CMSC 411, Fall 2017
Computer Architecture
Algorithms

SESSION 2:	SHERMAN 013, TUESDAY/THURSDAY, 5:30-6:45PM
SESSION 3:	SHERMAN 013, TUESDAY/THURSDAY, 7:10PM - 8:25PM

Professor:	Russ Cain
eMail:	rpcain@umbc.edu
Office:	ITE 200F
Hours:	Tuesdays & Thursdays 6:45-7:10 PM
	& by appointment 8:25-9:00 PM

TA:	Dayuan Tan
eMail:	dayuan1@umbc.edu
Office:	ITE 344B
Hours:	Tuesday and Thursday, 2-4:30pm
	& by appointment

Schedule

Week	Date	Class	Reading	HMWK
1		This is dynamic and will change during semester		
	31-Aug	Lecture 1: Introduction, Overview, History	1.1-1.4	
2	5-Sep	Lecture 2: Performance Evaluation and Metrics	1.6-1.10	HW#1
	7-Sep	Logic and Making ICs	A, 1.5	
3	12-Sep	VHDL	4.13	HW#2
	14-Sep	Instruction Set Architecture - ARM	2.1-2.11	
4	19-Sep	Instruction Set Architecture - ARM, MIPS, x86	2.16-2.19	HW#3
	21-Sep	Arithmetic/Carry/Floating Point	3.5-3.9	
5	26-Sep	Arithmetic/Carry/Floating Point	4.1-4.4	HW#4
	28-Sep	ALU		
6	3-Oct	Pipelining 1	3.1-3.4	HW#5
	5-Oct	Pipelining 2		
7	10-Oct	Pipelining 3	4.5-4.6	HW#6
	12-Oct	Review		
8	17-Oct	Algorithms		
	19-Oct	Midterm		
9	24-Oct	Hardware Algorithmsand Project		
	26-Oct	Cache	4.7-4.14	
10	31-Oct	Cache	5.1-5.3	HW#7
	2-Nov	Cache Performance	5.4-5.45	
11	7-Nov	Virtual Machines & Memory	5.6-5.7	HW#8
	9-Nov	I/O Types & Performance, Busses		
12	14-Nov	Multicores, Multiprocessors, & Clusters	6.1-6.5	HW#9
	16-Nov	Embedded Computers	6.6	
13	21-Nov	Microprogramming (Dayuan)		HW#10
	23-Nov	THANKSGIVING		
14	28-Nov	The Mill		
	30-Nov	Superscalar(Dayuan)	C	
15	5-Dec	Quantum Computing and Neural Nets		
	7-Dec	TBD		
16	12-Dec	Review		
	14-Dec	Final Exam, 6-8 PM & 8:30-10:30 P, In class		

Lectures

- *Last Lecture*
  - *A fun midterm review*
- *This Lecture*
  - *Hardware tradeoffs*
  - *Algorithms*

COMPARISON OF RIPPLE VS CLA

- *n-bit ripple*
  - *Delay = 2n*
- *n-bit CLA*
  - *Delay = 5 or 3 + 2[log n]*

- *8-bit ripple Delay 16.*
- *8-bit CLA Delay 5 or 8 or 2x faster.*
- *32-bit ripple Delay 64.*
- *32-bit CLA Delay 5 or 13 or 5x faster*
- *64-bit ripple Delay 128.*
- *64-bit CLA Delay 5 or 15 or 9x faster*

- *CLA looks much better. . .until we compute the cost!*

N-BIT CLA COST ANALYSIS

n-bit CLA Cost Analysis

Cost of each CLA Cell:

$$\overbrace{5}^{\text{Sum}} + \overbrace{1}^{\text{Gen.}} + \overbrace{1}^{\text{Prop.}} = 7$$

Cost of logic to compute c_i given p and g signals:

$$\overbrace{i}^{\text{OR Gate}} + \overbrace{\sum_{j=2}^{i+1} j-1}^{\text{AND Gates}} = i + \frac{i(i+1)}{2} = \frac{i(i+3)}{2}$$

Cost of logic to compute c_1 to c_n:

$$\sum_{i=1}^{n} \frac{i(i+3)}{2} = \frac{1}{6}n(n+1)(n+5)$$

Cost of entire n-bit CLA:

$$\overbrace{7n}^{\text{CLA Cells}} \overbrace{+\frac{1}{6}n(n+1)(n+5)}^{\text{Carry Gen Logic}} \approx \frac{n^3}{6}$$

COST OF SELECTED SIZES

- *n-bit ripple*
  - *10n*
- *n-bit CLA*
  - *Delay = 7n + 1/6 n (n + 1) (n + 5)*

- *8-bit ripple: 80 gates*
- *8-bit CLA: 212 gates or 2.65x cost of ripple adder*
- *32-bit ripple: 320 gates*
- *32-bit CLA: 6736 gates or 21x cost of ripple adder*
- *64-bit ripple: 640 gates*
- *64-bit CLA: 48299 gates or 75x cost of ripple adder*

§3.3 Multiplication

MULTIPLICATION

- *Start with long-multiplication approach*

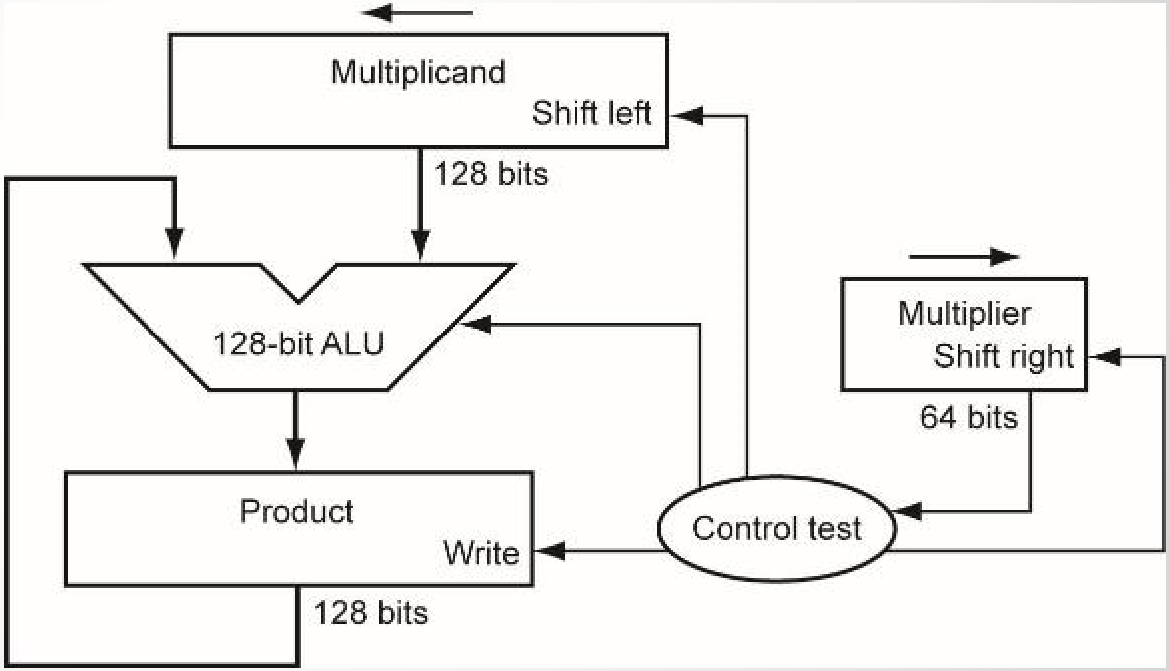
multiplicand

multiplier

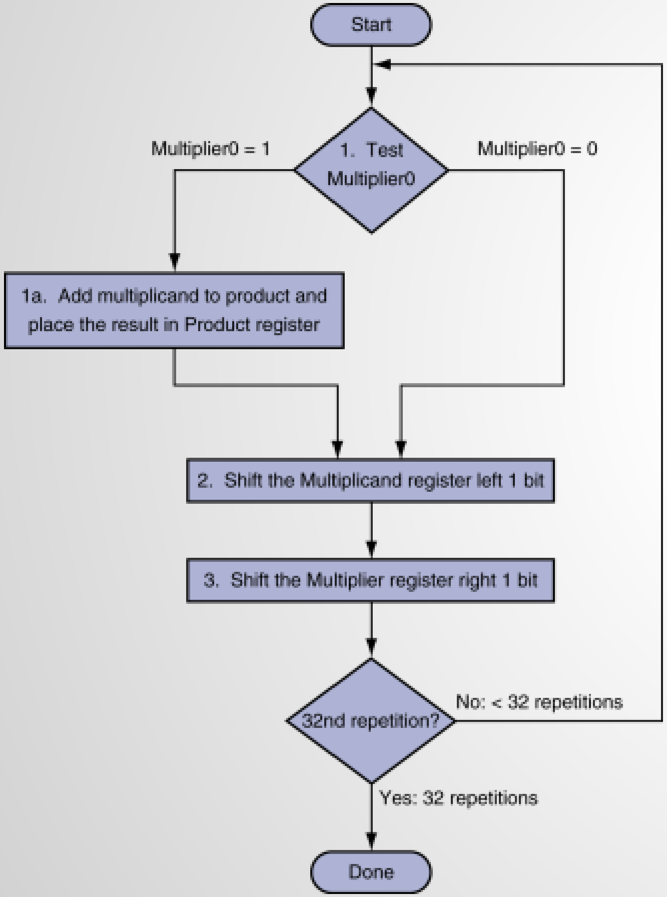
product

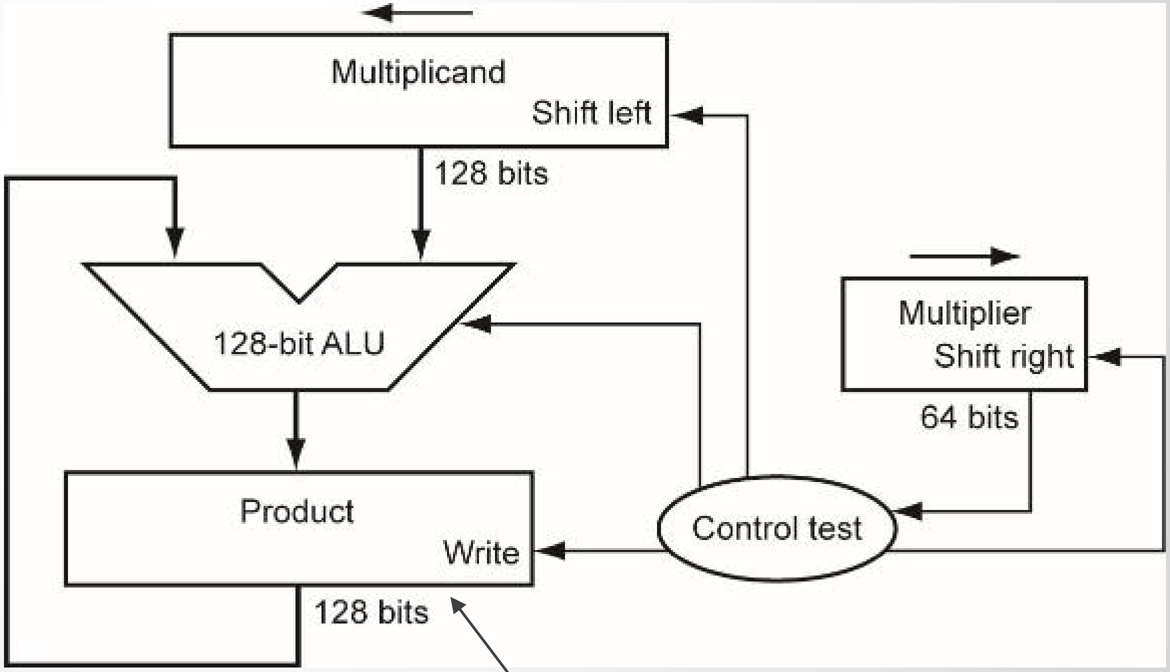
```
      1000
  ×   1001
  --------
      1000
     0000
    0000
   1000
  --------
   1001000
```

Length of product is the sum of operand lengths



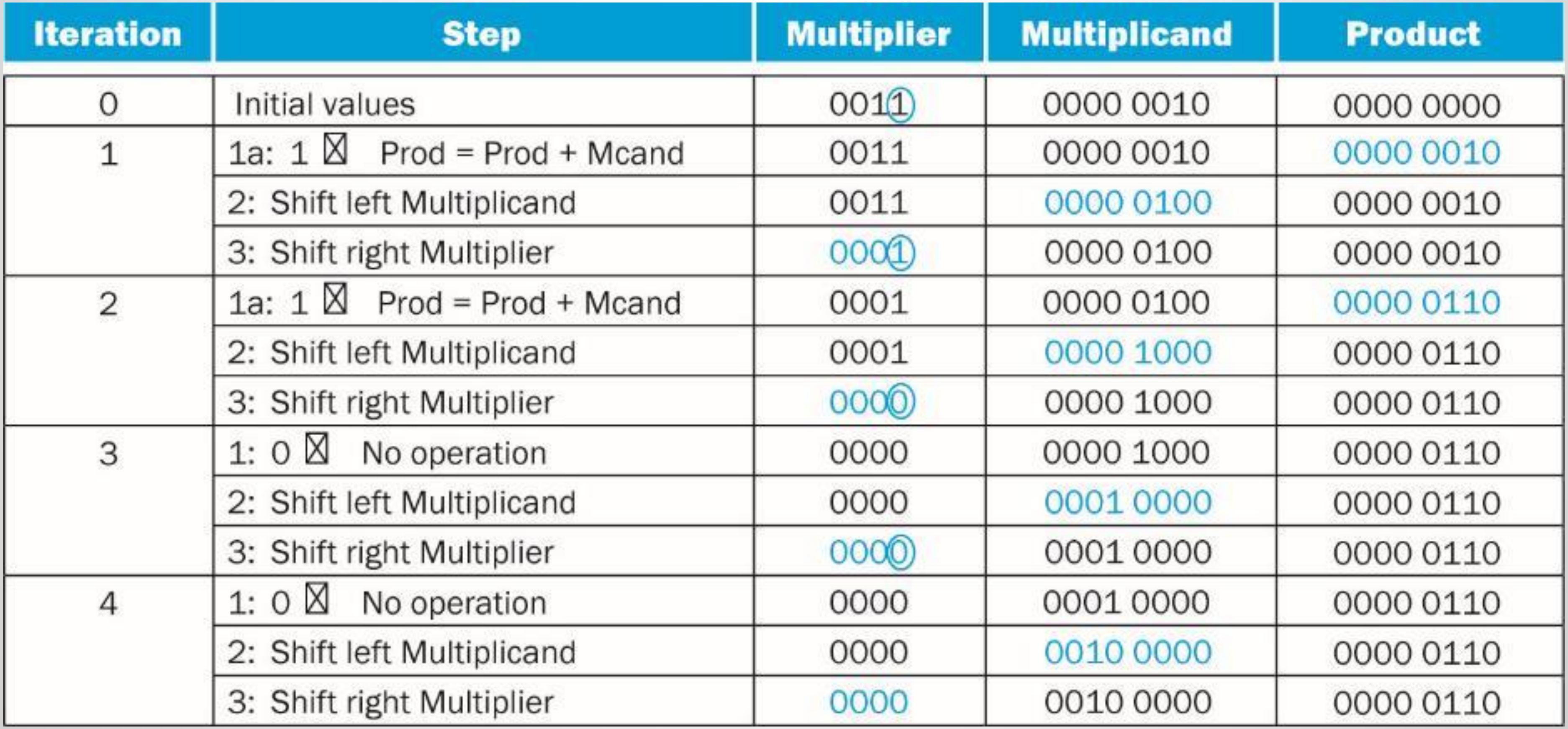
MULTIPLICATION HARDWARE


Start
1. Test Multiplier0
Multiplier0 = 1
Multiplier0 = 0
1a. Add multiplicand to product and place the result in Product register
2. Shift the Multiplicand register left 1 bit
3. Shift the Multiplier register right 1 bit
32nd repetition?
No: < 32 repetitions
Yes: 32 repetitions
Done



Initially 0

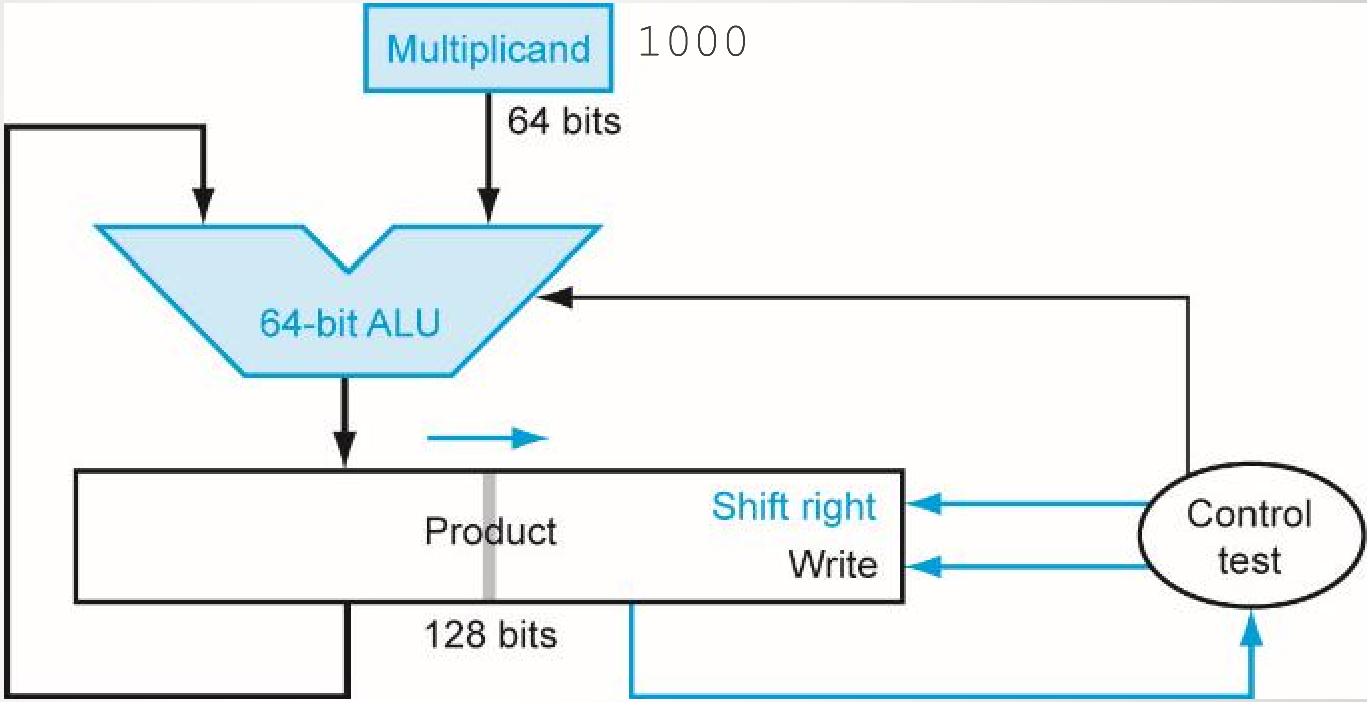
MULTIPLY ALGORITHM



Iteration	Step	Multiplier	Multiplicand	Product
0	Initial values	0011	0000 0010	0000 0000
1	1a: 1 ☒ Prod = Prod + Mcand	0011	0000 0010	0000 0010
	2: Shift left Multiplicand	0011	0000 0100	0000 0010
	3: Shift right Multiplier	0001	0000 0100	0000 0010
2	1a: 1 ☒ Prod = Prod + Mcand	0001	0000 0100	0000 0110
	2: Shift left Multiplicand	0001	0000 1000	0000 0110
	3: Shift right Multiplier	0000	0000 1000	0000 0110
3	1: 0 ☒ No operation	0000	0000 1000	0000 0110
	2: Shift left Multiplicand	0000	0001 0000	0000 0110
	3: Shift right Multiplier	0000	0001 0000	0000 0110
4	1: 0 ☒ No operation	0000	0001 0000	0000 0110
	2: Shift left Multiplicand	0000	0010 0000	0000 0110
	3: Shift right Multiplier	0000	0010 0000	0000 0110

OPTIMIZED MULTIPLIER

- *Perform steps in parallel: add/shift*



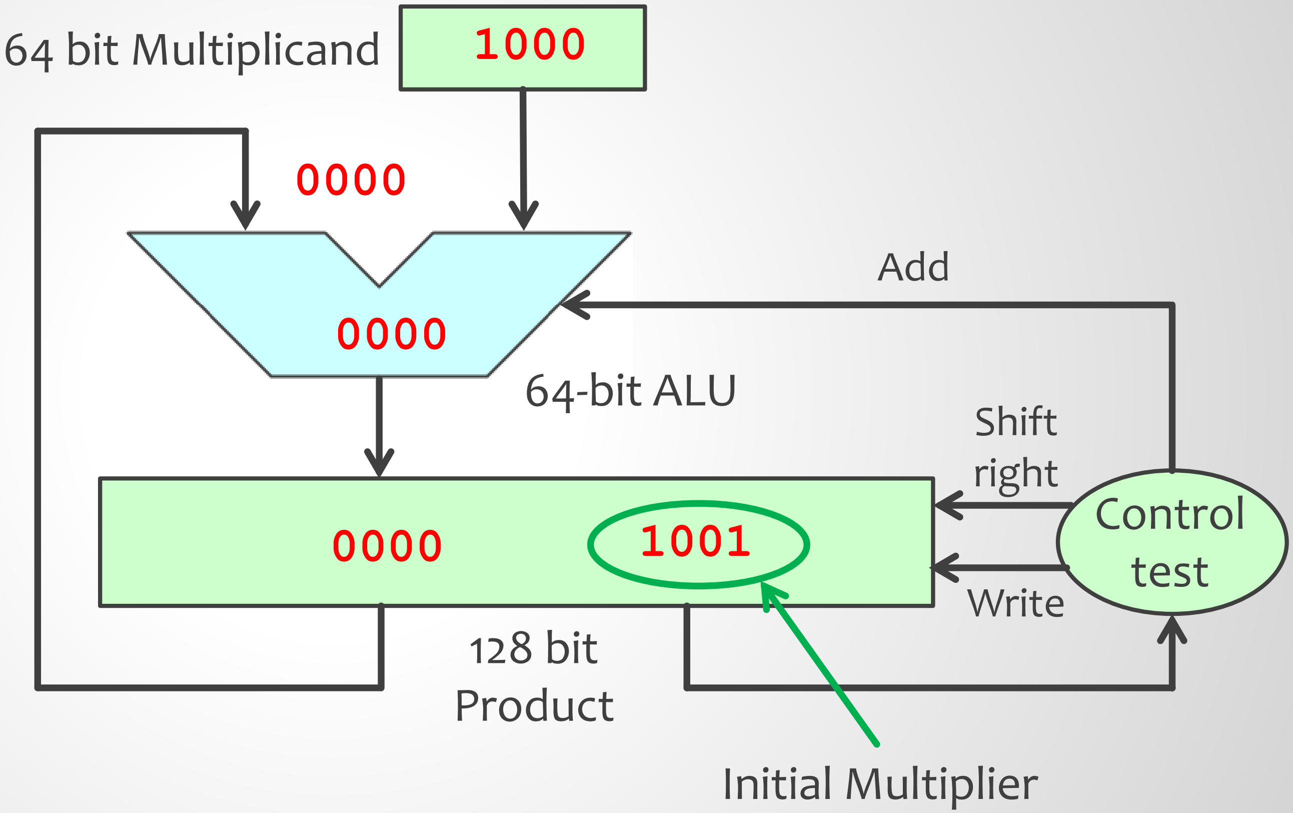
1000

```
         Product Register
                0000 Multiplier
                0000 1001

shift           0000 0100 → 1
add             1000 0100

shift           0100 0010 → 0

shift           0010 0001 → 0

shift           0001 0000 → 1
add             1001 0000

shift           0100 1000 → 1
```

- *One cycle per partial-product addition*
  - *That's ok, if frequency of multiplications is low*

OPTIMIZED MULTIPLIER

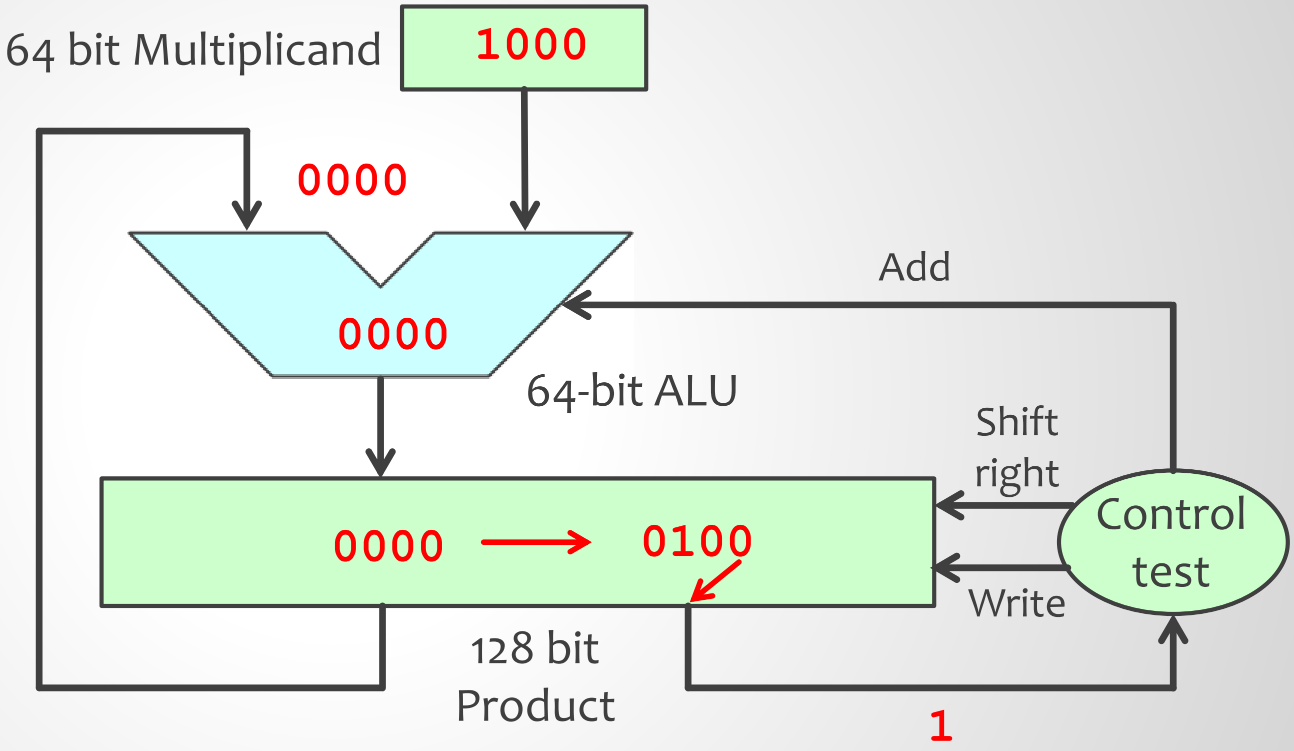
Time step: 0000



```
             Product Register
                 0000 Multiplier
                 0000 1001

shift            0000 0100 → 1
add              1000 0100

shift            0100 0010 → 0

shift            0010 0001 → 0

shift            0001 0000 → 1
add              1001 0000

shift            0100 1000 → 1
```

OPTIMIZED MULTIPLIER

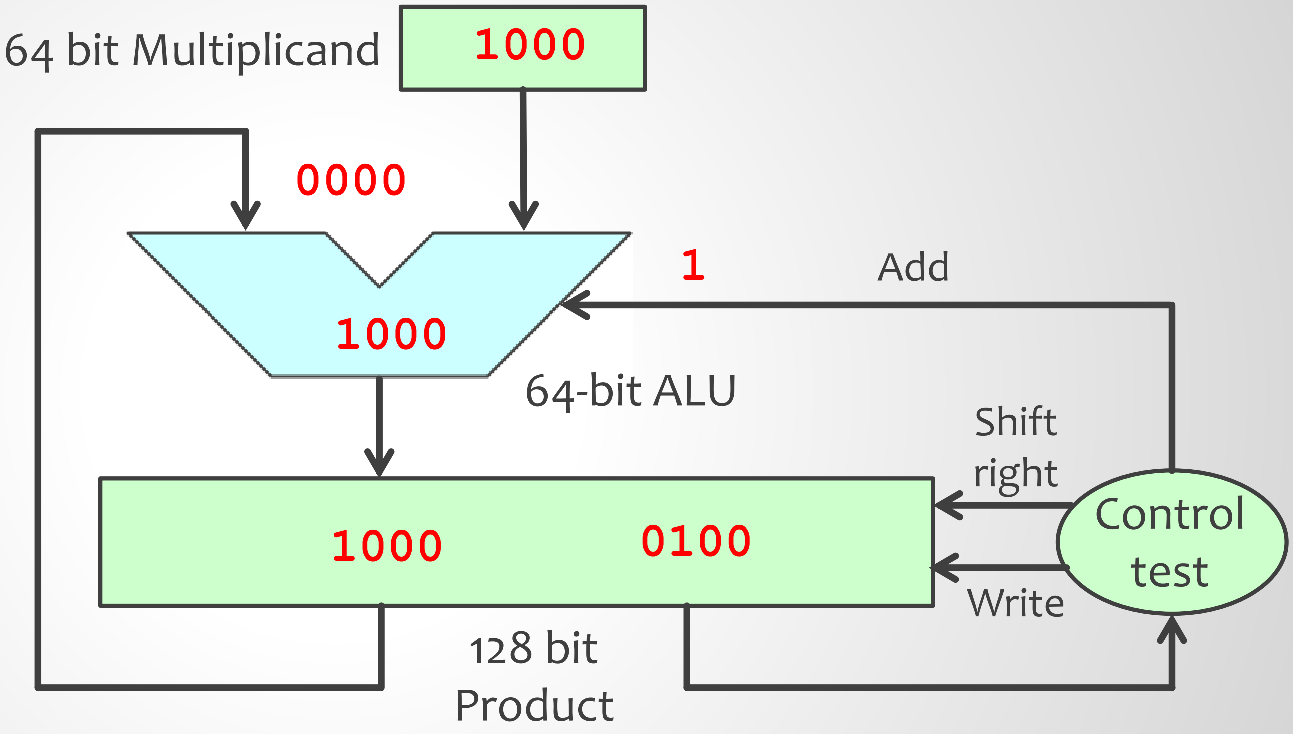
Time step: 0001



```
    Product Register
          0000 Multiplier
          0000 1001

shift     0000 0100 → 1
add       1000 0100

shift     0100 0010 → 0

shift     0010 0001 → 0

shift     0001 0000 → 1
add       1001 0000

shift     0100 1000 → 1
```

OPTIMIZED MULTIPLIER

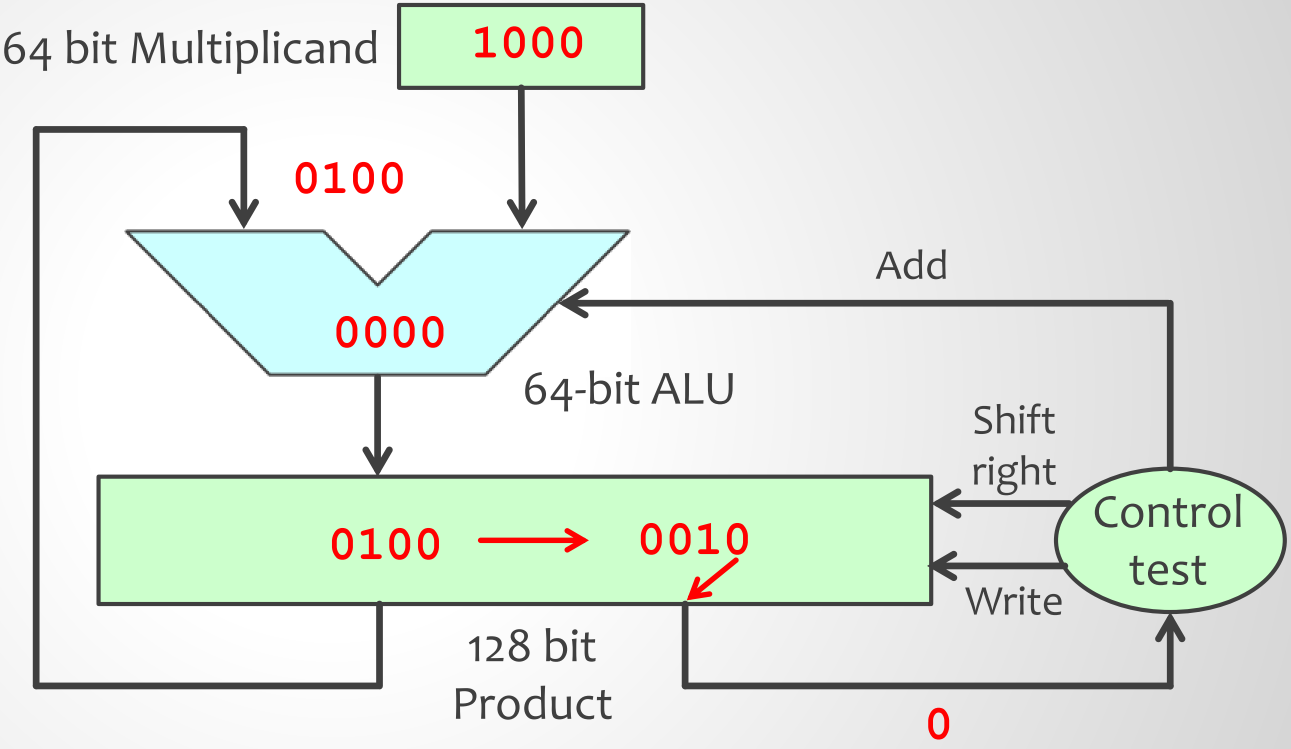
Time step: 0002



```
                Product Register
                    0000 Multiplier
                    0000 1001

shift               0000 0100 → 1
add                 1000 0100

shift               0100 0010 → 0

shift               0010 0001 → 0

shift               0001 0000 → 1
add                 1001 0000

shift               0100 1000 → 1
```

OPTIMIZED MULTIPLIER

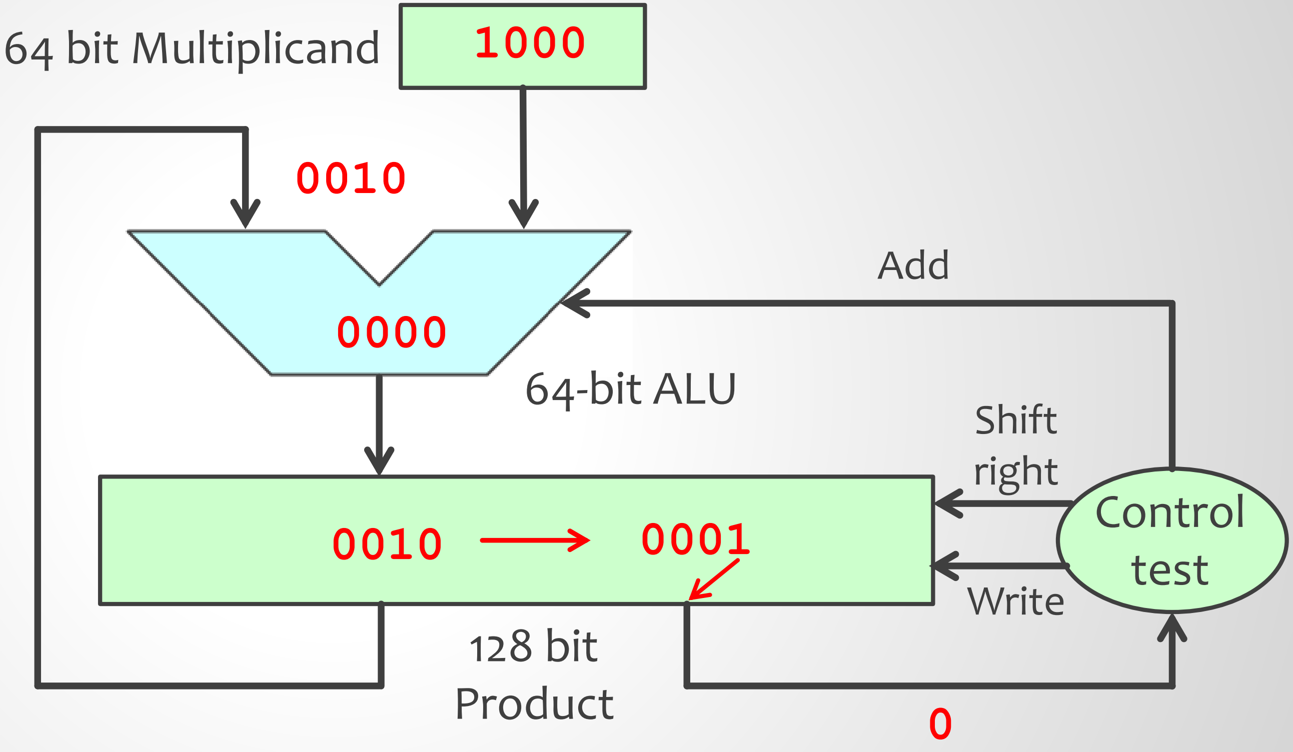
Time step: 0004



```
            Product Register
                0000 Multiplier
                0000 1001

shift           0000 0100 → 1
add             1000 0100

shift           0100 0010 → 0

shift           0010 0001 → 0

shift           0001 0000 → 1
add             1001 0000

shift           0100 1000 → 1
```

OPTIMIZED MULTIPLIER

Time step: 0005

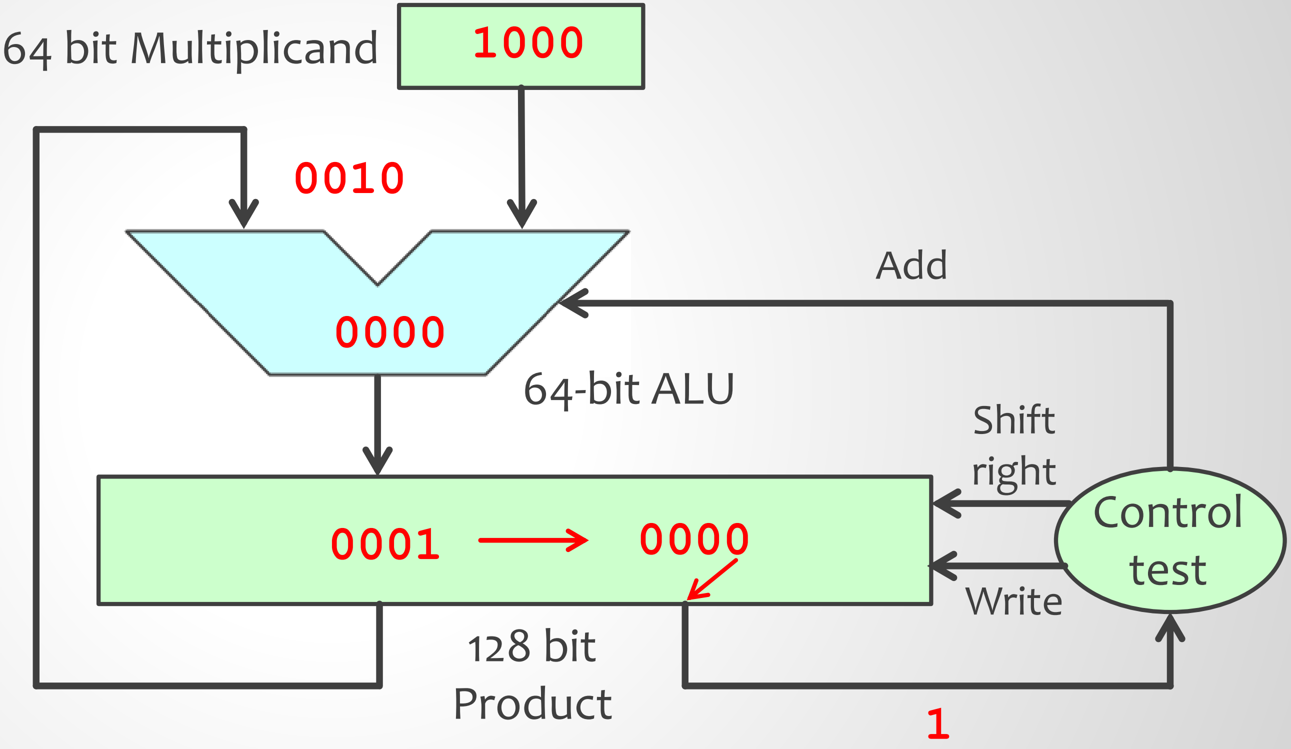


Product Register

```
              0000 Multiplier
              0000 1001

shift         0000 0100 → 1
add           1000 0100

shift         0100 0010 → 0

shift         0010 0001 → 0

shift         0001 0000 → 1
add           1001 0000

shift         0100 1000 → 1
```

OPTIMIZED MULTIPLIER

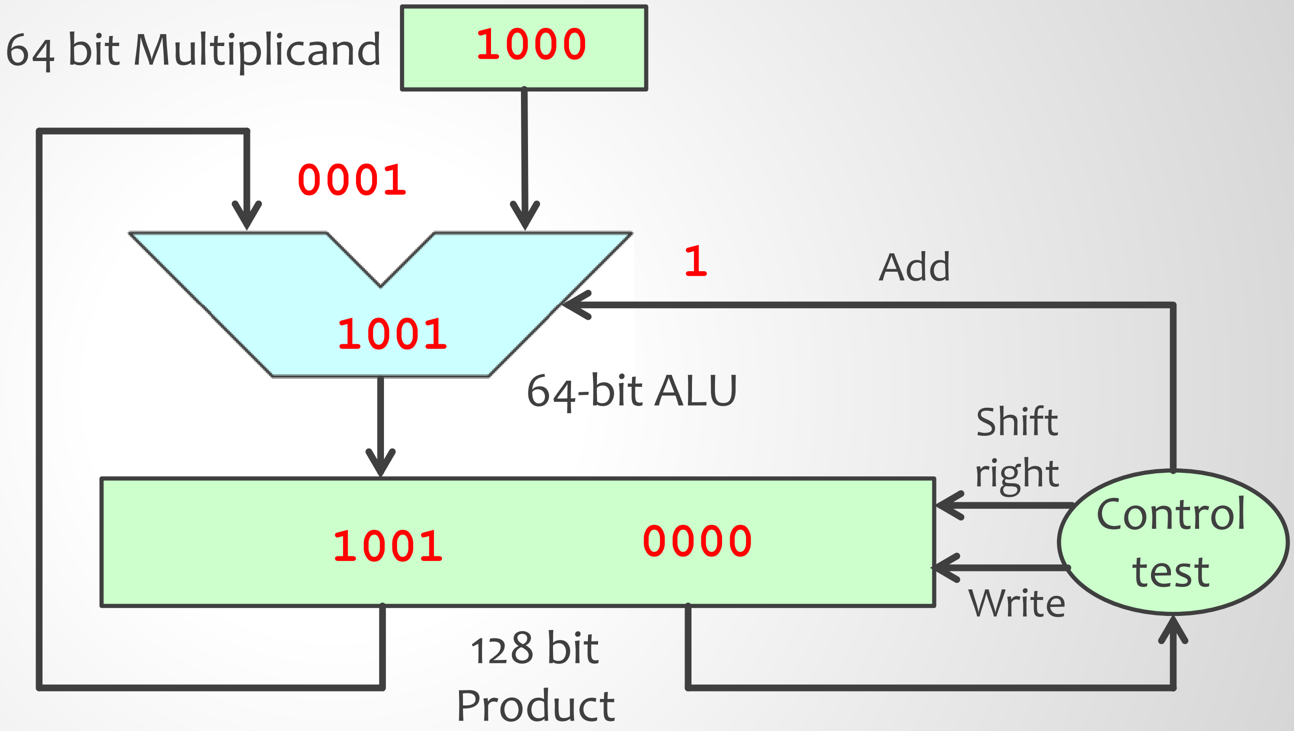
Time step: 0006



```
         Product Register
              0000 Multiplier
              0000 1001

shift         0000 0100 → 1
add           1000 0100

shift         0100 0010 → 0

shift         0010 0001 → 0

shift         0001 0000 → 1
add           1001 0000

shift         0100 1000 → 1
```

OPTIMIZED MULTIPLIER

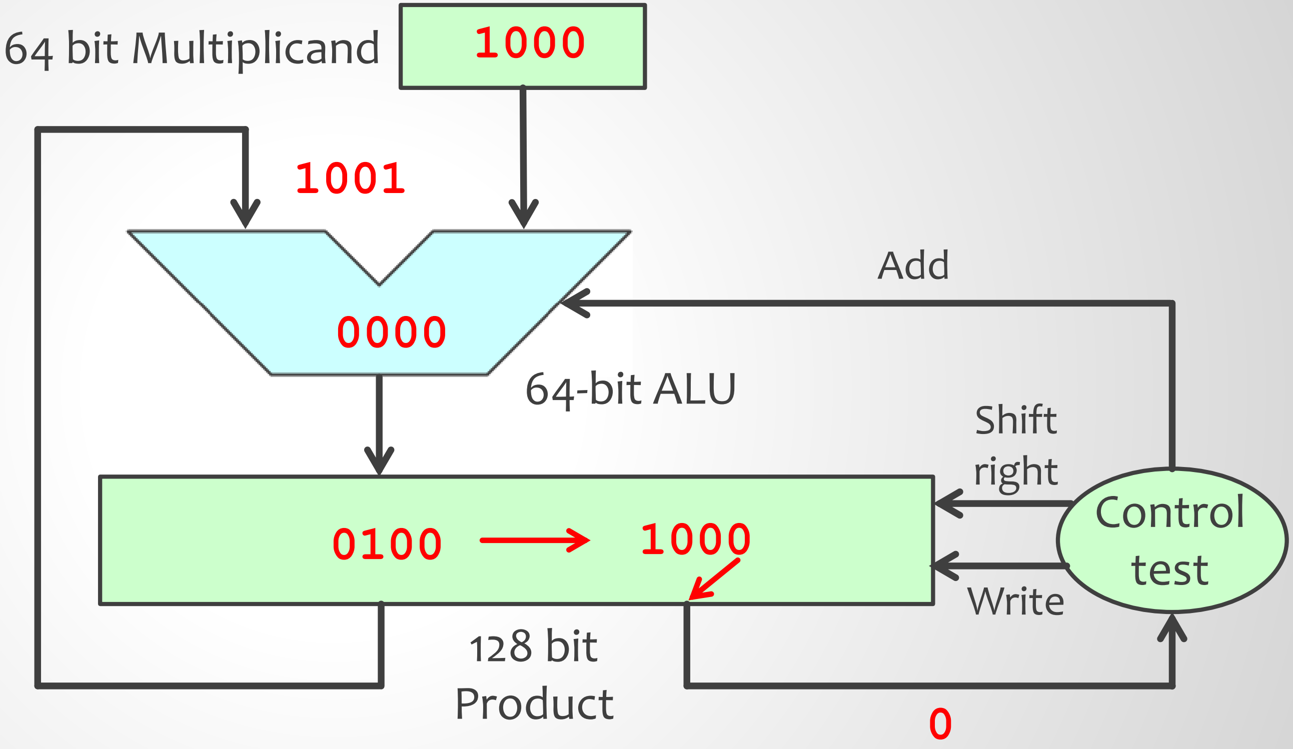
Time step: 0007



```
                Product Register
                    0000 Multiplier
                    0000 1001

shift               0000 0100 → 1
add                 1000 0100

shift               0100 0010 → 0

shift               0010 0001 → 0

shift               0001 0000 → 1
add                 1001 0000

shift               0100 1000 → 1
```

OPTIMIZED MULTIPLIER

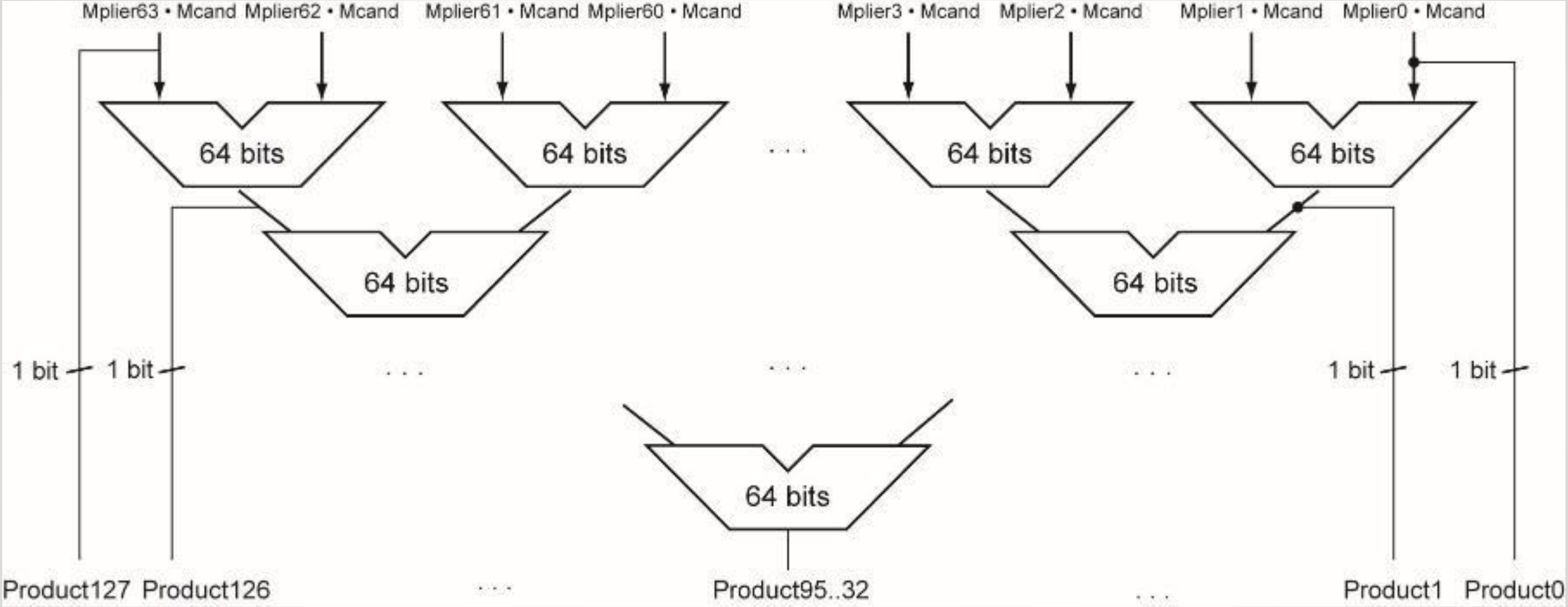
Time step: 0008



```
        Product Register
            0000 Multiplier
            0000 1001

shift       0000 0100 → 1
add         1000 0100

shift       0100 0010 → 0

shift       0010 0001 → 0

shift       0001 0000 → 1
add         1001 0000

shift       0100 1000 → 1
```

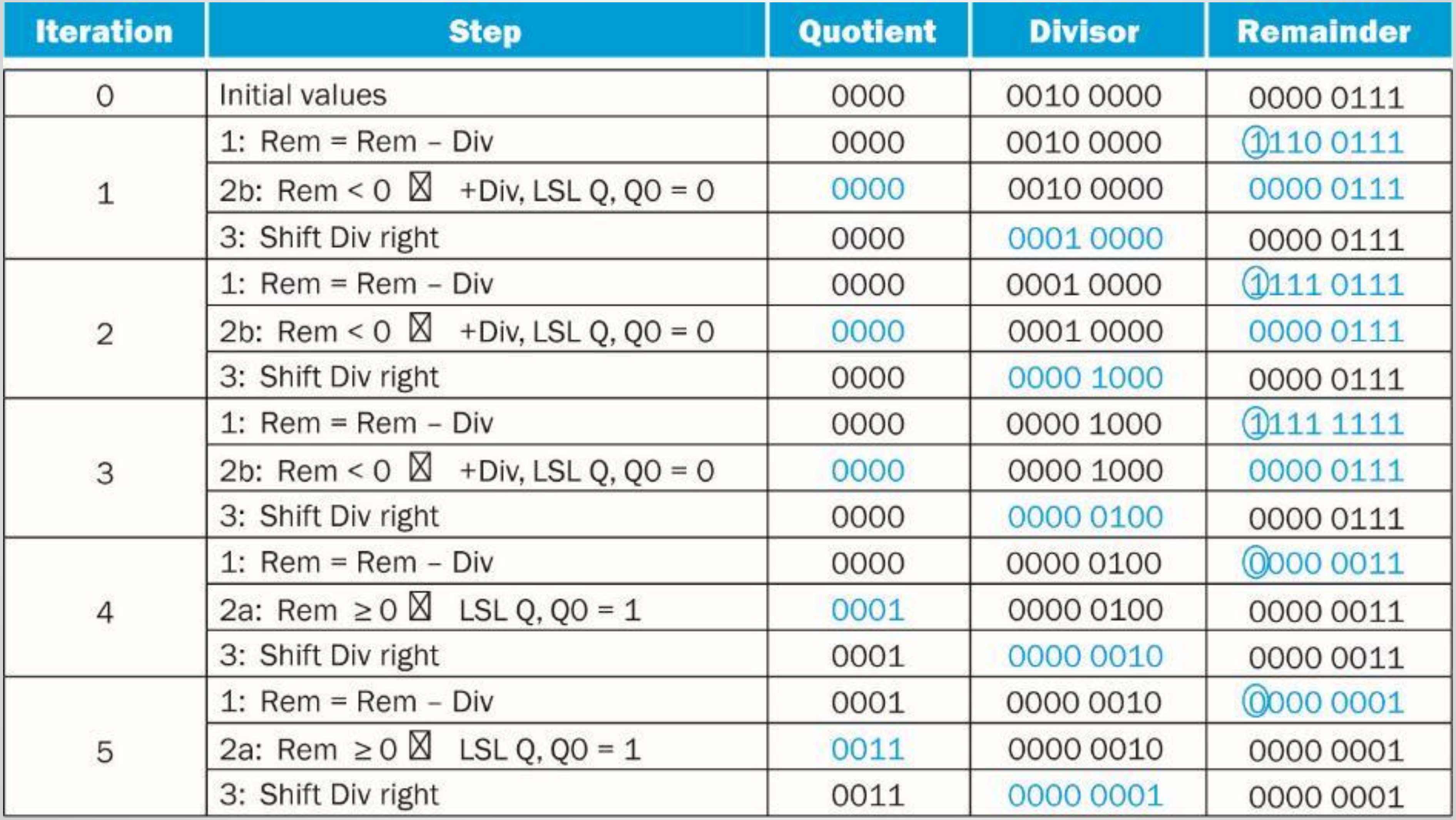
FASTER MULTIPLIER

- *Uses multiple adders*
    - *Cost/performance tradeoff*



- *Can be pipelined*
    - *Several multiplication performed in parallel*
- *Sacrifice speed for area*

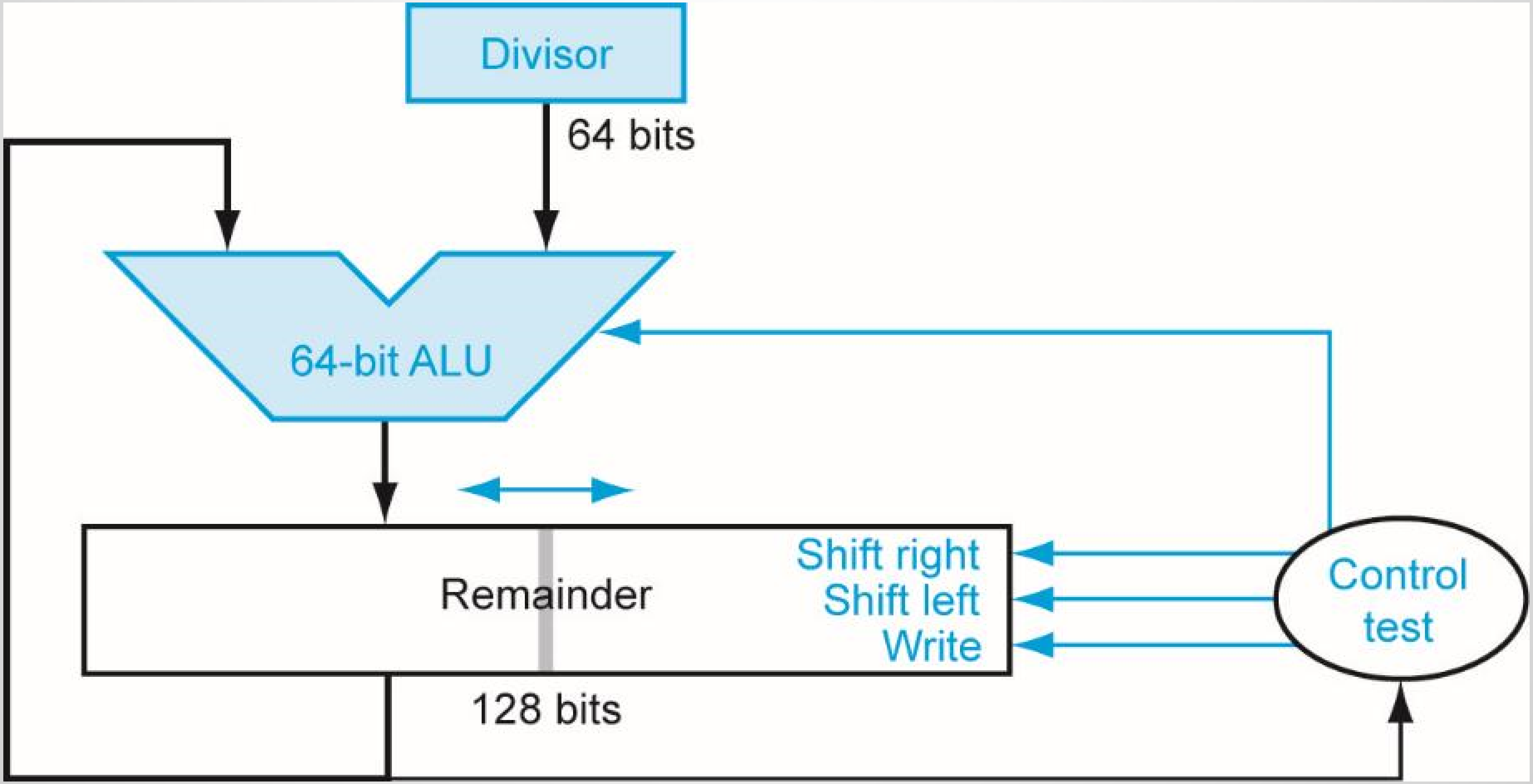
DIVIDE ALGORITHM



Iteration	Step	Quotient	Divisor	Remainder
0	Initial values	0000	0010 0000	0000 0111
1	1: Rem = Rem – Div	0000	0010 0000	1110 0111
	2b: Rem < 0 ⊠ +Div, LSL Q, Q0 = 0	0000	0010 0000	0000 0111
	3: Shift Div right	0000	0001 0000	0000 0111
2	1: Rem = Rem – Div	0000	0001 0000	1111 0111
	2b: Rem < 0 ⊠ +Div, LSL Q, Q0 = 0	0000	0001 0000	0000 0111
	3: Shift Div right	0000	0000 1000	0000 0111
3	1: Rem = Rem – Div	0000	0000 1000	1111 1111
	2b: Rem < 0 ⊠ +Div, LSL Q, Q0 = 0	0000	0000 1000	0000 0111
	3: Shift Div right	0000	0000 0100	0000 0111
4	1: Rem = Rem – Div	0000	0000 0100	0000 0011
	2a: Rem ≥ 0 ⊠ LSL Q, Q0 = 1	0001	0000 0100	0000 0011
	3: Shift Div right	0001	0000 0010	0000 0011
5	1: Rem = Rem – Div	0001	0000 0010	0000 0001
	2a: Rem ≥ 0 ⊠ LSL Q, Q0 = 1	0011	0000 0010	0000 0001
	3: Shift Div right	0011	0000 0001	0000 0001

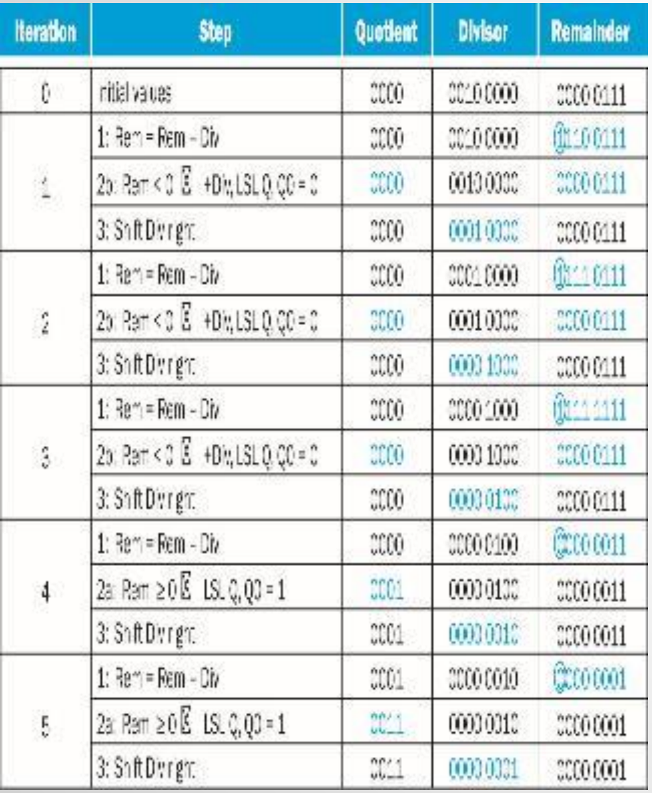
OPTIMIZED DIVIDER

- *One cycle per partial-remainder subtraction*
- *Looks a lot like a multiplier!*
  - *Same hardware can be used for both*

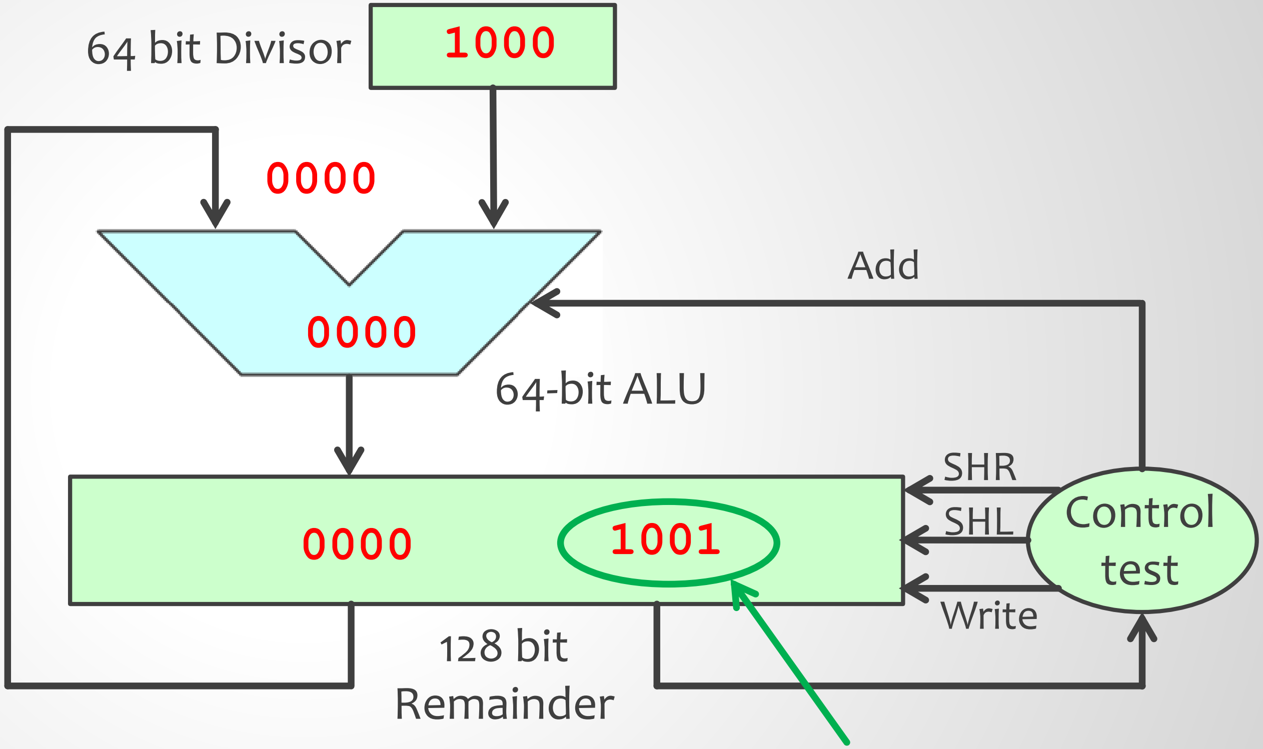


OPTIMIZED DIVIDER

Time step: 0000



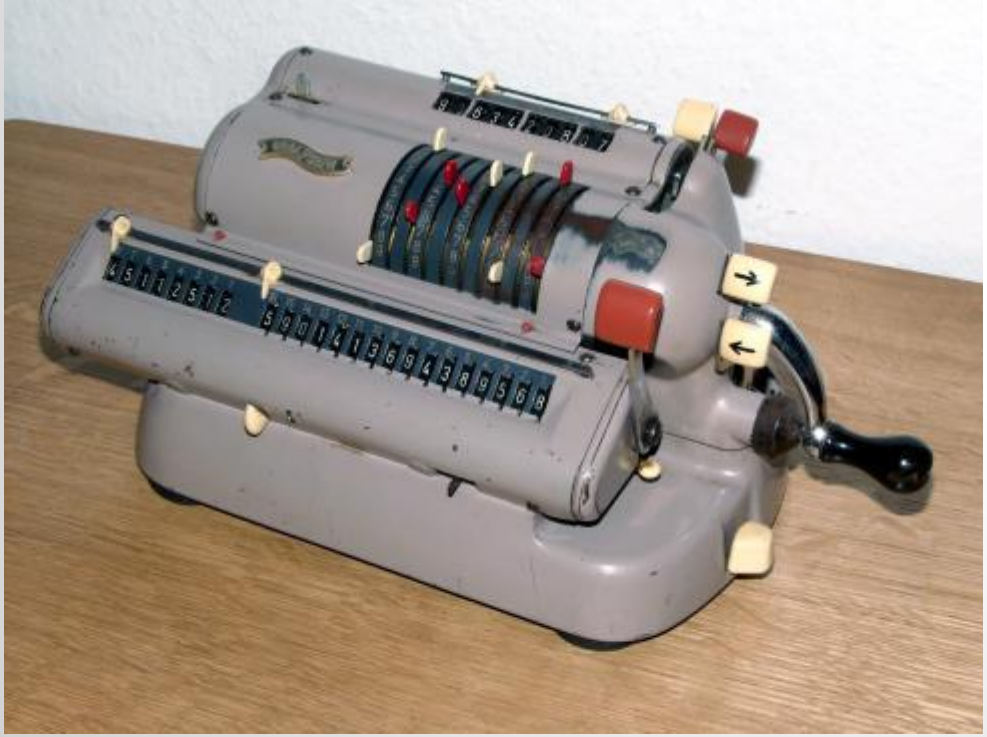
Iteration	Step	Quotient	Divisor	Remainder
0	Initial values	0000	0010 0000	0000 0111
1	1: Rem = Rem - Div	0000	0010 0000	1110 0111
	2b: Rem < 0 ⇒ +Div, LSL Q, Q0 = 0	0000	0010 0000	0000 0111
	3: Shift Div right	0000	0001 0000	0000 0111
2	1: Rem = Rem - Div	0000	0001 0000	1111 0111
	2b: Rem < 0 ⇒ +Div, LSL Q, Q0 = 0	0000	0001 0000	0000 0111
	3: Shift Div right	0000	0000 1000	0000 0111
3	1: Rem = Rem - Div	0000	0000 1000	1111 1111
	2b: Rem < 0 ⇒ +Div, LSL Q, Q0 = 0	0000	0000 1000	0000 0111
	3: Shift Div right	0000	0000 0100	0000 0111
4	1: Rem = Rem - Div	0000	0000 0100	0000 0011
	2a: Rem ≥ 0 ⇒ LSL Q, Q0 = 1	0001	0000 0100	0000 0011
	3: Shift Div right	0001	0000 0010	0000 0011
5	1: Rem = Rem - Div	0001	0000 0010	0000 0001
	2a: Rem ≥ 0 ⇒ LSL Q, Q0 = 1	0011	0000 0010	0000 0001
	3: Shift Div right	0011	0000 0001	0000 0001



Initial Quotient

SIGNED MULTIPLICATION WITH BOOTH'S ALGORITHM

- *Invented by Andrew Donald Booth in 1950 while doing research on crystallography at Birkbeck College in Bloomsbury, London.*
- *Originally proposed to reduce addition steps*
- *Bonus: Works for two's complement numbers*
- *Uses shifting, addition, and subtraction*



A Walther WSR160 arithmometer from 1960. Each turn of the crank handle adds (up) or subtracts (down) the operand set to the top register from the value in the accumulator register at the bottom. Shifting the adder left or right multiplies the effect by ten.

BOOTH'S ALGORITHM

- *Observation: If we can both add and subtract, there are multiple ways to create a product*
- *Example: multiply 2_{ten} by 6_{ten} (0010_{two} X 0110_{two})*
  - *Product = (2 X 2) + (2 X 4) OR*
  - *Product = (2 X -2) + (2 X 8)*

Regular Algorithm

```
       0010
x      0110
+      0000   shift
+     0010    shift + add
+    0010     shift + add
+   0000      shift
   00001100
```

Booth's Algorithm

```
       0010
x      0110
       0000    shift
-     0010     shift + subtract
     0000      shift
+   0010       shift + add
   00001100
```

BOOTH'S ALGORITHM CONTINUED

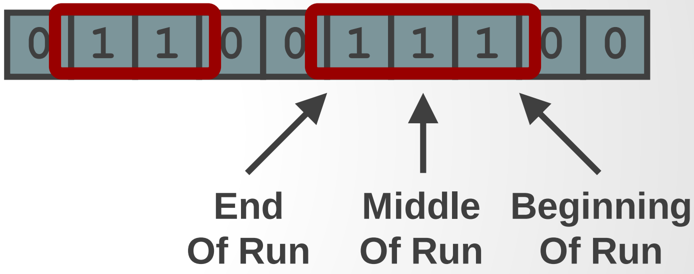
- *Question:*
  - *How do we know when to subtract?*
  - *When do we know when to add?*

- *Answer: look for "runs of 1s" in multiplier*
  - *Example:* $00\underline{111}00\underline{11}$
  - *Working from Right to Left, any "run of 1's" is equal to:*
    - *- value of first digit that's one*
    - *+value of first digit that's zero*
  - *Example :* $00\underline{111}00\underline{11}$
    - *First run: -1 + 4 = 3*
    - *Second run: -16 + 128 = 112*
    - *Total: 112 + 3 = 115*

- *Why is this faster?*

IMPLEMENTING BOOTH'S ALGORITHM

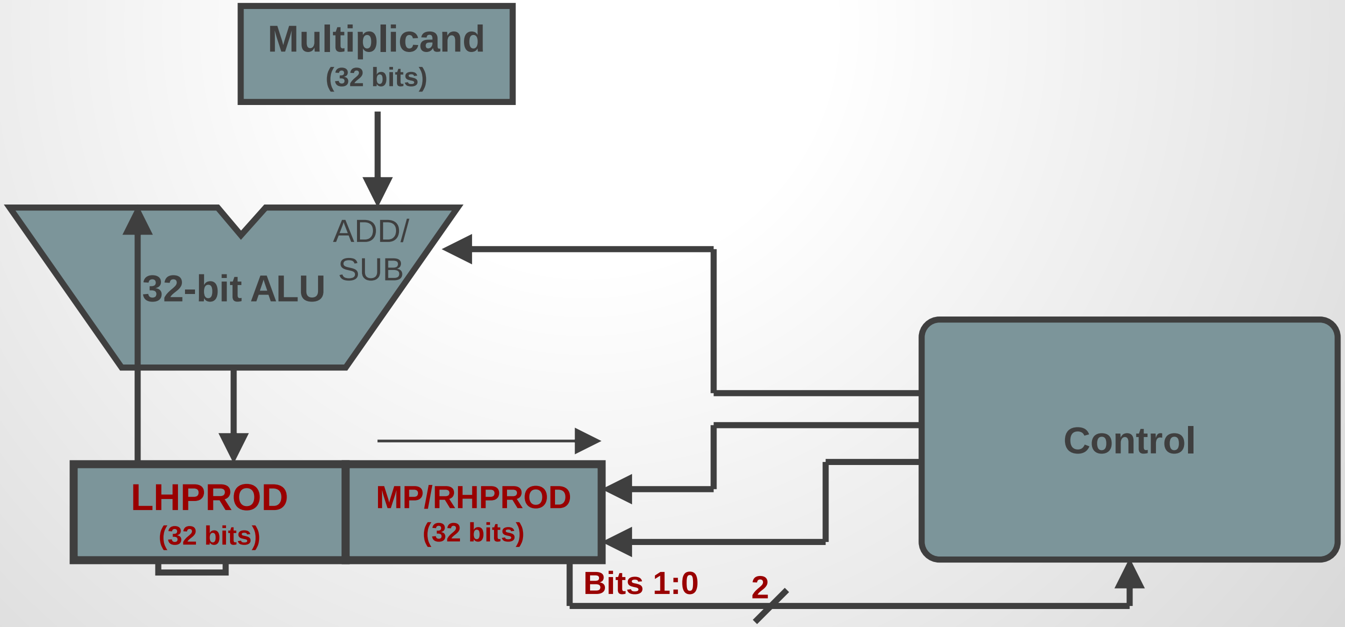
- *Scan multiplier bits from right to left*
- *Recognize the beginning and in of a run looking at only 2 bits at a time*
  - *"Current" bit a_i*
  - *Bit to right of "current" bit a_{i-1}*



Bit a_i	Bit a_{i-1}	Explanation
1	0	Begin Run of 1's
1	1	Middle of Run of 1's
0	1	End of Run
0	0	Middle of Run of 0's

IMPLEMENTING BOOTH'S ALGORITHM

- *Key idea: test 2 bits of multiplier at once*
  - *10 - subtract (beginning of run of 1's)*
  - *01 - add (end of run of 1's)*
  - *00, 11 - do nothing (middle of run of 0's or 1's)*



BOOTH'S ALGORITHM EXAMPLE

Multiply 4 X -9

```
   00100
 X 10111
```

```
          LHProd LHProd
           00000101110
Step 0    +11100               (sub 4 = add -4)
           11100101110
Step 1     11110010111         (shift after add)
Step 2     11111001011         (shift without add)
Step 3     11111100101         (shift without add)
          +00100               (add +4)
           00011100101
Step 4     00001110010         (shift after add)
          +11100               (sub 4 = add -4)
           11101110010
Step 5     11110111001         (shift after add)

            1111011100 = -(0000100011 + 1)
                       = -(0000100100)
                       = -36 = 4 X -9!
```

Remember

```
 4 = 00100
-4 = 11100
```

BOOTH'S ALGORITHM EXAMPLE

Multiply -9 X -13

```
  10111
x 10011
```

```
           LHProd LHProd
            00000100110
Step 0     +01001              (sub -9 = add 9)
            01001100110
Step 1      00100110011        (shift after add)
Step 2      00010011001        (shift without add)
           +10111              (add -9)
Step 3      11001011001        (shift after add)
            11100101100
Step 4      11110010110
           +01001              (add 9)
            00111010110
Step 5      00011101011        (shift after add)
```

```
0001110101 = 64+32+16+4+1
           = 117
```

Remember

```
 9 = 01001
-9 = 10111
```

BOOTH'S ALGORITHM EXAMPLE

Multiply 4 X -9

```
   00100
 x 10111
```

Use 6-bit

Remember

```
 4 = 000100
-4 = 111100
```

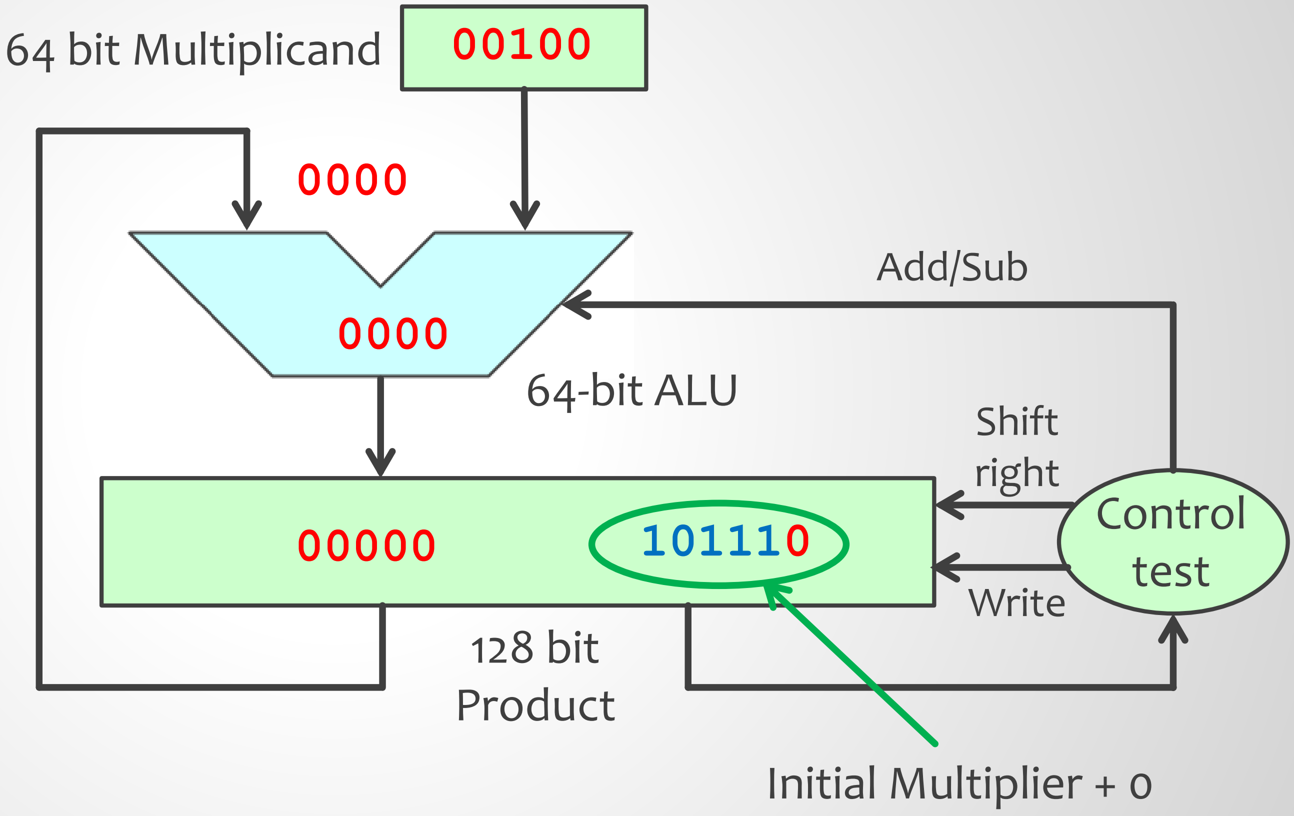
```
 000000101110
+111100               (sub 4 / add -4)
 111100101110
 111110010111         (shift after add)
 111111001011         (shift w/ no add)
 111111100101         (shift w/ no add)
+000100               (add +4)
 000011100101
 000001110010         (shift after add)
+111100               (sub 4 / add -4)
 111101110010
 111110111001
```

Drop leftmost & rightmost bit

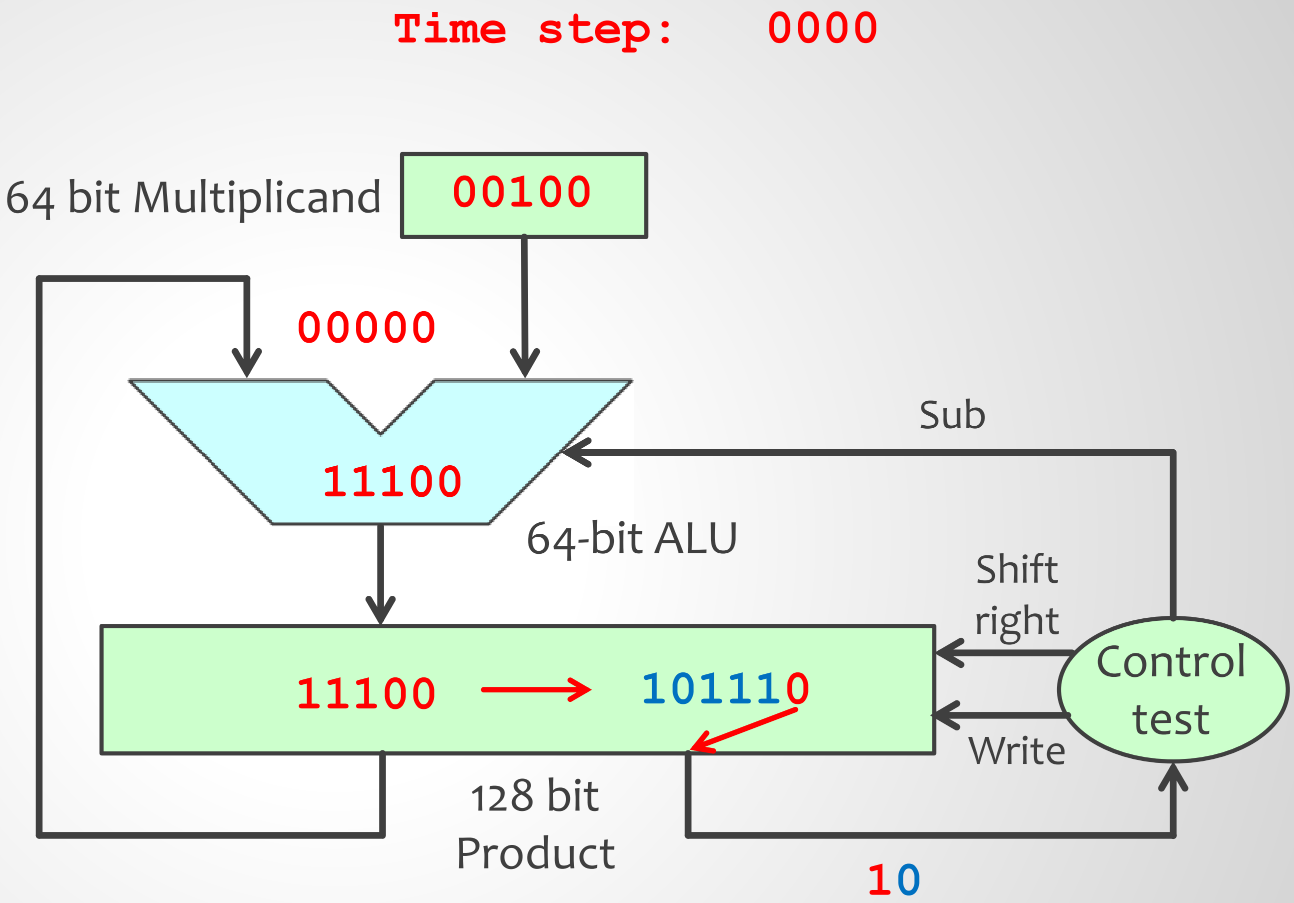
```
1111011100 = -(0000100011 + 1)
           = -(0000100100)
           = -36 = 4 X -9!
```

OPTIMIZED BOOTH MULTIPLIER

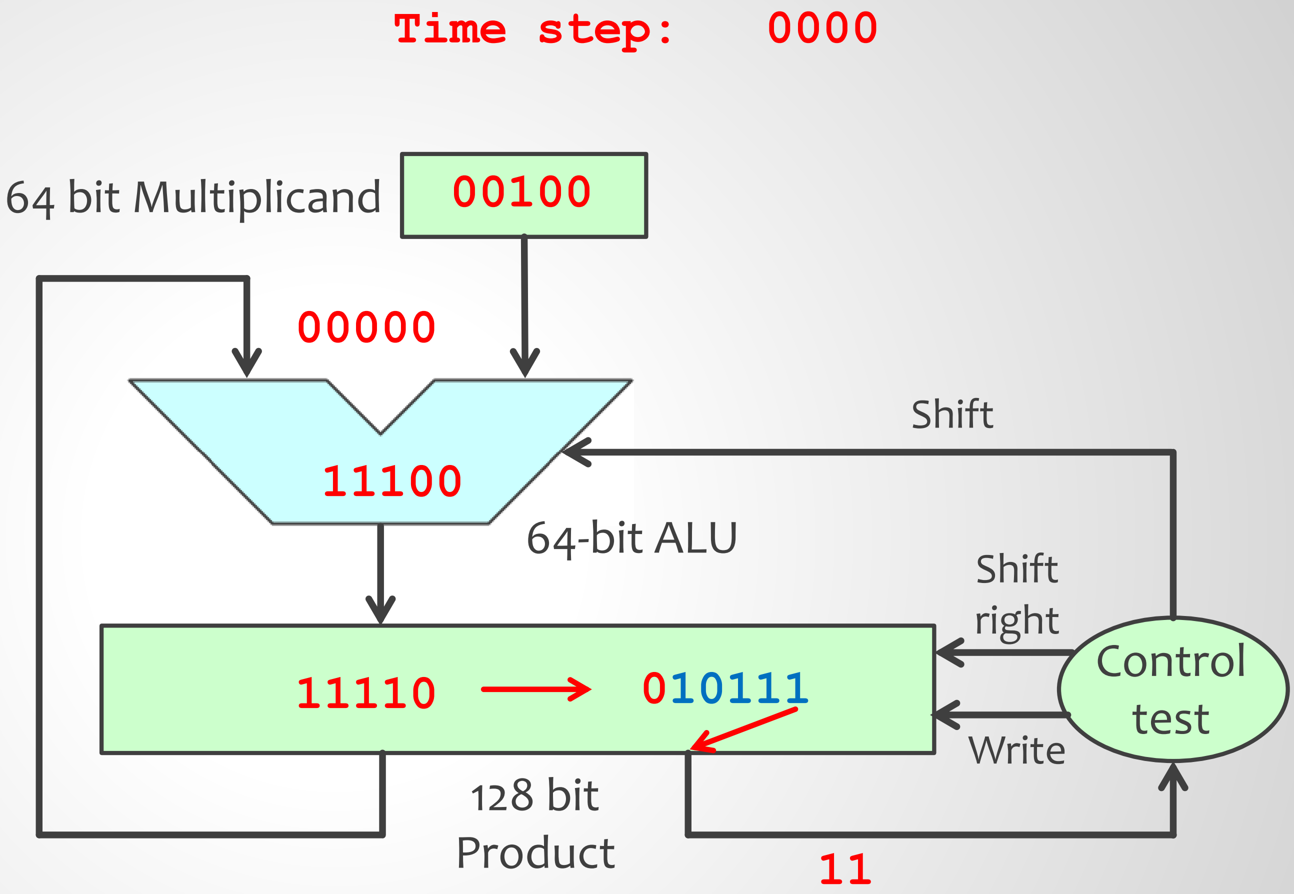
Time step: 0000



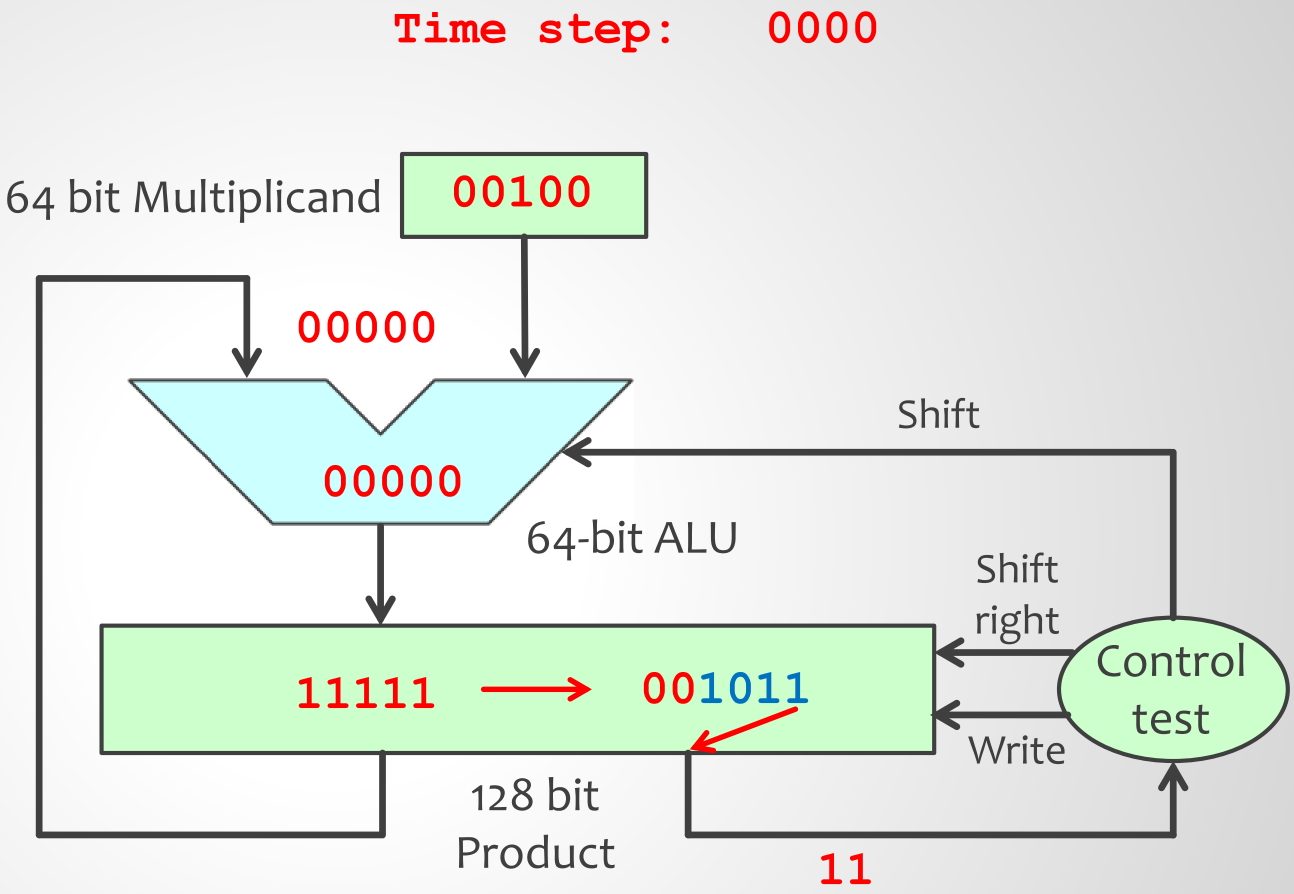
OPTIMIZED BOOTH MULTIPLIER



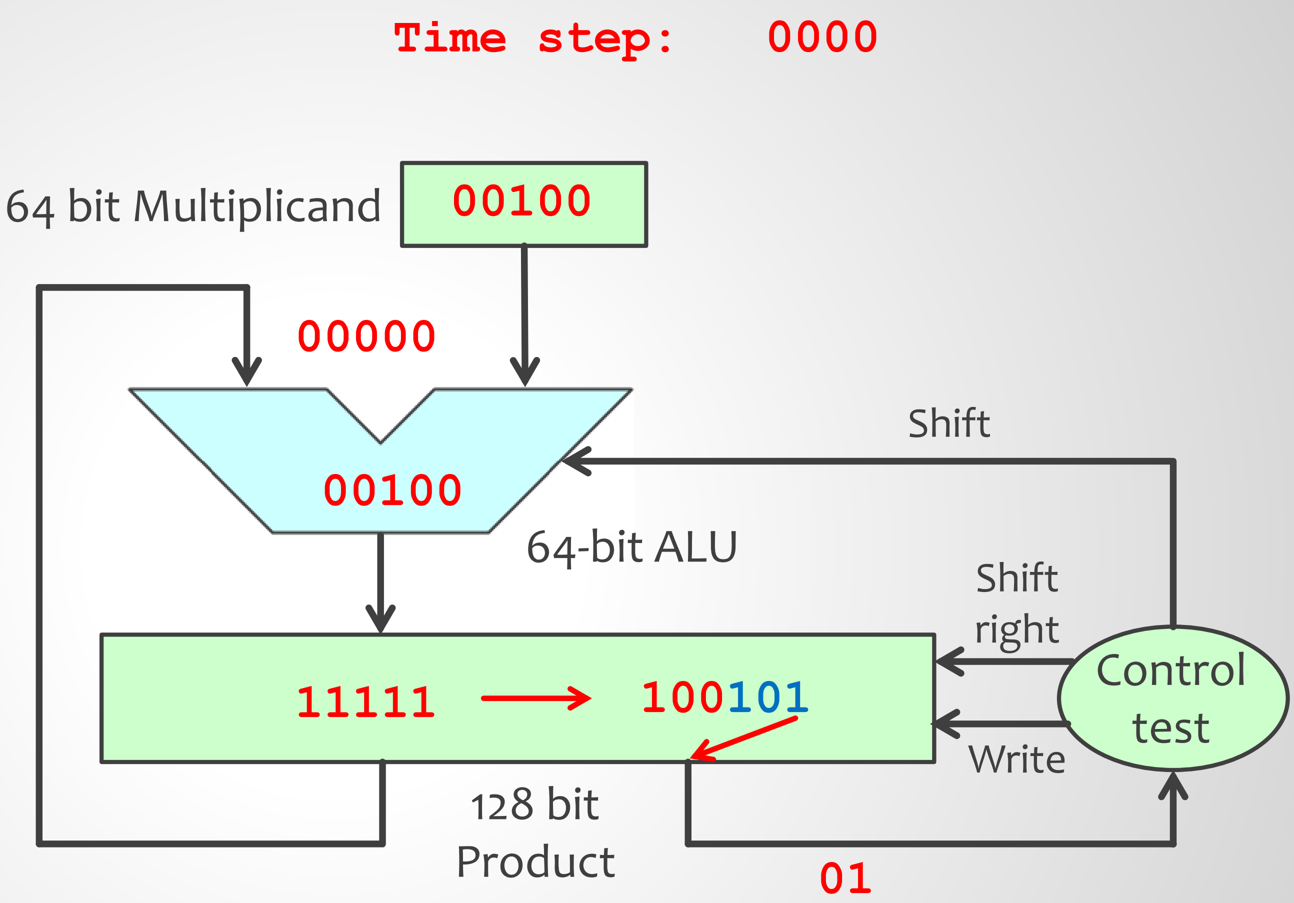
OPTIMIZED BOOTH MULTIPLIER



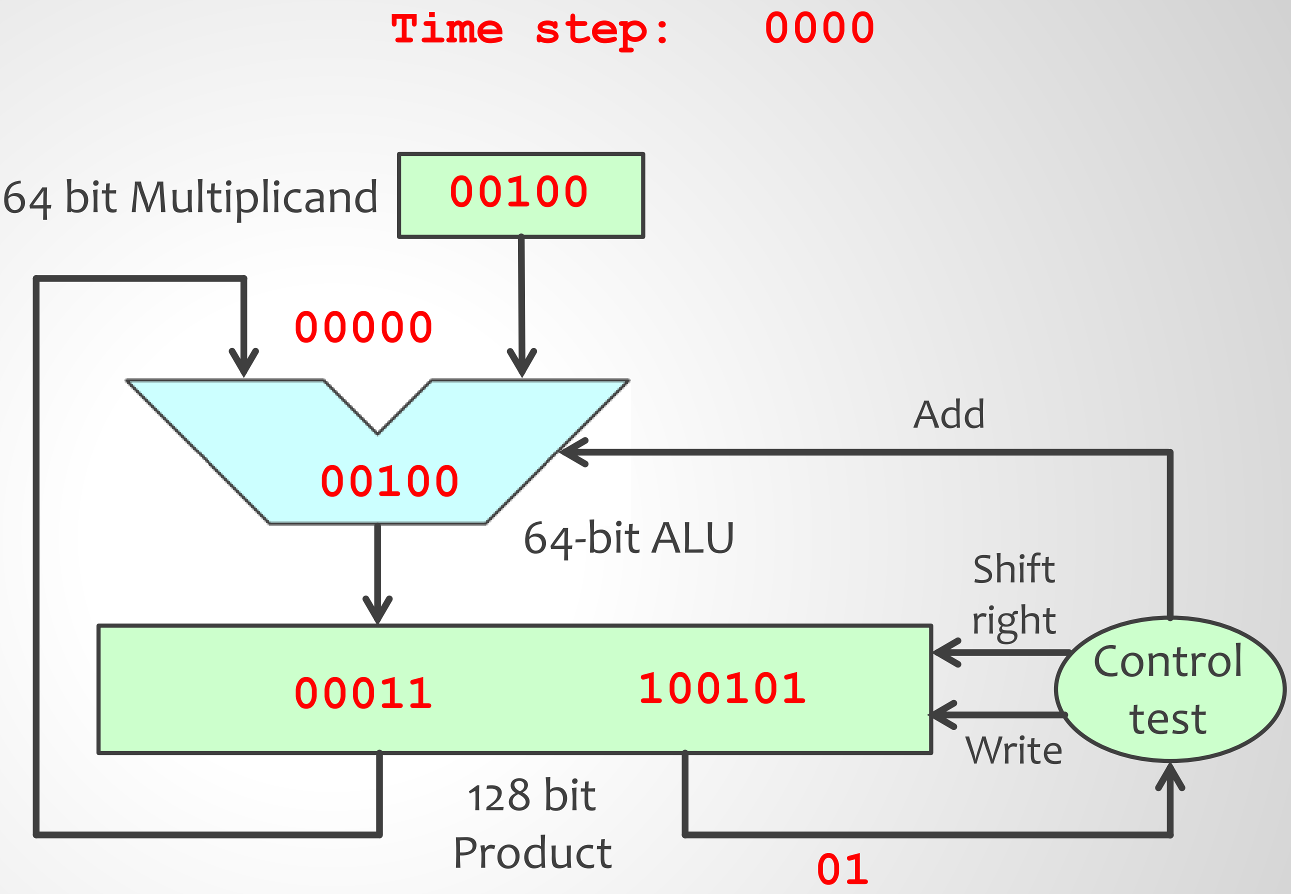
OPTIMIZED BOOTH MULTIPLIER



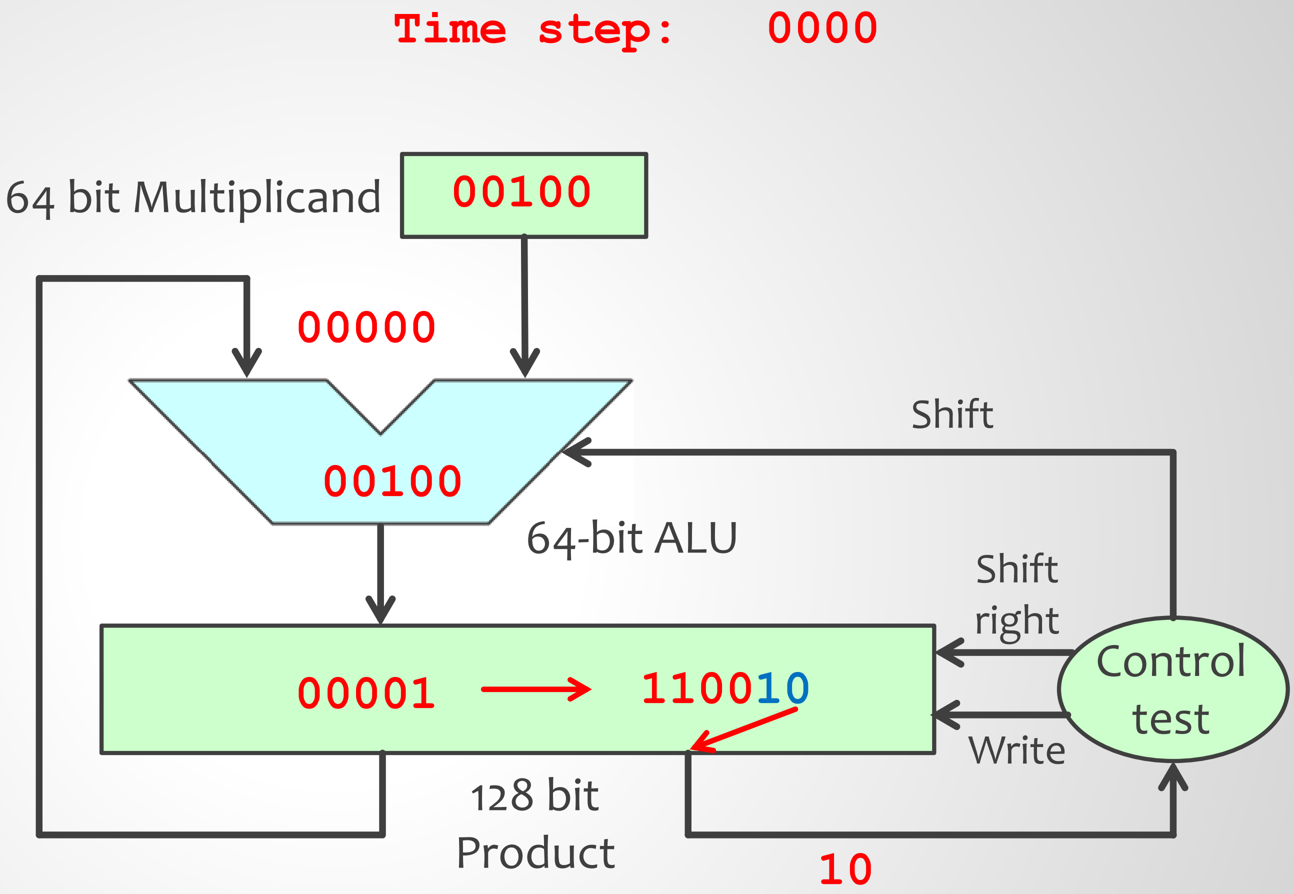
OPTIMIZED BOOTH MULTIPLIER



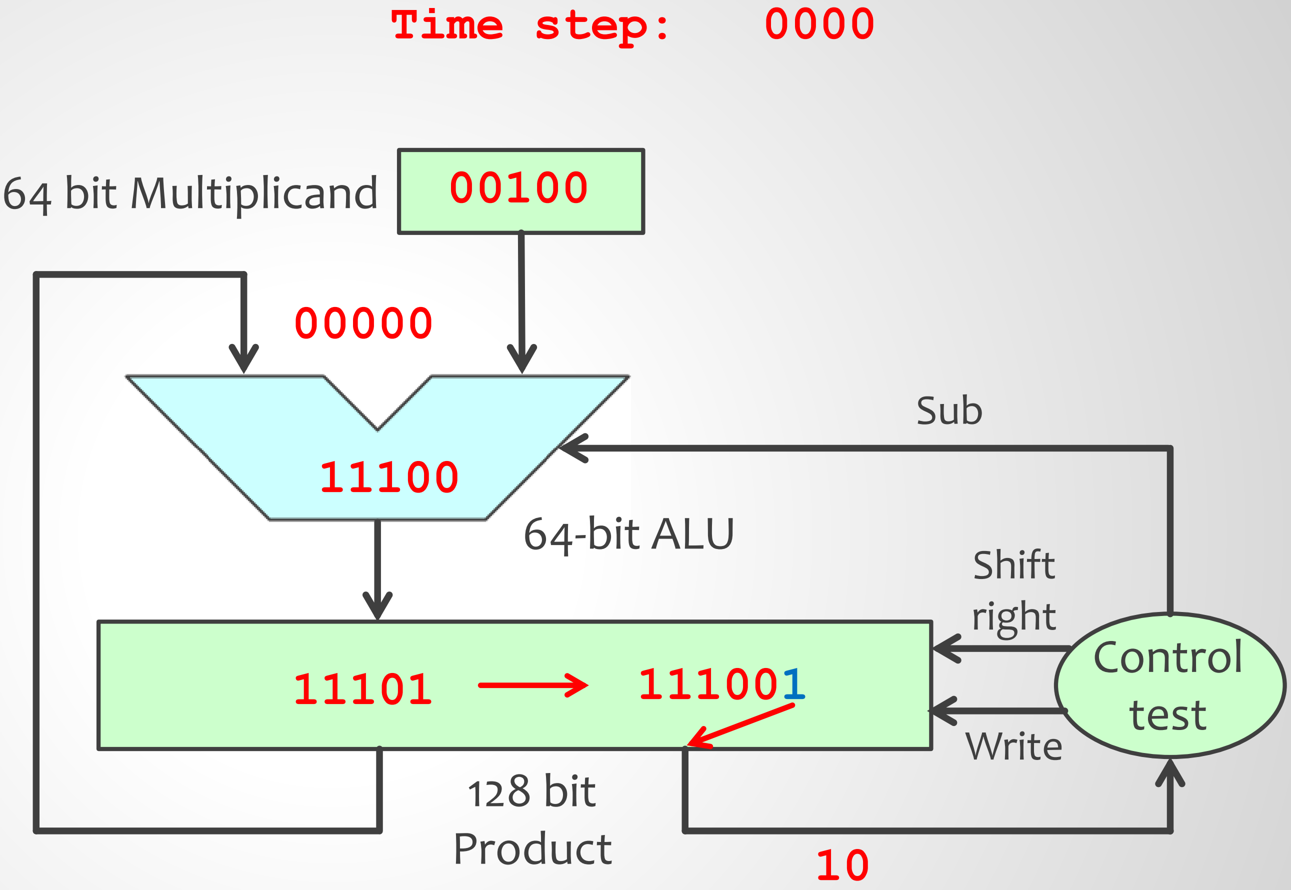
OPTIMIZED BOOTH MULTIPLIER



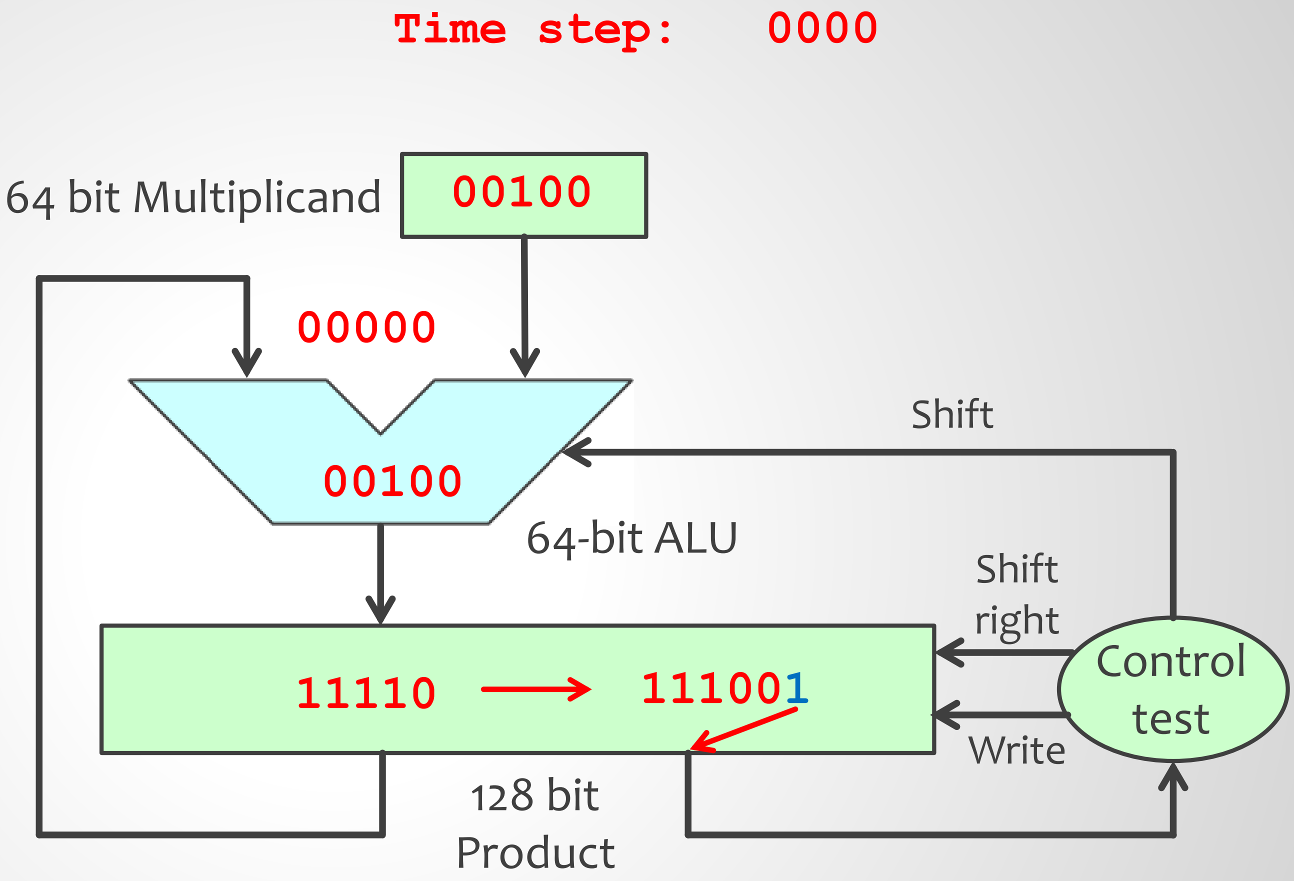
OPTIMIZED BOOTH MULTIPLIER



OPTIMIZED BOOTH MULTIPLIER



OPTIMIZED BOOTH MULTIPLIER



WHAT IS CORDIC?

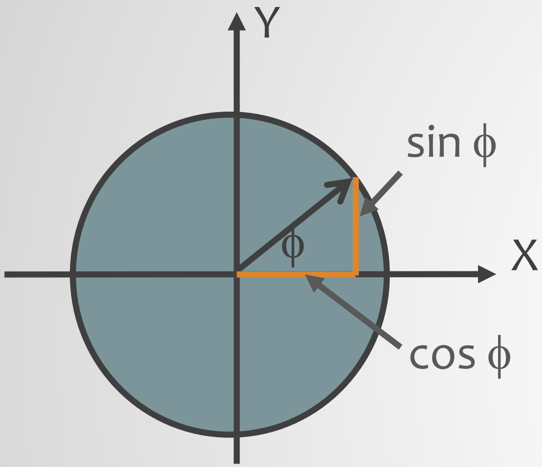
- *How to evaluate trigonometric functions?*
  - *Table lookup*
  - *Polynomial approximations*
  - *CORDIC*

- *CORDIC (COordinate Rotation DIgital Computer)*
  - *Introduced in 1959 by Jack E. Volder*
  - *Rotate vector (1,0) by ϕ to get (cos ϕ, sin ϕ)*
  - *Can evaluate many functions*
  - *Rotation reduced to shift-add operations*
  - *Convergence method (iterative)*
    - *N iterations for N-bit accuracy*
  - *Delay / hardware costs comparable to division or square rooting!*

BASIC CORDIC TRANSFORMATIONS

Basic idea

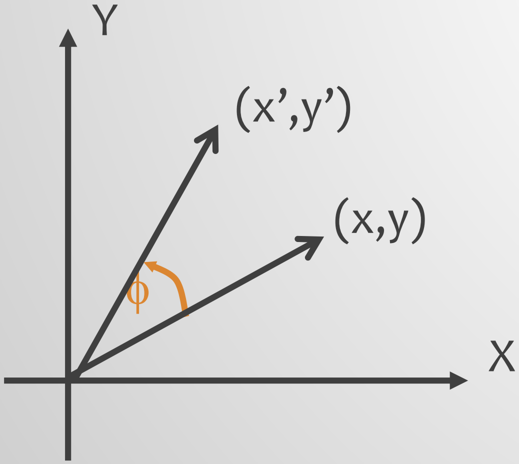
- *Rotate (1,0) by f degrees to get (x,y): x=cos(f), y=sin(f)*



- *Rotation of any (x,y) vector*

$$x' = x * \cos(\phi) - y * \sin(\phi)$$
$$y' = y * \cos(\phi) + x * \sin(\phi)$$

$$\text{Note}: \frac{\sin(\phi)}{\cos(\phi)} = \tan(\phi)$$



- *Rearrange as:*

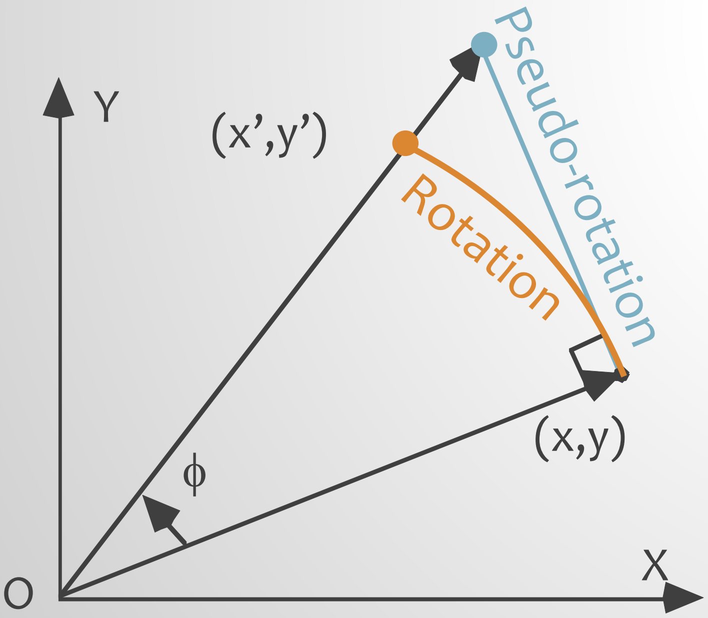
$$x' = \cos(\phi) * [x - y * \tan(\phi)]$$
$$y' = \cos(\phi) * [y + x * \tan(\phi)]$$

ROTATION AND MAGNITUDE COMPONENTS

$$x' = \cos(\phi) * [x - y * \tan(\phi)]$$

$$y' = \cos(\phi) * [y + x * \tan(\phi)]$$

- *Two components:*
  - *cos(ϕ)*
    - *Reduces the magnitude of the vector*
    - *If don't multiply* ➔ *pseudo rotation*
  - *tan(ϕ)*
    - *Rotates the vector*
    - *Break ϕ into a series of successively shrinking angles α_i such that:*
      tan (α_i) = 2^{-i} ⟵ Shift operation
    - *Should we use all α_i's?*



[© Oxford U Press]

PRE-COMPUTATION OF TAN(Αᵢ)

- *Find α_i such that tan(α_i) = 2^{-i} : (or, α_i = $\tan^{-1}(2^{-i})$)*

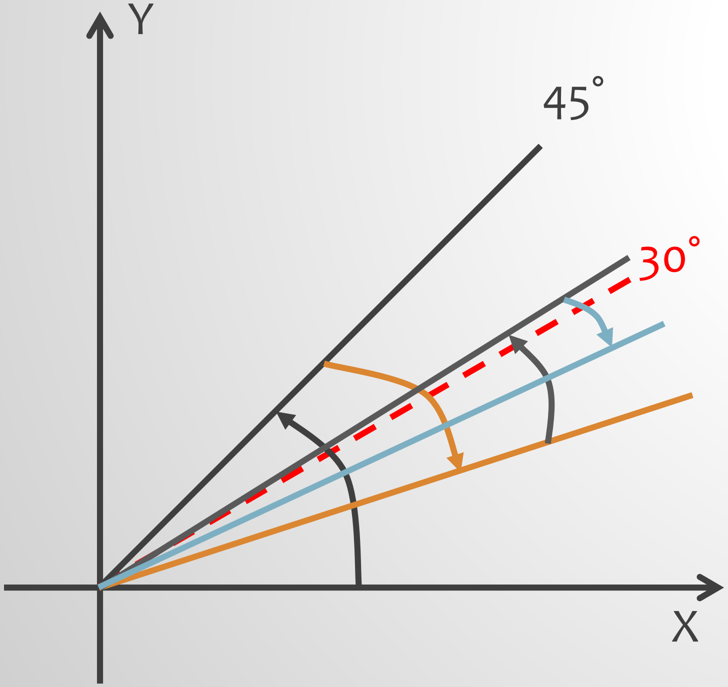
i	α_i	tan(α_i)	
0	45.0°	1	$= 2^{-0}$
1	26.6°	0.5	$= 2^{-1}$
2	14.0°	0.25	$= 2^{-2}$
3	7.1°	0.125	$= 2^{-3}$
4	3.6°	0.0625	$= 2^{-4}$
5	1.8°	0.03125	$= 2^{-5}$
6	0.9°	0.015625	$= 2^{-6}$
7	0.4°	0.0078125	$= 2^{-7}$
8	0.2°	0.00390625	$= 2^{-8}$
9	0.1°	0.001953125	$= 2^{-9}$

- *Note: decreasing α_i.*
  - *Possible to write any angle $\phi = \pm\alpha_0 \pm \alpha_1 \pm \ldots \pm \alpha_9$ as long as $-99.7^\circ \le \phi \le 99.7^\circ$ (which covers –90..90)*
  - *Convergence possible: $\alpha_i \le \Sigma^{N}_{j=i+1}\alpha_{i+1}$*
    - *10^{-5} deg accuracy*

EXAMPLE: REWRITING ANGLES IN TERMS OF A_I

- *Example:* ϕ*=30.0°*
  - *Start with* α_0 *= 45.0* (> 30.0)
  - *45.0 – 26.6 = 18.4* (< 30.0)
  - *18.4 + 14.0 = 32.4* (> 30.0)
  - *32.4 – 7.1 = 25.3* (< 30.0)
  - *25.3 + 3.6 = 28.9* (< 30.0)
  - *28.9 + 1.8 = 30.7* (> 30.0)
  - *30.7 – 0.9 = 29.8* (< 30.0)
  - *29.8 + 0.4 = 30.2* (> 30.0)
  - *30.2 – 0.2 = 30.0* (= 30.0)
  - *30.1 + 0.1 = 30.1* (> 30.0)

$$\begin{aligned}\phi &= 30.0\\ &\approx 45.0 - 26.6 + 14.0 - 7.1 + 3.6\\ &\quad + 1.8 - 0.9 + 0.4 - 0.2 + 0.1\\ &= 30.1\end{aligned}$$



WHY ANY ANGLE CAN BE FORMED FROM OUR LIST?

- ***Analogy:*** *Paying a certain amount while using all currency denominations (in positive or negative direction) exactly once; red values are fictitious.*

$20 $10 $5 $3 $2 $1 $.50 $.25 $.20 $.10 $.05 $.03 $.02 $.01

- ***Example:*** *Pay $12.50*

$20 – $10 + $5 – $3 + $2 – $1 – $.50 + $.25 – $.20 – $.10 + $.05 + $.03 – $.02 – $.01

- *Convergence is possible as long as each denomination is no greater than the sum of all denominations that follow it.*
- *Domain of convergence: –$42.16 to +$42.16*
- *We can guarantee convergence with actual denominations if we allow multiple steps at some values:*

$20 $10 $5 $2 $2 $1 $.50 $.25 $.10 $.10 $.05 $.01 $.01 $.01 $.01

- ***Example:*** *Pay $12.50*

$20 – $10 + $5 – $2 – $2 + $1 + $.50+$.25–$.10–$.10–$.05+$.01–$.01+$.01–$.01

- *We will see later that in hyperbolic CORDIC, convergence is guaranteed only if certain "angles" are used twice.*

ANGLE RECODING

- *The selection of angles during pseudorotations can be viewed as recoding the angle in a specific number system*

- *For example, an angle of 30 °is recoded as the following digit string, with each digit being 1 or –1:*

45.0	**26.6**	**14.0**	**7.1**	**3.6**	**1.8**	**0.9**	**0.4**	**0.2**	**0.1**
1	**–1**	**1**	**–1**	**1**	**1**	**–1**	**1**	**–1**	**1**

- *The money-exchange analogy also lends itself to this recoding view*

- *For example, a payment of $12.50 is recoded as:*

$20	**$10**	**$5**	**$3**	**$2**	**$1**	**$.50**	**$.25**	**$.20**	**$.10**	**$.05**	**$.03**	**$.02**	**$.01**
1	**–1**	**1**	**–1**	**1**	**–1**	**–1**	**1**	**–1**	**–1**	**1**	**1**	**–1**	**–1**

ROTATION REDUCTION

- *Rewrite in terms of α_i: ($0 \le i \le n$)*

$$x' = \cos(\phi) * [x - y * \tan(\phi)]$$

$$y' = \cos(\phi) * [y + x * \tan(\phi)]$$

- *Where:*

$$x_{i+1} = \cos(\alpha_i) * [x_i - y_i * d_i * \tan(\alpha_i)]$$

$$y_{i+1} = \cos(\alpha_i) * [y_i + x_i * d_i * \tan(\alpha_i)]$$

➔

$$\boxed{\begin{aligned} x_{i+1} &= K_i * [x_i - y_i * d_i * 2^{-i}] \\ y_{i+1} &= K_i * [y_i + x_i * d_i * 2^{-i}] \end{aligned}}$$

- *What about K_i's?*

$$K_i = \cos(\alpha_i) = \cos(\tan^{-1}(2^{-i}))$$

$$d_i = \pm 1$$

$$\text{Note:}$$

$$\cos(\alpha_i) = \cos(-\alpha_i)$$

TAKING CARE OF THE MAGNITUDE

- *Observations:*
  - *We choose to always use ALL α_i terms, with +/- signs*
  - $K_i = \cos(\alpha_i) = \cos(-\alpha_i)$
  - *At each step, we multiply by* $\cos(\alpha_i)$ *[constant?]*
- *Let the multiplications aggregate to:*

$$x_{i+1} = K_i * [x_i - y_i * d_i * 2^{-i}]$$
$$y_{i+1} = K_i * [y_i + x_i * d_i * 2^{-i}]$$

  - *Multiply this constant ONLY ONCE at the end*

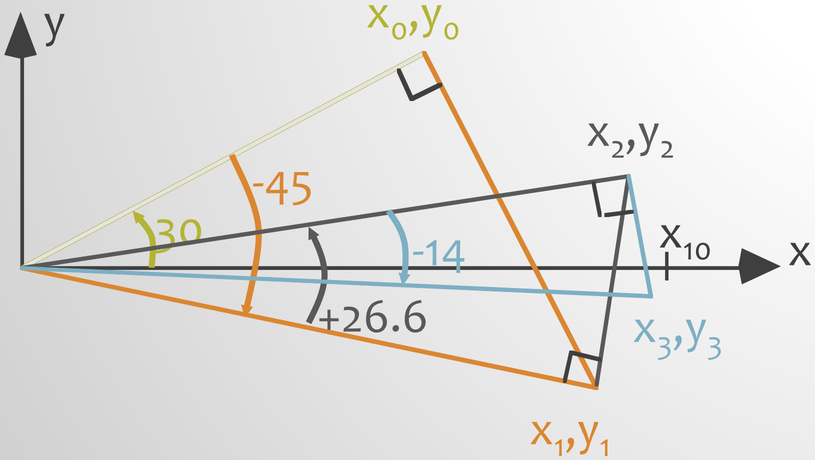
$$K = \prod_{i=0}^{n} K_i \quad n \to \infty,\ K = 0.607\,252\,935...$$

Even better: start with (K,0) instead of (1,0)
➔ no multiplication at all!

HARDWARE REALIZATION: CORDIC ROTATION MODE

Similar to non-restoring sqrt

- *To simplify the hardware:*
  - *First rotate by ϕ, then rotate by $-d_i*\alpha_i$ to get 0 (no subtraction to compare ϕ & current angle)*
- *Algorithm: (z is the current angle)*
  - *Mode: rotation: "at each step, try to make z zero"*
  - Initialize $x=0.607253$, $y=0$, $z=\phi$
  - For $i = 0 \rightarrow n$
  - $d_i = 1$ when $z > 0$, else $d_i = -1$
  - $x_{i+1} = x_i - d_i * 2^{-i} * y_i$
  - $y_{i+1} = y_i + d_i * 2^{-i} * x_i$
  - $z_{i+1} = z_i - d_i * \alpha_i$
  - *Result: $x_n = \cos(\phi)$, $y_n = \sin(\phi)$*
  - *Precision: n bits ($\tan^{-1}(2^{-i}) \approx 2^{-i}$)*



[© Oxford U Press]

CORDIC ROTATION MODE C CODE

```
// downloaded (and modified by Kia) from
// www.execpc.com/~geezer/embed/cordic.c
#include <stdio.h>
#include <math.h>

#define  AG_CONST    0.6072529350
#define  FXD(X)      ((long int)((X) * 65536.0))

typedef long int fixed;       /* 16.16 fixed-point */

static const fixed Alpha[] = { FXD(45.0), FXD(26.565),
FXD(14.0362), FXD(7.12502), FXD(3.57633), FXD(1.78991),
FXD(0.895174), FXD(0.447614), FXD(0.223811), FXD(0.111906),
FXD(0.055953), FXD(0.027977) };

int main(void){
  fixed X, Y, CurrAngle;
  unsigned i;

  X = FXD(AG_CONST);            /* AG_CONST * cos(0) */
  Y = 0;                        /* AG_CONST * sin(0) */
  CurrAngle=FXD(28.027);
  for (i = 0; i < 12; i++){
     fixed NewX;

     if (CurrAngle > 0) {
        NewX=X - (Y >> i);
        Y += (X >> i);
        X = NewX;
        CurrAngle -= Alpha[i]; }
     else {
        NewX = X + (Y >> i);
        Y- = (X >> i);
        X = NewX;
        CurrAngle += Alpha[i];
     } // if-else
  } // for (i = …
  printf(“cos(28.027)=%6.4f, sin()=%6.4f\n”, x/65536.0, y/65536.0);
} // main
```

Only used:
>0, >>, +
Integer ops

USING CORDIC IN ROTATION MODE

$$x^{(i+1)} = x^{(i)} - d_i\, y^{(i)}\, 2^{-i}$$

$$y^{(i+1)} = y^{(i)} + d_i\, x^{(i)}\, 2^{-i}$$

$$\left.\begin{aligned} z^{(i+1)} &= z^{(i)} - d_i \tan^{-1} 2^{-i} \\ &= z^{(i)} - d_i\, e^{(i)} \end{aligned}\right\}$$

Make z converge to 0 by choosing $d_i = \mathrm{sign}(z^{(i)})$

$$x^{(m)} = \cancel{K}(\cancel{x} \cos z - \overset{0}{\cancel{y \sin z}})$$

$$y^{(m)} = \cancel{K}(\underset{0}{\cancel{y \cos z}} + \cancel{x} \sin z)$$

$$z^{(m)} = 0$$

where $K = 1.646\ 760\ 258\ 121\ \ldots$

- *For k bits of precision in results, k CORDIC iterations are needed, because* $\tan^{-1} 2^{-i} \cong 2^{-I}$ *for large I*
- *Start with* $x = 1/K = 0.607\ 252\ 935\ \ldots$ *and* $y = 0$ *to find cos z and sin z*
- *Convergence of z to 0 is possible because each of the angles in our list is more than half the previous one or, equivalently, each is less than the sum of all the angles that follow it*
- *Domain of convergence is* $-99.7° \le z \le 99.7°$*, where 99.7° is the sum of all the angles in our list; the domain contains* $[-\pi/2, \pi/2]$ *radians*

USING CORDIC IN VECTORING MODE

$$x^{(i+1)} = x^{(i)} - d_i\, y^{(i)}\, 2^{-i}$$

$$\left.\begin{aligned} y^{(i+1)} &= y^{(i)} + d_i\, x^{(i)}\, 2^{-i} \end{aligned}\right\}$$

Make *y* converge to 0 by choosing $d_i = -\text{sign}(x^{(i)} y^{(i)})$

$$z^{(i+1)} = z^{(i)} - d_i \tan^{-1} 2^{-i}$$

$$= z^{(i)} - d_i\, e^{(i)}$$

$$x^{(m)} = K(x^2 + y^2)^{1/2}$$

$$y^{(m)} = 0$$

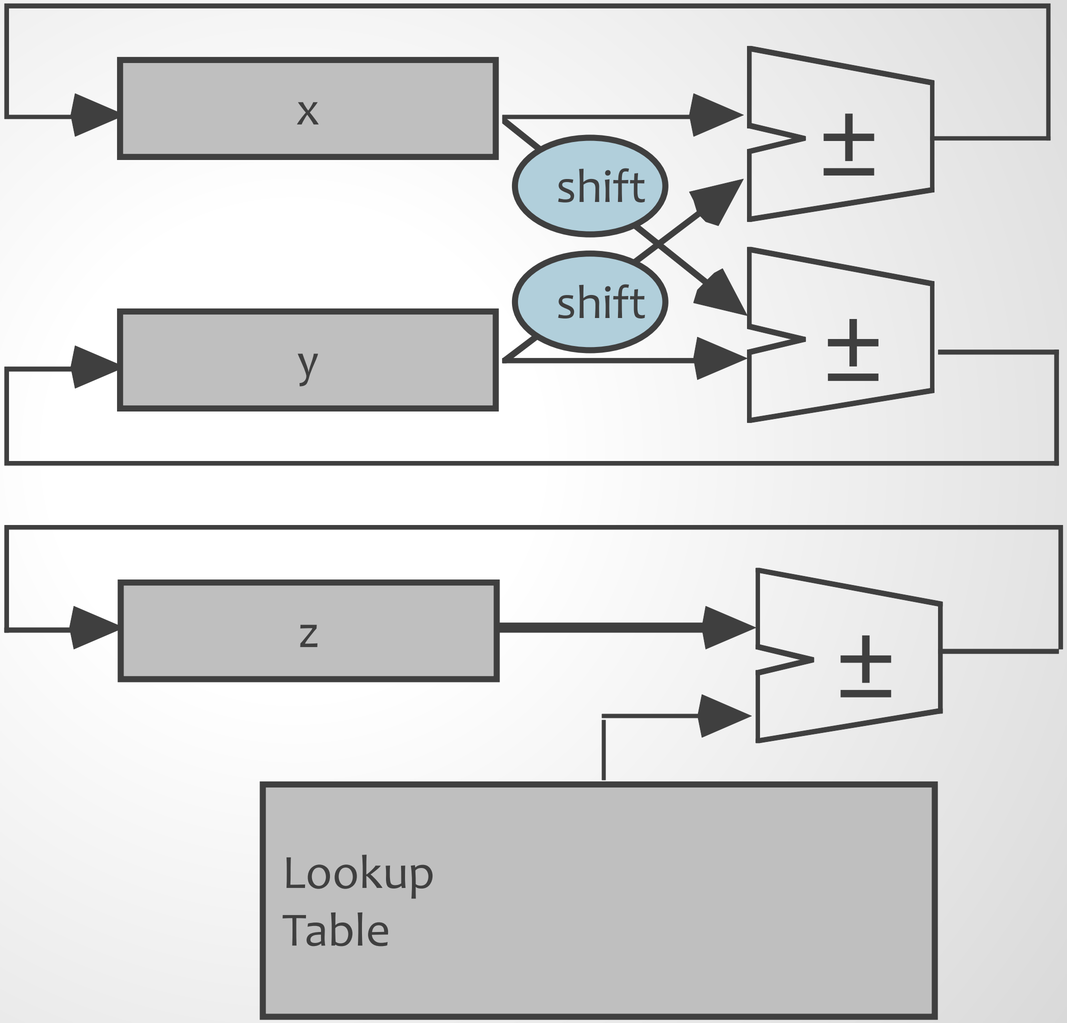
$$z^{(m)} = \underset{0}{\cancel{z}} + \tan^{-1}(y/\cancel{x})$$

where $K = 1.646\ 760\ 258\ 121\ \ldots$

- *For k bits of precision in results, k CORDIC iterations are needed, because* $\tan^{-1} 2^{-i} \cong 2^{-i}$ *for large I*
- *Start with* $x = 1$ *and* $z = 0$ *to find* $\tan^{-1} y$
- *Even though the computation above always converges, one can use the relationship* $\tan^{-1}(1/y) = \pi/2 - \tan^{-1} y$ *to limit the range of fixed-point numbers encountered*
- *Other trig functions: tan z obtained from sin z and cos z via division; inverse sine and cosine (*$\sin^{-1} z$ *and* $\cos^{-1} z$*) discussed later*

CORDIC HARDWARE

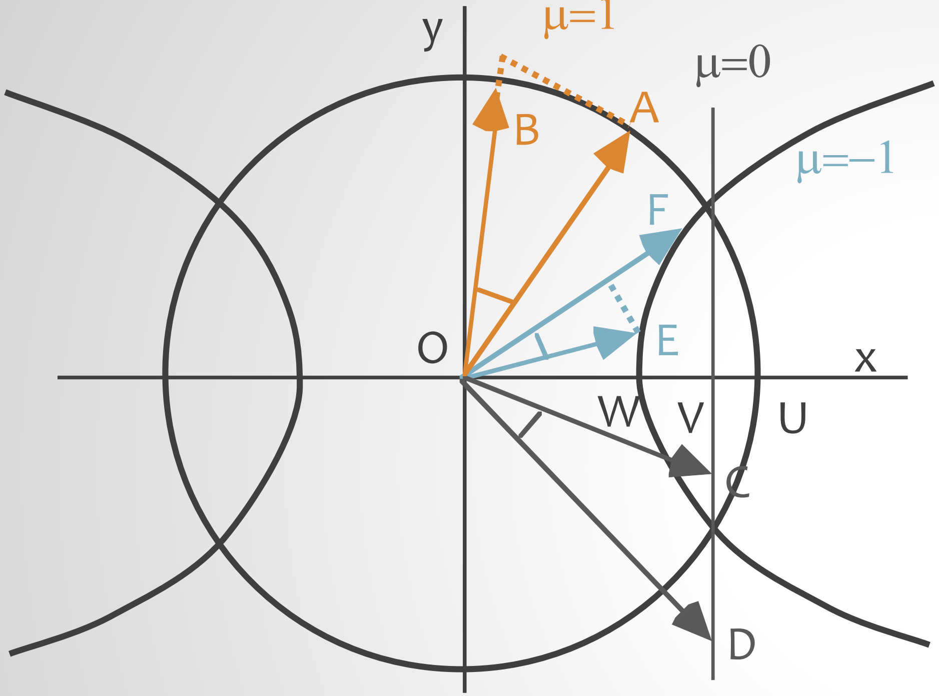
- *[© Oxford U Press]*



CORDIC VECTORING MODE

- *Difference with rotation mode?*
  - *When choosing d_i, instead of trying to make z converge to 0, try to make y_i zero*
  - $d_i = -\text{sign}(x_i * y_i)$
- *Variables will converge to:*
  - $x_n = 1 / K (x^2 + y^2)^{½}$
  - $y_n = 0$
  - $z_n = z + \tan^{-1}(y / x)$
- *Application?*
  - *If start with x = 1, z = 0, the final*
  - $z = \tan^{-1}(y)$

GENERALIZED CORDIC



- *Generalized CORDIC iteration:*
  - $x_{i+1} = x_i - \mu * d_i * 2^{-i} * y_i$
  - $y_{i+1} = y_i + d_i * 2^{-i} * x_i$
  - $z_{i+1} = z_i - d_i * e(i)$
- *Variations:*
  - $\mu = 1$ - *Circular rotations (basic CORDIC)* $e^{(i)} = \tan^{-1} 2^{-i}$
  - $\mu = 0$ - *Linear rotations* $e^{(i)} = 2^{-i}$
  - $\mu = -1$ - *Hyperbolic rotations* $e^{(i)} = \tanh^{-1} 2^{-i}$

μ	Function	e(i)
1	Circular rotation (basic CORDIC)	$\tan^{-1}(2^{-i})$
0	Linear rotation	2^{-i}
-1	Hyperbolic rotation	$\tanh^{-1}(2^{-i})$

[© Oxford U Press]

VARIOUS CORDIC APPLICATIONS

- *Directly computes:*
  - *sin, cos, sinh, cosh*
  - tan^{-1}, $tanh^{-1}$
  - *Division, multiplication*
- *Also directly computes:*
  - $tan^{-1}(y/x)$
  - $y + x * z$
  - $(x^2 + y^2)^{1/2}$
  - $(x^2 - y^2)^{1/2}$
  - $e^Z = sinh(z) + cosh(z)$

[© Wilde]

VARIOUS CORDIC APPLICATIONS (CONT.)

- *Indirectly computes:*

$$\tan z = \frac{\sin z}{\cos z}$$

$$\tanh z = \frac{\sinh z}{\cosh z}$$

$$\ln w = 2 \tanh^{-1} \left| \frac{w-1}{w+1} \right|$$

$$\log_b w = K . \ln w$$

$$w^t = e^{t \ln w}$$

$$\cos^{-1} w = \tan^{-1} \frac{\sqrt{1-w^2}}{w}$$

$$\sin^{-1} w = \tan^{-1} \frac{w}{\sqrt{1-w^2}}$$

$$\cosh^{-1} w = \ln \left(w + \sqrt{1-w^2} \right)$$

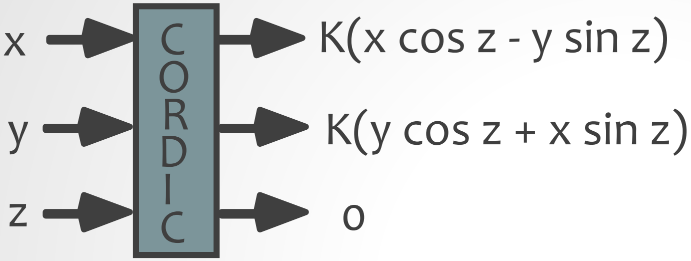
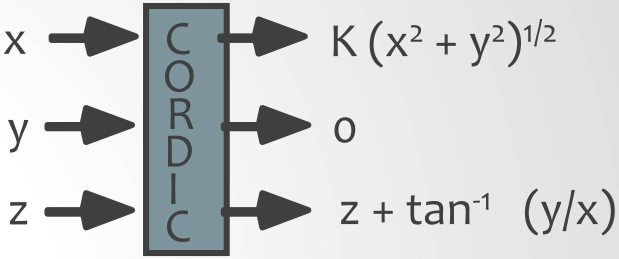
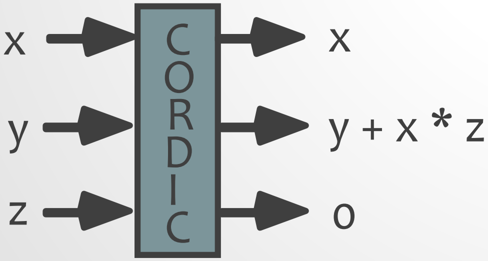
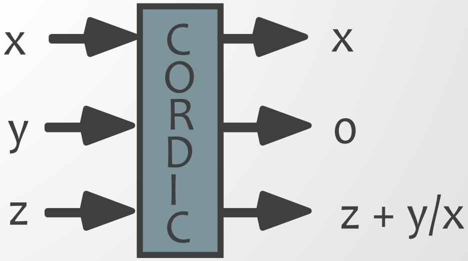
$$\sinh^{-1} w = \ln \left(w + \sqrt{1+w^2} \right)$$

$$\sqrt{w} = \sqrt{(w+1/4)^2 - (w-1/4)^2}$$

[© Wilde]

SUMMARY OF CORDIC APPLICATIONS

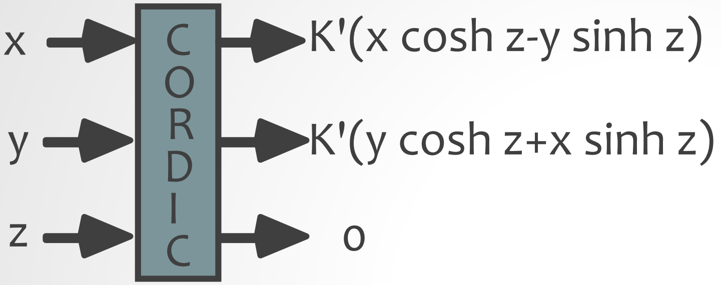
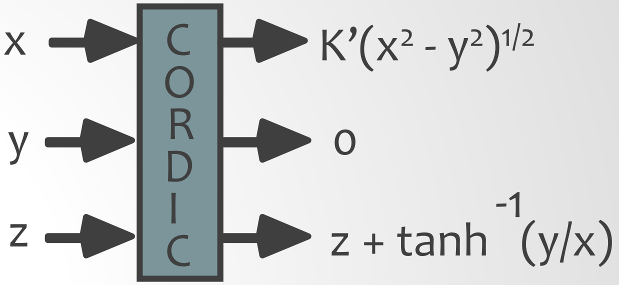
[© Oxford U Press & Paroo, p. 371]

	Rotation mode d_i=sign(z_i), $z_i \to 0$	Vectoring mode d_i=-sign($x_i y_i$), $y_i \to 0$
μ = 1 Circular e(i) = $\tan^{-1}2^{-i}$	 $K(x \cos z - y \sin z)$ $K(y \cos z + x \sin z)$ 0 For cos & sin, set x=1/K, y=0 tan z = sin z / cos z	 $K (x^2 + y^2)^{1/2}$ 0 $z + \tan^{-1}\ (y/x)$ For $\tan^{-1}$, set x = 1, z = 0 $\cos^{-1}w = \tan^{-1}[(1-w^2)^{1/2}/w]$ $\sin^{-1}w = \tan^{-1}[w/(1-w^2)^{1/2}]$
μ = 0 Linear e(i)= 2^{-i}	 x y + x * z 0 For multiplication, set y = 0	 x 0 z + y/x For division, set z = 0

Note: in linear mode, limited input range (convergence)

(CONT.)

- *[© Oxford U Press] & Par00], p. 371*

	Rotation mode d_i=sign(z_i), $z_i \rightarrow 0$	Vectoring mode d_i=-sign($x_i y_i$), $y_i \rightarrow 0$
μ = -1 Hyperbolic e(i) = $\tanh^{-1} 2^{-i}$	 x y z CORDIC K'(x cosh z-y sinh z) K'(y cosh z+x sinh z) 0 For cosh & sinh, set x=1/K', y=0 tanh z = sinh z / cosh z exp(z) = sinh z + cosh z w^t = exp(t ln w)	 x y z CORDIC K'(x² - y²)^1/2 0 z + tanh^-1(y/x) For $\tanh^{-1}$ set x = 1, z = 0 ln w=2 $\tanh^{-1}$ |(w - 1)/(w + 1)| $w^{1/2}=[(w+1/4)^2 - (w-1/4)^2]^{1/2}$ $\cosh^{-1} w = \ln(w + (1-w^2)^{1/2})$ $\sinh^{-1} w = \ln(w + (1+w^2)^{1/2})$

In the μ = -1 case, steps 4, 13, 40, 121, ... , j , 3j + 1, ... must be repeated for the method to converge. These repetitions are incorporated in the constant K' below.

$$x_{i+1} = x_i - \mu * d_i * 2^{-i} * y_i$$
$$y_{i+1} = y_i + d_i * 2^{-i} * x_i$$
$$z_{i+1} = z_i - d_i * e(i)$$

$\mu \in \{-1,0,1\}$, di $\in \{-1,1\}$

K = 1.646 760 258 121...

K'= 0.828 159 360 960 2...

CORDIC SPEEDUP METHODS

$$x^{(i+1)} = x^{(i)} - \mu\, d_i y^{(i)} 2^{-i}$$
$$y^{(i+1)} = y^{(i)} + d_i\, x^{(i)} 2^{-i}$$
$$z^{(i+1)} = z^{(i)} - d_i\, e^{(i)}$$

- *Skipping some rotations*
  - *Must keep track of expansion via the recurrence:*
  - $(K_{(i+1)})^2 = (K_{(i)})^2\, (1 \pm 2^{-2i})$
  - *This additional work makes variable-factor CORDIC less cost-effective than constant-factor CORDIC*

$$x^{(k)} = x^{(k/2)} - y^{(k/2)} z^{(k/2)}$$
$$y^{(k)} = y^{(i)} + x^{(k/2)} z^{(k/2)}$$
$$z^{(k)} = z^{(k/2)} - z^{(k/2)}$$

- *Early termination*
  - *Do the first k/2 iterations as usual, then combine the remaining k/2 into a single multiplicative step:*
  - *For very small z, we have* $\tan^{-1} z \simeq z \simeq \tan z$
  - *Expansion factor not an issue because contribution of the ignored terms is provably less than ulp*

$d_i \in \{-2, -1, 1, 2\}$ or
$\{-2, -1, 0, 1, 2\}$

- *High-radix CORDIC*

REFERENCES AND COPYRIGHT

- *Kia Bazargan*
  - *EE 5324 – VLSI Design II, University of Minnesota*
- *Textbook referenced*
  - *[Par00] B. Parhami "Computer Arithmetic: Algorithms and Hardware Designs" Oxford University Press, 2000.*
  - *This presentation is intended to support the use of the textbook Computer Arithmetic: Algorithms and Hardware Designs (Oxford U. Press, 2nd ed., 2010, ISBN 978-0-19-532848-6). It is updated regularly by the author as part of his teaching of the graduate course ECE 252B, Computer Arithmetic, at the University of California, Santa Barbara. Instructors can use these slides freely in classroom teaching and for other educational purposes. Unauthorized uses are strictly prohibited. © Behrooz Parhami*
- *Slides used(Modified by Kia when necessary)*
  - *[©Oxford U Press] © Oxford University Press, New York, 2000. http://www.ece.ucsb.edu/Faculty/Parhami/files_n_docs.htm*
  - *[©Wilde] © Prof. Doran Wilde, Lecture notes on the Computer Arithmetic Course, ECE Dept., Brigham Young University*

TO PROBE FURTHER…

- *Tutorials*
  - *http://cnmat.cnmat.berkeley.edu/~norbert/cordic/node3.html*
  - *http://www.execpc.com/~geezer/embed/cordic.htm (including C code)*
  - *http://bsvi.ru/uploads/CORDIC--_10EBA/cordic.pdf*
- *Papers*
  - *Survey paper on FPGA implementation of CORDIC algorithms: http://www.andraka.com/files/crdcsrvy.pdf*
  - *http://www.taygeta.com/cordic_refs.html*
- *Hardware implementations*
  - *http://www.free-ip.com/cordic/*
  - *http://www.stanford.edu/~chet/cordic.html*

CONCLUDING REMARKS

- *This Lecture:*
  - *Algorithms*
- *Summary*
  - *Useful on embedded or specialty applications to meet performance requirements*
- *Next Lecture:*
  - *Project*
- *Questions?*