Lesson 1. The Arithmetical-Logical Unit

Computer Structure and Organization

Degree in Computer Sciences

Degree in Computers Engneering

Double Degree in Computer Science Engineering and Business Administration and Management

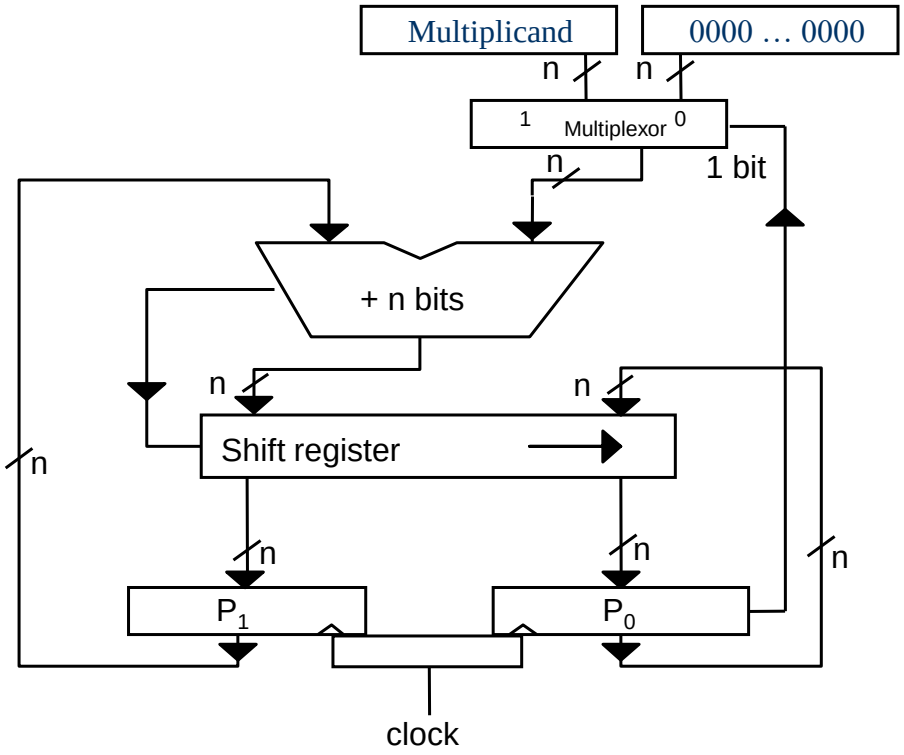
Automatic Department

Contents

ALU algorithms (I)
Mutiplication: ADD-SHIFT (I)

- **Multiplication. Add-Shift algorithm (A x B)**
- Only works with unsigned numbers
- An initial phase and two steps (add and shift) per bit of the Multiplier
- Initiate P_0 = B
- Initiate P_1 = 0
- Initiate **SR** = 0

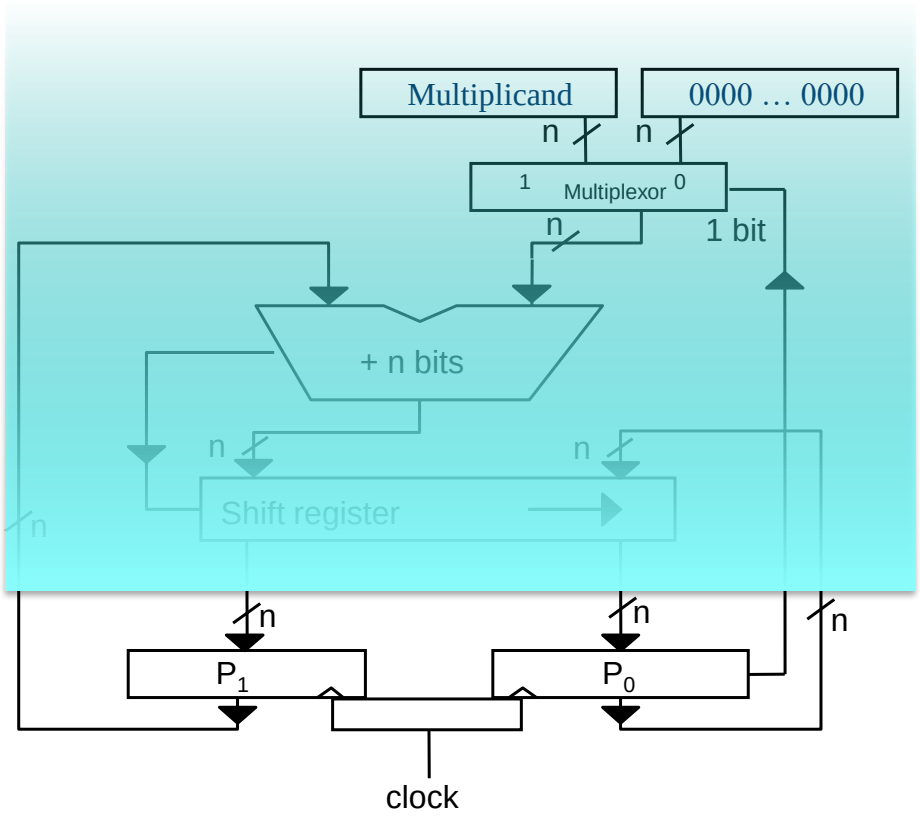
(SR means Shift Register)



ALU algorithms (II)
Mutiplication: ADD-SHIFT (II)

- **Multiplication. Add-Shift algorithm (A x B)**
- **First step**:

  1.- IF **LSB** of P_0 = 1
  Then ADD P_1 , **Multiplicand**
  Else ADD P_1, 0
  2.- Store the result in the upper bits of the **Shift Register**
  3.- Copy P_0 in the lower bits of the **Shift Register**



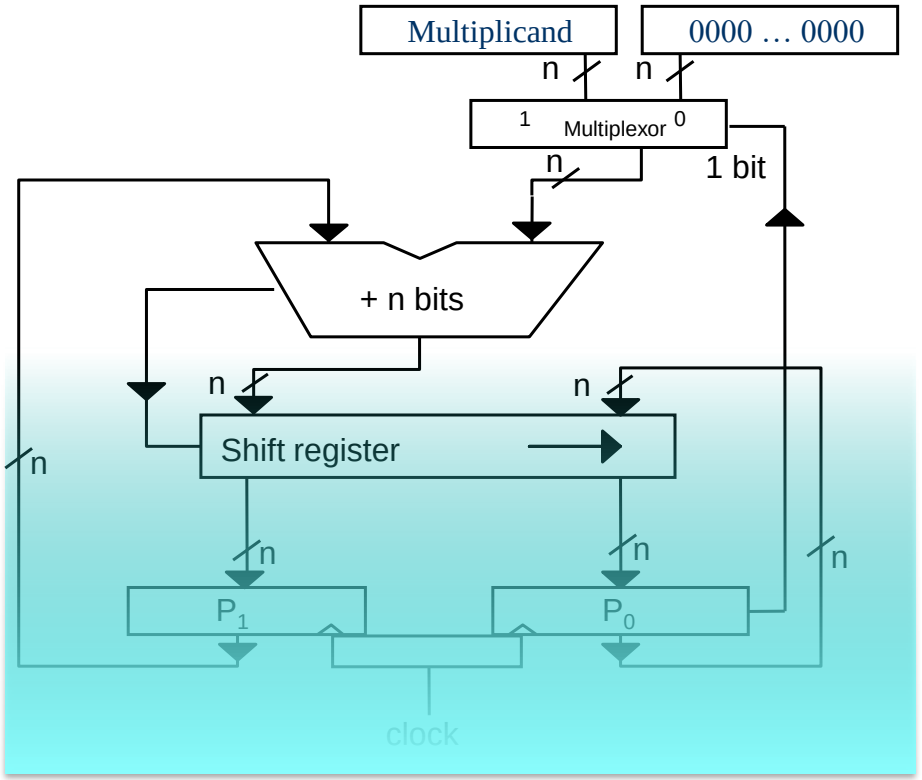
ALU algorithms (III)
Mutiplication: ADD-SHIFT (III)

- **Multiplication. Add-Shift algorithm (A x B)**
- **Second step**:

  1.- Shift once to the left the **Shift Register** by using previous carry output

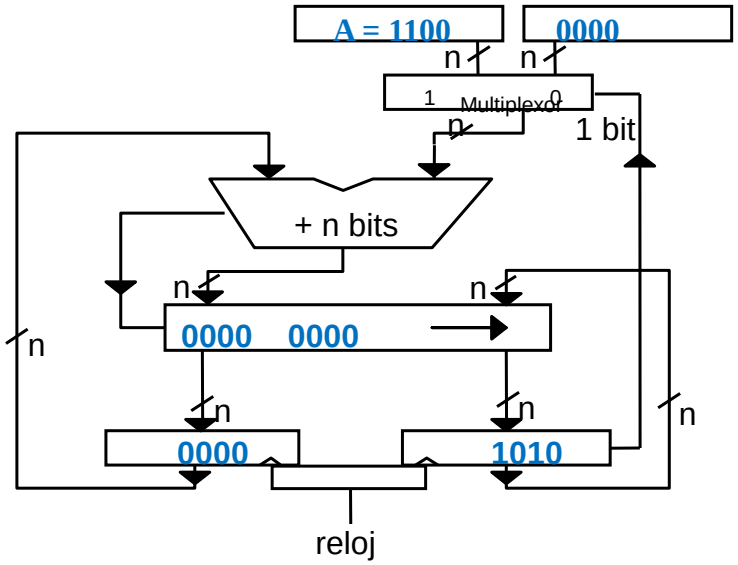
  2.- Store in P_1 the upper bits of the **Shift Register**

  3.- Store in P_0 the lower bits of the **Shift Register**



ALU algorithms (IV)
Mutiplication: ADD-SHIFT (IV)

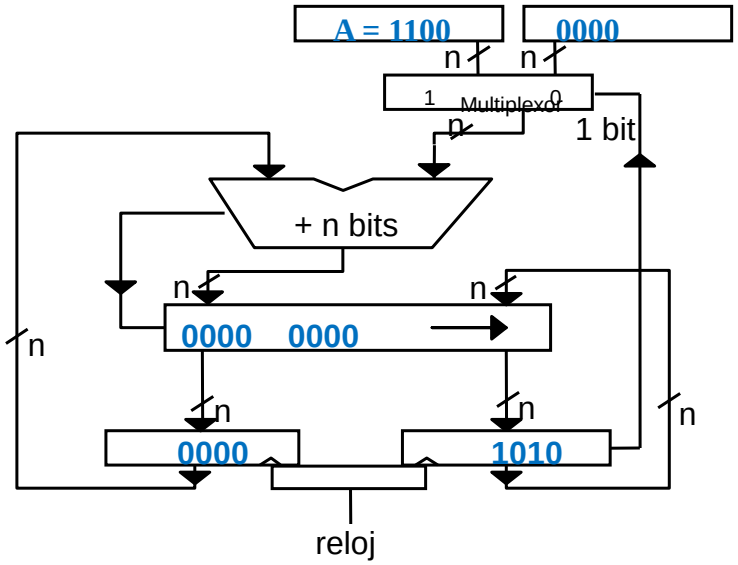
- **Get A x B if A = 1100 y B = 1010**



Shift register	P		Operation
	P_1	P_0	

ALU algorithms (V)
Mutiplication: ADD-SHIFT (V)

- **Get A x B if A = 1100 y B = 1010**

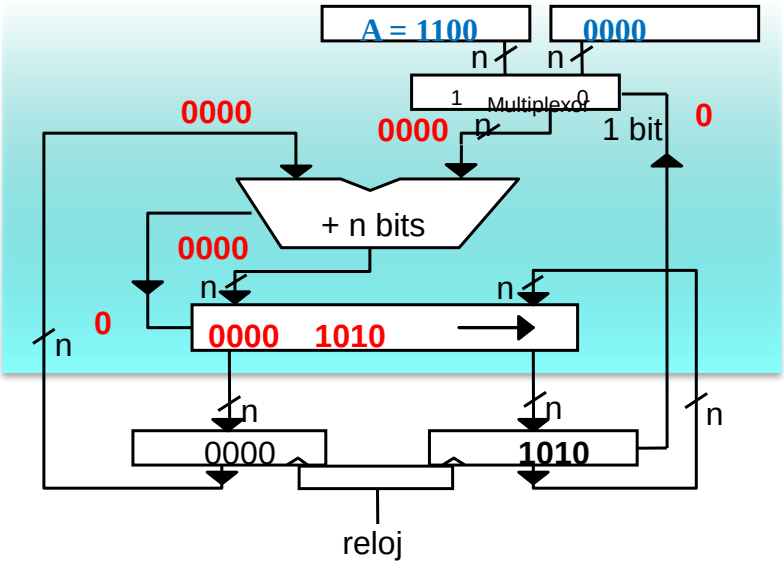


Shift register	P		Operation
	P_1	P_0	
0000 0000	0000	1010	Initial State

- **Initial state:**
  - **Shift Register = 0000 0000**
  - **P_1 = 0**
  - **P_0 = B = 1010**

ALU algorithms (VI)
Mutiplication: ADD-SHIFT (VI)

- **Get A x B if A = 1100 y B = 1010**



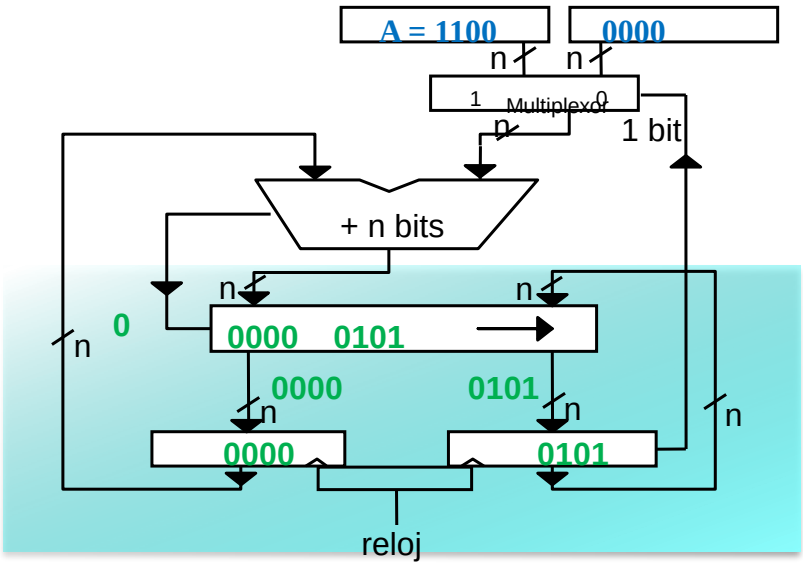
Shift register	P		Operation
	P_1	P_0	
0000 0000	0000	1010	Initial State
0000 1010	**“”**	**“”**	**Add 1**

- **Step 1, add 1:**
  - **LSB (P_0) = 0 → upper (SR) = P_1 + 0**
  - **Carry Output = 0**
  - **P_0 → Lower (SR)**

(SR means Shift Register)

ALU algorithms (VII)
Mutiplication: ADD-SHIFT (VII)

- **Get A x B if A = 1100 y B = 1010**



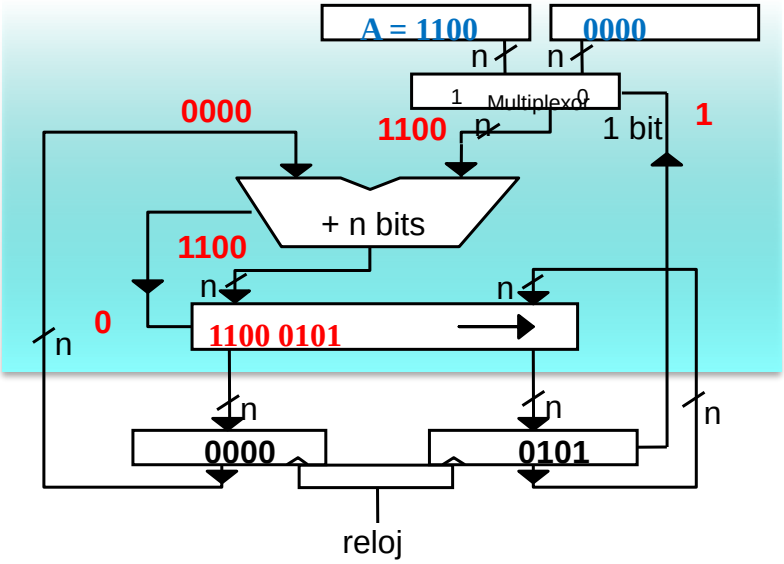
Shift register	P		Operation
	P_1	P_0	
0000 0000	0000	1010	Initial State
0000 1010	“”	“”	Add 1
0000 0101	**0000**	**0101**	**Shift 1**

- **Step 2, shift 1:**
  - **Shift SR, 1 using carry output (0)**
  - **Upper (SR) → P_1**
  - **Lower (SR) → P_0**

(SR means Shift Register)

ALU algorithms (VIII)
Mutiplication: ADD-SHIFT (VIII)

- **Get A x B if A = 1100 y B = 1010**



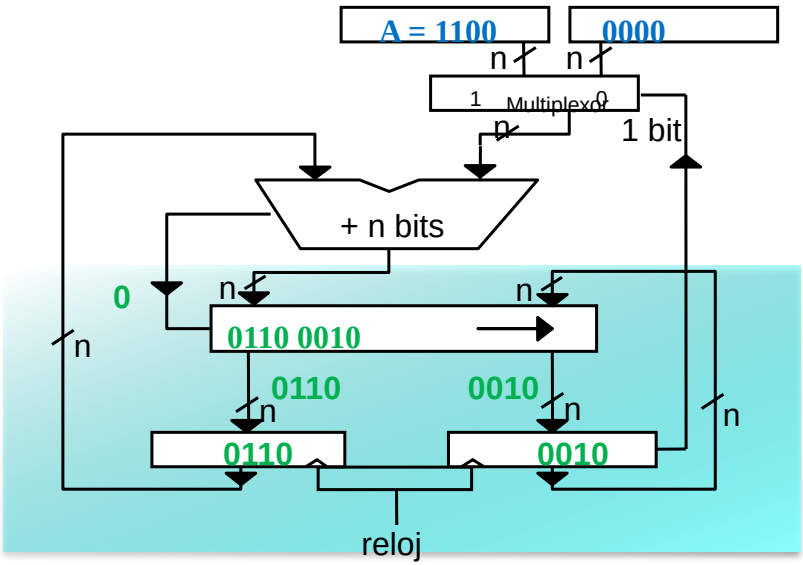
Shift register	P		Operation
	P_1	P_0	
0000 0000	0000	1010	Initial State
0000 1010	“”	“”	Add 1
0000 0101	0000	0101	Shift 1
1100 0101	**“”**	**“”**	**Add 2**

- **Step 1, add 2:**
  - **LSB (P_0) = 1 → upper (SR) = P_1 + A**
  - **Carry Output = 0**
  - **P_0 → Lower (SR)**

(SR means Shift Register)

ALU algorithms (IX)
Mutiplication: ADD-SHIFT (IX)

- **Get A x B if A = 1100 y B = 1010**



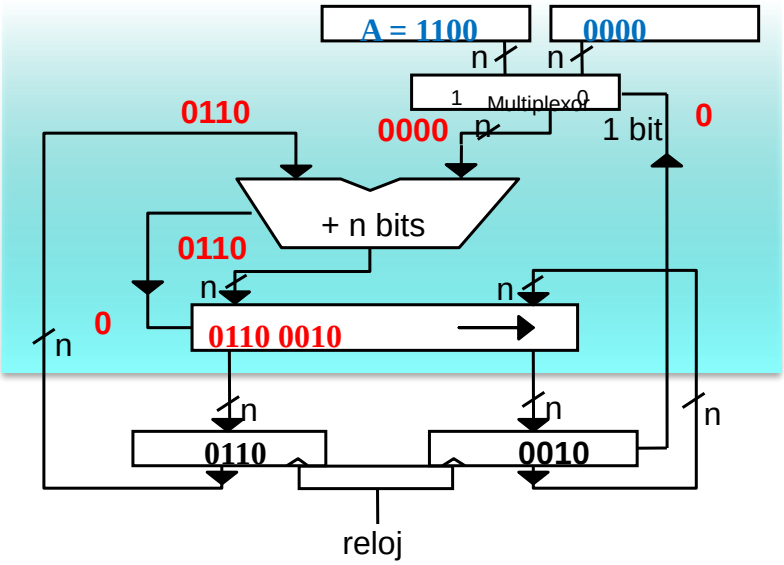
Shift register	P		Operation
	P_1	P_0	
0000 0000	0000	1010	Initial State
0000 1010	“”	“”	Add 1
0000 0101	0000	0101	Shift 1
1100 0101	“”	“”	Add 2
0110 0010	**0110**	**0010**	**Shift 2**

- **Step 2, shift 2:**
  - **Shift SR, 1 using carry output (0)**
  - **Upper (SR) → P_1**
  - **Lower (SR) → P_0**

(SR means Shift Register)

ALU algorithms (X)
Mutiplication: ADD-SHIFT (X)

- **Get A x B if A = 1100 y B = 1010**



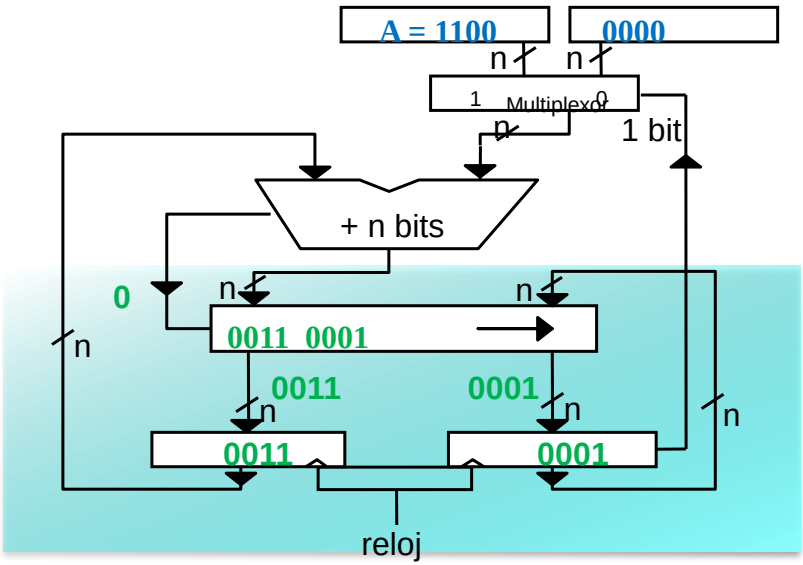
Shift register	P		Operation
	P_1	P_0	
0000 0000	0000	1010	Initial State
0000 1010	“”	“”	Add 1
0000 0101	0000	0101	Shift 1
1100 0101	“”	“”	Add 2
0110 0010	0110	0010	Shift 2
0110 0010	**“”**	**“”**	**Add 3**

- **Step 1, add 3:**
  - **LSB (P_0) = 0 → upper (SR) = P_1 + 0**
  - **Carry Output = 0**
  - **P_0 → Lower (SR)**

(SR means Shift Register)

ALU algorithms (XI)
Mutiplication: ADD-SHIFT (XI)

- **Get A x B if A = 1100 y B = 1010**



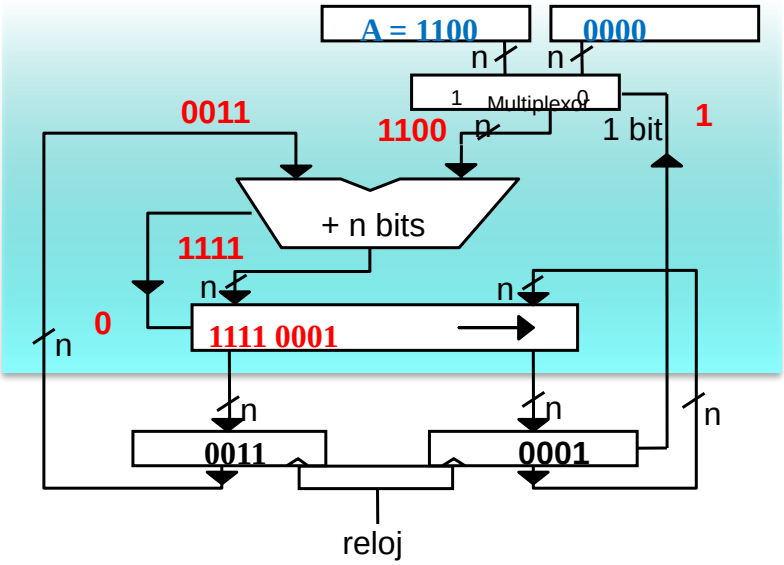
Shift register	P		Operation
	P_1	P_0	
0000 0000	0000	1010	Initial State
0000 1010	“”	“”	Add 1
0000 0101	0000	0101	Shift 1
1100 0101	“”	“”	Add 2
0110 0010	0110	0010	Shift 2
0110 0010	“”	“”	Add 3
0011 0001	**0011**	**0001**	**Shift 3**

- **Step 2, shift 3:**
  - **Shift SR, 1 using carry output (0)**
  - **Upper (SR) → P_1**
  - **Lower (SR) → P_0**

(SR means Shift Register)

ALU algorithms (XII)
Mutiplication: ADD-SHIFT (XII)

- **Get A x B if A = 1100 y B = 1010**



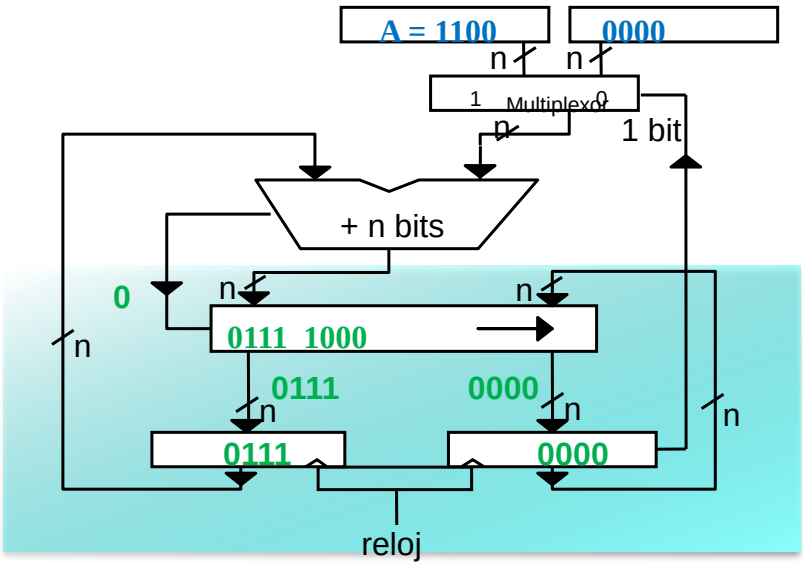
Shift register	P		Operation
	P_1	P_0	
0000 0000	0000	1010	Initial State
0000 1010	“”	“”	Add 1
0000 0101	0000	0101	Shift 1
1100 0101	“”	“”	Add 2
0110 0010	0110	0010	Shift 2
0110 0010	“”	“”	Add 3
0011 0001	0011	0001	Shift 3
1111 0001	..	..	Add 4

- **Step 1, add 4:**
  - **LSB (P_0) = 1 → upper (SR) = P_1 + A**
  - **Carry Output = 0**
  - **P_0 → Lower (SR)**

(SR means Shift Register)

ALU algorithms (XIII)
Mutiplication: ADD-SHIFT (XIII)

- **Get A x B if A = 1100 y B = 1010**



Shift register	P		Operation
	P_1	P_0	
0000 0000	0000	1010	Initial State
0000 1010	“”	“”	Add 1
0000 0101	0000	0101	Shift 1
1100 0101	“”	“”	Add 2
0110 0010	0110	0010	Shift 2
0110 0010	“”	“”	Add 3
0011 0001	0011	0001	Shift 3
1111 0001	..	..	Add 4
0111 1000	**0111**	**1000**	**Shift 4**

- **Step 2, shift 4 and finish:**
  - **Shift SR, 1 using carry output (0)**
  - **Upper (SR) → P_1**
  - **Lower (SR) → P_0**

(SR means Shift Register)

ALU algorithms (XIV)
Mutiplication: ADD-SHIFT (XIV)

- Original Add-Shift algorithm only works with unsigned numbers.

ALU algorithms (XV)
Mutiplication: ADD-SHIFT (XV)

- Original Add-Shift algorithm only works with unsigned numbers.
- What can we do to multiply signed numbers?

ALU algorithms (XVI)
Mutiplication: ADD-SHIFT (XVI)

- Original Add-Shift algorithm only works with unsigned numbers.
- What can we do to multiply signed numbers?
- Complement 2 and complement 1 adjusts for the algorithm

ALU algorithms (XVII)
Mutiplication: ADD-SHIFT (XVII)

- **Original Add-Shift algorithm only works with unsigned numbers.**
- **What can we do to multiply signed numbers?**
- **Complement 2 and complement 1 adjusts for the algorithm**
  - **Complement 2**: if B is a negative number, Subtract A when last 1 bit arrives to multiplexor

ALU algorithms (XVIII)
Mutiplication: ADD-SHIFT (XVIII)

- **Original Add-Shift algorithm only works with unsigned numbers.**
- **What can we do to multiply signed numbers?**
- **Complement 2 and complement 1 adjusts for the algorithm**
  - **Complement 1**: if B is a negative number, Subtract A when last 1 bit arrives to multiplexor

ALU algorithms (XIX)
Mutiplication: ADD-SHIFT (XIX)

- **Original Add-Shift algorithm only works with unsigned numbers.**
- **What can we do to multiply signed numbers?**
- **Complement 2 and complement 1 adjusts for the algorithm**
  - **Complement 1**: if B is a negative number, Subtract A when last 1 bit arrives to multiplexor. In addition, P_1 must be initiated to A instead of 0

ALU algorithms (XX)
Mutiplication: ADD-SHIFT (and XX)

- **Original Add-Shift algorithm only works with unsigned numbers.**
- **What can we do to multiply signed numbers?**
- **Complement 2 and complement 1 adjusts for the algorithm**
  - **Complement 2**: if B is a negative number, Subtract A when last 1 bit arrives to multiplexor
  - **Complement 1**:if B is a negative number, Subtract A when last 1 bit arrives to multiplexor. In addition, P_1 must be initiated to A instead of 0

ALU algorithms (XXI)
Mutiplication: Booth's algorithms (I)

- **Multiplication. Booth Algorithm (C2 and usnsigned numbers)**
  - Searches for bit strings of 1's and 0's because the operand is shifted in these cases only.
  - We need define 3 numbers: A, S and P where:
    - A is the multiplicand concatenated **at its right side** with a number of zeros equal to its length in bits
    - S is the complement to 2 of the multiplicand concatenated **at its right side** with a number of zeros equal to its length in bits
    - P is the multiplier concatenated **at its left side** with a number of zeros equal to its length in bits
  - We need to add a column, denotated as **Q, at the right side**, with and initial value of 0

ALU algorithms (XXII)
Mutiplication: Booth's algorithms (II)

- We must compare the LSB of P and the Q bit. Depending of the combination we must do the following actions:
  - 00 → Shift
  - 01 → P + A and shift
  - 10 → P + S and shift
  - 11 → Shift
- The number of steps of this algorithm is the same as the number of bits of the multiplier

ALU algorithms (XXIII)
Mutiplication: Booth's algorithms (III)

- Let's be **A = 1100** and **B = 0110**. Both number are represented in complement to 2.
- **A = 1100_2** = -4_{10} and **B = 0110_2** = 6_{10}, so A x B = -4_{10} x 6_{10} = -24_{10}
- **-A = C2(A) = 0100_2** = 4_{10}

- First step is to prepare A, S, P and Q.

Action	Temporary Products	Q
A	**1100** 0000	0
S	**0100** 0000	0
P	0000 **0110**	0

ALU algorithms (XXIV)
Mutiplication: Booth's algorithms (IV)

Action	Temporary Products	Q
A	1100 0000	0
S	0100 0000	0
P	0000 0110	0
P'	0000 0011	0

00 → Shift
01 → P + A and shift
10 → P + S and shift
11 → Shift

ALU algorithms (XXV)
Mutiplication: Booth's algorithms (V)

Action	Temporary Products	Q
A	1100 0000	0
S	0100 0000	0
P	0000 0110	0
P'	0000 0011	0
P' + S	0100 0011	0
P''	0010 0001	1

00 → Shift
01 → P + A and shift
10 → P + S and shift
11 → Shift

ALU algorithms (XXVI)
Mutiplication: Booth's algorithms (VI)

Action	Temporary Products	Q
A	1100 0000	0
S	0100 0000	0
P	0000 0110	0
P'	0000 0011	0
P' + S	0100 0011	0
P''	0010 0001	1
P'''	0001 0000	1

00 → Shift
01 → P + A and shift
10 → P + S and shift
11 → Shift

ALU algorithms (XXVII)
Mutiplication: Booth's algorithms (VII)

Action	Temporary Products	Q
A	1100 0000	0
S	0100 0000	0
P	0000 0110	0
P'	0000 0011	0
P' + S	0100 0011	0
P''	0010 0001	1
P'''	0001 0000	1
P''' + A	1101 0000	1
P''''	1110 1000	0

00 → Shift
01 → P + A and shift
10 → P + S and shift
11 → Shift

ALU algorithms (XXVIII)
Mutiplication: Booth's algorithms (& VIII)

Action	Temporary Products	Q
A	1100 0000	0
S	0100 0000	0
P	0000 0110	0
P'	0000 0011	0
P' + S	0100 0011	0
P''	0010 0001	1
P'''	0001 0000	1
P''' + A	1101 0000	1
P''''	1110 1000	0
Result	**1110 1000**	**0**

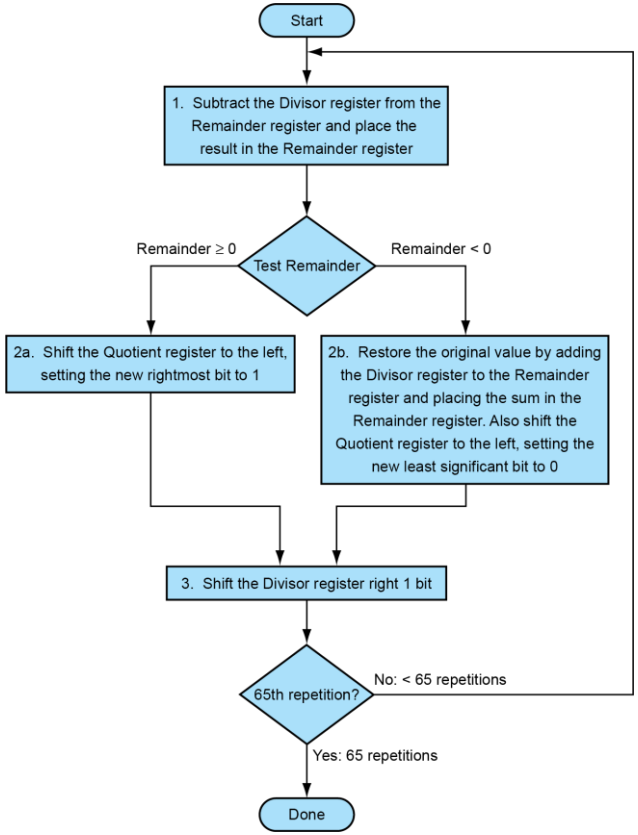
00 → Shift
01 → P + A and shift
10 → P + S and shift
11 → Shift

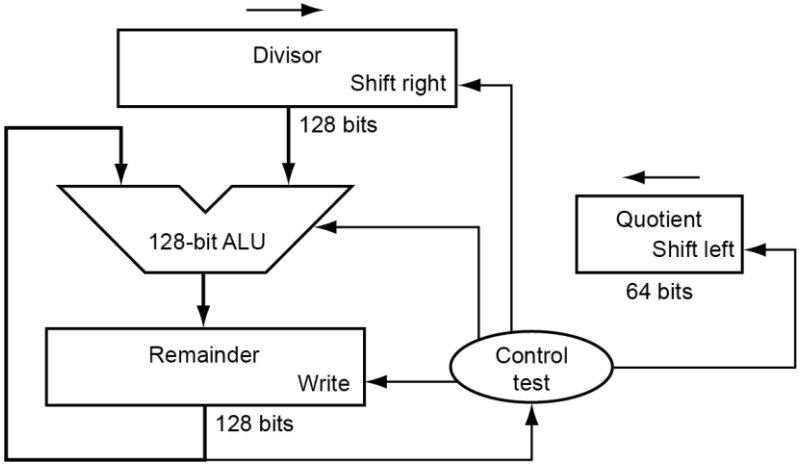
ALU algorithms (XXIX)
Division:with Remainder restoration (I)

- **Division. Unsigned "With Remainder" Restoration Division Algorithm**
  - A = 0100011, B = 0011, Quotient = 1011, Remainder = 0010

```
 0 1 0 0 0 1 1   | 0011
+1 1 0 1 ↓       |  1 0 1 1
1 0 0 0 1 0      |
  + 1 1 0 1      |
    1 1 1 1      |
    0 0 1 0 1    |  ← Restauración
    + 1 1 0 1    |
    1 0 0 1 0 1  |
      +1 1 0 1   |
      1 0 0 1 0  |
```

ALU algorithms (XXX)
Division:with Remainder restoration (II)





Images from David A. Patterson and John L. Hennessy. Computer Organization and Design. RISC-V Edition

ALU algorithms (XXXI)
Division:with Remainder restoration (III)

- **Division with remainder restoration (A / B)**
- Only works with unsigned numbers
- An initial phase and three steps per bit of the Dividend
- Initiate **Divisor** = B (left half)
- Initiate **Remainder** = A
- Initiate **Quotient** = 0

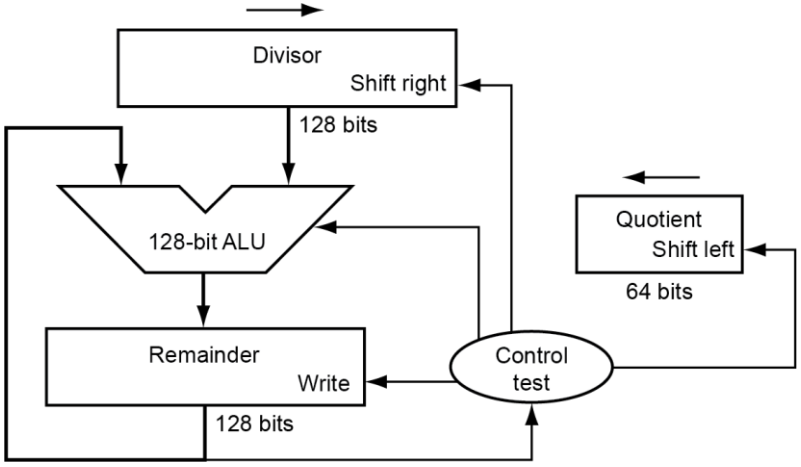


Image from David A. Patterson and John L. Hennessy.
Computer Organization and Design. RISC-V Edition

ALU algorithms (XXXII)
Division:with Remainder restoration (IV)

- **Division with remainder restoration (A / B)**
- **First step**:
  Remainder = Remainder - Divisor

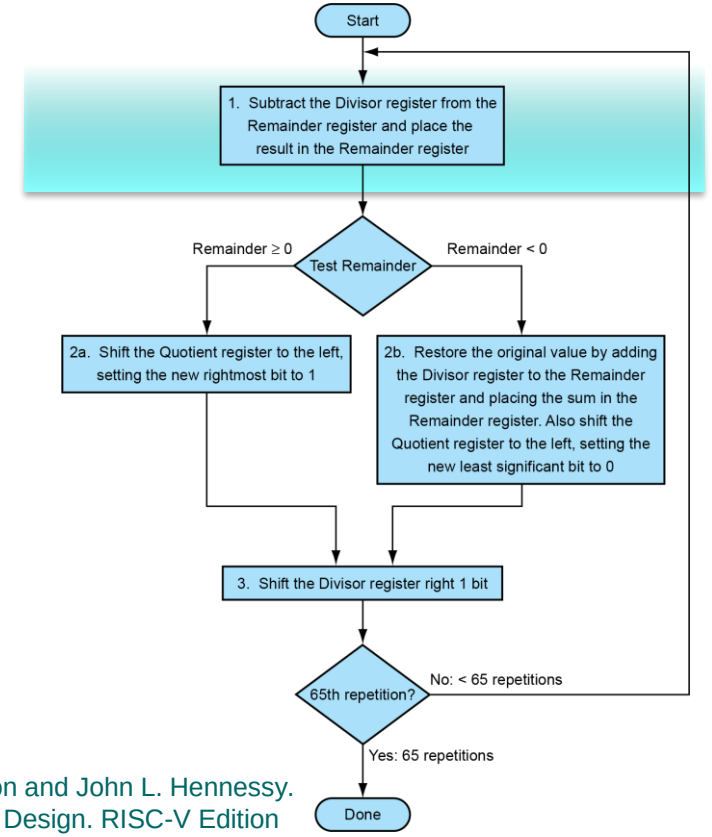


Image from David A. Patterson and John L. Hennessy. Computer Organization and Design. RISC-V Edition

ALU algorithms (XXXIII)
Division:with Remainder restoration (V)

- **Division with remainder restoration (A / B)**

- **Second steps**:

  2a → If **Remainder** > 0 then **Quotient** must be shifted to the left by imputing a 1 on the right

  2b → If **Remainder** < 0 then Restore previous **Remainder** **Quotient** must be shifted to the left by imputing a 0 on the right

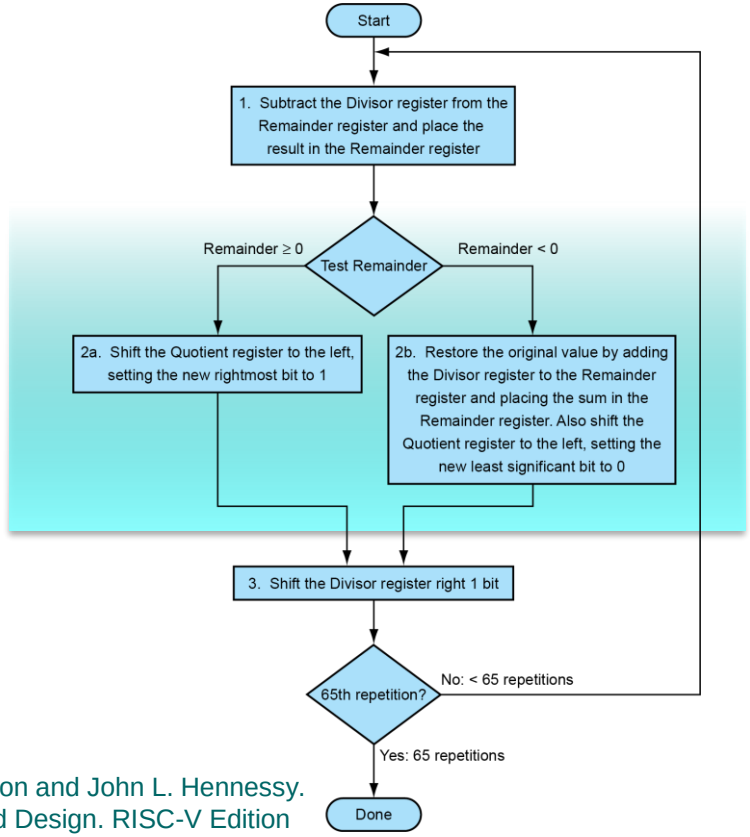


Image from David A. Patterson and John L. Hennessy. Computer Organization and Design. RISC-V Edition

ALU algorithms (XXXIV)
Division:with Remainder restoration (VI)

- **Division with remainder restoration (A / B)**

- **Third step**:
  Shift one position to the right the **Divisor**

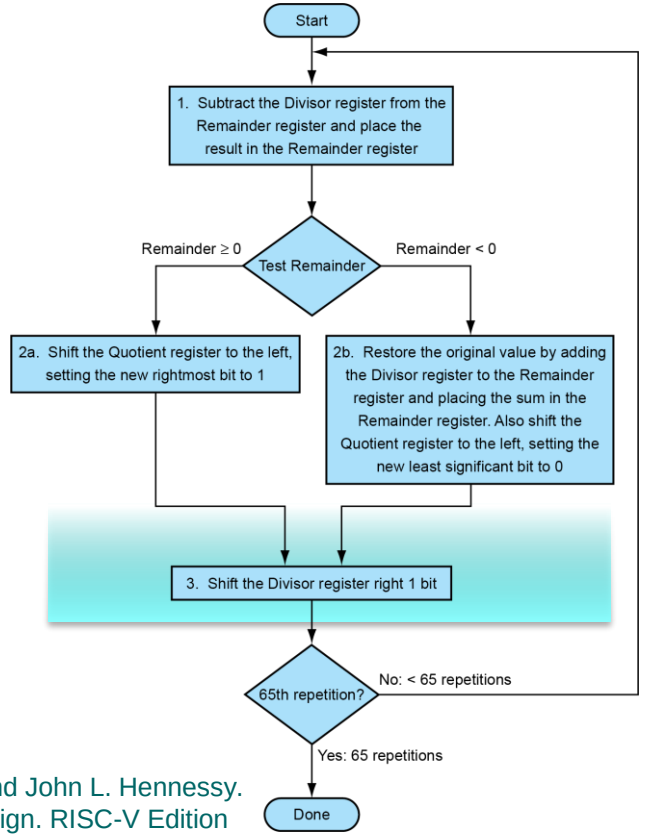


Image from David A. Patterson and John L. Hennessy. Computer Organization and Design. RISC-V Edition

ALU algorithms (XXXV)
Division:with Remainder restoration (VII)

Get A / B if A = 1101 y B = 0100

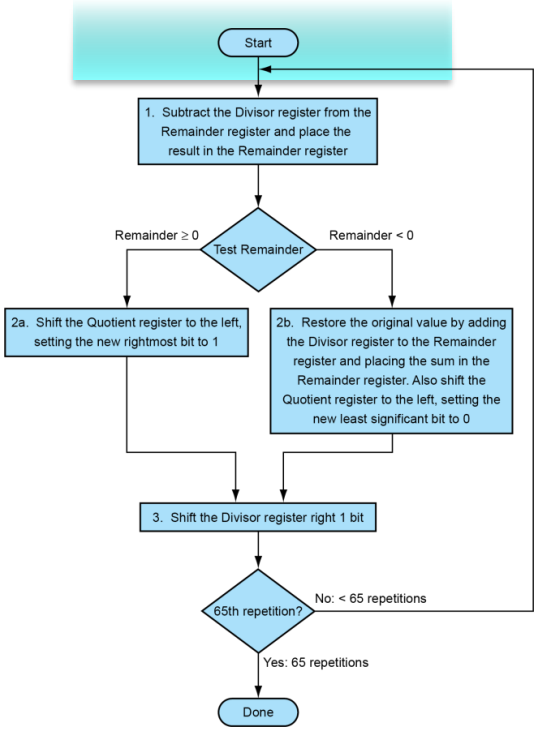


Image from David A. Patterson and John L. Hennessy. Computer Organization and Design. RISC-V Edition

#	Step	Quotient	Divisor	Remainder
0	Initial value	0000	0100 0000	0000 1101
1				
2				
3				

ALU algorithms (XXXVI)
Division:with Remainder restoration (VIII)

Get A / B if A = 1101 y B = 0100

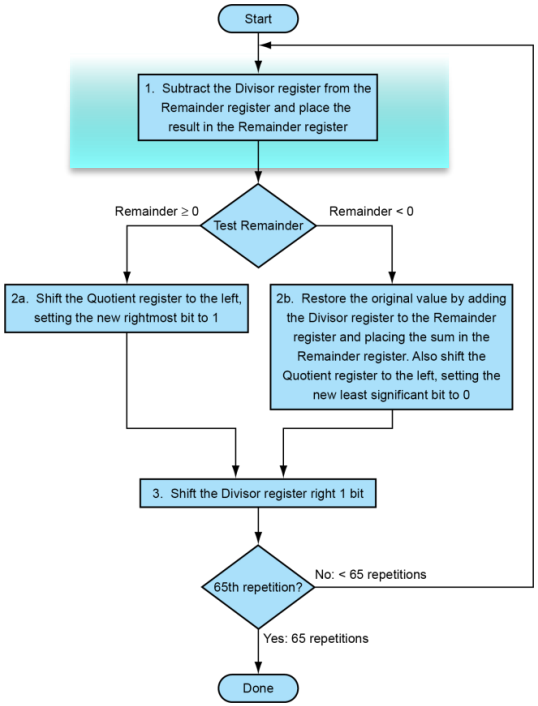


Image from David A. Patterson and John L. Hennessy. Computer Organization and Design. RISC-V Edition

#	Step	Quotient	Divisor	Remainder
0	Initial value	0000	0100 0000	0000 1101
1	R ← R - D	0000	0100 0000	1100 1101
2				
3				

ALU algorithms (XXXVII)
Division:with Remainder restoration (IX)

Get A / B if A = 1101 y B = 0100

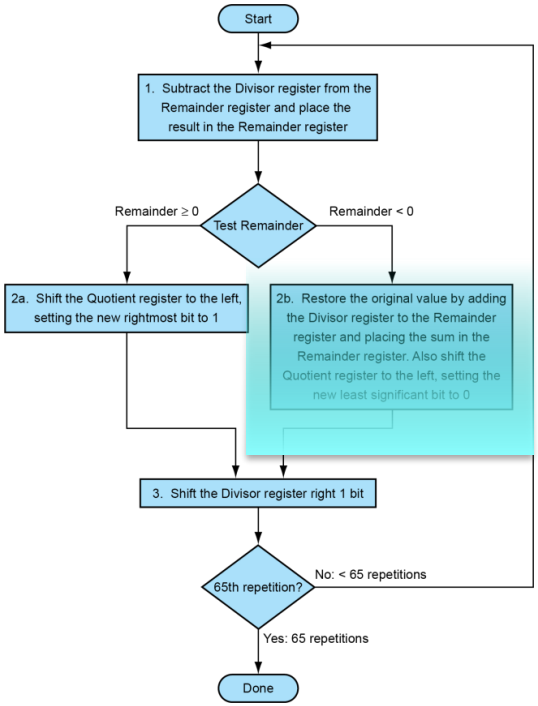


Image from David A. Patterson and John L. Hennessy. Computer Organization and Design. RISC-V Edition

#	Step	Quotient	Divisor	Remainder
0	Initial value	0000	0100 0000	0000 1101
1	R ← R - D	0000	0100 0000	1100 1101
	Neg → Rest	**0000**	**0100 0000**	**0000 1101**
2				
3				

ALU algorithms (XXXVIII)
Division:with Remainder restoration (X)

Get A / B if A = 1101 y B = 0100

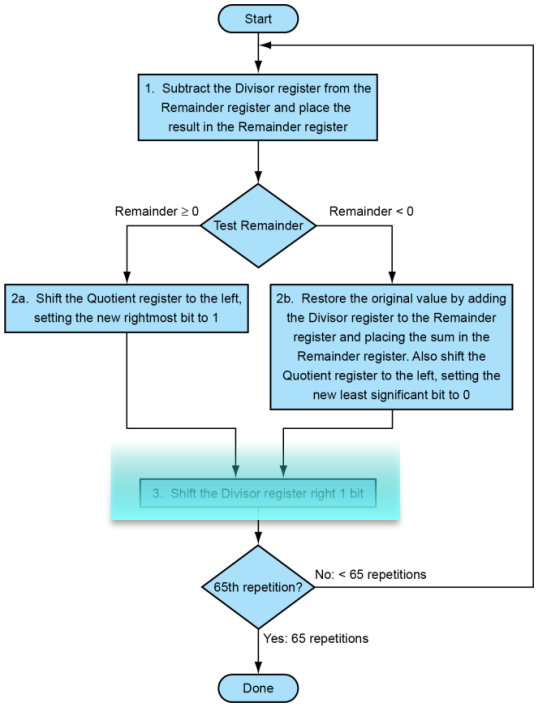


Image from David A. Patterson and John L. Hennessy. Computer Organization and Design. RISC-V Edition

#	Step	Quotient	Divisor	Remainder
0	Initial value	0000	0100 0000	0000 1101
1	R ← R - D	0000	0100 0000	1100 1101
	Neg → Rest	0000	0100 0000	0000 1101
	Shift Divisor	**0000**	**0010 0000**	**0000 1101**
2				
3				

ALU algorithms (XXXIX)
Division:with Remainder restoration (XI)

Get A / B if A = 1101 y B = 0100

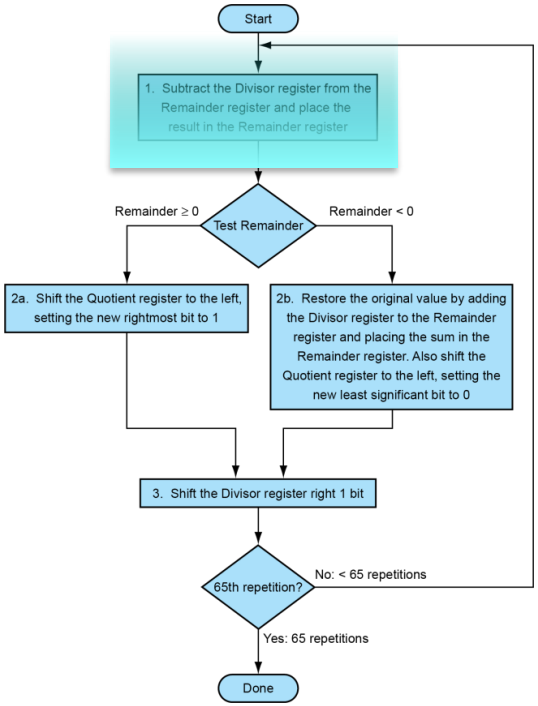


Image from David A. Patterson and John L. Hennessy. Computer Organization and Design. RISC-V Edition

#	Step	Quotient	Divisor	Remainder
0	Initial value	0000	0100 0000	0000 1101
1	R ← R - D	0000	0100 0000	1100 1101
	Neg → Rest	0000	0100 0000	0000 1101
	Shift Divisor	0000	0010 0000	0000 1101
2	**R ← R - D**	**0000**	**0010 0000**	**1110 1101**
3				

ALU algorithms (XL)
Division:with Remainder restoration (XII)

Get A / B if A = 1101 y B = 0100

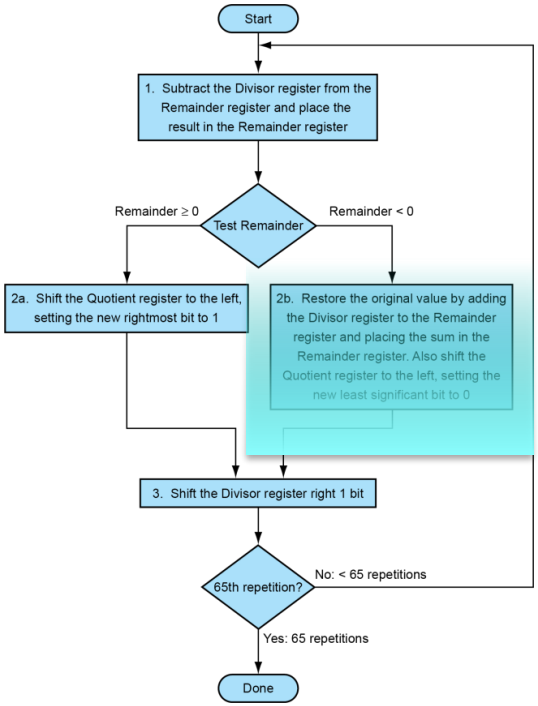


Image from David A. Patterson and John L. Hennessy. Computer Organization and Design. RISC-V Edition

#	Step	Quotient	Divisor	Remainder
0	Initial value	0000	0100 0000	0000 1101
1	R ← R - D	0000	0100 0000	1100 1101
	Neg → Rest	0000	0100 0000	0000 1101
	Shift Divisor	0000	0010 0000	0000 1101
2	R ← R - D	0000	0010 0000	1110 1101
	Neg → Rest	**0000**	**0010 0000**	**0000 1101**
3				

ALU algorithms (XLI)
Division:with Remainder restoration (XIII)

Get A / B if A = 1101 y B = 0100

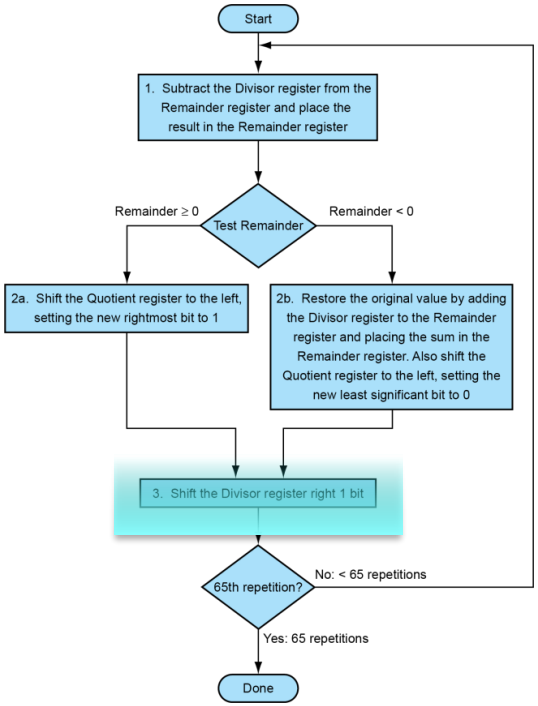


Image from David A. Patterson and John L. Hennessy. Computer Organization and Design. RISC-V Edition

#	Step	Quotient	Divisor	Remainder
0	Initial value	0000	0100 0000	0000 1101
1	R ← R - D	0000	0100 0000	1100 1101
	Neg → Rest	0000	0100 0000	0000 1101
	Shift Divisor	0000	0010 0000	0000 1101
2	R ← R - D	0000	0010 0000	1110 1101
	Neg → Rest	0000	0010 0000	0000 1101
	Shift Divisor	**0000**	**0001 0000**	**0000 1101**
3				

ALU algorithms (XLII)
Division:with Remainder restoration (XIV)

Get A / B if A = 1101 y B = 0100

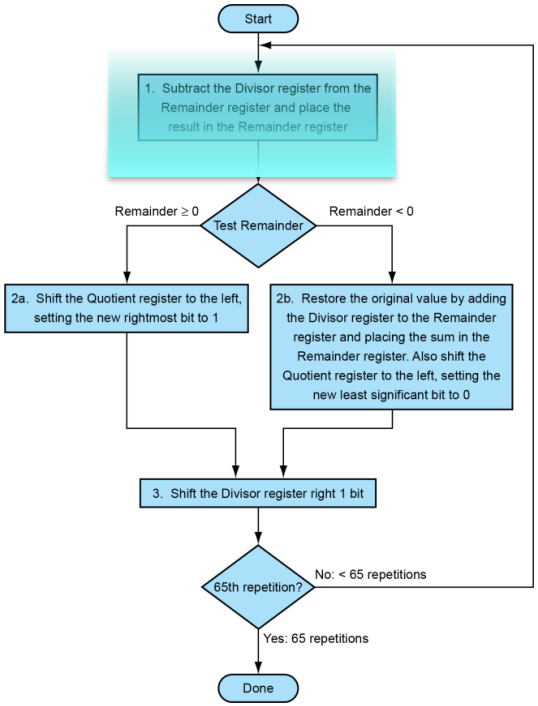


Image from David A. Patterson and John L. Hennessy. Computer Organization and Design. RISC-V Edition

#	Step	Quotient	Divisor	Remainder
0	Initial value	0000	0100 0000	0000 1101
1	R ← R - D	0000	0100 0000	1100 1101
	Neg → Rest	0000	0100 0000	0000 1101
	Shift Divisor	0000	0010 0000	0000 1101
2	R ← R - D	0000	0010 0000	1110 1101
	Neg → Rest	0000	0010 0000	0000 1101
	Shift Divisor	0000	0001 0000	0000 1101
3	**R ← R - D**	**0000**	**0001 0000**	**1111 1101**

ALU algorithms (XLIII)
Division:with Remainder restoration (XV)

Get A / B if A = 1101 y B = 0100

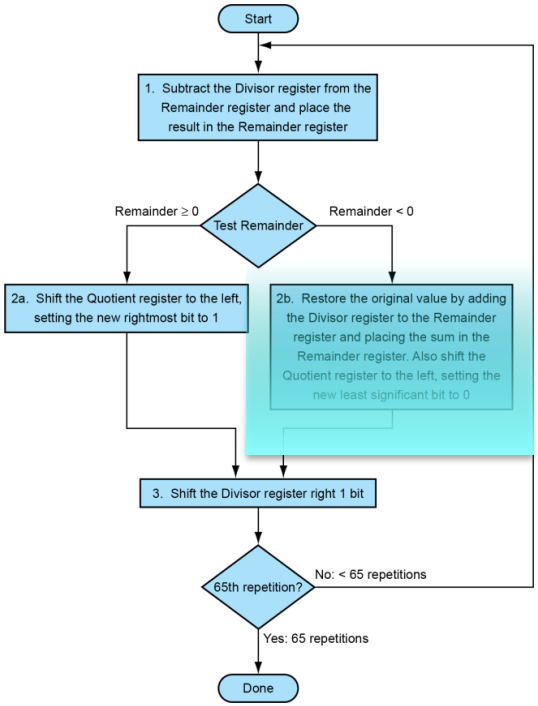


Image from David A. Patterson and John L. Hennessy. Computer Organization and Design. RISC-V Edition

#	Step	Quotient	Divisor	Remainder
0	Initial value	0000	0100 0000	0000 1101
1	R ← R - D	0000	0100 0000	1100 1101
	Neg → Rest	0000	0100 0000	0000 1101
	Shift Divisor	0000	0010 0000	0000 1101
2	R ← R - D	0000	0010 0000	1110 1101
	Neg → Rest	0000	0010 0000	0000 1101
	Shift Divisor	0000	0001 0000	0000 1101
3	R ← R - D	0000	0001 0000	1111 1101
	Neg → Rest	**0000**	**0001 0000**	**0000 1101**

ALU algorithms (XLIV)
Division:with Remainder restoration (XVI)

Get A / B if A = 1101 y B = 0100

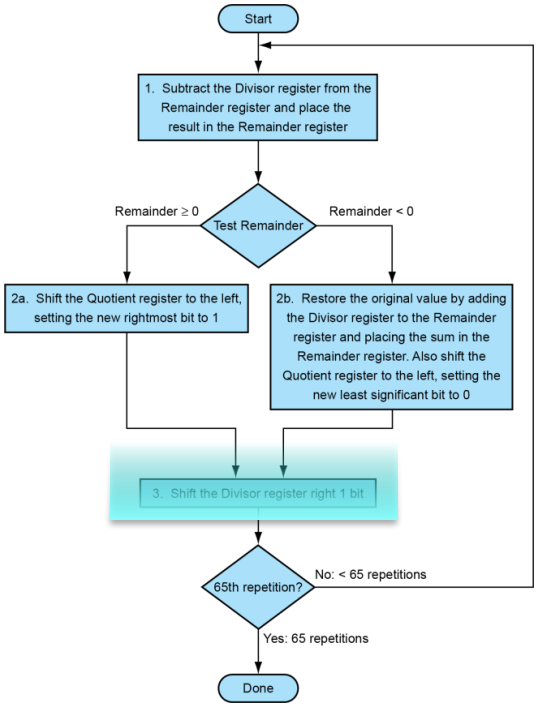


Image from David A. Patterson and John L. Hennessy. Computer Organization and Design. RISC-V Edition

#	Step	Quotient	Divisor	Remainder
0	Initial value	0000	0100 0000	0000 1101
1	R ← R - D	0000	0100 0000	1100 1101
	Neg → Rest	0000	0100 0000	0000 1101
	Shift Divisor	0000	0010 0000	0000 1101
2	R ← R - D	0000	0010 0000	1110 1101
	Neg → Rest	0000	0010 0000	0000 1101
	Shift Divisor	0000	0001 0000	0000 1101
3	R ← R - D	0000	0001 0000	1111 1101
	Neg → Rest	0000	0001 0000	0000 1101
	Shift Divisor	**0000**	**0000 1000**	**0000 1101**

ALU algorithms (XLV)
Division:with Remainder restoration (XVII)

Get A / B if A = 1101 y B = 0100

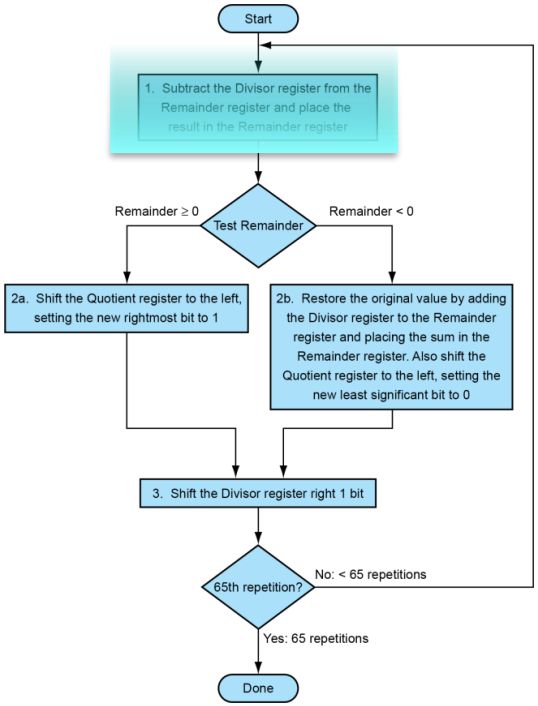


Image from David A. Patterson and John L. Hennessy. Computer Organization and Design. RISC-V Edition

#	Step	Quotient	Divisor	Remainder
	CONTINUATION			
4	R ← R - D	0000	0000 1000	0000 0101
5				

ALU algorithms (XLVI)
Division:with Remainder restoration (XVIII)

Get A / B if A = 1101 y B = 0100

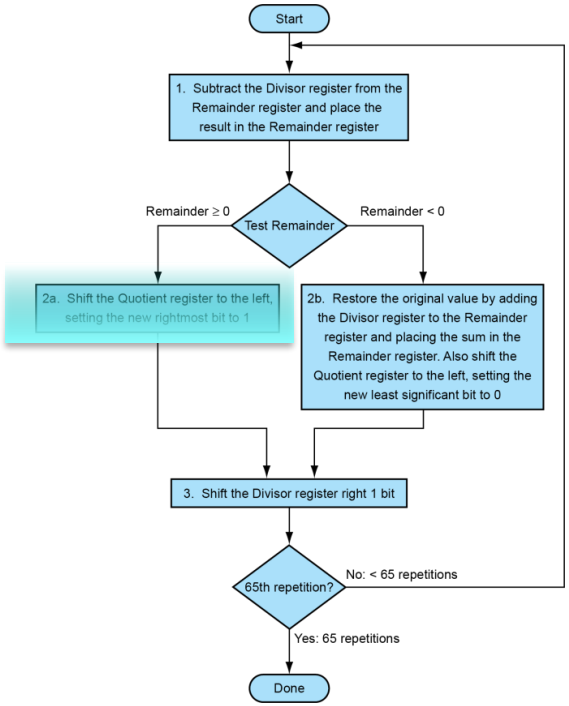


Image from David A. Patterson and John L. Hennessy. Computer Organization and Design. RISC-V Edition

#	Step	Quotient	Divisor	Remainder
	CONTINUATION			
4	R ← R - D	0000	0000 1000	0000 0101
	Pos → Q=1	0001	0000 1000	0000 0101
5				

ALU algorithms (XLVII)
Division:with Remainder restoration (XIX)

Get A / B if A = 1101 y B = 0100

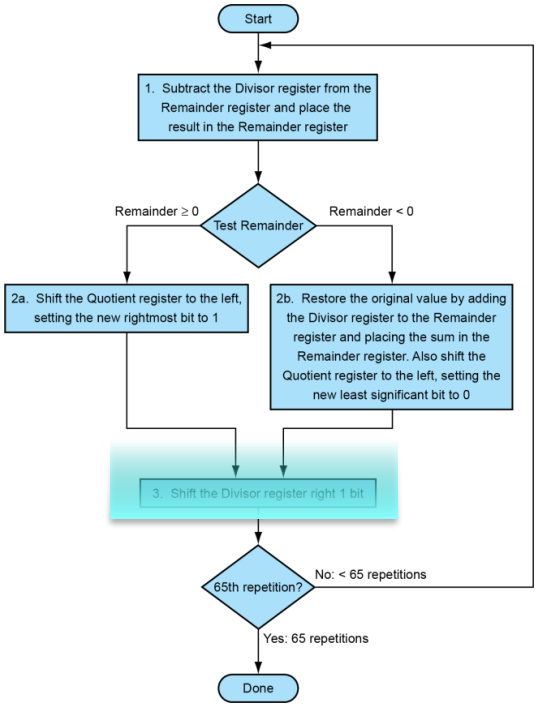


Image from David A. Patterson and John L. Hennessy. Computer Organization and Design. RISC-V Edition

#	Step	Quotient	Divisor	Remainder
	CONTINUATION			
4	R ← R - D	0000	0000 1000	0000 0101
	Pos → Q=1	0001	0000 1000	0000 0101
	Shift Divisor	**0001**	**0000 0100**	**0000 0101**
5				

ALU algorithms (XLVIII)
Division:with Remainder restoration (XX)

Get A / B if A = 1101 y B = 0100

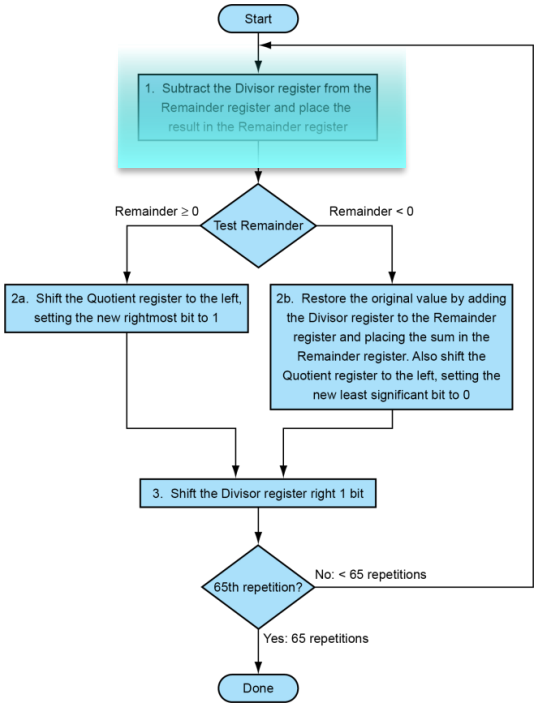


Image from David A. Patterson and John L. Hennessy. Computer Organization and Design. RISC-V Edition

#	Step	Quotient	Divisor	Remainder
	CONTINUATION			
4	R ← R - D	0000	0000 1000	0000 0101
	Pos → Q=1	0001	0000 1000	0000 0101
	Shift Divisor	0001	0000 0100	0000 0101
5	R ← R - D	0001	0000 0100	0000 0001

ALU algorithms (XLIX)
Division:with Remainder restoration (XXI)

Get A / B if A = 1101 y B = 0100

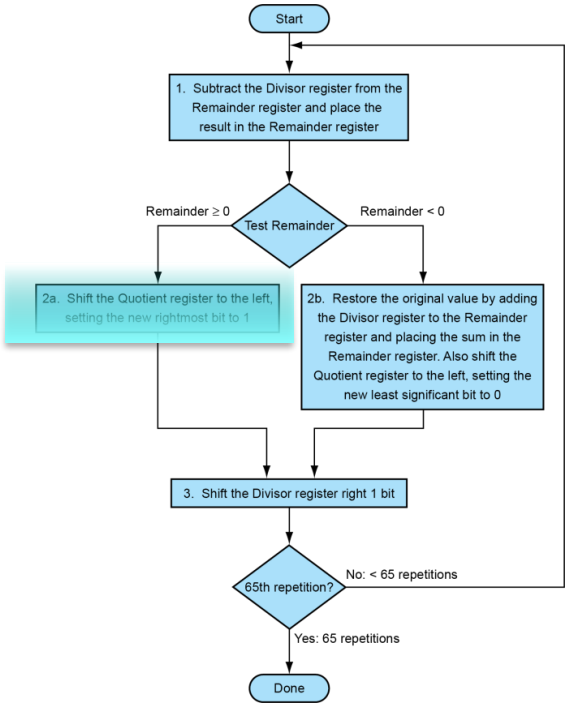


Image from David A. Patterson and John L. Hennessy. Computer Organization and Design. RISC-V Edition

#	Step	Quotient	Divisor	Remainder
	CONTINUATION			
4	R ← R - D	0000	0000 1000	0000 0101
	Pos → Q=1	0001	0000 1000	0000 0101
	Shift Divisor	0001	0000 0100	0000 0101
5	R ← R - D	0001	0000 0100	0000 0001
	Pos → Q=1	**0011**	**0000 0100**	**0000 0001**

ALU algorithms (L)
Division:with Remainder restoration (&XXII)

Get A / B if A = 1101 y B = 0100

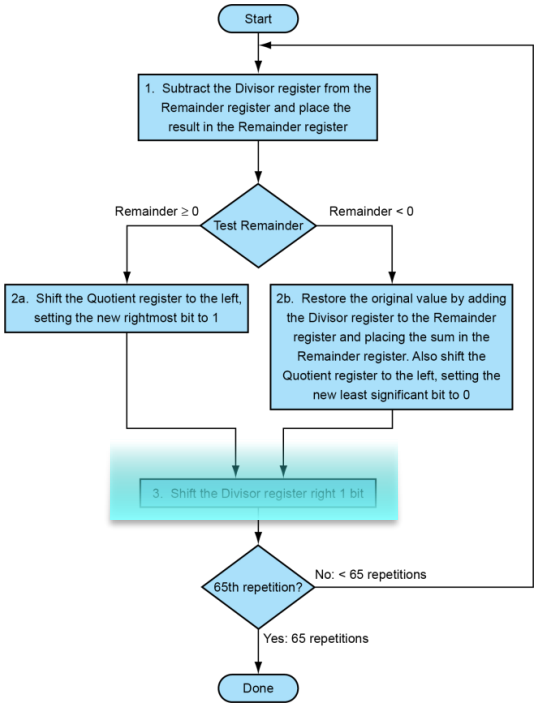


Image from David A. Patterson and John L. Hennessy. Computer Organization and Design. RISC-V Edition

#	Step	Quotient	Divisor	Remainder
	CONTINUATION			
4	R ← R - D	0000	0000 1000	0000 0101
	Pos → Q=1	0001	0000 1000	0000 0101
	Shift Divisor	0001	0000 0100	0000 0101
5	R ← R - D	0001	0000 0100	0000 0001
	Pos → Q=1	0011	0000 0100	0000 0001
	Shift Divisor	**0011**	**0000 0010**	**0000 0001**

Result A / B:

- **Quotient: 0011 (3)**
- **Remainder: 0001 (1)**

ALU algorithms (LI)
Floating Point Multiplicacion and Division

- **Floating point multiplication and division**
  - Use fixed point algorithms
  - Algorithms are applied to mantisass
  - Exponent are added (multiplication) or subtract (division)
  - Result of the operation must be normalized and rounded
  - It’s possible to employ guard digits to improve the accurate of the result

ALU algorithms (LII)
Speeding up Multiplicacion and Division (I)

Speeding up Multiplications

- Faster multiplications are possible by essentially providing one 64-bit adder for each bit of the multiplier: one input is the multiplicand ANDed with a multiplier bit, and the other is the output of a prior adder.
- A straightforward approach would be to connect the outputs of adders on the right to the inputs of adders on the left, making a stack of adders 64 high. An alternative way to organize these 64 additions is in a parallel tree
- Rather than use a single 64-bit adder 63 times, this hardware “unrolls the loop” to use 63 adders and then organizes them to minimize delay

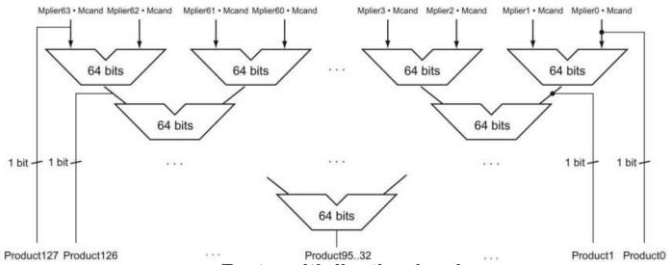


Image from David A. Patterson and John L. Hennessy. Computer Organization and Design. RISC-V Edition

ALU algorithms (and LIII)
Speeding up Multiplicacion and Division (and II)

Speeding up Division.

- We used many adders to speed up multiply, but we cannot do the same trick for divide. The reason is that we need to know the sign of the difference before we can perform the next step of the algorithm, whereas with multiply we could calculate the 64 partial products immediately

- The **SRT** division technique tries to predict several quotient bits per step. It relies on subsequent steps to correct wrong predictions. A typical value today is 4 bits.

- The key is guessing the value to subtract. With binary division, there is only a single choice. These algorithms use 6 bits from the remainder and 4 bits from the divisor to index a table that determines the guess for each step

ALU and Von Neumann's Architecure (I) The Flag Register.

- There are flip-flops related to the result of the arithmetic and logic operation performed in the computer. All of them are collected in one register called the Flags Register
- The aim of the flag register is to show some important characteristics of the operation results.
- Most frequent flags are:
  - Zero, Signed, Overflow and Carry
  - Parity, Auxiliar Carry, Trap, Interrupt Enable, …
- Exception conditions. Some of them are based on the content of the flags register. E.g. overflow, parity error,…

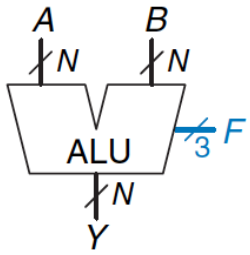
ALU and Von Neumann's Architecure (II) Overflow conditions.

- An overflow happens when we have no room space to store the full result.

- The overflow condition depends on the representation system:
  - **Unsigned numbers (binary)**: $Overflow = c_{n-1} \oplus \bar{S} / R$
  - **Sign-magnitude**: $Overflow = c_{n-1} \oplus \bar{S} / R$
  - **Complement to 1**: $Overflow = c_{n-1} \oplus c_{n-2}$
  - **Complement to 2**: $Overflow = c_{n-1} \oplus c_{n-2}$

ALU circuits example (I)

- An Arithmetic/Logical Unit (ALU) combines a variety of mathematical and logical operations into a single unit as we have seen.
- For instance, a typical ALU might perform addition, subtraction, different shifts, magnitude comparison, AND, and OR operations.



$F_{2:0}$	Function
000	A AND B
001	A OR B
010	A + B
011	not used
100	A AND $\overline{B}$
101	A OR $\overline{B}$
110	A – B
111	SLT

Images from David A. Harris and Sarah Harris. Digital Design and Computer Architecture, RISC-V Edition

ALU circuits example (II)

- The ALU contains an N-bit adder and N two-input AND and OR gates.
- The ALU also contains inverters and a multiplexer to invert input B when the F2 control signal is set.
- The 4 to 1 multiplexer chooses the desired function based on the F1:0 control signals
- SLT is done by computing S = A − B. If S is negative A is less than B.
- The zero extend unit produces an N-bit output by concatenating its 1-bit input with 0's in the most significant bits.

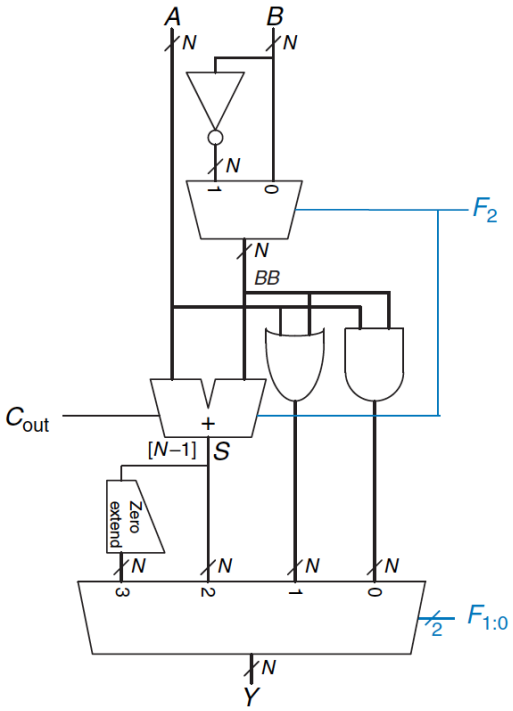
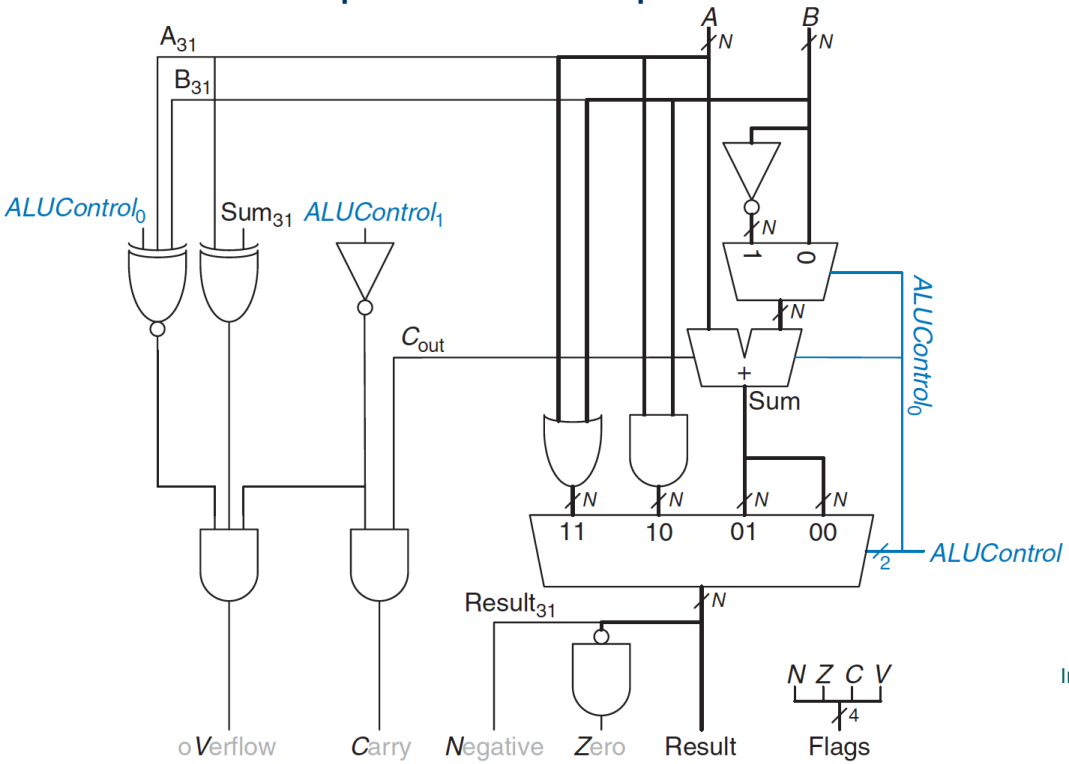


Image from David A. Harris and Sarah Harris. Digital Design and Computer Architecture, RISC-V Edition

ALU circuits example (and III)

- ALUs produce extra outputs, called flags, that indicate information about the ALU output. For example, overflow, zero, carry …



$ALUControl_{1:0}$	Function
00	Add
01	Subtract
10	AND
11	OR

Images from David A. Harris and Sarah Harris. Digital Design and Computer Architecture, RISC-V Edition

References

- “Organización y Diseño de Computadores. Edición RISC-V”, David A. Patterson, John Hennessy. Morgan-Kaufman 2018. Capítulo 3.
- “Diseño digital y arquitectura informática, edición RISC-V, primera edición”, Sarah Harris y David Harris, Morgan Kaufmann 2021. Capítulo 5.

.

- Estructura y diseño de computadoras. David A. Patterson y John L. Hennessy. Reverté, 2000. Capítulo 4
- “Organización y Arquitectura de Computadores”. William Stallings. Diapositivas de la octava edición. 2010. Capítulo 8