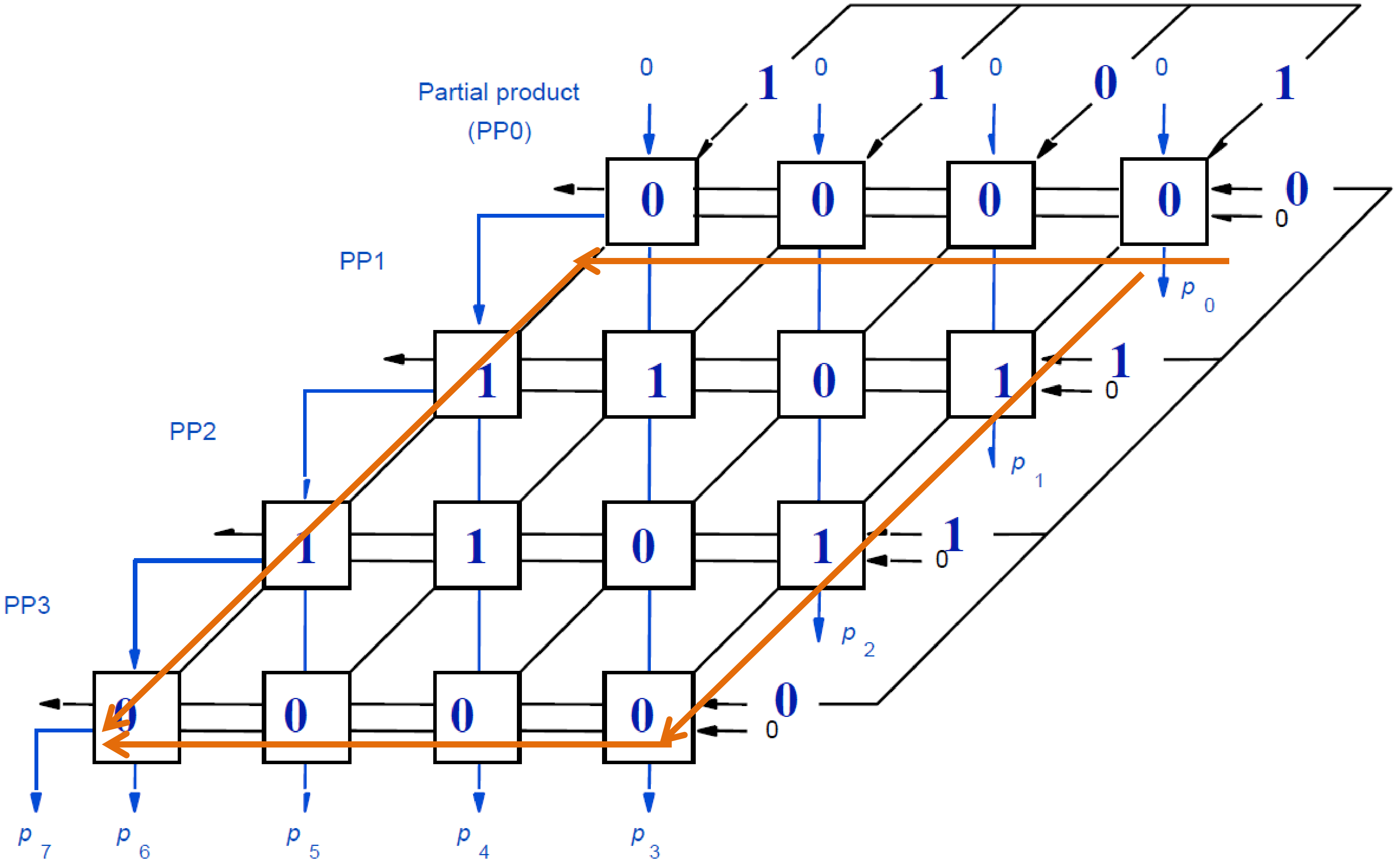


Multiplicand
Partial product (PP0)
0
m_3
0
m_2
0
m_1
0
m_0
q_0
0
p_0
PP1
q_1
0
p_1
PP2
q_2
0
p_2
Multiplier
PP3
q_3
0
p_7
p_6
p_5
p_4
p_3
Bit of incoming partial product (PPi)
m_j
q_i
cell
FA
Carry-out
Carry-in
Bit of outgoing partial product [PP(i+1)]

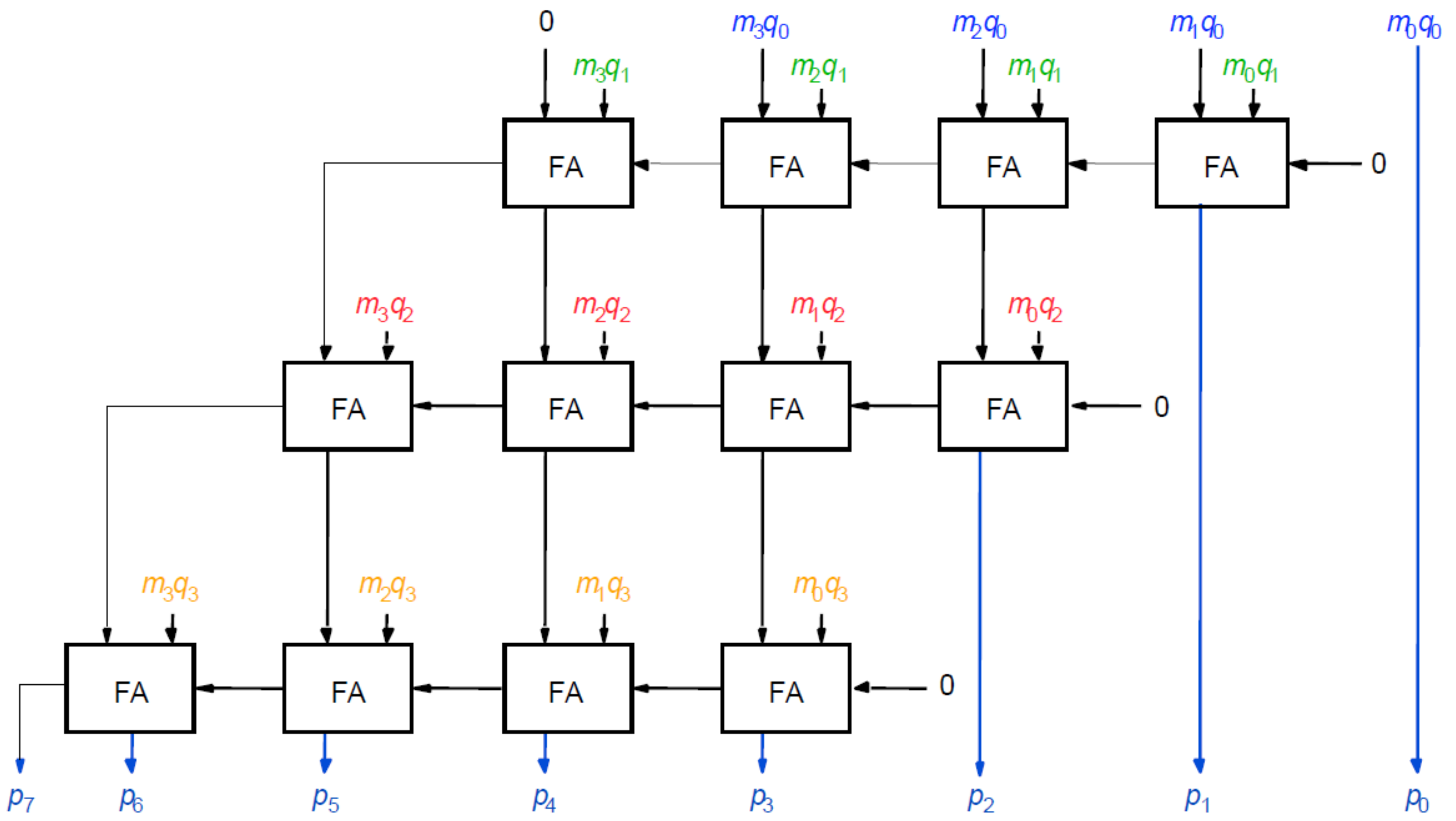
Multipler Array Example



Multiplier Array Critical Path?



Multiplier Array Critical Path



A + B + C

- A = 0 1 0 1 (5)
- B = 0 1 1 0 (6)
- C = 0 0 1 1 (3)
- A + B + C = 1110 (14)

Carry Save Adder: A + B

$$
\begin{array}{rrrrr}
 & 0 & 1 & 0 & 1 \\
+ & 0 & 1 & 1 & 0 \\
\hline
 & 0 & 0 & 1 & 1 \\
 & 0 & 1 & 0 & 0
\end{array}
\begin{array}{l}
\\ \\ \\ \text{SUM} \\ \text{CARRY}
\end{array}
$$

No need to propagate Carry

Redudant Representation

- Represent a number with two bits per digit
- (Sum, Carry)
  - (1,0)(1,0)(1,0)(0,0)
  - 1 1 1 0
- How about this:
  - (1,0)(0,0)(1,0)(0,1)?
  - 1100

Carry Save Adder: (A+B) + C

SUM 0 0 1 1

A + B

CARRY 0 1 0 0

0 0 1 1

C

---

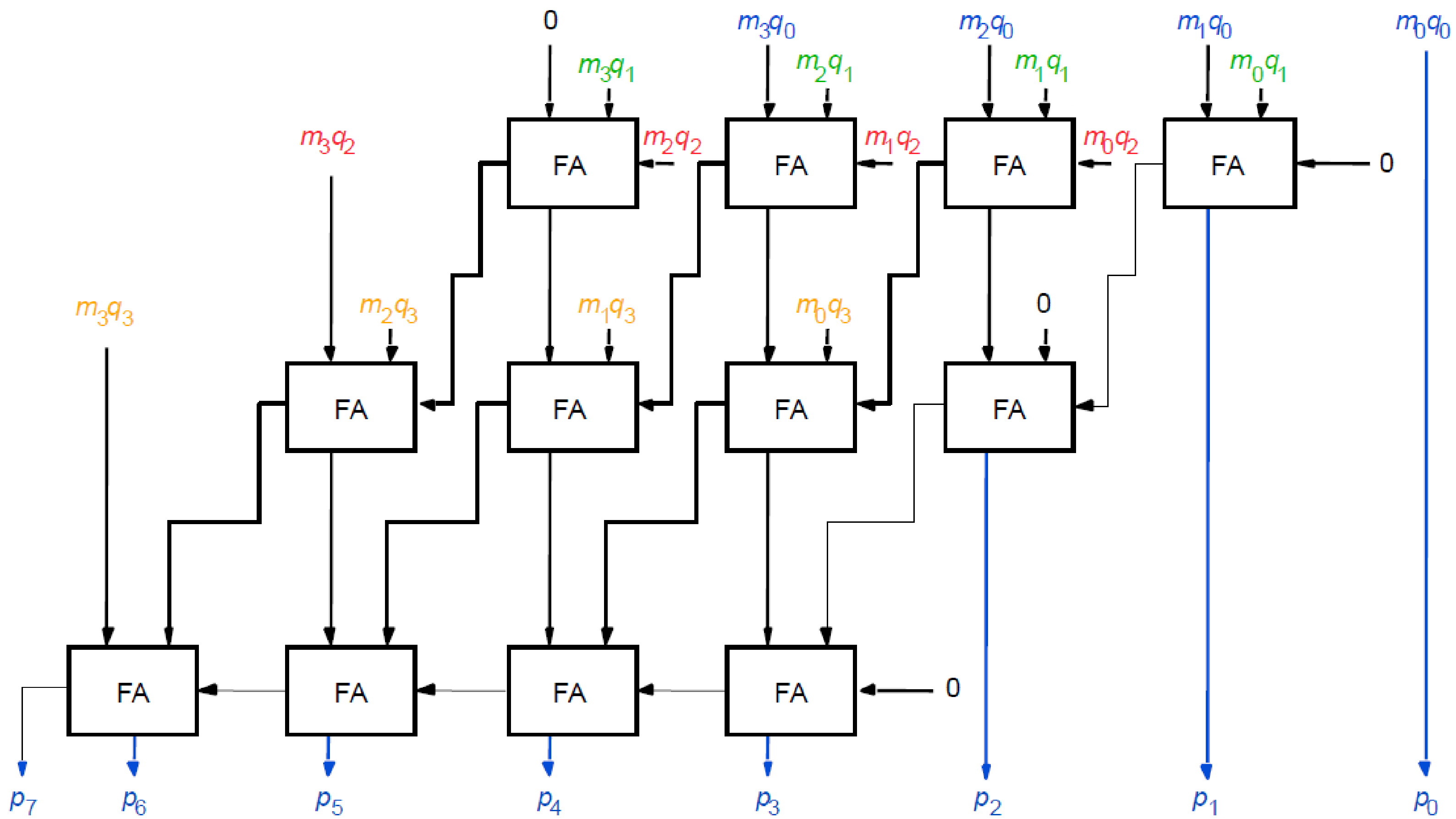
1 0 0 0

Which number is this?

0 0 1 1

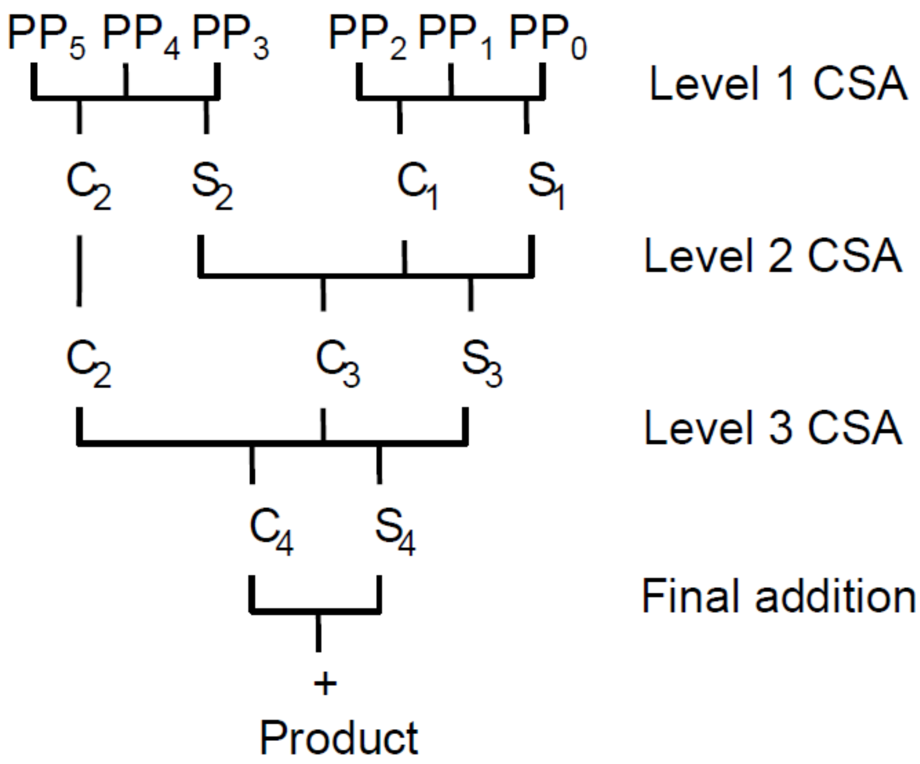
1110

Carry Save Adder Multiplier Array



Wallace Tree

- Addition done hierarchically
  - Groups of 3 partial products added in parallel
  - Produce (sum, carry)
  - (sum, carry) + another output



Signed Number Multiplication

```
      1101 (-3) Mand
      0101 (5)  Mier
  -----------------
   1111101
   000000
   11101
   0000
  -----------------
  11110001 (-15)
```

Signed Number Multiplication

```
     111001 (-7) Mand
     001110 (14)  Mier
-----------------------
000000000000
11111111001
1111111101
111111101
-----------------------
111110011110 (-98)
```

And now for something different

- Recall:
  - **$A \times (B + C) = A \times B + A \times C$**
- 14 = 16 – 2
- 001110 = 010000 - 000010

001110

010000

000010

Signed Number Multiplication

111001 (-7) Mand

0+1001-10 (14) Mier

000000000000

000000111

Inverse of -7

000000000

00000000

Skip these and do a shift by 2

1111001

Booth's Multiplication

11110011110 (-98)

Booth Encoding

001110000111111000000

0+100-1000+100000-1000000

000111111111000011110

00+100000000-1000+1000-10

010101010101010101010

+1-1+1-1+1-1+1-1+1-1+1-1+1-1+1-1+1-1+1-1

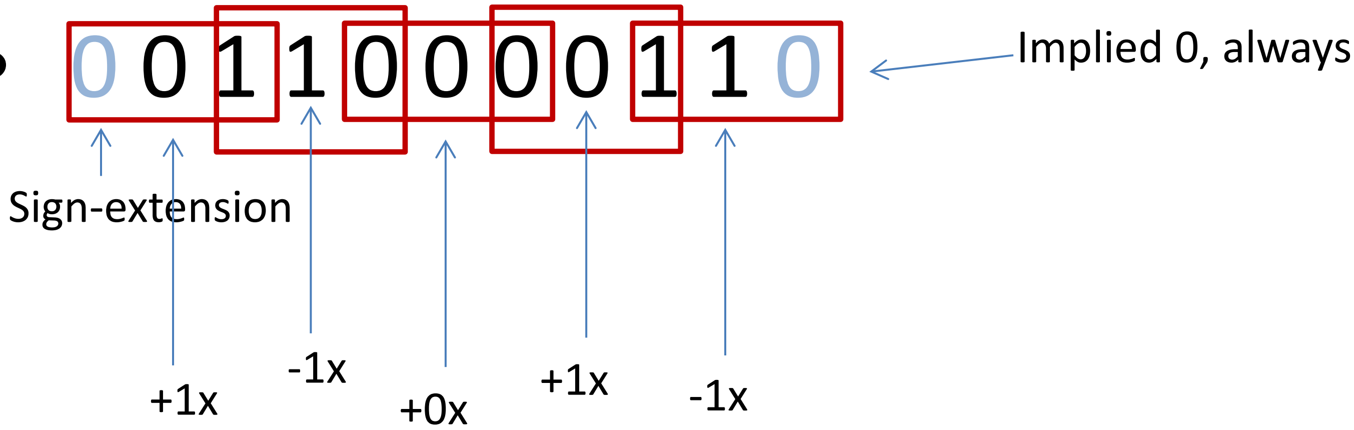
Booth Encoding Generation

A simple table can be used to generate the booth encoding. As bits in the multiplier are shifted right do:

- Each time there is a transition from a 0 to a 1, the multiplicand is added
- Each time there is a transition from 1 to 0 the inverse of the multiplicand is added
- Otherwise do nothing

Multiplier		Action at position *i*
Bit *i*	Bit *i-1*	
0	0	0xM
0	1	1xM
1	0	-1xM
1	1	0xM

2-bit Booth Recoding

- 0 1 1 0 0 0 0 1 1
- 

2-bit Booth Recoding

- 1 0 1 1 1 (-9)
- 1 1 0 1 1 1 0

Implied 0, always

Sign-extension

-1x

+2x

-1x

Bit-Pair Booth Recoding Example

Example: Multiply `11001 (-7) * 10111 (-9)`

```
1 1 0 1 1 1 0
 -1  +2  -1
```

Doing the multiplication:

```
          1 1 0 0 1   (-7)
         -1  +2  -1   (-9)
---------------------
0 0 0 0 0 0 0 1 1 1
1 1 1 1 0 0 1 0
0 0 0 1 1 1
---------------------
0 0 0 0 1 1 1 1 1 1   (63)
```

Bit-pair recoding results in fewer additions

– Multiplier arrays will benefit from this as well

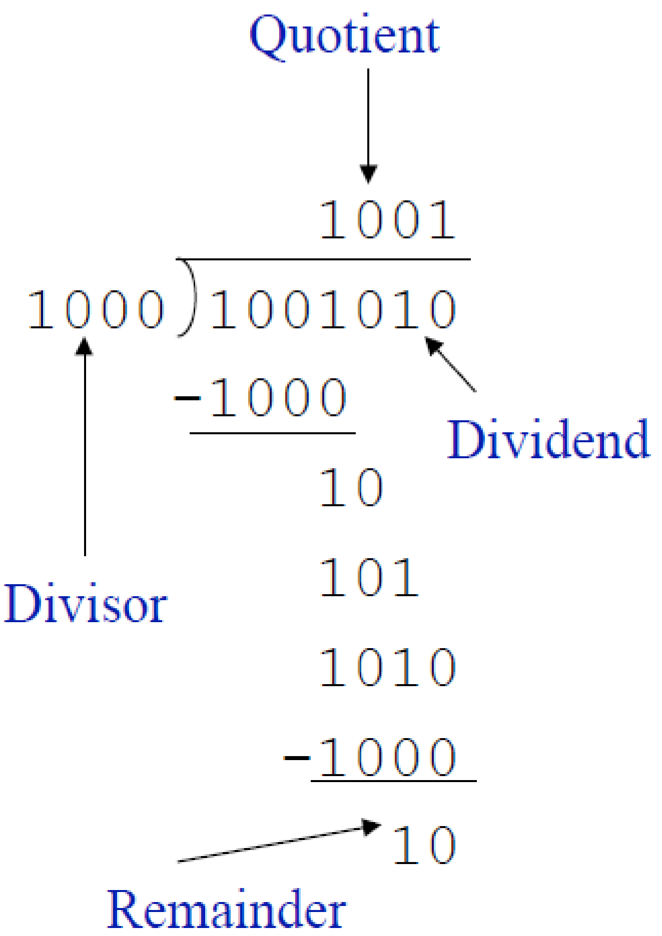
Generating the bit-pair Recording

Multiplier		Next bit in Multiplier *(i-1)*	Action at position *i*
i+1	*i*		
0	0	0	0xM
0	0	1	1xM
0	1	0	1xM
0	1	1	2xM
1	0	0	-2xM
1	0	1	-1xM
1	1	0	-1xM
1	1	1	0xM

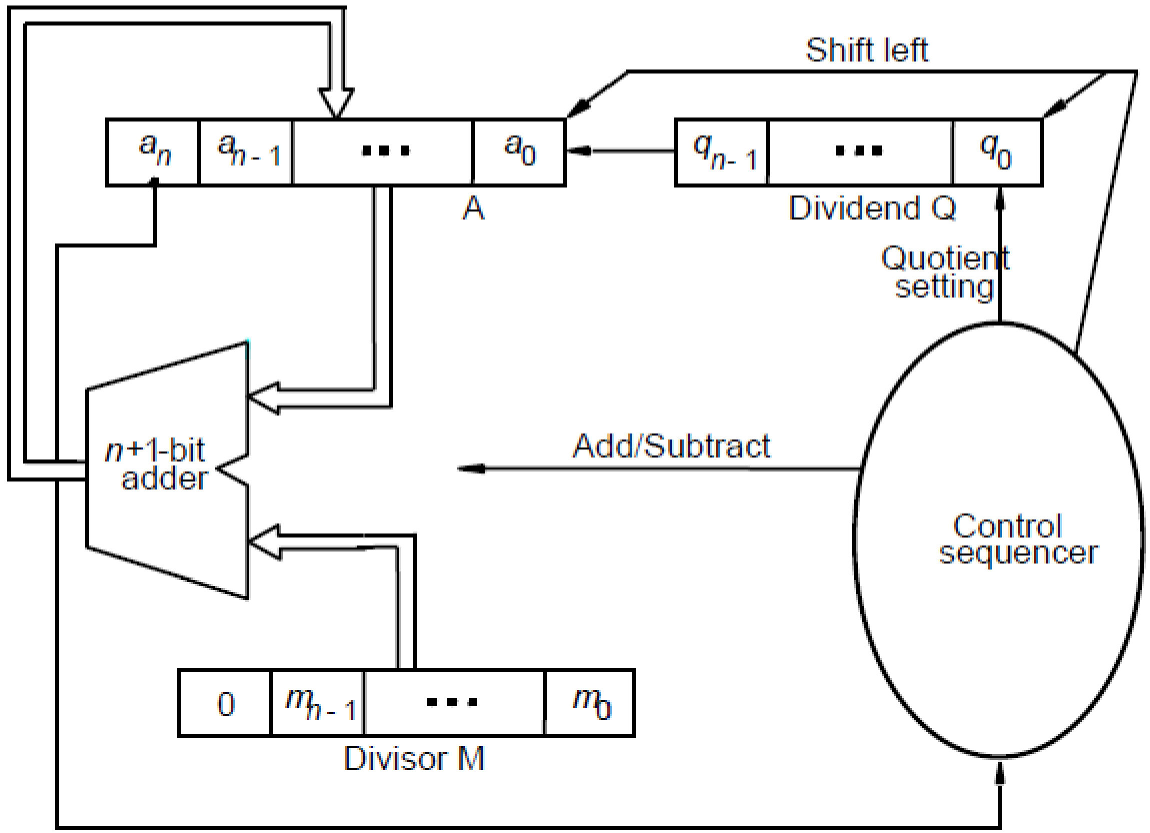
Division

Division in binary follows the same process as long division by hand:

1. Find the smallest number of digits from the dividend that will be greater than the divisor. For every digit that needs to be taken from the dividend, put a 0 in the quotient.
2. When the portion of the dividend is large enough, add a 1 in to the quotient and subtract the divisor
3. Repeat until no digits in the dividend are left.



Simple Divisor



Register A holds the *current remainder* for the division

– The current divisor is compared with the remainder by subtracting it from the remainder and checking the sign of the result.

Divisor Operation

- Steps for restoring division:
- 1. Subtract divisor from current remainder
- 2. Test the new remainder:
  - 1. If positive put a 1 in the quotient
  - 2. If negative put a 0 in the quotient and add the divisor back to the remainder to “restore” it back to its original value.
- 3. Shift the quotient and dividend left by 1 bit
- 4. Repeat until all bits in the dividend are consumed.
- **Restoring Division:**
  - Divisor is added back when larger than remainder

Restoring Division Example

Example: $1000\overline{)\boxed{1001010}}$

Operation	Sign	Register A (Remainder)	Quotient
Shift	0	0100	1010
Subtract	1	1100	1010**0**
Restore	0	0100	10100
Shift	0	1001	0100
Subtract	0	0001	0100**1**
Shift	0	0010	1001
Subtract	1	1010	1001**0**
Restore	0	0010	10010
Shift	0	0101	0010
Subtract	1	1101	0010**0**
Restore	0	0101	00100
Shift	0	1010	0100
Subtract	0	0010	0100**1**

Non-Restoring Division

- If R is negative we add D to restore value
- Can we avoid this?
- At each step we want to do: R – D
- When negative we do: R – D + D
- Then shift left: 2 x R
- Then subtract D: 2xR – D
- What if we don't restore: **R – D**
- Shift: 2x(R – D) = **2xR – 2xD**
- And we want **2xR – D**
- What to do? **(2xR – 2xD) + D**

Non-Restoring Division Example

Example: $1000 \overline{)\,1001010}$

Operation	Sign	Register A (Remainder)	Quotient
Shift	0	0100	1010
Subtract	1	1100	1010**0**
Shift	1	1001	0100
Add/Restore	0	0001	0100**1**
Shift	0	0010	1001
Subtract	1	1010	1001**0**
Shift	1	0101	0010
Add/Restore	1	1101	0010**0**
Shift	1	1010	0100
Add/Restore	0	0010	0100**1**

SRT Division Pointer

- Sweeney, Robertson, and Tocher Division
- SRT is essentially the same as non-restoring division, but is performed in a higher radix:
  - SRT guesses several digits of the quotient at a time
  - SRT has pre-computed tables that allow it to find the right bits depending on remainder and divisor
  - Like non-restoring, if it overshoots, it makes up for it in following cycles

Prof. Natalie Enright Jerger

On-Chip Networks

Natalie Enright Jerger
University of Toronto

Li-Shiuan Peh
Princeton University

SYNTHESIS LECTURES ON COMPUTER ARCHITECTURE #8

MORGAN & CLAYPOOL PUBLISHERS