# Multiply

Does X 2 == Left Shift?

$$\begin{array}{r} 1 \\ \times\ 10 \\ \hline 10 \end{array} \qquad \begin{array}{r} 10 \\ \times\ 10 \\ \hline 100 \end{array} \qquad \begin{array}{r} 100 \\ \times\ 10 \\ \hline 1000 \end{array}$$

Works for powers of 2.

How about in the general case?

$$2x = x + x \longrightarrow \begin{array}{rcccccccc} & c_n & & c_{j+1} & c_j & & & \\ & x_n & \cdots & x_{j+1} & x_j & 1 & \cdots & 0 \\ + & x_n & \cdots & x_{j+1} & x_j & 1 & \cdots & 0 \\ \hline c_{n+1} & s_n & \cdots & s_{j+1} & s_j & 0 & \cdots & 0 \end{array}$$

right-most 1

$$\begin{array}{rr} \overset{\text{carry}}{c_{j+1}} & c_j \\ & x_j \\ + & x_j \\ \hline & s_j \end{array} \quad \begin{cases} x_j = 0 & \begin{array}{rr} 0 & c_j \\ & 0 \\ + & 0 \\ \hline & s_j = c_j \end{array} \\ x_j = 1 & \begin{array}{rr} 1 & c_j \\ & 1 \\ + & 1 \\ \hline & s_j = c_j \end{array} \end{cases}$$

$$s_j = c_j$$

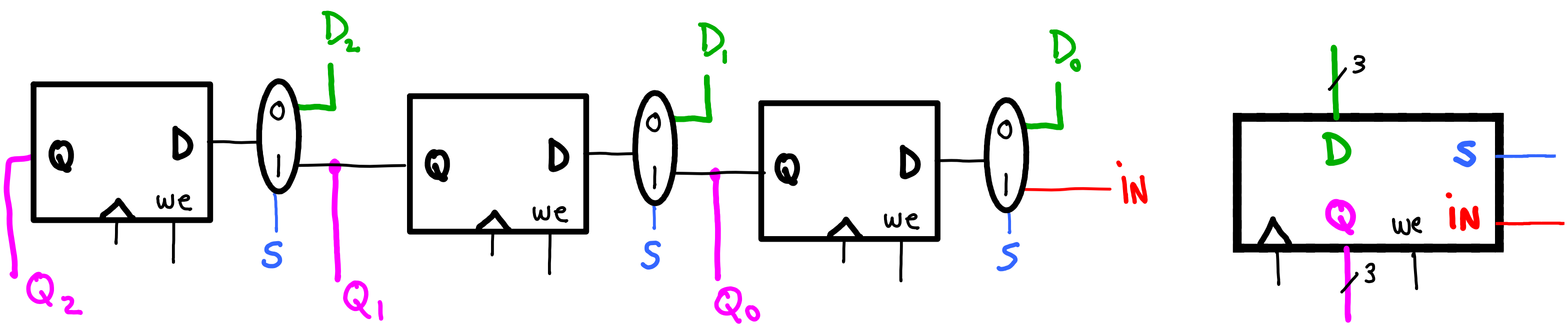
$$c_{j+1} = x_j$$

$$s_{j+1} = c_{j+1} = x_j$$

left shift

All bits of $x$ are shifted left.

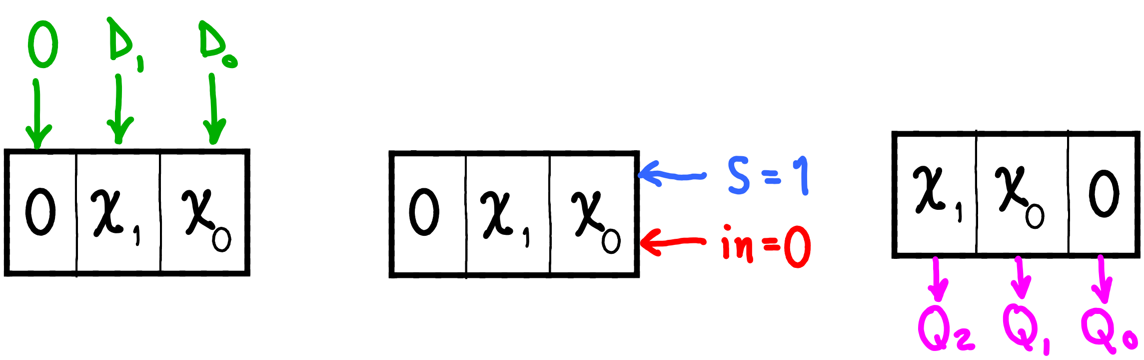
**X2 MULT == Parallel load, Left-Shift Register**




**Parallel Load :** we=1 and S=0: Q[2:0] <=== D[2:0]

**Shift Left :** we=1 and S=1: Q[2:0] <=== { Q[1:0], IN }

## 2-bit X 2 multiplier → 3-bit result




Signed numbers:

1. make unsigned;
2. multiply;
3. make signed;

**General MULTIPLY:** $y \times x$

$y$ : Multiplier

SUM of partial products ($PP_k$)

$PP_k$ == $x$ Left-Shifted $k$

$k$-th bit of $y$ is,

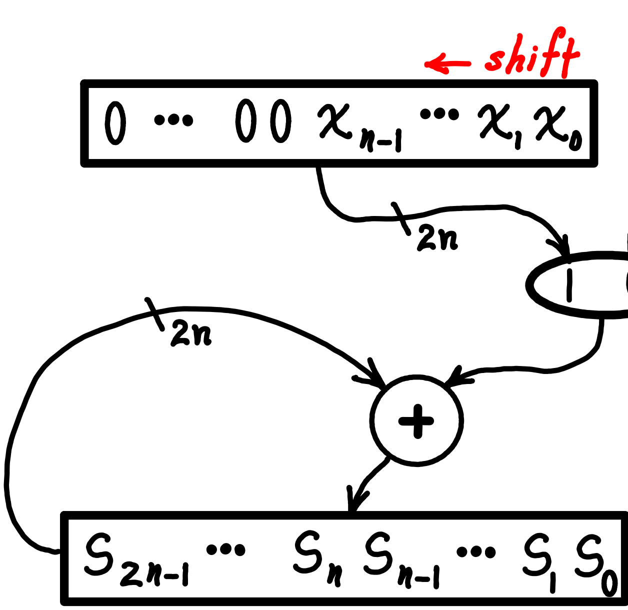
0 : add 0

1 : add $PP_k$

$$5 \times x == (101) \times (x_n x_{n-1} \ldots x_1 x_0)$$

$$== \quad (001) \times x_n x_{n-1} \ldots x_1 x_0 + (000) \times x_n x_{n-1} \ldots x_1 x_0 + (100) \times x_n x_{n-1} \ldots x_1 x_0$$

$$== \quad x_n x_{n-1} \ldots x_1 x_0 \text{ (0 left shifts)} + 0\ 0\ 0 \ldots 0\ 0\ 0 \text{ (1 left shift)} + x_n x_{n-1} \ldots x_1 x_0\ 0\ 0 \text{ (2 left shifts)}$$




$n$-bit MULTIPLIER

```
S = 0;
for  i=0;  i < n;  i++
     ADD;
     SHIFT;
```

--- Cost, Hardware:

3.5 2n-bit registers,
1 2n-bit ADD
1 2n-bit MUX: O( 2^n )
1 controller (iterator)

===> O( 2n ) + O( 2^n )

--- Cost, Delay per iteration

logn for MUX per iteration
2n for ADD per iteration

===> O( n ( logn + 2n ) )

Can we do better?

--- Hardware cost?

We can get rid of MUX
(How? Hint, write-enable.)

Other ways to use hardware?

--- Delay cost?
Alternative methods?

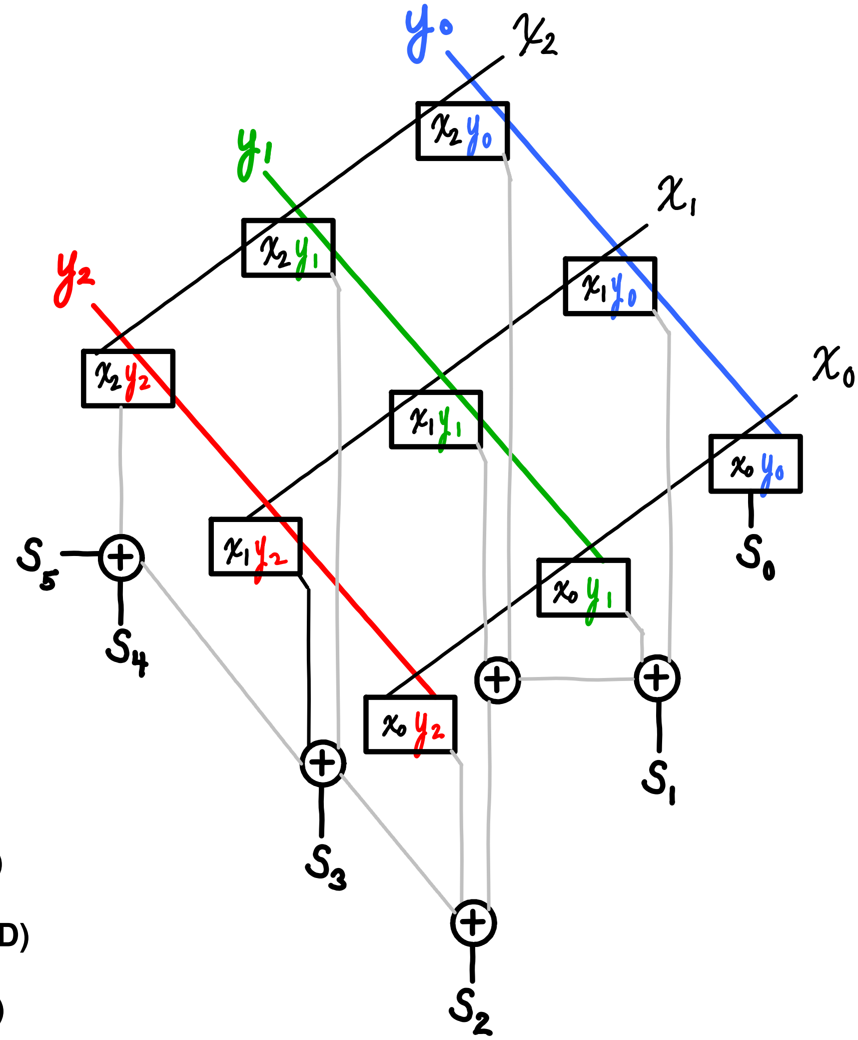
**3-bit Parallel Array Multiplier**

**--- bit-wise MULT == AND**

**--- 9 1-bit MULTs in parallel**

**--- 6-bit output**

**--- 6-step ADD delay**



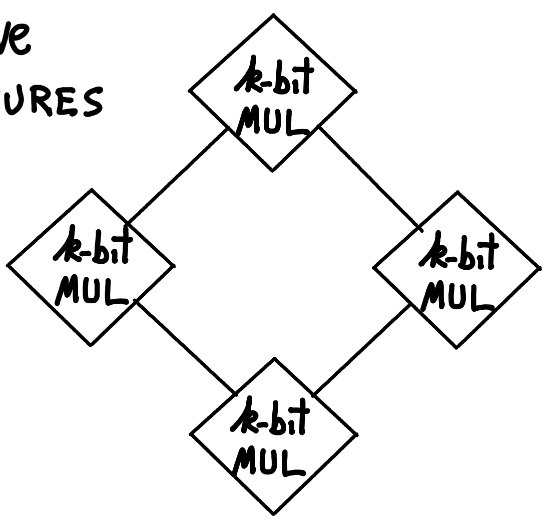

**n-bit Array Multiplier**

**--- Hardware: O( n^2 ) (1-bit ANDs)**
**+**
**O( 2n ) ( 1 2n-bit ADD)**

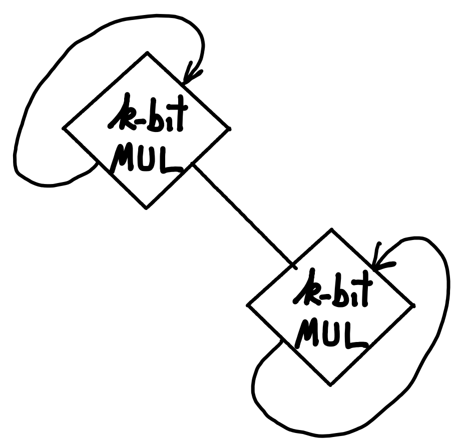
**--- Delay: O( 2n ) (2n-bit ADD)**

Is there something in between?

recursive STRUCTURES

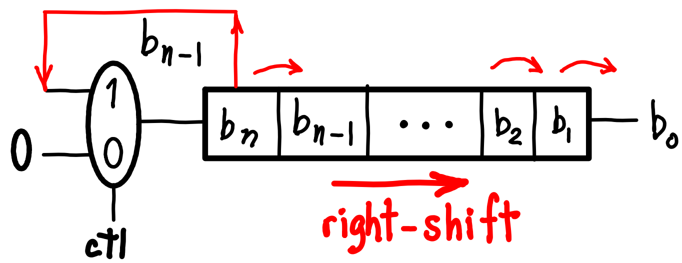



iterated recursive STRUCTURES



# Div

**IF** y * 2 === Left-Shift **THEN** y / 2 === Right-Shift




ctl = 0

Logical R-Shift
fills zeroes at left

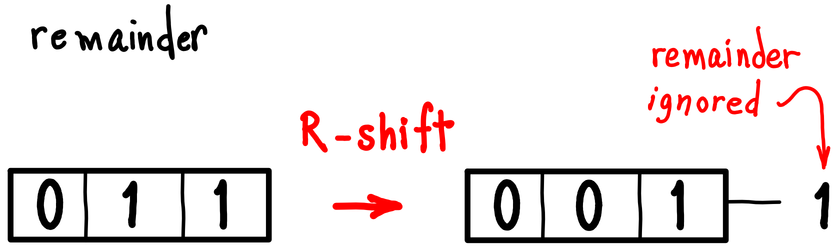
ctl = 1

Arithmetic R-Shift
2s-comp. sign extension

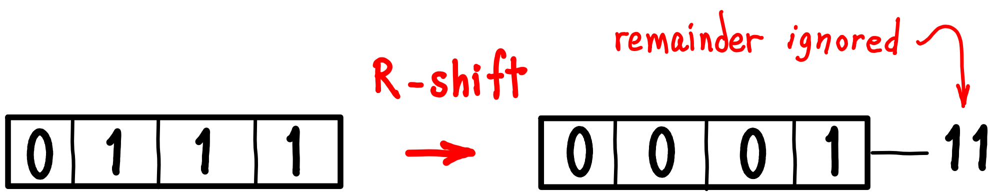
R-Shift(n) == divide-by-2^n . If divisor is not power of 2?

Integer Division = drop remainder

$3 \div 2 = 1$




$7 \div 4 = 1$




$x = q \cdot k + r$ $\begin{cases} k \text{ is divisor} \\ q \text{ is quotient} \\ r \text{ is remainder} \end{cases}$

$q = \#k\text{s in } x$

$\boxed{k\ k\ k\ k\ k\ k\ k\ k\ k\ r}$

think, unary representation

**divBySubtraction( x, k )**

```
q = 0
r = x

LOOP
    ( r < k )?  return q

    r = r - k
    q++
```

Time = $O(q)$

**divByAddition( x, k )**

```
q   = 0
sum = 0

LOOP
    sum = sum + k

  ( sum > x )? return q

    q++
```

We'd like $O(\log q)$ = # bits of $q$ → long division

1. Try $n$-*th* power of 10, $q_n 00\ldots0$

$$x \Longleftarrow x - k \times q_n 00\ldots0$$

IF $x < 0$ $\quad q_n = 0$

$$k\,\overline{)\,x\,}^{\,q_n 00\ldots0} \qquad - k \cdot q_n 00\ldots0 \qquad x$$

$$3\,\overline{)\,176\,}^{\,100} \qquad - 3 \cdot 100 \qquad -124$$

2. Try $(n-1)$-*th* power of 10

$$x \Longleftarrow x - k \times q_{n-1} 00\ldots0$$

IF $x$ non-negative $\quad$ save $q_{n-1}$

$$x \Longleftarrow x - k \times q_{n-1} 00\ldots0$$

$$3\,\overline{)\,176\,}^{\,50} \qquad - 3 \cdot 50 \qquad 26$$

save → 50

ADD

$$3\,\overline{)\,26\,}^{\,8} \qquad - 3 \cdot 8 \qquad 2$$

save → 8

58

Repeat until $x < k$

$q$ <== sum of saved partial quotients

$$== q_n\, q_{n-1} \ldots q_{n-2}\, q_1\, q_0$$

We can implement this method in hardware. In binary, $q_n$ is always 1 or 0.

$$x = kq = k q_n 2^n + \underbrace{k q_{n-1}}_{\text{partial product}} 2^{n-1} \cdots + k q_0 2^0$$

Try $q_i = 1$ $\qquad x - k\, 1\, 2^i$

IF non-negative, save $q_i = 1$

ELSE save $q_i = 0$

**Binary INTEGER (unsigned) DIVISON**
$x = kq + r$ **k = divisor**, **q = quotient**, r = remainder (ignore for now). **FIND q**.

try $q_n = 1$

$$\Big(x - k\cdot\underbrace{1\cdot 2^{n}}_{\text{L-shift } n}\Big) \geq 0 \ ? \quad \begin{cases} \text{yes: } q_n = 1 \\ \text{no: } q_n = 0 \end{cases}$$

| $q_n$ | 0 | 0 | 0 | 0 | ⋯ | 0 | 0 |
|---|---|---|---|---|---|---|---|

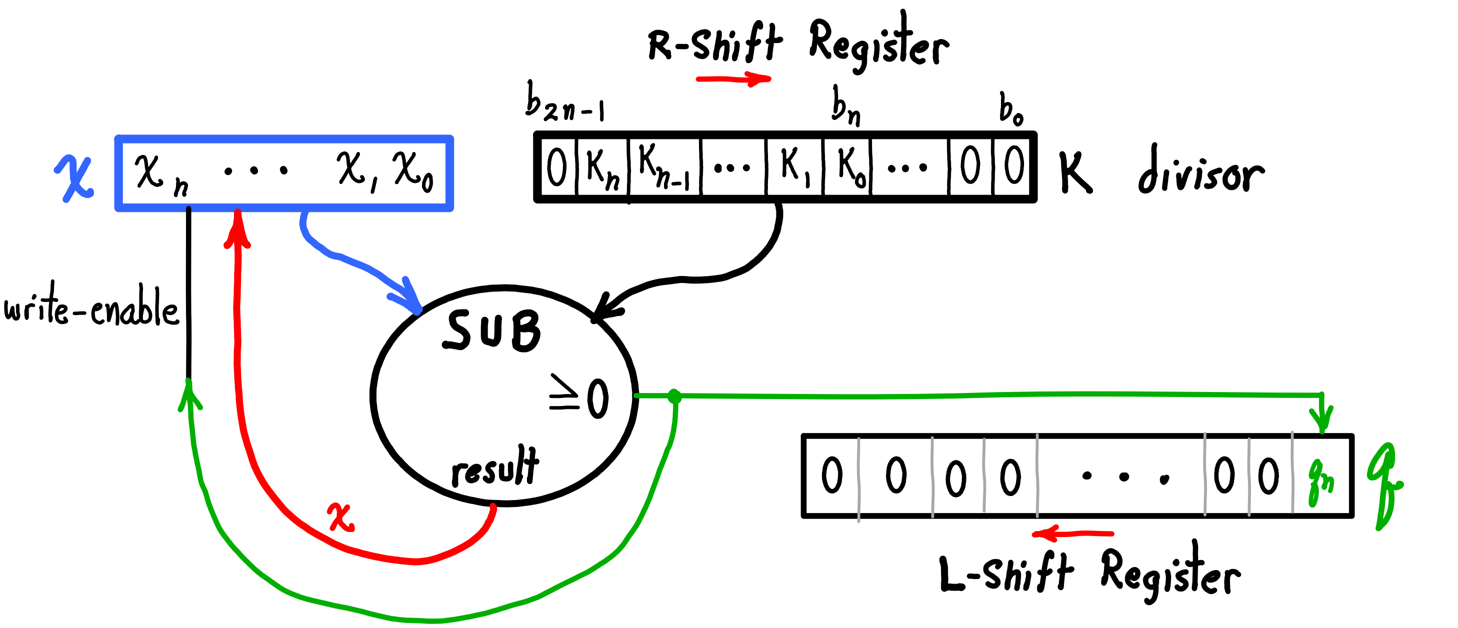
$q$

$$x \leftarrow k\, q_{n-1} 2^{n-1} + \cdots + k\, q_0 2^0$$

$$\Big(x - k\cdot\underbrace{1\cdot 2^{n-1}}_{\text{L-shift } n-1}\Big) \geq 0 \ ? \quad \begin{cases} \text{yes: } 1 \\ \text{no: } 0 \end{cases}$$

| $q_n$ | $q_{n-1}$ | 0 | 0 | ⋯ | 0 | 0 | 0 |
|---|---|---|---|---|---|---|---|

$q$




**After each SUB**

register $q$ gets 1 or 0 as low bit ( $q_i$ )

register $x$ is written if $q_i$ = 1

register $k$ Right-Shifted ( initially, $k$ is Left-Shifted $n$ bits )

register $q$ Left-Shifted ( after $n$ shifts $q_n$ is left-most bit in $q$ )

$$\text{time} = n\,\underline{O(2n)}$$

2n bit SUB

**Approximate Methods**

**MUL via approx. log**

$$N = \overset{b_{n-1}}{0}\,0\,0\,|\,\overset{b_k}{0}\,0\,|\,|\,0\,|\,|\,0\,|\,0\,|\,0\,\overset{b_0}{0}$$

$$N = b_k 2^k + \left(b_{k-1}2^{k-1} + b_{k-2}2^{k-2} + \cdots b_0\right)$$

$$= 1\cdot 2^k + \left(b_{k-1}2^{k-1} + b_{k-2}2^{k-2} + \cdots b_0\right)$$

$$= 2^k\left(1 + b_{k-1}2^{-1} + b_{k-2}2^{-2} + \cdots + b_0 2^{-k}\right)$$

$$= 2^k\left(1.\,b_{k-1}\,b_{k-2}\ldots b_0\right)$$

$$\log(N) = k + \log(1.\,b_{k-1}\,b_{k-2}\ldots b_0)$$

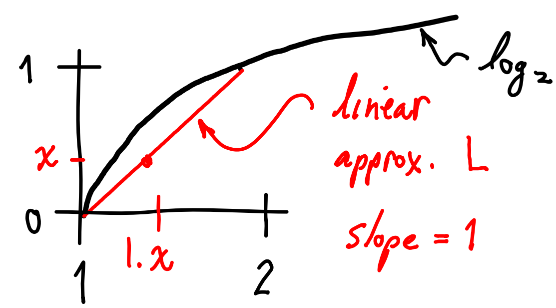
$L$ = linear approx. log

$$L(N) = k + (0.\,b_{k-1}\,b_{k-2}\ldots b_0)$$

$$+\quad L(M) = j + (0.\,b_{j-1}\,b_{j-1}\ \ldots\ \ldots\ b_0)$$

$$k+j + (b_0 \bullet b_{-1}\ \cdots\ b_{-j})$$

$$k+j+b_0 + (0\,.\,b_{-1}\cdots b_{-j})$$




$$L(1.x) = x$$

$$L^{-1}(x) = 1.x$$

$$L^{-1} \rightarrow 2^{k+j+b_0} \times L^{-1}(0.b_{-1}\cdots b_{-j})$$

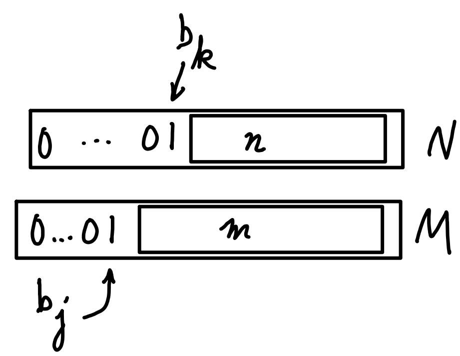
$$= 2^r \times (1.\,b_{-1}\cdots b_{-j})$$

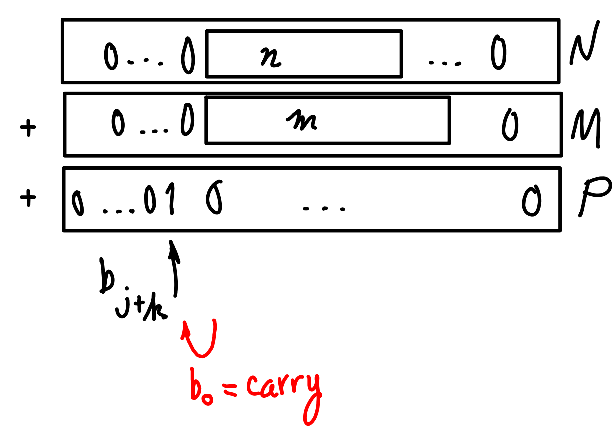
$$= 2^r + b_{-1}2^{r-1} + b_{-2}2^{r-2} + \cdots + b_{-j}2^{r-j}$$

$$= 0001\ b_{-1}\ b_{-2}\ldots b_{-j}\ 0\ldots 0$$

$r^{th}$ position

$k+j+b_0$



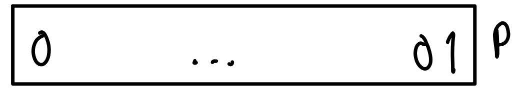



```
k = n
until( L-Shift( N ).carry_out == 1 )       k--

j = n
until( L-Shift( M ).carry_out == 1 )       j--

L-Shift( P, k+j )
R-Shift( N, n-(k+j) )
R-Shift( M, n-(k+j) )

P <== N + M + P
```




5 $O(n)$ shifts

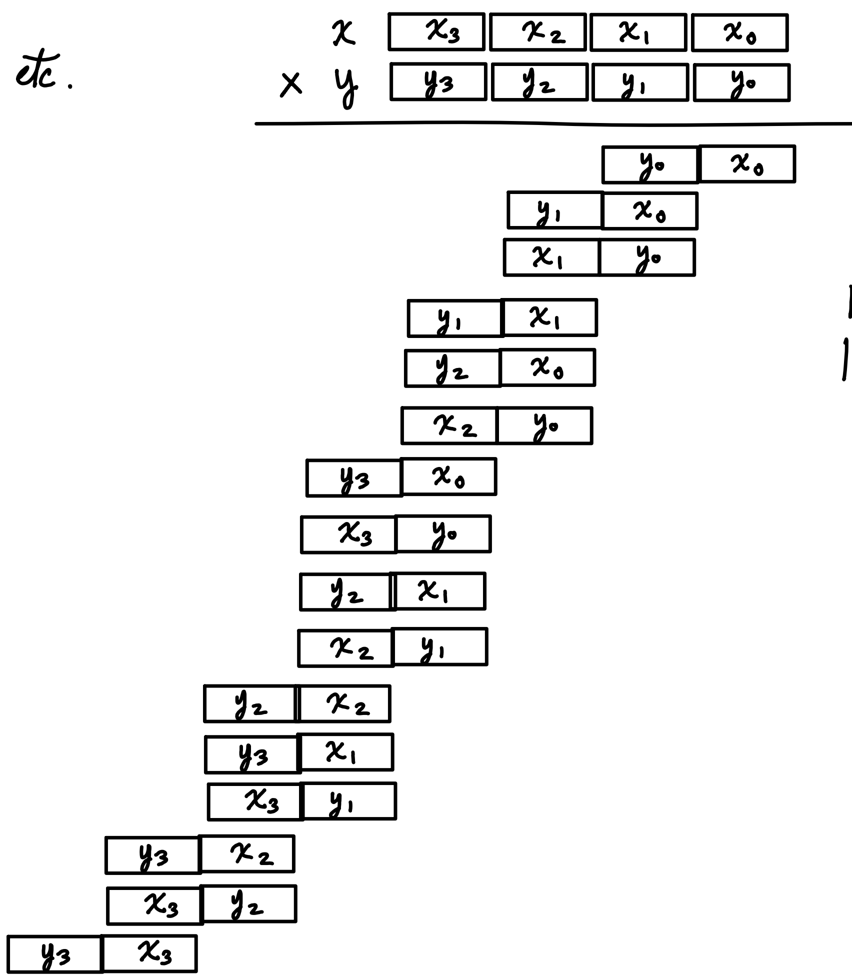
2 $O(n)$ ADDs

$N \times M$ error $\approx$ 0.11

※ bits $= n = 4z$

$x_0$ is $z$ bits

$x_1$ is $z$ bits, etc.




16 ADDs

16 MULs

$\log(A)(1+\tfrac{1}{8}) + \log(B)(1+\tfrac{1}{8})$ error in logs of $\tfrac{1}{8}$

$= (\log A + \log B)(1+\tfrac{1}{8})$

$\dfrac{2^{(\log A + \log B)(1+1/8)}}{A\cdot B} = (1+e)$ error in product of $\cancel{8}$ $e$

$\dfrac{(A\cdot B)^{(1+1/8)}}{A\cdot B} = \dfrac{AB}{AB}(AB)^{1/8} = (1+e)$

$(AB)^{1/8} = (1+e)$

$A = 2^k = B$ $\quad (2^{2k})^{1/8} = (1+e)$

$\dfrac{AB^{(1+1/2k)}}{AB}$

$2^{2k/8} = (1+e) < \cancel{(1+\varepsilon)}(1+2^{1/\varepsilon})$

$\dfrac{2^{2k(1+1/2k)}}{2^{2k}} = 2^{2k+1-2k}$

$< ((2^{\varepsilon}+1)/2^{\varepsilon})$

$2k/8 < \log(2^{\varepsilon}+1) - \cancel{A}\,\varepsilon$

$= 2^1$

$< \cancel{8}\log(2^2+1) - 2 \approx 1/2$ $\quad \varepsilon = 2$

$2k < 8$

$n$ bits $\rightarrow k = n$

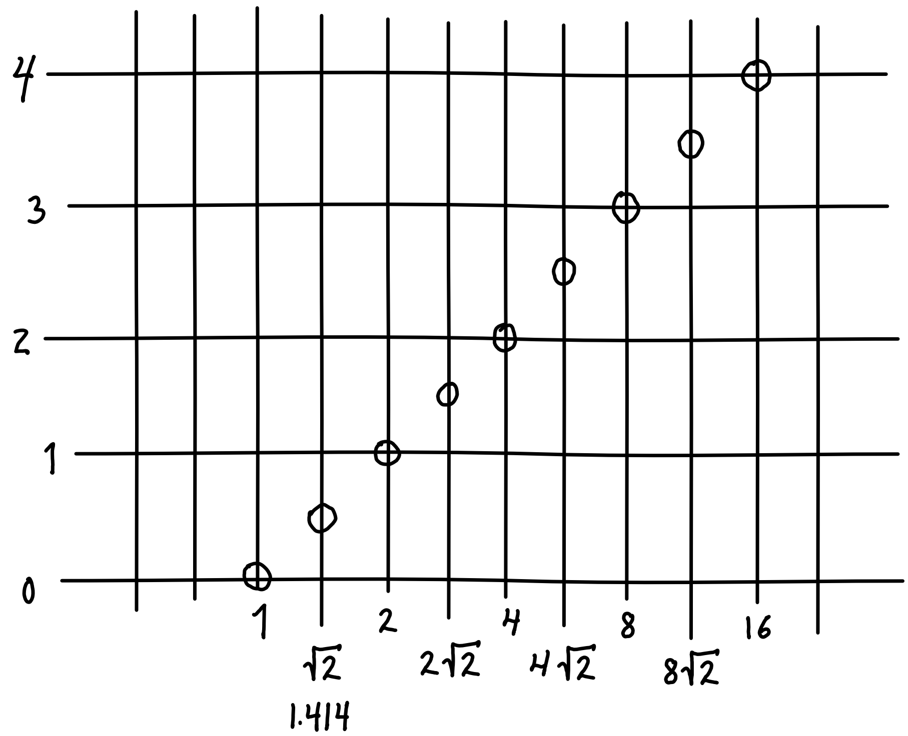
$2(32) < 8$

if error in log < 1/64
error in mult $\approx 0$ $\quad 4k < 8$

$\log(A)(1+1/128) + \log(B)(1+1/128)$

$= (\log A + \log B)(1+1/128)$

$\dfrac{2^{(\log A + \log B)}(1+1/128)}{A\cdot B} = \dfrac{A\cdot B^{129/128}}{AB} = \dfrac{(2^{32}\cdot 2^{32})^{129/128}}{2^{32}\cdot 2^{32}} =$




**3 X 4** ( notation, let RT2 == SQRT(2) )

**Interpolate to log(3):**
4 - 2.828 == 1.172 value range from 2 RT2 to 4
3 - 2.828 == 0.172 value range from 2 RT2 to 3
0.172 / 1.172 == 0.148 value fractional range to 3
2 - 1.5 = .5 log range from log(2 RT2) to log(4)
.5 X 0.148 = 0.074 interpolate log fractional part to log(3))
log( 3 ) = 1.5 + 0.074 = 1.574 interpolate log(3)

real value is about 1.585, we are off by about 1 part in 160

**mult by adding logs:**
log(3) + log(4) == (1.574 + 2 ) == 3.574

**Interpolate to exp2( 3.574 ):**
3.5 - 3.574 == 0.074 (range of logs from 3.574 to 4)
0.074 / 0.5 == 0.148 (fractional range of logs)
16 - 8(1.414) = 16 - 11.31 == 4.69 (range of values)
(0.148)(4.68) == 0.694 (fractional part of range)
11.31 + 0.694 == 12.004 off by about 1/3000