Chapter 3
Arithmetic for Computers

Da-Wei Chang, OSES Lab.
CSIE Dept., NCKU

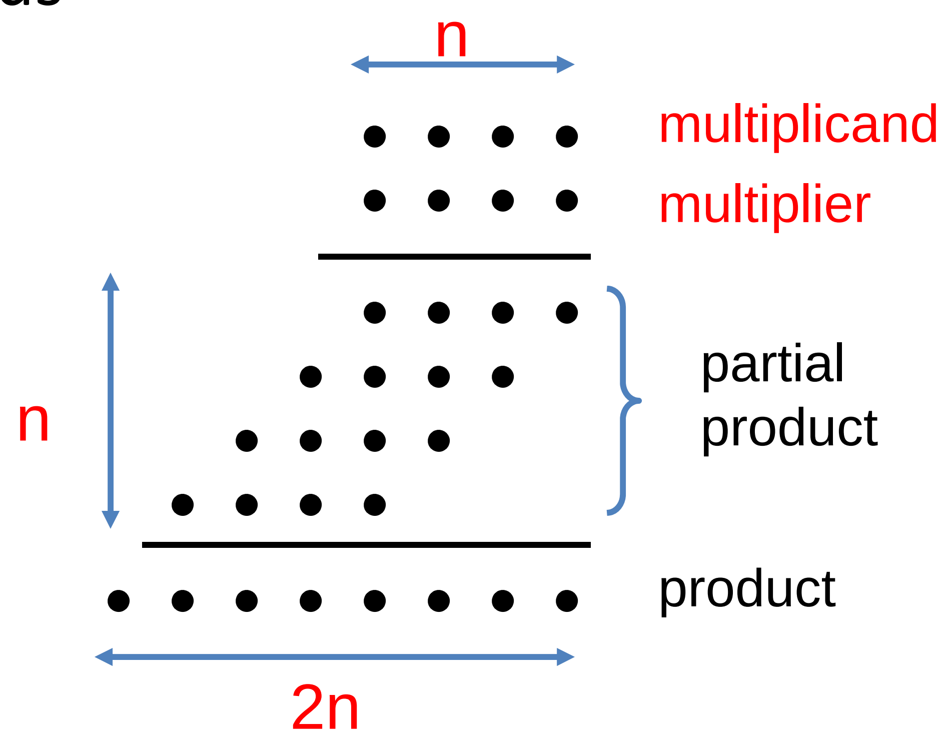
Overview

- We have described add/sub (and overflow detection) before
- Multiplication
- Division
- Floating point operations

Multiplication

- Binary multiplication is just a *bunch* of right shifts and adds

```
    0010
×   0011
--------
    0010
   0010
  0000
 0000
--------
 0000110
```



Multiplication

- Start with long-multiplication approach

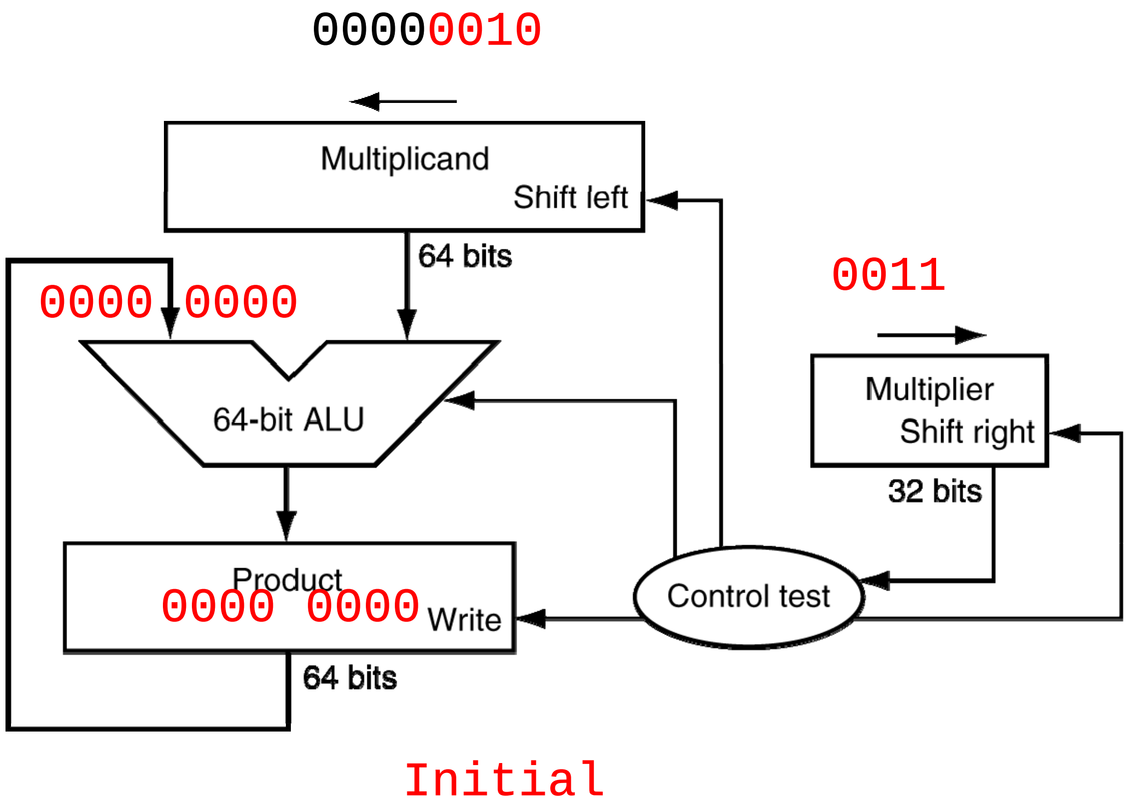
multiplicand

multiplier

```
     0010
  ×  0011
  -------
     0010
    0010
   0000
  000
  -------
  0000110
```

product

Length of product is the sum of operand lengths



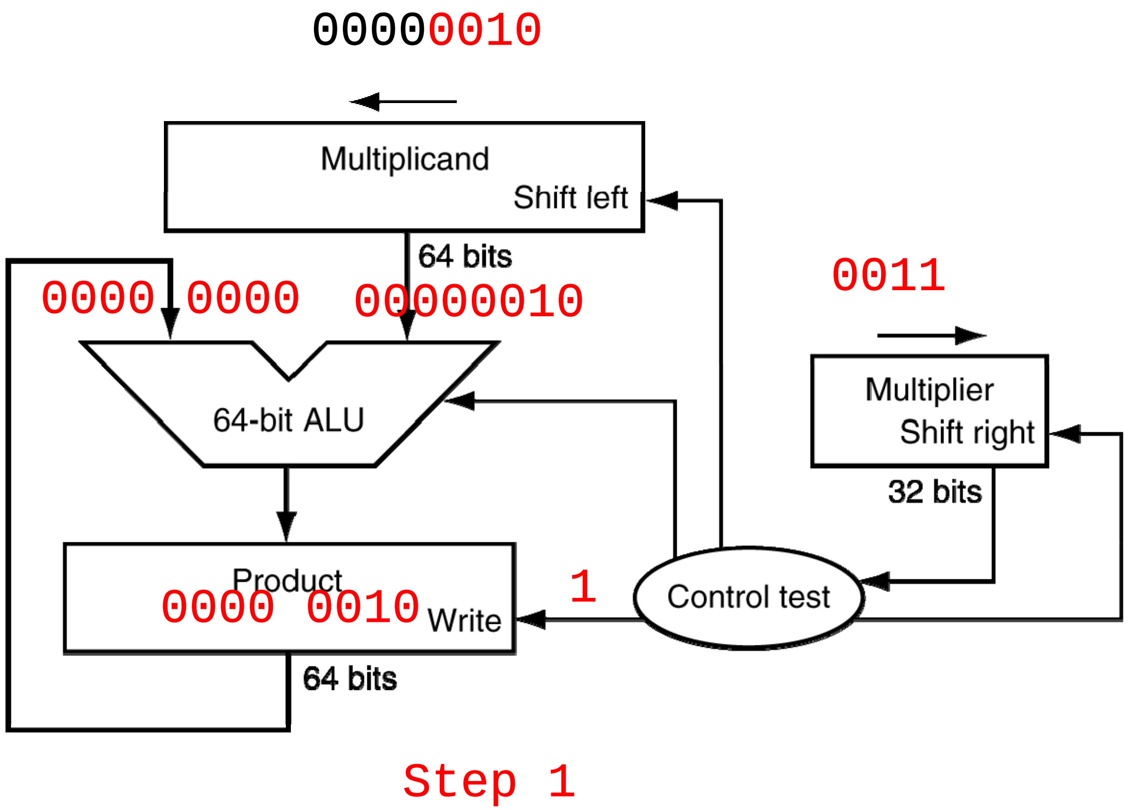
Multiplication

- Start with long-multiplication approach

multiplicand
multiplier
product

```
      0010
  ×   0011
  --------
      0010
     0010
    0000
   000
  --------
   0000110
```

Length of product is the sum of operand lengths



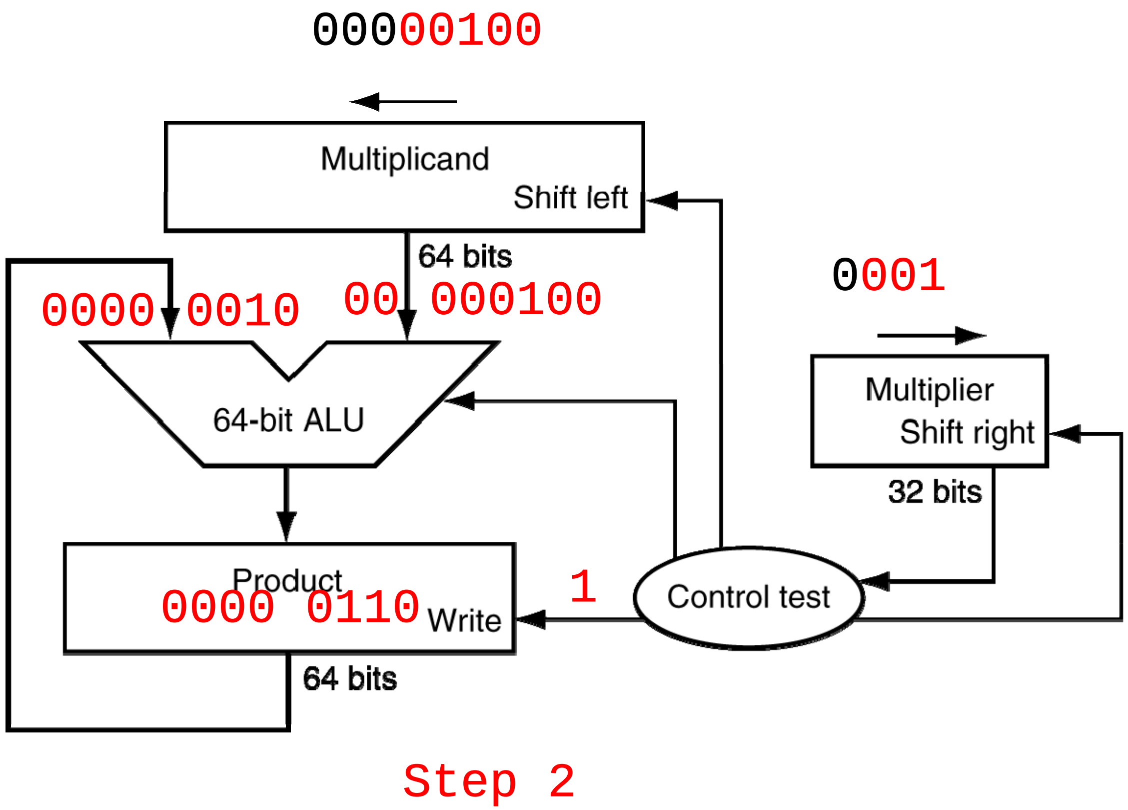
Multiplication

- Start with long-multiplication approach

multiplicand
multiplier
product

```
     0010
  ×  0011
  -------
     0010
    0010
   0000
  000
  -------
  0000110
```

Length of product is the sum of operand lengths



Multiplication

- Start with long-multiplication approach

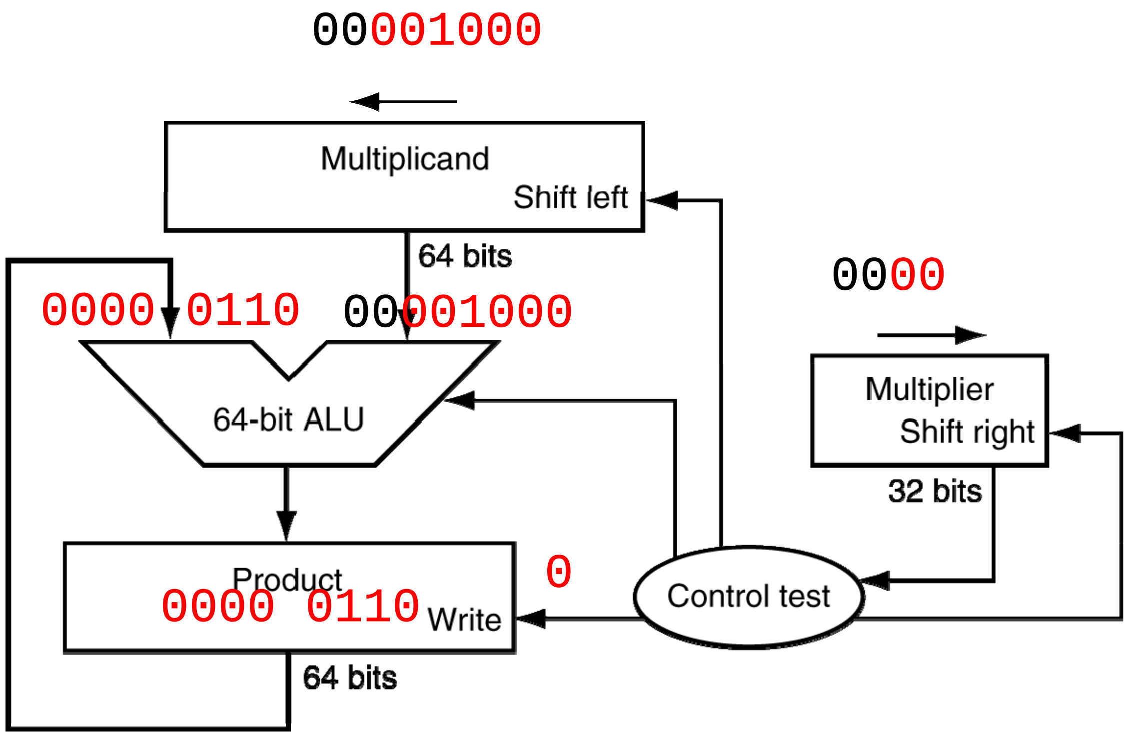
multiplicand

multiplier

product

```
      0010
  ×   0011
  --------
      0010
     0010
    0000
   000
  --------
   0000110
```

Length of product is the sum of operand lengths



Step 3

Multiplication

- Start with long-multiplication approach

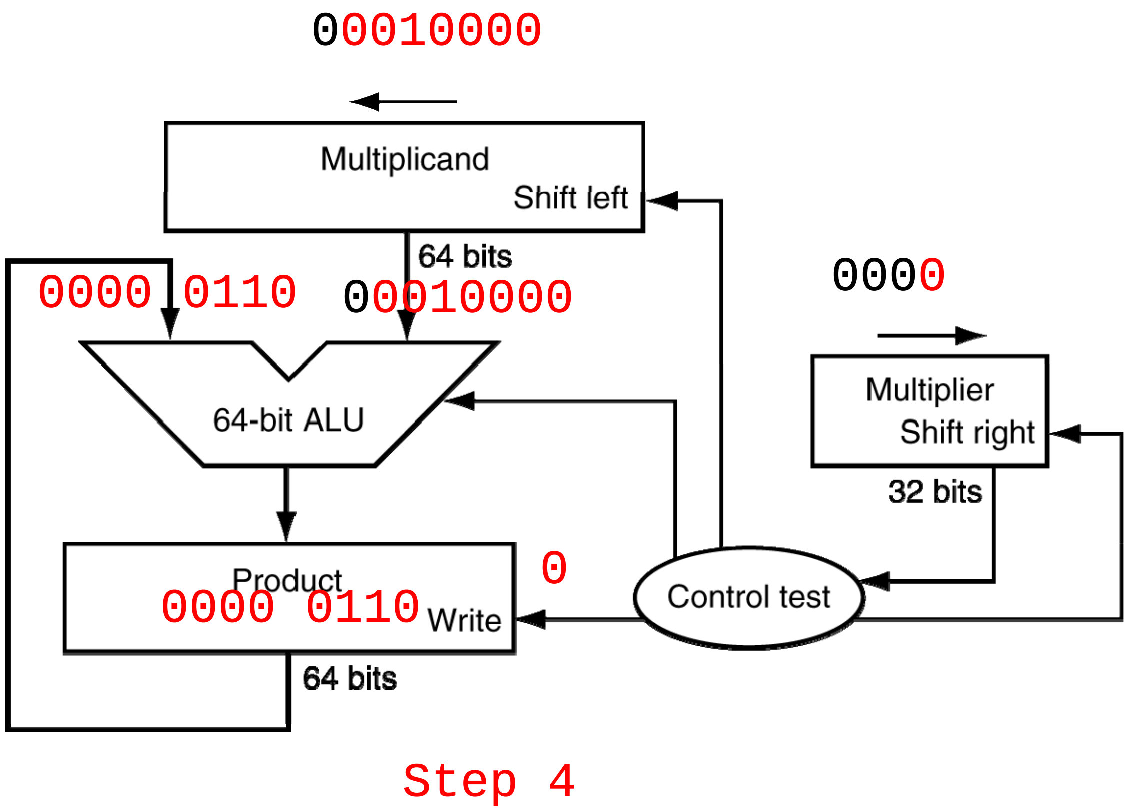
multiplicand

multiplier

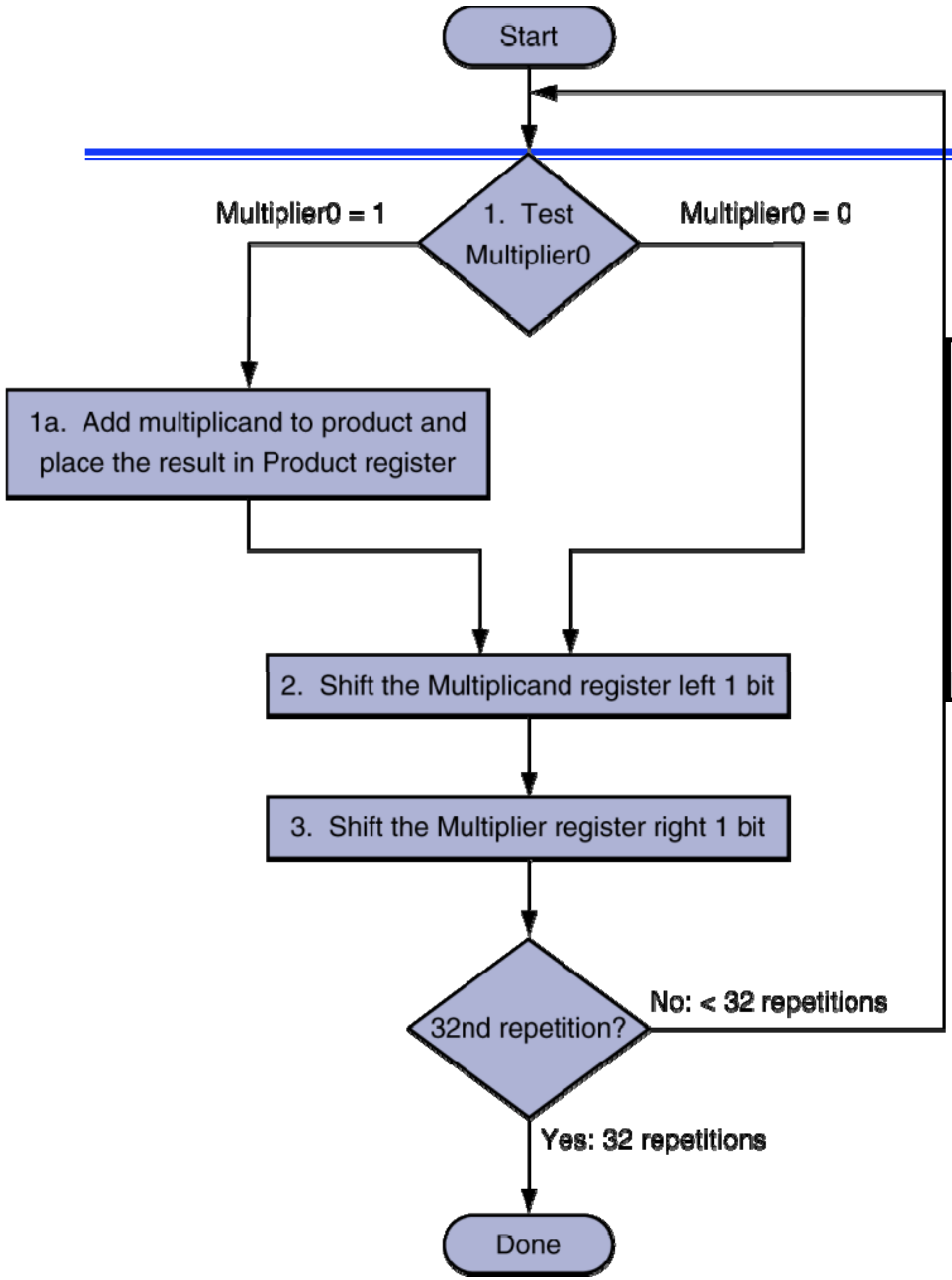
product

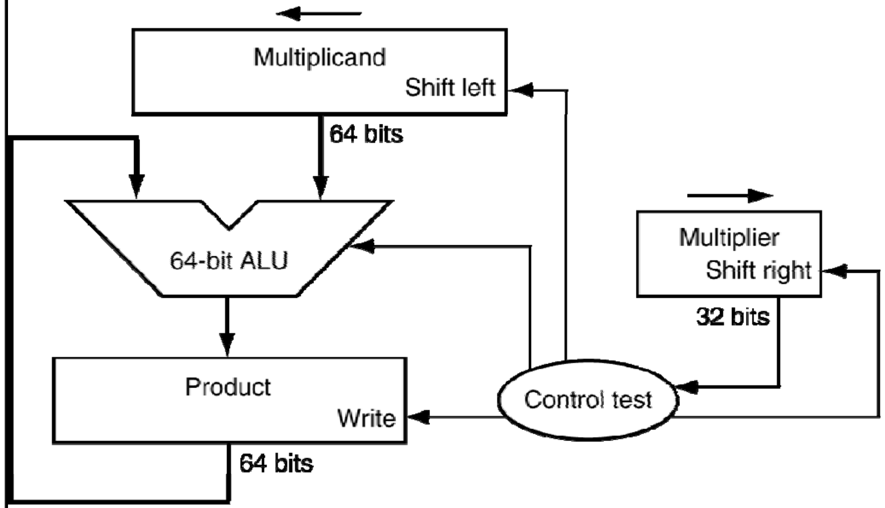
```
     0010
  ×  0011
  -------
     0010
    0010
   0000
  000
  -------
  0000110
```

Length of product is the sum of operand lengths



Multiplication Hardware





Multiplicand: 64 bits
Product: 64 bits
Multiplier: 32 bits

Optimized Multiplier

- Observations: Two ways of multiplication
  - Shift multiplicand left or shift product right

```
   1000
×  1011
-------
   1000
```

```
    1000
×   1011
--------
    1000
   1000
--------
   11000
```

```
    1000
×   1011
--------
   11000
  0000
--------
   11000
```

```
    1000
×   1011
--------
  011000
 1000
--------
 1011000
```

Shift multiplicand left 1 bit and add

Shift multiplicand left 2 bit but no add

```
   1000
×  1011
-------
   1000
```

```
    1000
×   1011
--------
     1000
    1000
--------
    11000
```

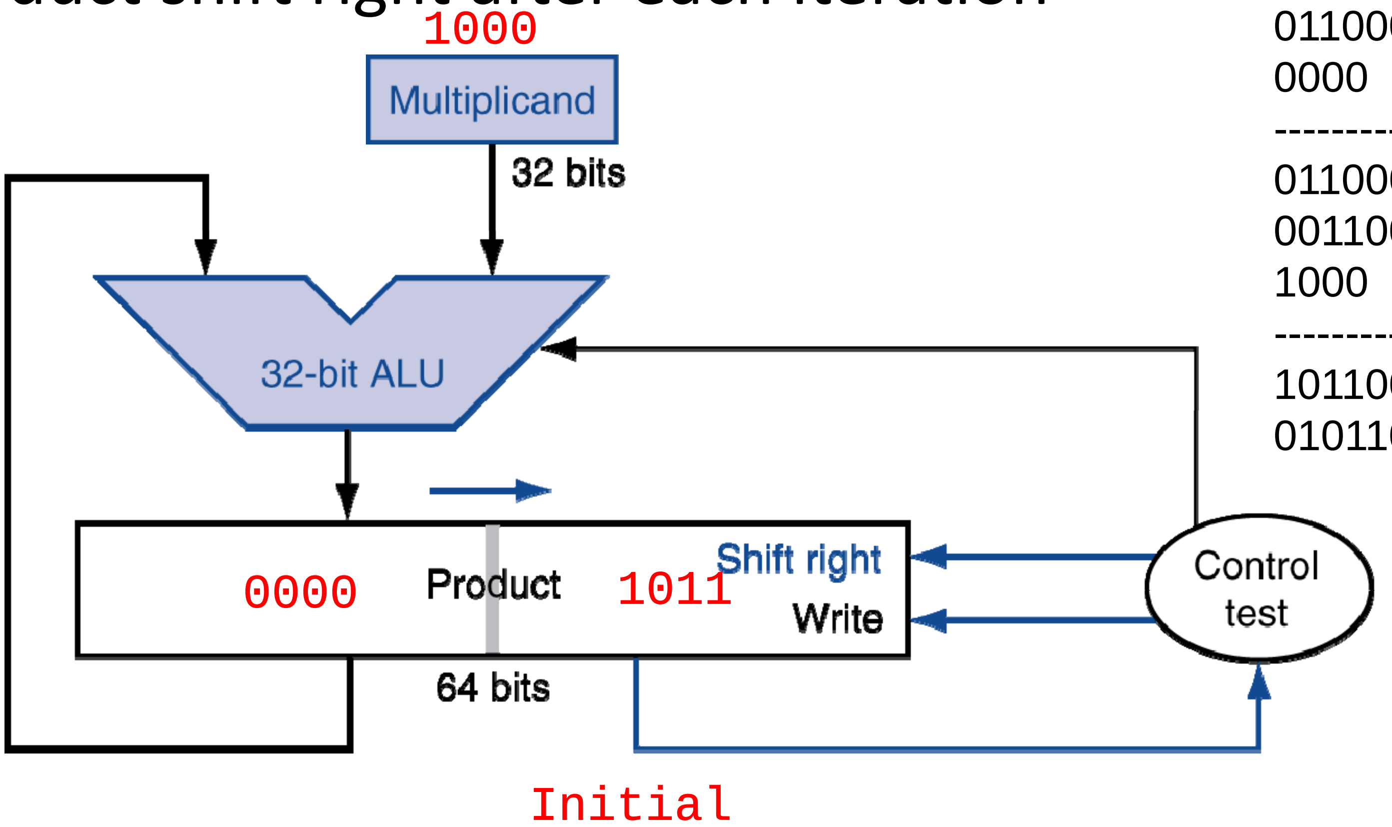
```
    1000
×   1011
--------
     11000
    0000
--------
    011000
```

```
    1000
×   1011
--------
     011000
    1000
--------
    1011000
```

product shift right and add

Optimized Multiplier

- Initial: 32 bit multiplicand, multiplier is stored in right side of product
- Product shift right after each iteration



1000
1011 add

10001011 shift
01000 101
1000 add

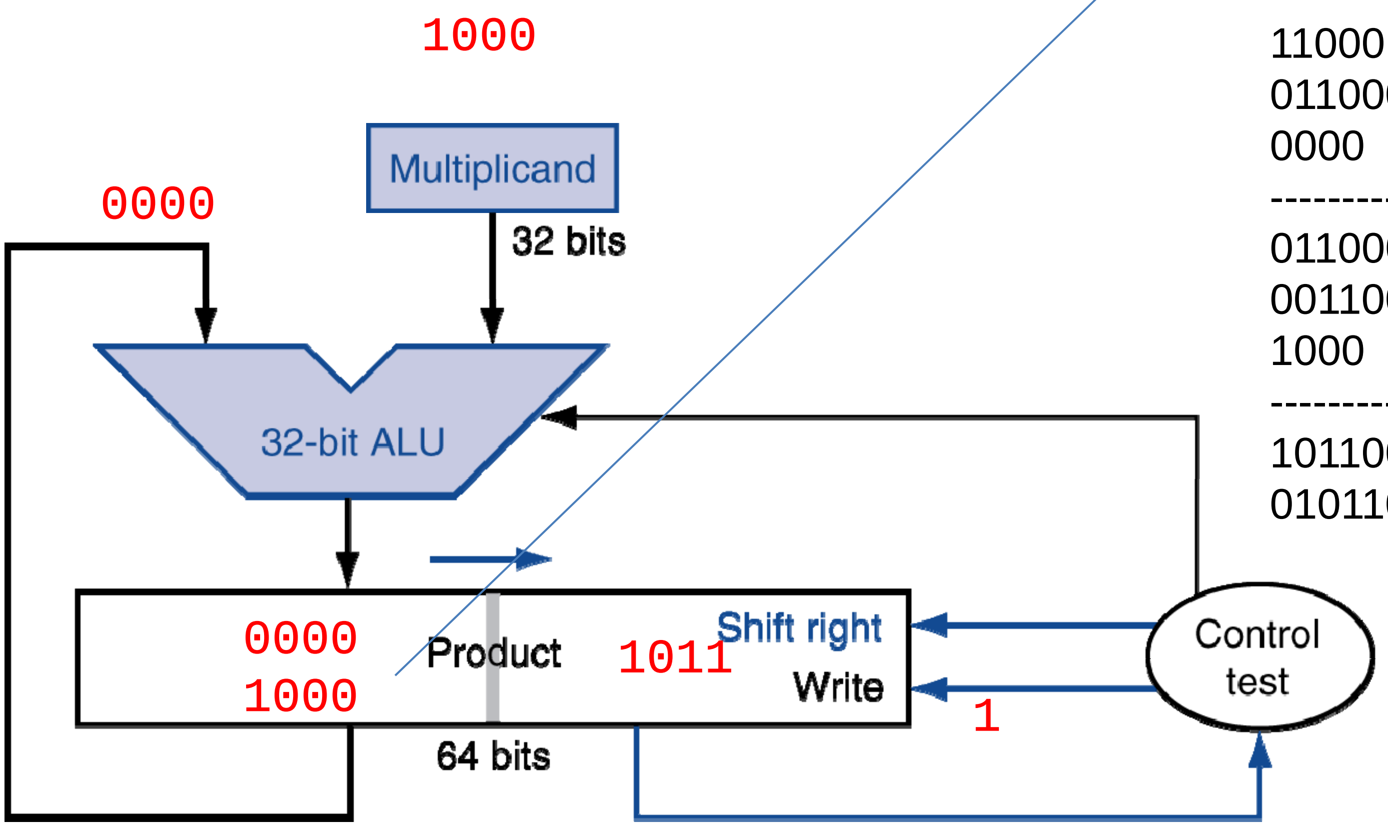
11000 101 shift
011000 10
0000 add

011000 10 shift
0011000 1
1000 add

1011000 1 shift
01011000

Optimized Multiplier

- Step 1



Step 1

1000
1011 add

10001011 shift
01000 101
1000 add

11000 101 shift
011000 10
0000 add

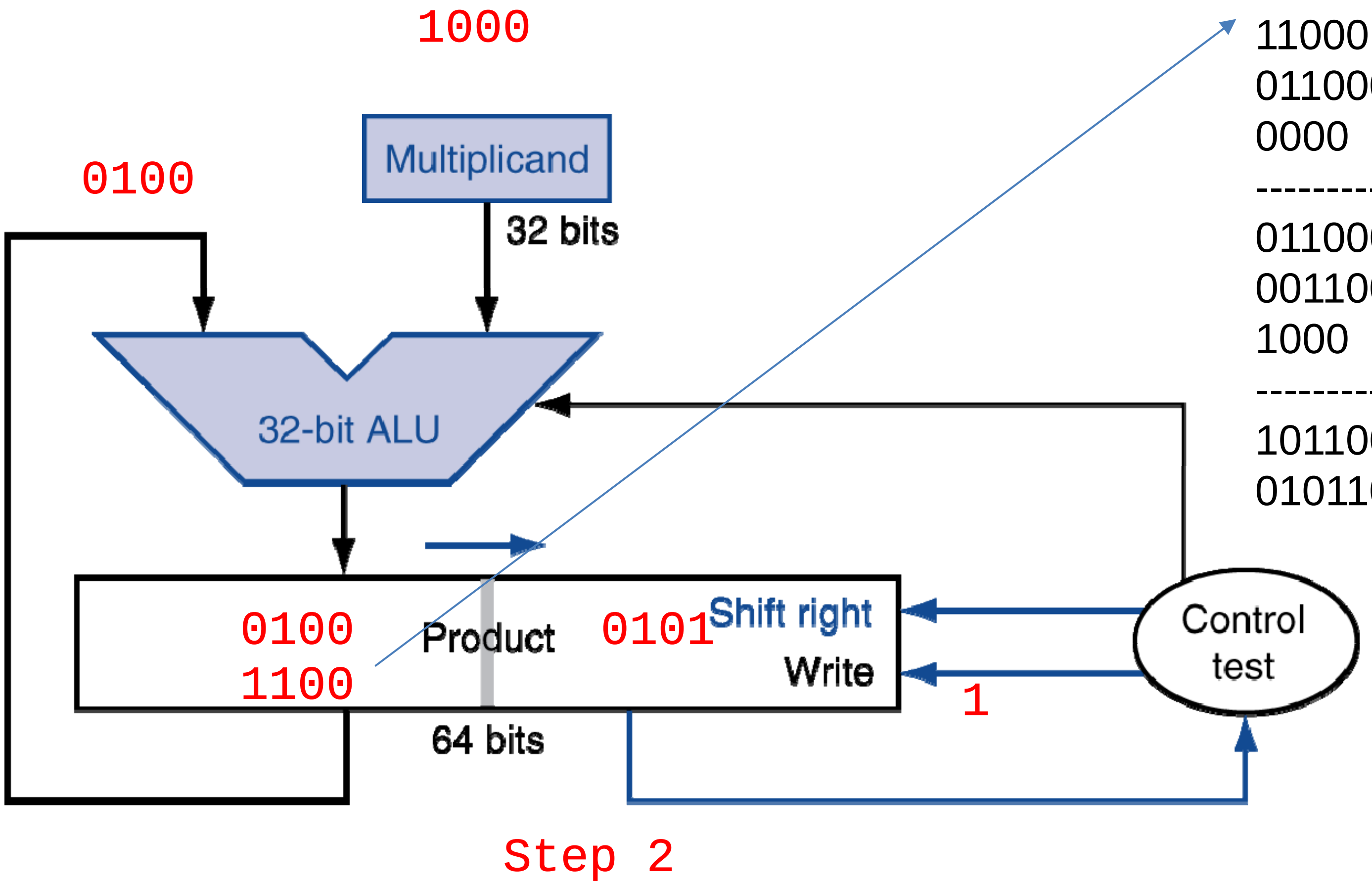
011000 10 shift
0011000 1
1000 add

1011000 1 shift
01011000

Optimized Multiplier

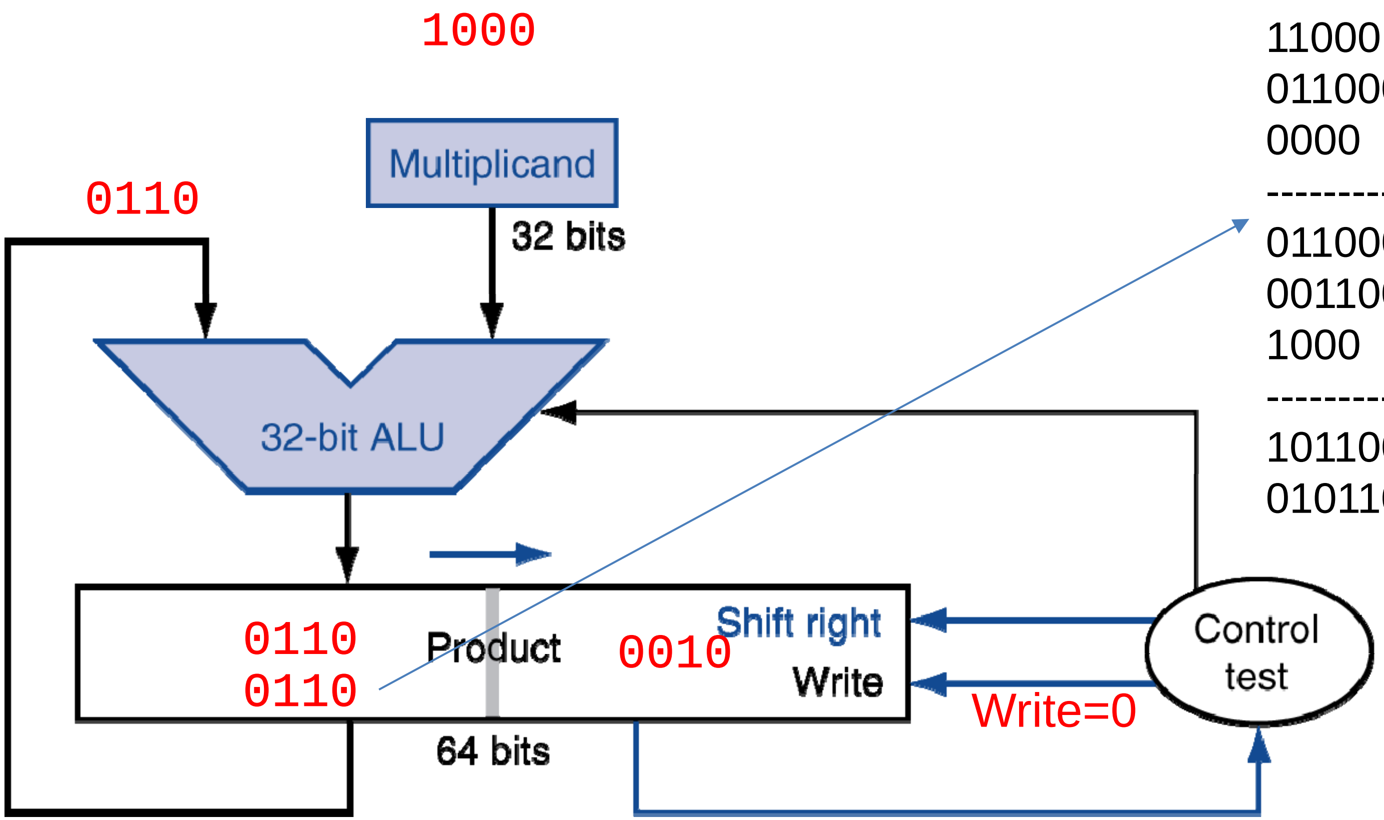
- Step 2

```
1000
1011          add
------------
10001011   shift
01000 101
1000          add
------------
11000 101  shift
011000 10
0000          add
----------------
011000 10   shift
0011000 1
1000          add
----------------------
1011000 1 shift
01011000
```



Optimized Multiplier

- Step 3



Step 3

1000
1011 add

10001011 shift
01000 101
1000 add

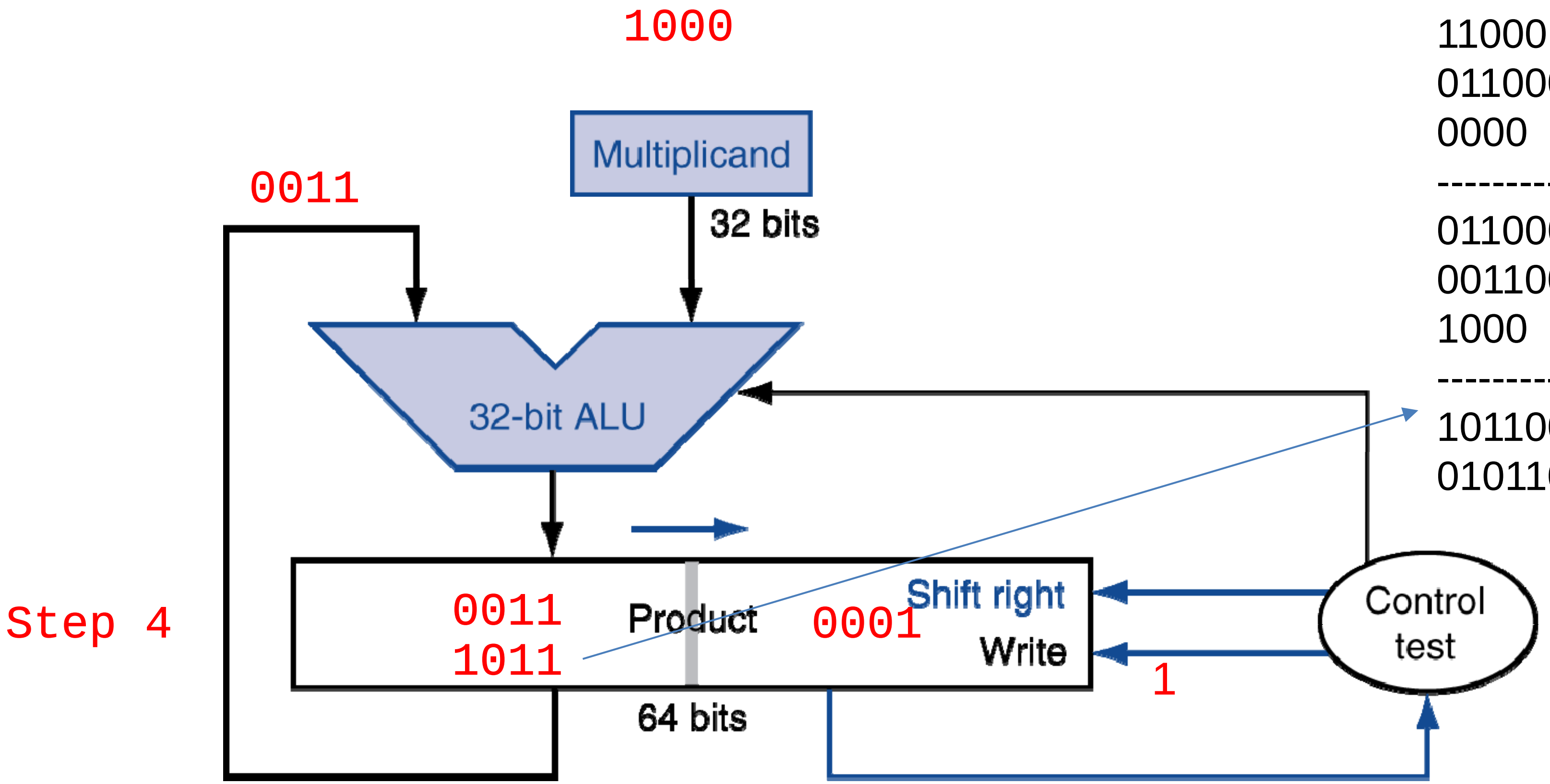
11000 101 shift
011000 10
0000 add

011000 10 shift
0011000 1
1000 add

1011000 1 shift
01011000

Optimized Multiplier

- Step 4:



1000
1011 add

10001011 shift
01000 101
1000 add

11000 101 shift
011000 10
0000 add

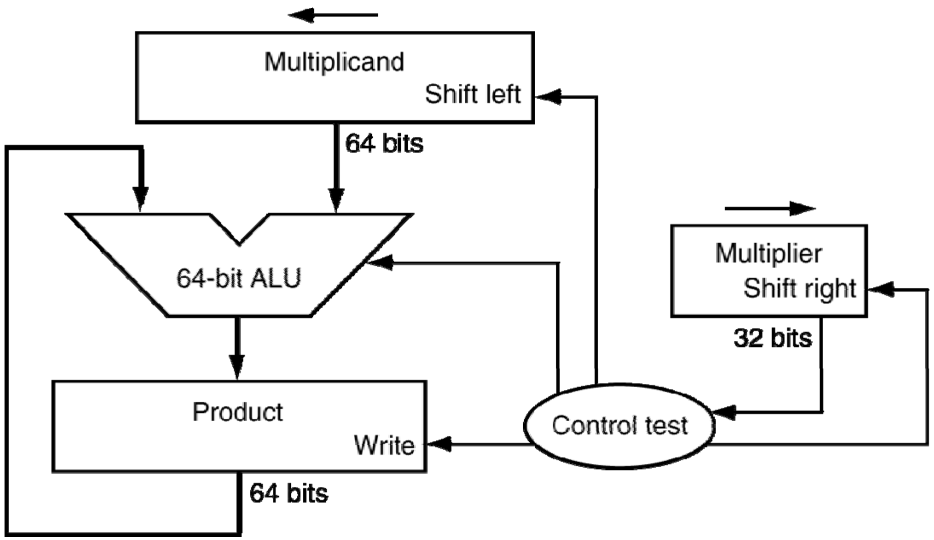
011000 10 shift
0011000 1
1000 add

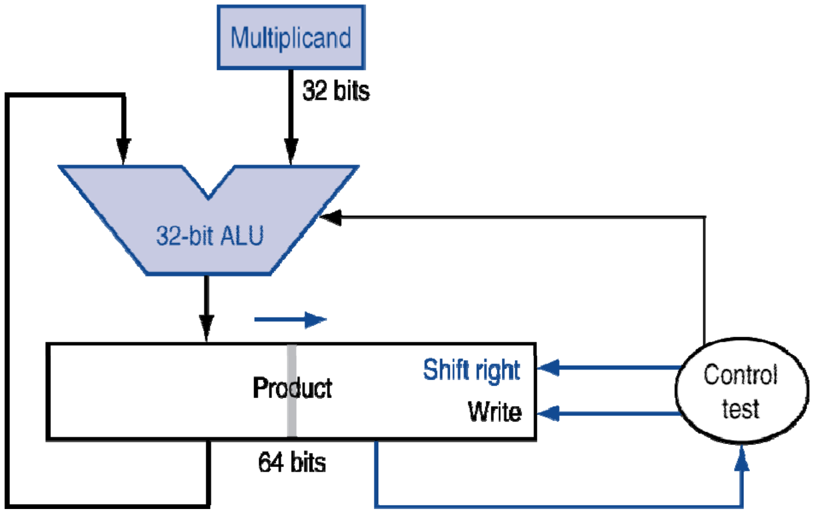
1011000 1 shift
01011000

Final product: 01011000

Two versions of multiplier

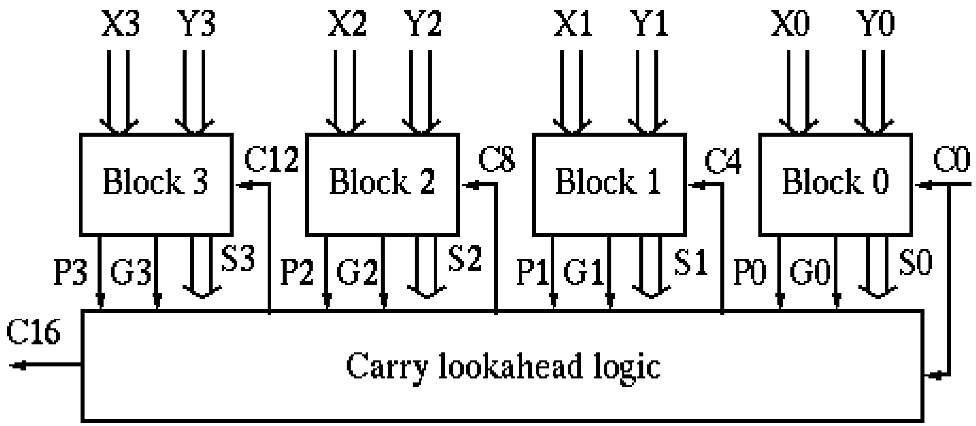
- Compare the two versions of multiplier



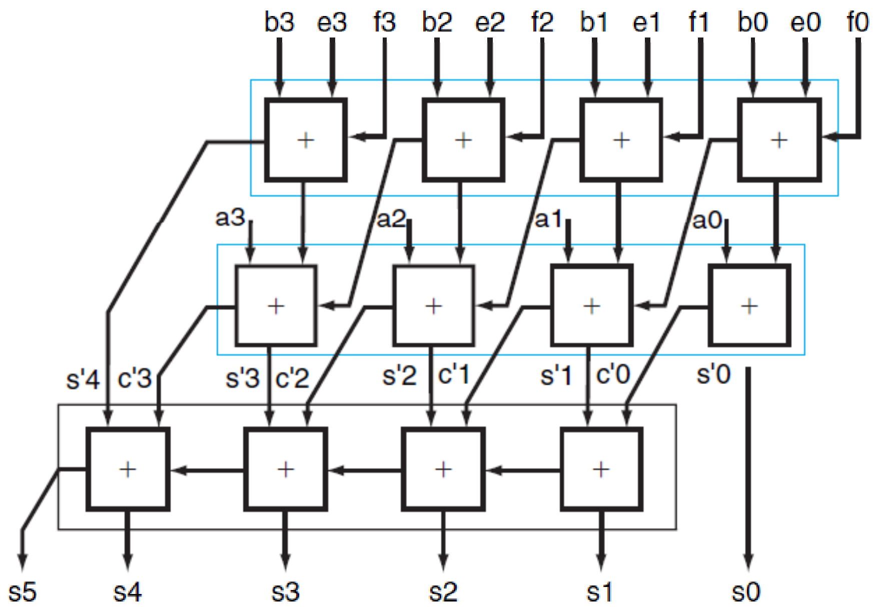


Faster Multiplication

- Uses faster adder
  - *Add* is repetitively performed
  - Faster adder can improve the speed of multiplication
  - E.g. carry lookahead adder, carry save adder, etc.



Carry lookahead adder



Carry save adder

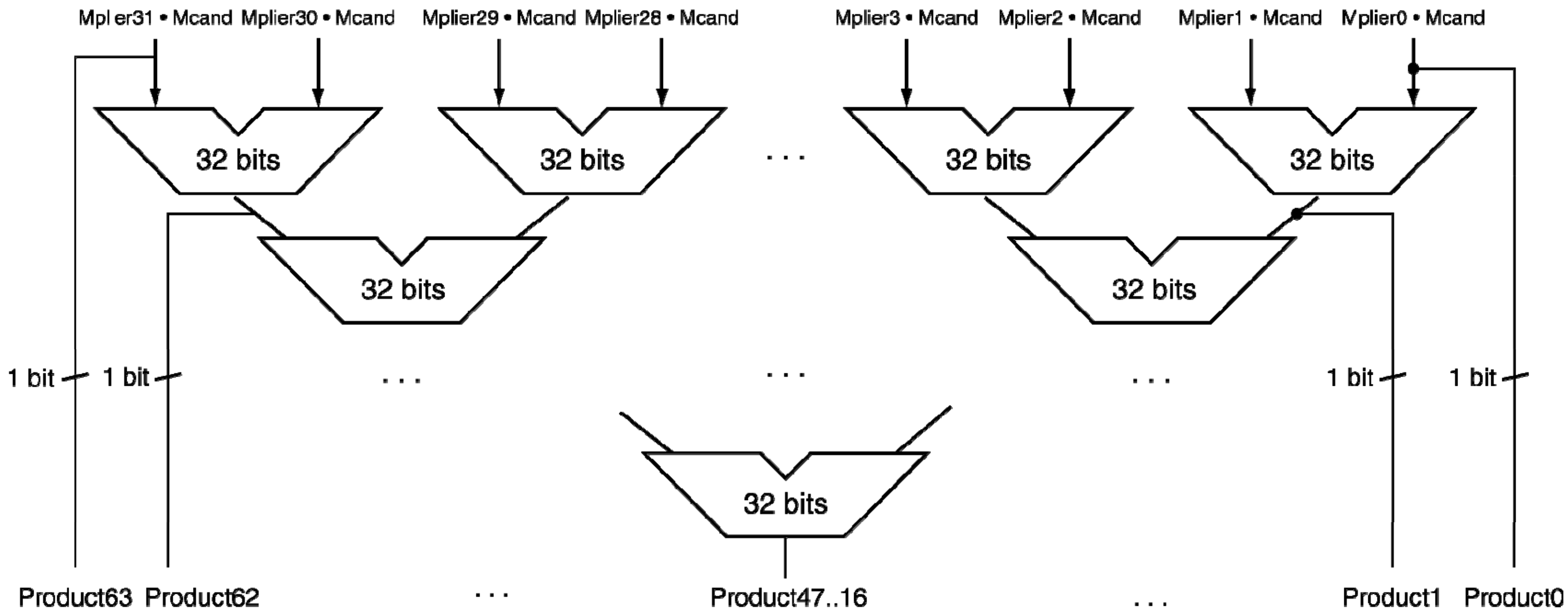
Faster Multiplier

- Perform addition in parallel
  - Uses multiple adders
  - Time = (time of *add*) * $\log_2(32)$

```
   0010
×  0011
   0010
  0010
 0000
0000
0000110
```

Perform addition
in parallel

```
   0010
×  0011
  00110
00000
0000110
```



MIPS Multiplication

- Two special 32-bit registers for product
  - HI: most-significant 32 bits
  - LO: least-significant 32-bits
- Instructions
  - mult rs, rt   # HI|LO = $rs * $rt , result is stored in 64 bit HI|LO
  - mfhi rd / mflo rd
    - Move from HI/LO to rd
  - mul rd, rs, rt
    - Least-significant 32 bits of product is moved to $rd (use when you know the product is less than 32 bits)

Example

- Write assembly code that compute 5*12 - 74

```
ori     $t0, $0, 12      # put 12 into $t0
ori     $t1, $0,  5      # put 5 into $t1
mult    $t0, $t1         # lo = 5x12
mflo    $t1              # $t1 = 5x12
addi    $t1, $t1,-74     # $t1 = 5x12 - 74
```

Division

```
quotient               1001
dividend            ________
divisor     1000 ) 1001010
                  -1000
                     10
                     101
                     1010
                    -1000
remainder              10
```

n-bit operands yield *n*-bit quotient and remainder

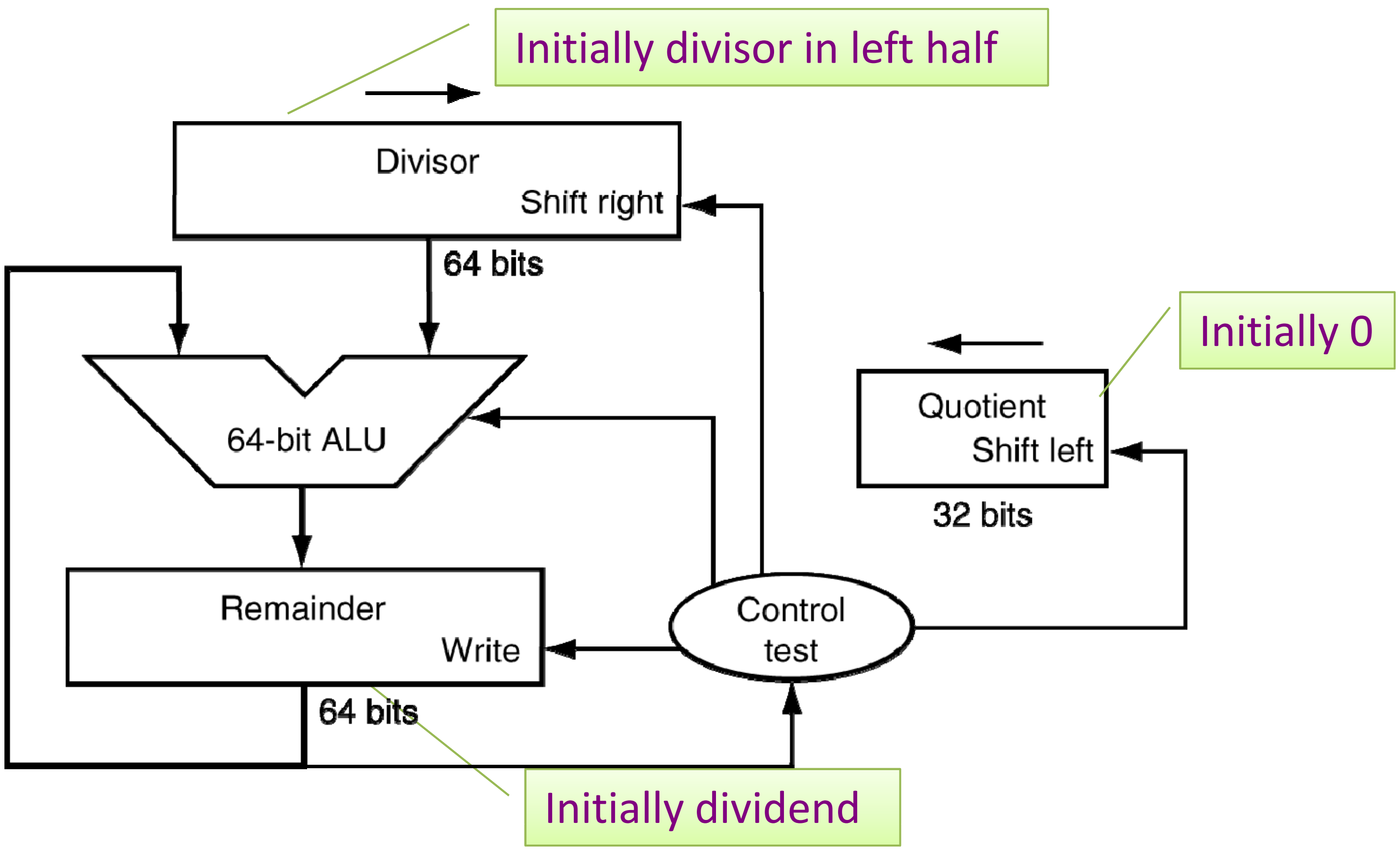
- Check if divisor = 0
- Long division approach
  - If divisor ≤ dividend
    - 1 bit in quotient, subtract
  - Otherwise
    - 0 bit in quotient, bring down next dividend bit
- Division is just a bunch of shifts and subtracts

Restoring division

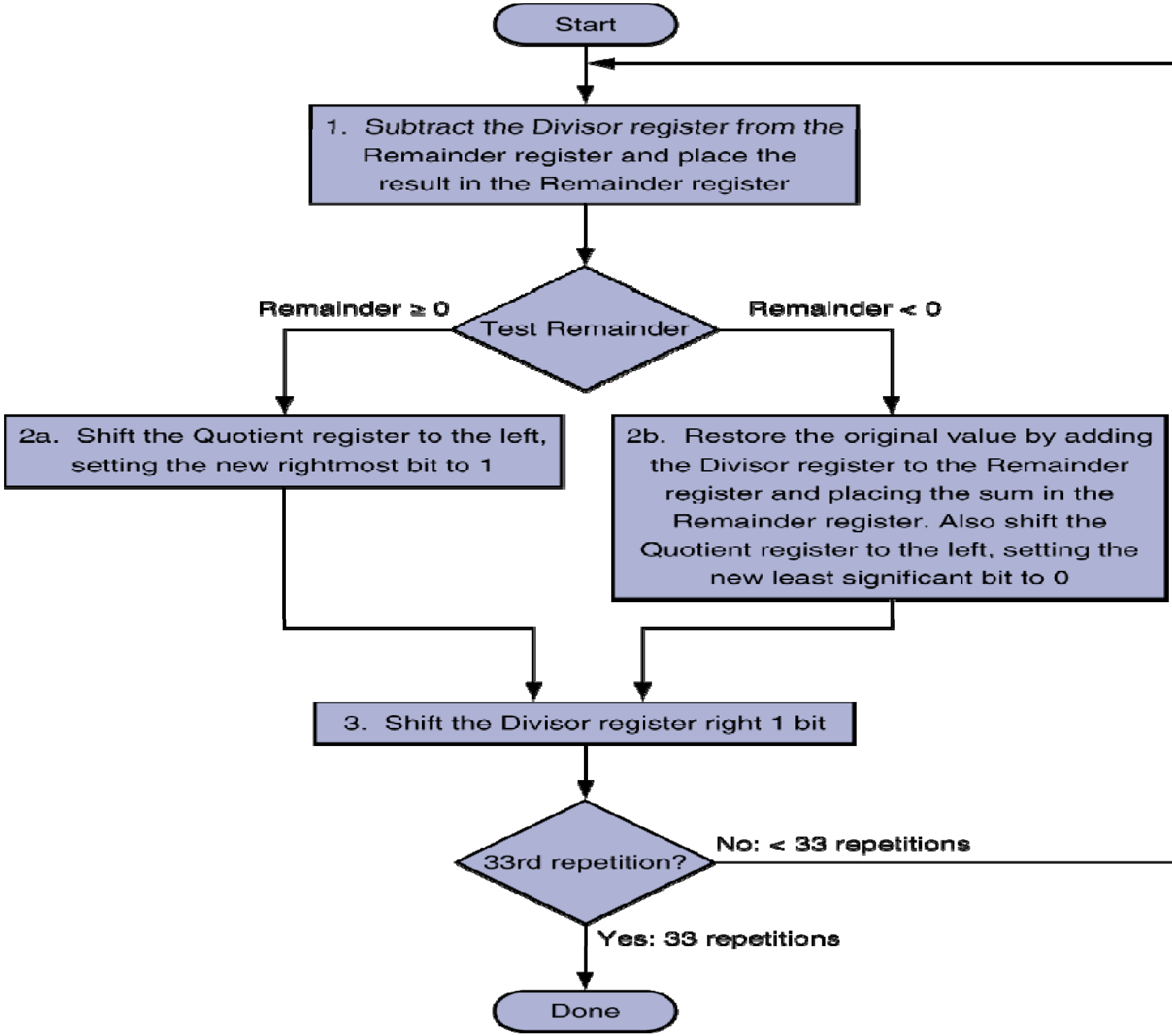
Do the subtract, and if remainder goes < 0, add divisor back

A Division Hardware

- Division is similar to multiplication, so is hardware



Division Steps

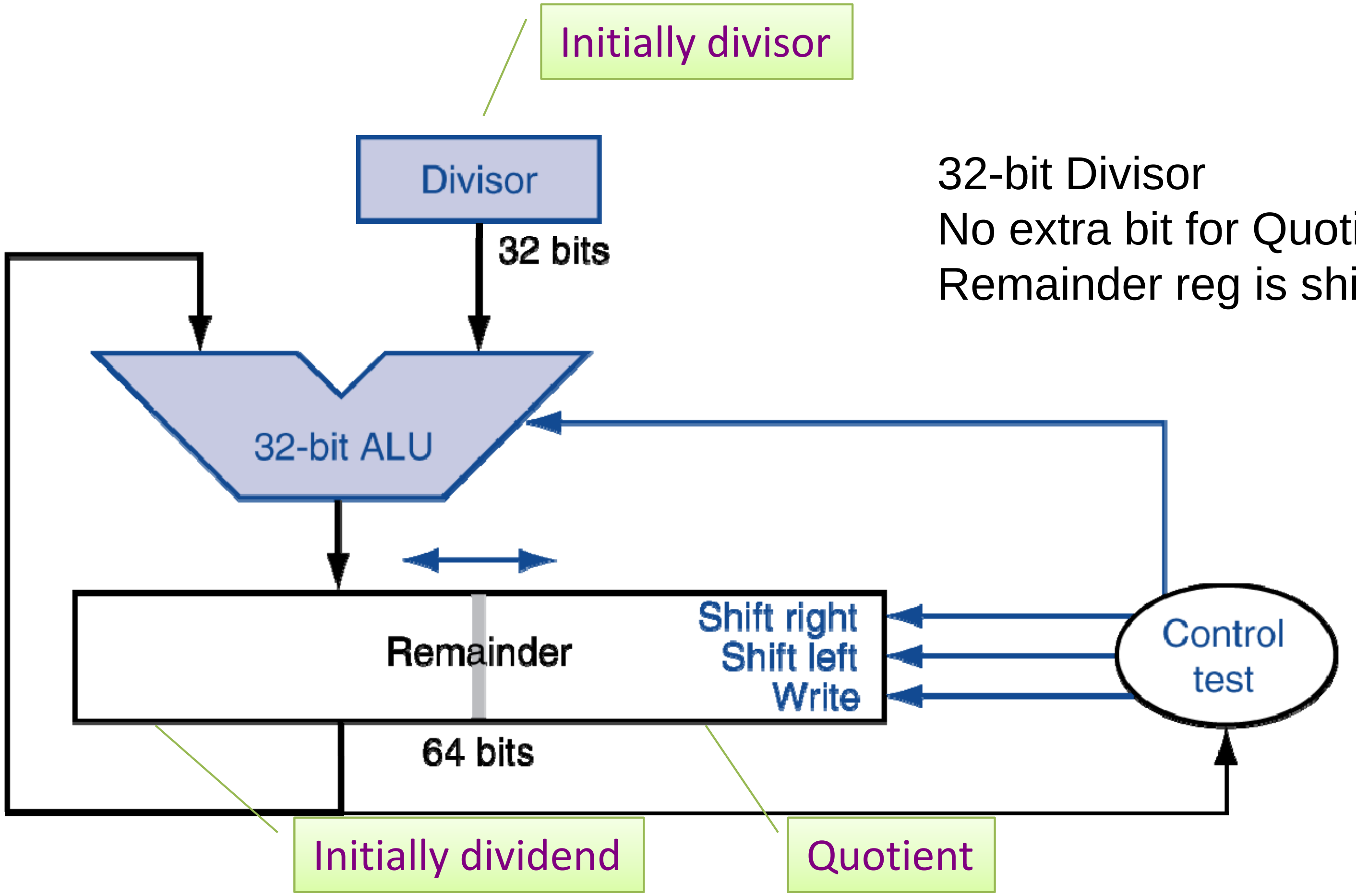


A Divide Example

Dividing 7 by 2 (4-bit version)

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 $\Longrightarrow$ +Div, sll 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 $\Longrightarrow$ +Div, sll 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 $\Longrightarrow$ +Div, sll 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 $\Longrightarrow$ sll 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 $\Longrightarrow$ sll Q, Q0 = 1	0011	0000 0010	0000 0001
	3: Shift Div right	0011	0000 0001	0000 0001

Improved Divider Hardware



32-bit Divisor
No extra bit for Quotient
Remainder reg is shifted left

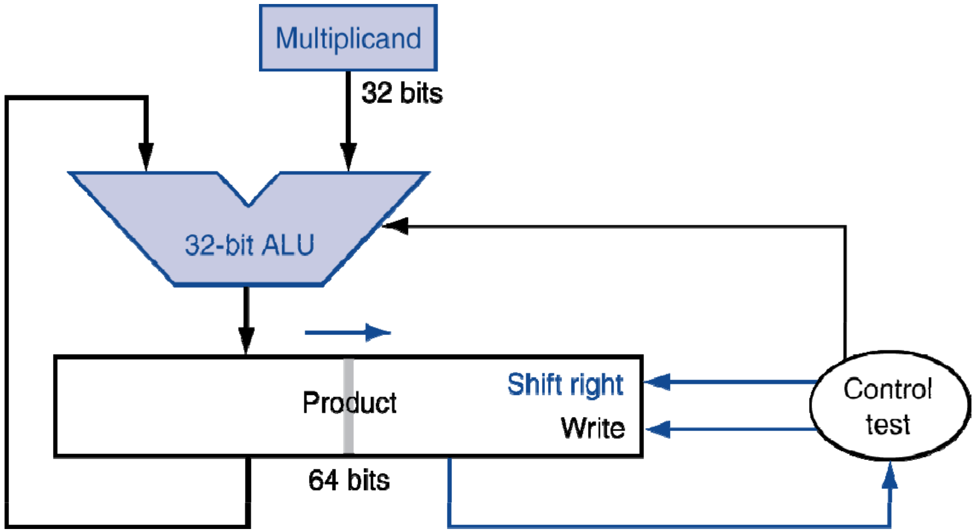
Signed Division

- Signed division
  - Divide using absolute values
  - Adjust sign of quotient and remainder as required
- Negate the quotient if the signs of divisor and dividend disagree

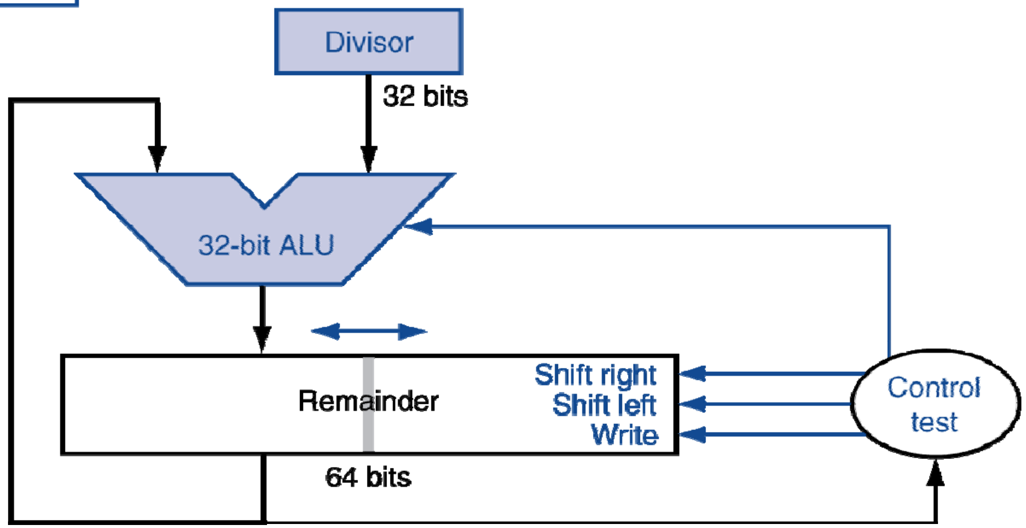
Faster Division

- Can’t use parallel hardware as in multiplier
  - Subtraction is conditional on sign of remainder
    - Done sequentially
    - Division is slower than multiplication
- Faster dividers (e.g. SRT division) generate multiple quotient bits per step
  - Still require multiple steps

Multiplier and Divider



Same structure…
32-bit reg
64-bit reg that allows shift left/right
32-bit ALU that can do add/sub



MIPS Divide Instruction

- 32-bit HI/LO reg are used by both multiply and divide instructions

- Divide Instructions
  - `div rs, rt  /  divu rs, rt`
  - Reminder in HI and the quotient in LO

```
div      $s0, $s1          # lo = $s0 / $s1
                           # hi = $s0 mod $s1
```

- Instructions `mfhi rd` and `mflo rd` are provided to move the quotient and reminder to user accessible registers

- No overflow or divide-by-0 checking
  - Divide ignores overflow so software must determine if the quotient is too large. Software must also check the divisor to avoid division by 0.

An Example

- Calculate 13/5, put the quotient in $t1, and reminder in $t0

```
.text
      .globl  main

main:
ori        $t5, $zero, 13    # put 13 into $5
ori        $t6, $zero,  5    # put 5 into $6
div        $t5, $t6          #  Lo = $t5 / $t6   (integer quotient)
                             #  Hi = $t5 mod $t6   (remainder)
mfhi        $t0              #  move reminder from Hi to $t0:   $t0 = Hi
mflo        $t1              #  move quotient from Lo to $t1:   $t1 = Lo
```