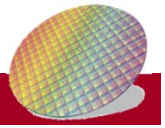
Chapter 3

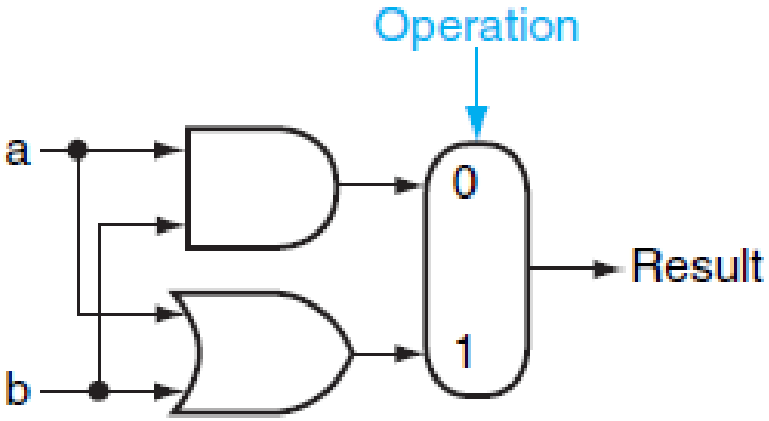
Arithmetic for Computers



Basic Arithmetic Logic Unit

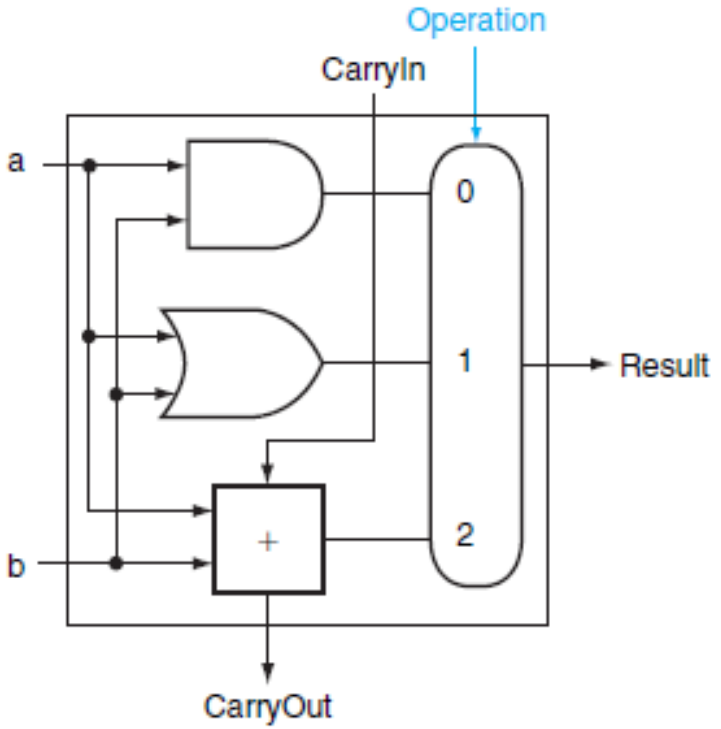
From B.5 (Constructing a Basic Arithmetic Logic Unit)

- Basic ALU



One-bit ALU that performs AND and OR

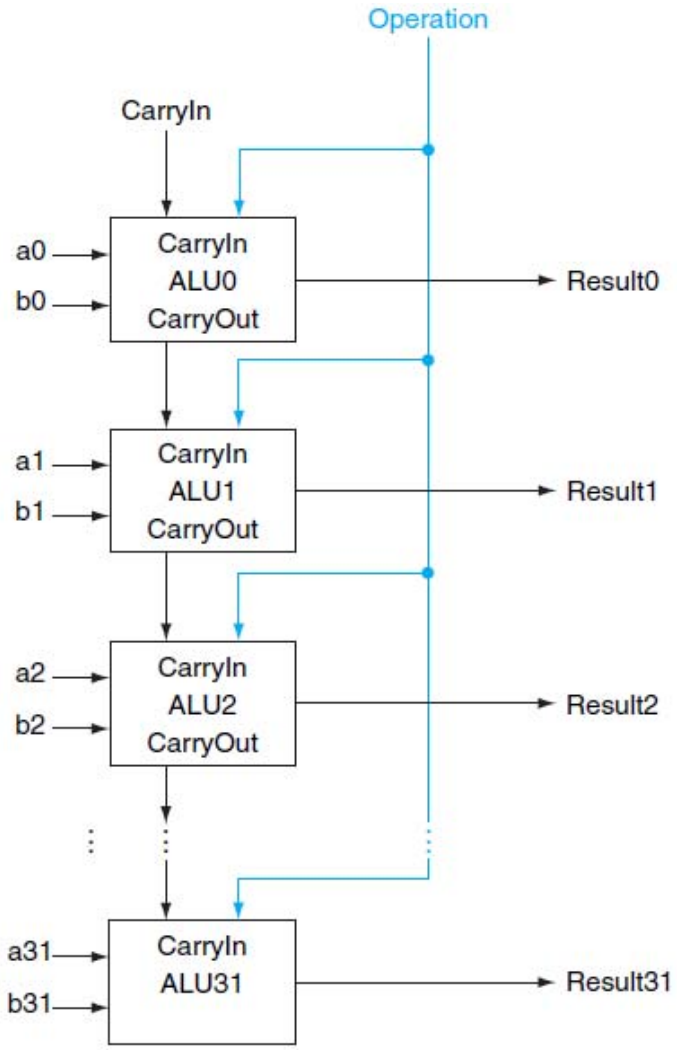
Operation(Op.)	Funct.
0	a AND b
1	a OR b



Operation(Op.)	Funct.
0	a AND b
1	a OR b
2	a + b

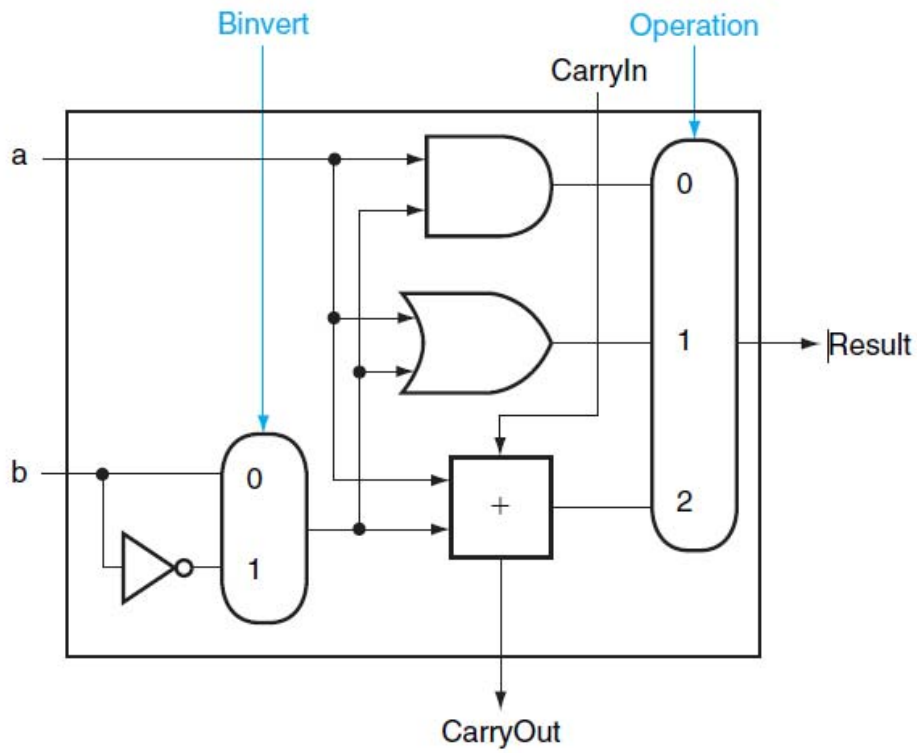
32-bit ALU

- Cascading 1-bit ALU to 32-bit ALU
- carry-out is the carry-in of the next bit



Enhanced Arithmetic Logic Unit

- ALU that performs (a AND b), (a OR b) and (a + b) and (a- b=a + $\bar{b}$+1)
- To perform a-b => Binvert=1 and carryIn=1

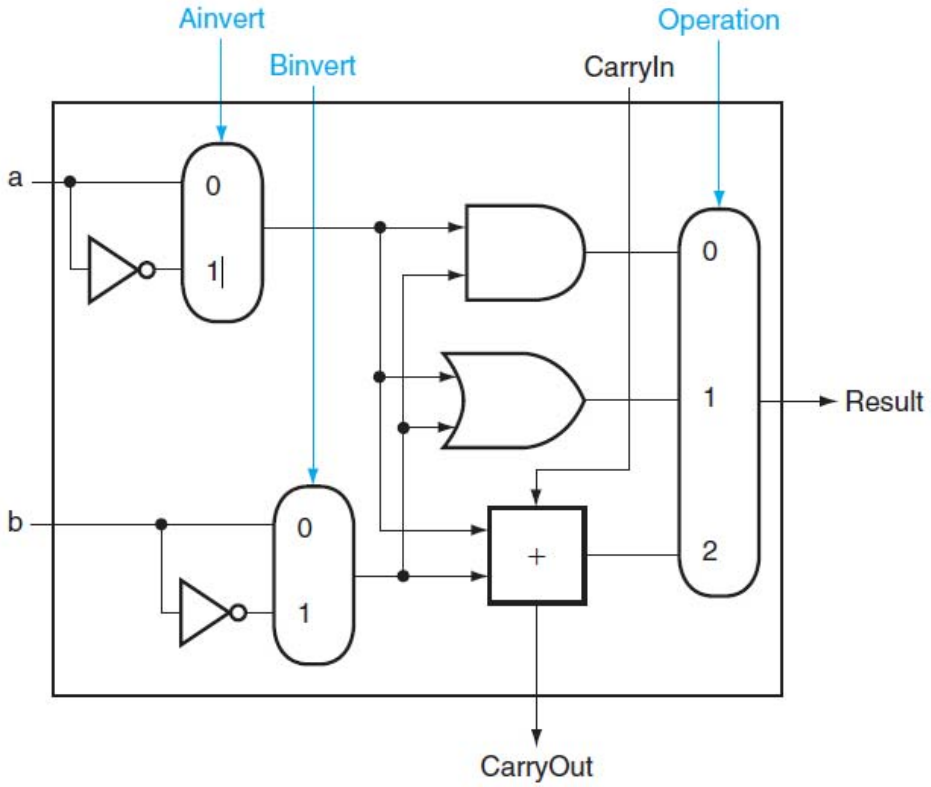


a-b = a+$\bar{b}$ +1

Binvert	CarryIn	Op.	Function
0	X	0	a and b
0	X	1	a or b
0	0	2	a + b
1	1	2	a-b

Enhanced Arithmetic Logic Unit

- Enhanced with NOR and NAND



$$\overline{ab} = \bar{a} \vee \bar{b}$$

$$\overline{a \vee b} = \bar{a}\ \bar{b}$$

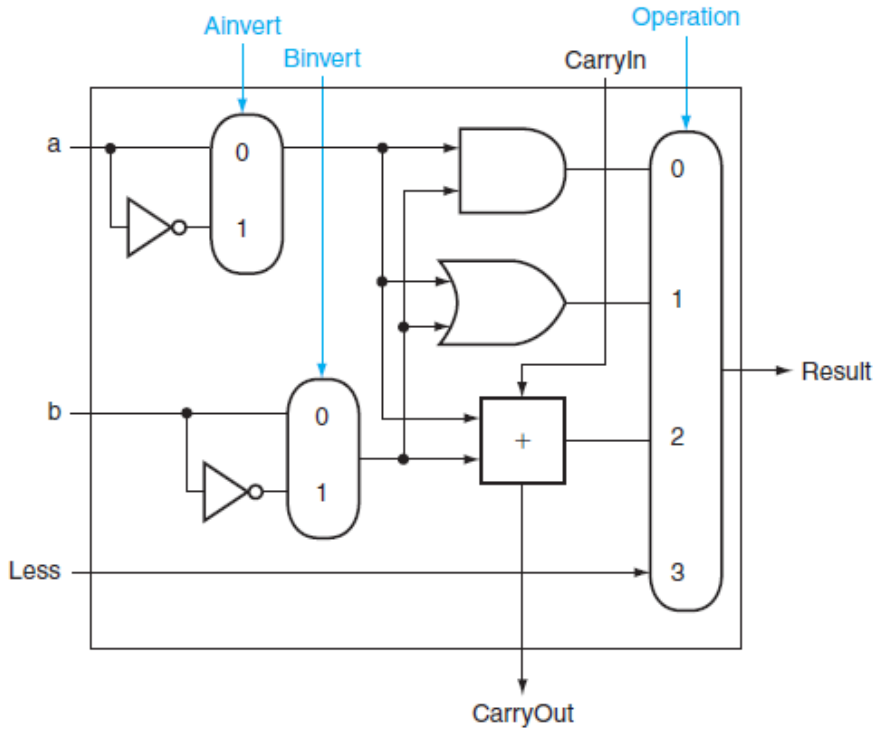
Ainvert	Binvert	CarryIn	Op.	Func.
0	0	X	0	a and b
0	0	X	1	a or b
0	0	0	2	a + b
0	1	1	2	a-b
1	1	X	0	$\overline{a+b}$
1	1	X	1	$\overline{ab}$

ALUs with Set Less Than

Review: Set less than

slt $t0 $t1 $t2 =>When $t1 < $t2 , $t0 =1, otherwise $t0=0

- Use a-b to implement slt
  - When a-b < 0, signed bit =1
  - When a-b >=0, signed bit =0
- Less signal is the LSB that
  - Connect to the signed bit of MSB (See next slide)
  - Other signals are assigned to 0

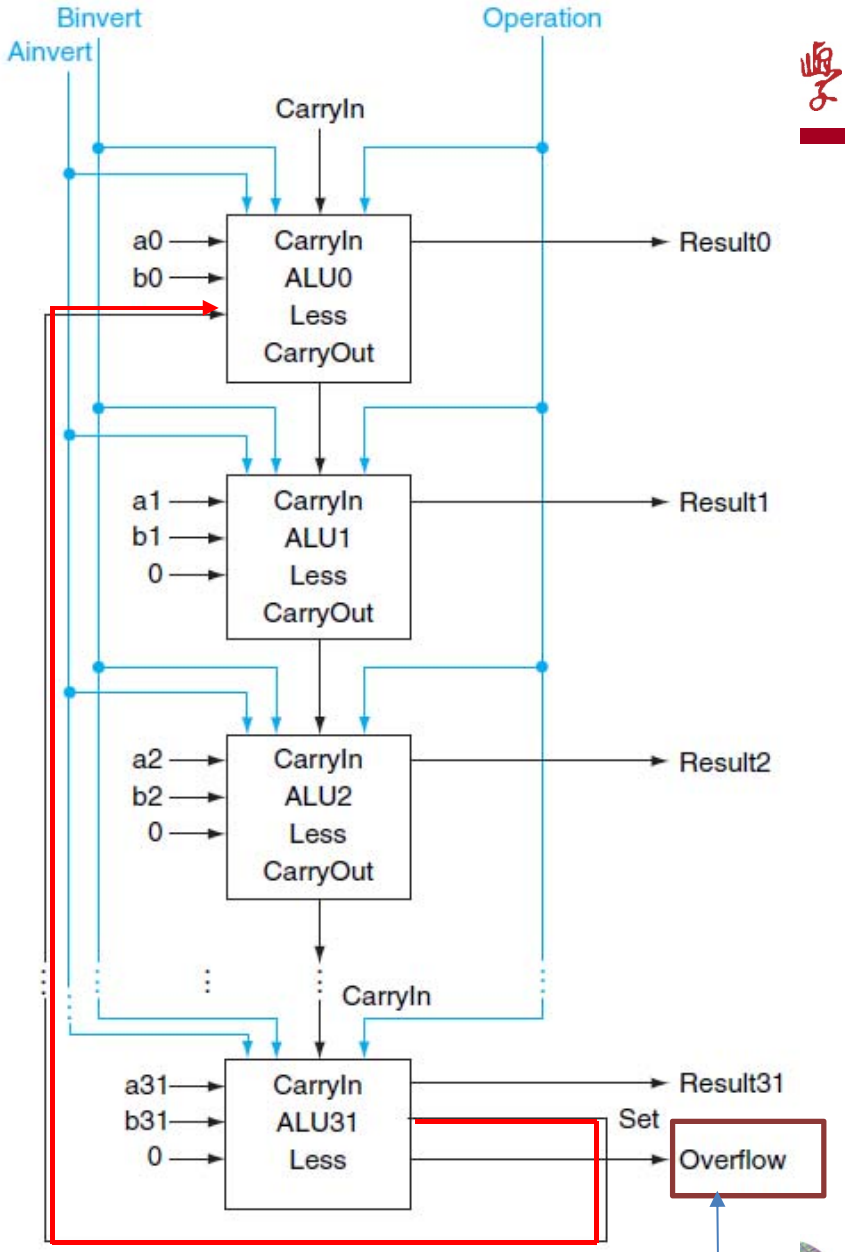


32-bit ALU with Set Less than

- Less signal=>
  - Connect LSB to the signed bit of MSB
  - Other signals are assigned to 0

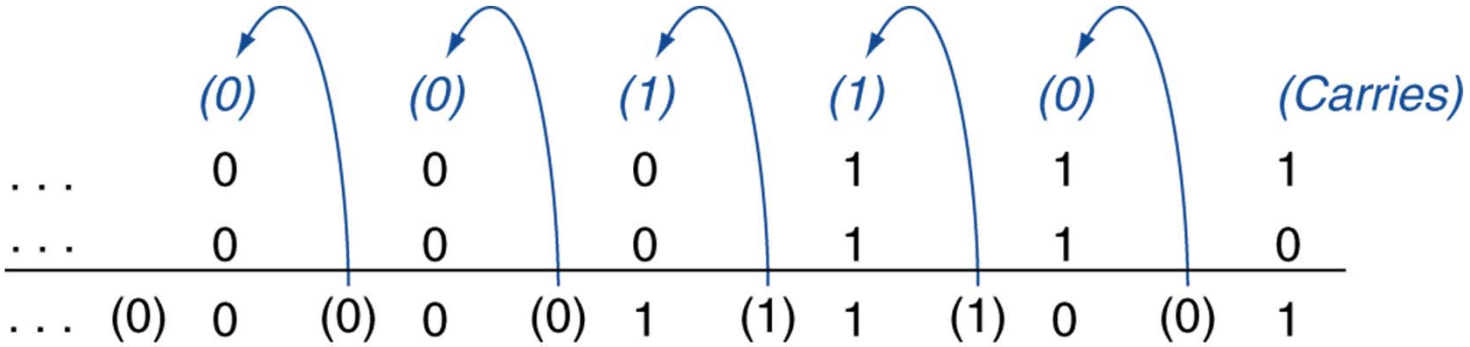
When a31…a0 < b31….b0 , result is 0…….1, otherwise 0……0

Note that MSB is different than other bits=> it has one additional signal (Overflow) which will be discussed later



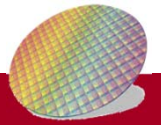
Integer Addition and Subtraction

- Addition Example: 7 + 6



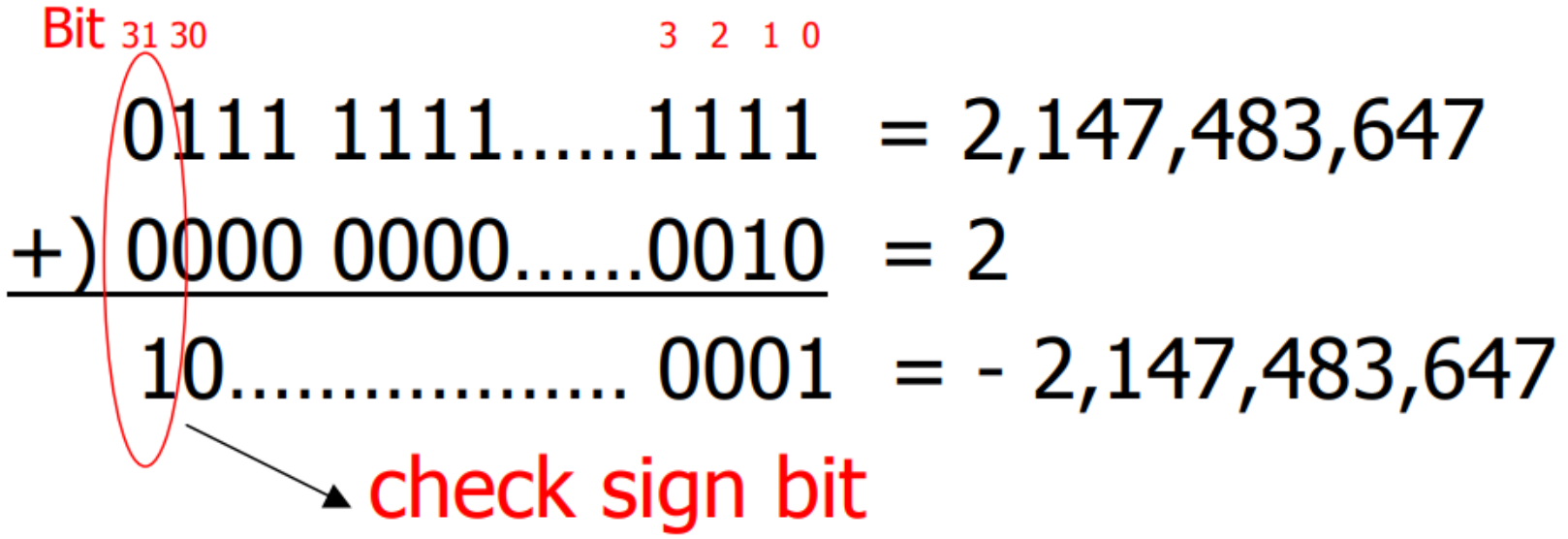
- Subtraction Example: 7-6 = 7+(-6)

00000111	+7: 0000 0000 … 0000 0111
– 00000110	+–6: 1111 1111 … 1111 1010
00000001	+1: 0000 0000 … 0000 0001



Overflow

- Overflow : Sum/difference of two 32-bit numbers is too large to be represented in 32 bits.



Overflow condition

- Exception: An unscheduled event that disrupts program execution; used to detect overflow.
- Situation that overflow occurs for signed integers
- Instruction that causes exception when overflow: add, addi, sub
- Instruction that does not cause exception when overflow: addu, addiu, subu

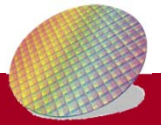
Operation	A	B	Result when Overflow
A+B	A>=0	B>=0	<0
A+B	<0	<0	>=0
A-B	A>=0	B<0	<0
A-B	A<0	B>=0	>=0

$$\begin{array}{r} 0111 \\ +0001 \\ \hline 1000 \end{array}$$

7+1 ≠ -8

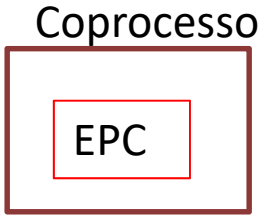
$$\begin{array}{r} 1111 \\ +1000 \\ \hline 0111 \end{array}$$

-1+(-8) ≠ 7



How to handle overflow in HW?

- MIPS detects overflow and causes exception (which is essentially an unplanned procedure call.
- Steps: (HW interrupt)
  - The address of the instruction that caused overflow is saved in a EPC (exception program counter) register ($R14 in Coprocessor 0)
  - Jump to the service routine for that exception
  - Return to the program
- MIPS uses
  - Instruction "move from CoProc 0 (MFC0)"to copy EPC into a register so that MIPS software can return to the offending instruction via "jr $reg"instruction.



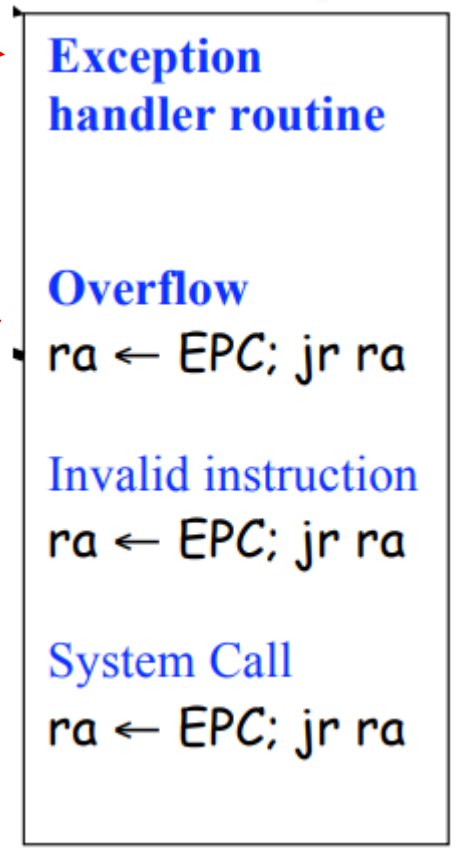
Exception flow

add $t0, $t1, $t2

Overflow

(Address is saved in EPC)

Next instruction

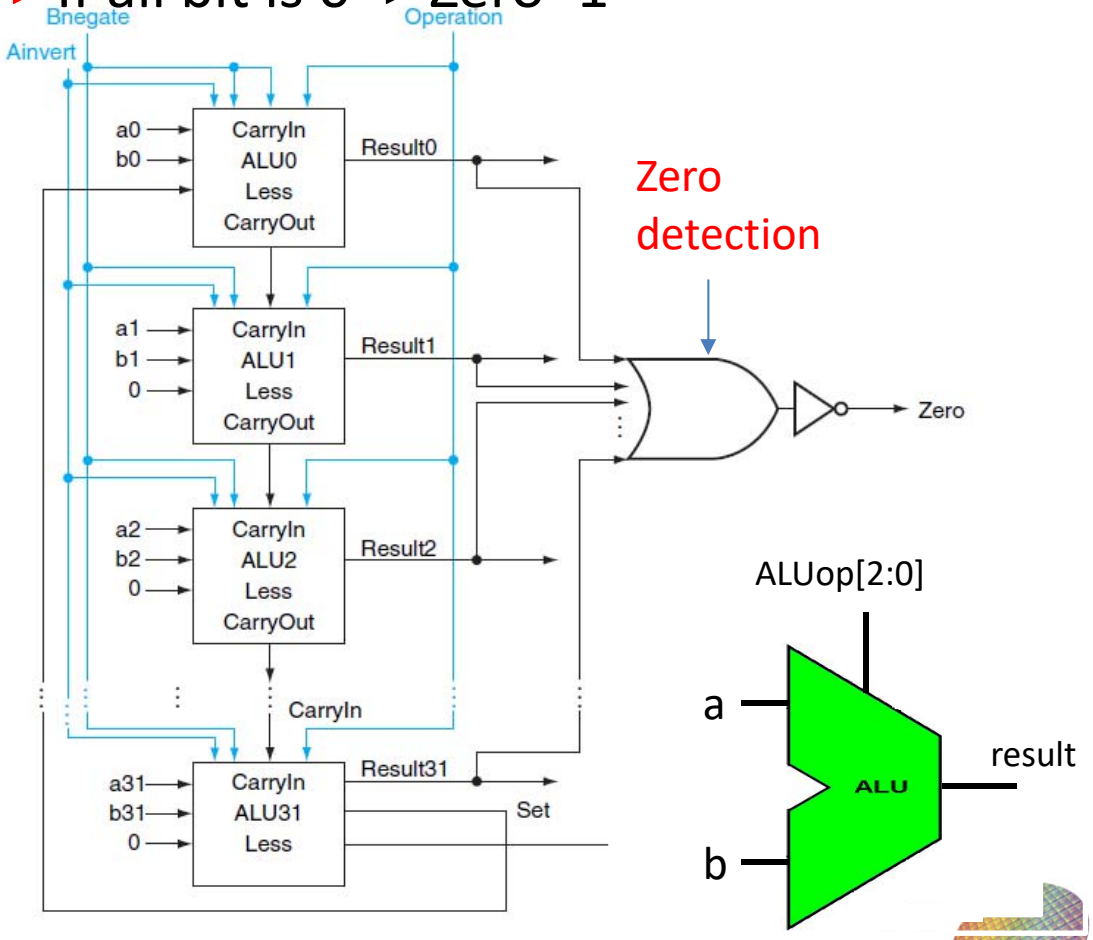


Final 32-bit ALU

- Binvert is compatible to CarryIn => Connect Binvert to CarryIn => is renamed to Bnegate
- Add Zero detection circuit => If all bit is 0=> Zero=1

Ainvert	Binvert	CarryIn	Op.	Func.
0	0	X	0	a and b
0	0	X	1	a or b
0	0	0	2	a + b
0	1	1	2	a - b
0	1	1	3	slt

Bnegate	Op[1:0]	Func.
0	00	a and b
0	01	a or b
0	10	a + b
1	10	a - b
1	11	slt



Recap: Faster adder-Carry Lookahead

- Taught in digital system design course

$$g_i = a_i b_i$$

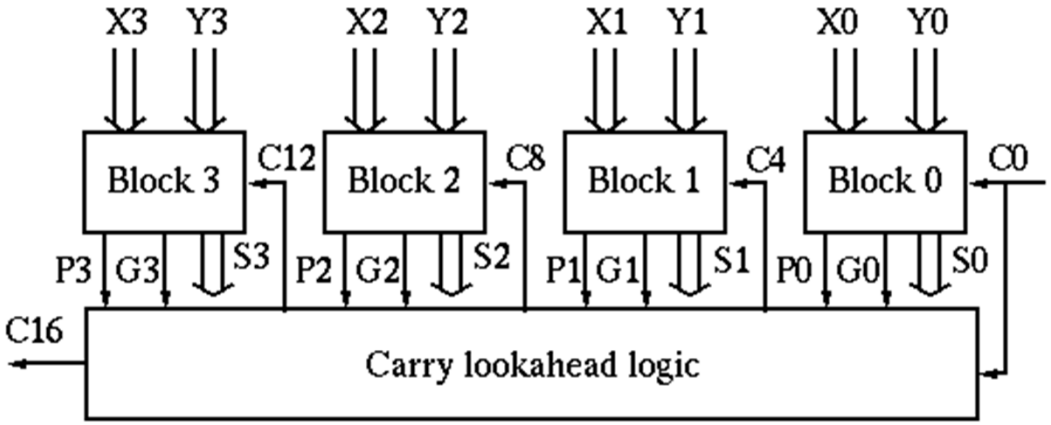
$$p_i = a_i + b_i$$

$$c1 = g0 + (p0 \cdot c0)$$

$$c2 = g1 + (p1 \cdot g0) + (p1 \cdot p0 \cdot c0)$$

$$c3 = g2 + (p2 \cdot g1) + (p2 \cdot p1 \cdot g0) + (p2 \cdot p1 \cdot p0 \cdot c0)$$

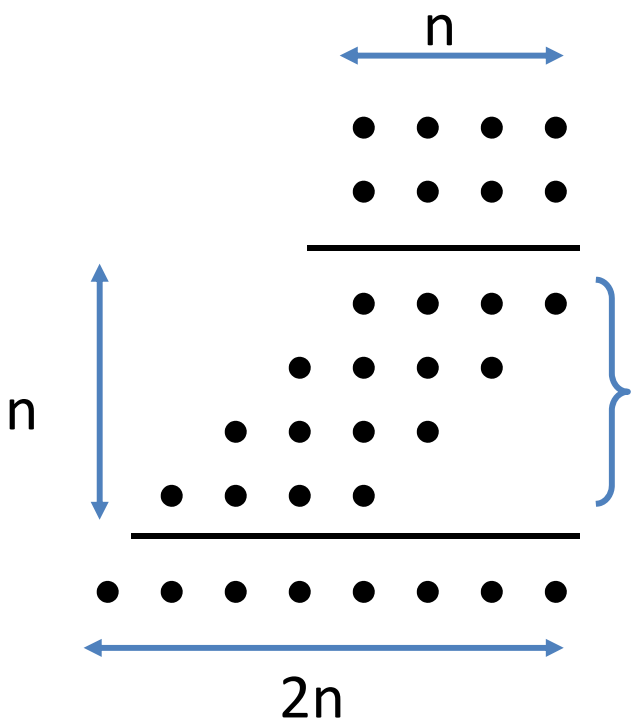
$$c4 = g3 + (p3 \cdot g2) + (p3 \cdot p2 \cdot g1) + (p3 \cdot p2 \cdot p1 \cdot g0) + (p3 \cdot p2 \cdot p1 \cdot p0 \cdot c0)$$



Multiplication

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

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



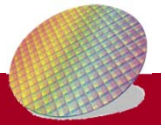
multiplicand

multiplier

partial product array

can be formed in parallel ar added in parallel for faster multiplication

double precision product



Multiplication

- Start with long-multiplication approach

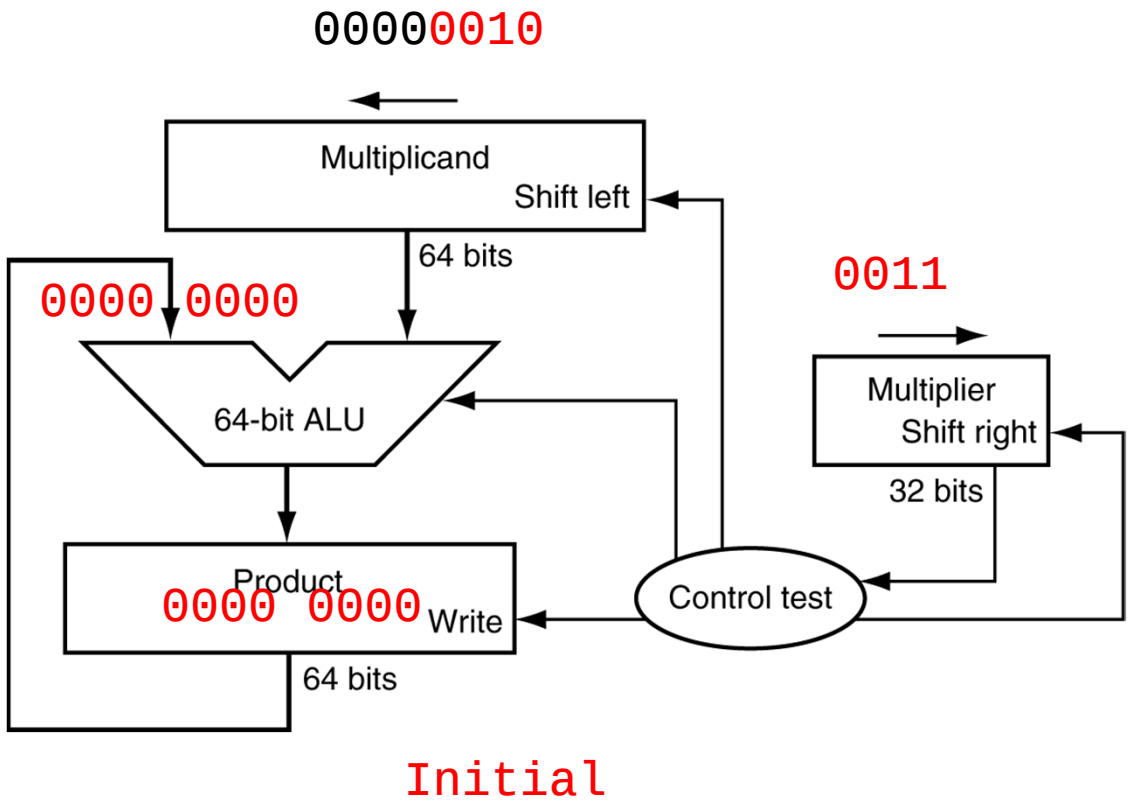
multiplicand

multiplier

product

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

Length of product is the sum of operand lengths



Multiplication

- Start with long-multiplication approach

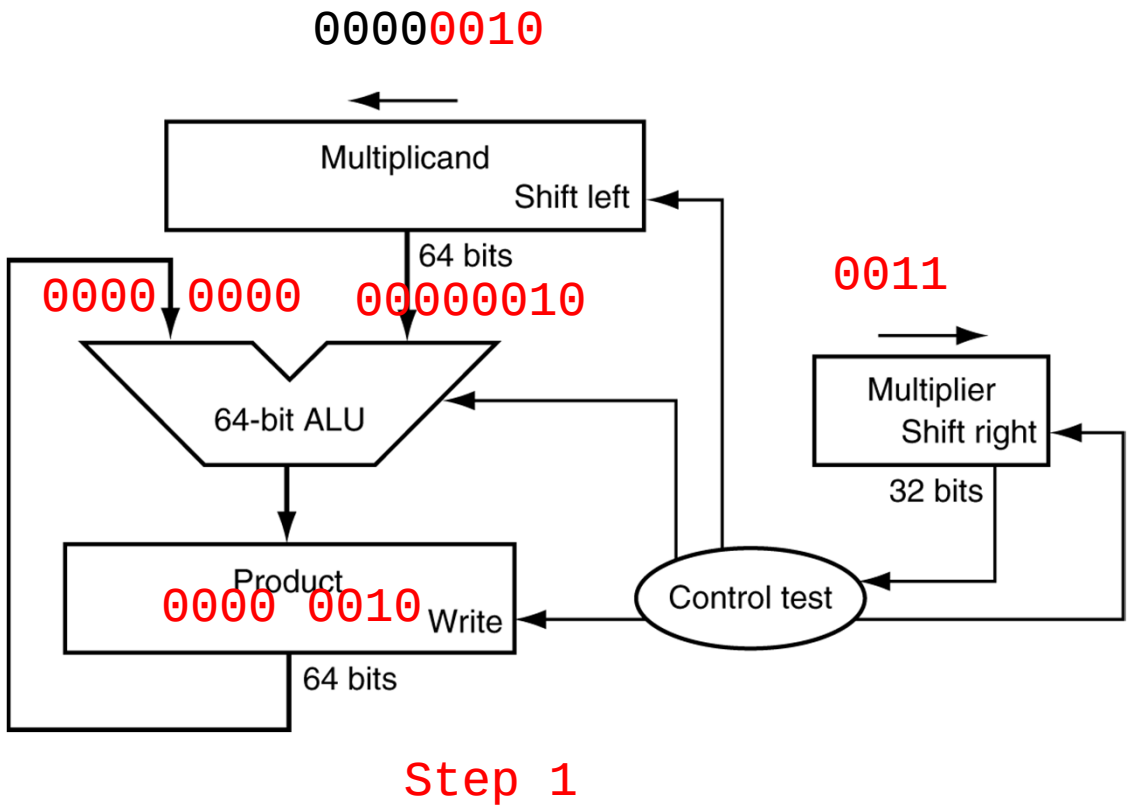
multiplicand

multiplier

product

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

Length of product is the sum of operand lengths



Multiplication

- Start with long-multiplication approach

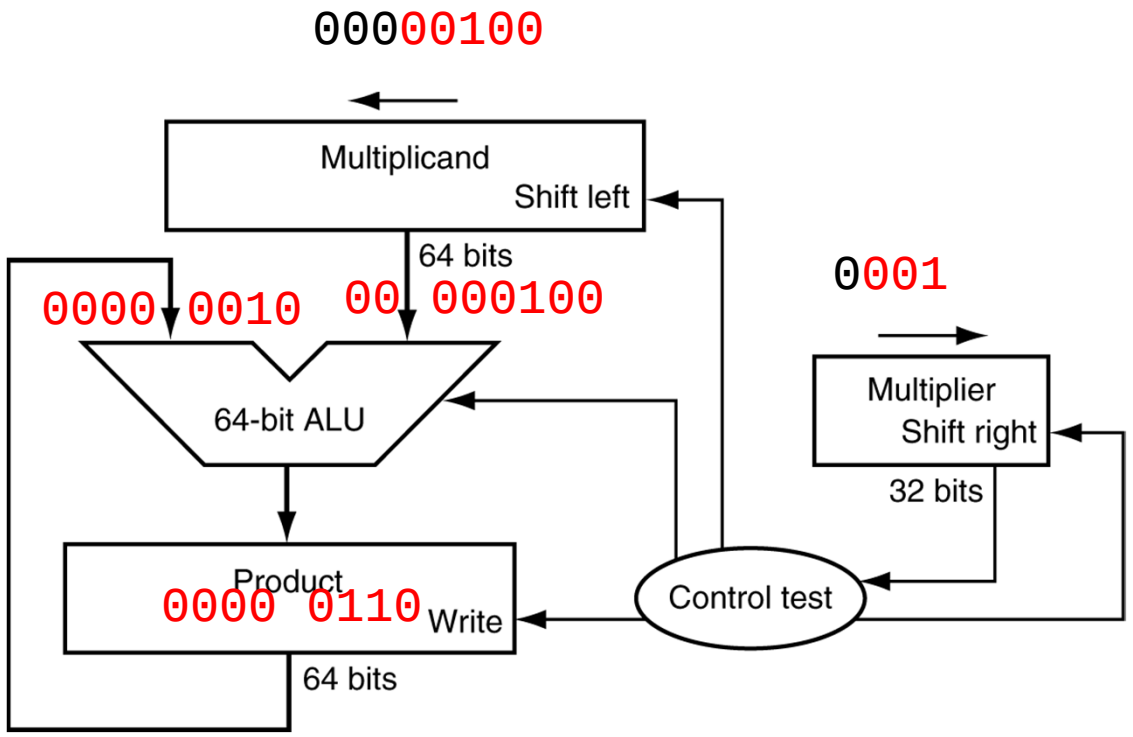
multiplicand

multiplier

product

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

Length of product is the sum of operand lengths



Step 2

Multiplication

- Start with long-multiplication approach

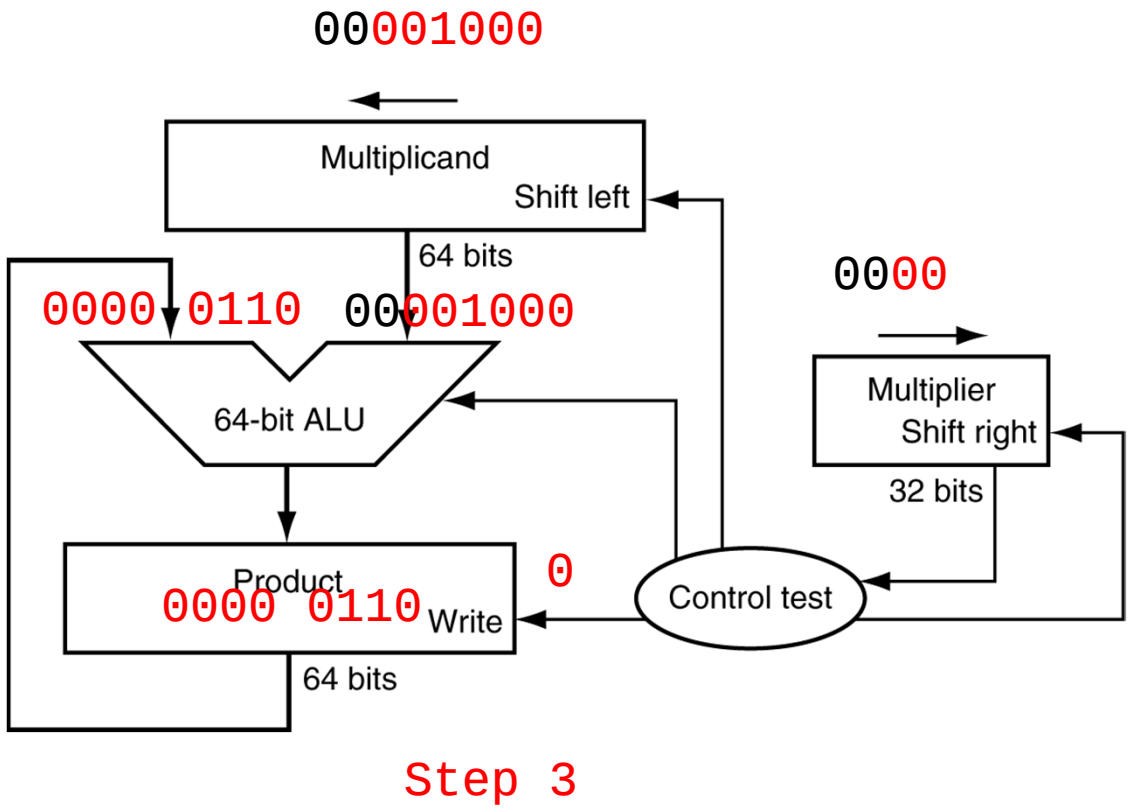
multiplicand

multiplier

product

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

Length of product is the sum of operand lengths



Multiplication

- Start with long-multiplication approach

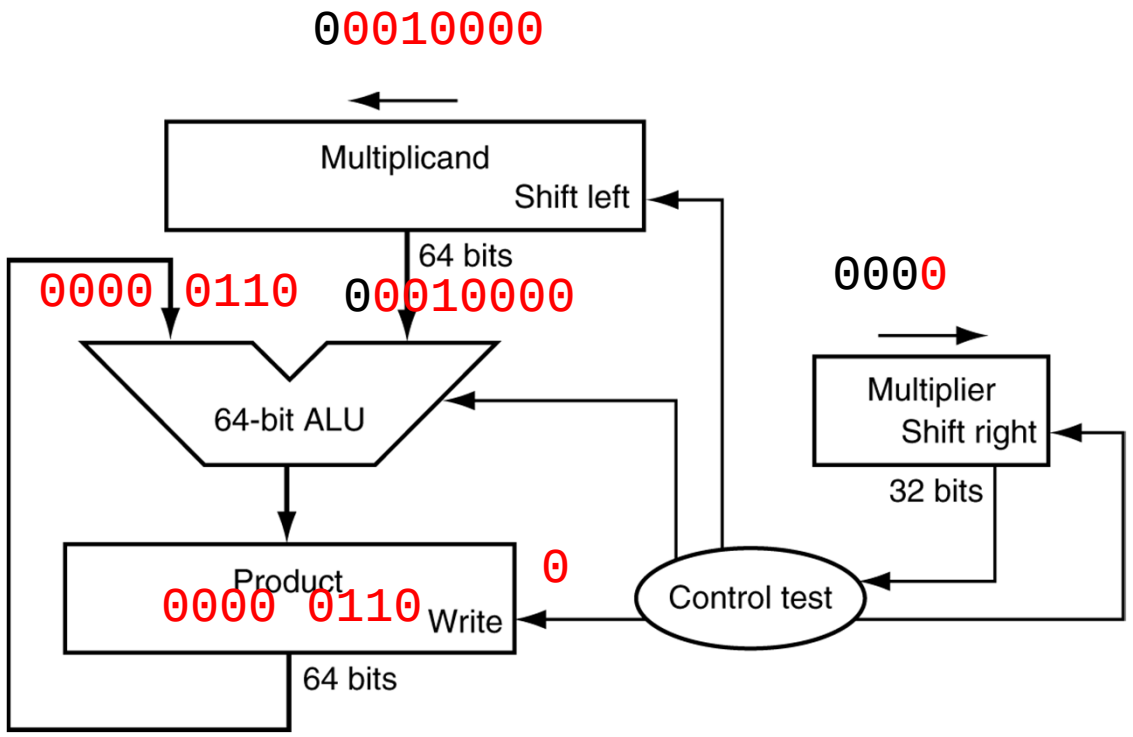
multiplicand

multiplier

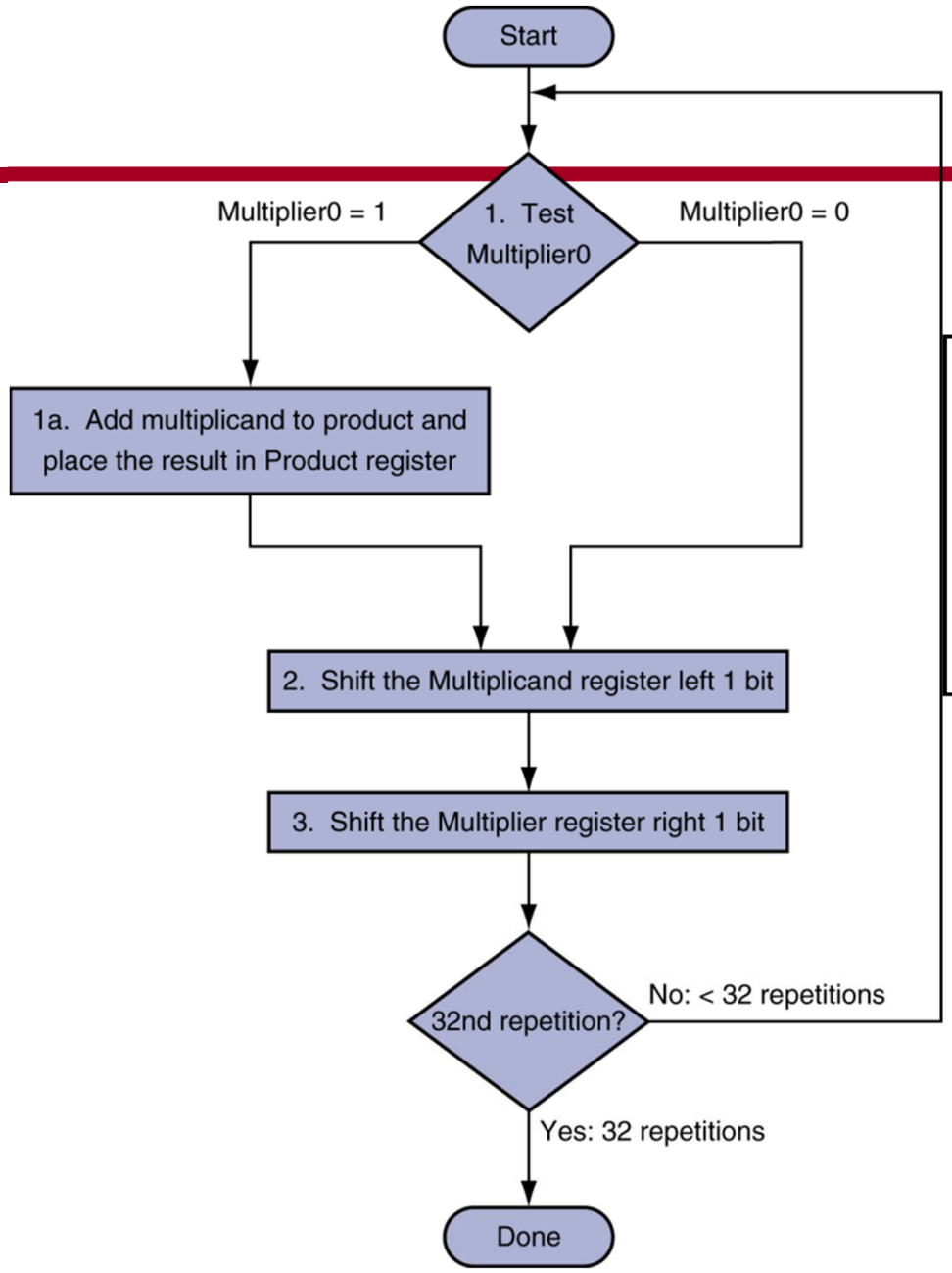
product

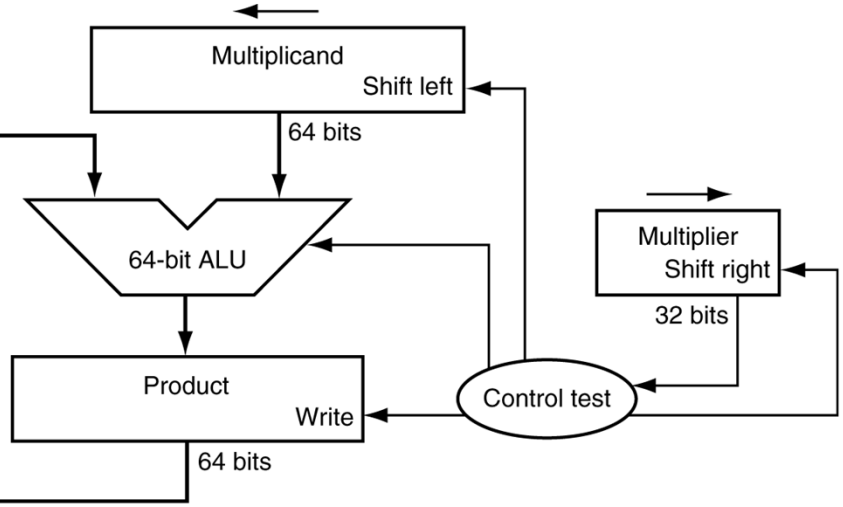
```
  0010
× 0011
------
  0010
 0010
0000
000
-------
0000110
```

Length of product is the sum of operand lengths



Multiplication 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
```

Shift multiplicand left 1 bit and add

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

Shift multiplicand left 2 bit but no add

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

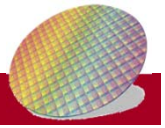
```
    1000
  × 1011
  ------
    1000
```

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

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

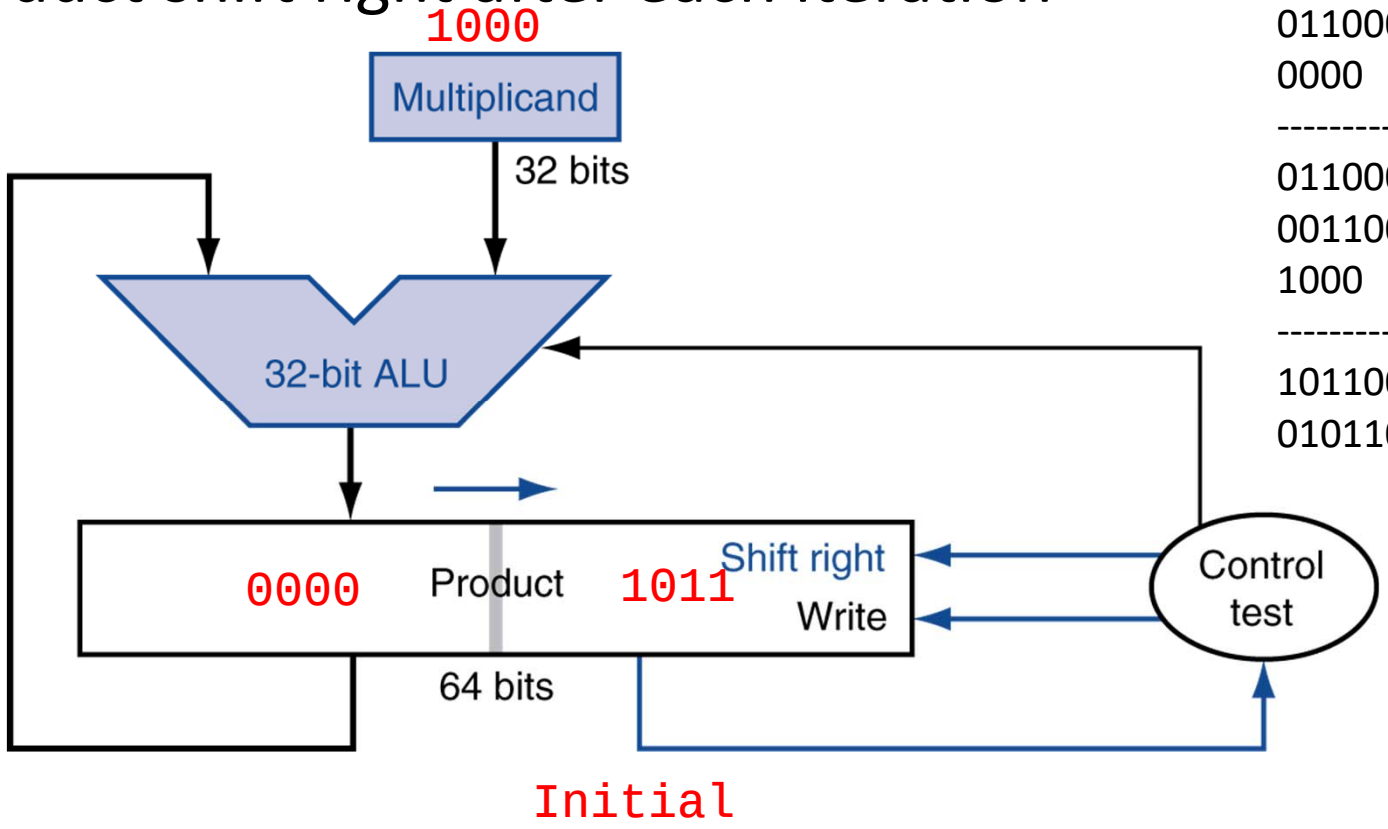
```
    1000
  × 1011
  ------
  011000
  1000
  -----
  1011000
```

product shift right and add



Optimized Multiplier

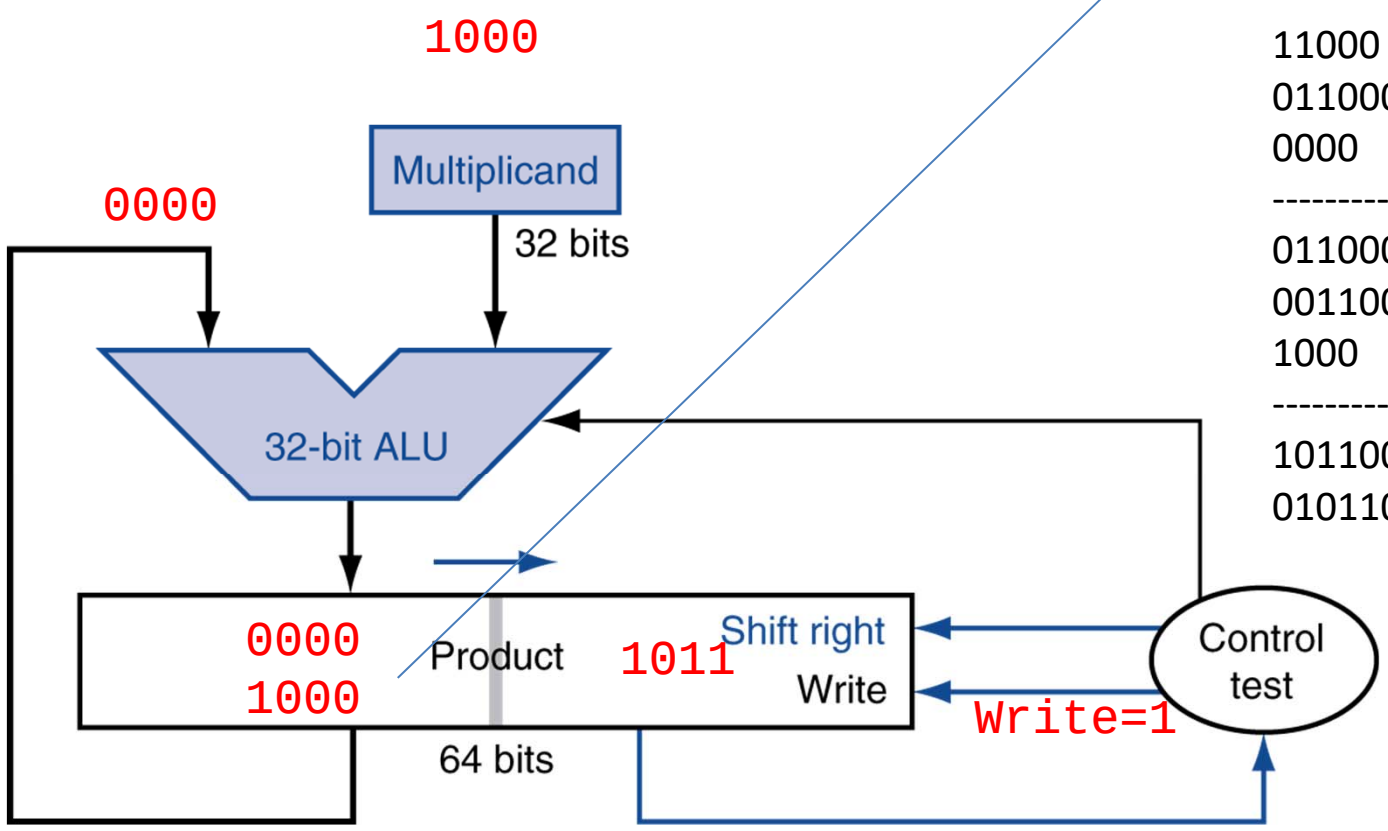
- Initial: 32 bit multiplicand, multiplier is stored in right side of product
- Product shift right after each iteration




```
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

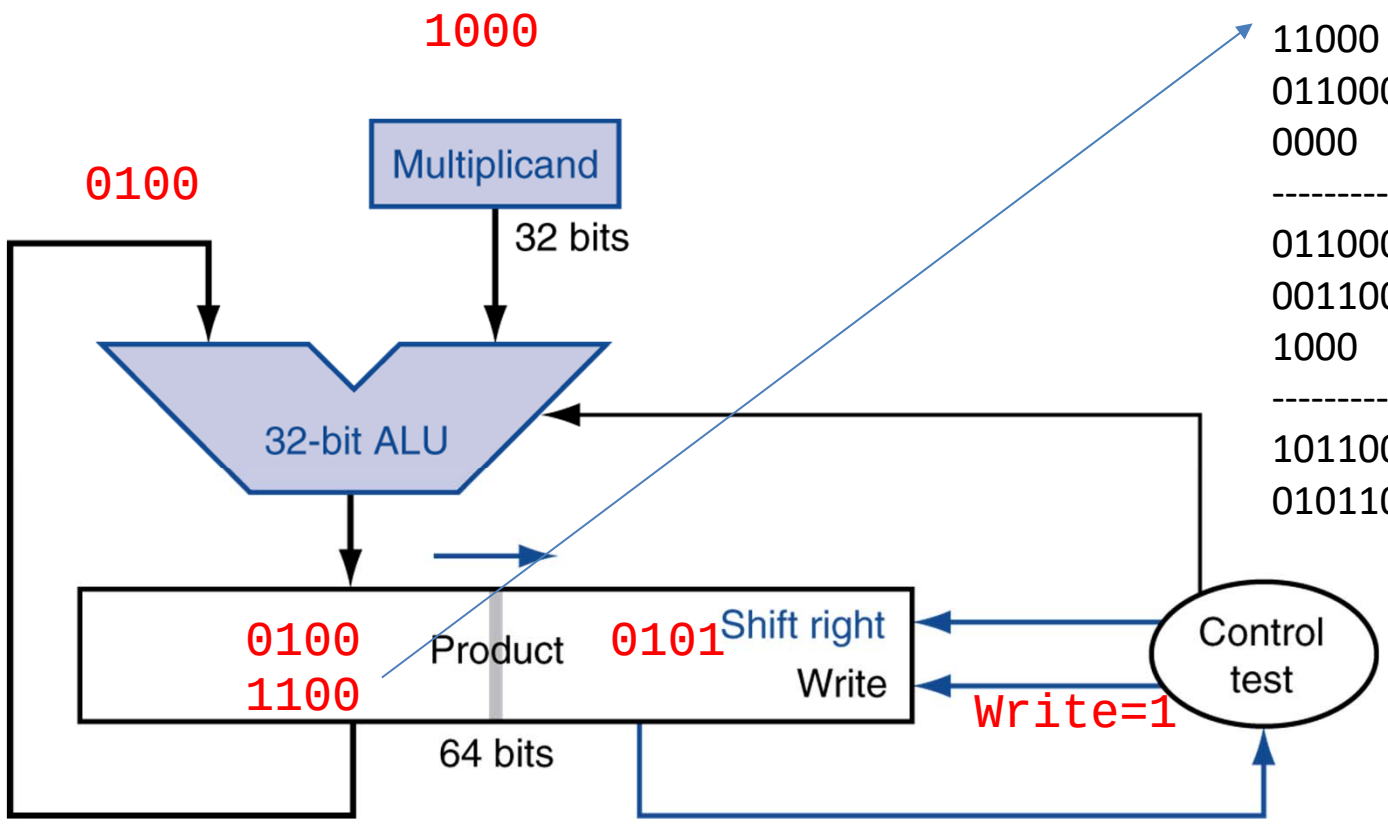
- Step 1




```
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

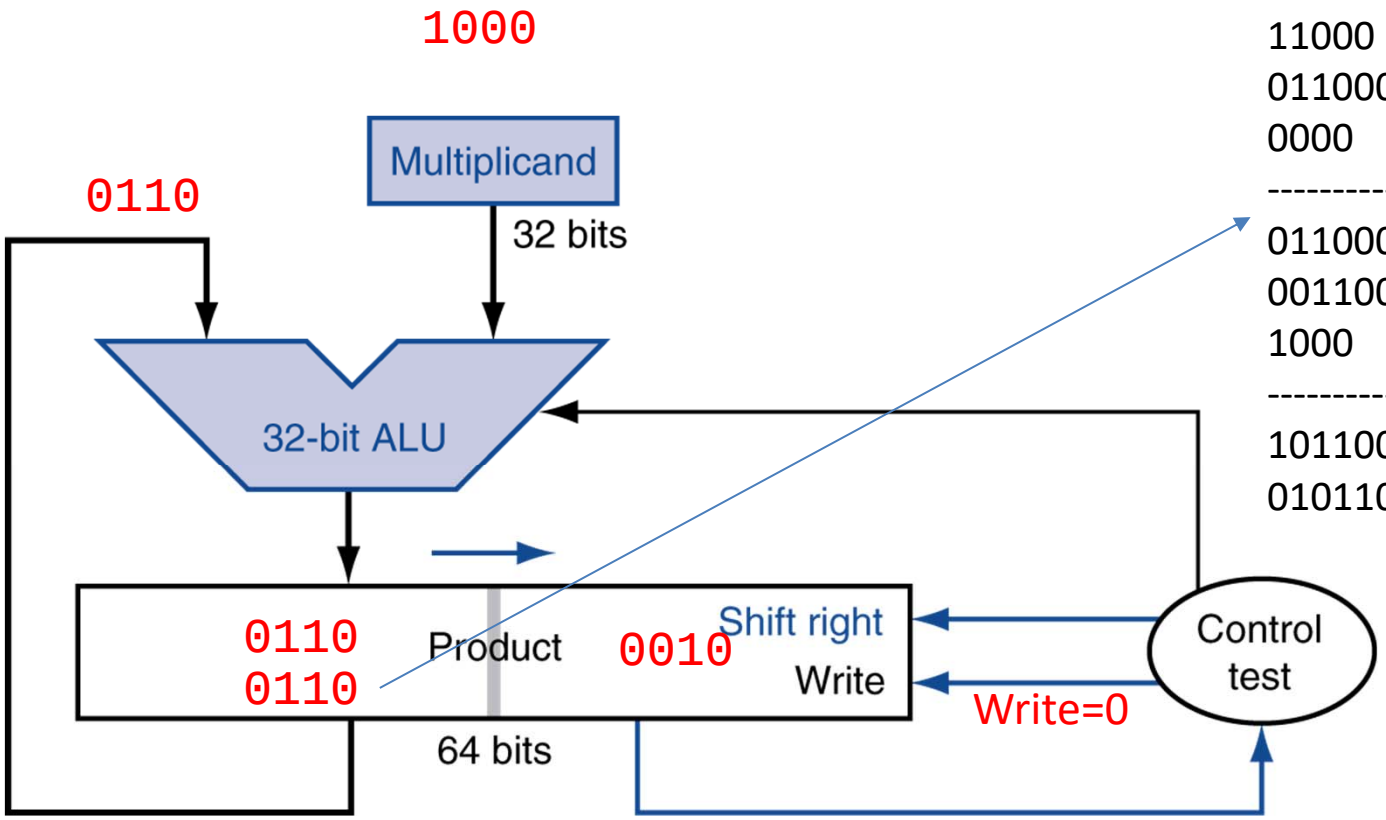
11000 101 shift
011000 10
0000 add

011000 10 shift
0011000 1
1000 add

1011000 1 shift
01011000

Optimized Multiplier

- Step 3



1000
1011 add

10001011 shift
01000 101
1000 add

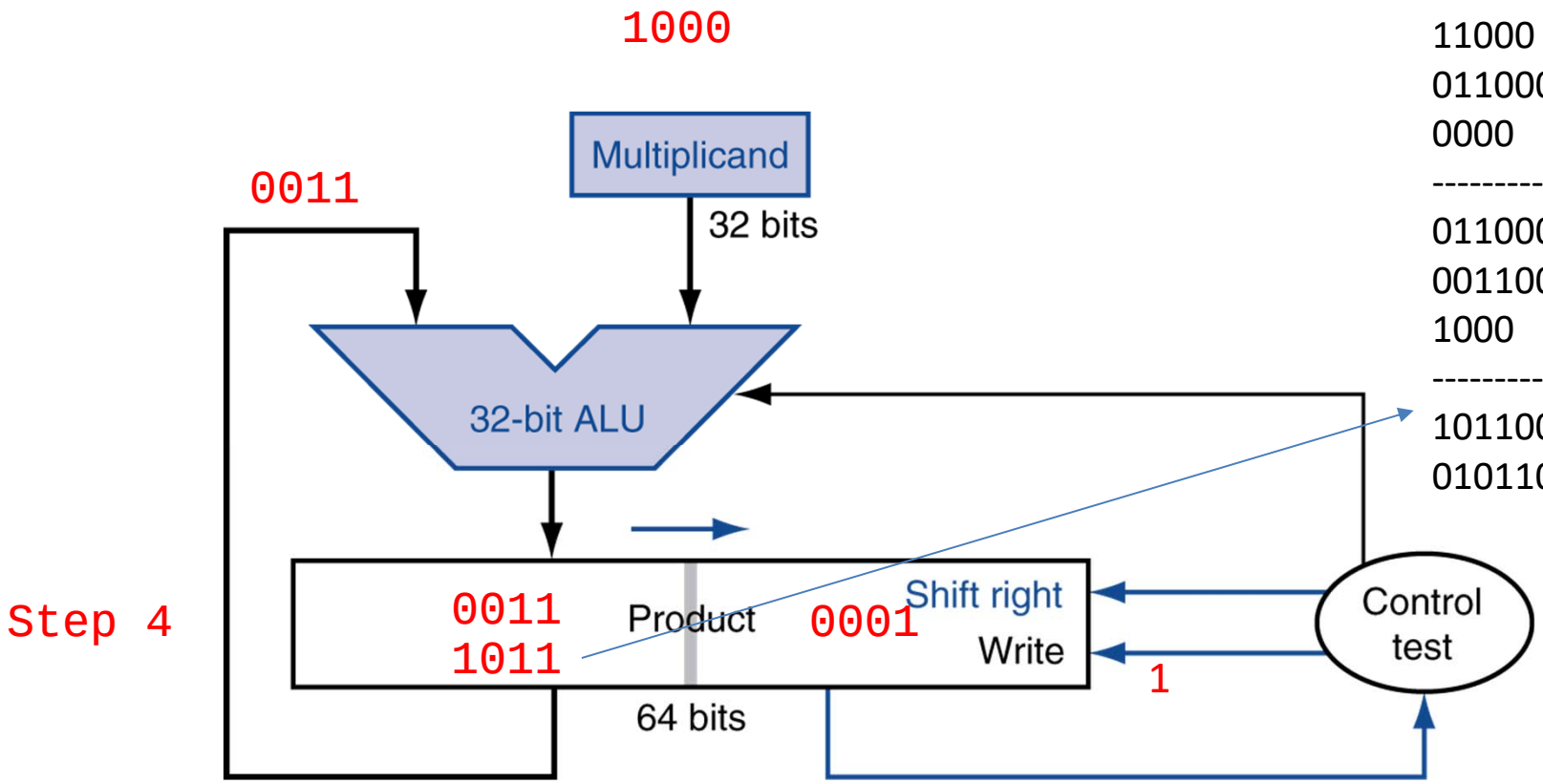
11000 101 shift
011000 10
0000 add

011000 10 shift
0011000 1
1000 add

1011000 1 shift
01011000

Optimized Multiplier

- Step 4:

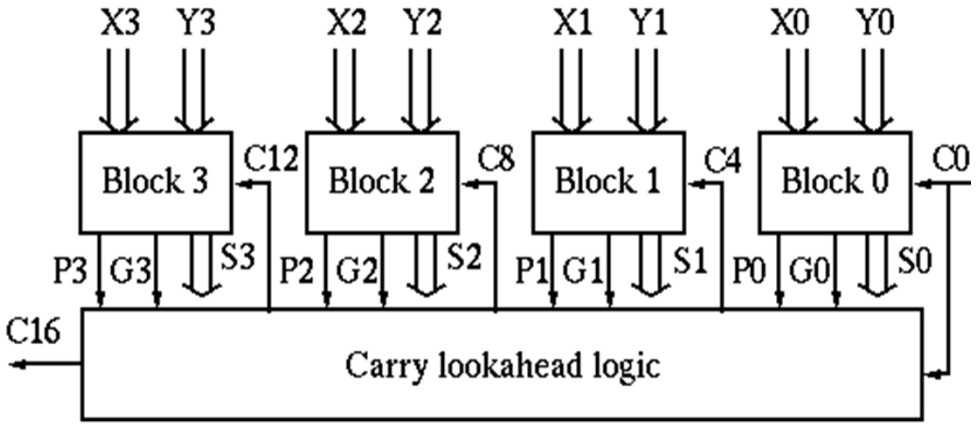



```
1000
1011        add
------------
10001011   shift
01000 101
1000        add
------------
11000 101  shift
011000 10
0000        add
----------------
011000 10   shift
0011000 1
1000        add
-----------------------
1011000 1 shift
01011000
```

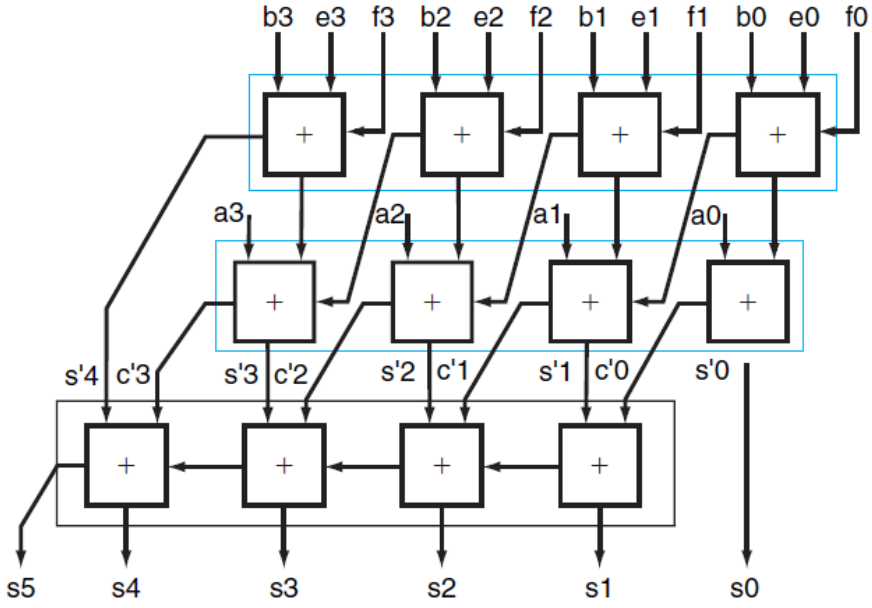
Final product: 01011000

Faster Multiplier

- Uses faster adder
  - Addition is repetitively performed
  - Faster adder can improve multination speed
  - E.g. carry lookahead adder, carry save adder, etc.
  - See appendix C



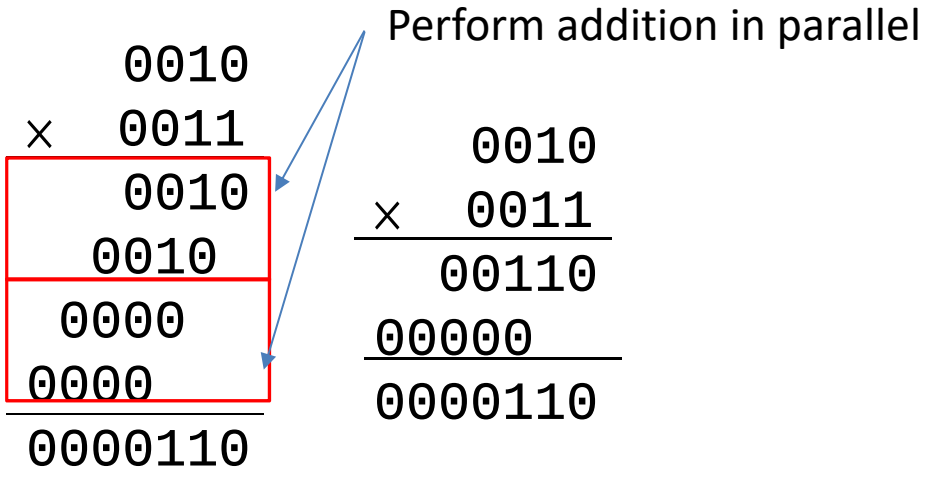
Carry lookahead adder

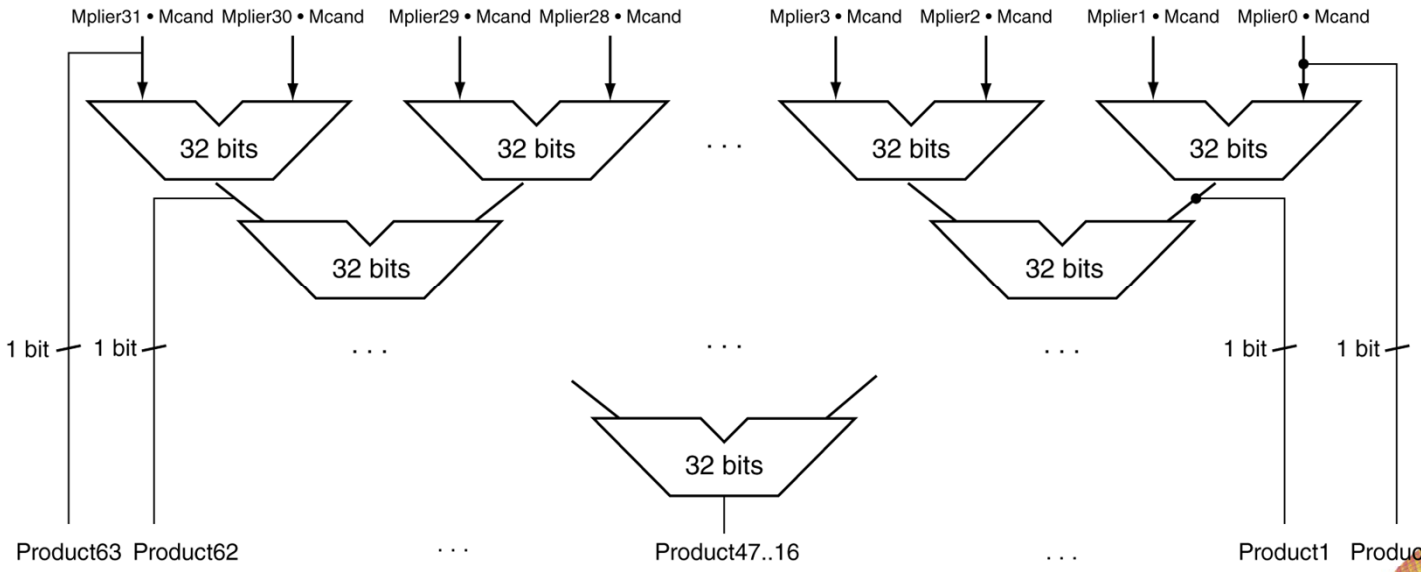


Carry save adder

Faster Multiplier

- Perform addition in parallel
  - Uses multiple adders
  - Can be pipelined to reduce critical paths





MIPS Multiplication-mul

- mul

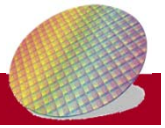
mul $rd $rs $rt

$rd = $rs * $rt

Only low-order 32 bits of the product is preserved in $rd

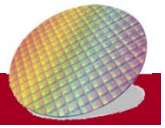
Without Overflow

```
.text
    addi $s0, $zero, 10
    addi $s1, $zero, 4
    mul $t0, $s0, $s1
    li $v0, 1
    add $a0, $zero, $t0
    syscall
```



MIPS Multiplication-mult

- If the product may be larger than 32 bit => use mult
- 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
    - Can test HI value to see if product overflows 32 bits

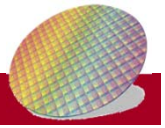


Hidden

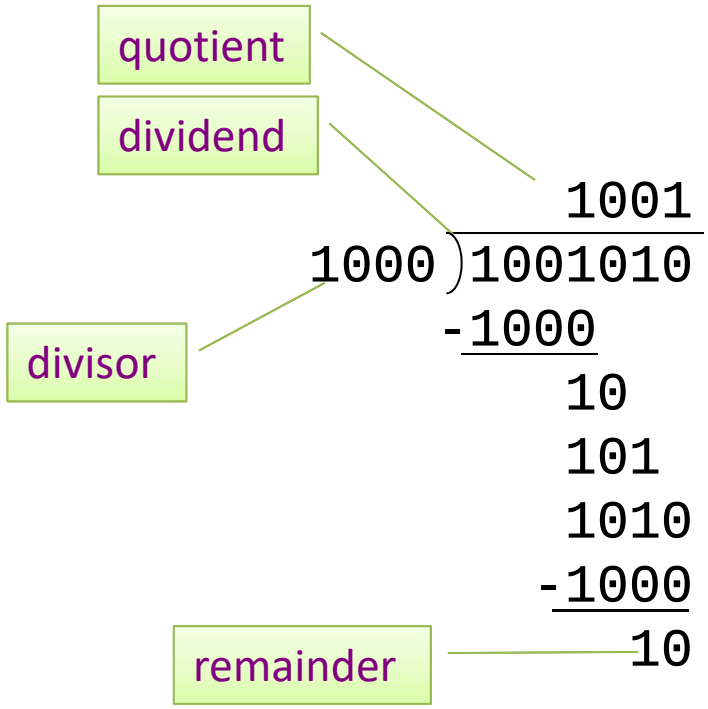
Example

- Write a program that evaluates the formula 5*12 - 74.

```
## Program to calculate 5*12 - 74
.text
        addi    $t0, $0, 12        # put 12 into $t0
        addi    $t1, $0,  5        # put 5 into $t1
        mult    $t0, $t1           # lo = 5x12
        mflo    $t1                # $t1 = 5x12
        addi   $t1, $t1,-74        # $t1 = 5x12 - 74
```

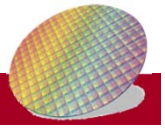


Division

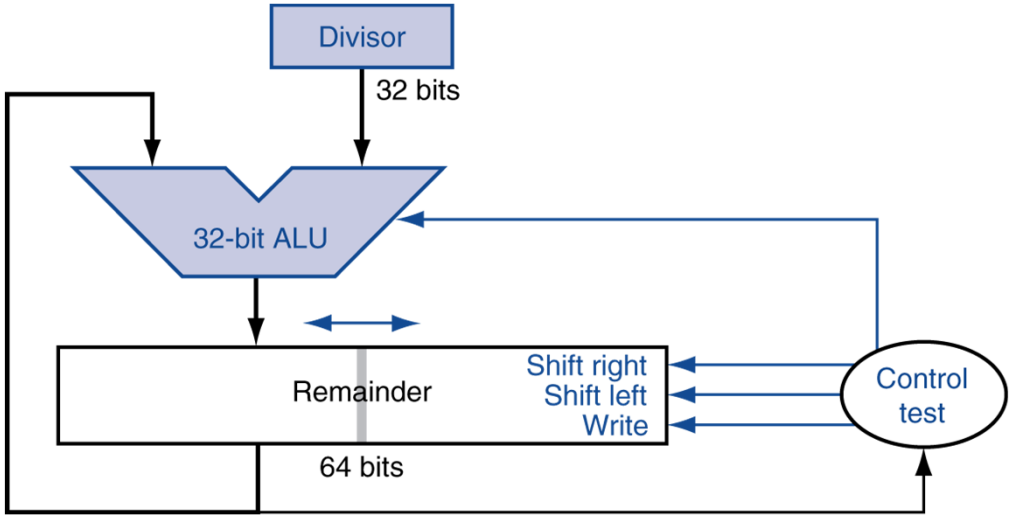


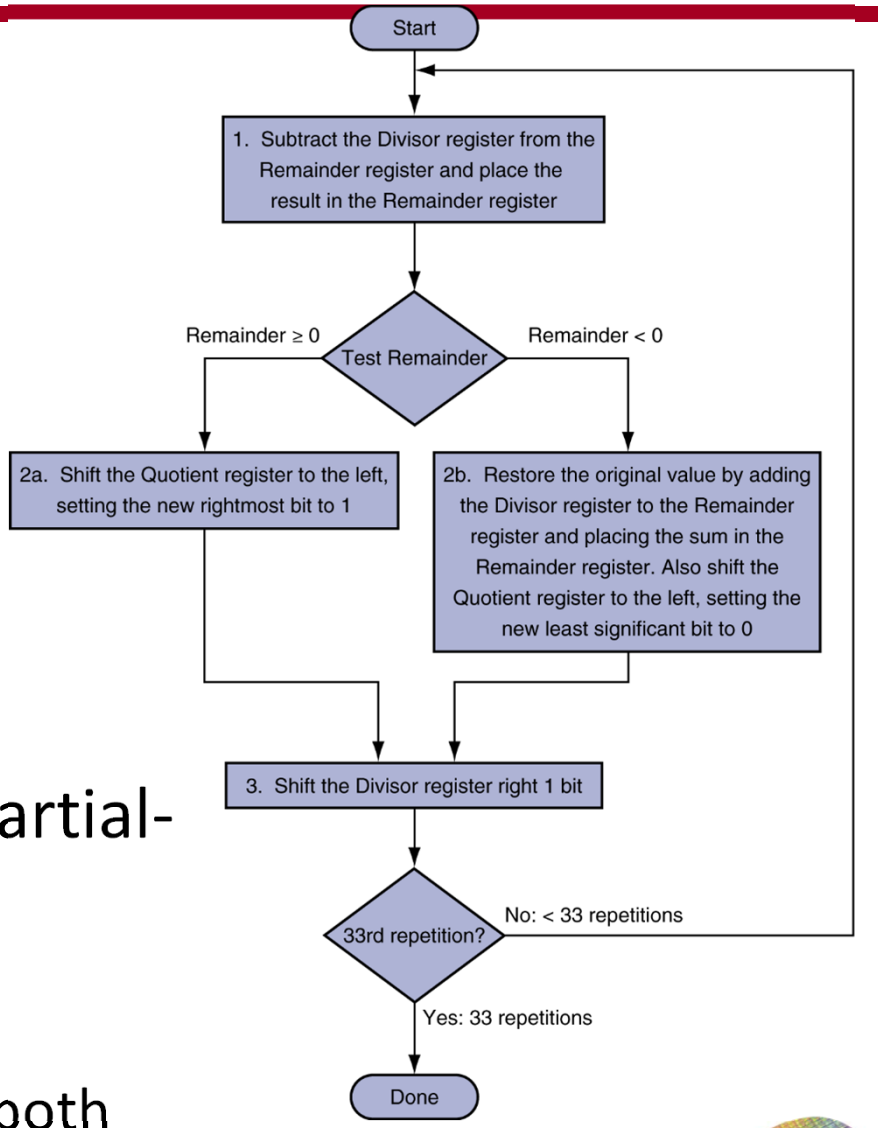
n-bit operands yield *n*-bit quotient and remainder

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



First version of Division hardware

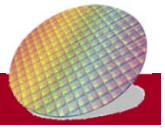




- One cycle is needed for each partial-remainder subtraction
- Looks a lot like a multiplier!
  - Same hardware can be used for both

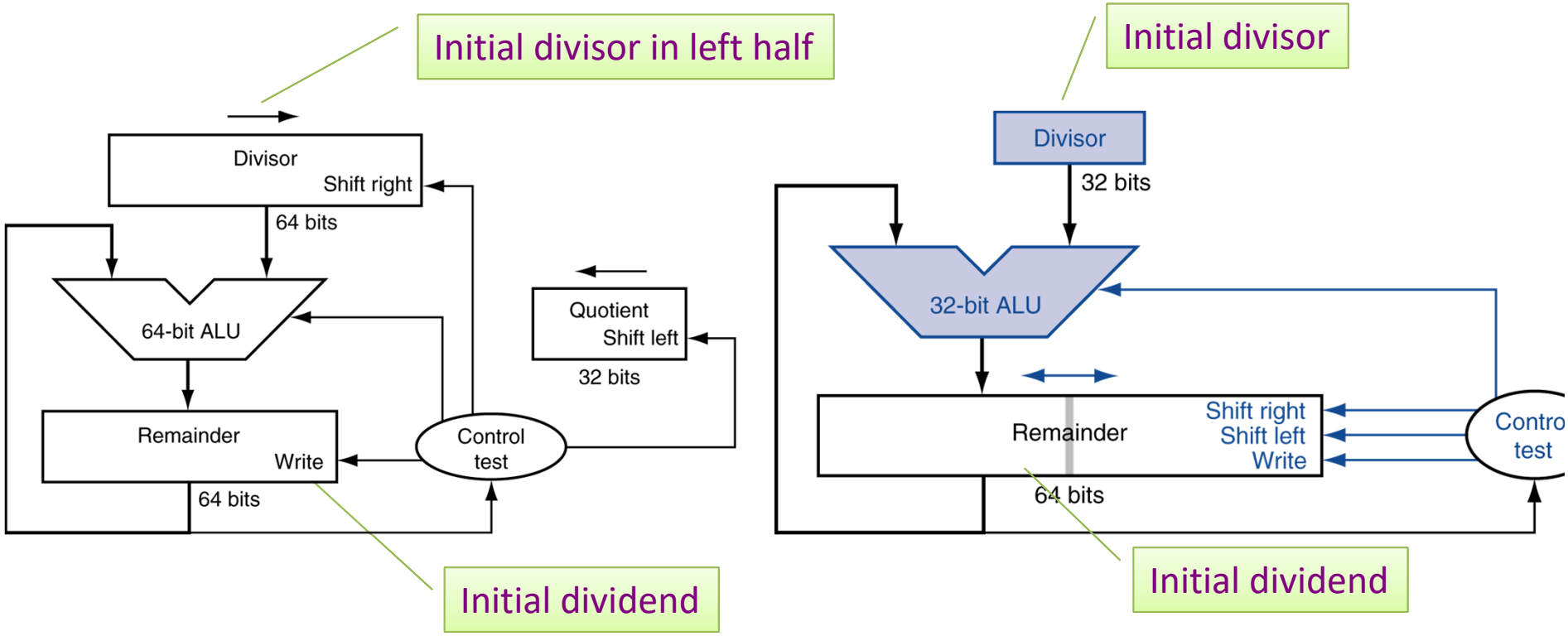
Restoring Division Example

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



Optimized division hardware

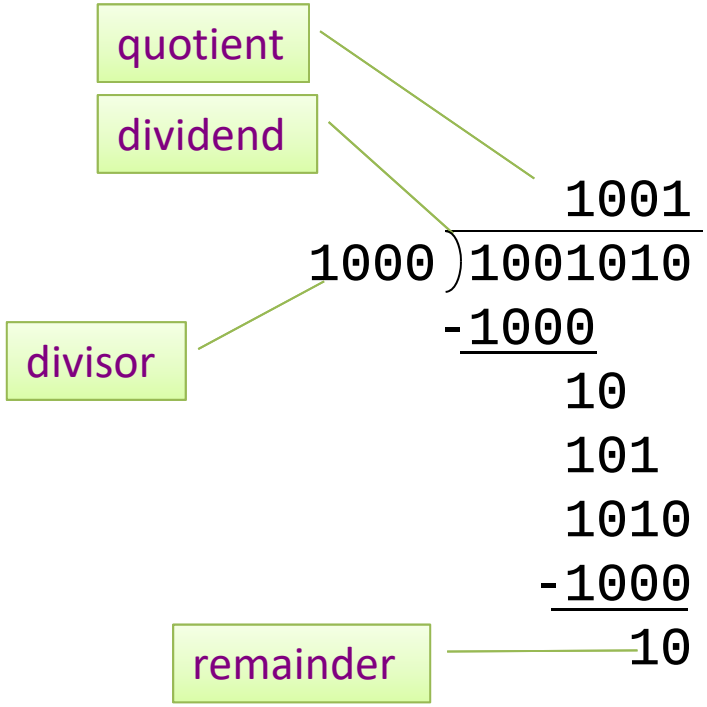
- Division is similar to multiplication, so is hardware



Improved hardware
32-bit Divisor
No extra bit for Quotient

Why Division is slower?

- Division is slower than multiplication because
  - Need reminder to decide next quotient bit
  - Division is done sequentially
  - Can't be done in parallel
- Different Division (skipped)
  - Restoring
  - Nonrestoring
  - SRT



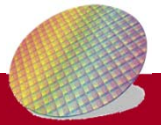
n-bit operands yield *n*-bit quotient and remainder

MIPS Division

- Use HI/LO registers for result
  - HI: 32-bit remainder
  - LO: 32-bit quotient
- Instructions (only two operands)
  - `div rs, rt`
  - `divu rs, rt`

```
div  $s0, $s1   # lo = $s0 / $s1
                # hi = $s0 mod $s1
mfhi    $t0   #move reminder to $t0
mflo    $t1   #move quotient to $t1
```

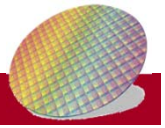
  - Use mfhi rd and mflo rd are provided to move the quotient and reminder to user accessible registers
  - No overflow or divide-by-0 checking
    - Software must perform checks if required



Division 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
                         #  used to get at result of product or quotient
```



Backup slides