(1)

```
127   63   31
128   64   32   16   8   4   2   1
~~[illegible]~~ 1   1   1   1   1   1   1
```

or 1111
1111

(1')

Multiplication

↳ Repeated Addition →

4-~~bit~~ *4	→ 4 steps
8 *8	→ 8 steps
256 *256	→ 256 steps

+5
+3 > multiply

```
multiplicand ←       0101
multiplier   ←       0011          ] Traditional
             ----------------        method
                     0101
                    0101x
                   0000xx
                  0000xxx
             ----------------
                  0001111
               0 0                  = +15
             ----------------
                    + 8 4 2 1
```

[partial product]

How many ?
→ The number of partial product is equal to no. of bits in multiplier.

→ The answer will not exceed (m+n) bits. [4-bit * 4bit] = [8 bit]

→ The partial product is either the multiplicand or the zero.

→ The multiplier will decide the ~~pro~~ partial product.

Array Multiplier

→ We can directly map the multiplication process (traditional one) into a hardware design.

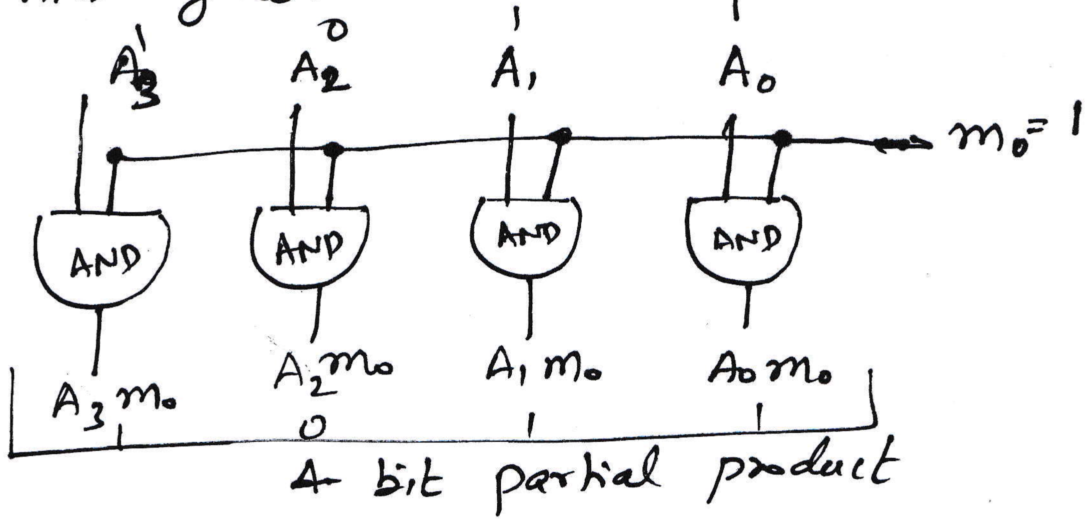
→ The array of cells is used to generate partial products

→ Only change is that instead of adding the partial product at end ~~every stage~~, it will be added at every stage.

→ Genereting partial products requires just an AND gate.

→

1011
11(01)

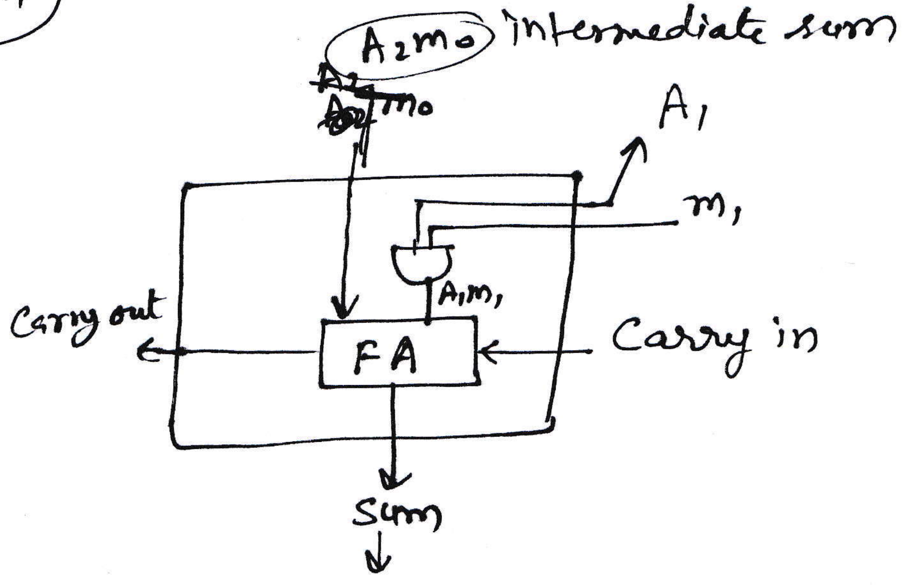


→ Realization or implementation of ~~n bit~~ adding n bit partial product is not an easy task.

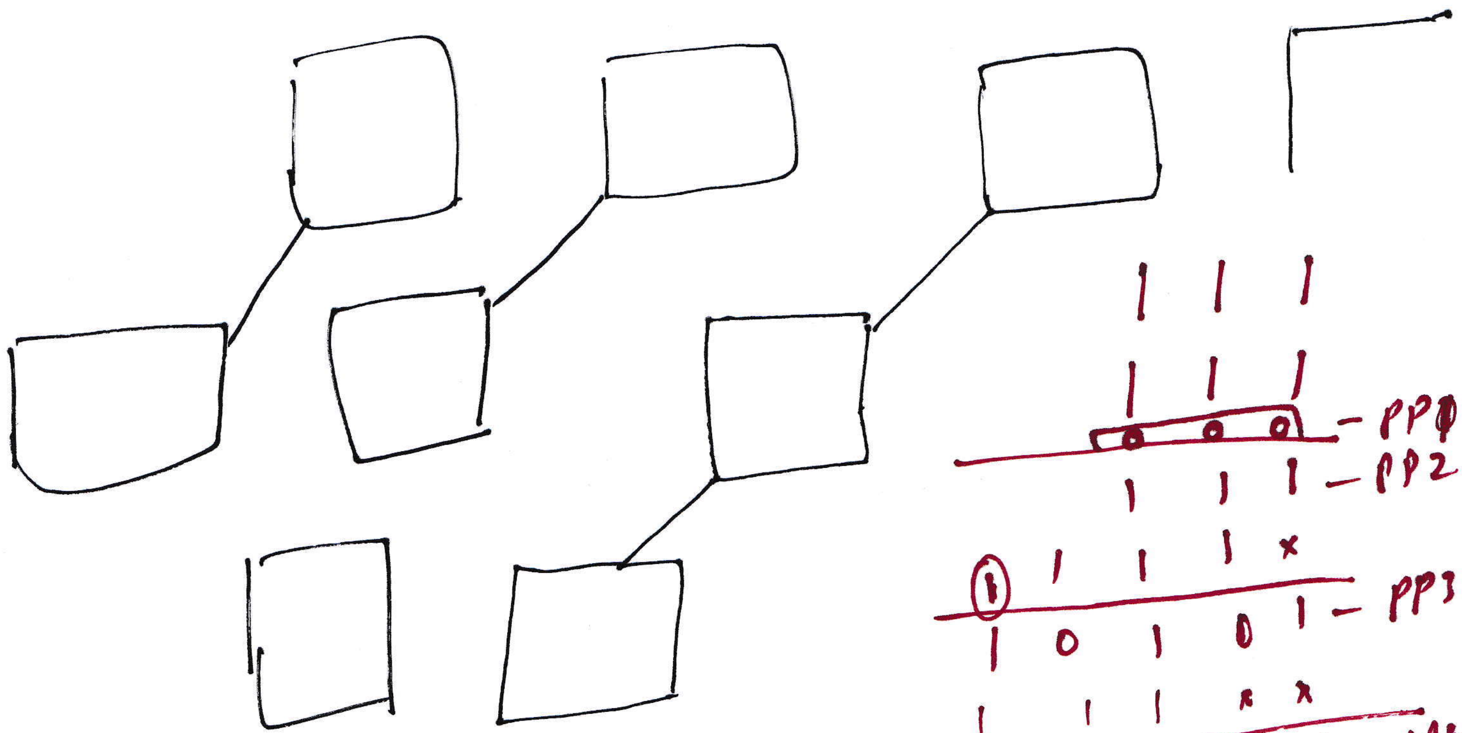
→ There will be n^2 partial products.

→ We will be needing ~~some~~ full adders at every step to add the current partial product & previous partial product.

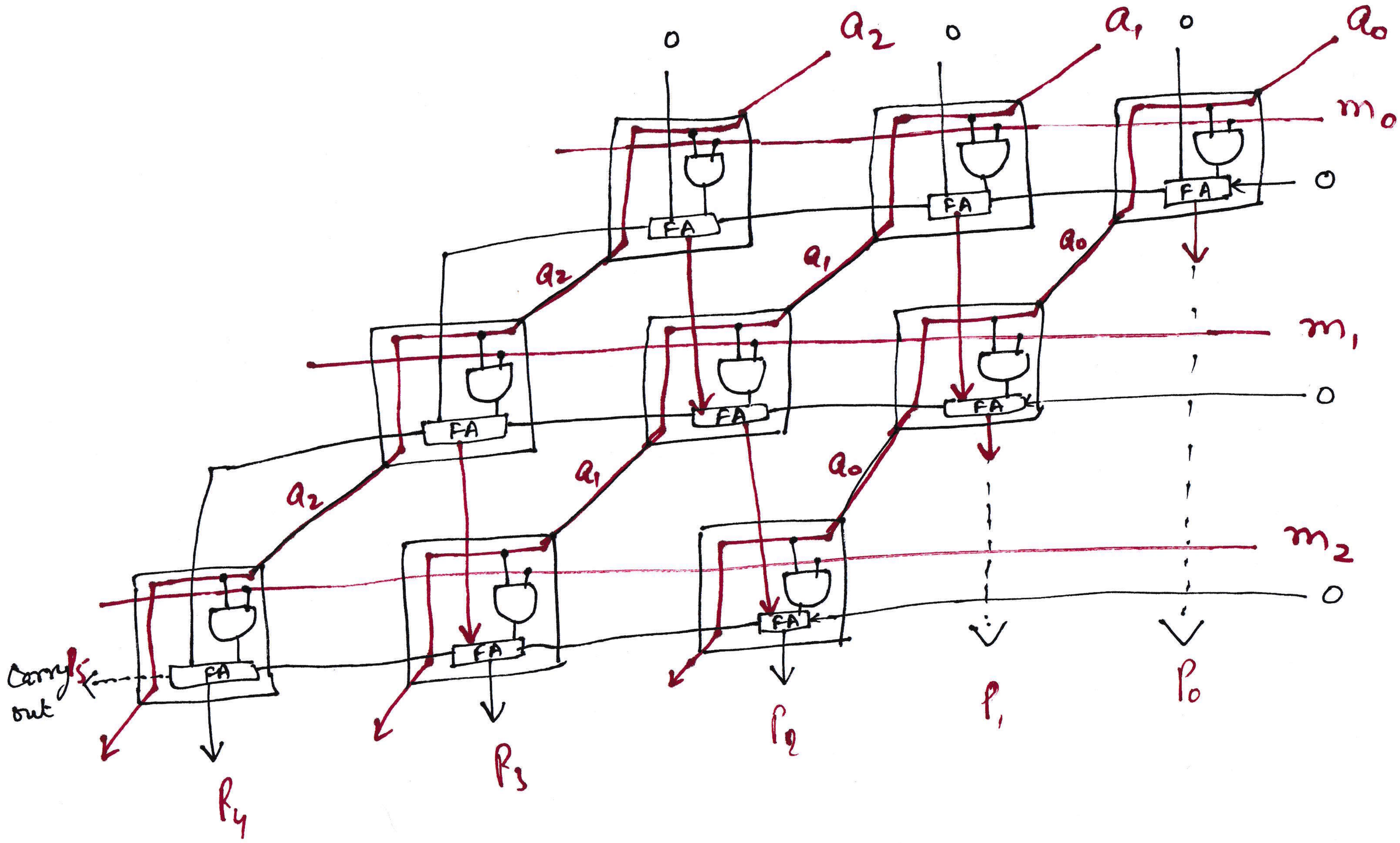
A_1m_1



$$\begin{array}{ccccccc}
 & & & A_3 & A_2 & A_1 & A_0 \\
 & & & m_3 & m_2 & m_1 & m_0 \\
\hline
 & & & A_3m_0 & A_2m_0 & A_1m_0 & A_0m_0 \\
 & & A_3m_1 & A_2m_1 & A_1m_1 & A_0m_1 & x \\
 & A_3m_2 & A_2m_2 & A_1m_2 & A_0m_2 & x & x \\
A_3m_3 & A_2m_3 & A_1m_3 & A_0m_3 & x & x & x \\
\hline
\end{array}$$



$$\begin{array}{cccccccl}
 & & & & 1 & 1 & 1 & \\
 & & & & 1 & 1 & 1 & \\
 & & & & 0 & 0 & 0 & - PP\phi \\
\hline
 & & & & 1 & 1 & 1 & - PP2 \\
 & & \textcircled{1} & 1 & 1 & 1 & x & \\
\hline
 & & 1 & 0 & 1 & 0 & 1 & - PP3 \\
 & & 1 & 1 & 1 & x & x & \\
\hline
\textcircled{1} & 1 & 0 & 0 & 0 & 1 & & PP4
\end{array}$$


0
a_2
0
a_1
0
a_0
m_0
0
FA
FA
FA
a_2
a_1
a_0
m_1
0
FA
FA
FA
a_2
a_1
a_0
m_2
0
FA
FA
FA
Carry out
P_4
P_3
P_2
P_1
P_0

Performance Analysis of Array Multiplier

→ It is extremely inefficient and requires very large amt. of hardware.

→ n^2 multiplication cells for $n \times n$ multiplier circuit.

→ ~~A~~ It is very fast.

Sequential Multiplier

→ At Every step, check the multiplier bit from LSB.

if bit is 1, the
  pp is the Multiplicand.

else
  pp is zero.

→ Add all the partial products.

→ How many registers are required?
multiplicand, multiplier, partial products, result (product)

→ We ~~require~~ will use one big registers to store final output, which will be used to store all intermediate partial products also.
The register is the accumalater.

→ This will save the wastage due to of taking multiple registers.

→ 4-bit register	-	multiplicand
4-bit "	-	multiplier
8-bit "	-	Accumalator

→ Initially the Accu. will have zero.

$\boxed{0000\ 0000}$

→ At every step, we will add the PP into the accu. and perform Right-shift (RS).

```
(+5)   0101
(+3)   0011
-----------------
        0101
(1)    0101X
      0000XX
     0000XXX
-----------------
     0001111  ✓
```

```
        | 0000 0000 |
      + 0101
add     | 0101 0000 |
RS      | 0010 1000 |
      + 0101
add     | 0111 1000 |
RS      | 0011 1100 |
      + 0000
RS      | 0001 1110 |
      + 0000
RS      | 0000 1111 |   (+15)
```

$$\begin{array}{r} +13 \\ +11 \\ \hline \boxed{143} \end{array}$$

Sequential multiplier

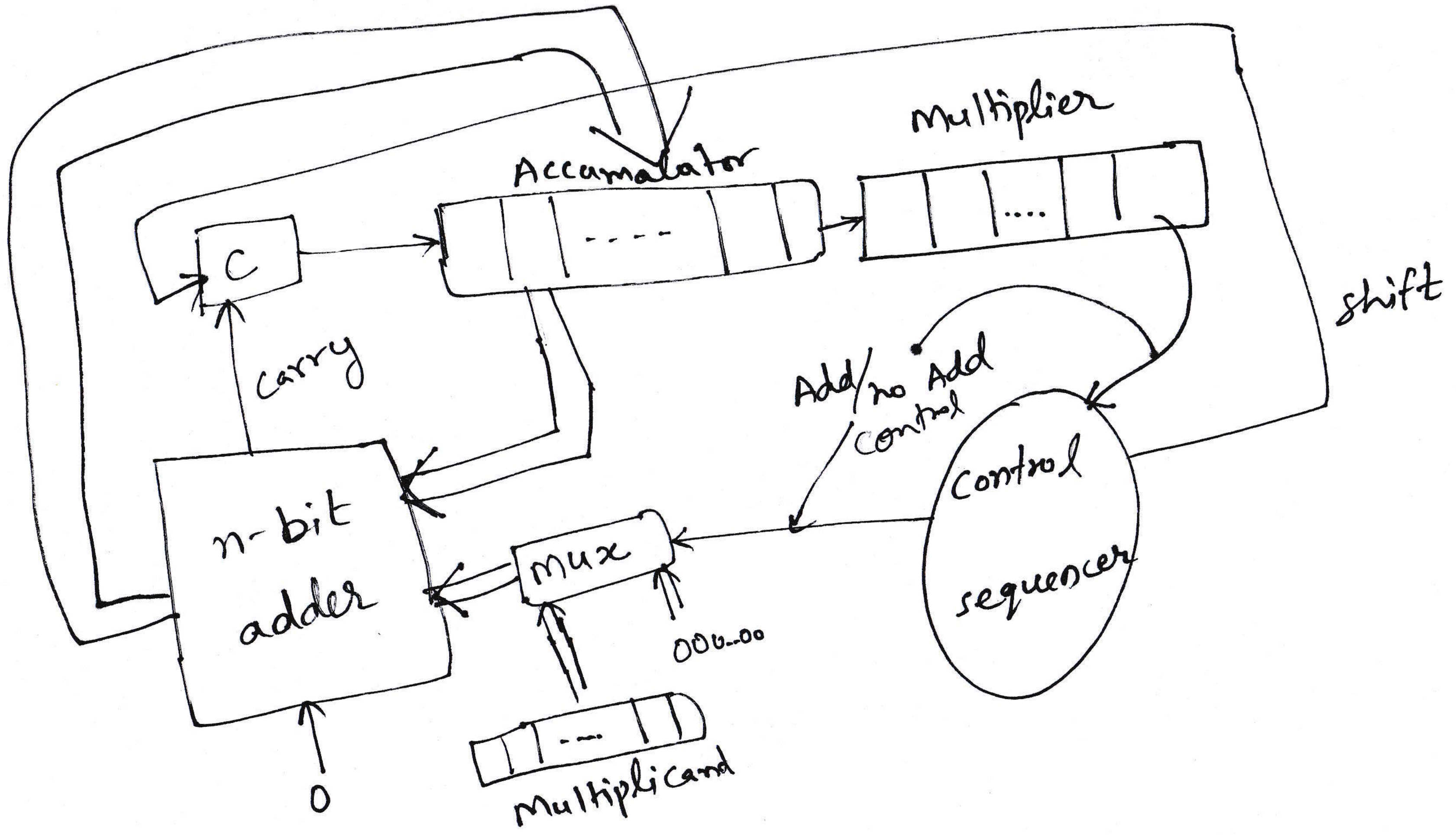
```
        [1 1 0 1]  ← Multiplicant

[0]     [0 0 0 0]        [1 0 1 1]  ← Multiplier
                          ④ ③ ② ①
 C       Accum.

                              ①
[0]      1 1 0 1         1 0 1 1    [Add]
 0 ---   0 1 1 0         1 1 0 1    [shift-Right]
[0]      1 1 0 1         1 1 0 1    [Add]
                              ②
        ---------------------------
         0 0 1 1         1 1 0 1    [Right shift]
[1] ---  1 0 0 1         1 1 1 0
         0 0 0 0         1 1 1 0    [No add]
                              ③
        ---------------------------
       → 1 0 0 1         1 1 1 0    [Right-shift]
[0] ---  0 1 0 0         1 1 1 1
         1 1 0 1                    [Add]
                              ④
        ---------------------------
[1]      0 0 0 1         1 1 1 1    [Right shift]
 1 ---   1 0 0 0         1 1 1 1
[0]
       128 ~~64~~ ~~32~~ ~~16~~   8 4 2 1
```

$\boxed{= 143}$

→ We performed three add operations.


Accamalator
Multiplier
C
carry
shift
Add/no Add control
n-bit adder
mux
Control sequencer
000..00
0
Multiplicand