Clocked Sequential System Design

Example 1 –
Multipliers
(Gradeschool, Modified Gradeschool)

Multiply Example

```
        10111001   (185)
      x 11010111   (215)
      ----------
        10111001
       10111001
      10111001
     00000000
    10111001
   00000000
  10111001
 10111001
----------------
1001101101011111 (39775)
```

```
          10111001
        x 11010111
        ----------

  0000000000000000  <- Start
                       with zero
```

```
          10111001
        x 11010111
        ----------
          10111001 <- 1st Partial
  0000000000000000    Product (PP)
```

```
          10111001
        x 11010111
        ----------
          10111001
  0000000010111001 <- sum of Prod, PP
```

```
          10111001
        x 11010111
        ----------
         10111001 <- second PP
  0000000010111001
```

```
        10111001
      x 11010111
      ----------
       10111001
0000001000101011 <- running sum
```

```
        10111001
      x 11010111
      ----------
      10111001   <- third PP
0000001000101011
```

```
        10111001
      x 11010111
      ----------
      10111001
0000001100001111 <- running sum
```

```
        10111001
      x 11010111
      ----------
     00000000    <- fourth PP
0000001100001111
```

```
   10111001
 x 11010111
 ----------
  00000000
0000001100001111 <- running sum
```

```
   10111001
 x 11010111
 ----------
 10111001     <- fifth PP
0000001100001111
```

```
          10111001
        x 11010111
        ----------
      10111001
  0000111010011111 <- running sum
```

```
          10111001
        x 11010111
        ----------
     00000000      <- sixth PP
  0000111010011111
```

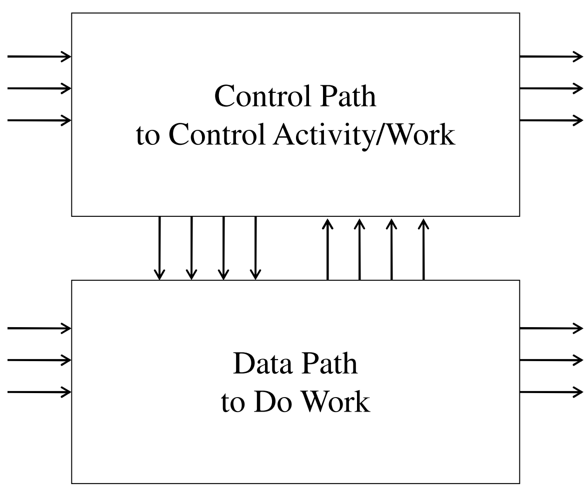
```
         10111001
       x 11010111
       ----------
    00000000
0000111010011111 <- running sum
```

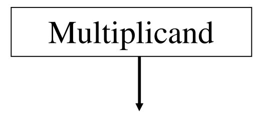
```
         10111001
       x 11010111
       ----------
  10111001      <- seventh PP
0000111010011111
```

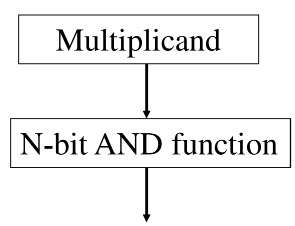
```
        10111001
      x 11010111
      ----------
  10111001
0011110011011111 <- running sum
```

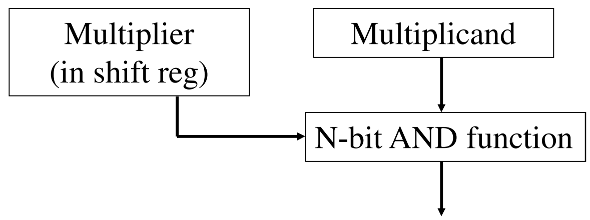
```
        10111001
      x 11010111
      ----------
 10111001        <- final PP
0011110011011111
```

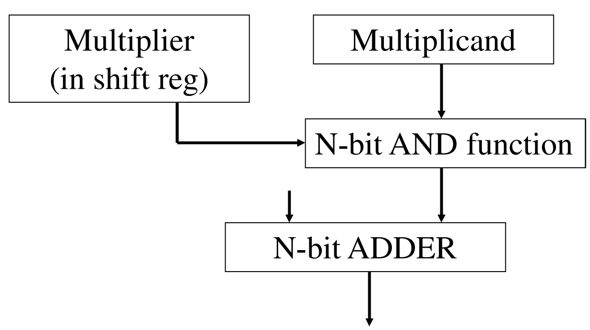
```
          10111001
        x 11010111
        ----------
   10111001
  1110101100011111 <- final sum
```

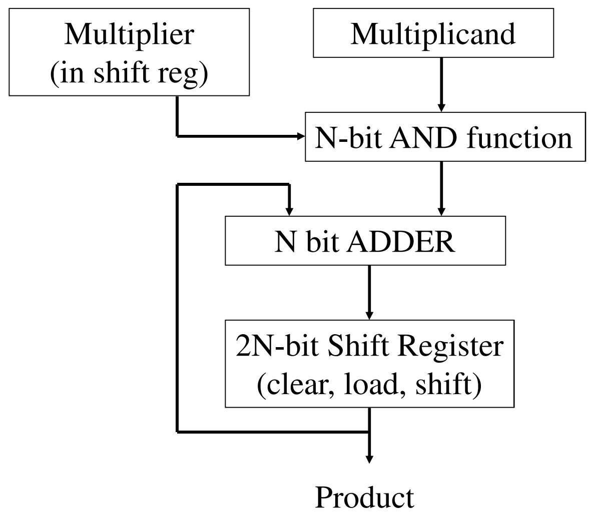



Multiplicand

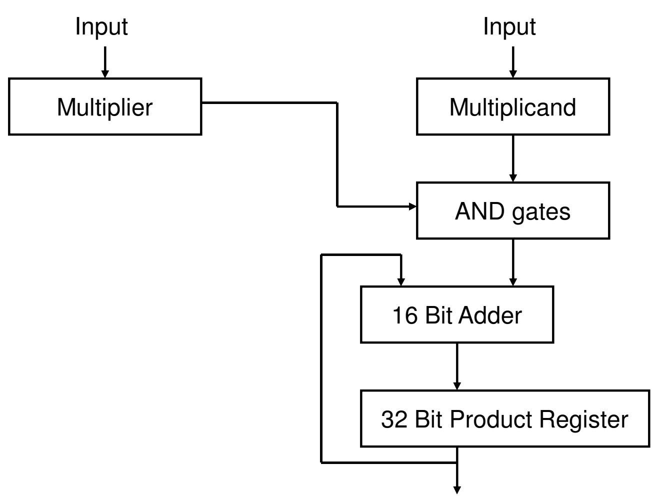

Multiplicand
N-bit AND function


Multiplier
(in shift reg)
Multiplicand
N-bit AND function

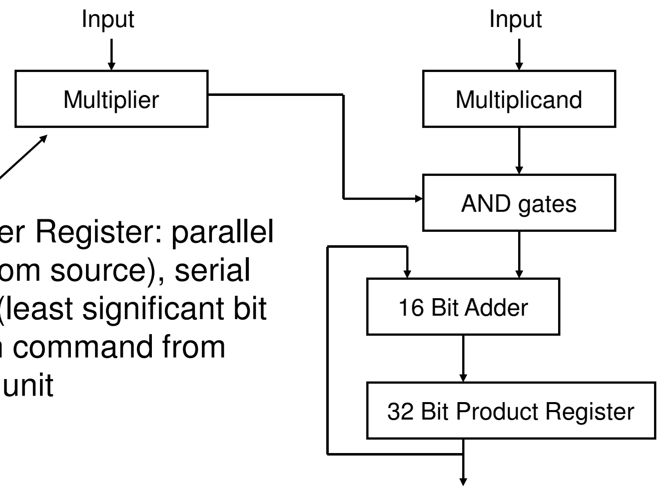

Multiplier
(in shift reg)
Multiplicand
N-bit AND function
N-bit ADDER



Multiplication – Simple Gradeschool Algorithm for 16 Bits (32 Bit Result)

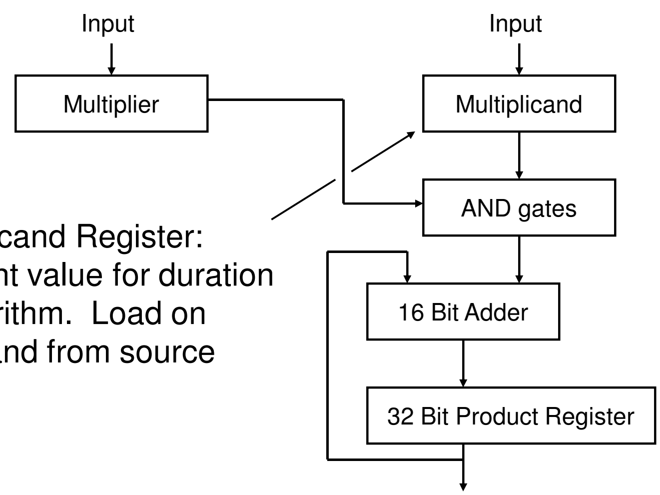


Multiplication – Simple Gradeschool Algorithm for 16 Bits (32 Bit Result)



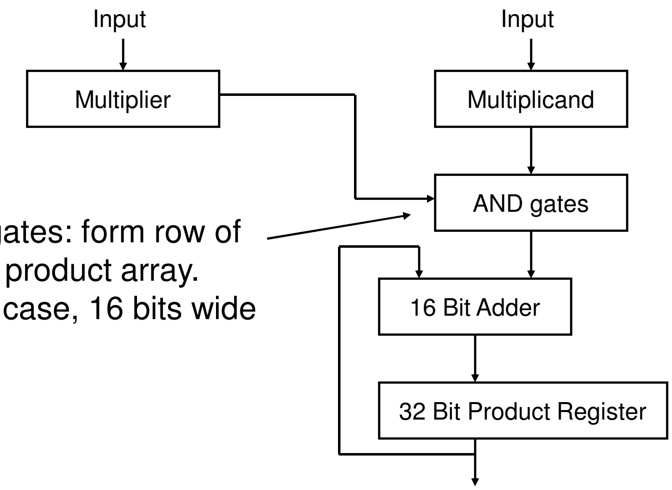
Multiplier Register: parallel load (from source), serial output (least significant bit first) on command from control unit

Multiplication – Simple Gradeschool Algorithm for 16 Bits (32 Bit Result)



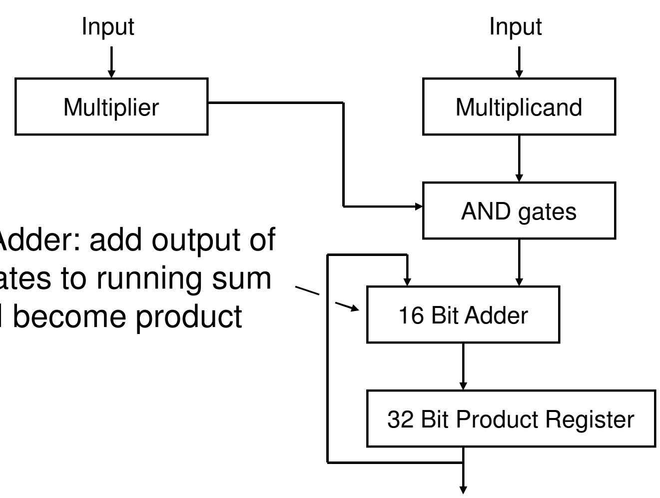
Multiplicand Register: constant value for duration of algorithm. Load on command from source

Multiplication – Simple Gradeschool Algorithm for 16 Bits (32 Bit Result)



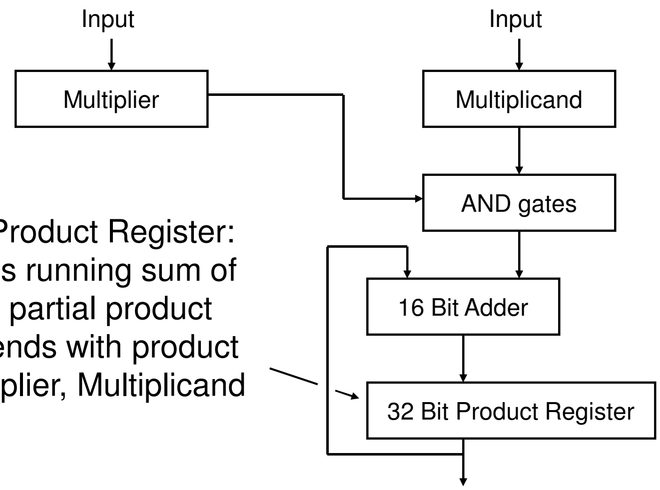
AND gates: form row of partial product array.
In this case, 16 bits wide

Multiplication – Simple Gradeschool Algorithm for 16 Bits (32 Bit Result)



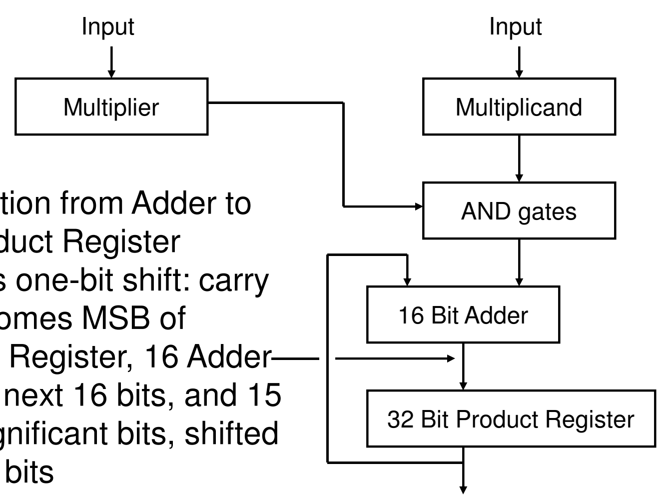
16 Bit Adder: add output of AND gates to running sum that will become product

Multiplication – Simple Gradeschool Algorithm for 16 Bits (32 Bit Result)



32 Bit Product Register: contains running sum of rows of partial product array; ends with product of Multiplier, Multiplicand

Multiplication – Simple Gradeschool Algorithm for 16 Bits (32 Bit Result)



Connection from Adder to the Product Register includes one-bit shift: carry out becomes MSB of Product Register, 16 Adder outputs next 16 bits, and 15 least significant bits, shifted product bits

Register: asynchronous and synchronous behavior

```
NAME_OF_PROC:
process ( <clock, reset, set go here> ) is

begin

  if <asynchronous signals tested here> then
    <put set, reset stuff here>
  elsif RISING_EDGE ( clock ) then
     < put synchronous stuff here, in particular… >
    if <enabling condition> then
      <action/activity of register goes here>
    end if;
  end if;

end process NAME_OF_PROC;
```

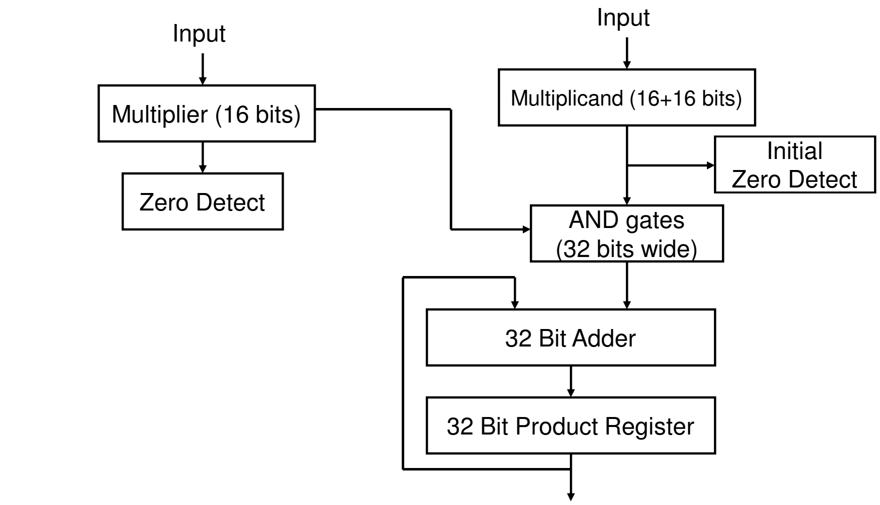
Multiplication – Simple Gradeschool Algorithm for 16 Bits (32 Bit Result)

Step 1:
- Clear Product Register
- Clear Counter
- Load Multiplicand
- Load Multiplier

Step 2: (repeat 16 times)
- Increment Counter
- Load Product Register
- Shift Multiplier

Step 3:
- Done

Implement the Modified Gradeschool Multiplication Algorithm



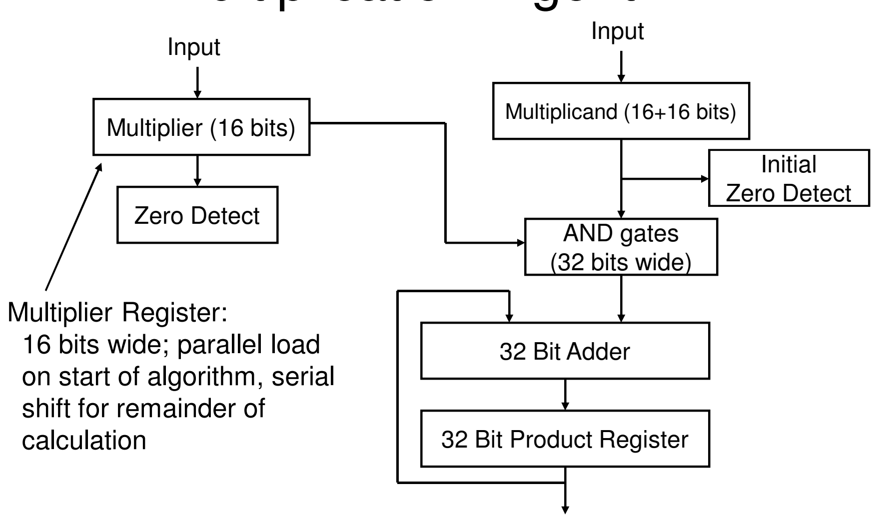
Biggest Unsigned Binary Multiply – 16 Bits × 16 Bits

```
                1111111111111111
                1111111111111111
                ----------------
                1111111111111111
               1111111111111111
              1111111111111111
             1111111111111111
            1111111111111111
           1111111111111111
          1111111111111111
         1111111111111111
        1111111111111111
       1111111111111111
      1111111111111111
     1111111111111111
    1111111111111111
   1111111111111111
  1111111111111111
 1111111111111111
--------------------------------
11111111111111100000000000000001
```

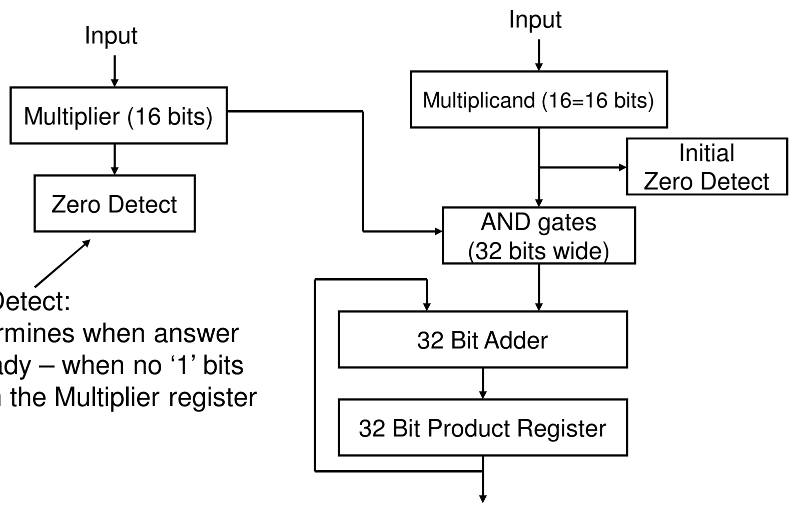
Biggest Unsigned Binary Multiply – 16 Bits × 16 Bits

```
                1111111111111111
                1111111111111111
                ----------------
0000000000000000111111111111111 1
00000000000000011111111111111110
00000000000000111111111111111100
00000000000001111111111111111000
00000000000011111111111111110000
00000000000111111111111111100000
00000000001111111111111111000000
00000000011111111111111110000000
00000000111111111111111100000000
00000001111111111111111000000000
00000011111111111111110000000000
00000111111111111111100000000000
00001111111111111111000000000000
00011111111111111110000000000000
00111111111111111100000000000000
01111111111111111000000000000000
--------------------------------
11111111111111100000000000000001
```

Implement the Modified Gradeschool Multiplication Algorithm

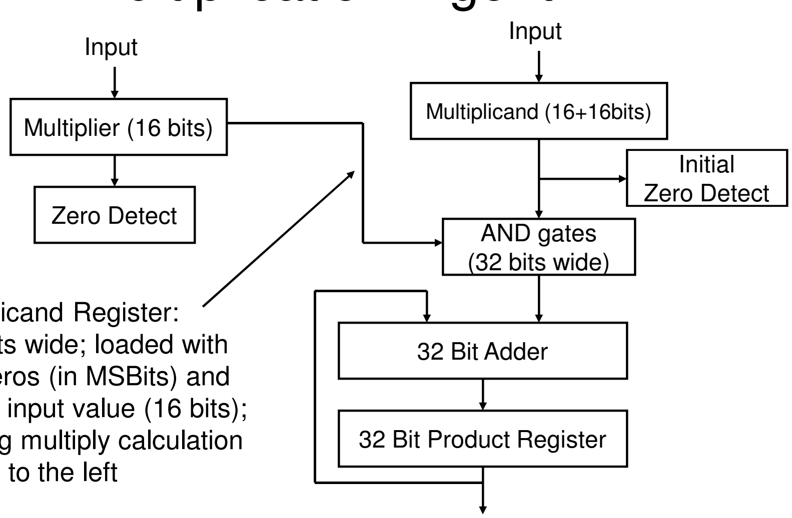


Implement the Modified Gradeschool Multiplication Algorithm



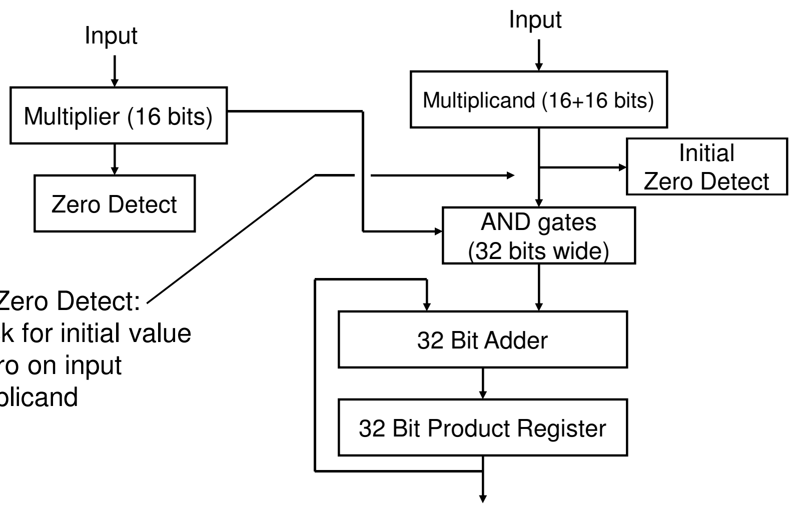
Zero Detect:
Determines when answer is ready – when no ‘1’ bits left in the Multiplier register

Implement the Modified Gradeschool Multiplication Algorithm



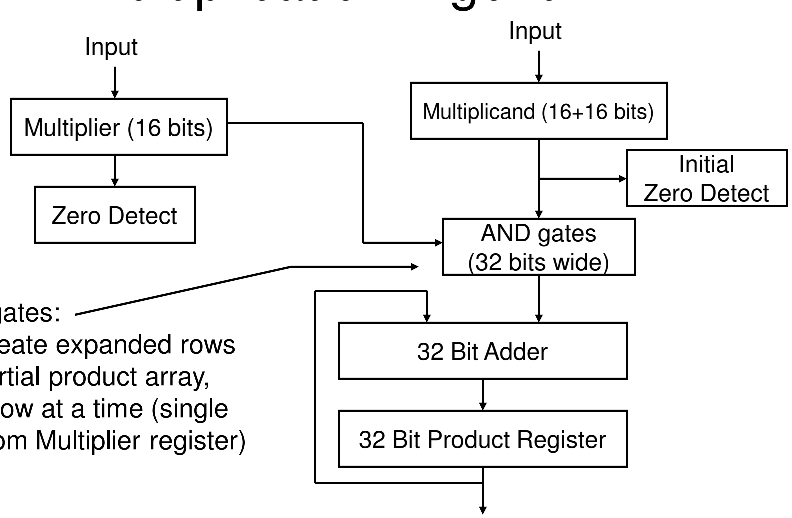
Multiplicand Register:
32 bits wide; loaded with 16 zeros (in MSBits) and initial input value (16 bits); during multiply calculation shifts to the left

Implement the Modified Gradeschool Multiplication Algorithm



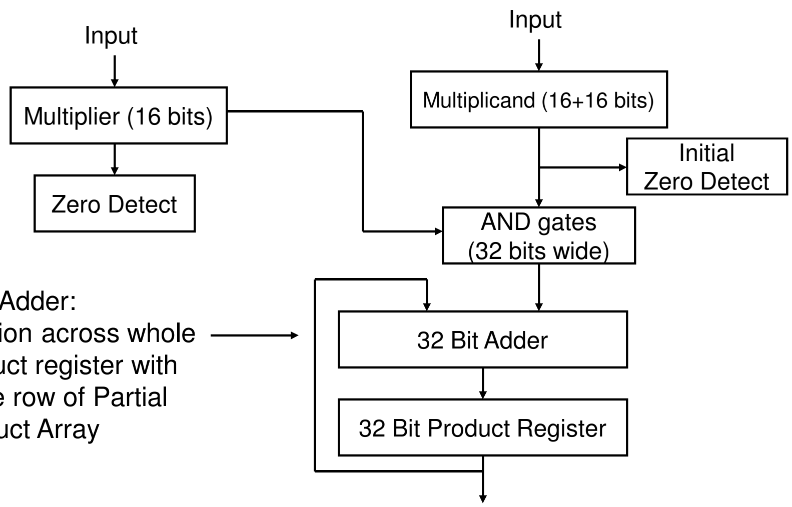
Initial Zero Detect:
Check for initial value of zero on input Multiplicand

Implement the Modified Gradeschool Multiplication Algorithm



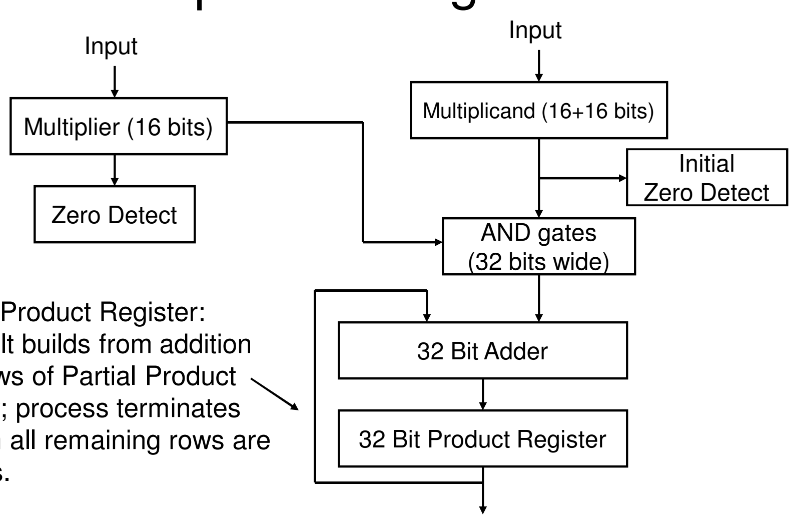
AND gates:
To create expanded rows of partial product array, one row at a time (single bit from Multiplier register)

Implement the Modified Gradeschool Multiplication Algorithm



32 Bit Adder:
Addition across whole product register with entire row of Partial Product Array

Implement the Modified Gradeschool Multiplication Algorithm



32 Bit Product Register:
Result builds from addition of rows of Partial Product Array; process terminates when all remaining rows are zeros.

Multiplication – Modified Gradeschool Algorithm for 16 Bits (32 Bit Result)

Step 1:
- Clear Product Register
- Load Multiplicand
- Load Multiplier

Step 2:
- If MIER = 0 OR MCAND = 0, done

Step 3:
- Load Product Register
- Shift Multiplier and Multiplicand

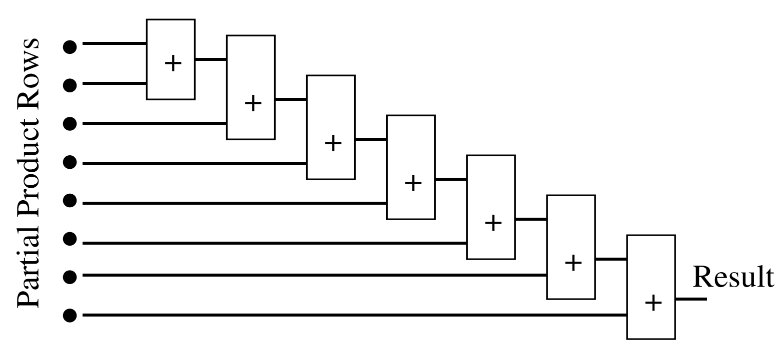
Step 4:
- If MIER = 0, then Done
- else Go to Step 3

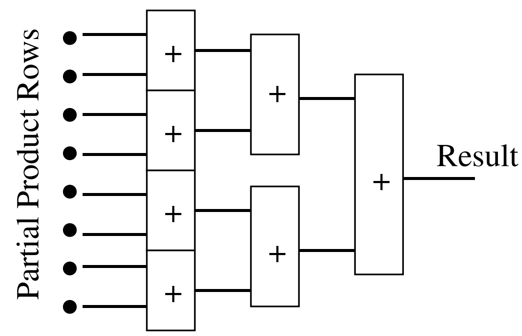
Newer Multiply Techniques

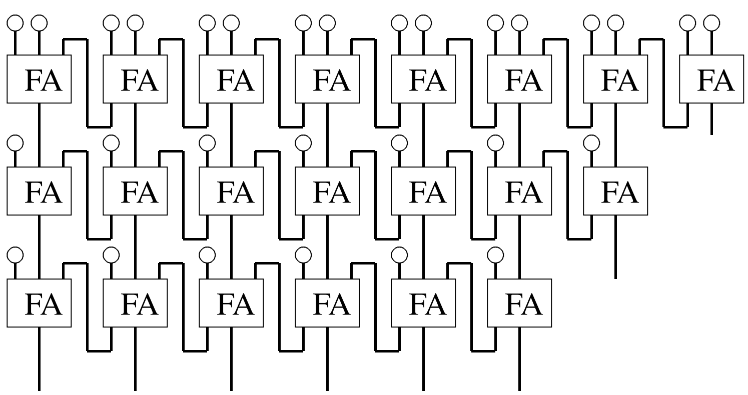
Use More Gates!

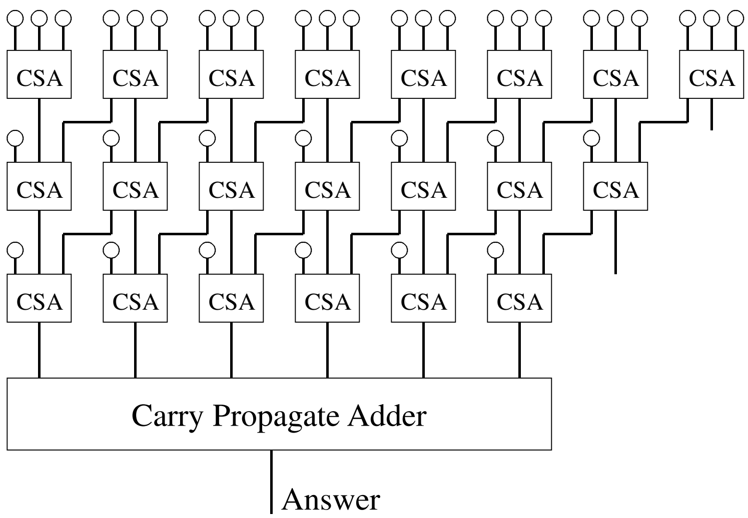
Very Simple Example

```
      10111001
    x 11010111
    ----------
      10111001→●
     10111001-→●
    10111001--→●
   00000000---→●
  10111001----→●
 00000000-----→●
10111001------→●
10111001-------→●
----------------
1001101101011111
```

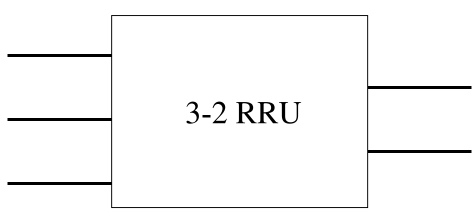



Partial Product Rows
+
+
+
+
+
+
+
Result


FA



Carry-Save Adder:
Minimal Row Reduction Unit (RRU)
Input: 3 rows
Output: 2 rows

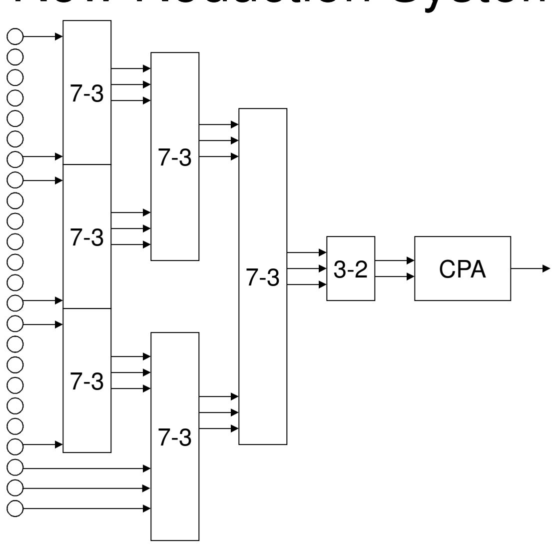


Note: Care must be taken to make sure that the significance of the bits is handled properly

Row Reduction: any combination of 2^N-1 rows → N rows

2^N-1		N
3	$\longrightarrow$	2
7	$\longrightarrow$	3
15	$\longrightarrow$	4
31	$\longrightarrow$	5

Row Reduction System – 24 Bits



Division – (Gradeschool)

Division Process

A is integer (32 bits)
B is integer (32 bits)

Form A/B to give
  MQ – the quotient
  ACC – the remainder

Explicatory Example: 4 bits

A is integer (4 bits)
B is integer (4 bits)

Form A/B to give
- MQ – the quotient (4 bits)
- ACC – the remainder (4 bits)

A / B :
- let A = 0111
- let B = 0101

Answer should be: 0001
with a remainder: 0010

Grade School Approach

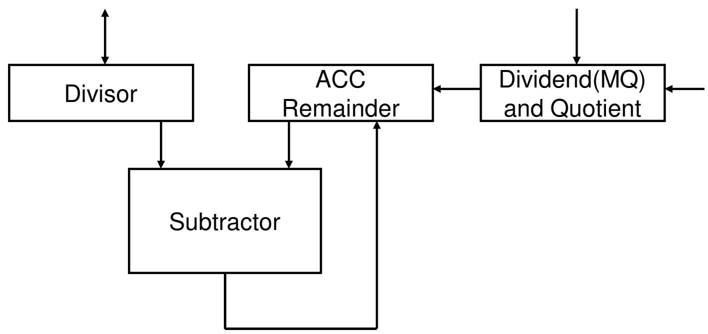
```
               1
0101 | 0111
       0101
       ----
       0010
```

Grade School Approach

```
                            Quotient
               1
0101 | 0111
       0101                 Dividend
Divisor ----
       0010

           Remainder
```

Activity of Divide



More General Approach

`0101 ⟌ 0000 0111`

Denominator of initial problem

Start condition: all zeros (ACC)

Numerator of initial problem (MQ)

Initial conditions: set up registers as shown here, then left-shift the ACC-MQ pair

More General Approach

`0101 ⌈0000 1110`

First bit of answer: zero, since 0101 is bigger than 0000, so remember that and then left shift ACC-MQ

More General Approach

`0101 ⌈0001 110x   0`

Second bit of answer: zero, since 0101 is bigger than 0001, so remember that and then left shift ACC-MQ

More General Approach

```
0101 ⌐ 0011 10xx   00
```

Third bit of answer: zero, since 0101
is bigger than 0011, so remember that
and then left shift ACC-MQ

More General Approach

```
0101 ⌐ 0111 0xxx   000
```

Fourth bit of answer: one, since 0101
is smaller than 0111, so replace ACC with
0111 – 0101 = 0010, and shift only MQ
with the ‘1’ bit

More General Approach

```
0101 ⌈0010 xxxx   0001
```

Final result: 0001 with remainder 0010

More General Approach (2)

```
1011 ⌈0000 0010
```

Initial conditions: set up registers as shown here, then left-shift the ACC-MQ pair
(Problem here is 2 / 11)

More General Approach (2)

`1011 ⌈0000 0100`

First bit of answer: zero, since 1011 is bigger than 0000, so remember that and then left shift ACC-MQ

More General Approach (2)

`1011 ⌈0000 100x   0`

Second bit of answer: zero, since 1011 is bigger than 0000, so remember that and then left shift ACC-MQ

More General Approach (2)

```
1011 ⌈ 0001  00xx   00
```

Third bit of answer: zero, since 1011 is bigger than 0001, so remember that and then left shift ACC-MQ

More General Approach (2)

```
1011 ⌈ 0010  0xxx   000
```

Fourth bit of answer: zero, since 1011 is bigger than 0010, so remember that, and since last step of algorithm, leave ACC as is and shift only MQ

More General Approach (2)

```
1011 ⌈0010 xxxx   0000
```

Final result: 0000 with remainder 0010

More General Approach (3)

```
0010 ⌈0000 1101
```

Initial conditions: set up registers as shown here, then left-shift the ACC-MQ pair
(Problem here is 13 / 2)

More General Approach (3)

`0010 ⌈ 0001 1010`

First bit of answer: zero, since 0010 is bigger than 0001, so remember that and then left shift ACC-MQ

More General Approach (3)

`0010 ⌈ 0011 010x   0`

Second bit of answer: one, since 0010 is less than 0011, so replace ACC with 0011 – 0010 = 0001, then remember the ‘1’ and left shift ACC-MQ

More General Approach (3)

```
0010 ⌈ 0010 10xx   01
```

Third bit of answer: one, since 0010 is same as 0010, so replace ACC with 0010 – 0010 = 0000, remember the ‘1’ and then left shift ACC-MQ

More General Approach (3)

```
0010 ⌈ 0001 0xxx   011
```

Fourth bit of answer: zero, since 0010 is bigger than 0001, so remember that, and since last step of algorithm, leave ACC as is and shift only MQ

More General Approach (3)

```
0010 ⎾0001 xxxx   0110
```

Final result: 0110 with remainder 0001

Division Algorithm

Initialization:

- Divisor (D_S) <= input value
- Dividend (MQ) <= input value
- Remainder (ACC) <= zero
- Count <= zero

Note: this algorithm takes advantage of the fact that the MQ register starts with Dividend and ends with Quotient

Division Algorithm

Step 2: Left shift ACC-MQ pair

Step 3: if ACC >= D_S then
  ACC <= ACC – D_S; FF <= ‘1’;
 else
  FF <= ‘0’;
 end if;
 Count <= Count + 1;

Division Algorithm

Step 4: if Last Iteration then
  left shift MQ-FF;
 else
  left shift ACC-MQ-FF;
 end if;

Step 5: if Count < Termination then
  return to Step 3;
 else
  Done;
 end if;

High Speed Division

- Use High Speed Multiplier
- Follows Newton-Raphson iteration method
- Can find correct answer after few iterations

High Speed Divide

$$\frac{\mathrm{A}}{\mathrm{B}}$$

High Speed Divide

$$\frac{A \times f_0}{B \times f_0}$$

High Speed Divide

$$\frac{A \times f_0}{B \times f_0}$$

let $B = 1 - x$, then

$x = 1 - B$, and let

$f_0 = 1 + x = 2 - B$

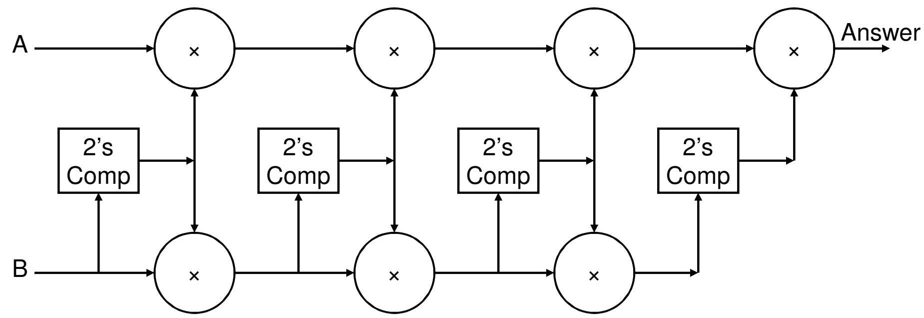
High Speed Divide

$$\frac{A \times f_0}{B \times f_0}$$

$$\begin{aligned}
&\text{let } B = 1 - x\text{, then} \\
&x = 1 - B\text{, and let} \\
&f_0 = 1 + x = 2 - B \\
&\text{AND:} \\
&B \times f0 = (1 - x) \times (1 + x) \\
&\qquad = 1 - x^2
\end{aligned}$$

High Speed Divide

$$\frac{A \times f_0 \times f_1 \times f_2 \times f_3 \times f_4}{B \times f_0 \times f_1 \times f_2 \times f_3 \times f_4}$$



Floating Point Arithmetic

Addition of Floating Point Numbers

1634.75 = 11001100010.1100
= $1.10011000101100 \times 2^{10}$

IEEE: $10 + 127 = 137_{10} = 10001001_2$ so,
0 10001001 10011000101100000000000

Addition of Floating Point Numbers

1634.75 = 11001100010.1100
= $1.10011000101100 \times 2^{10}$

IEEE: $10 + 127 = 137_{10} = 10001001_2$ so,
0 10001001 10011000101100000000000

498.0625 = 111110010.0001
= $1.111100100001 \times 2^{8}$

IEEE: $8 + 127 = 135_{10} = 10000111_2$ so,
0 10000111 11110010000100000000000

Addition of Floating Point Numbers

Step 1: Determine larger of two numbers by comparing the exponents

Number A: 10001001
Number B: 10000111
A – B = 00000010

Number A is bigger than Number B by a factor of about 4 (2^2)

Addition of Floating Point Numbers

Step 2: Represent numbers in same format of user's choosing. Choice here is two words (64 bits) with p=56.
Note: keep track of fact that this is x 2^{10}

0000 0001. 1001 1000 1011 0000 0000 0000 0000 0000
0000 0000 0111 1100 1000 0100 0000 0000 0000 0000

Addition of Floating Point Numbers

Step 3: do the addition:

0000 0001. 1001 1000 1011 0000 0000 0000 0000 0000
0000 0000 0111 1100 1000 0100 0000 0000 0000 0000

0000 0010 0001 0101 0011 0100 0000 0000 0000 0000

Addition of Floating Point Numbers

Step 4: Post normalize: (restore normalized condition)
and adjust exponent
0000 0001 0000 1010 1001 1010 0000 0000 0000 0000
x 2^1

So, final IEEE representation:
0 10001010 00001010100110100000000

Addition of Floating Point Numbers

1634.75 = 11001100010.1100
= $1.10011000101100 \times 2^{10}$

IEEE: $10 + 127 = 137_{10} = 10001001_2$ so,
0 10001001 10011000101100000000000

-1555.55 = 11000010011.10001100110011001100
= $1.100001001110001100110011001100 \times 2^{10}$

IEEE: $10 + 127 = 137_{10} = 10001001_2$ so,
1 10001001 10000100111000110011001

Addition of Floating Point Numbers

Step 1: Determine larger of two numbers by comparing the exponents

Number A: 10001001
Number B: 10001001
A – B = 00000000

Number A is same order of magnitude as Number B; no alignment necessary

Addition of Floating Point Numbers

Step 2: Represent numbers in same format of user's choosing. Choice here is two words (64 bits) with p=56.
Note: keep track of fact that this is x 2^{10}

0000 0001. 1001 1000 1011 0000 0000 0000 0000 0000
0000 0001. 1000 0100 1110 0011 0011 0011 0011 0000

Addition of Floating Point Numbers

Step 3: do the addition (in this case, subtraction):

0000 0001. 1001 1000 1011 0000 0000 0000 0000 0000
0000 0001. 1000 0100 1110 0011 0011 0011 0011 0000

0000 0000 0001 0011 1100 1100 1100 1100 1101 0000

Addition of Floating Point Numbers

Step 4: Post normalize: (restore normalized condition)
and adjust exponent

0000 0001 0011 1100 1100 1100 1100 1101 0000
x 2^{-4}

So, final IEEE representation:

0 10000101 00111100110011001100110