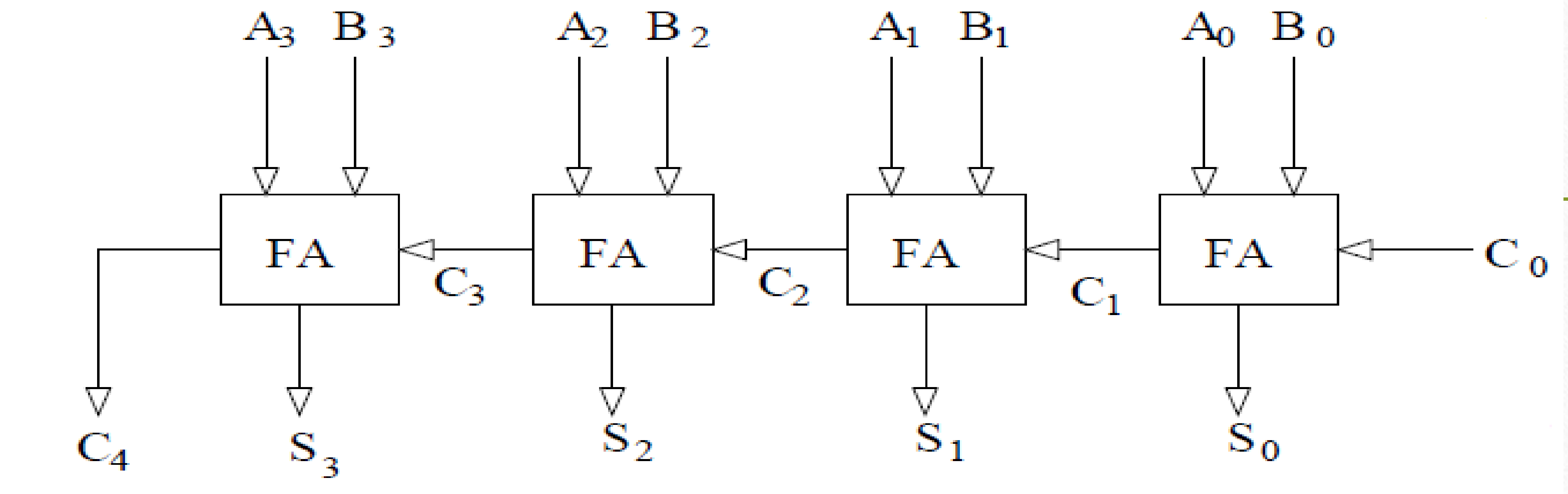
Data Representation and Manipulation

Part 2



A Faster Carry : Carry Look Ahead Adder

Faster Carry Or Look Ahead Adder:

- In a row of adders: try to generate what is the next carry into column on left
- All the bits of numbers A and B are known at beginning.
- CarryIn to each adder is unknown.

Faster Carry Or Look Ahead Adder:

- $c1 = (a0 \cdot c0) + (b0 \cdot c0) + (a0 \cdot b0)$
- $c2 = (a1 \cdot c1) + (b1 \cdot c1) + (a1 \cdot b1)$
- **$c2 = (a1 \cdot a0 \cdot b0) + (a1 \cdot a0 \cdot c0) \cdot (a1 \cdot b0 \cdot c0) + (b1 \cdot a0 \cdot b0) + (b1 \cdot a0 \cdot c0) + (b1 \cdot b0 \cdot c0) + (b1 \cdot a1)$**

can predict the next carry based on a, b values and c0.

$c2 = (a1 \cdot a0 \cdot b0) + (a1 \cdot a0 \cdot c0) \cdot (a1 \cdot b0 \cdot c0) +$
$(b1 \cdot a0 \cdot b0) + (b1 \cdot a0 \cdot c0) + (b1 \cdot b0 \cdot c0) + (b1 \cdot a1)$

- Therefore, additional gates, added into each adder.

- Next level, gets more complicated. Simple substitution into the same formula.
- Hardware quickly adds up and can get expensive for each bit
- Carry look ahead adder, much more efficient and some level of look ahead is added into each adding unit.
- **Fast carry using propagate and generate : Read about it in the text B.6**

Convert this Fractional number to IEEE Floating Point Representation

Start: 42.3125

42 = 101010

.3125x2 = 0.625 Apply the same algorithm from previous slides

.625x2 = 1.25

.25x2 = 0.5

.5x2 = 1

.3125 = .0101

→ 42.3125 = 101010.0101 = 1.010100101x2^5 **Need to Normalize : Only one leading 1**

Sign bit: 0 (pos)

Exponent – 127 = 5 → Exponent = 132 = 10000100

IEEE754: 0 10000100 01010010100000000000000

Final 32 Bits Representation

Floating-Point Addition

Only 1 leading digit before decimal

- Decimal example: $9.54 \times 10^2 + 6.83 \times 10^1$

  (assume we can only store two digits to right of decimal point)

  1. Match exponents: $9.54 \times 10^2 + .683 \times 10^2$
  2. Add significands, with sign: 10.223×10^2
  3. Normalize: 1.0223×10^3
  4. Check for exponent overflow/underflow
  5. Round: 1.02×10^3
  6. May have to normalize again

- Same idea works for binary

A: 0 10000100 0101001010...
B: 1 10000011 0001001010...

A's exponent: 5
B's exponent: 4

A's mantissa: 1.0101001010...
B's mantissa: 1.0001001010...

Must shift B's mantissa, exponent by 1 so they become 5

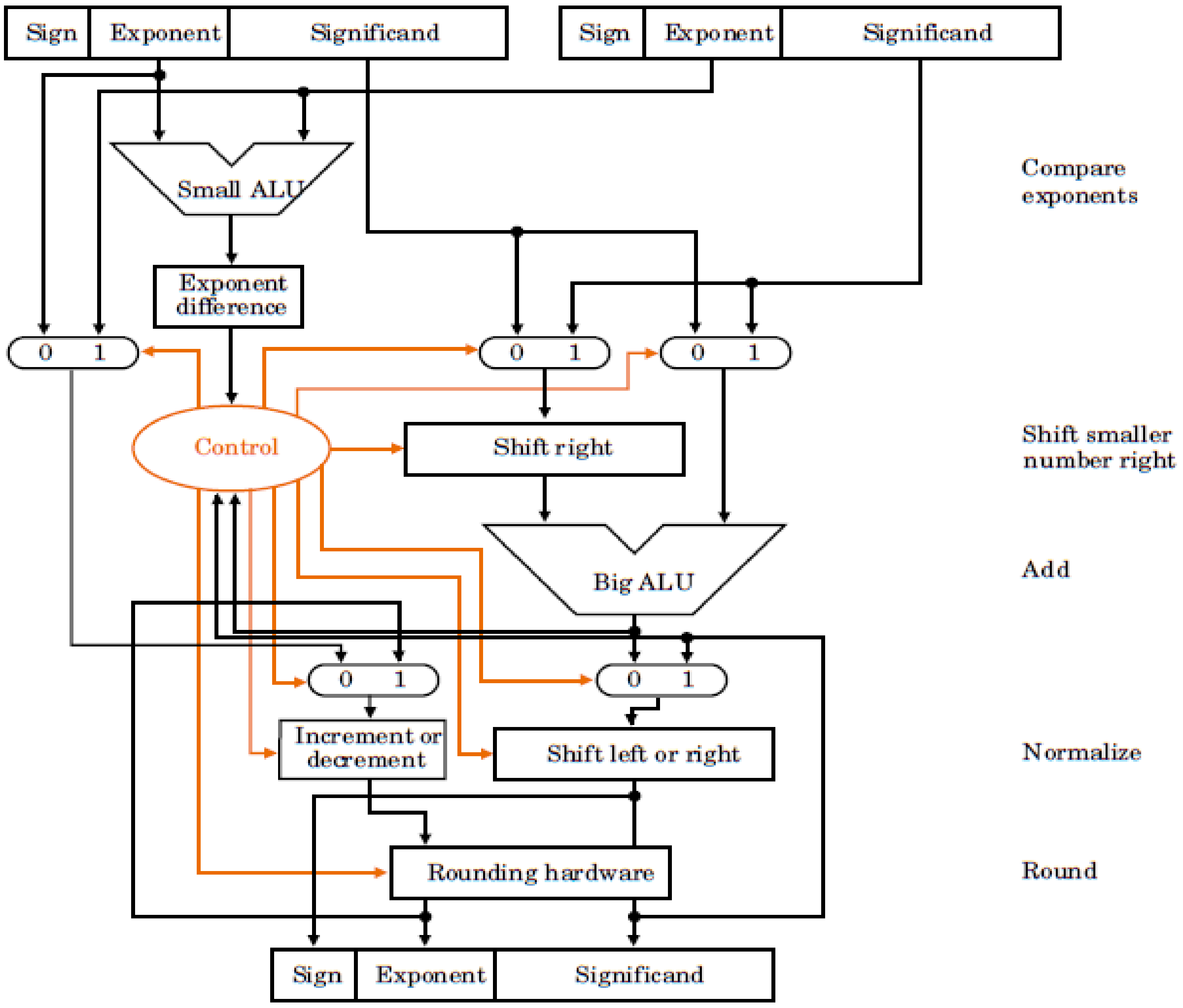
– 0.1000100101

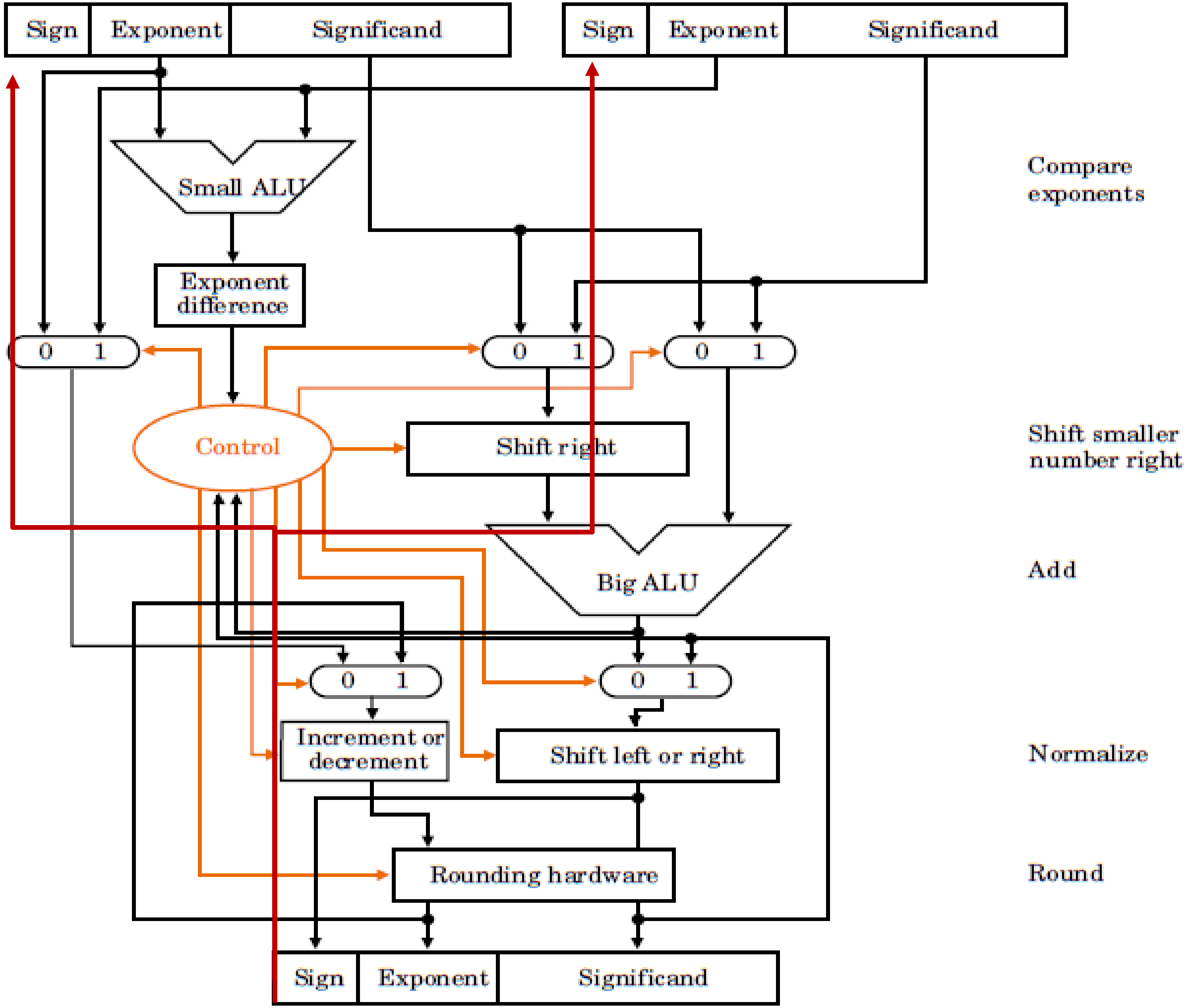
Because we are adding two numbers of different signs, we use signed magnitude addition: subtract the smaller mantissa from the larger mantissa, and keep the sign of the larger

$$
\begin{array}{r}
1.0101001010\ldots \times 2^5 \\
-\ 0.10001001010\ldots \times 2^5 \\
\hline
(+)\ 0.11001001010\ldots \times 2^5
\end{array}
$$

Performing Subtration

Normalize: 1.1001001010... x2^4
Sign bit = 0
Exponent -127 = 4 -> exponent = 131 = 10000011
Answer: 0 10000011 10010010100000000000000 = 25.15625


Sign
Exponent
Significand
Sign
Exponent
Significand
Small ALU
Exponent difference
0 1
0 1
0 1
Control
Shift right
Big ALU
0 1
0 1
Increment or decrement
Shift left or right
Rounding hardware
Sign
Exponent
Significand
Compare exponents
Shift smaller number right
Add
Normalize
Round



Deriving the Sign Bit:
If opposite signs

Additional logic
to determine which is the greater
Number

In this case (ALU)
performs a subtract

Convert 0.375 to binary

- A) 0.11
- B)0.0101
- C) 0.01101
- D) 1.11
- E) 0.011

Floating Point

- Single Precision: Exponent bits in 8 precision :Bias of 127
- Double Precision: 11 bits Exponent : Bias
- Based on *COAD Text (H&P)* Representing 0:
  - 0000 0000 RESERVED for Zero
  - 1111 1111 RESERVED for exceptions beyond scope of normal floating point numbers. Such as De-normalized numbers

Therefore lowest exponent is -126, highest positive exponent +127

Convert this binary numb to binary fp notation according to IEEE754 Standard

- [1 10000001 01010100000000000….]
- A) + 1.010101000… x 2^{129}
- B) - 1.010101000… x 2^{2}
- C) + 1.010101000… x 2^{2}
- D) - 1.010101000… x 2^{-2}
- E) + 1.010101000… x 2^{257}

Floating-Point Multiplication

- Decimal example: $(9.54 \times 10^2) \times (6.83 \times 10^1)$
  (assume we can only store two digits to right of decimal point)

  1. Add exponents: $2 + 1 = 3$
     (Note: exponents stored in biased notation)
  2. Multiply significands: $9.54 \times 6.83 = 65.1582$
  3. Unnormalized result: 65.1582×10^3
  4. Normalize: 6.51582×10^4
  5. Check for overflow/underflow
  6. Round: 6.52×10^4
     (May need to renormalize)
  7. Set sign

When examining the algorithm for FP Multiplication, which step(s) can move closer to the beginning.

Floating-Point Multiplication

- Decimal example: $(9.54 \times 10^2) \times (6.83 \times 10^1)$
  (assume we can only store two digits to right of decimal point)

1. Add exponents: $2 + 1 = 3$
   (Note: exponents stored in biased notation)
2. Multiply significands: $9.54 \times 6.83 = 65.1582$
3. Unnormalized result: 65.1582×10^3
4. Normalize: 6.51582×10^4
5. Check for overflow/underflow
6. Round: 6.52×10^4
   (May need to renormalize)
7. Set sign

A) Normalize result

B) Round Result

C) Set sign

D) Check for overflow/underflow

E) Add exponents

Example : Multiplication of two floating Point Numbers

A: 0 10000100 0101001010...
B: 1 10000011 0001001010...

A = 42.3125 and B = –17.15625.

A's exponent: 5
B's exponent: 4

A+B'S unbiased exponent: 9
-> Exponent – 127 = 9
-> Exponent = 136 = 10001000

Multiply Mantissas:

$$
\begin{array}{r}
1.0101001010... \\
\underline{0.0001001010...} \\
1.01101010111101100 10...
\end{array}
$$

Sign = (A's sign + B's sign) %2 = (0 + 1) % 2 = 1
= 1 10001000 01101010111101100100000 = -725.9238281

Multiplying Two Numbers

$$
\begin{array}{cccccccc l}
 & & & & 1 & 1 & 0 & 1 & \text{Multiplicand} \\
 & & & & 1 & 0 & 1 & 1 & \text{Multiplier} \\
\cline{5-8}
 & & & & 1 & 1 & 0 & 1 & \\
 & & & 1 & 1 & 0 & 1 & & \\
 & & 0 & 0 & 0 & 0 & & & \\
 & 1 & 1 & 0 & 1 & & & & \\
\hline
1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & \text{Product}
\end{array}
$$

Multiplying Two Numbers : With a Decimal in the number

$$
\begin{array}{rrrrrrrrl}
 & & & & 1. & 1 & 0 & 1 & \text{Multiplicand} \\
 & & & & 1 & 0 & 1. & 1 & \text{Multiplier} \\
\hline
 & & & & 1 & 1 & 0 & 1 & \\
 & & & 1 & 1 & 0 & 1 & & \\
 & & 0 & 0 & 0 & 0 & & & \\
 & 1 & 1 & 0 & 1 & & & & \\
\hline
1 & 0 & 0 & 0. & 1 & 1 & 1 & 1 & \text{Product}
\end{array}
$$

Multiplying Two Numbers : With a Decimal in the number

$$
\begin{array}{cccccccc l}
 & & & & .1 & 1 & 0 & 1 & \text{Multiplicand} \\
 & & & & .1 & 0 & 1 & 1 & \text{Multiplier} \\
\cline{5-8}
 & & & & 1 & 1 & 0 & 1 & \\
 & & & 1 & 1 & 0 & 1 & & \\
 & & 0 & 0 & 0 & 0 & & & \\
 & 1 & 1 & 0 & 1 & & & & \\
\hline
1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & \text{Product}
\end{array}
$$

Multiplying Two Numbers : With a Decimal in the number

$$
\begin{array}{rcccccccccl}
 & & & & & .1 & 1 & 0 & 1 & & \text{Multiplicand} \\
 & & & & & .1 & 0 & 1 & 1 & & \text{Multiplier} \\
\cline{6-9}
 & & & & & 1 & 1 & 0 & 1 & & \\
 & & & & 1 & 1 & 0 & 1 & & & \\
 & & & 0 & 0 & 0 & 0 & & & & \\
 & & 1 & 1 & 0 & 1 & & & & & \\
\cline{2-9}
 & .1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & & \text{Product}
\end{array}
$$

Example : Multiplication of two floating Point Numbers

A: 0 10000100 0101001010...
B: 1 10000011 0001001010...

A = 42.3125 and B = –17.15625.

A's exponent: 5
B's exponent: 4

A+B'S unbiased exponent: 9
-> Exponent – 127 = 9
-> Exponent = 136 = 10001000

Multiply Mantissas:

$$\begin{array}{r} 1.0101001010... \\ \underline{0.0001001010...} \\ 1.01101010111110110010... \end{array}$$

Sign = (A's sign + B's sign) %2 = (0 + 1) % 2 = 1
= 1 10001000 01101010111101100100000 = -725.9238281

Accuracy in Floating-Point Multiplication

- When multiplying two floating-point numbers, the significands are multiplied together
- If the significands have n bits of precision each, the result can have $2n$ bits of precision
- How many bits do we need to keep during the computation?
- Our multiplication examples will have $n = 3$

- Example:

$$
\begin{array}{rcccccccc}
 & & 1 & . & 1 & 1 & & & \times\ 2^2 \\
\times & & 1 & . & 1 & 1 & & & \times\ 2^1 \\
\hline
 & 1 & 1 & . & 0 & 0 & 0 & 1 & \times\ 2^3
\end{array}
$$

- In above example, only top 3 bits are needed for final result of 1.10×2^4
- Example: Do we need circled (fourth) bit?

$$
\begin{array}{rcccccccc}
 & & 1 & . & 1 & 0 & & & \times\ 2^2 \\
\times & & 1 & . & 1 & 0 & & & \times\ 2^1 \\
\hline
 & 1 & 0 & . & 0 & \boxed{1} & 0 & 0 & \times\ 2^3
\end{array}
$$

- With three bits, 10.0×2^3 is normalized to 1.00×2^4, which is incorrectly rounded
- With four bits, 10.01×2^3 is normalized to 1.001×2^4, and correctly rounded up to 1.01×2^4

- Example: Do we need circled (fourth,fifth) bits?

$$
\begin{array}{ccccccccc}
 & & 1 & . & 0 & 1 & & & \times\ 2^2 \\
\times & & 1 & . & 1 & 0 & & & \times\ 2^1 \\
\hline
 & 0 & 1 & . & 1 & \boxed{1} & \boxed{1} & 0 & \times\ 2^3
\end{array}
$$

- With three bits, 01.1×2^3 is normalized to 1.10×2^3, which is incorrectly rounded
- With four bits, 01.11×2^3 is normalized to 1.110×2^3, and rounded to 1.11×2^3, which is incorrectly rounded
- With five bits, 01.111×2^3 is normalized to 1.111×2^3, rounded to 10.0×2^3, and normalized again to 1.00×2^4, which is correctly rounded

Floating-Point Architectural Issues

- To maintain n bits of accuracy after an operation, preserve $n+2$ bits during the computation (the two extra bits are sometimes called *guard* and *round*)
- Separate floating-point registers?
- Separate floating-point coprocessors?
- Rounding or truncating?
- What to do about overflow (same issue as for integer arithmetic)?

Integer Multiplication (not FP)

```
        1 1 0 1   Multiplicand
        1 0 1 1   Multiplier
      ---------
        1 1 0 1
      1 1 0 1
    0 0 0 0
  1 1 0 1
---------------
1 0 0 0 1 1 1 1   Product
```

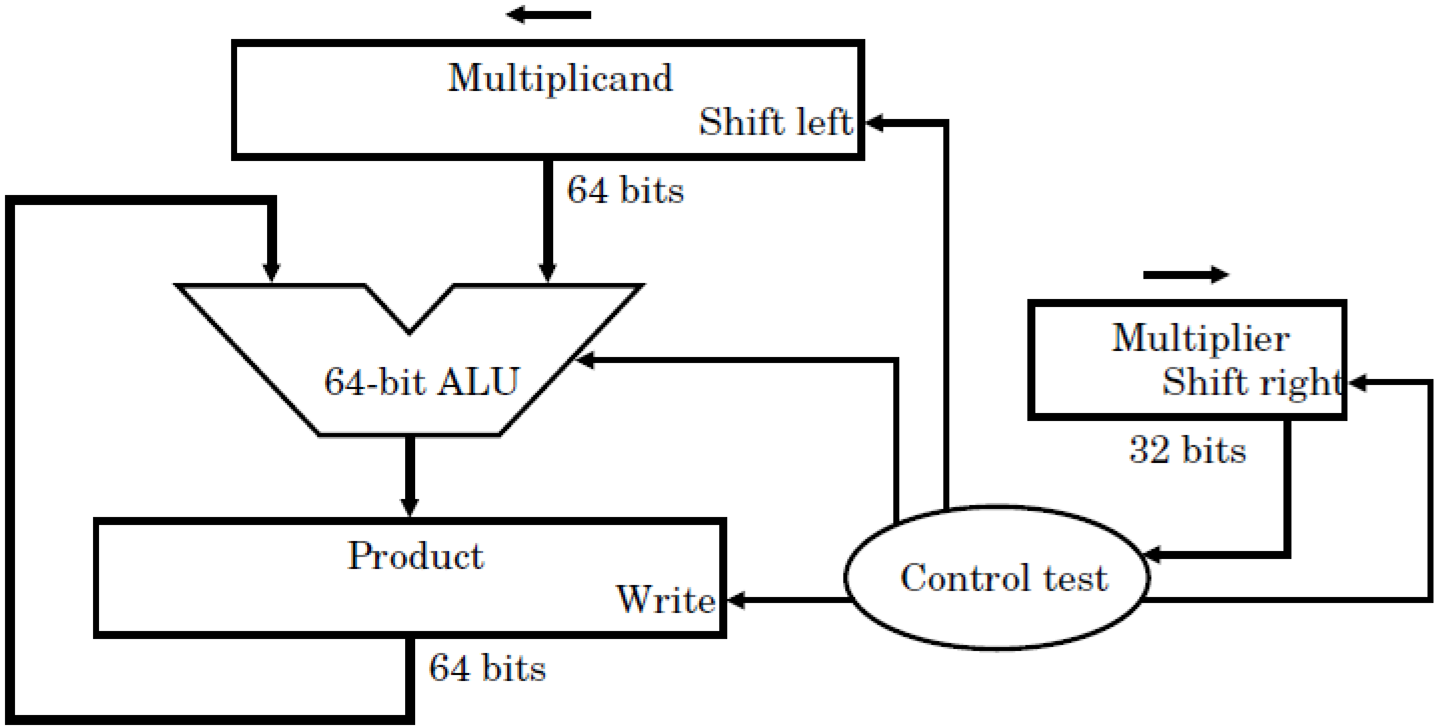
*Understand the components
Of hardware that are necessary
For integer Multiplication

*Understand the improvements
Achieved in the third version
Of multiplication hardware

What is result of : 0011 x 0011

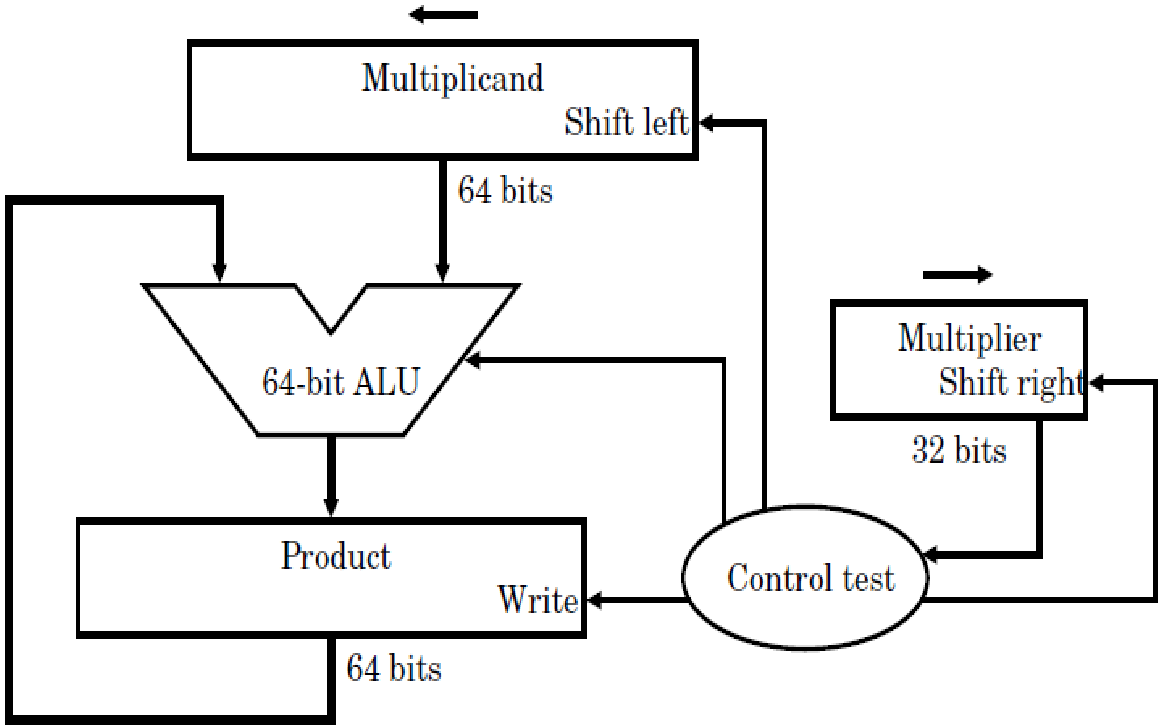
- A)1011
- B)1001
- C)0111
- D) 1111
- E) none

Multiplication Hardware, First Version

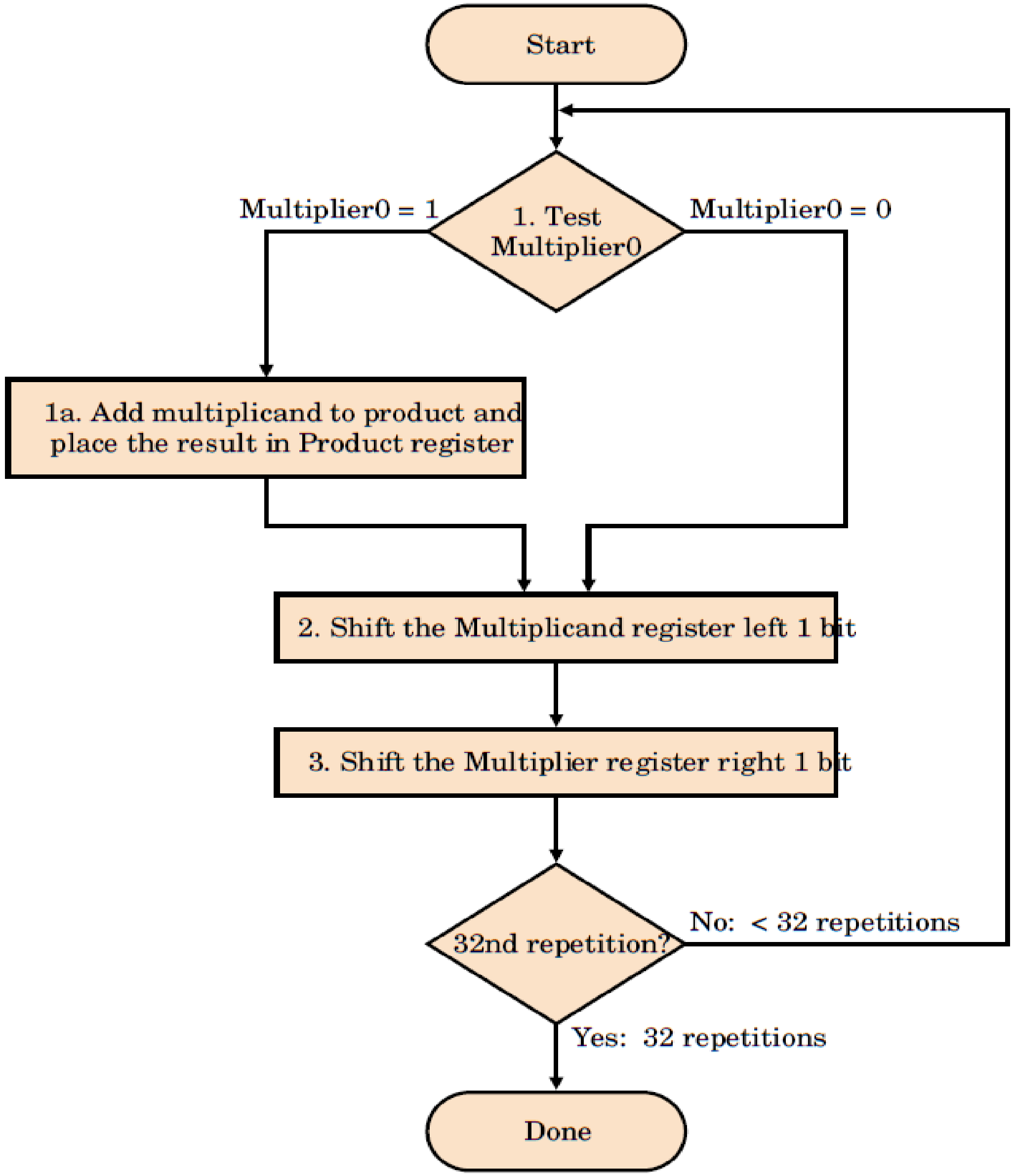


We do have a 64 bit product register… Low and High

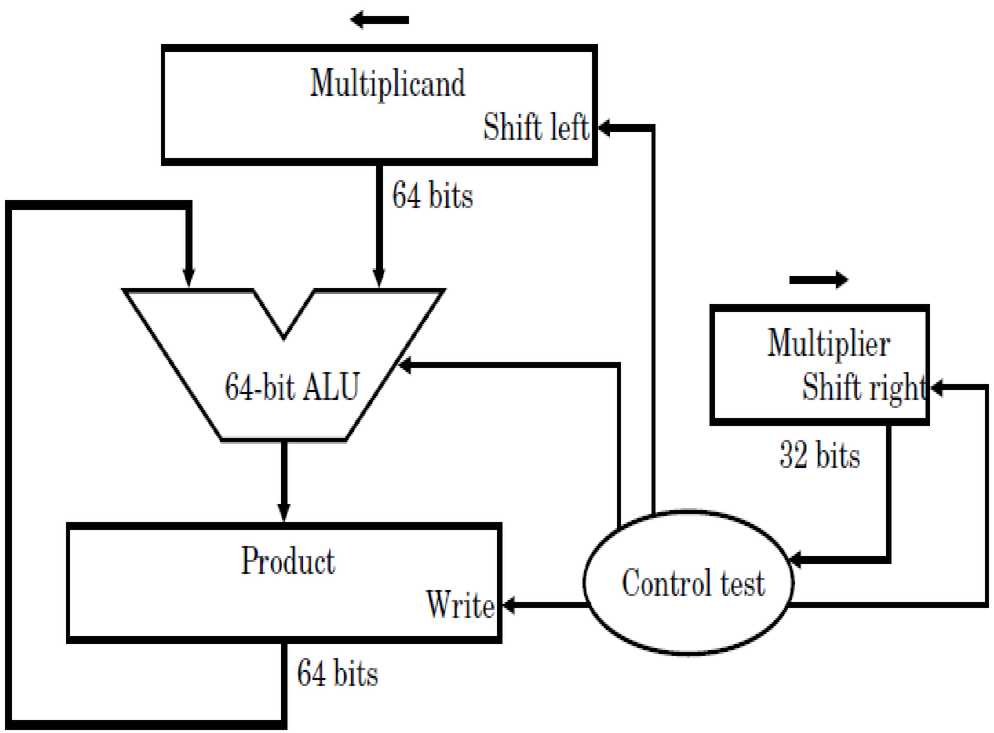
Multiplication Hardware, First Version




```
       1 1 0 1   Multiplicand
       1 0 1 1   Multiplier
     ---------
       1 1 0 1
     1 1 0 1
   0 0 0 0
 1 1 0 1
---------------
1 0 0 0 1 1 1 1  Product
```



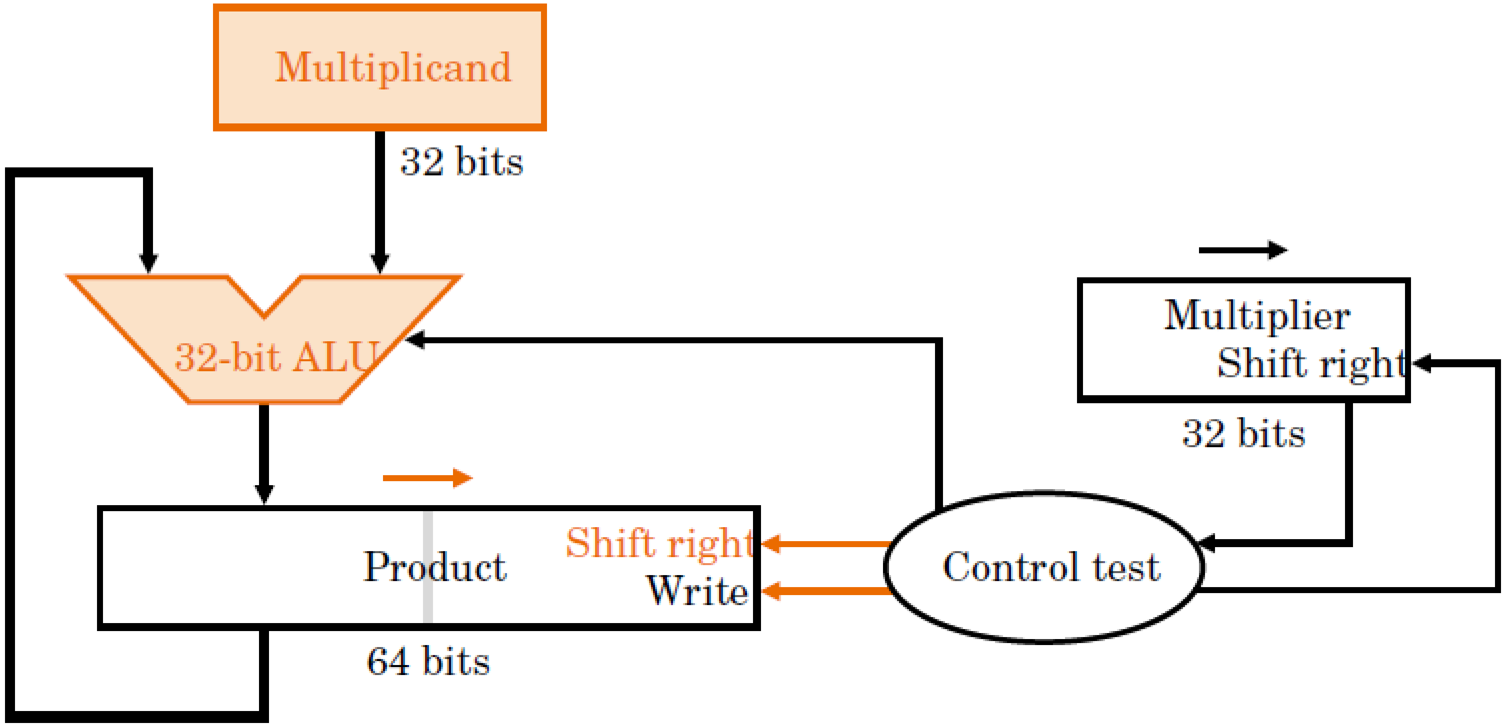
Multiplication Hardware, First Version



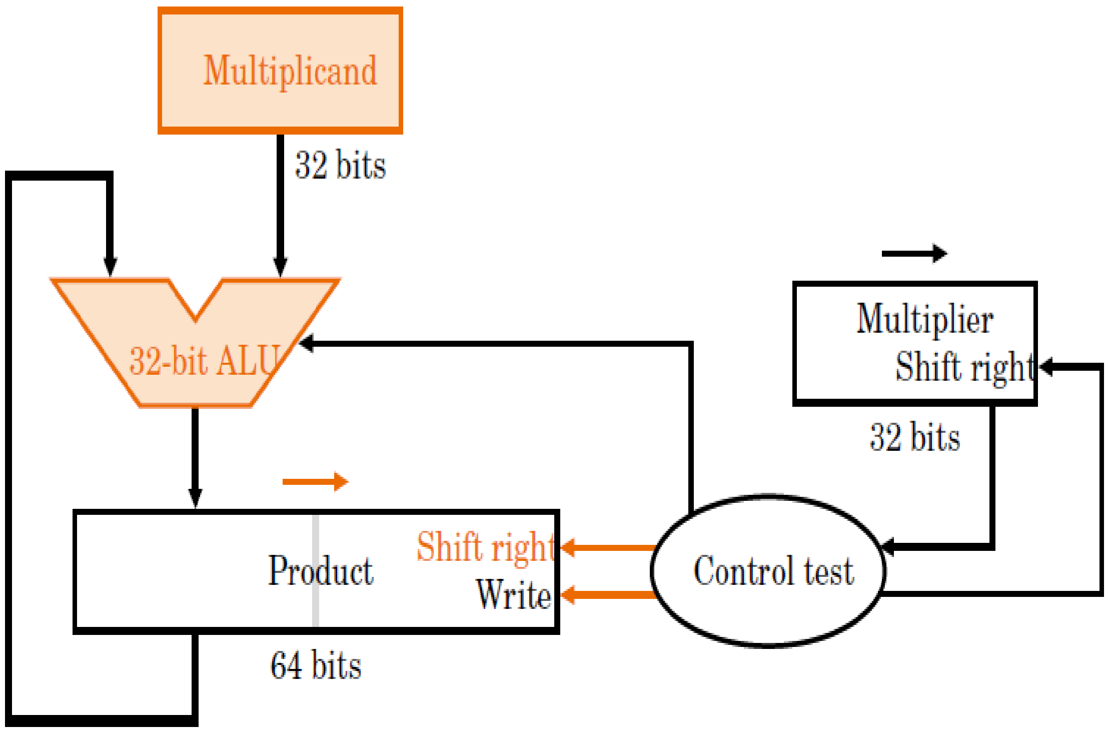
Iteration	Step	Multiplier	Multiplicand	Product
0	Initial Values	1011	0000 1101	0000 0000
1	Add mpcd to prod			0000 1101
	Shift left mpcd		0001 1010	
	Shift right mplr	0101		
2	Add mpcd to prod			0010 0111
	Shift left mpcd		0011 0100	
	Shift right mplr	0010		
3	No operation			
	Shift left mpcd		0110 1000	
	Shift right mplr	0001		
4	Add mpcd to prod			1000 1111
	Shift left mpcd		1101 0000	
	Shift right mplr	0000		

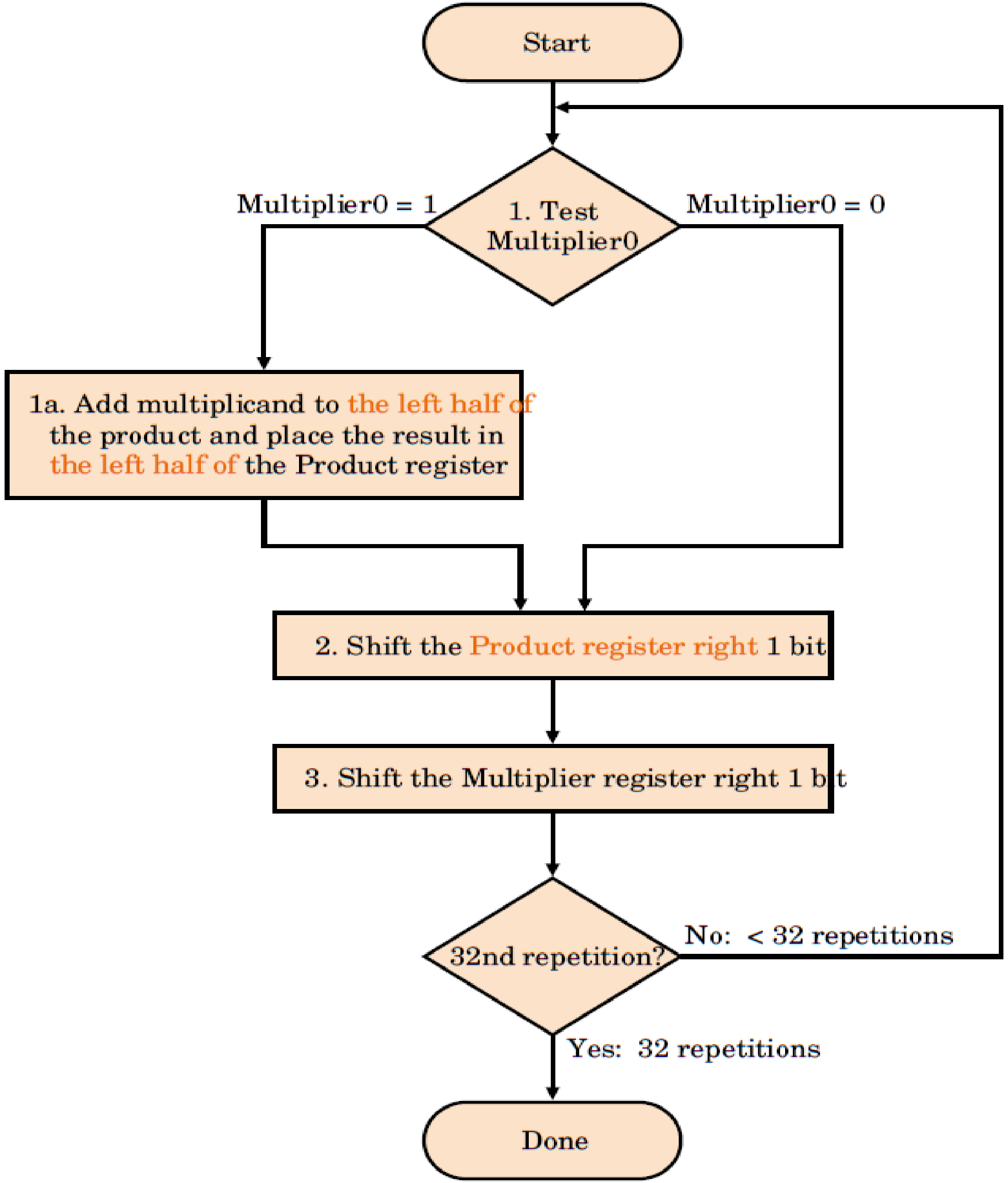
```
        1 1 0 1   Multiplicand
        1 0 1 1   Multiplier
        -------
        1 1 0 1
      1 1 0 1
    0 0 0 0
  1 1 0 1
---------------
1 0 0 0 1 1 1 1   Product
```

Multiplication Hardware, Second Version

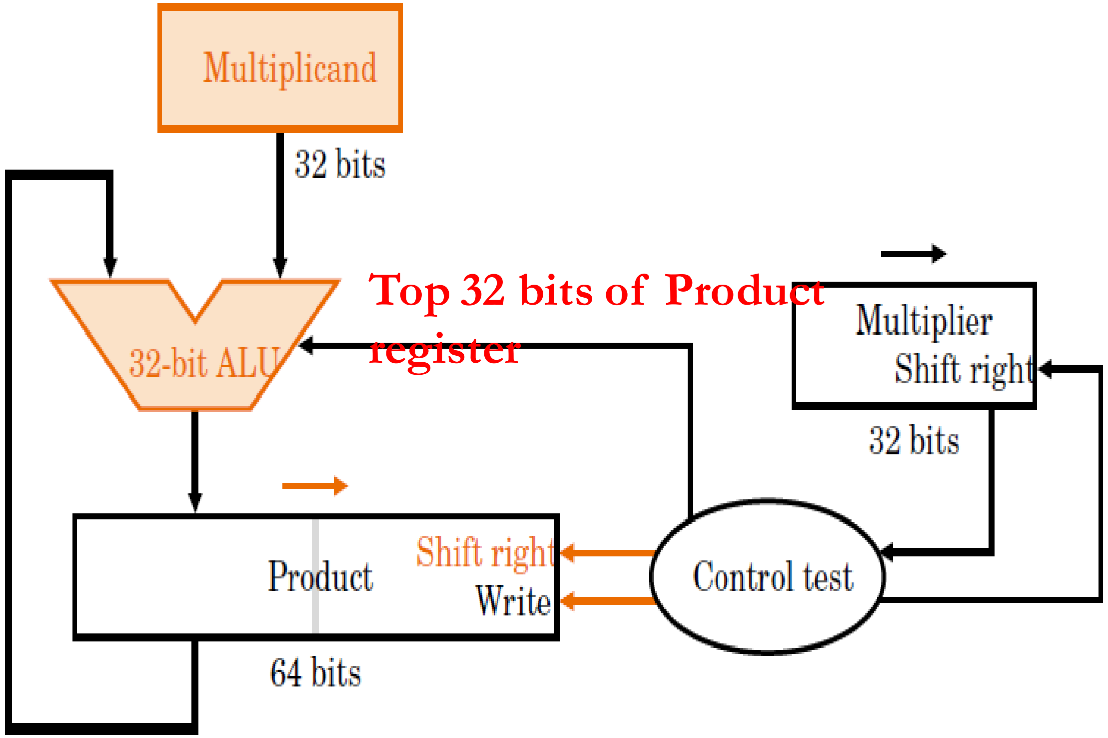


Multiplication Hardware, Second Version





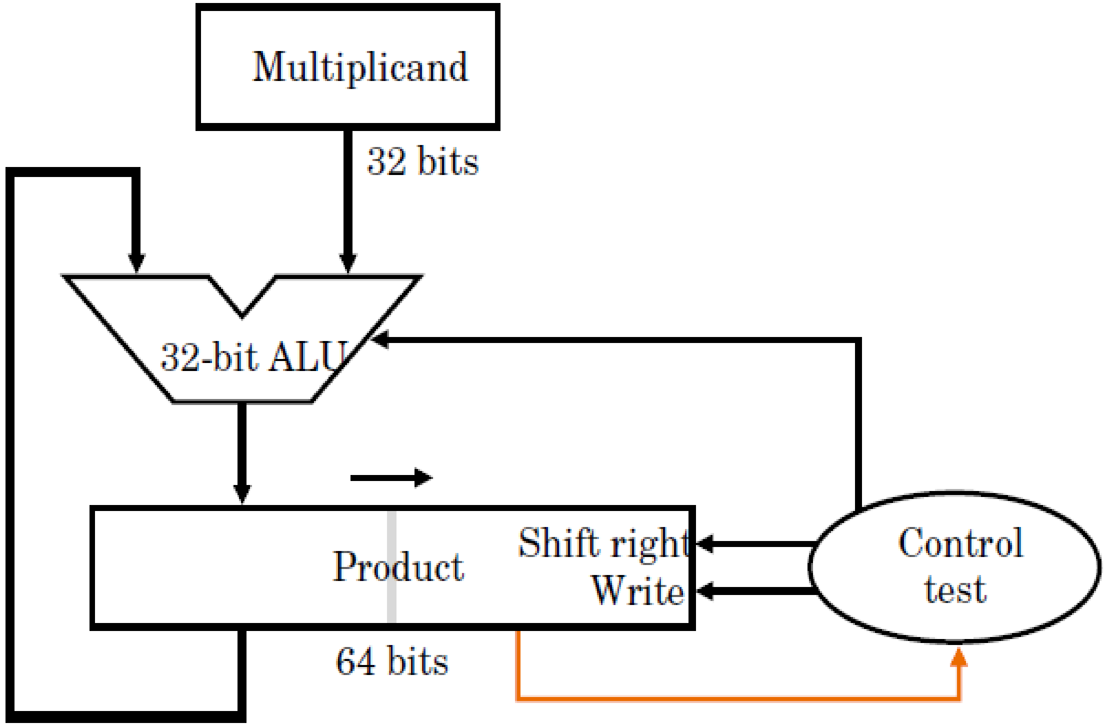
Multiplication Hardware, Second Version




```
    1 1 0 1   Multiplicand
    1 0 1 1   Multiplier
    -------
    1 1 0 1
  1 1 0 1
0 0 0 0
1 1 0 1
---------------
1 0 0 0 1 1 1 1   Product
```

Iteration	Step	Multiplier	Multiplicand	Product
0	Initial Values	1011	1101	0000 0000
1	Add mpcd to prod			1101 0000
	Shift right prod		1101	0110 1000
	Shift right mplr	0101		
2	Add mpcd to prod			$\boxed{1}$0011 1000
	Shift right prod		1101	1001 1100
	Shift right mplr	0010		
3	No operation			
	Shift right prod		1101	0100 1110
	Shift right mplr	0001		
4	Add mpcd to prod			$\boxed{1}$0001 1110
	Shift right prod		1101	1000 1111
	Shift right mplr	0000		

Multiplication Hardware, Third Version



Multiplier = 1011, Multiplicand = 1101

Iteration	Step	Multiplicand	Product
0	Initial Values	1101	0000 1011
1	Add mpcd to prod		1101 1011
	Shift right prod	1101	0110 1101
2	Add mpcd to prod		1 0011 1101
	Shift right prod	1101	1001 1110
3	No operation		
	Shift right prod	1101	0100 1111
4	Add mpcd to prod		1 0001 1111
	Shift right prod	1101	1000 1111

```
      1 1 0 1   Multiplicand
      1 0 1 1   Multiplier
      -------
      1 1 0 1
    1 1 0 1
  0 0 0 0
1 1 0 1
---------------
1 0 0 0 1 1 1 1   Product
```

Save One Register: Multiplier

Use 64 bits of Product Register Efficiently

Multiply Signed numbers:

- Best choice to Multiply two numbers:
  - Keep track of signs
  - Multiply as two positive numbers (convert to positive if negative)
  - Change answer to negative if needed. (this work for 64 bit results)

  - Third hardware version can allow 2s complement negative numbers
  - Where correct answer is in lower 32 bits

Multiplication Faster:

- Can add hardware to make multiplication faster:
- Instead of 1-32 bit adder (ALU)
  - Use 32 additions into a stack of adders: output of one addition feeds directly into next adder.
  - With the appropriate bit of multiplier ANDed with Multiplicand
    - If 0 nothing gets added
    - If 1 multiplicand gets added to summation.

What is NOT considered a major improvement from the first version to third version

- A) First version 64 bit register for product →came down to 32bit register for product in third version
- B) Removed need for separate register for Multiplier
- C) Reduced size of ALU from 64bits to 32bits.
- D) Used left half of product register more effectively
- E) Do not need 64 bit Multiplicand anymore. Reduced this to 32bits