Multiplication

- More complicated than addition
  - accomplished via shifting and addition
- More time and more area
- Let's look at 3 versions based on grade school algorithm

```
  01010010   (multiplicand)
x01101101    (multiplier)
```

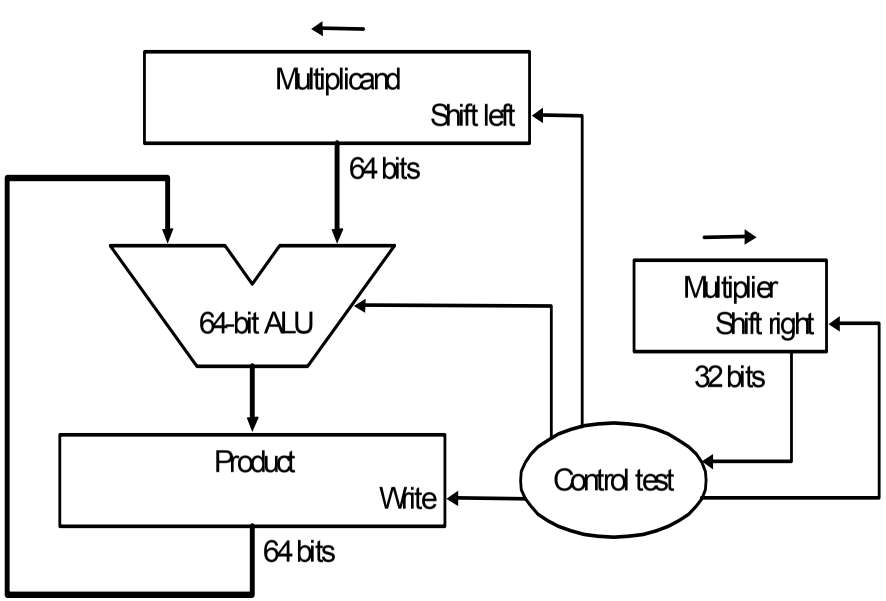
- Negative numbers: convert and multiply
- Use other better techniques like Booth's encoding

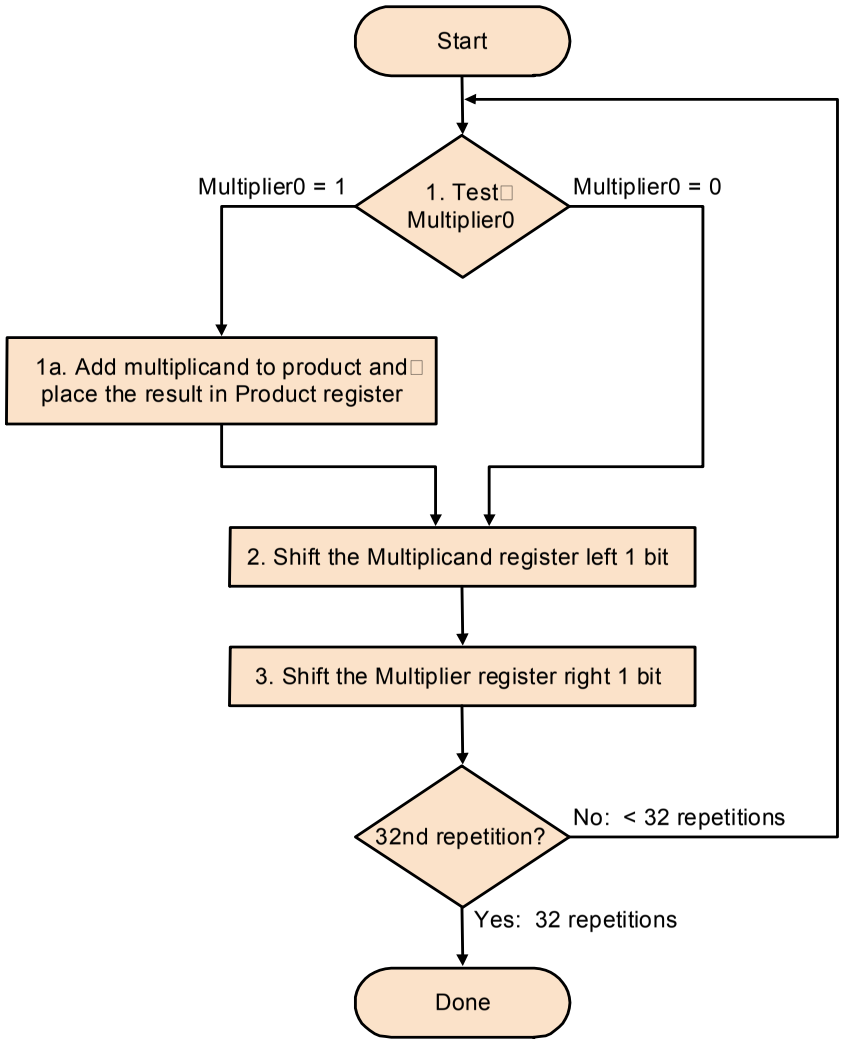
Multiplication

```
  01010010  (multiplicand)
 x01101101  (multiplier)
 ----------
  00000000
  01010010   x1
 ----------
   01010010
  000000000   x0
 ----------
    00101001 0
  0101001000   x1
 ----------
    01100110 10
 01010010000   x1
 ----------
    10000101 010
 00000000000   x0
 ----------
    01000010 1010
 01010010000   x1
 ----------
    01110011 01010
 01010010000   x1
 ----------
    10001011 101010
 00000000000   x0
 ----------
 0100010111 01010
```

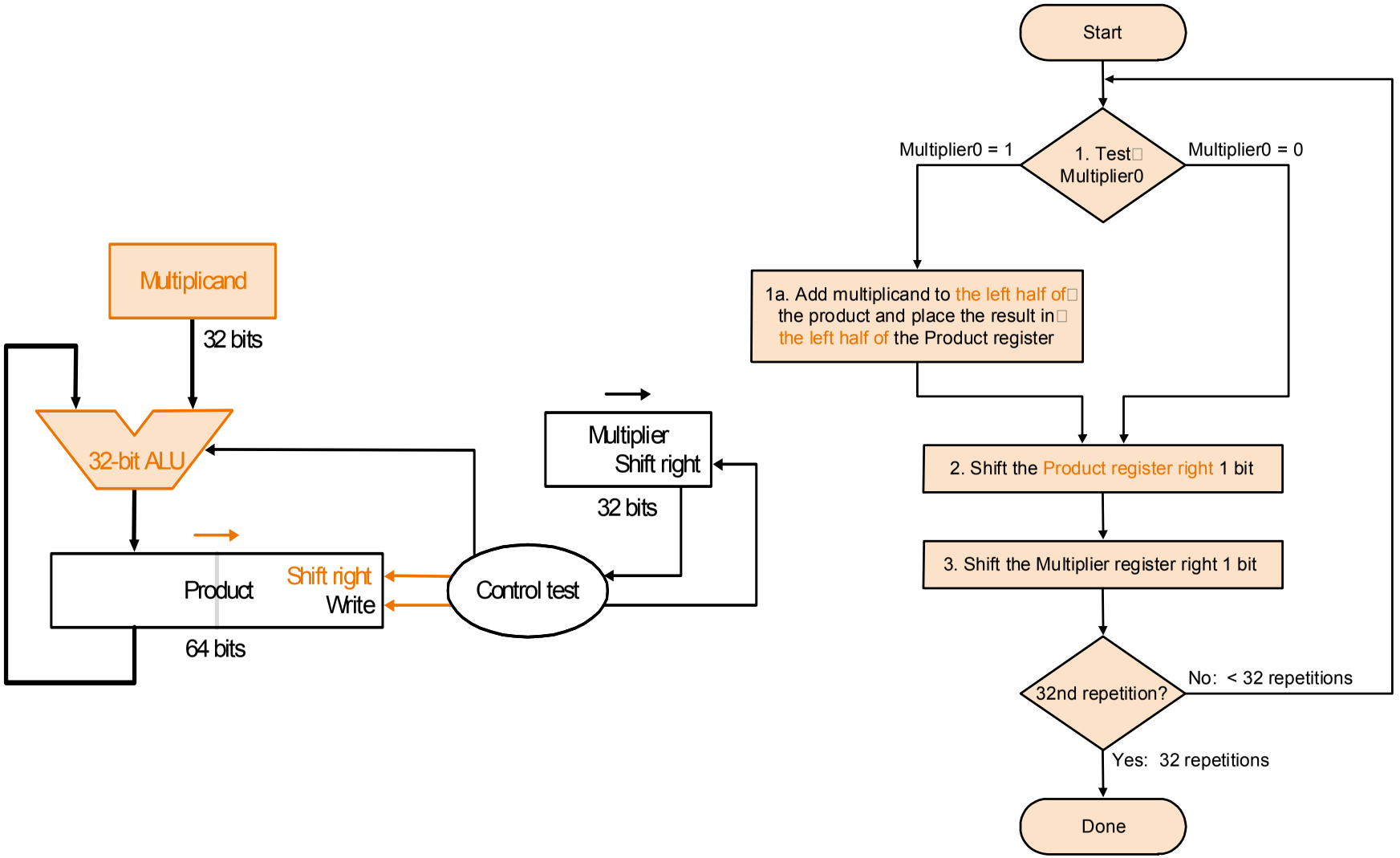
```
  01010010  (multiplicand)
 x01101101  (multiplier)
 ----------
  00000000
  01010010   x1
 ----------
   01010010
  000000000   x0
 ----------
    00101001 0
  0101001000   x1
 ----------
    01100110 10
 01010010000   x1
 ----------
    10000101 010
 00000000000   x0
 ----------
    01000010 1010
 01010010000   x1
 ----------
    01110011 01010
 01010010000   x1
 ----------
    10001011 101010
 00000000000   x0
 ----------
 0100010111 01010
```

Multiplication: Implementation

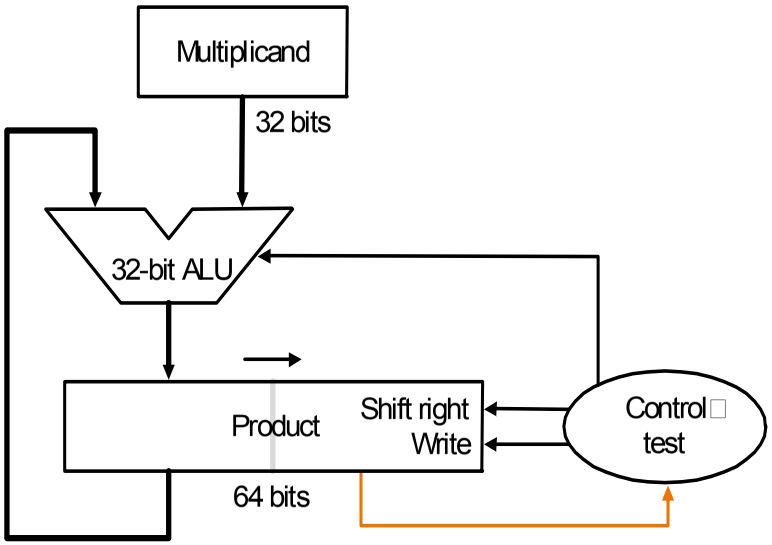


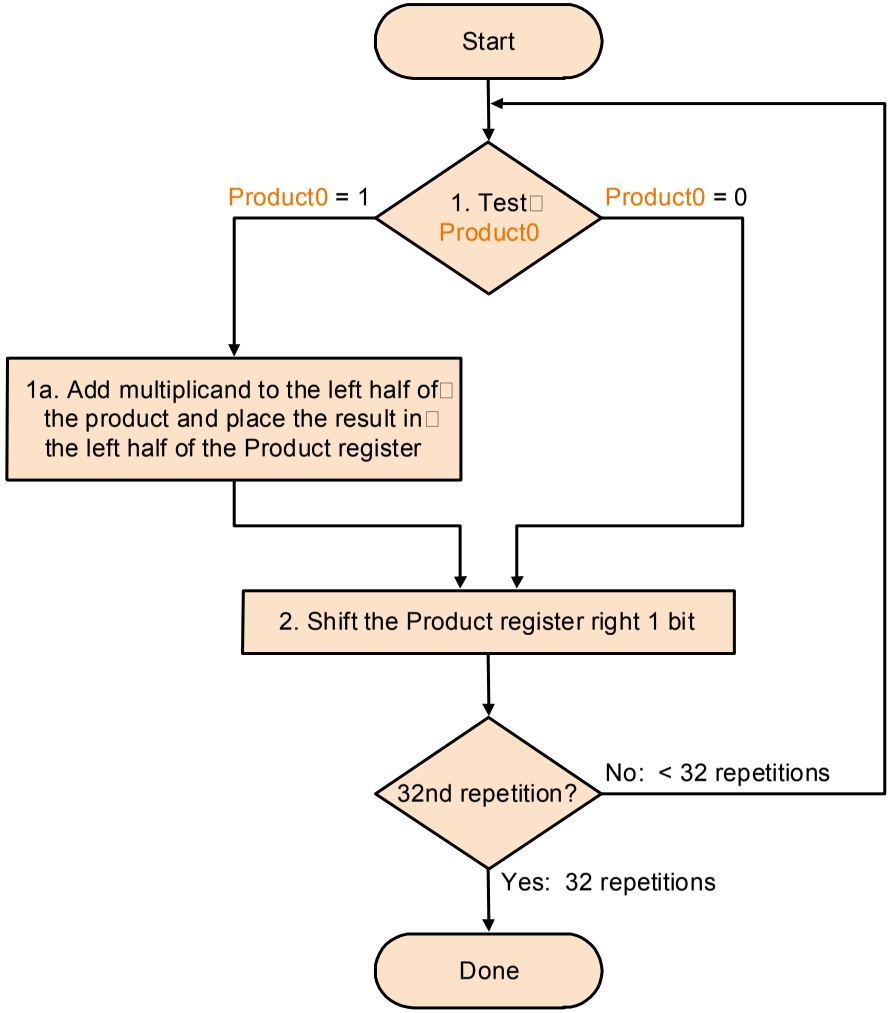


Second Version



Final Version





Multiplication Example

Itera-tion	multi-plicand	Orignal algorithm	
		Step	Product
0	0010	Initial values	0000 0110
1	0010	1:0 ⇒ no operation	0000 0110
	0010	2: Shift right Product	0000 0011
2	0010	1a:1⇒ prod = Prod + Mcand	0010 0011
	0010	2: Shift right Product	0001 0001
3	0010	1a:1⇒ prod = Prod + Mcand	0011 0001
	0010	2: Shift right Product	0001 1000
4	0010	1:0 ⇒ no operation	0001 1000
	0010	2: Shift right Product	0000 1100

Signed Multiplication

- Let Multiplier be Q[n-1:0], multiplicand be M[n-1:0]
- Let F = 0 (shift flag)
- Let result A[n-1:0] = 0….00
- For n-1 steps do
  - A[n-1:0] = A[n-1:0] + M[n-1:0] x Q[0] /* add partial product */
  - F<= F .or. (M[n-1] .and. Q[0]) /* determine shift bit */
  - Shift A and Q with F, i.e.,
  - A[n-2:0] = A[n-1:1]; A[n-1]=F; Q[n-1]=A[0]; Q[n-2:0]=Q[n-1:1]
- Do the correction step
  - A[n-1:0] = A[n-1:0] - M[n-1:0] x Q[0] /* subtract partial product */
  - Shift A and Q while retaining A[n-1]
  - This works in all cases excepts when both operands are 10..00

Booth's Encoding

- Numbers can be represented using three symbols, 1, 0, and -1
- Let us consider -1 in 8 bits
  - One representation is 1 1 1 1 1 1 1 1
  - Another possible one 0 0 0 0 0 0 0 -1
- Another example +14
  - One representation is 0 0 0 0 1 1 1 0
  - Another possible one 0 0 0 1 0 0 -1 0
- We do not explicitly store the sequence
- Look for transition from previous bit to next bit
  - 0 to 0 is 0; 0 to 1 is -1; 1 to 1 is 0; and 1 to 0 is 1
- Multiplication by 1, 0, and -1 can be easily done
- Add all partial results to get the final answer

Using Booth's Encoding for Multiplication

- Convert a binary string in Booth's encoded string
- Multiply by two bits at a time
- For n bit by n-bit multiplication, n/2 partial product
- Partial products are signed and obtained by multiplying the multiplicand by 0, +1, -1, +2, and -2 (all achieved by shift)
- Add partial products to obtain the final result
- Example, multiply 0111 (+7) by 1010 (-6)
- Booths encoding of 1010 is -1 +1 -1 0
- With 2-bit groupings, multiplication needs to be carried by -1 and -2
-

```
1 1 1 1 0 0 1 0    (multiplication by -2)
1 1 1 0 0 1 0 0    (multiplication by -1 and shift by 2 positions)
```

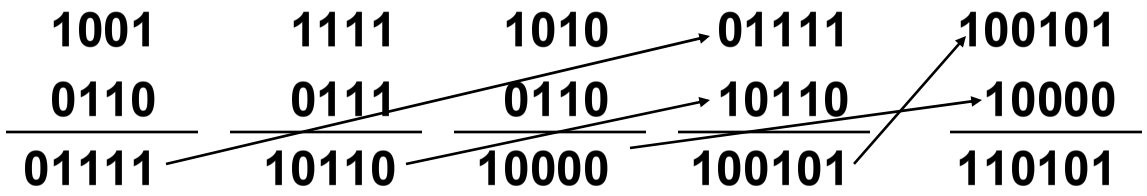
- Add the two partial products to get 11010110 (-42) as result

Booth's algorithm (Neg. multiplier)

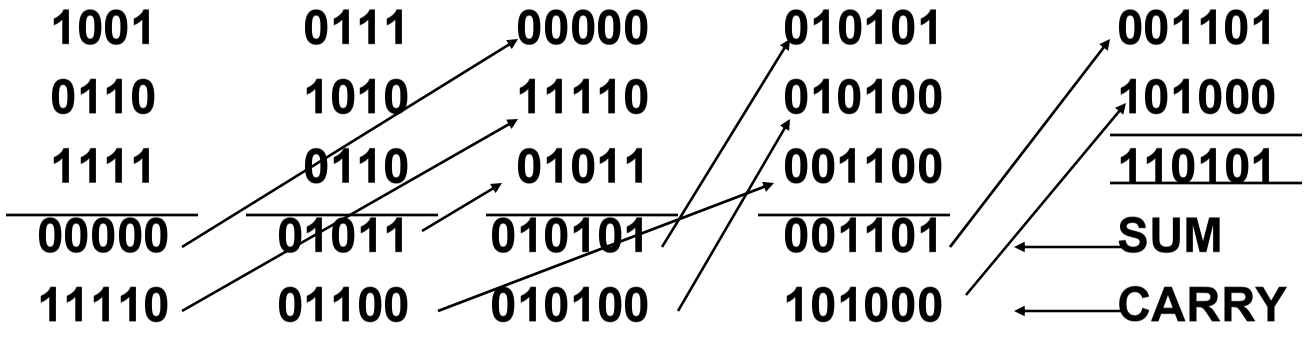
Itera-tion	multi-plicand	Booth's algorithm	
		Step	Product
0	0010	Initial values	0000 1101 0
1	0010	1c: 10⇒ prod = Prod - Mcand	1110 1101 0
	0010	2: Shift right Product	1111 0110 1
2	0010	1b: 01⇒ prod = Prod + Mcand	0001 0110 1
	0010	2: Shift right Product	0000 1011 0
3	0010	1c: 10⇒ prod = Prod - Mcand	1110 1011 0
	0010	2: Shift right Product	1111 0101 1
4	0010	1d: 11 ⇒ no operation	1111 0101 1
	0010	2: Shift right Product	1111 1010 1

Carry-Save Addition

- Consider adding six set of numbers (4 bits each in the example)
- The numbers are 1001, 0110, 1111, 0111, 1010, 0110 (all positive)
- One way is to add them pair wise, getting three results, and then adding them again



- Other method is add them three at a time by saving carry



Carry-Save Addition for Multiplication

- n-bit carry-save adder take 1FA time for any n
- For n x n bit multiplication, n or n/2 (for 2 bit at time Booth's encoding) partial products can be generated
- For n partial products n/3 n-bit carry save adders can be used
- This yields 2n/3 partial results
- Repeat this operation until only two partial results are remaining
- Add them using an appropriate size adder to obtain 2n bit result
- For n=32, you need 30 carry save adders in eight stages taking 8T time where T is time for one-bit full adder
- Then you need one carry-propagate or carry-look-ahead adder

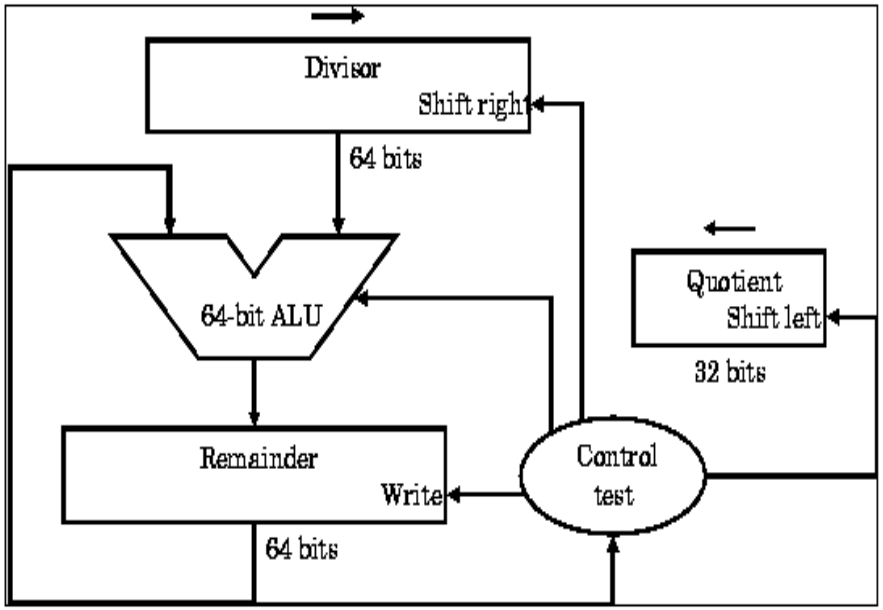
Division

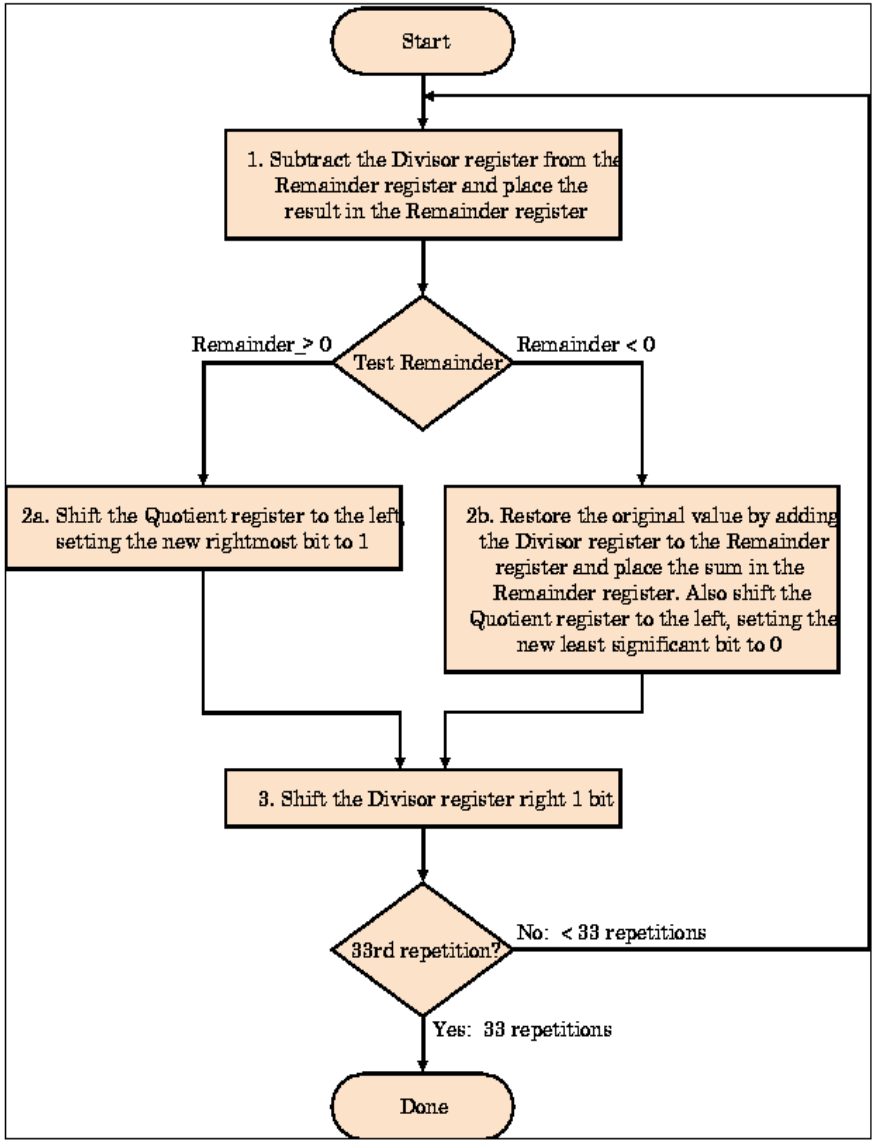
- Even more complicated
  - can be accomplished via shifting and addition/subtraction
- More time and more area
- We will look at 3 versions based on grade school algorithm

```
 0011 | 0010 0010   (Dividend)
------
```

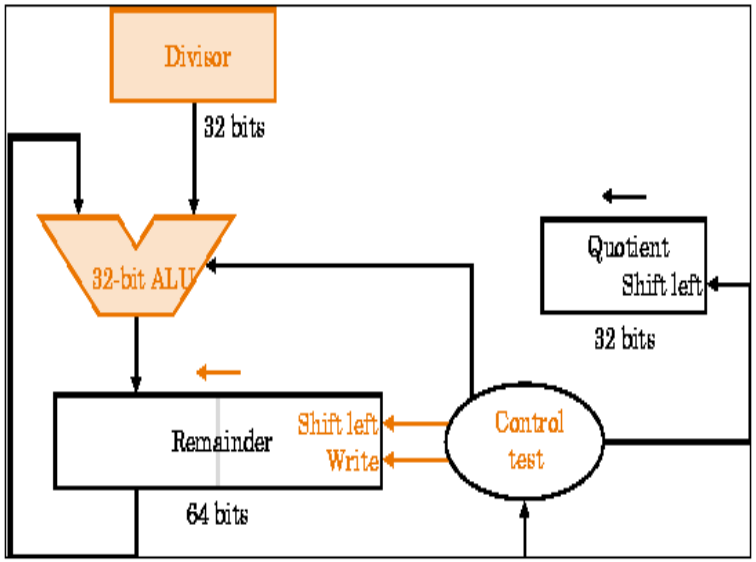
- Negative numbers: Even more difficult
- There are better techniques, we won't look at them

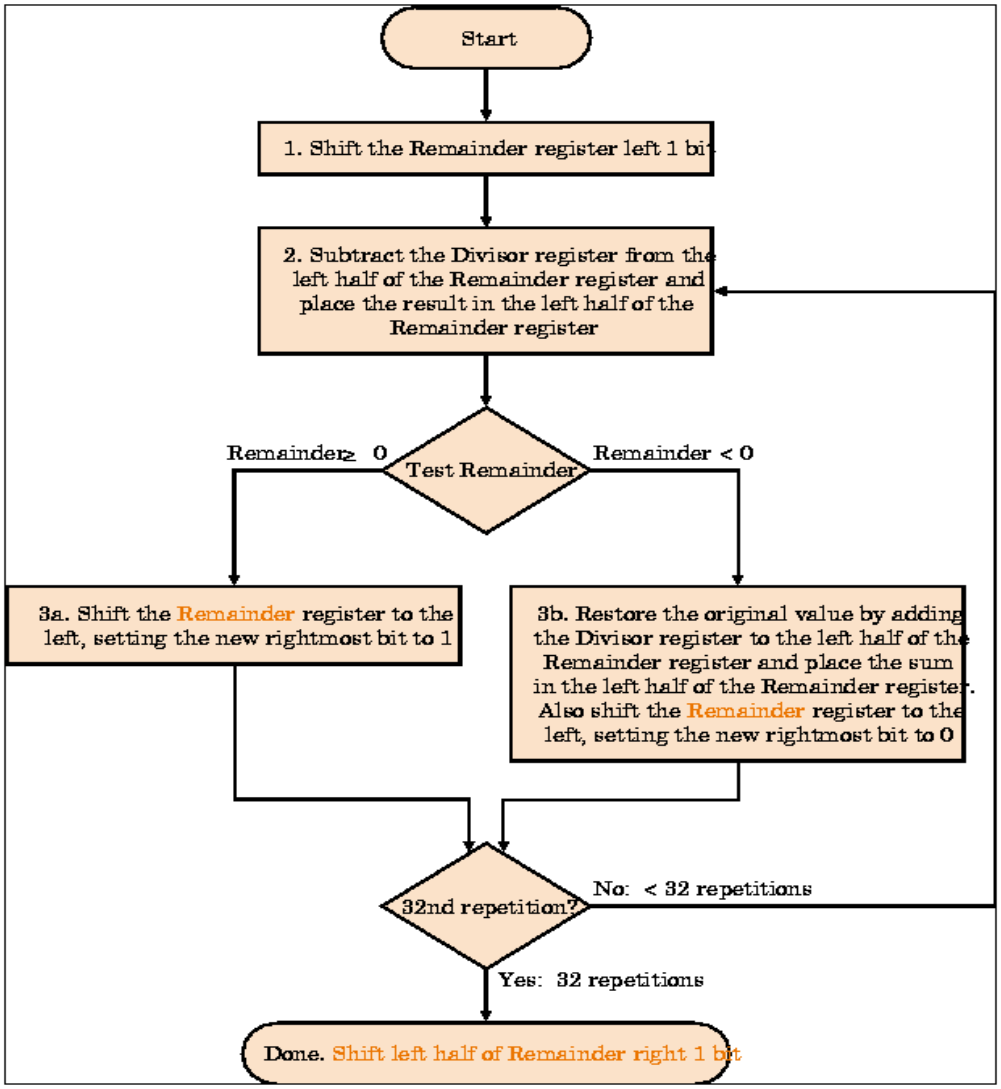
Division, First Version



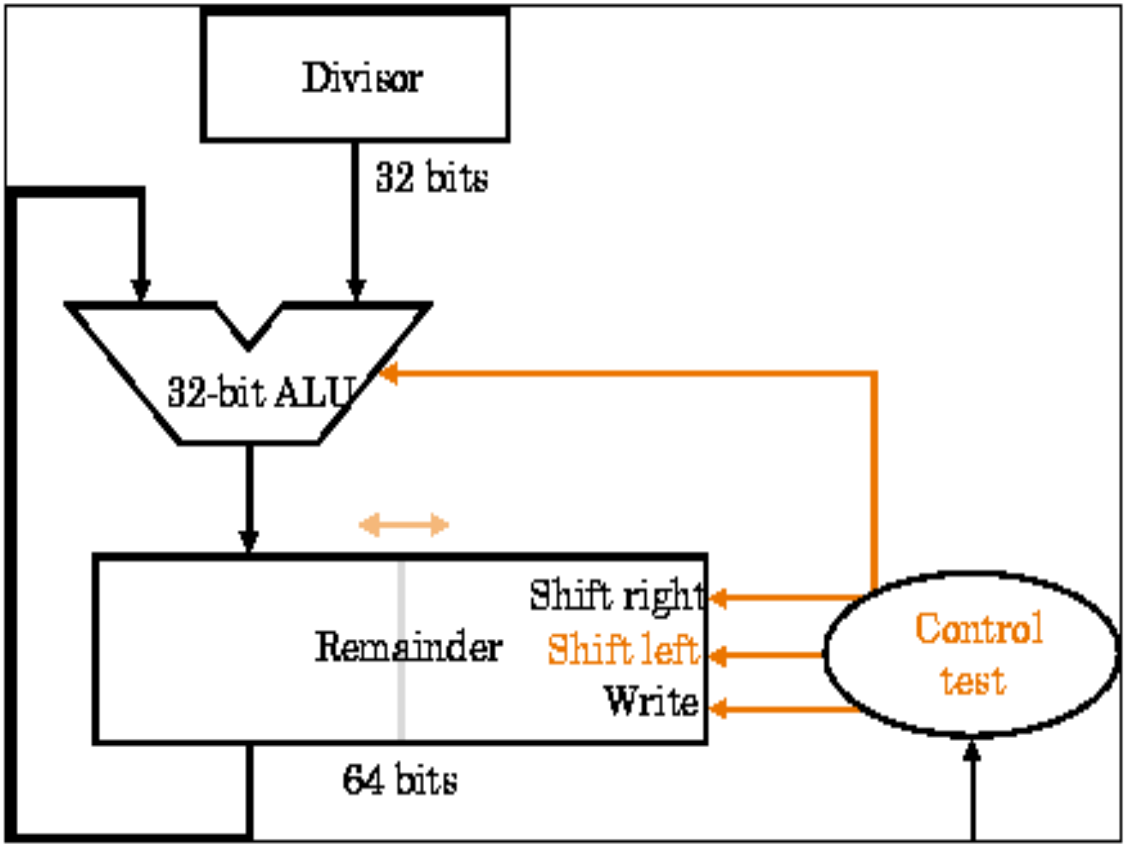


Division, Second Version





Division, Final Version



Restoring Division

Iteration	Divisor	Divide algorithm	
		Step	Remainder
0	0010	Initial values	0000 0111
	0010	Shift Rem left 1	0000 1110
1	0010	2: Rem = Rem - Div	1110 1110
	0010	3b: Rem < 0 ⇒ + Div, sll R, R0 = 0	0001 1100
2	0010	2: Rem = Rem - Div	1111 1100
	0010	3b: Rem < 0 ⇒ + Div, sll R, R0 = 0	0011 1000
3	0010	2: Rem = Rem - Div	0001 1000
	0010	3a: Rem ≥ 0 ⇒ sll R, R0 = 1	0011 0001
4	0010	2: Rem = Rem - Div	0001 0001
	0010	3a: Rem ≥ 0 ⇒ sll R, R0 = 1	0010 0011
Done	0010	shift left half of Rem right 1	0001 0011

Non-Restoring Division

Iteration	Divisor	Divide algorithm	
		Step	Remainder
0	0010	Initial values	0000 1110
1	0010	1: Rem = Rem - Div	1110 1110
	0010	2b: Rem < 0 ⇒,sll R, R0 = 0	1101 1100
	0010	3b: Rem = Rem + Div	1111 1100
2	0010	2b: Rem < 0 ⇒ sll R, R0 = 0	1111 1000
	0010	3b: Rem = Rem + Div	0001 1000
3	0010	2a: Rem > 0 ⇒ sll R, R0 = 1	0011 0001
	0010	3a: Rem = Rem - Div	0001 0001
4	0010	2a: Rem > 0 ⇒ sll R, R0 = 1	0010 0011
Done	0010	shift left half of Rem right 1	0001 0011