

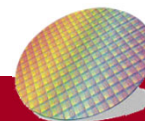


成功大學

National Cheng Kung University

Chapter 3

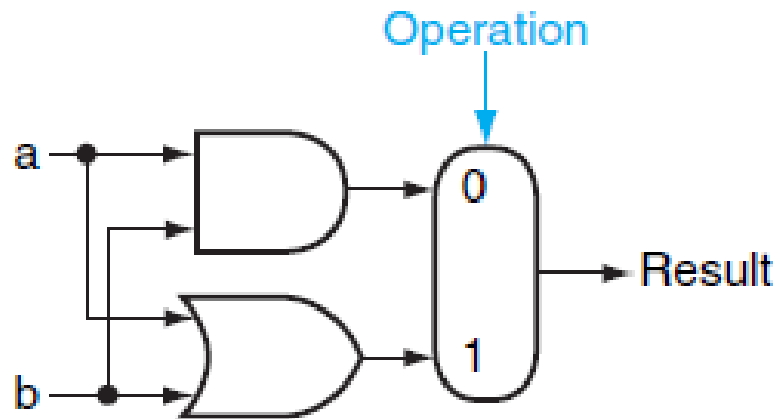
Arithmetic for Computers



Basic Arithmetic Logic Unit

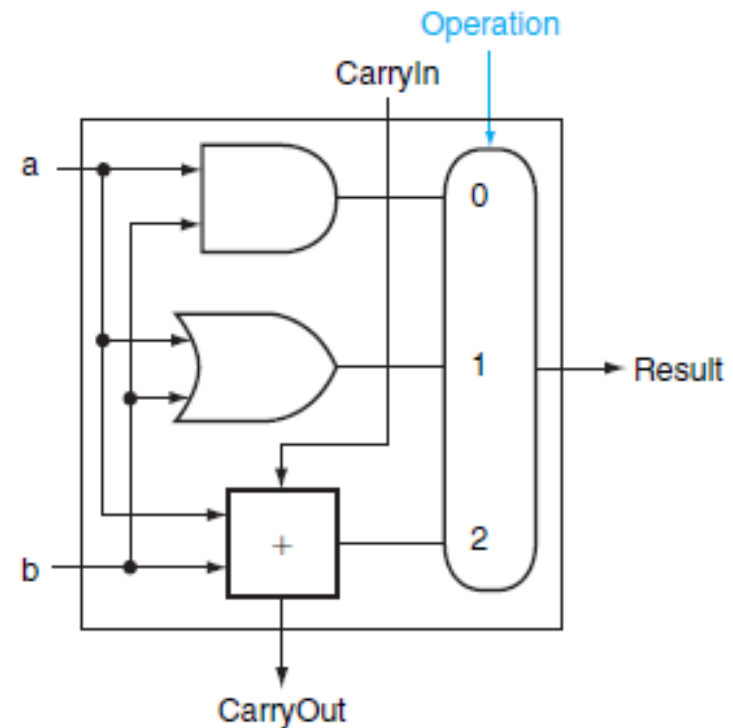
Contents adopted from B.5 (**Constructing a Basic Arithmetic Logic Unit**)

- Basic ALU

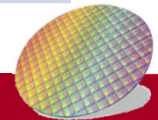


One-bit ALU that performs **AND** and **OR**

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

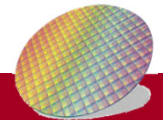
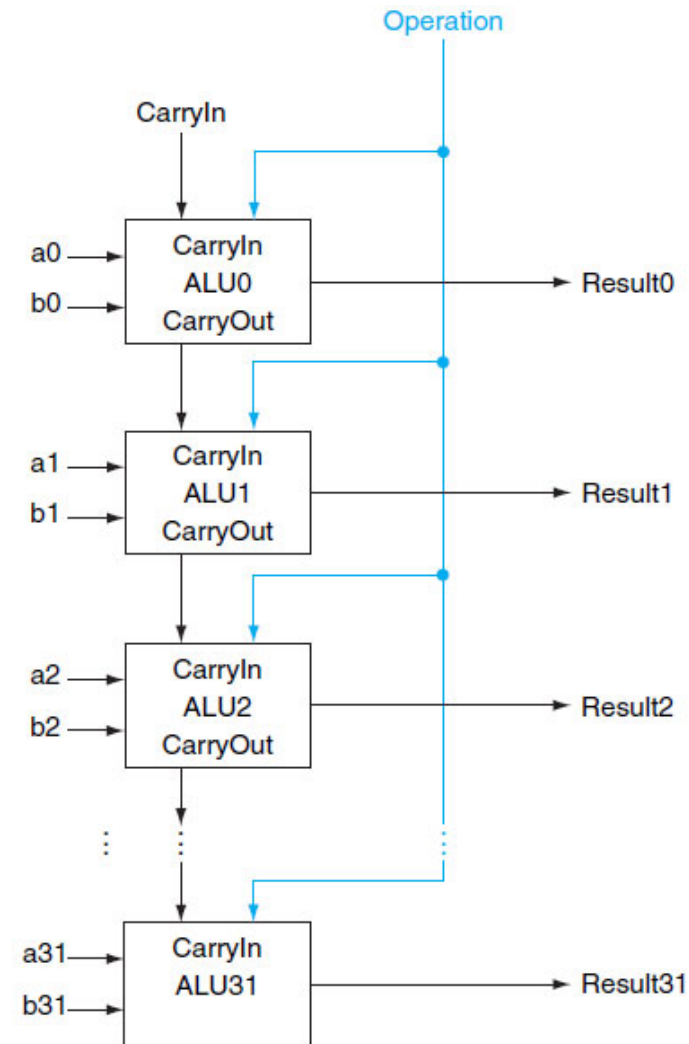


Operation(Op.)	Funct.
0	a AND b
1	a OR b
2	a + b



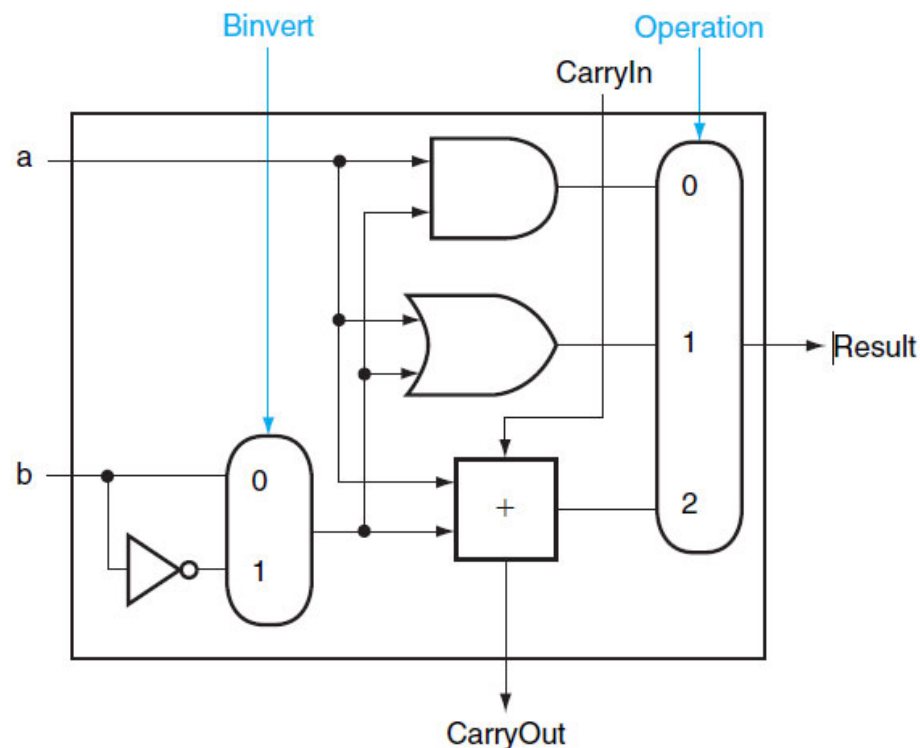
32-bit ALU

- Cascading 1-bit ALU to 32-bit ALU
- “CarryOut” is the carry-in of the next bit



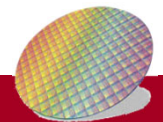
Enhanced Arithmetic Logic Unit

- 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**



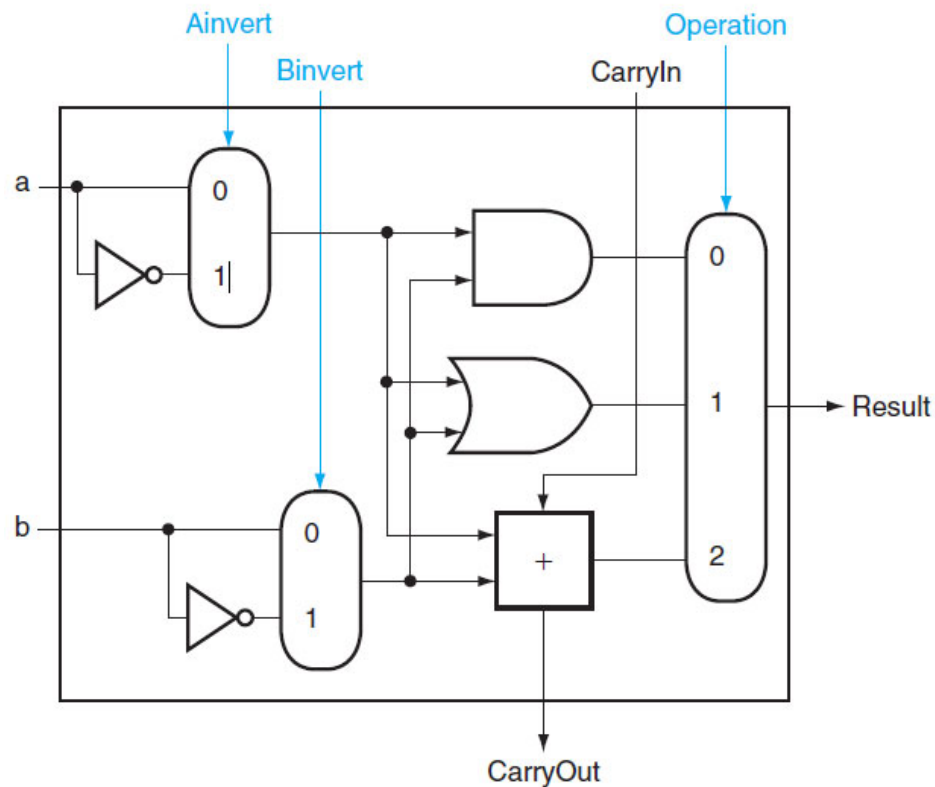
$$a - b = a + \bar{b} + 1$$

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

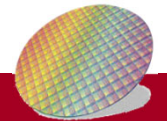
- Enhanced with **NOR** and **NAND**



$$\overline{a \cap b} = \bar{a} \cup \bar{b}$$

$$\overline{a \cup b} = \bar{a} \cap \bar{b}$$

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	$\overline{a \cup b}$
1	1	X	1	$\overline{a \cap b}$



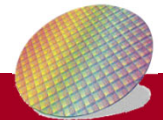
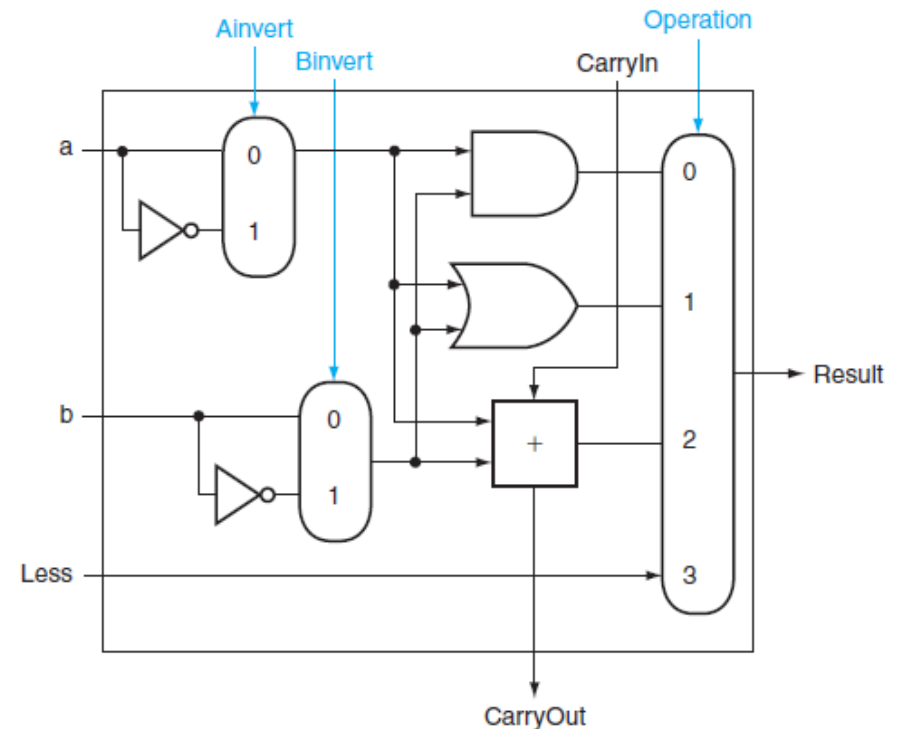
ALUs with Set Less Than

Review: slt (Set less than) instruction

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**, therefore
 - Connect to the **signed** bit of MSB (See next slide)
 - Other signals are assigned to 0

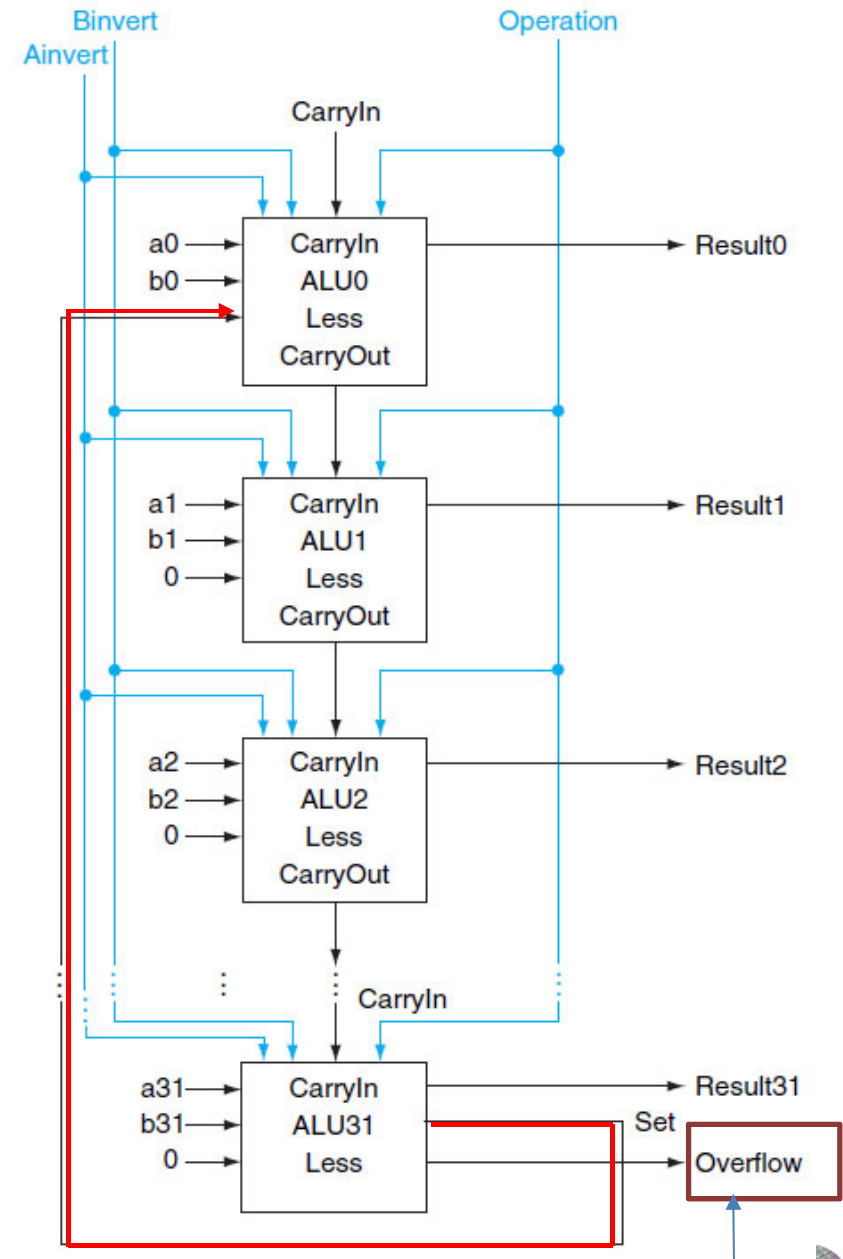


32-bit ALU with Set Less than

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

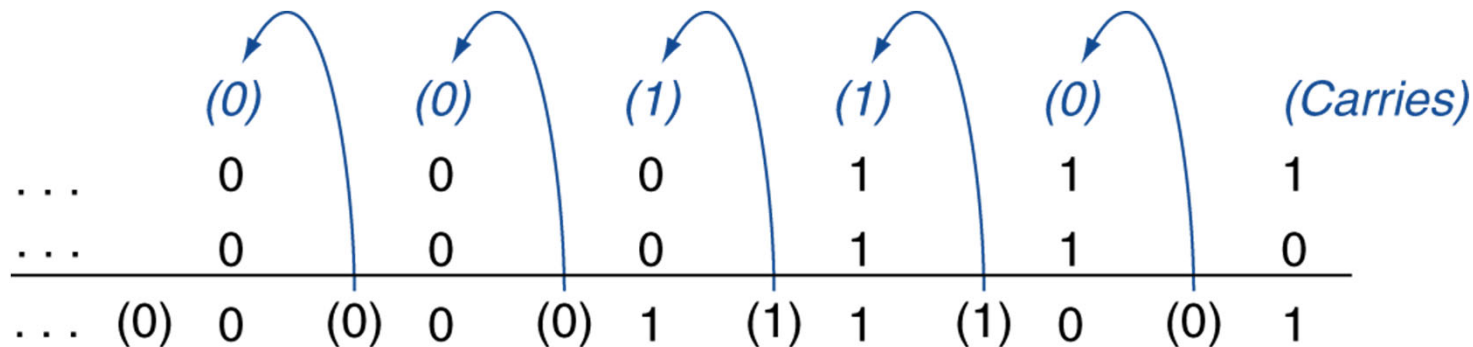
When $a_{31} \dots a_0 < b_{31} \dots b_0$, result is
00.....0001,
otherwise result is
00.....0000

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



Integer Addition and Subtraction

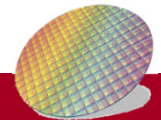
- Addition Example: $7 + 6$



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

$$\begin{array}{r}
 00000111 \\
 - 00000110 \\
 \hline
 00000001
 \end{array}$$

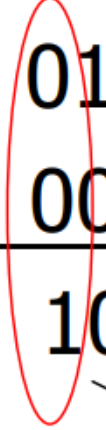
$$\begin{array}{r}
 +7: 0000\ 0000\ \dots\ 0000\ 0111 \\
 +(-6): 1111\ 1111\ \dots\ 1111\ 1010 \\
 \hline
 +1: 0000\ 0000\ \dots\ 0000\ 0001
 \end{array}$$

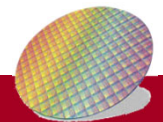


Overflow

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

$$\begin{array}{rcl}
 \text{Bit } 31 & 30 & \\
 0111 & 1111 \dots 1111 & = 2,147,483,647 \\
 +) & 0000 & 0000 \dots 0010 = 2 \\
 \hline
 10 \dots 0001 & & = -2,147,483,647
 \end{array}$$


 check sign bit



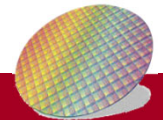
Overflow condition

- The following table shows situation that overflow occurs for signed integers
- Instruction that **causes** exception (an unscheduled event that disrupts program execution) 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

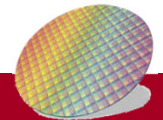
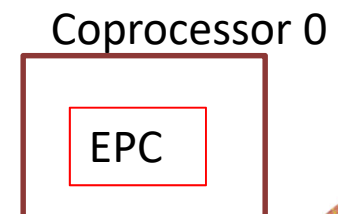
$$\begin{array}{r} 0111 \\ +0001 \\ \hline 1000 \end{array} \quad 7+1 \neq -8$$

$$\begin{array}{r} 1111 \\ +1000 \\ \hline 0111 \end{array} \quad -1+(-8) \neq 7$$

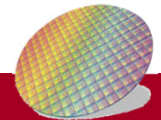
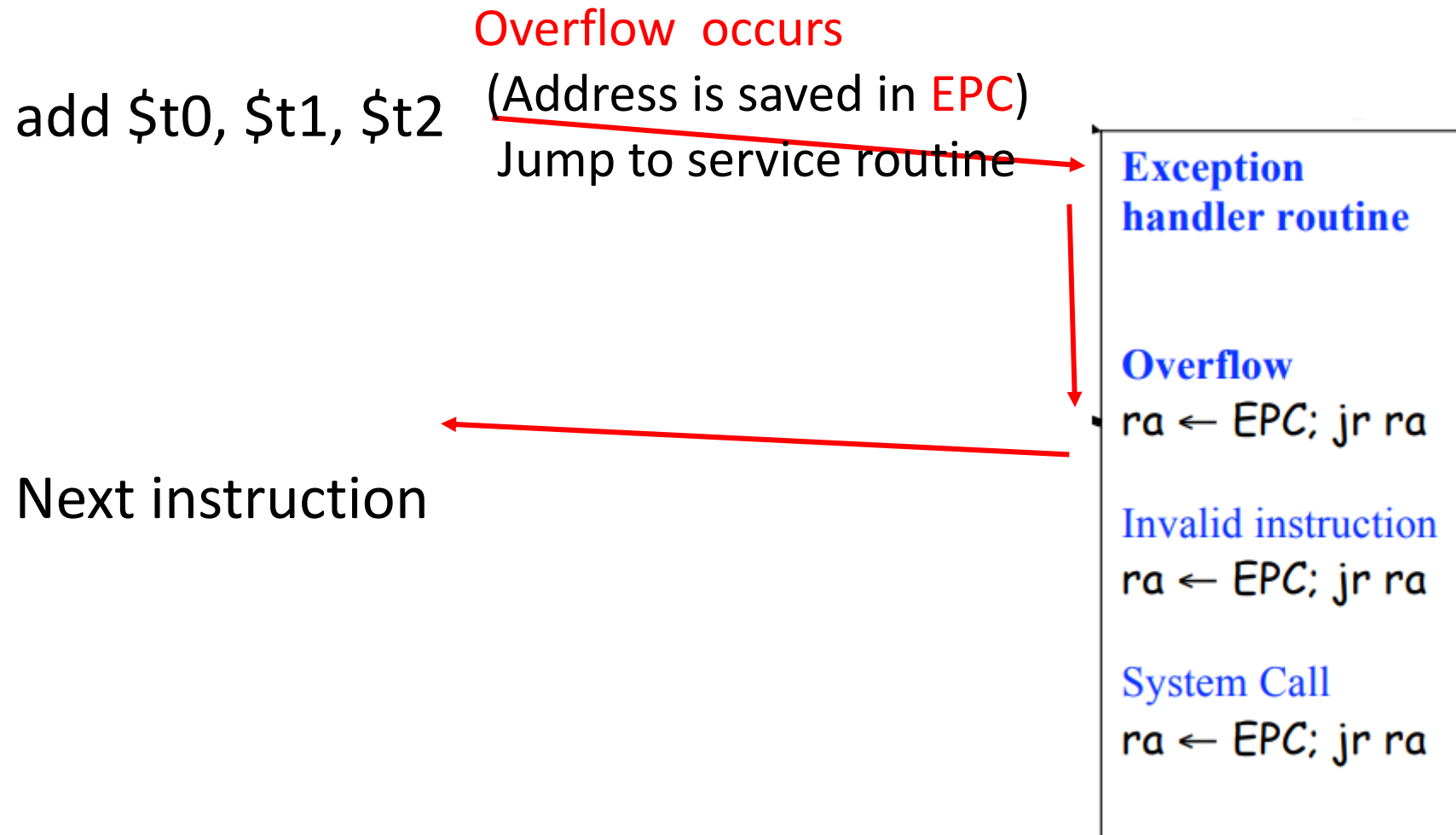


How to handle overflow in HW?

- 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

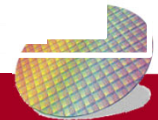
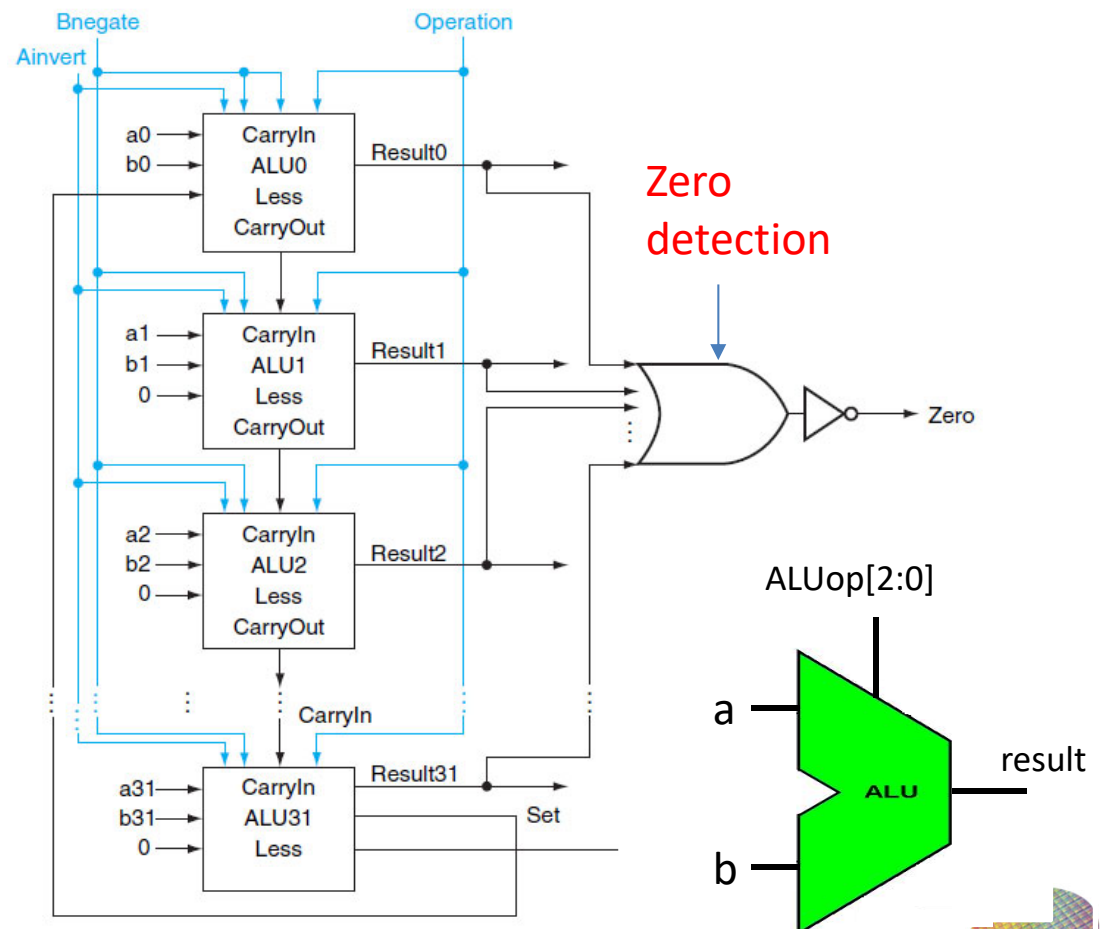


Final 32-bit ALU

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

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

$$g_i = a_i b_i$$

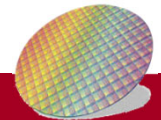
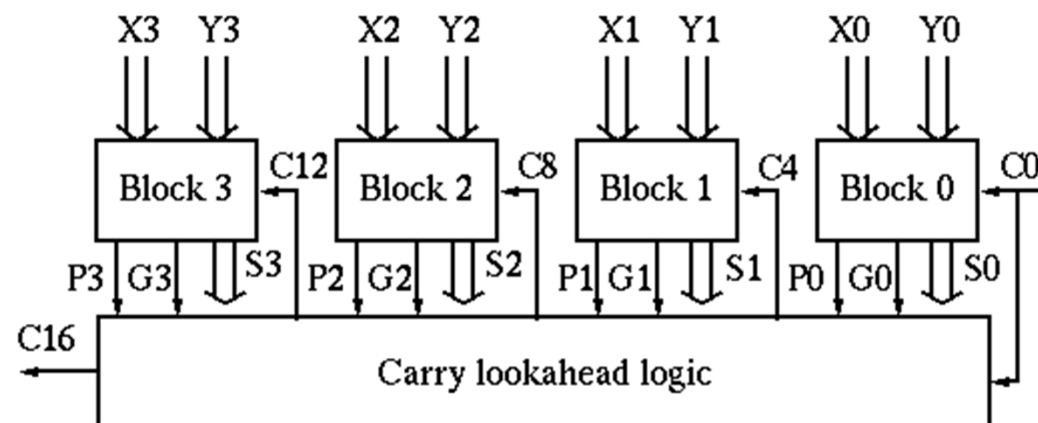
$$p_i = a_i + b_i$$

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

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

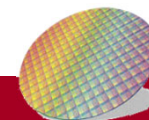
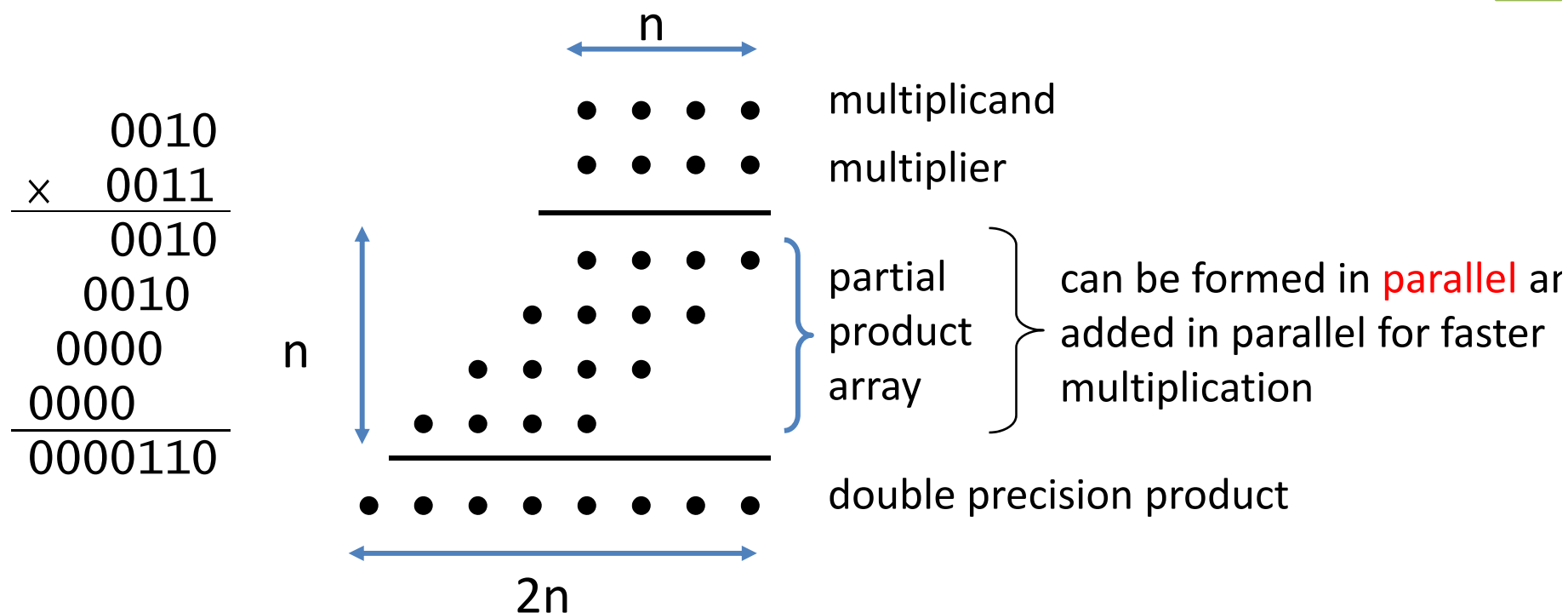
$$c3 = g2 + (p2 \cdot g1) + (p2 \cdot p1 \cdot g0) + (p2 \cdot p1 \cdot p0 \cdot c0)$$

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



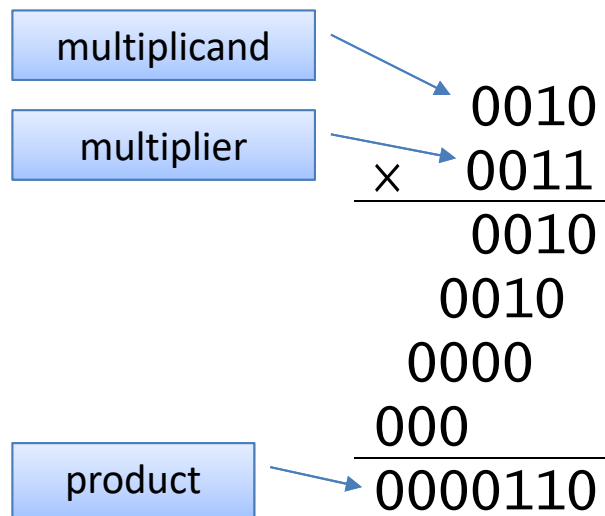
Multiplication

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

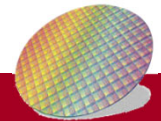
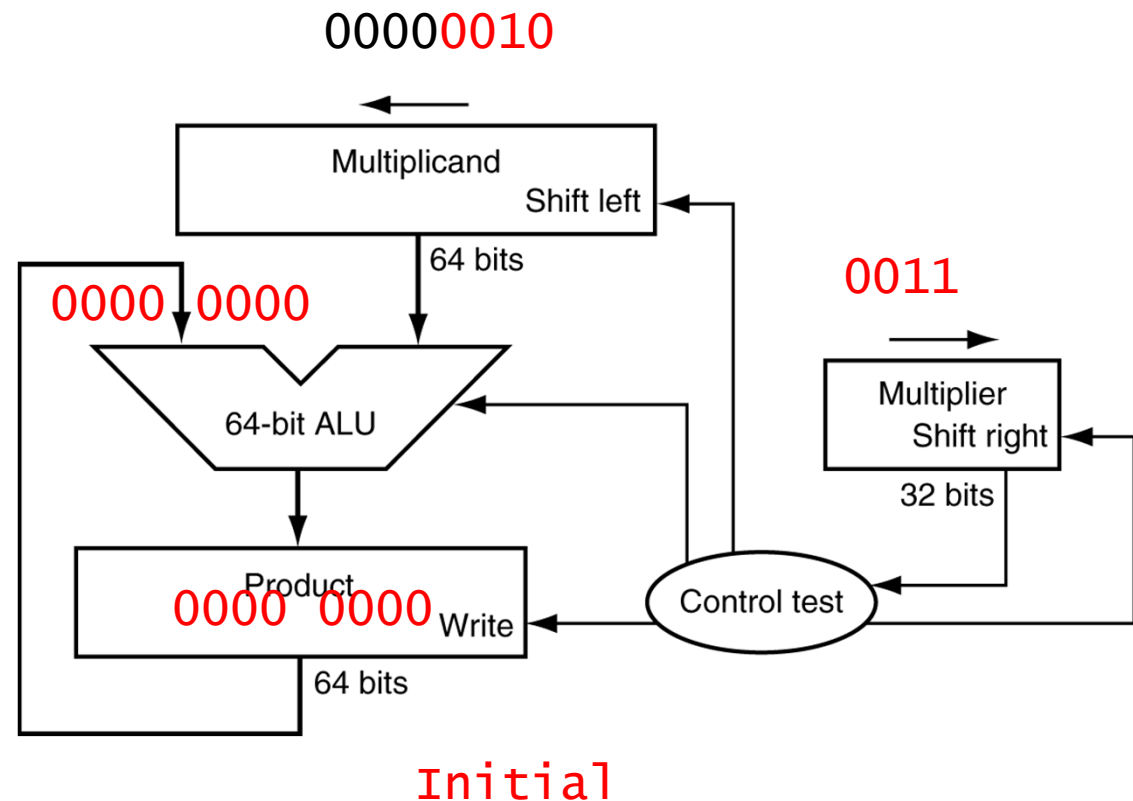


Multiplication

- Start with long-multiplication approach

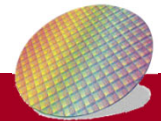
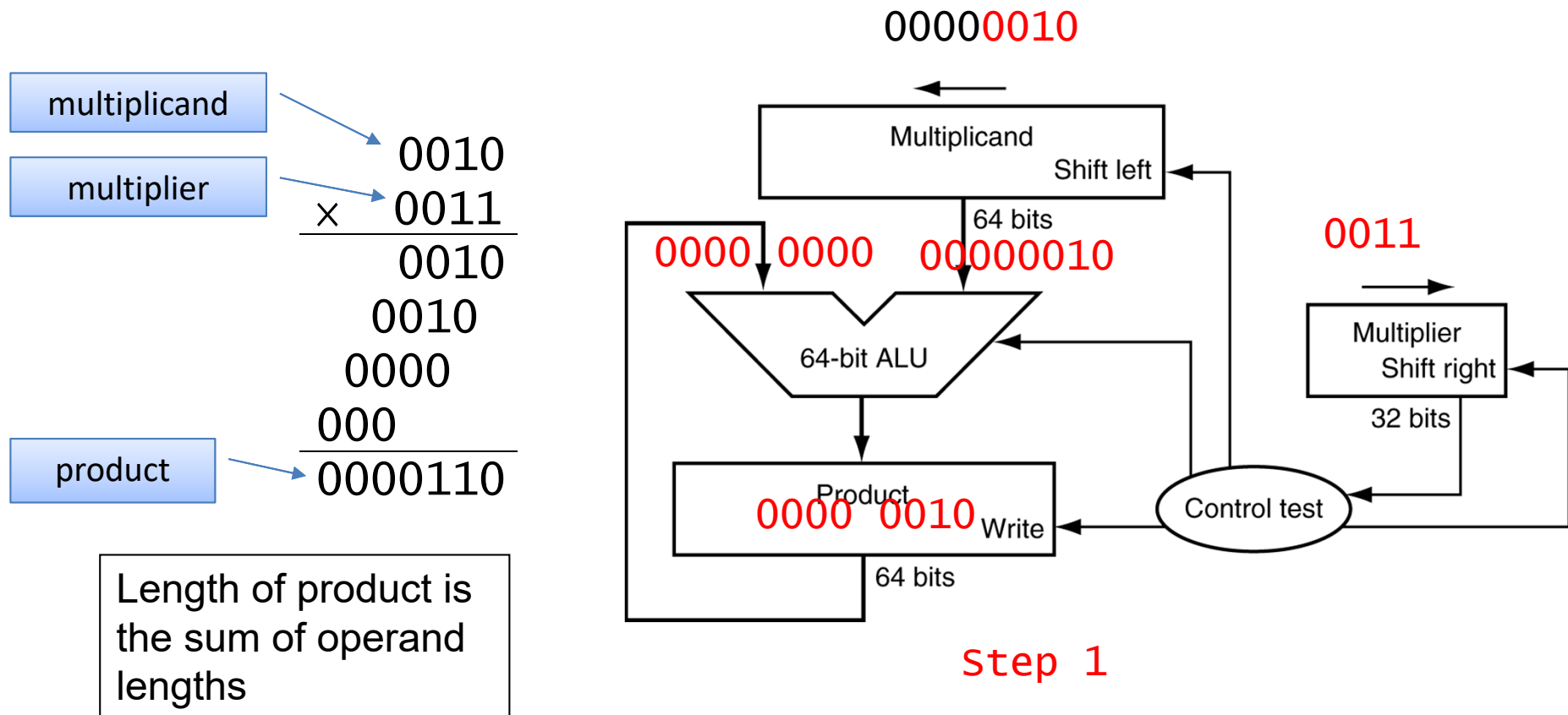


Length of product is the sum of operand lengths



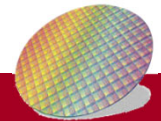
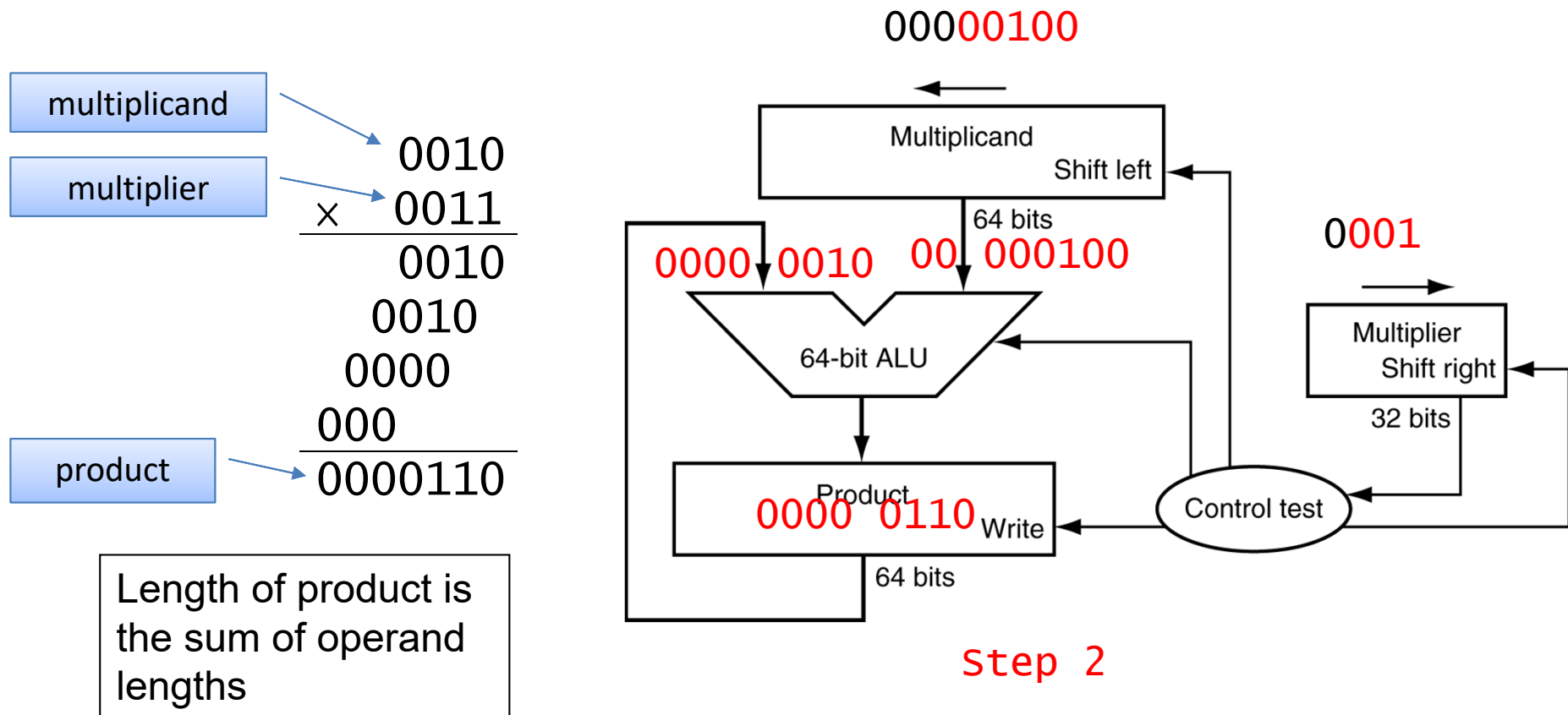
Multiplication

- Start with long-multiplication approach



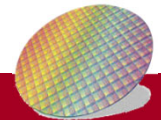
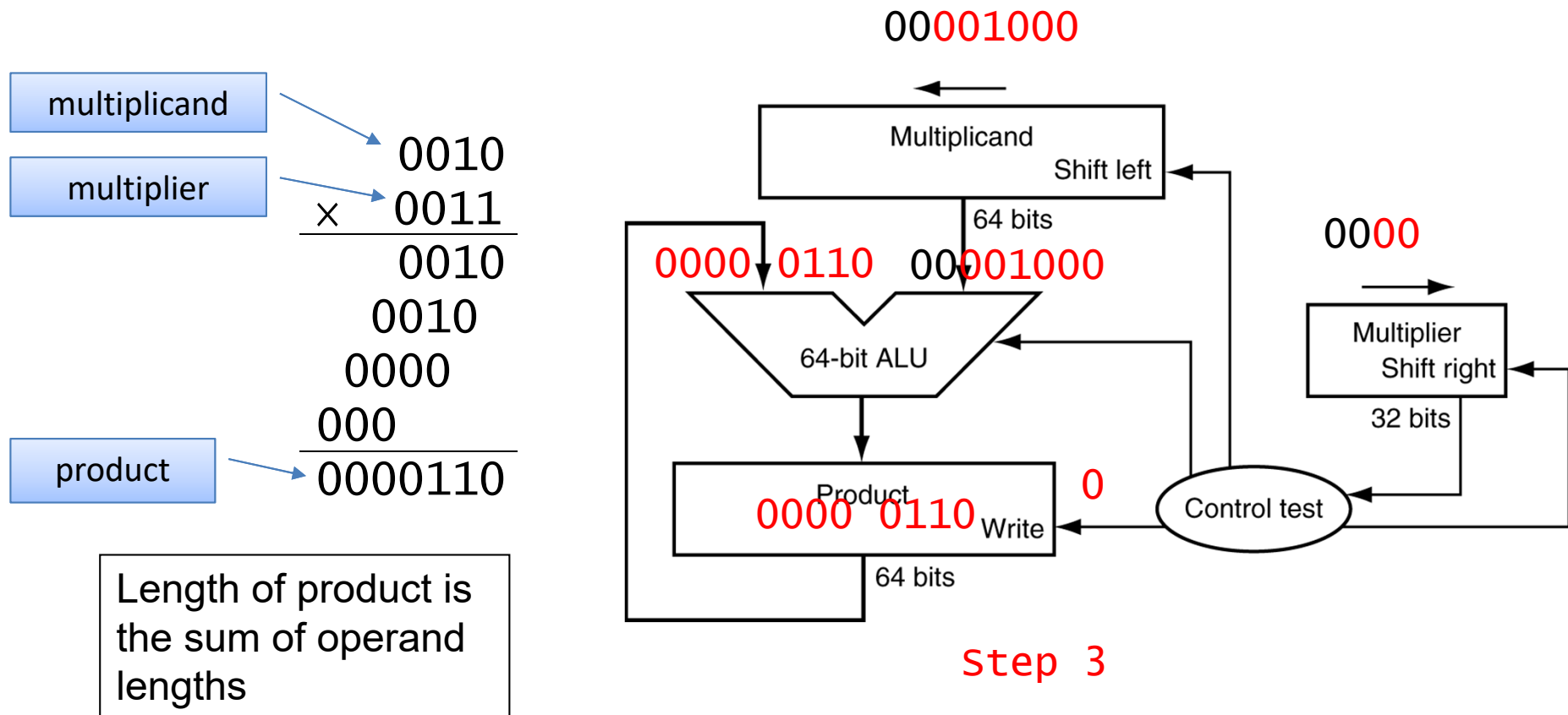
Multiplication

- Start with long-multiplication approach



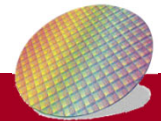
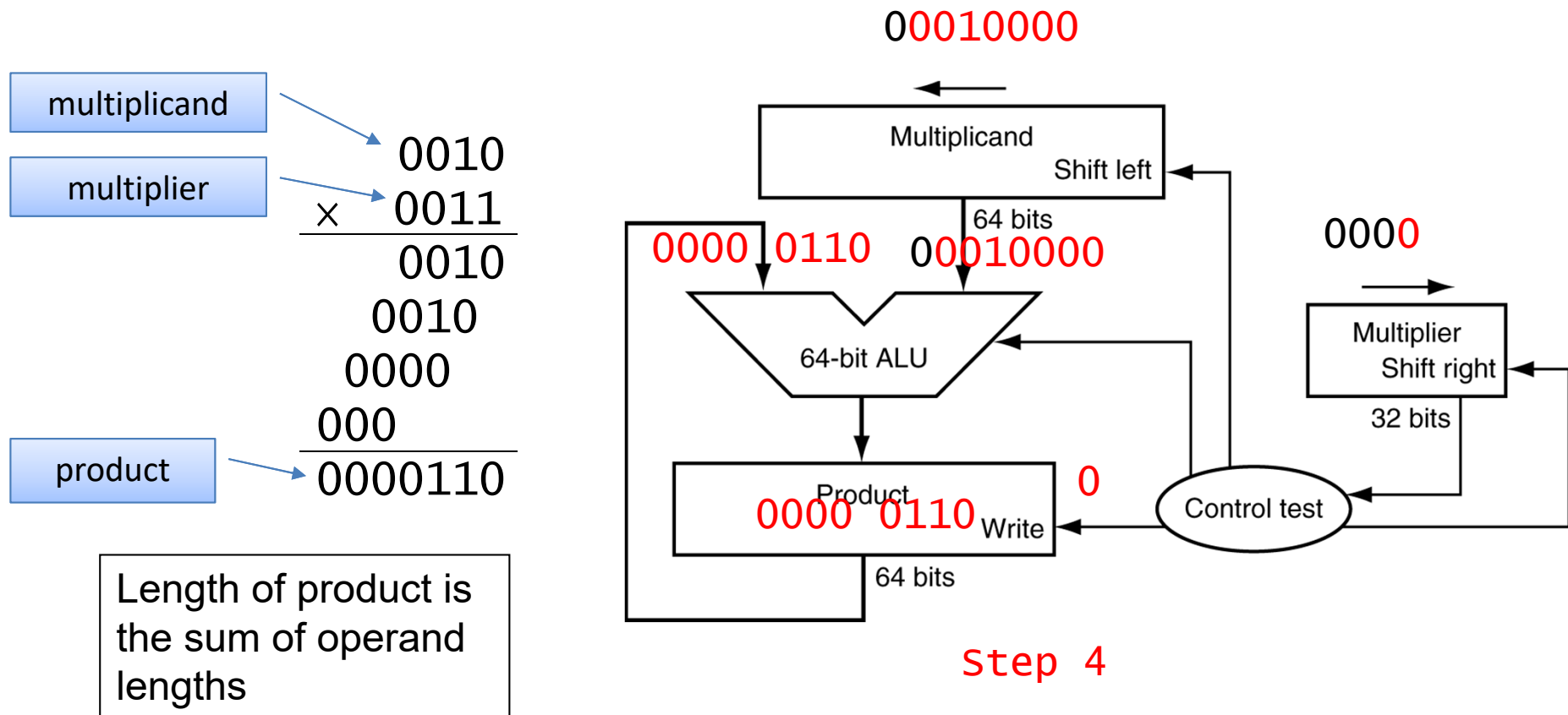
Multiplication

- Start with long-multiplication approach

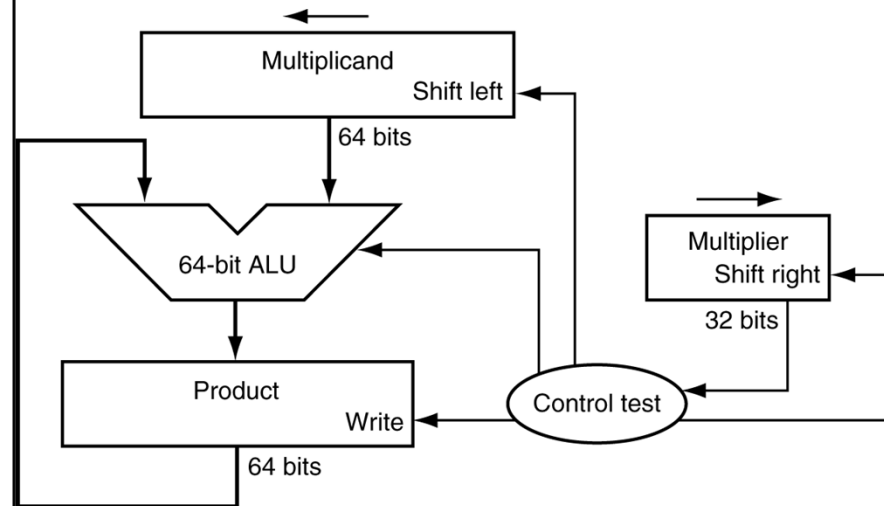
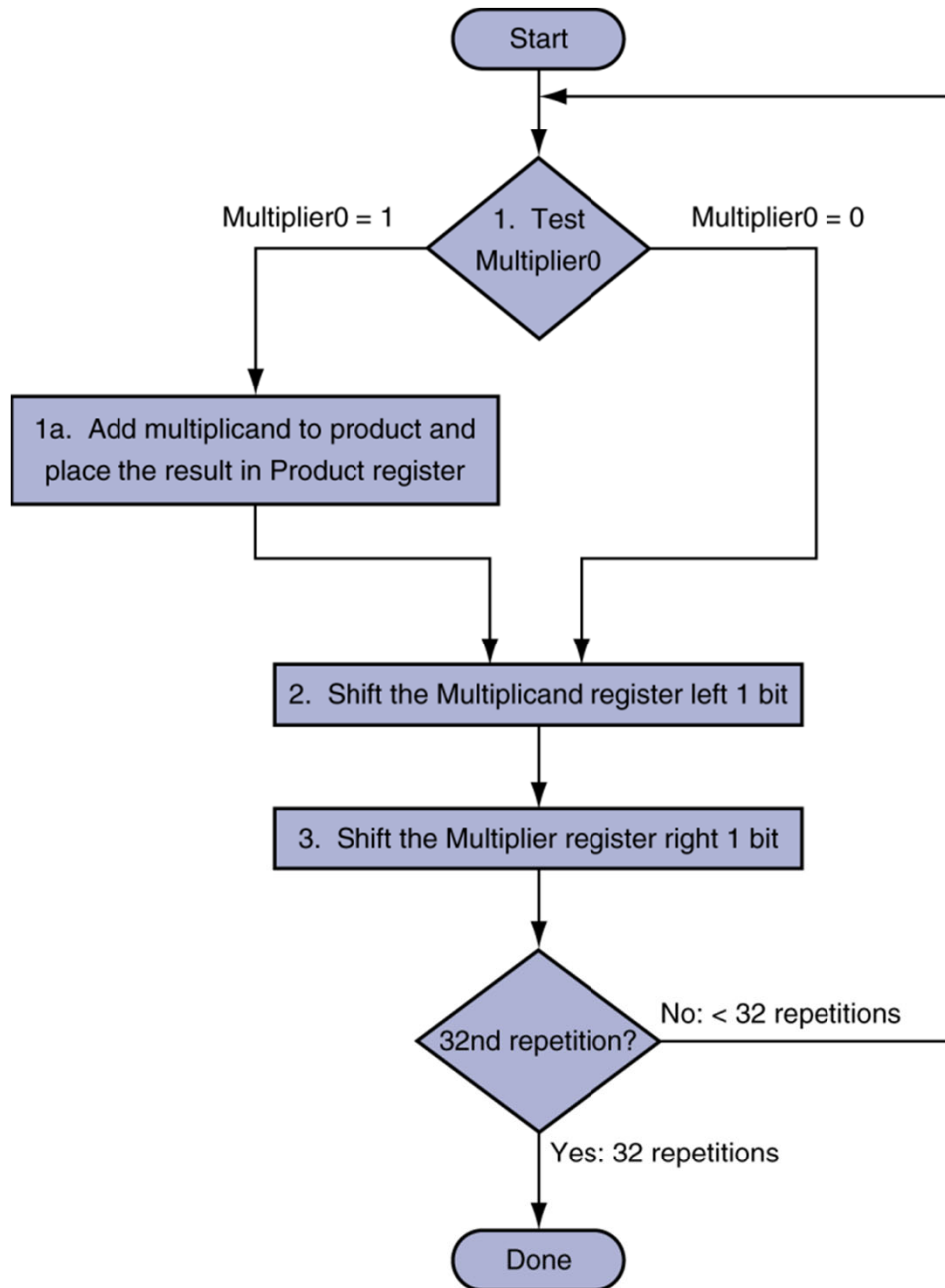


Multiplication

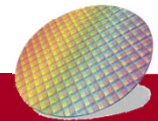
- Start with long-multiplication approach



Multiplication Hardware



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



Optimized Multiplier

- Observations: Two ways of multiplication
 - (1) Shift **multiplicand left** or (2) shift **product right**

Method 1

$\begin{array}{r} 1000 \\ \times 1011 \\ \hline 1000 \end{array}$	$\begin{array}{r} 1000 \\ \times 1011 \\ \hline 1000 \\ 1000 \\ \hline 11000 \end{array}$	$\begin{array}{r} 1000 \\ \times 1011 \\ \hline 11000 \\ 0000 \\ \hline 011000 \end{array}$	$\begin{array}{r} 1000 \\ \times 1011 \\ \hline 011000 \\ 1000 \\ \hline 1011000 \end{array}$
---	---	---	---

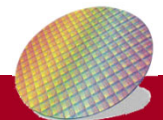
Shift multiplicand
left 1 bit and add

Shift multiplicand
left 2 bit but **no** add

Method 2

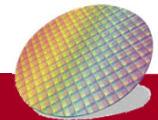
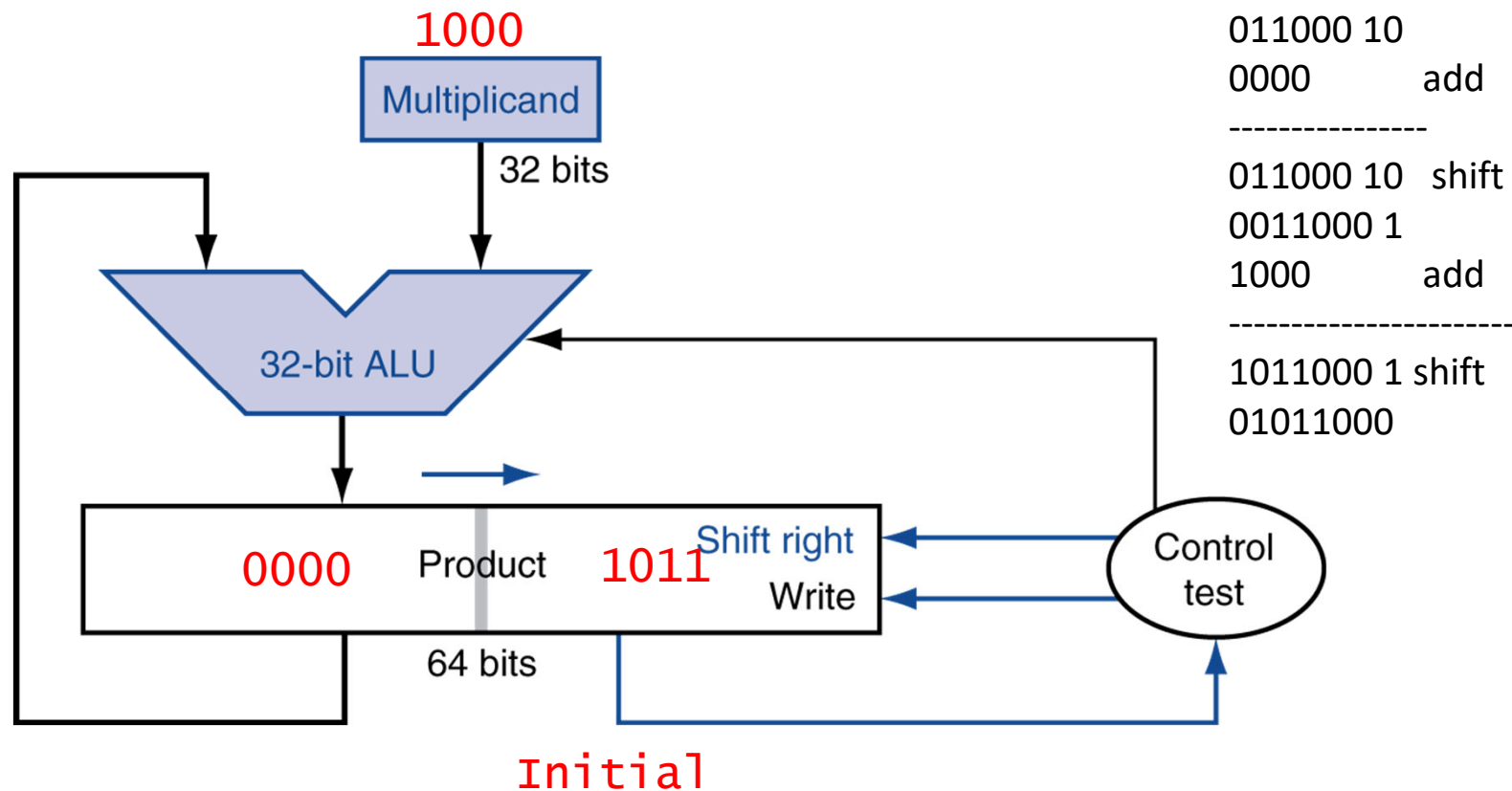
$\begin{array}{r} 1000 \\ \times 1011 \\ \hline 1000 \end{array}$	$\begin{array}{r} 1000 \\ \times 1011 \\ \hline 1000 \\ 1000 \\ \hline 11000 \end{array}$	$\begin{array}{r} 1000 \\ \times 1011 \\ \hline 11000 \\ 0000 \\ \hline 011000 \end{array}$	$\begin{array}{r} 1000 \\ \times 1011 \\ \hline 011000 \\ 1000 \\ \hline 1011000 \end{array}$
---	---	---	---

Shift product right and add



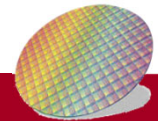
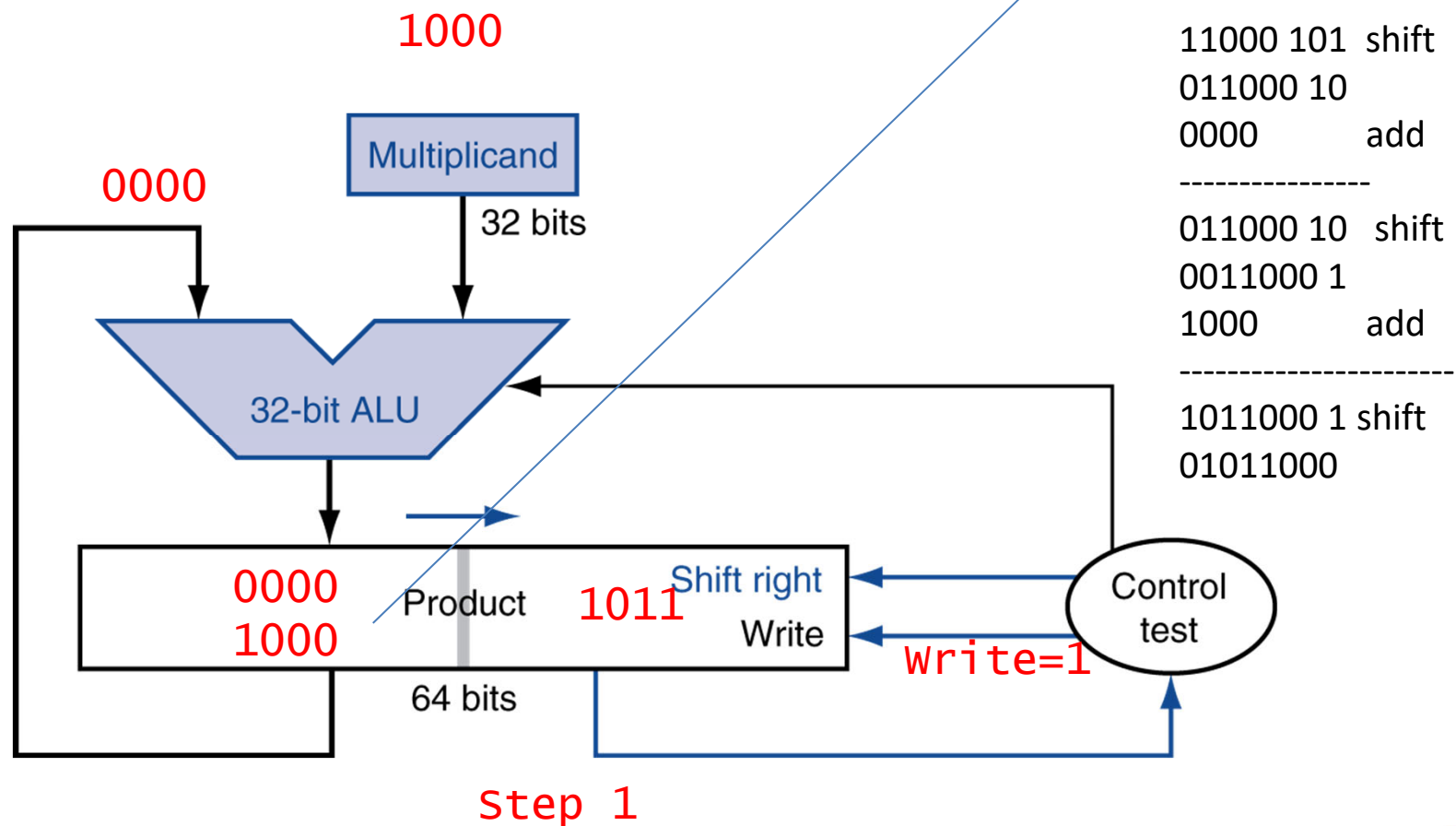
Optimized Multiplier

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



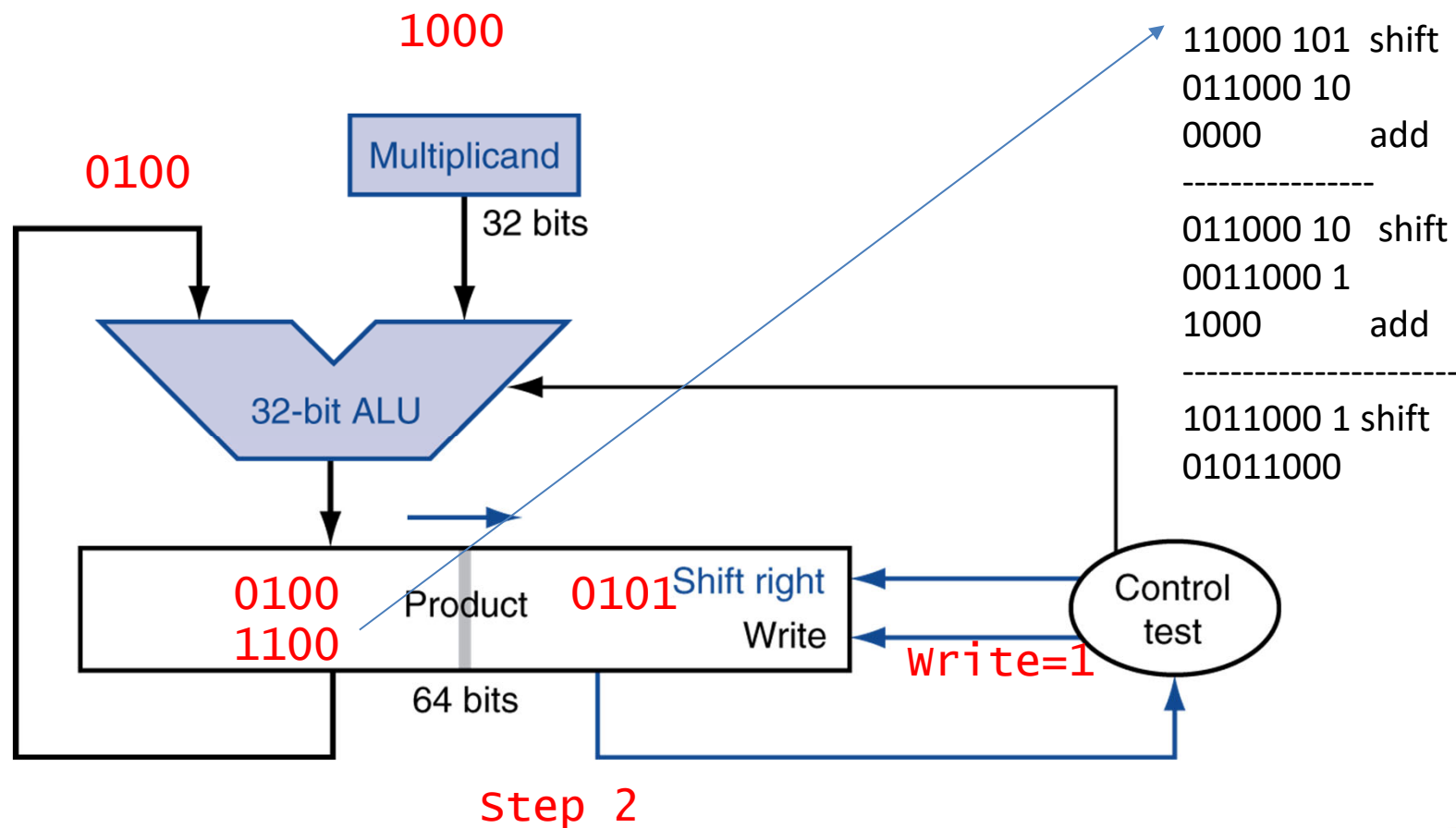
Optimized Multiplier

- Step 1



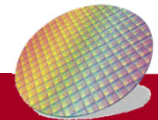
Optimized Multiplier

- Step 2



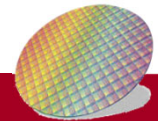
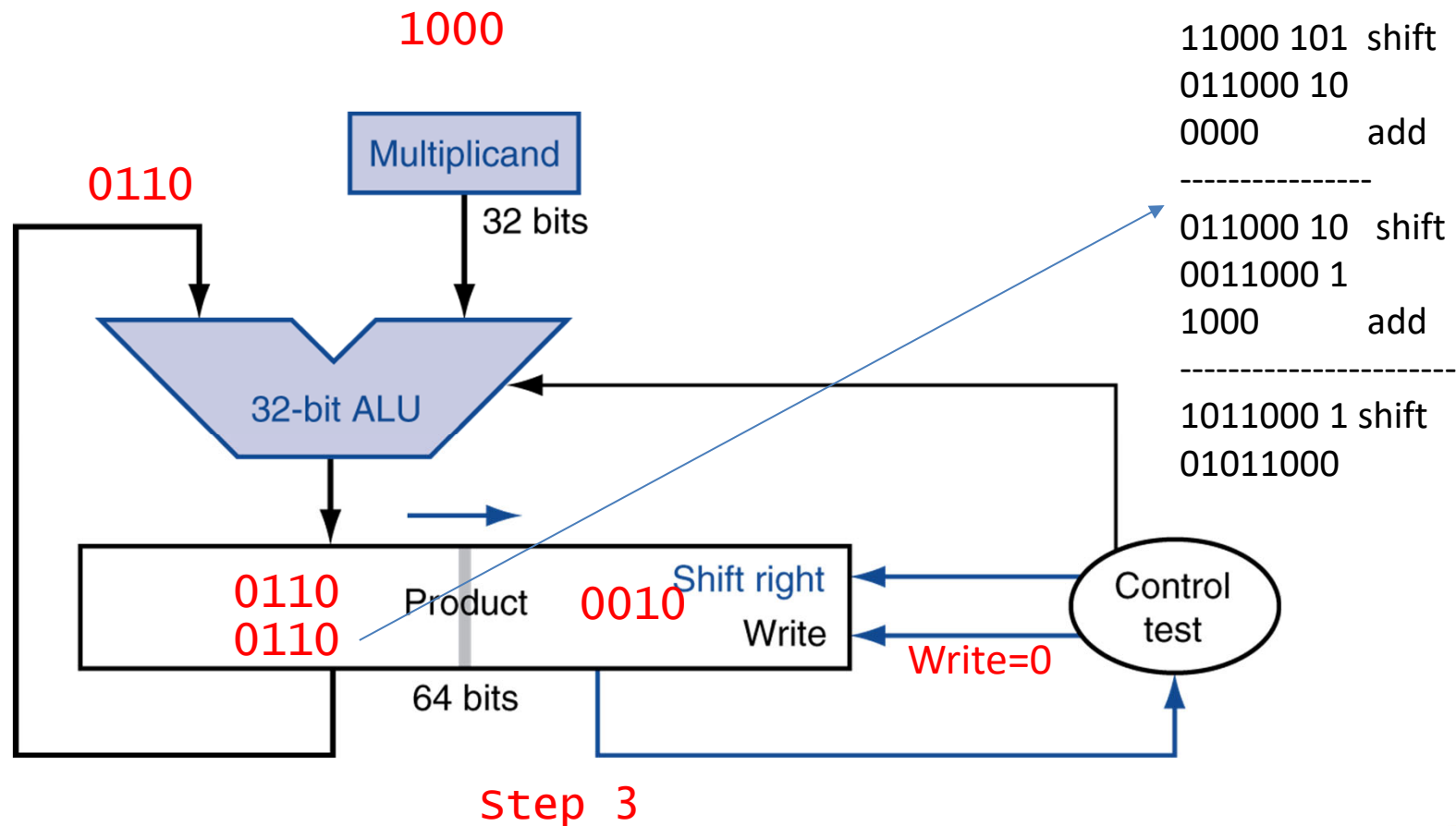
```

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

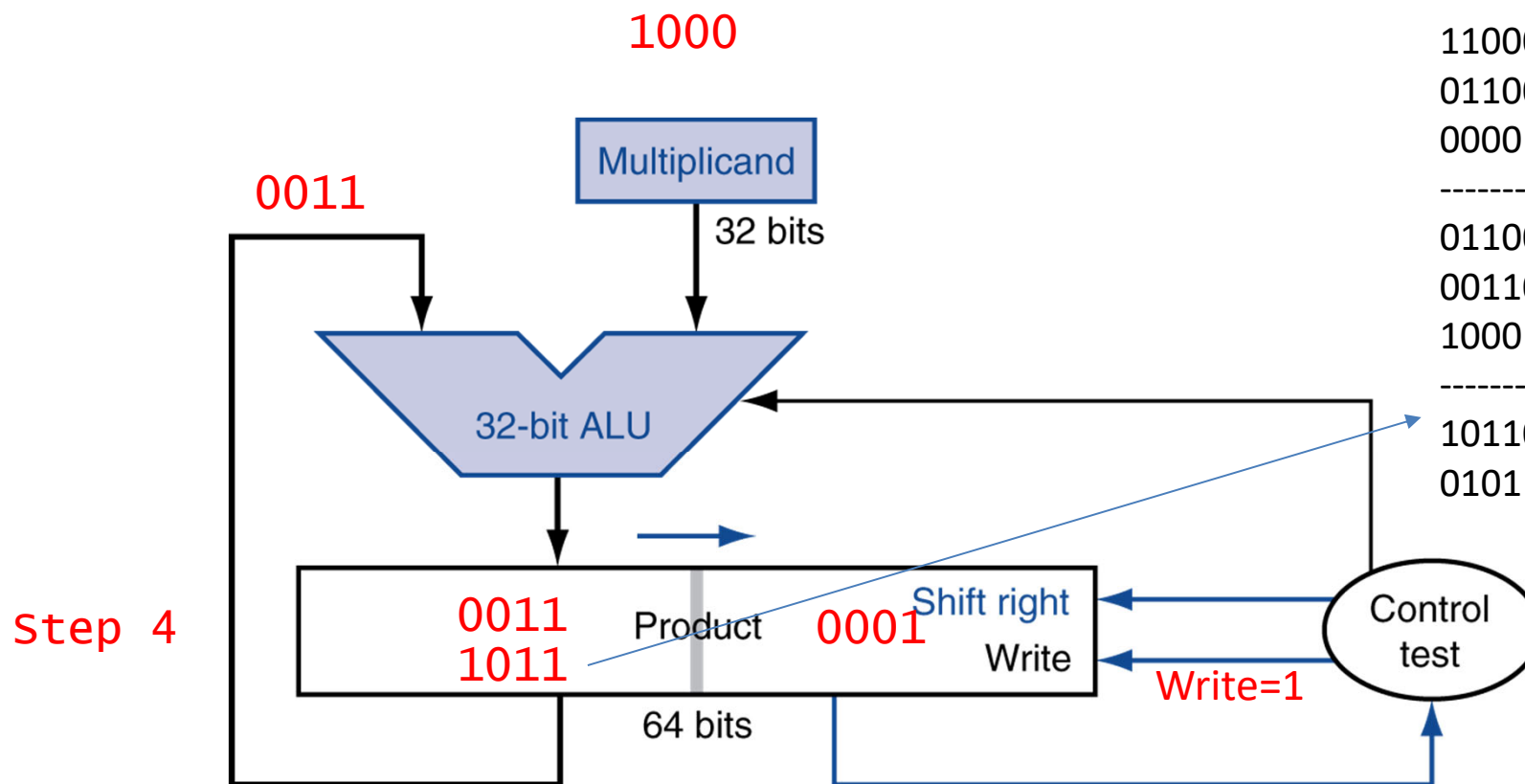


Optimized Multiplier

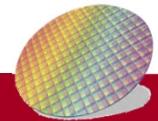
- Step 4:

```

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
    
```

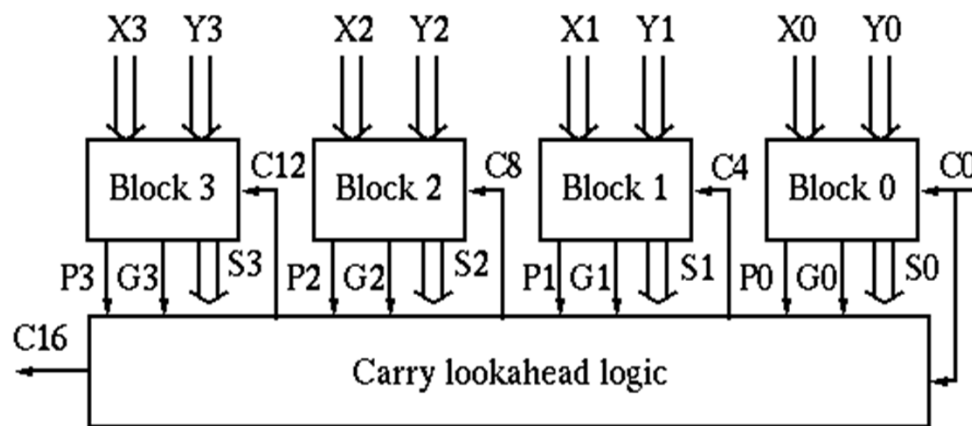


Final product: 01011000

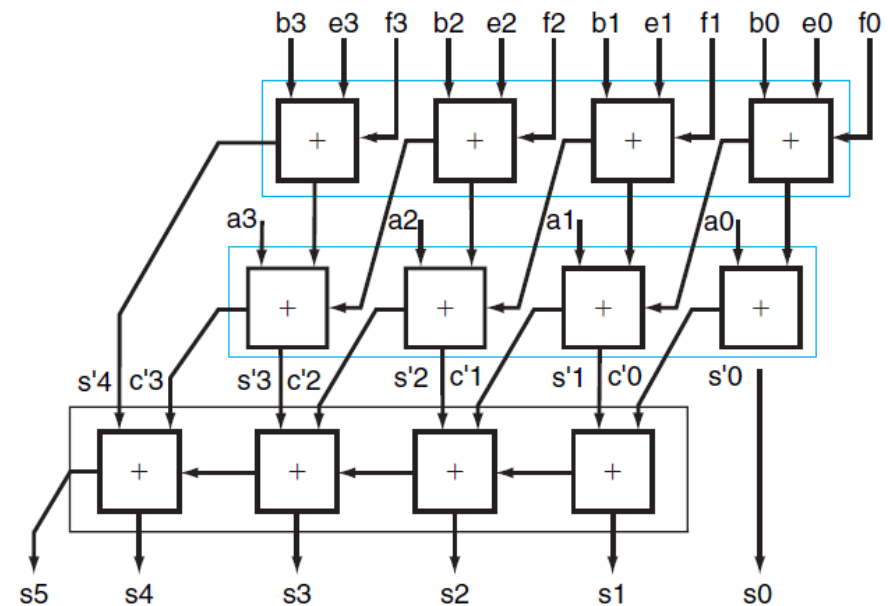


Faster Multiplier

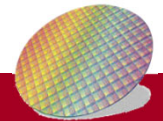
- Uses faster adder
 - Addition is repetitively performed
 - Faster adder can improve multiplication speed
 - E.g. carry lookahead adder, carry save adder, etc.
 - See appendix C



Carry lookahead adder



Carry save adder

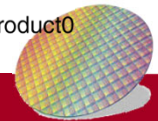
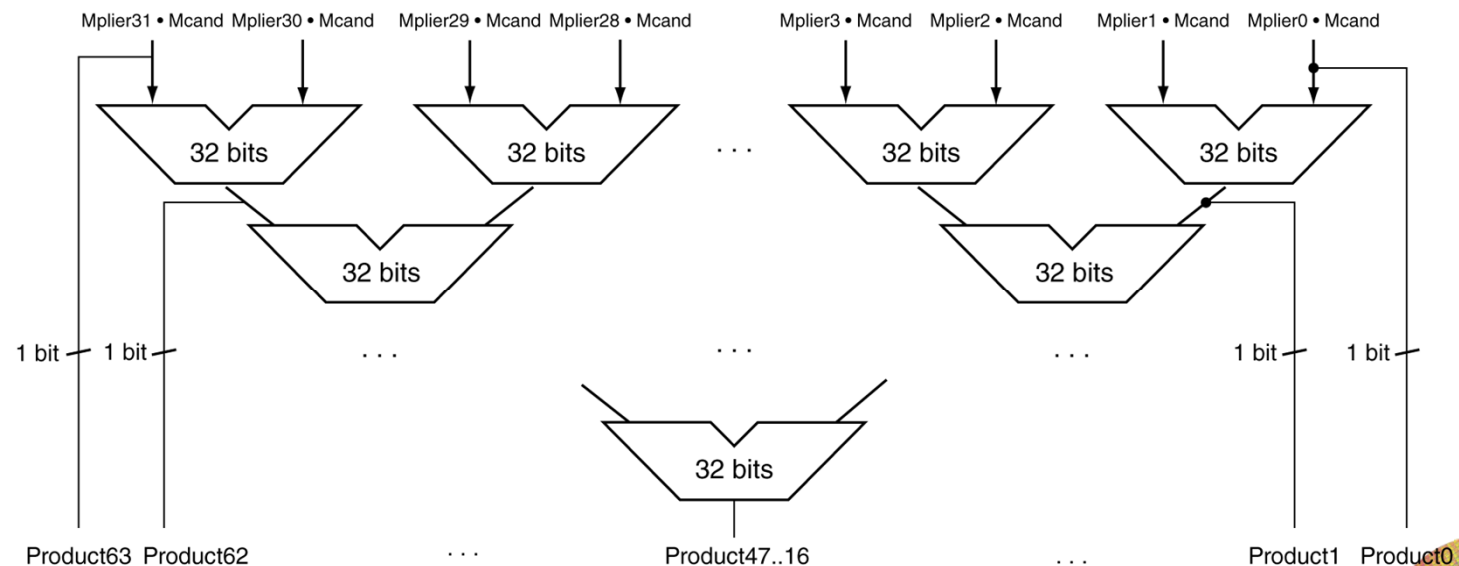


Faster Multiplier

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

$$\begin{array}{r}
 0010 \\
 \times 0011 \\
 \hline
 0010 \\
 0010 \\
 0000 \\
 0000 \\
 \hline
 0000110
 \end{array}$$

Perform addition in parallel



MIPS Multiplication-mul

Multiplication instruction: mul

mul \$rd \$rs \$rt

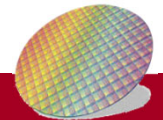
$\$rd = \$rs * \$rt$

- Result is in **\$rd**
- Only **low-order 32 bits** of the product are preserved in \$rd

.text

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

Without Overflow

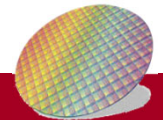


MIPS Multiplication-mult

- 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 register

- Use `mfhi rd` / `mflo rd` to access results
 - `mfhi rd`: Move from HI to rd
 - `mflo rd`: Move from LO to rd
 - Can test HI value to see if product overflows to 32 bits



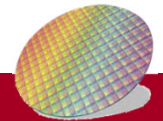
Example

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

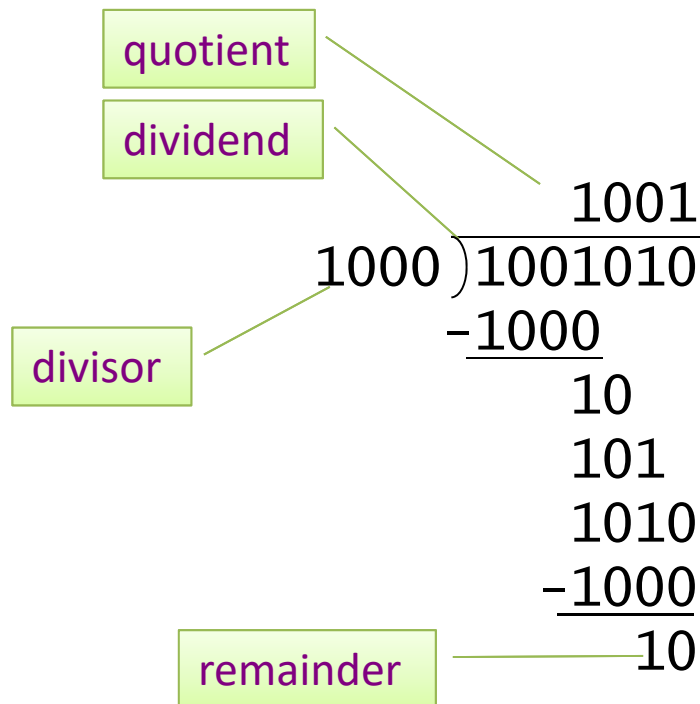
Program to calculate $5 * 12 - 74$

.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
```

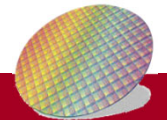


Division

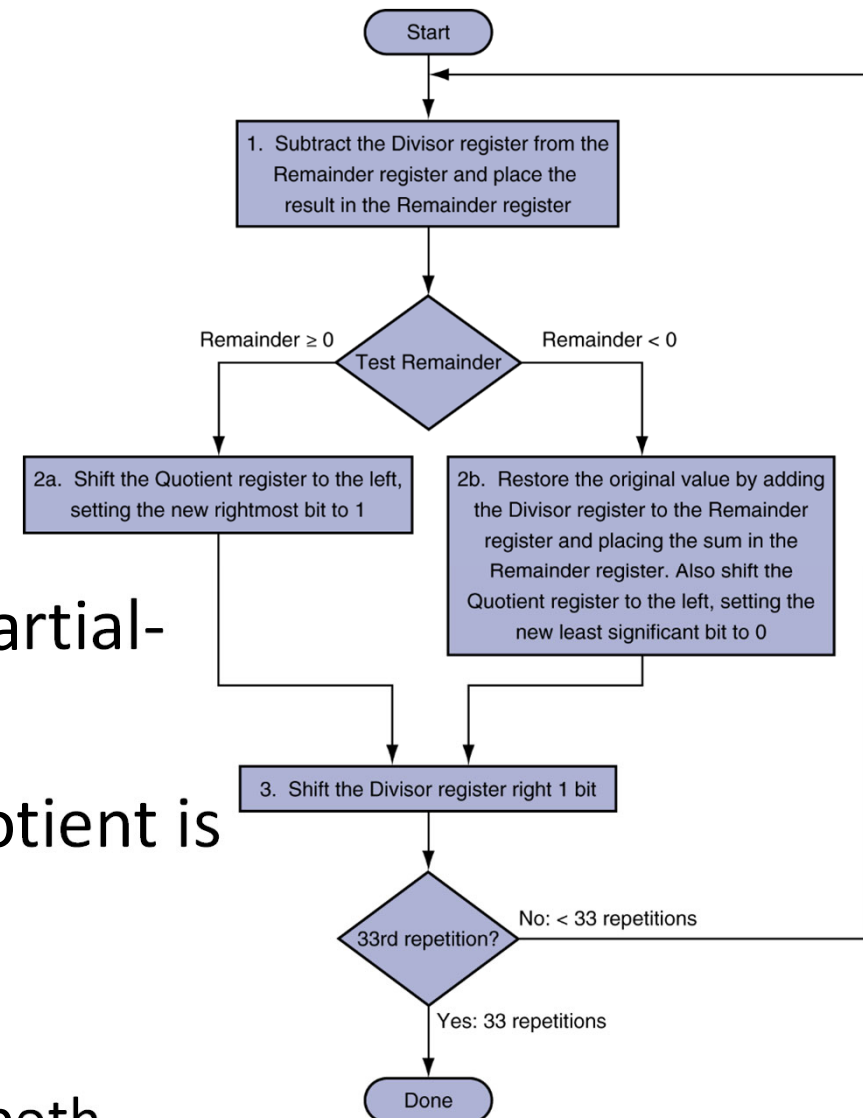
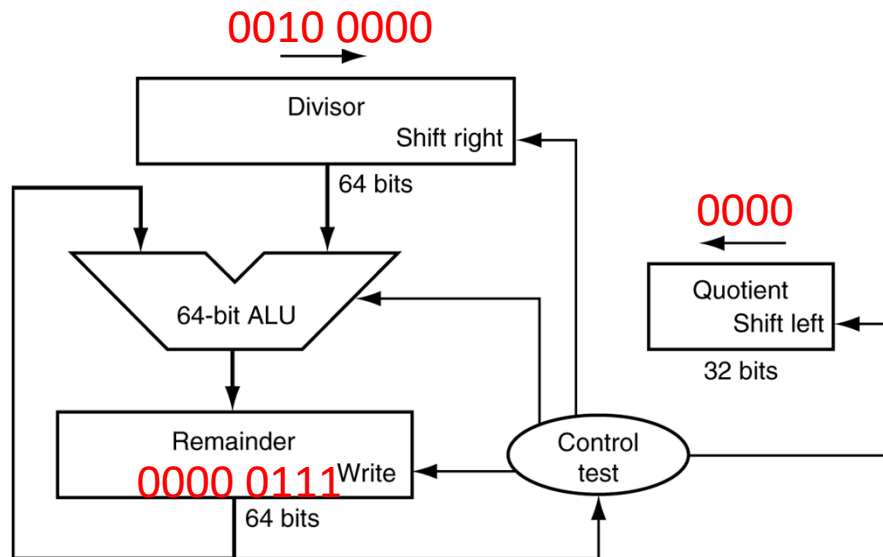


n -bit operands yield n -bit quotient and remainder

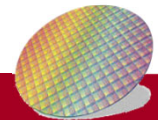
- Check if divisor = 0
- Long division approach
 - If divisor \leq dividend bits
 - 1 bit in quotient, subtract
 - Otherwise
 - 0 bit in quotient, bring down next dividend bit
- Division is just a bunch of **quotient digit guesses** and **left shifts** and **subtracts**
- **Restoring** division
 - Do the subtract, and if remainder goes < 0 , add divisor back



First version of Division hardware ($7 \div 2$)



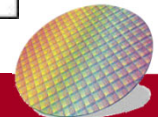
- One cycle is needed for each partial-remainder subtraction
- Divisor is shifted **right**, and Quotient is shifted **left**
- Looks a lot like a multiplier!
 - Same hardware can be used for both



Restoring Division Example

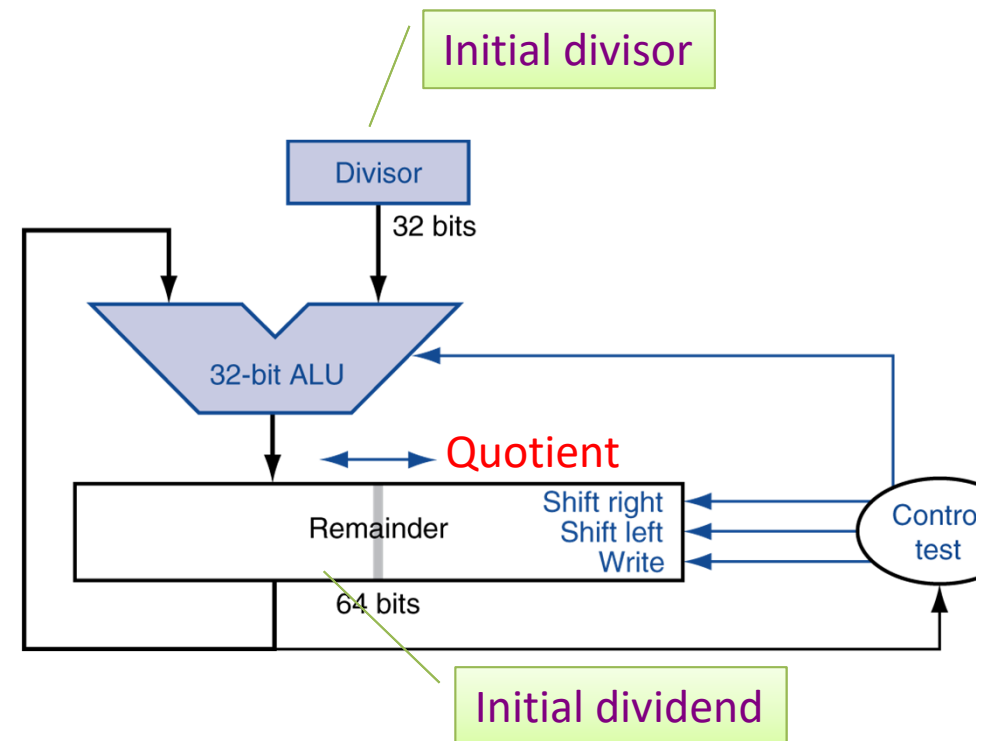
- $7 \div 2 = 0000\ 0111_2 / 0010_2$
- (Initial: Quotient=0000, Divisor=0010, Remainder=00000111)

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 \Rightarrow +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 \Rightarrow +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 \Rightarrow +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	①000 0011
	2a: Rem \geq 0 \Rightarrow 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	①000 0001
	2a: Rem \geq 0 \Rightarrow sll Q, Q0 = 1	0011	0000 0010	0000 0001
	3: Shift Div right	0011	0000 0001	0000 0001

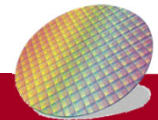


Optimized division hardware

- Division is similar to multiplication, so is hardware
- Improved version:
 - Shifting the operands and the quotient simultaneously with the subtraction
 - **Halves** the width of the **adder** and **registers** by noticing where there are **unused** portions of registers and adders

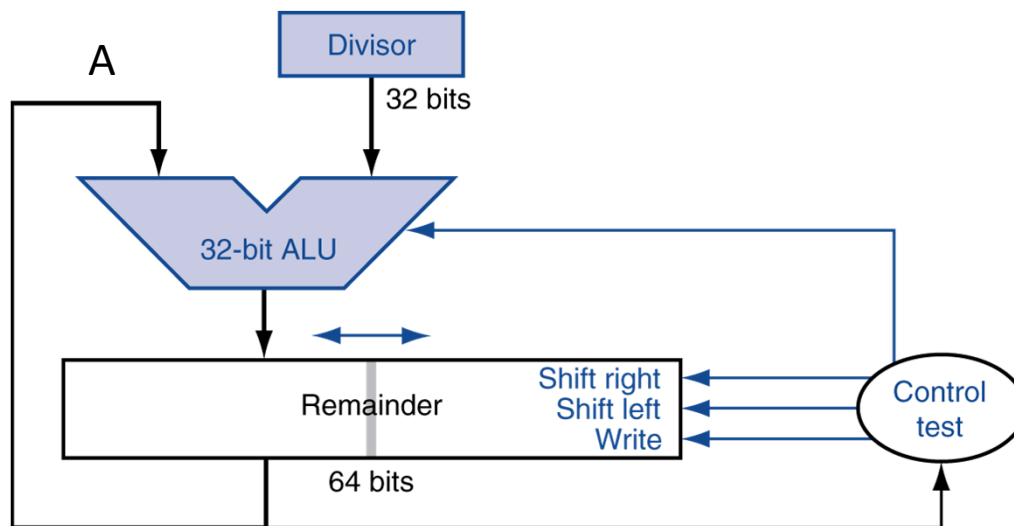


Improved hardware
 32-bit Divisor
 No extra bit for Quotient



Question: why do we need both “shift left” , and “shift right” for remainder register

- The algorithm of the improved version of division are removed



Init: Place the dividend in the remainder register and shift the remainder left 1 bit

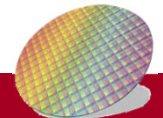
Step 1: subtract the divisor from **left part of Remainder** and place the result to the left half of the remainder

Step 2: If remainder < 0, restore, shift the remainder left 1 bit and goto step 4

Step 3: Shift the remainder left 1 bit and **set** the new right most bit to 1

Step 4: Repeat step 1 to step 3 32 times

Step 5: shift the left half of remainder **right** 1 bit



Question: why do we need both “shift left” , and “shift right” for remainder register

- The algorithm of the improved version of division are removed.

- Remainder is shifted left 1 bit and subtract divisor the divisor from the left half of remainder

Init: Place the dividend in the remainder register and shift the remainder left **1** bit

Step 1: subtract the divisor from **left part of Remainder** and place the result to the left half of the remainder

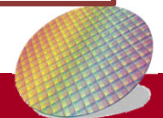
Step 2: If remainder < 0 , restore, shift the remainder left 1 bit and goto step 4

Step 3: Shift the remainder left 1 bit and **set** the new right most bit to 1

Step 4: Repeat step 1 to step 3 32 times

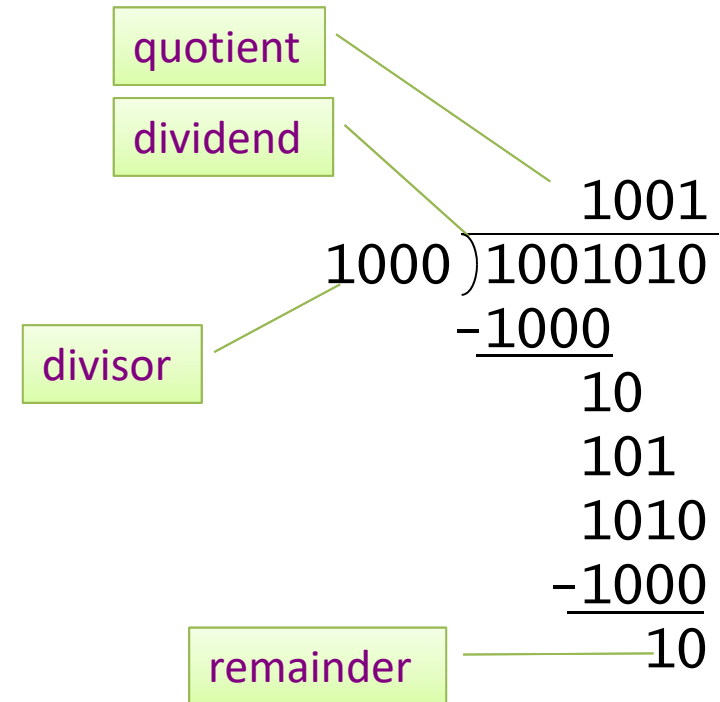
- Shift **right** the “left half of remainder” to get correct remainder

Step 5: shift the left half of remainder **right** 1 bit

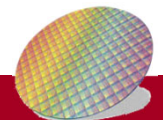


Why Division is slower?

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

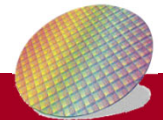


n -bit operands yield n -bit quotient and remainder



MIPS Division

- Use HI/LO registers for result
 - HI: 32-bit remainder
 - LO: 32-bit quotient
- Instructions (only two operands)
 - `div rs, rt`
 - # $LO = \$s0 / \$s1$, quotient in LO
 - # $HI = \$s0 \bmod \$s1$, remainder in HI
 - Use **mfhi rd** and **mflo rd** are provided to move the quotient and remainder to user accessible registers



Division Example

- Calculate $13/5$, put the quotient in **\$t1**, and remainder in **\$t0**

.text

.globl main

main:

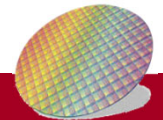
ori \$t5, \$zero, 13 # put 13 into \$5

ori \$t6, \$zero, 5 # put 5 into \$6

div \$t5, \$t6 # **Lo** = \$t5 / \$t6 (integer quotient)
 # **Hi** = \$t5 mod \$t6 (remainder)

mfhi \$t0 # move remainder 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





成功大學

National Cheng Kung University

Backup slides

