Arithmetic and Logic Unit
Second Part

Arquitectura de Computadoras

Arturo Díaz Pérez
Centro de Investigación y de Estudios Avanzados del IPN
Laboratorio de Tecnologías de Información
adiaz@cinvestav.mx

Multiplication

- Multiplication can't be that hard!
  - It's just repeated addition.
  - If we have adders, we can do multiplication also.
- Remember that the AND operation is equivalent to multiplication on two bits:

a	b	ab
0	0	0
0	1	0
1	0	0
1	1	1

a	b	a×b
0	0	0
0	1	0
1	0	0
1	1	1

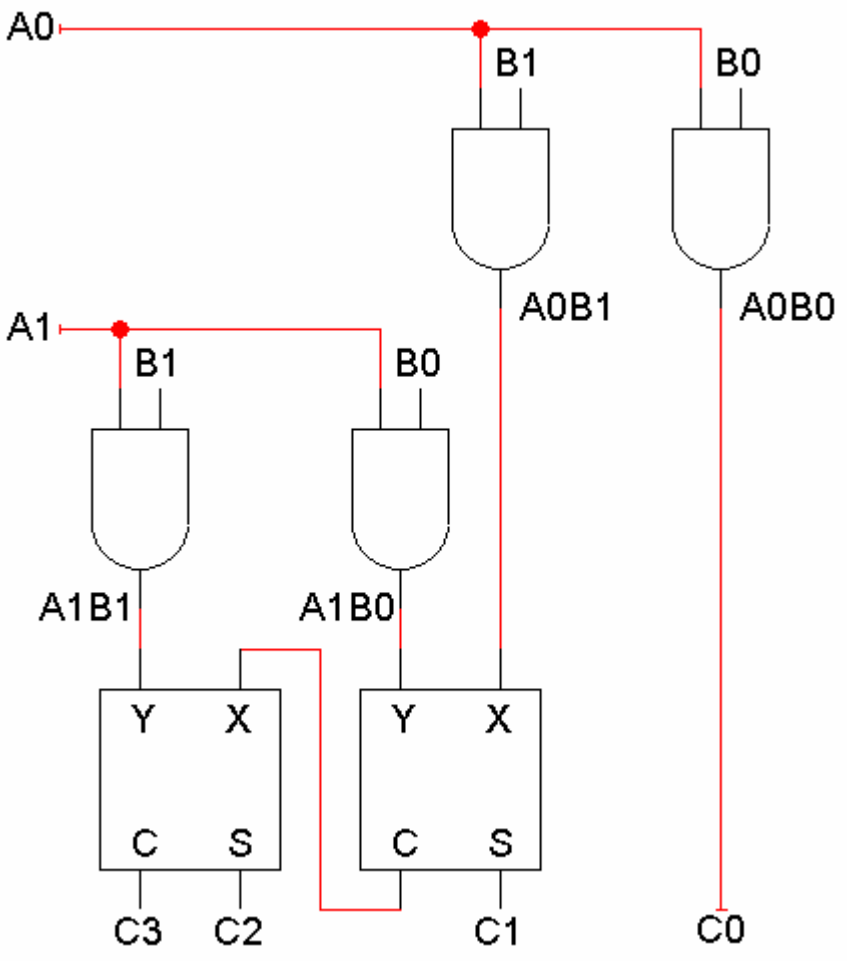
Binary multiplication example

				1	1	0	1	Multiplicand
			x	0	1	1	0	Multiplier
				0	0	0	0	Partial products
			1	1	0	1		
		1	1	0	1			
+	0	0	0	0				
	1	0	0	1	1	1	0	Product

- Since we always multiply by either 0 or 1, the partial products are always either 0000 or the multiplicand (1101 in this example).
- There are four partial products which are added to form the result.
  - We can add them in pairs, using three adders.
  - Even though the product has up to 8 bits, we can use 4-bit adders if we “stagger” them leftwards, like the partial products themselves.

A 2x2 binary multiplier

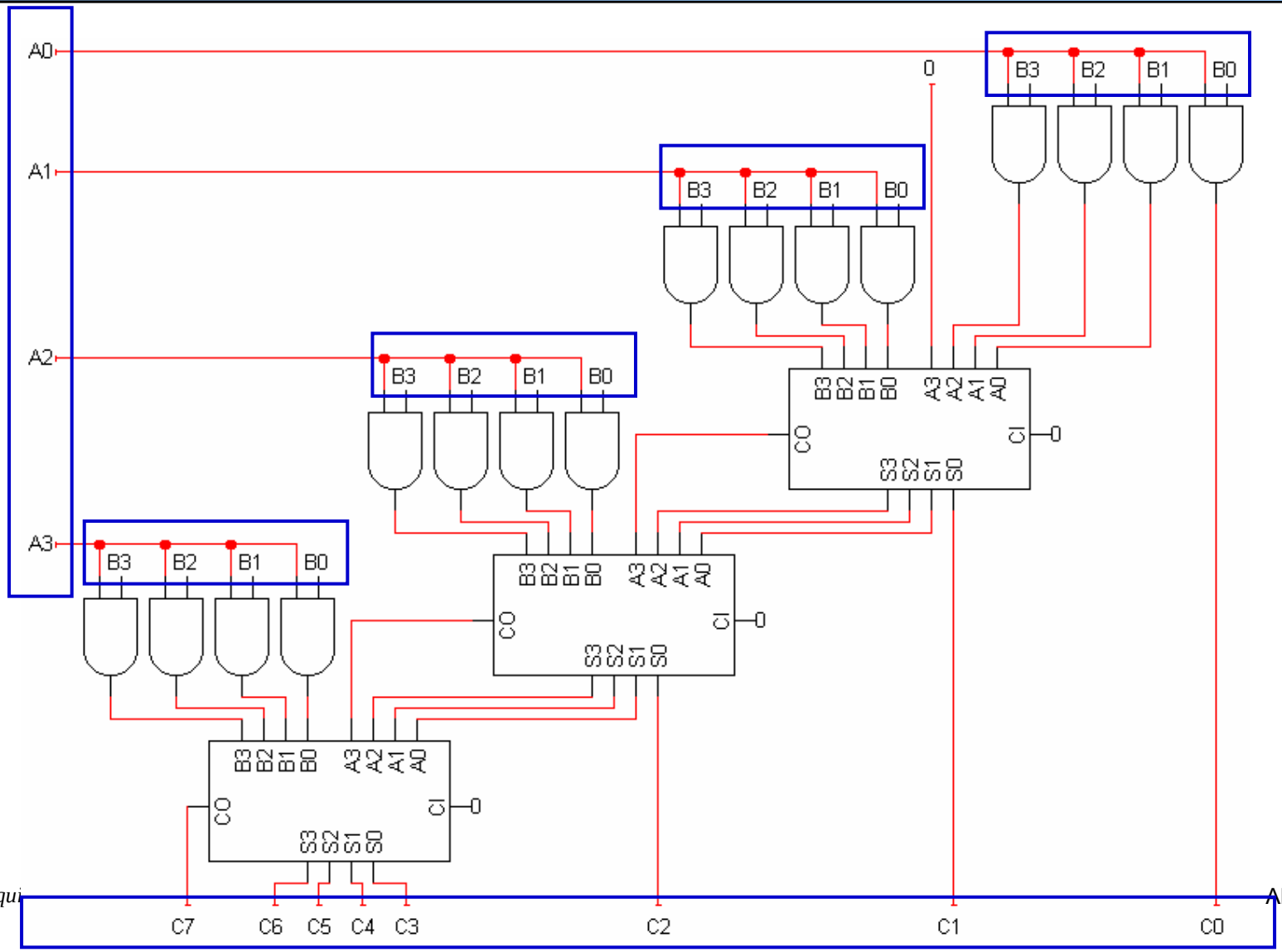
- The AND gates produce the partial products.
- For a 2-bit by 2-bit multiplier, we can just use two half adders to sum the partial products. In general, though, we'll need full adders.
- Here C_3-C_0 are the product, not carries!



$$
\begin{array}{rcccc}
 & & & B_1 & B_0 \\
 & & \times & A_1 & A_0 \\
\hline
 & & & A_0B_1 & A_0B_0 \\
+ & & A_1B_1 & A_1B_0 & \\
\hline
 & C_3 & C_2 & C_1 & C_0
\end{array}
$$

A 4x4 multiplier circuit

Laboratorio de
Tecnologías de Información



More on multipliers

- Notice that this 4-bit multiplier produces an 8-bit result
  - We could just keep all 8 bits
  - Or, if we needed a 4-bit result, we could ignore C4-C7, and consider it an overflow condition if the result is longer than 4 bits
- Multipliers are very complex circuits.
  - In general, when multiplying an m-bit number by an n-bit number:
    - » There are n partial products, one for each bit of the multiplier
    - » This requires n-1 adders, each of which can add m bits (the size of the multiplicand)
  - The circuit for 32-bit or 64-bit multiplication would be huge!

Multiplication: a special case

- In decimal, an easy way to multiply by 10 is to shift all the digits to the left, and tack a 0 to the right end.

  128 x 10 = 1280

- We can do the same thing in binary. Shifting left is equivalent to multiplying by 2:

  11 x 10 = 110 (in decimal, 3 x 2 = 6)

- Shifting left twice is equivalent to multiplying by 4:

  11 x 100 = 1100 (in decimal, 3 x 4 = 12)

- As an aside, shifting to the *right* is equivalent to *dividing* by 2.

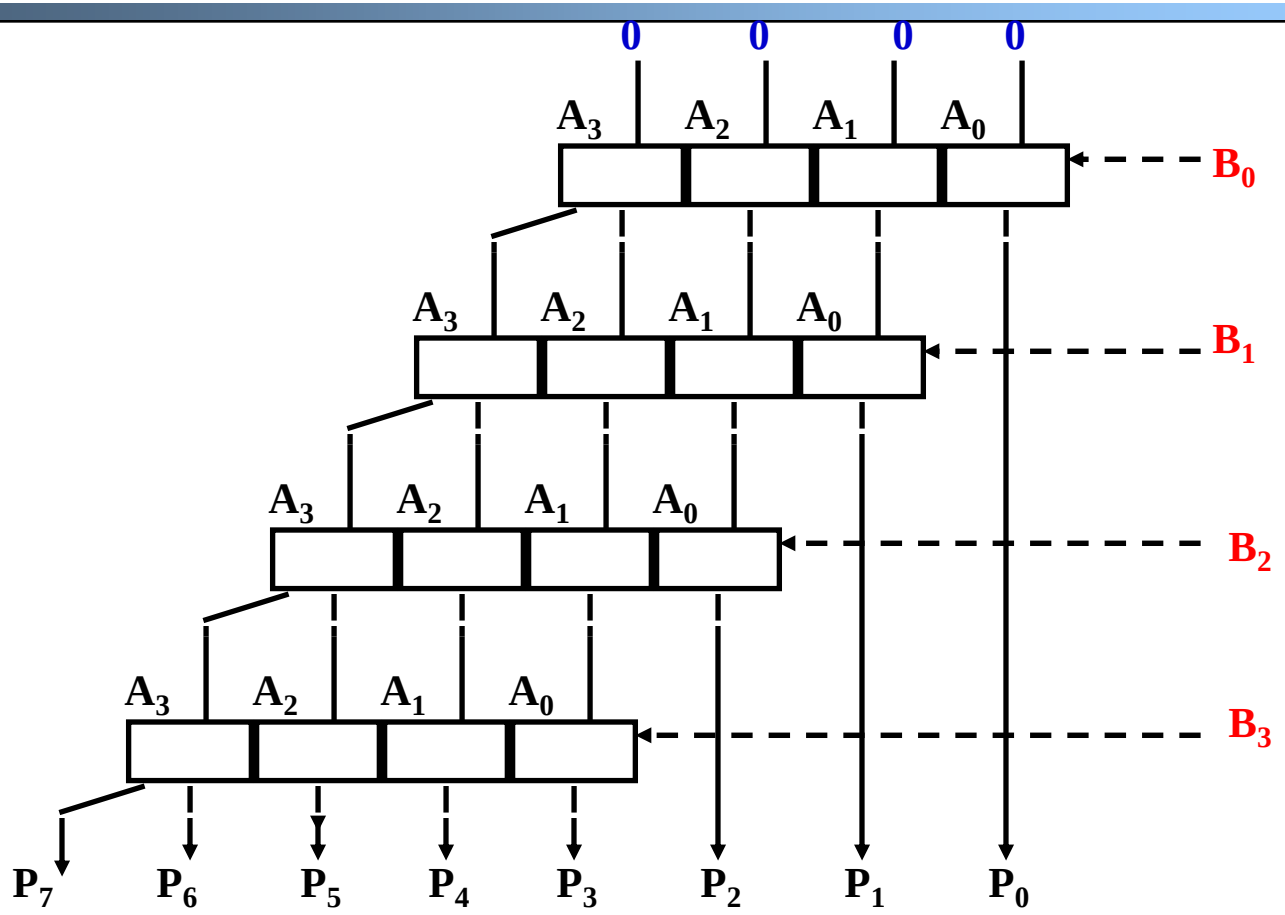
  110 ÷ 10 = 11 (in decimal, 6 ÷ 2 = 3)

Addition and multiplication summary

- Adder and multiplier circuits mimic human algorithms for addition and multiplication
- Adders and multipliers are built hierarchically
  - We start with half adders or full adders and work our way up
  - Building these functions from scratch with truth tables and K-maps would be pretty difficult
- The arithmetic circuits impose a limit on the number of bits that can be added. Exceeding this limit results in overflow
- There is a tradeoff between simple but slow circuits (ripple carry adders) and complex but fast circuits (carry lookahead adders)
- Multiplication and division by powers of 2 can be handled with simple shifting

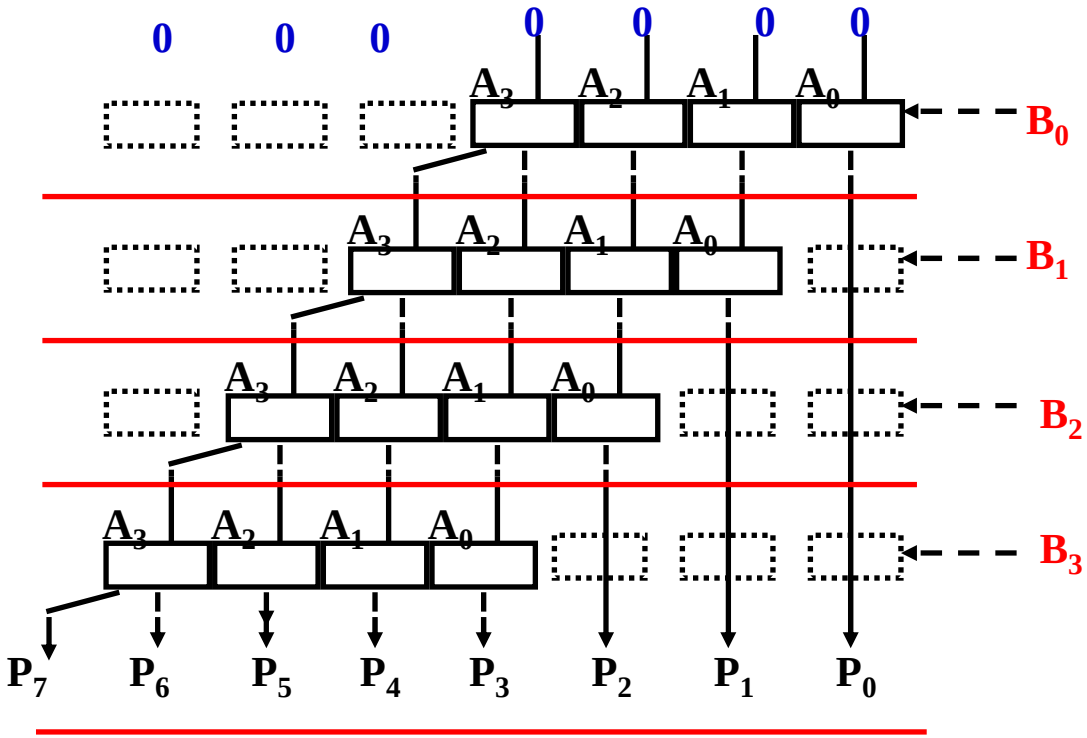
Unsigned Combinational Multiplier

Laboratorio de
Tecnologías de Información



- Stage i accumulates $A * 2^i$ if $B_i == 1$
- Q: How much hardware for 32 bit multiplier? Critical path?

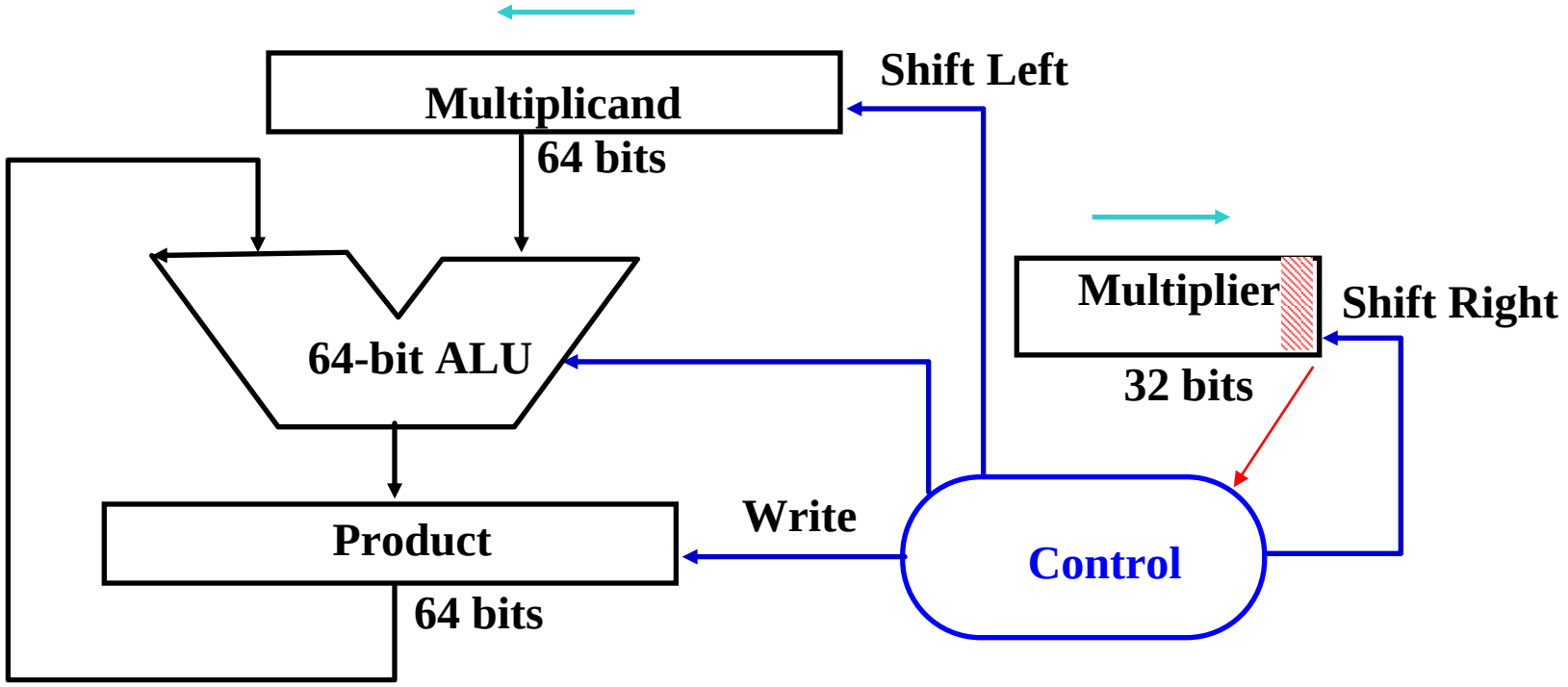
How does it work?



- at each stage shift A left (x 2)
- use next bit of B to determine whether to add in shifted multiplicand
- accumulate 2n bit partial product at each stage

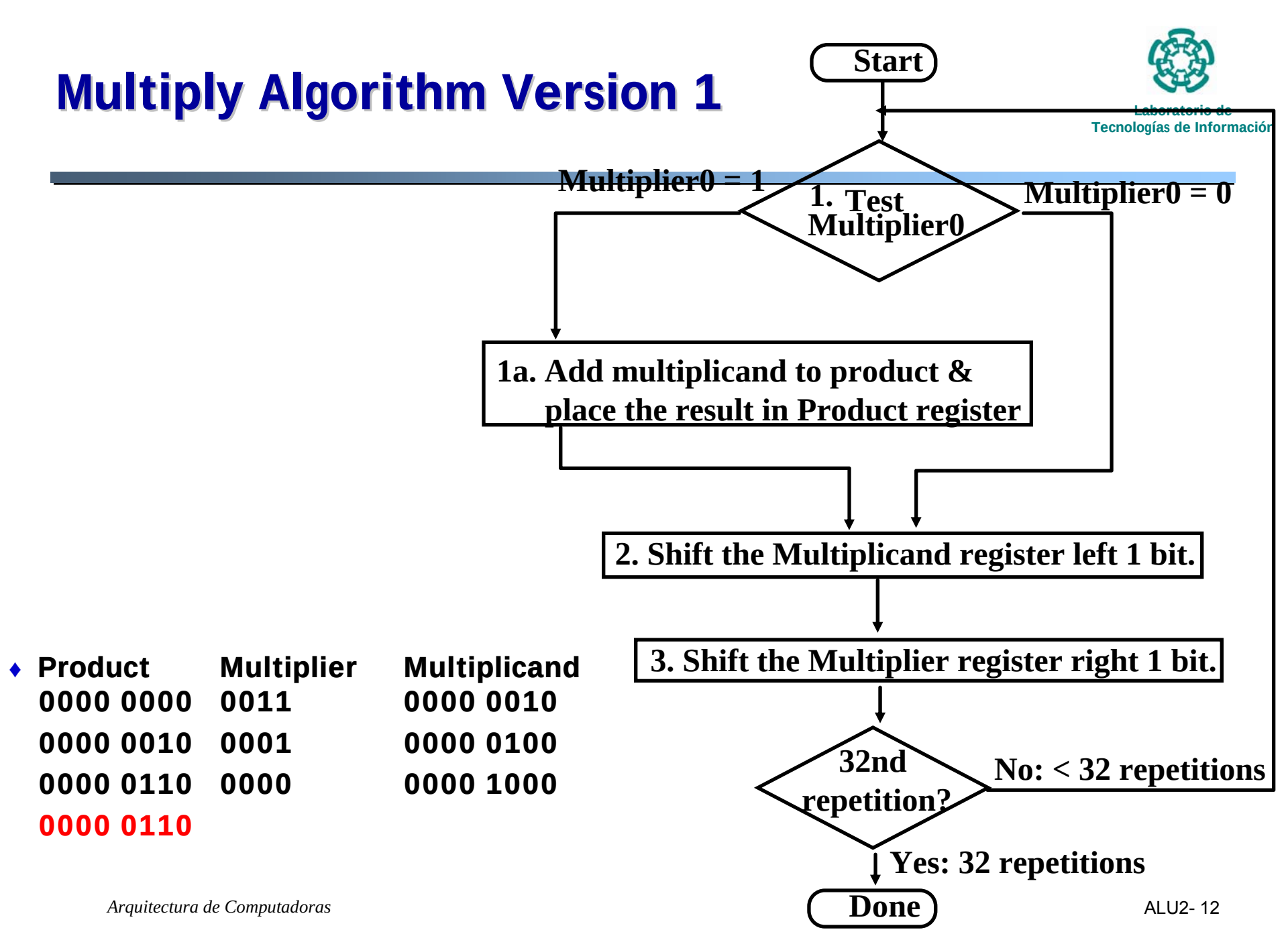
Unsigned shift-add multiplier (version 1)

- 64-bit Multiplicand reg, 64-bit ALU, 64-bit Product reg, 32-bit multiplier reg



Multiplier = datapath + control

Multiply Algorithm Version 1



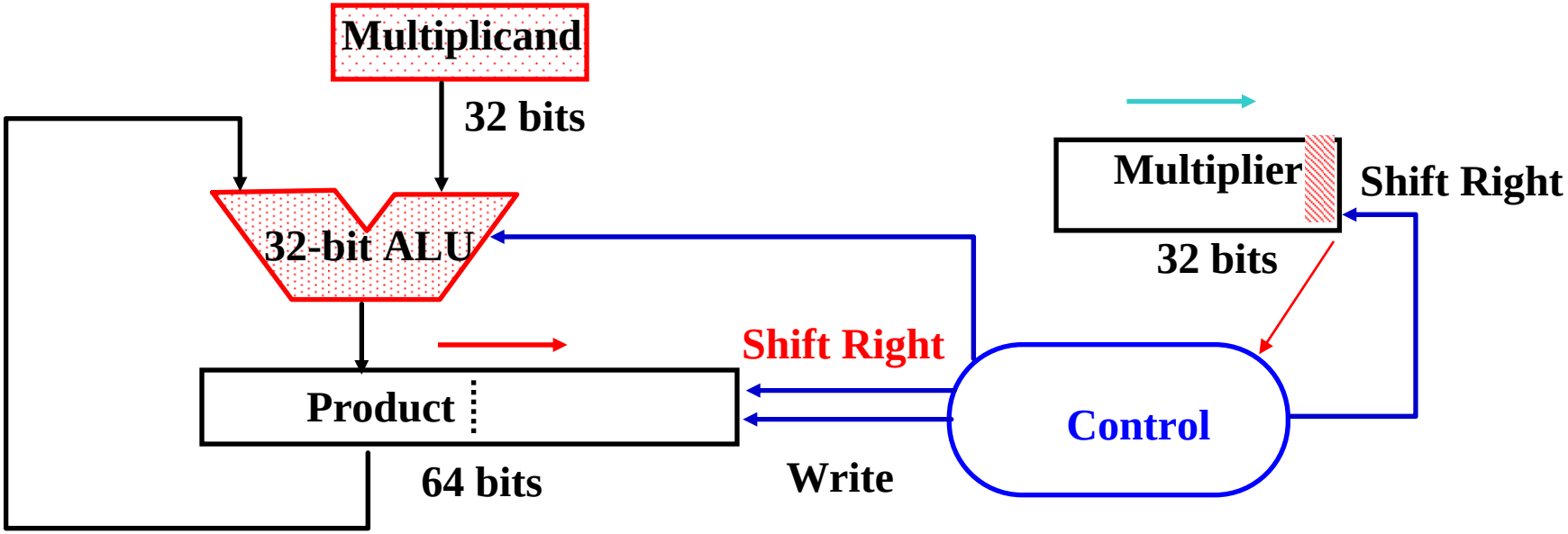
Product	Multiplier	Multiplicand
0000 0000	0011	0000 0010
0000 0010	0001	0000 0100
0000 0110	0000	0000 1000
0000 0110		

Observations on Multiply Version 1

- 1 clock per cycle => ≈ 100 clocks per multiply
  - Ratio of multiply to add 5:1 to 100:1
- Half of bits in multiplicand always 0
  => 64-bit adder is wasted
- 0's inserted in left of multiplicand as shifted
  => least significant bits of product never changed once formed
- Instead of shifting multiplicand to left, shift product to right?

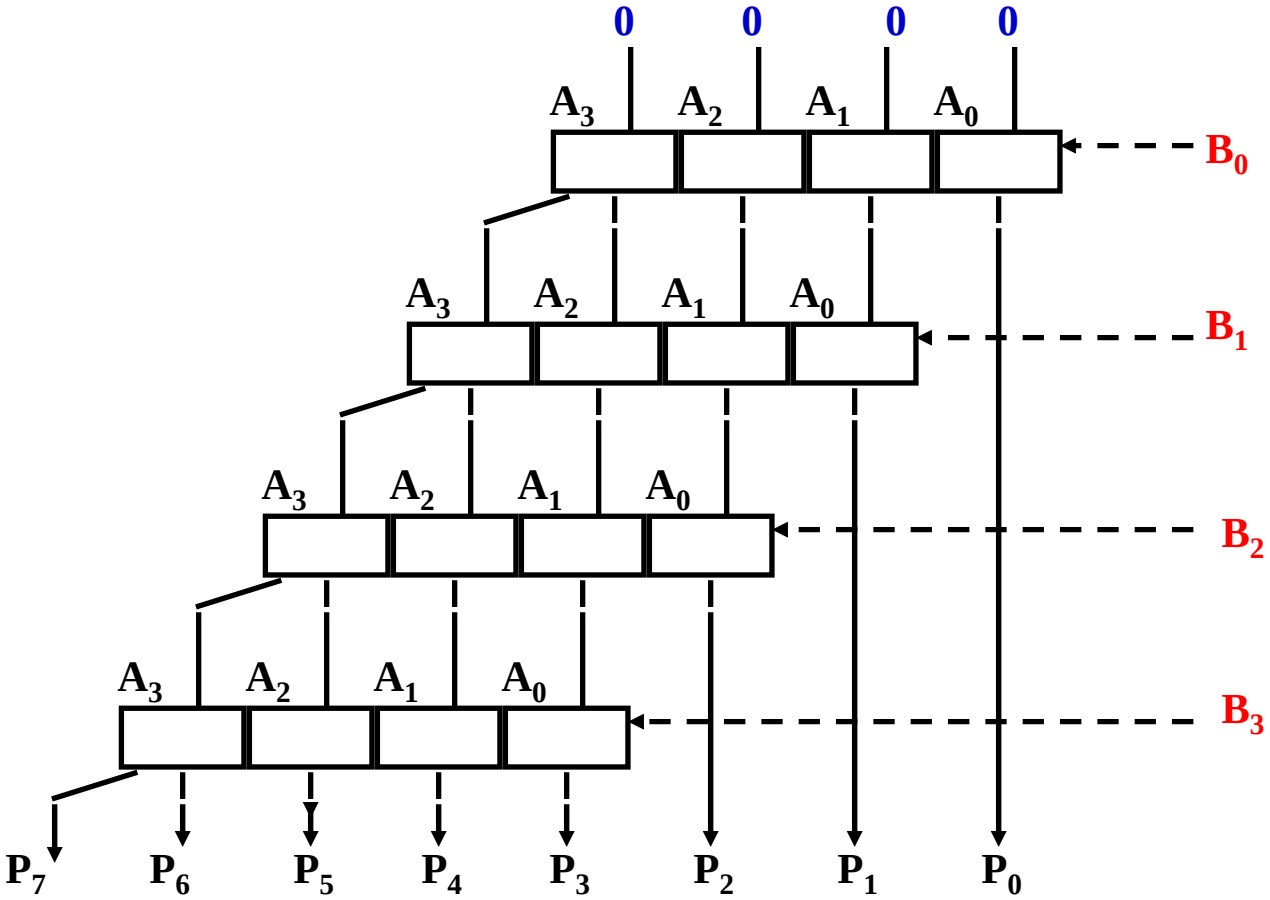
MULTIPLY HARDWARE Version 2

- 32-bit Multiplicand reg, 32 -bit ALU, 64-bit Product reg, 32-bit Multiplier reg

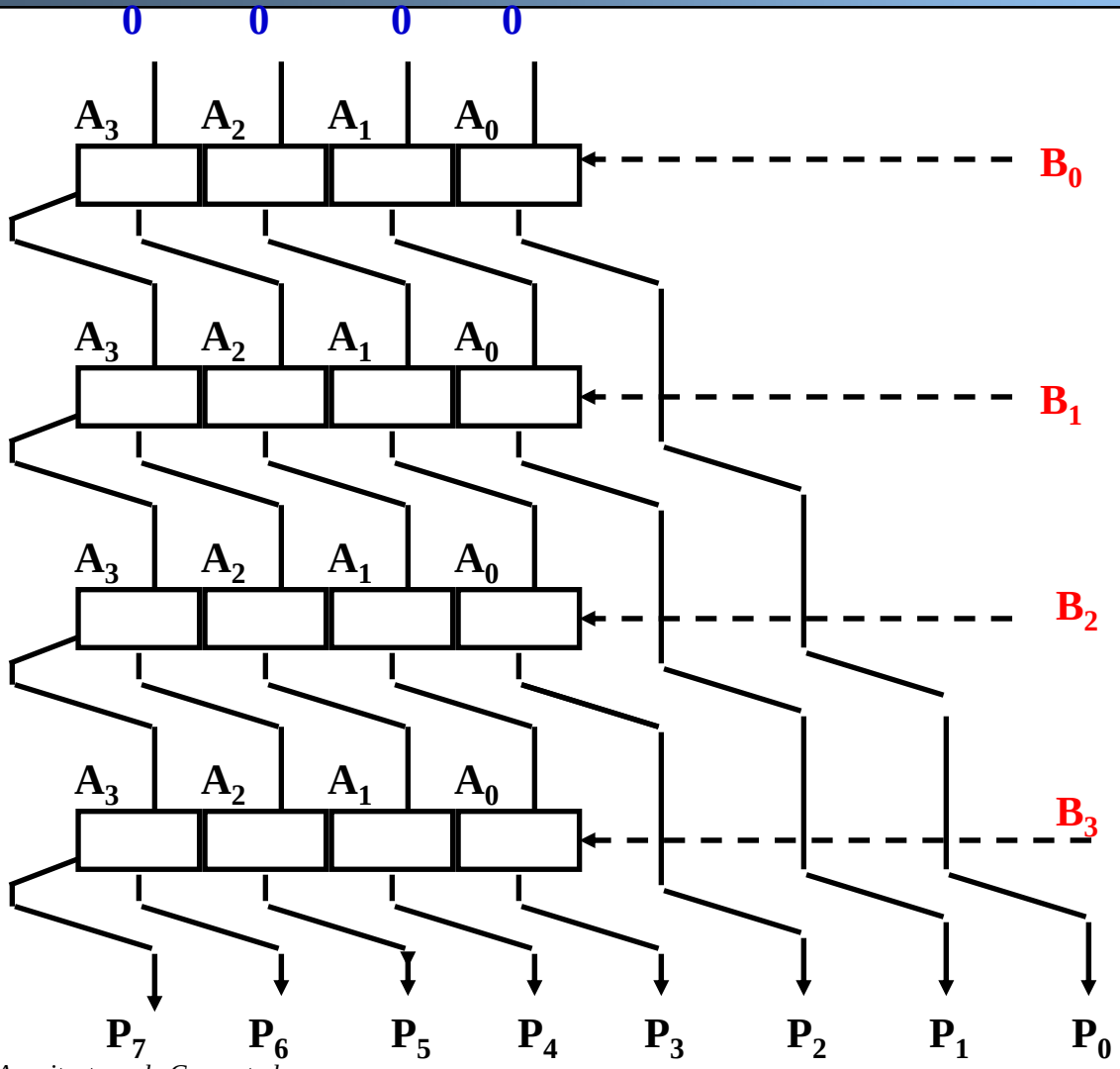


How to think of this?

Remember original combinational multiplier:

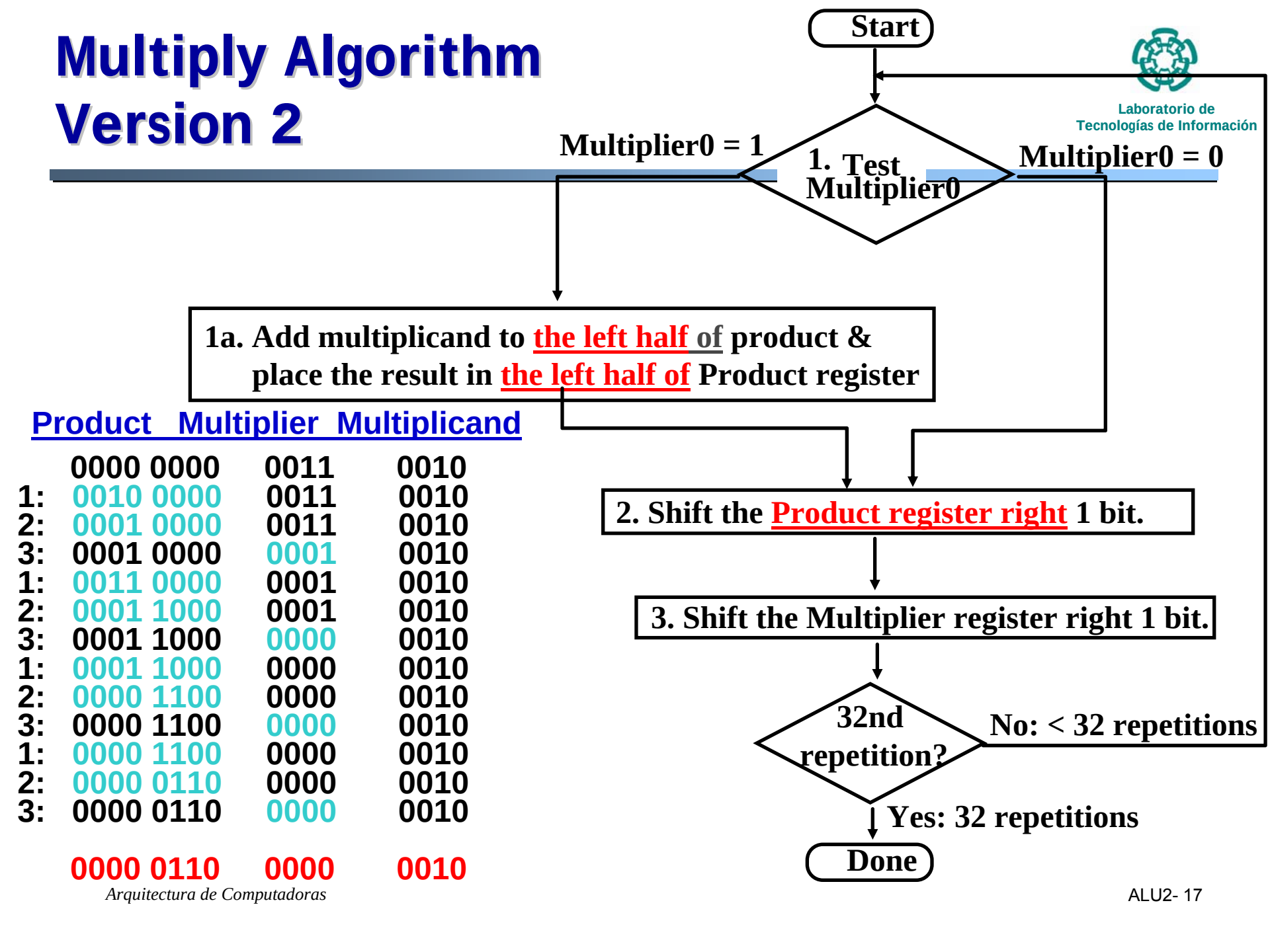


Simply warp to let product move right…



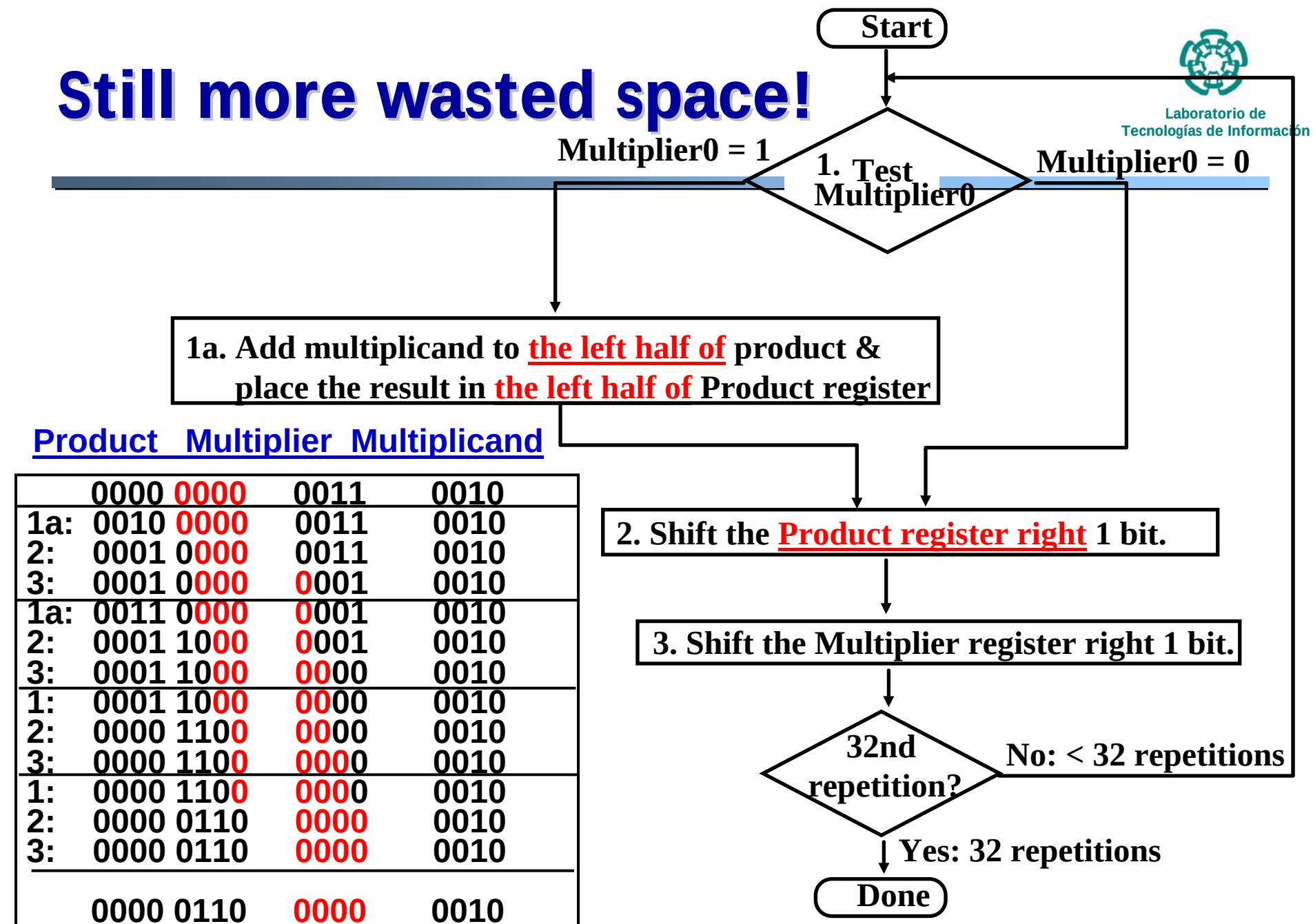
- Multiplicand stay's still and product moves right

Multiply Algorithm Version 2



	Product	Multiplier	Multiplicand
	0000 0000	0011	0010
1:	0010 0000	0011	0010
2:	0001 0000	0011	0010
3:	0001 0000	0001	0010
1:	0011 0000	0001	0010
2:	0001 1000	0001	0010
3:	0001 1000	0000	0010
1:	0001 1000	0000	0010
2:	0000 1100	0000	0010
3:	0000 1100	0000	0010
1:	0000 1100	0000	0010
2:	0000 0110	0000	0010
3:	0000 0110	0000	0010
	0000 0110	0000	0010

Still more wasted space!



	Product	Multiplier	Multiplicand
	0000 0000	0011	0010
1a:	0010 0000	0011	0010
2:	0001 0000	0011	0010
3:	0001 0000	0001	0010
1a:	0011 0000	0001	0010
2:	0001 1000	0001	0010
3:	0001 1000	0000	0010
1:	0001 1000	0000	0010
2:	0000 1100	0000	0010
3:	0000 1100	0000	0010
1:	0000 1100	0000	0010
2:	0000 0110	0000	0010
3:	0000 0110	0000	0010
	0000 0110	0000	0010

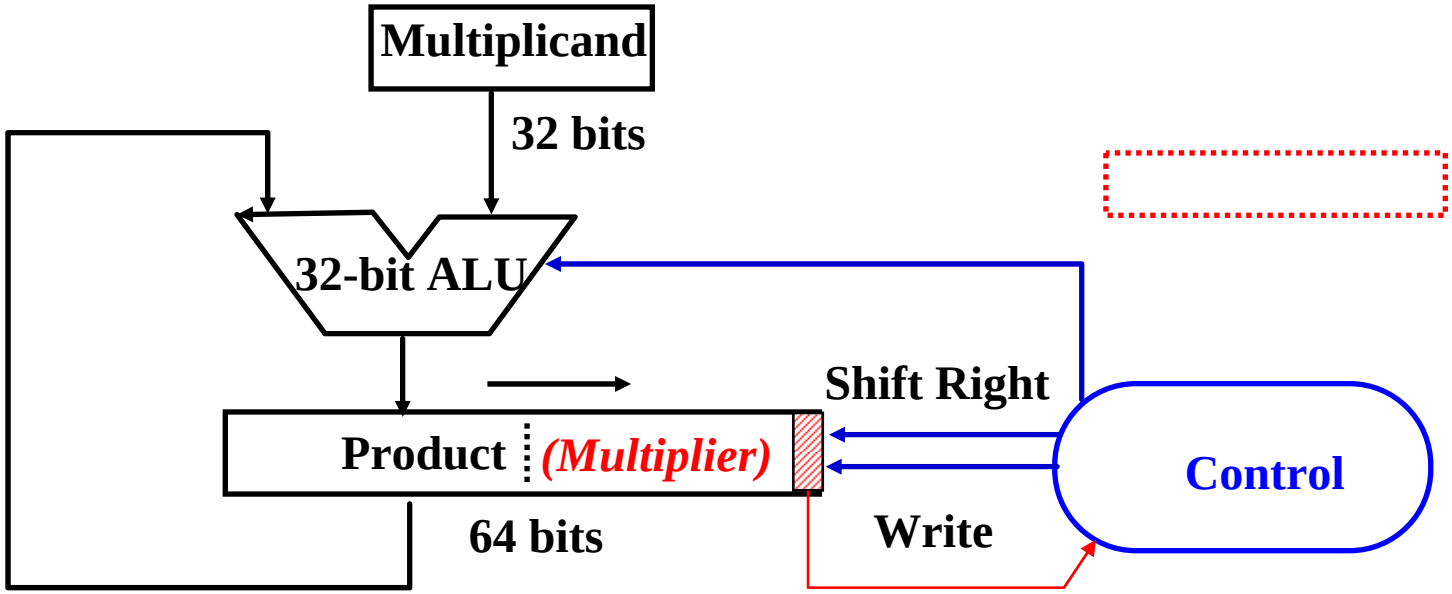
Observations on Multiply Version 2

Laboratorio de Tecnologías de Información

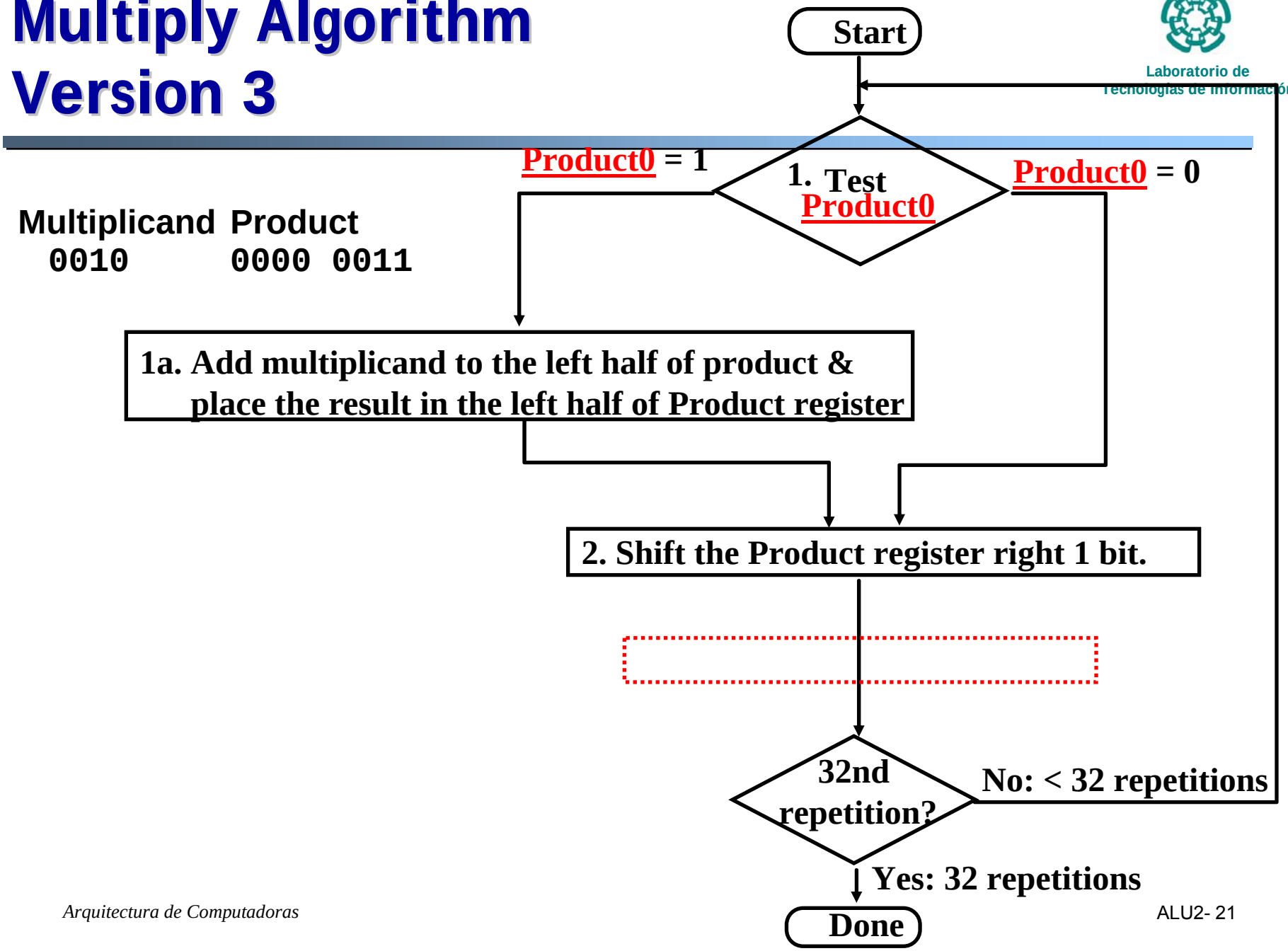
- Product register wastes space that exactly matches size of multiplier
  => combine Multiplier register and Product register

MULTIPLY HARDWARE Version 3

- 32-bit Multiplicand reg, 32 -bit ALU, 64-bit Product reg, (0-bit Multiplier reg)



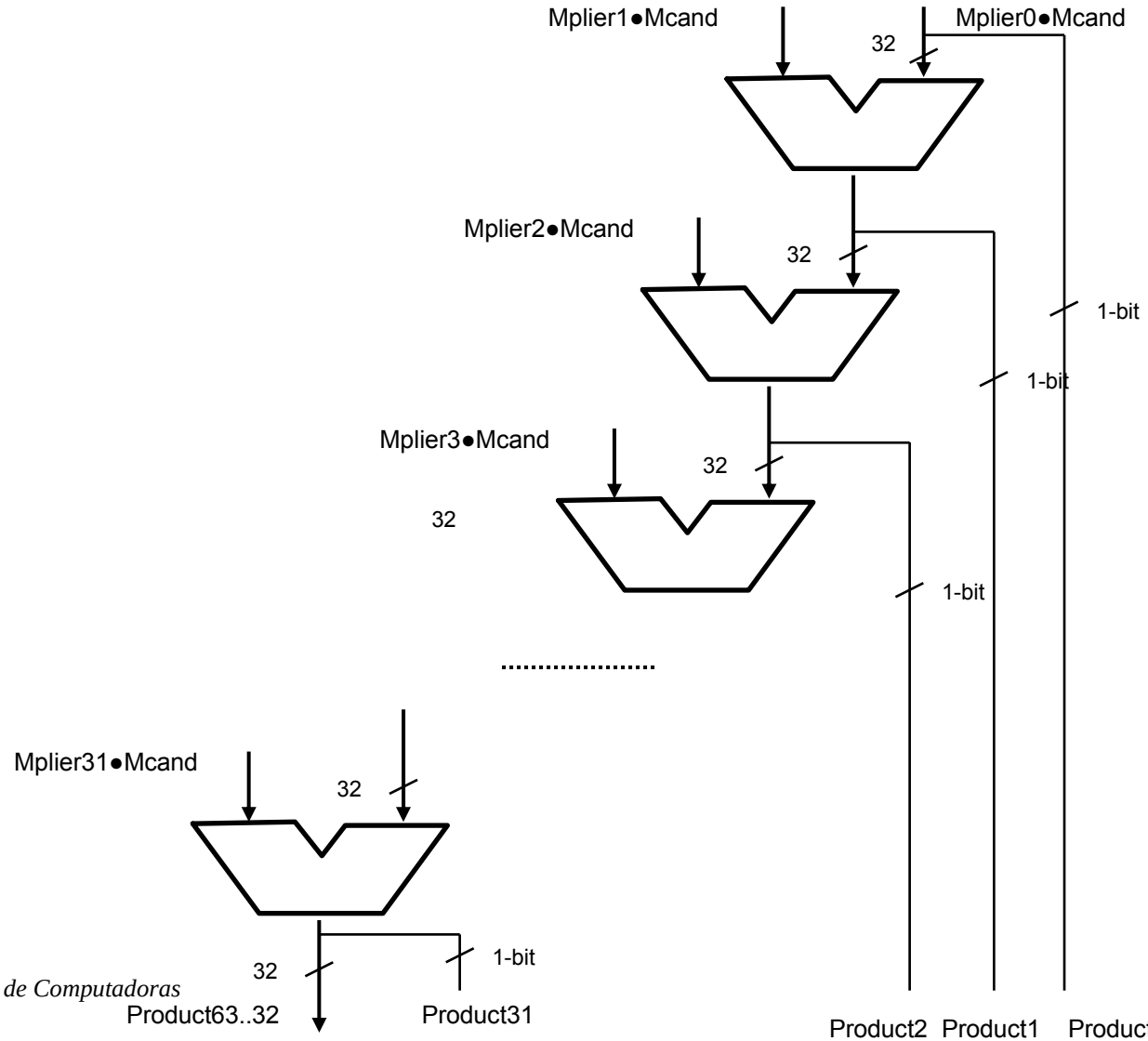
Multiply Algorithm Version 3



Observations on Multiply Version 3

- 2 steps per bit because Multiplier & Product combined
- MIPS registers Hi and Lo are left and right half of Product
- Gives us MIPS instruction MultU
- How can you make it faster?
- What about signed multiplication?
  - easiest solution is to make both positive & remember whether to complement product when done (leave out the sign bit, run for 31 steps)
  - apply definition of 2's complement
    - » need to sign-extend partial products and subtract at the end
  - Booth's Algorithm is elegant way to multiply signed numbers using same hardware as before and save cycles
    - » can handle multiple bits at a time

Fast Multiplication Hardware



Divide: Paper & Pencil


```
                    1001      Quotient
Divisor 1000  | 1001010       Dividend
              -1000
                  10
                 101
                1010
               -1000
                  10          Remainder (or Modulo result)
```

See how big a number can be subtracted, creating quotient bit on each step

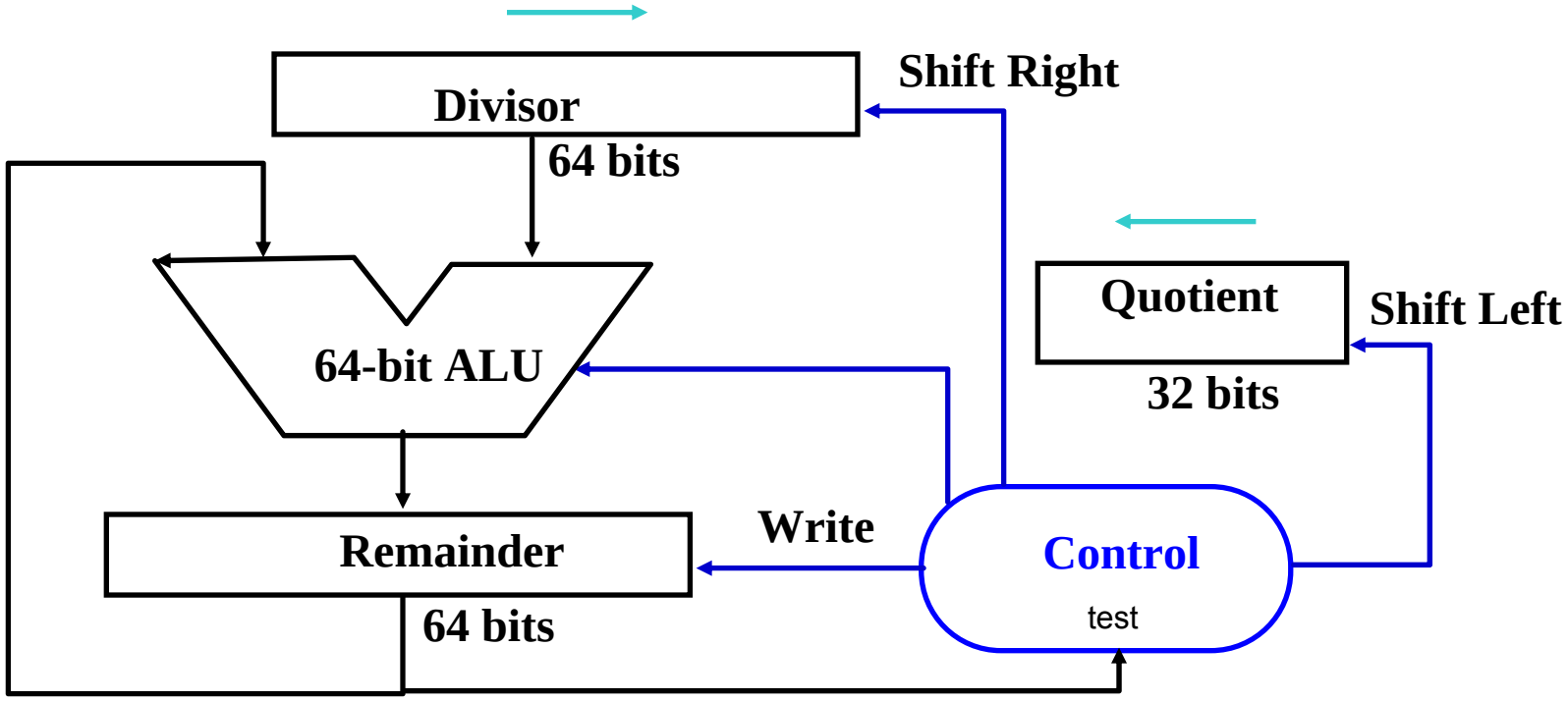
Binary => 1 * divisor or 0 * divisor

Dividend = Quotient x Divisor + Remainder
=> | Dividend | = | Quotient | + | Divisor |

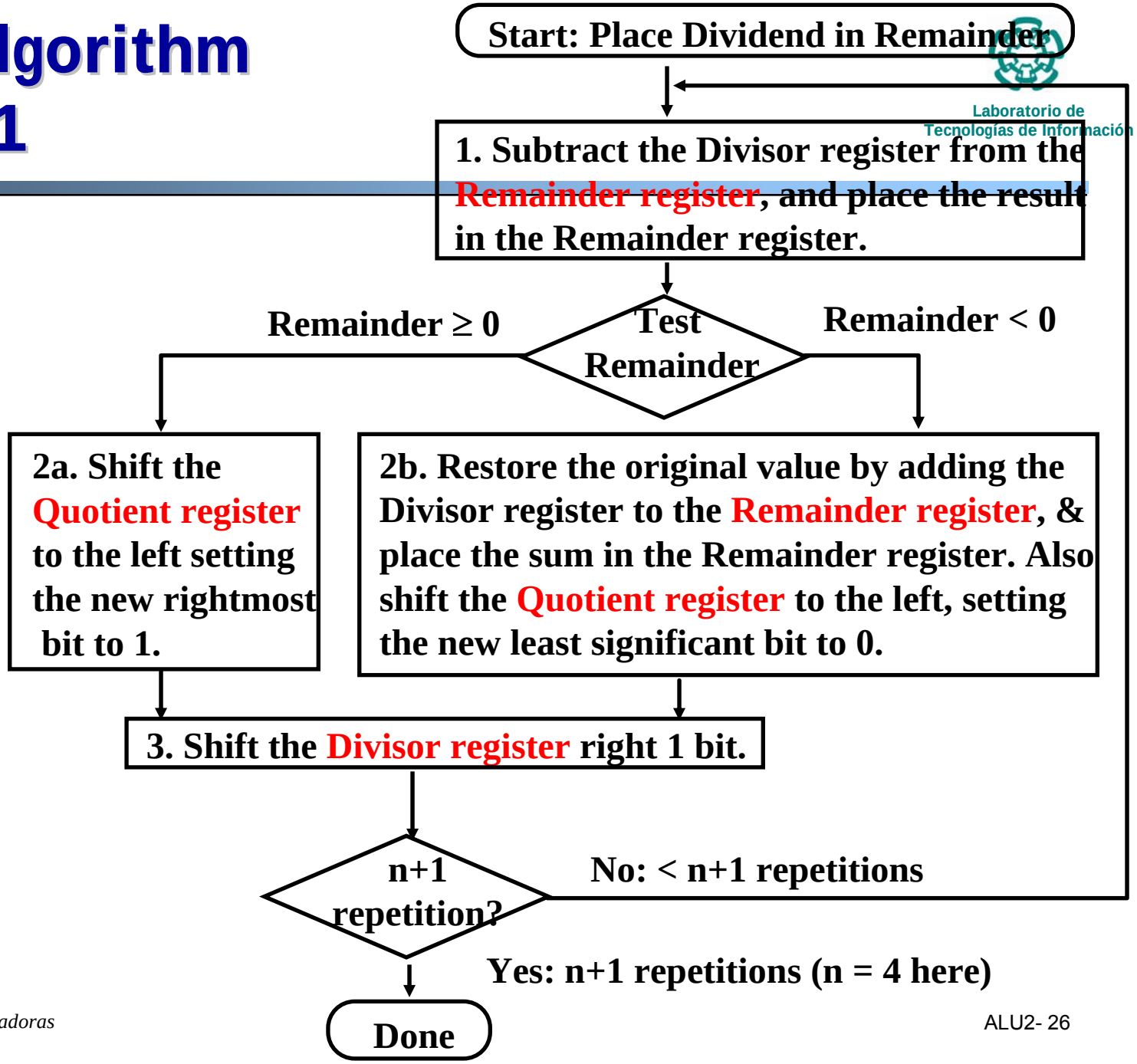
3 versions of divide, successive refinement

DIVIDE HARDWARE Version 1

- 64-bit Divisor reg, 64-bit ALU, 64-bit Remainder reg, 32-bit Quotient reg



Divide Algorithm Version 1



Divide Algorithm Version 1

Laboratorio de Tecnologías de Información

$10\overline{)111}$

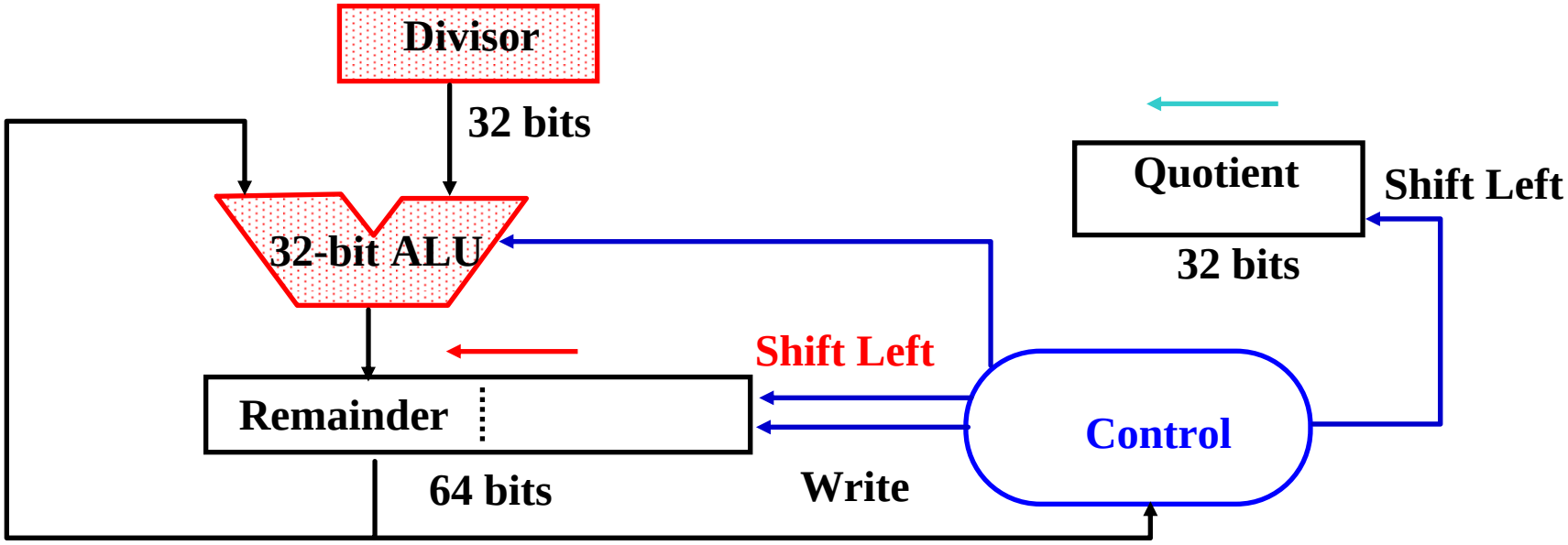
	Quotient	Divisor		Remainder	
0: Initial Values	0000	0010	0000	0000	0111
1: Rem = Rem-Div	0000	0010	0000	1110	0111
2b:Rem<0; +Div, sll Q<-0	0000	0010	0000	0000	0111
3: Shift Div Right	0000	0001	0000	0000	0111
1: Rem = Rem-Div	0000	0001	0000	1111	0111
2b:Rem<0; +Div, sll Q<-0	0000	0001	0000	0000	0111
3: Shift Div Right	0000	0000	1000	0000	0111
1: Rem = Rem-Div	0000	0000	1000	1111	1111
2b:Rem<0; +Div, sll Q<-0	0000	0000	1000	0000	0111
3: Shift Div Right	0000	0000	0100	0000	0111
1: Rem = Rem-Div	0000	0000	0100	0000	0011
2a:Rem>=0; sll Q <-1	0001	0000	0100	0000	0011
3: Shift Div Right	0001	0000	0010	0000	0011
1: Rem = Rem-Div	0001	0000	0010	0000	0001
2a:Rem>=0; sll Q <-1	0011	0000	0010	0000	0001
3: Shift Div Right	0011	0000	0001	0000	0001

Observations on Divide Version 1

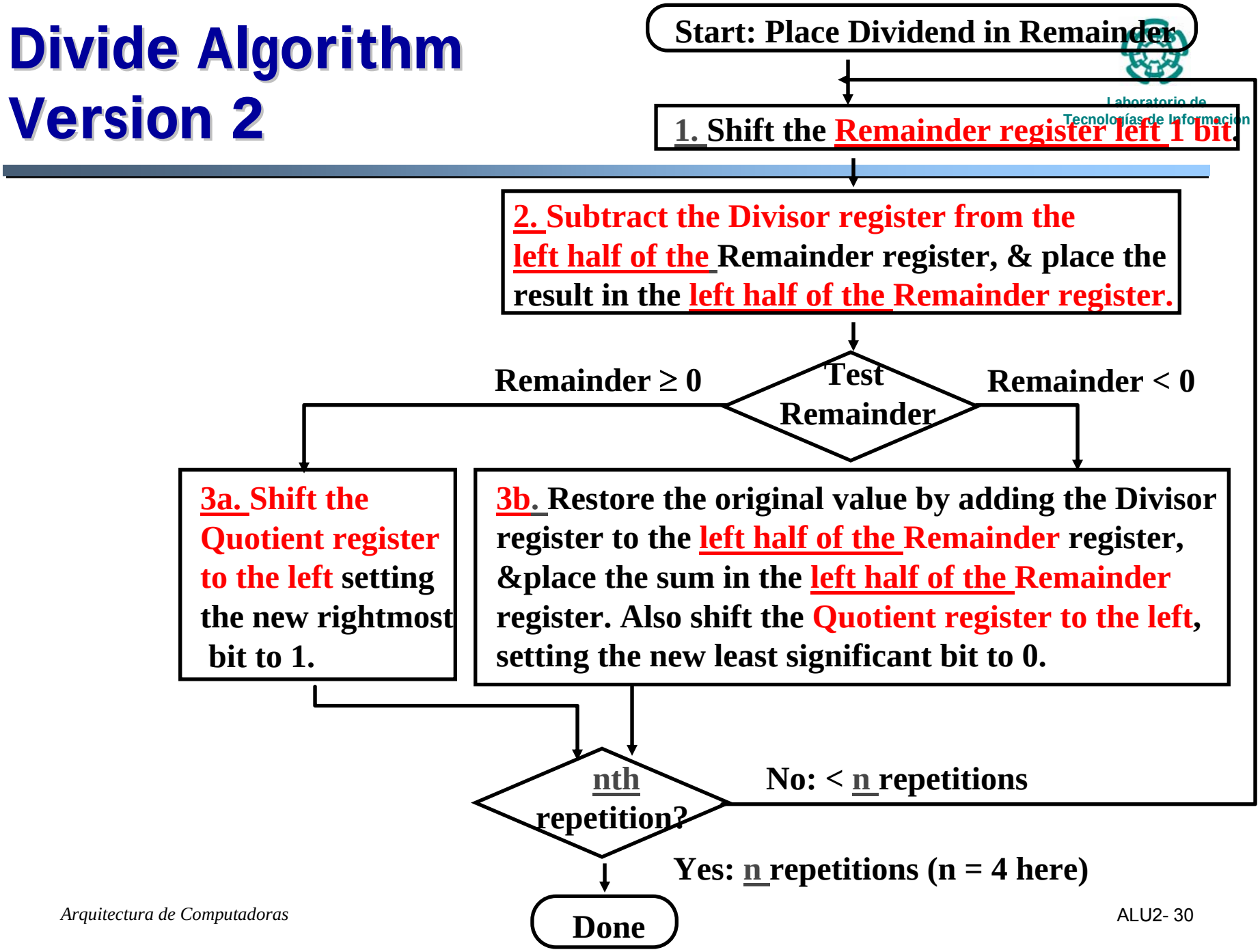
- Half of bits in divisor always 0
  => half of 64-bit adder is wasted
  => half of divisor is wasted
- Instead of shifting divisor to right,
  shift remainder to left?
- 1st step cannot produce a 1 in quotient bit (otherwise too big)
  => switch order to shift first and then subtract, can save 1 iteration

DIVIDE HARDWARE Version 2

- 32-bit Divisor reg, 32-bit ALU, 64-bit Remainder reg, 32-bit Quotient reg



Divide Algorithm Version 2



Divide Algorithm Version 2

Laboratorio de
Tecnologías de Información

	Quotient	Divisor	Remainder
0: Initial Values	0000	0010	0000 0111
1: sll Rem<-0	0000	0010	0000 1110
2: Rem = Rem-Div;	0000	0010	1110 1110
3b:Rem<0; Rem+=Div;sll Q<-0	0000	0010	0000 1110
1: sll Rem<-0	0000	0010	0001 1100
2: Rem = Rem-Div;	0000	0010	1111 1100
3b:Rem<0; Rem+=Div;sll Q<-0	0000	0010	0001 1100
1: sll Rem<-0	0000	0010	0011 1000
2: Rem = Rem-Div;	0000	0010	0001 1000
3a:Rem>0; sll Q<-1	0001	0010	0001 1000
1: sll Rem<-0	0001	0010	0011 0000
2: Rem = Rem–Div;	0001	0010	0001 0000
3a:Rem>0; sll Q<-1	0011	0010	0001 0000

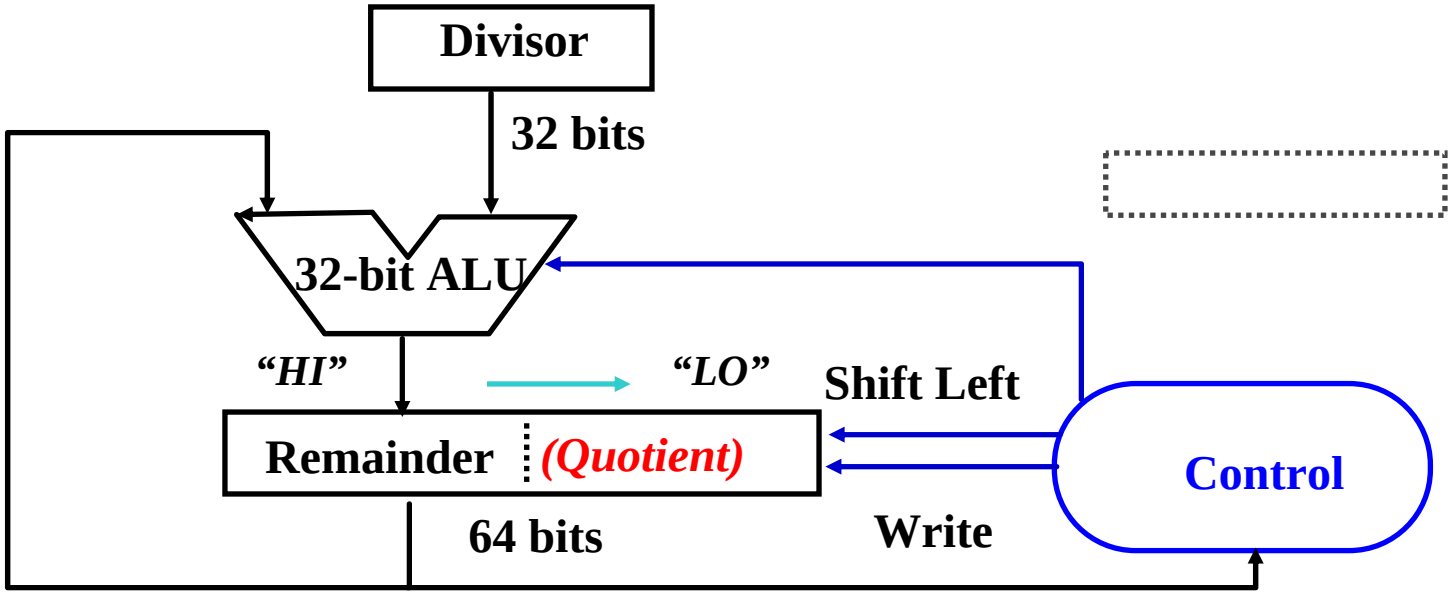
Observations on Divide Version 2

Laboratorio de
Tecnologías de Información

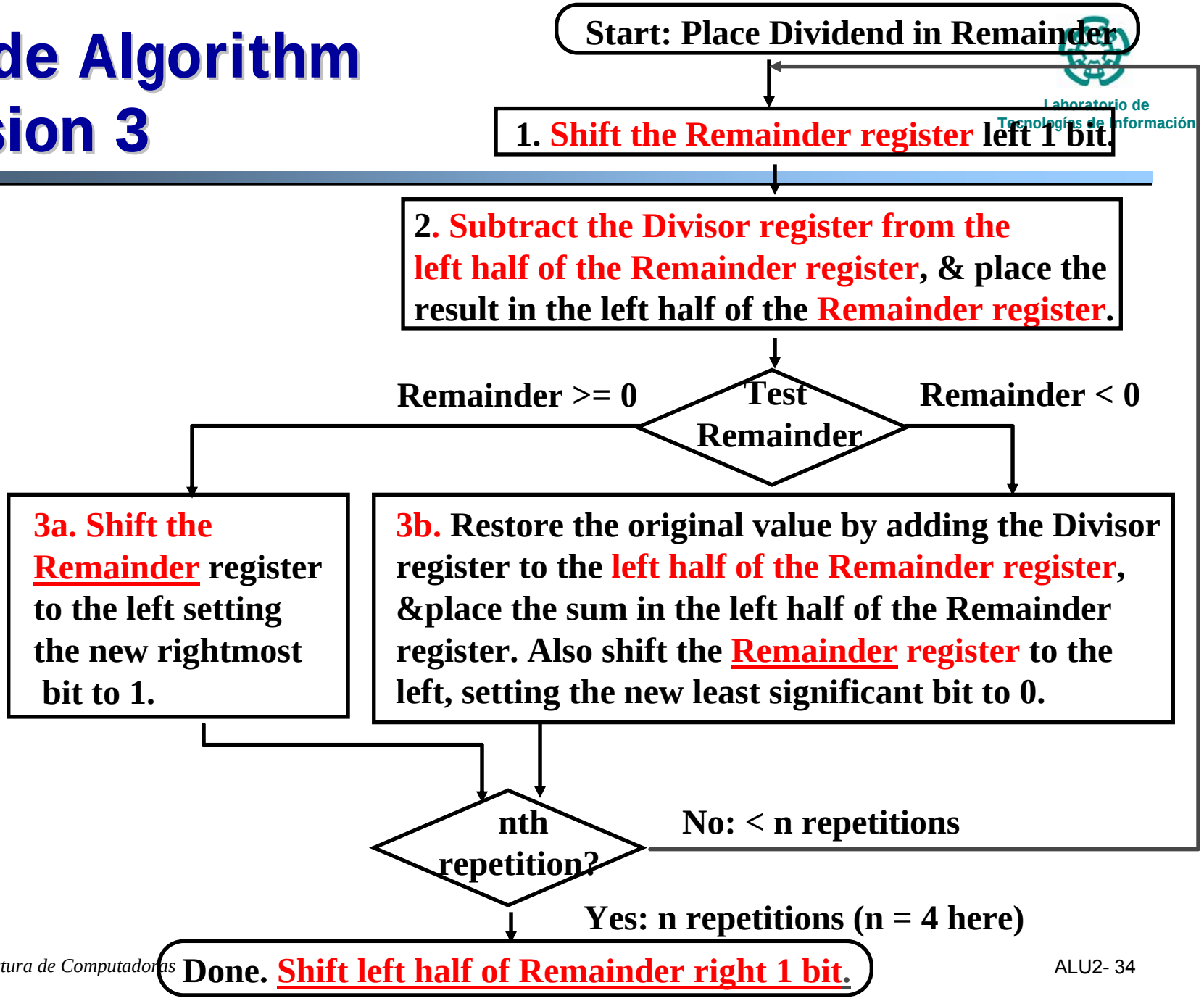
- Eliminate Quotient register by combining with Remainder as shifted left
  - Start by shifting the Remainder left as before.
  - Thereafter loop contains only two steps because the shifting of the Remainder register shifts both the remainder in the left half and the quotient in the right half
  - The consequence of combining the two registers together and the new order of the operations in the loop is that the remainder will shifted left one time too many.
  - Thus the final correction step must shift back only the remainder in the left half of the register

DIVIDE HARDWARE Version 3

- 32-bit Divisor reg, 32 -bit ALU, 64-bit Remainder reg, (0-bit Quotient reg)



Divide Algorithm Version 3



Divide Algorithm Version 3

Laboratorio de Tecnologías de Información

	Divisor	Remainder
0: Initial Values	0010	0000 0111
1: sll Rem<-0	0010	0000 1110
2: Rem = Rem-Div;	0010	1110 1110
3b:Rem<0; Rem+=Div;sll Rem<-0	0010	0000 1110
1: sll Rem<-0	0010	0001 1100
2: Rem = Rem-Div;	0010	1111 1100
3b:Rem<0; Rem+=Div;sll Rem<-0	0010	0001 1100
1: sll Rem<-0	0010	0011 1000
2: Rem = Rem-Div;	0010	0001 1000
3a:Rem>0; sll Rem<-1	0010	0001 1001
1: sll Rem<-0	0010	0011 0010
2: Rem = Rem–Div;	0010	0001 0010
3a:Rem>0; sll Rem<-1	0010	0001 0011

Observations on Divide Version 3

- Same Hardware as Multiply: just need ALU to add or subtract, and 63-bit register to shift left or shift right
- Hi and Lo registers in MIPS combine to act as 64-bit register for multiply and divide
- Signed Divides: Simplest is to remember signs, make positive, and complement quotient and remainder if necessary
  - Note: Dividend and Remainder must have same sign
  - Note: Quotient negated if Divisor sign & Dividend sign disagree e.g., –7 ÷ 2 = –3, remainder = –1
- Possible for quotient to be too large: if divide 64-bit integer by 1, quotient is 64 bits (“called saturation”)

Summary

- Multiply: successive refinement to see final design
  - 32-bit Adder, 64-bit shift register, 32-bit Multiplicand Register
- Division: successive refinement to see final design
  - 32-bit Adder, 64-bit shift register, 32-bit Divisor Register

Hw5: Assignments

Laboratorio de
Tecnologías de Información

1	Claudia Méndez Garza	Design an 8-bit carry look ahead adder Develop equations Draw a schematic Indicate delays
2	José Ramírez Uresti	Design a signed 8x8 multiplier + times +, - times – + times –, - times +
3	Víctor Echeverría Ríos	Design a signed 8x8 divider + times +, - times – + times –, - times +

- Prepare a ppt presentation explaining the design and send it by e-mail
- Due date: October 13th, 2007.