Unit 7
Computer Arithmetic

Arithmetic instructions manipulate data to produce solution for computational problems. The 4 basic arithmetic operations are addition, subtraction, multiplication and division. From these 4, it is possible to formulate other scientific problems by means of numerical analysis methods. Here, we'll discuss these 4 operations only on fixed-point binary data (there are other types too, viz. floating point binary data, binary-coded decimal data) and hence the unit named.

Addition and Subtraction

There are 3 ways of representing negative fixe-point binary numbers: signed magnitude, signed 1's complement or signed 2's complement. Singed 2's complemented form used most but occasionally we deal with signed magnitude representation.

Addition and Subtraction with signed-magnitude data

Everyday arithmetic calculations with paper and pencil for signed binary numbers are straight forward and are helpful on deriving hardware algorithm. When two signed numbers A and B are added are added are subtracted, we find 8 different conditions to consider as described in following table:

Operation	Add Magnitudes	Subtract Magnitudes		
		When $A > B$	When $A < B$	When $A = B$
$(+A) + (+B)$	$+(A + B)$			
$(+A) + (-B)$		$+(A - B)$	$-(B - A)$	$+(A - B)$
$(-A) + (+B)$		$-(A - B)$	$+(B - A)$	$+(A - B)$
$(-A) + (-B)$	$-(A + B)$			
$(+A) - (+B)$		$+(A - B)$	$-(B - A)$	$+(A - B)$
$(+A) - (-B)$	$+(A + B)$			
$(-A) - (+B)$	$-(A + B)$			
$(-A) - (-B)$		$-(A - B)$	$+(B - A)$	$+(A - B)$

Table: addition and subtraction of signed-magnitude numbers

Note: Brackets () for subtraction

Addition (subtraction) algorithm: when the signs of A and B are identical (different), add magnitudes and attach the sign of A to result. When the signs of A and b are different (identical), compare the magnitudes and subtract the smaller form larger.

Hardware Implementation

To implement the two arithmetic operations with hardware, we have to store numbers into two register A and B. let A_s and B_s be two flip-flops that holds corresponding signs. The result is transferred to A and A_s. A and A_s together form a accumulator.

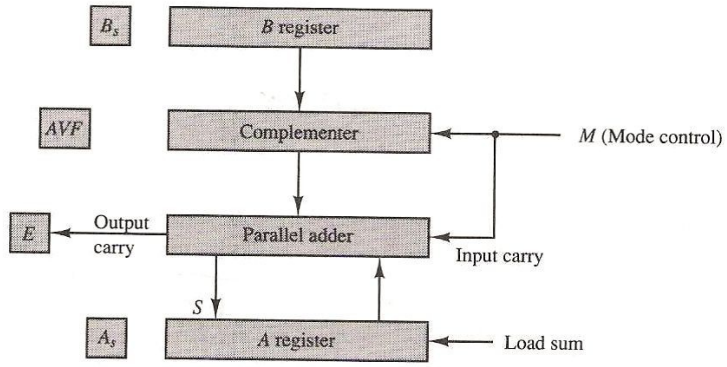


Fig: hardware for signed-magnitude addition and subtraction

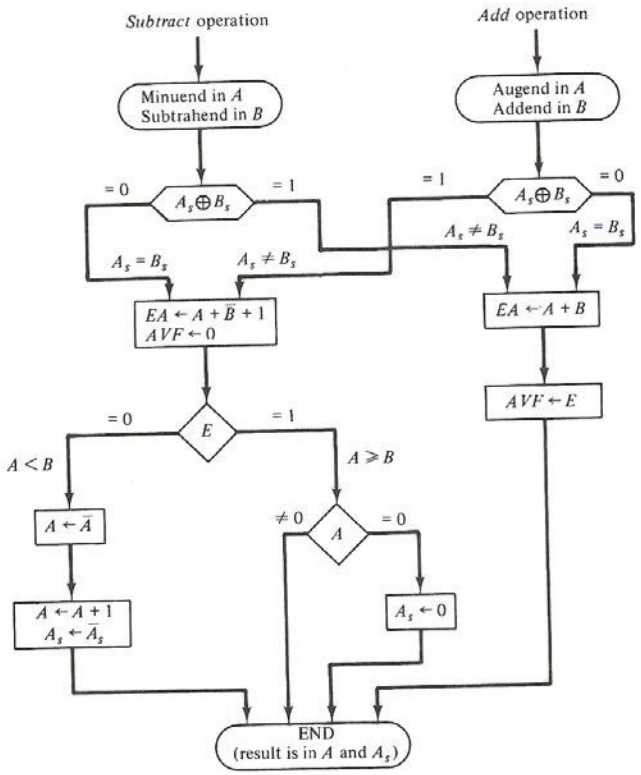
We need:

- Consists of two resisters A and B and sign flip-flops A_s and B_s.
- **A magnitude comparator**: to check if A>B, A<B or A=B.
- **A parallel adder**: to perform A+B
- **Two parallel subtractors**: for A-B and B-A
- The **sign** relationships are determined from an exclusive-OR gate with A_s and B_s as inputs.

Block Diagram Description: hardware above consists of registers A and B and sign flip-flops A_s and B_s. subtraction is done by adding A to the 2's complement of B. Output carry is transferred to flip-flop E, where it can be checked to determine the relative magnitude of two numbers. Add-overflow flip-flop AVF holds overflow bit when A and B are added. Addition of A and B is done through the parallel adder. The S output of adder is applied to A again. The complementer provides an output of B or B' depending on mode input M. Recalling unit 2, when M = 0, the output of B is transferred to the adder, the input carry is 0 and thus output of adder is A+B. when M=1, 1's complement of B is applied to the adder, input carry is 1 and output is S = A+B'+1 (i.e. A-B).

Hardware Algorithm

The flowchart for the H/W algorithm is given below:



→As and Bs are compared by an exclusive-OR gate. If output = 0, signs are identical, if 1 signs are different.

→For *add* operation identical signs dictate addition of magnitudes and for *subtraction*, different magnitudes dictate magnitudes be **added**. Magnitudes are added with a microoperation EA←A+B (EA is a resister that combines A and E). if E = 1, overflow occurs and is transferred to AVF.

→Two magnitudes are subtracted if signs are different for add operation and identical for subtract operation. Magnitudes are subtracted with a microoperation EA←A+B'+1. No overflow occurs if the numbers are subtracted so AVF is cleared to 0. E=1 indicates A>=B and number (this number is checked again for 0 to make positive 0 [A_s=0]) in A is correct result. E=0 indicates A<B, so we take 2's complement of A.

Fig: flowchart for add and subtract operations

Addition and Subtraction with signed 2's complement data

Guys, refer unit 1 once, addition and subtraction with signed 2's complement data are introduced there. Anyway, in signed 2's complement representation, the leftmost bit represents sign (0-positive and 1-negative). If sign bit is 1, entire number is represented in 2's complement form (+33=00100001 and -33=2's complement of 00100001 =11011111).

Addition: sign bits treated as other bits of the number. Carry out of the sign bit is discarded.

Subtraction: consists of first taking 2's complement of the subtrahend and then adding it to minuend. When two numbers of n-digits each are added and the sum occupies n+1 bits, overflow occurs which is detected by applying last two carries out of the addition to XOR gate. The overflow occurs when output of the gate is 1.

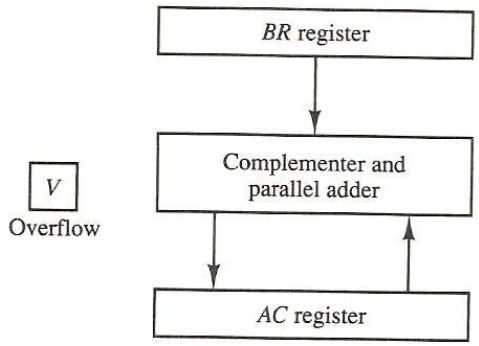


Fig: hardware for signed-2's complement addition and subtraction

→Register configuration is same as signed-magnitude representation except sign bits are not separated. The leftmost bits in AC and BR represent sign bits.

→Significant difference: sign bits are added are subtracted together with the other bits in complementer and parallel adder. The overflow flip-flop V is set to 1 if there is an overflow. Output carry in this case is discarded.

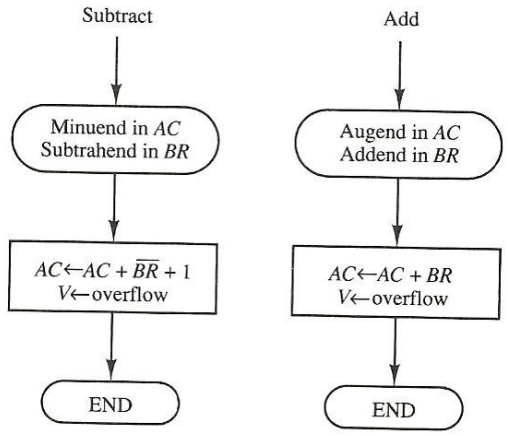


Fig: algorithm for addition & subtraction of numbers in signed-2's complement representation

Example: 33 + (-35)
AC = 33 = 00100001
BR = -35 = 2's complement of 35 = 11011101
AC + BR = 11111110 = -2 which is the result

Comparing this algorithm with its signed-magnitude counterpart, it is much easier to add and subtract numbers. For this reason most computers adopt this representation over the more familiar signed-magnitude.

Multiplication

Signed-magnitude representation

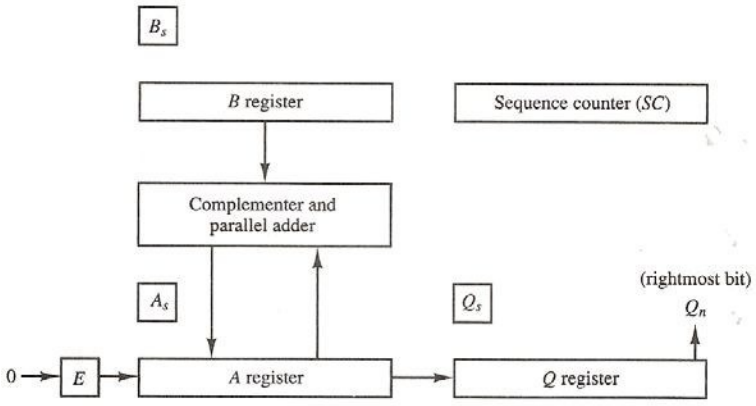
For this representation, multiplication is done by a process of successive shift and adds operations. As an example:

```
23      10111    Multiplicand
19    × 10011    Multiplier
        10111
       10111
      00000   +
     00000
    10111
437 110110101    Product
```

Process consists of looking successive bits of the multiplier, least significant bits first. If the multiplier bit is 1, the multiplicand is copied down; otherwise, zeros are copied down. Numbers copied down in successive lines are shifted one position Shifted left one position. Finally, numbers are added to form a product.

Hardware implementation for signed-magnitude data

It needs same hardware as that of addition and subtraction of signed-magnitude. In addition it needs two more registers Q and SC.

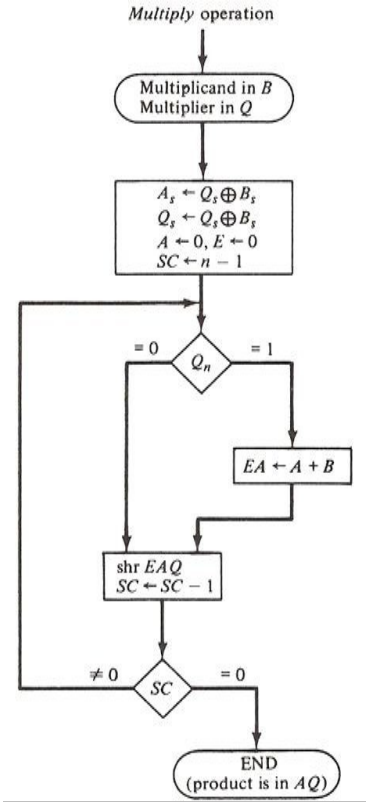


- ➔ Successively accumulate partial products and shift it right.
- ➔ Q ← multiplier and Qs ← sign.
- ➔ SC ← no. of bits in multiplier (magnitude only).
- ➔ SC is decremented after forming each partial product. When SC is 0, process halts and final product is formed.
- ➔ B ← multiplicand, B_s ← sign
- ➔ Sum of A and B forms a partial product

Fig: Hardware for multiply operation

Hardware Algorithm

Flowchart below shows a hardware multiply algorithm.



Example: B = 10111 (**Multiplicand**)
Q = 10011 (**Multiplier**)

Operation	E	A	Q	SC
Initial conf.	0	00000	10011	101
Iteration 1 (Q_n = 1)		00000		
EA← A+B		+ 10111		
PP1-->	0	10111		
shr EAQ, SC←SC-1	0	01011	11001	100
Iteration 2 (Q_n = 1)		01011		
EA← A+B		+ 10111		
PP2-->	1	00010	11001	
shr EAQ, SC←SC-1	0	10001	01100	011
Iteration 3 (Q_n = 0)				
shr EAQ, SC←SC-1	0	01000	10110	010
Iteration 4 (Q_n = 0)				
shr EAQ, SC←SC-1	0	00100	01011	001
Iteration 5 (Q_n = 1)		00100		
EA← A+B		+ 10111		
PP3-->	0	11011	01011	
shr EAQ, SC←SC-1	0	01101	10101	000
Final Product in AQ	0110110101			

Signed 2's complement representation

Booth multiplication Algorithm

Booth algorithm gives a procedure for multiplying binary integers in signed 2's complement notation.

Inspiration: String of 1's in the multiplier from bit weight 2^k to weight 2^m can be treated as $2^{k+1}-2^m$. As an example, binary number 001110 (+14) has string of 1's from 2^3 to 2^1 (k=3, m=1). So, this number can be represented as $2^{k+1} - 2m = 2^4 - 2^1 = 16 - 2 = 14$ (case is similar for -14 (110010) = $-2^4+2^2-2^1$). Thus, M * 14 = M * 2^4 – M * 2^1; product can be obtained by shifting multiplicand M four times left and subtracted M shifted left once.

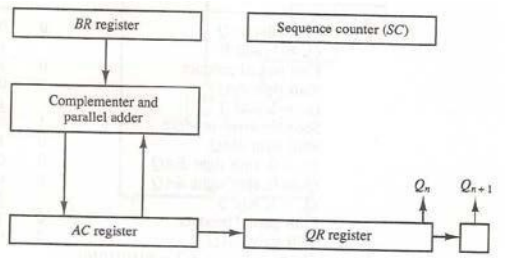
As in other multiplication schemes, Booth algorithm also requires examination of multiplier bits and shifting of the partial product. Prior to shifting multiplicand may be:

Subtracted <-- upon the encountering first least significant 1 in the string of 1's in the multiplier.

Added <-- upon encountering first 0 (left of it must be 1) in string of 0's in the multiplier.

Unchanged <-- when multiplier bit (Q_n) is identical to previous multiplier bit (Q_{n+1})

Hardware for Booth algorithm



- Here, sign bits are not separated.
- Registers A, B and Q are renamed to AC, BR and QR.
- Extra flip-flop Q_{n+1} appended to QR is needed to store almost lost right shifted bit of the multiplier (which along with current Q_n gives information about bit sequencing of multiplier, in fact no. of 1's gathered together).
- Pair Q_nQ_{n+1} inspect double bits of the multiplier.

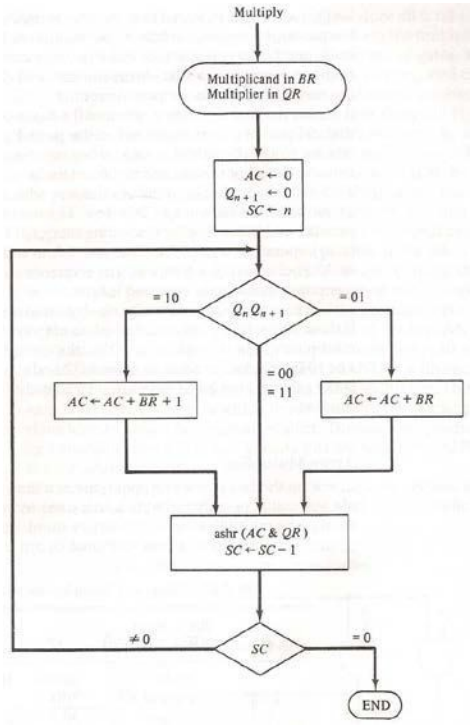
Hardware Booth algorithm

Numerical Example: **Booth algorithm**

BR = 10111 (**Multiplicand**)

QR = 10011 (**Multiplier**)

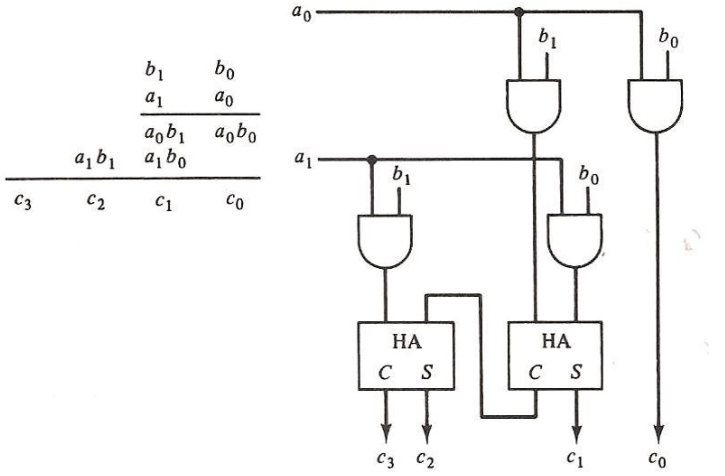
Q_nQ_{n+1}	$BR = 10111$ $\overline{BR} + 1 = 01001$	AC	QR	Q_{n+1}	SC
	Initial	00000	10011	0	101
1 0	Subtract BR	01001			
		01001			
	ashr	00100	11001	1	100
1 1	ashr	00010	01100	1	011
0 1	Add BR	10111			
		11001			
	ashr	11100	10110	0	010
0 0	ashr	11110	01011	0	001
1 0	Subtract BR	01001			
		00111			
	ashr	00011	10101	1	000



Array Multiplier

Checking the bits of the multiplier one at a time and forming partial products is a sequential operation requiring sequence of add and shift microoperations . The multiplication of two binary numbers can be done with one microoperation by using combinational circuit that forms product bits all at once. This is a fast way of multiplying two numbers since all it takes is the time to propagate through the gates that form the **multiplication array**.

Consider multiplication of two 2-bit numbers: Multiplicand = b_1b_0, Multiplier = a_1a_0, Product = $c_3c_2c_1c_0$



- Since multiplication of two bits is identical to AND operation and hence can be implemented with AND gate.
- In the diagram, partial products and formed and added by means of HA (half adders).

Fig: 2-bit by 2-bit array multiplier

A combinational circuit binary multiplier with more bits can be constructed in similar fashion. For j multiplier bits and k multiplicand bits, we need j*k AND gates and (j-1) k-bit adders to produce a product of j+k bits.

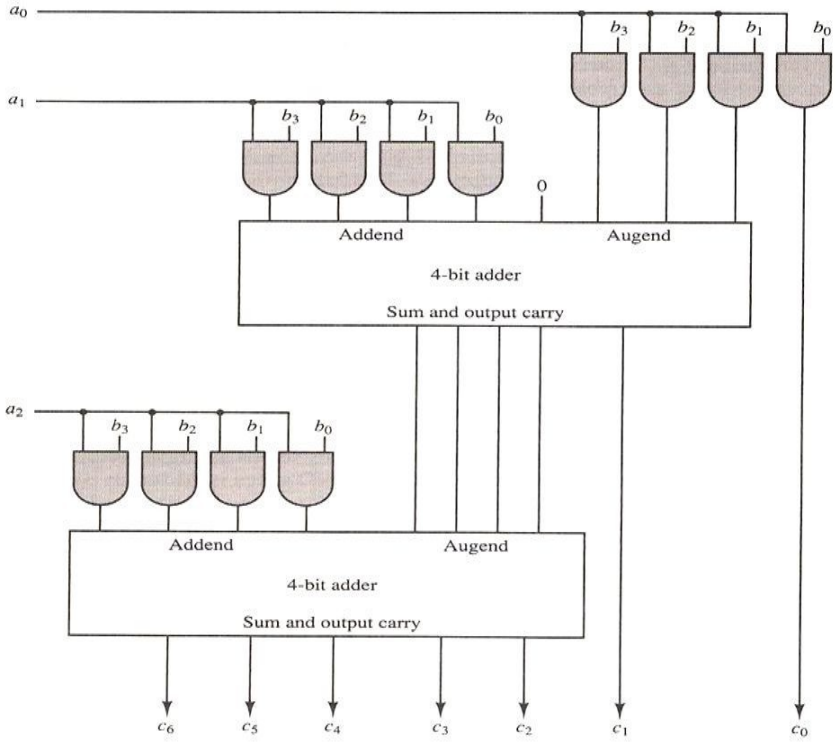


Fig: 4-bit by 3-bit array multiplier

Division Algorithms

Division of fixed-point binary numbers in signed-magnitude representation is done with successive compare, shift and subtract operations.

Example:

Divisor:		
$B = 10001$	11010	Quotient = Q
	0111000000	Dividend = A
	01110	5 bits of $A < B$, quotient has 5 bits
	011100	6 bits of $A \geqslant B$
	-10001	Shift right B and subtract; enter 1 in Q
	-010110	7 bits of remainder $\geqslant B$
	--10001	Shift right B and subtract; enter 1 in Q
	--001010	Remainder $< B$; enter 0 in Q; shift right B
	---010100	Remainder $\geqslant B$
	----10001	Shift right B and subtract; enter 1 in Q
	----000110	Remainder $< B$; enter 0 in Q
	-----00110	Final remainder

- Easier than decimal since quotient digits are 0 or 1.
- B ← divisor, A ← dividend, Q← Quotient
- Process consists of comparing a **partial remainder** with a divisor.

Hardware Implementation for Signed-Magnitude Data

While implementing division in digital system, we adopt slightly different approach. Instead of shifting divisor right, the partial remainder (or dividend) is shifted left. Hardware is similar to multiplication algorithm (not booth). Register EAQ is now shifted left with 0 inserted into Q_n (Obviously, previous value of E is lost). (I am not redrawing the diagram guys, it's all same as multiplication but EAQ is shifted left so change the direction of arrows at bottom).

Divide Overflow

- Division operation may result in a quotient with an overflow when working with finite size registers.
- Storing divisor in n-bit resister and dividend in 2 n-bit registers, then if quotient occupies n+1 bits, we say divide-overflow has occurred (since n+1 bit quotient can not be stored in standard n-bit Q-register and/or memory word).
- Talking about special case: **size** (dividend) = 2 * **size** (divisor). Divide-overflow condition will occur if *high-order half bits of the dividend* >= *divisor*. This condition is detected by DVF (**D**ivide-o**v**erflow **F**lip-flop).

Handling of overflow: its programmer's responsibility to detect DVF and take corrective measure. The best way is to use floating point data.

Hardware algorithm (Restoring algorithm)

Flowchart for hardware algorithm is shown below:

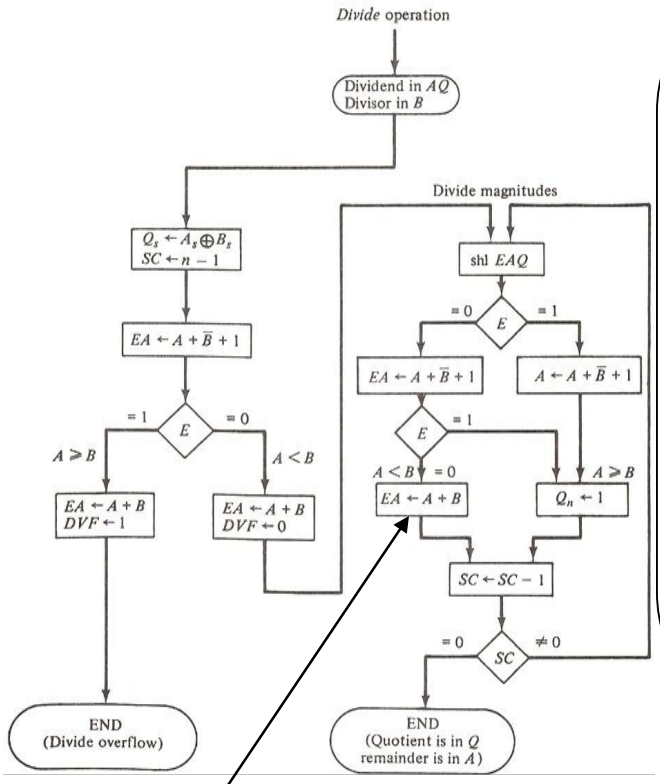


Fig: flow chart for divide operation

- B: Divisor, AQ: Dividend
- If A>=B (oh yes, magnitudes are compared subtracting one from another and testing E flip-flop), DVF is set and operation is terminated prematurely. If A<B, no overflow and dividend is restored by adding B to A (since B was subtracted previously to compare magnitudes).
- Division starts by left shifting AQ (dividend) with high order bit shifted to E. Then E=1, EA>B so B is subtracted from EA and Q_n is set to 1. If E=0, result of subtraction is stored in EA, again E is tested. E=1 signifies A>=B, thus Q_n is set to 1 and E=0 denotes A<B, so original number is **restored** by adding B to A and we leave 0 in Q_n.
- Process is repeated again with register A holding partial remainder. After n-1 times Q contains magnitude of Quotient and A contains remainder. Quotient sign in Q_s and remainder sign in A_s.

This is the restoring step. Different variant of division algorithm only have distinction at this step.

HEY! You may face Nonrestoring or comparison methods as long questions. Don't blame me for that since everything (hardware implementation and hardware algorithm) is same. Only difference is at this step.

!!!HEY: In each iteration, just after left - shifting EAQ, we test it for 0 or 1 and proceed accordingly which is not noted in example (Example is taken such that E is always 0 just after shifting).

Numerical Example: Binary division with digital hardware

Divisor $B = 10001$, $\overline{B} + 1 = 01111$

	E	A	Q	SC
Dividend:		01110	00000	5
shl EAQ	0	11100	00000	
add $\overline{B} + 1$		01111		
$E = 1$	1	01011		
Set $Q_n = 1$	1	01011	00001	4
shl EAQ	0	10110	00010	
Add $\overline{B} + 1$		01111		
$E = 1$	1	00101		
Set $Q_n = 1$	1	00101	00011	3
shl EAQ	0	01010	00110	
Add $\overline{B} + 1$		01111		
$E = 0$; leave $Q_n = 0$	0	11001	00110	
Add B		10001		
Restore remainder	1	01010		2
shl EAQ	0	10100	01100	
Add $\overline{B} + 1$		01111		
$E = 1$	1	00011		
Set $Q_n = 1$	1	00011	01101	1
shl EAQ	0	00110	11010	
Add $\overline{B} + 1$		01111		
$E = 0$; leave $Q_n = 0$	0	10101	11010	
Add B		10001		
Restore remainder	1	00110	11010	0
Neglect E				
Remainder in A:		00110		
Quotient in Q:			11010	

Other division algorithms

Method described above is **restoring method** in which *partial remainder* is restored by adding the divisor to the negative result. Other methods:

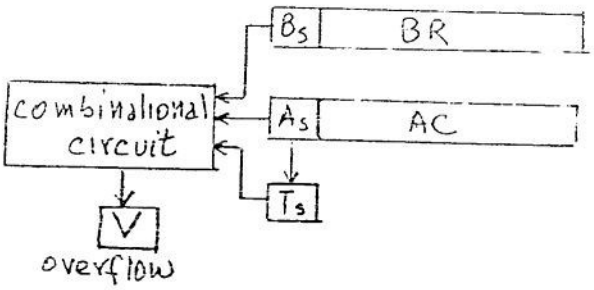
Comparison method: A and B are compared prior to subtraction. Then if A>=B, B is subtracted form A. if A<B nothing is done. The partial remainder is then shifted left and numbers are compared again. Comparison inspects end-carry out of the parallel adder before transferring to E.

Nonrestoring method: In contrast to restoring method, when A-B is negative, B is not added to restore A but instead, negative difference is shifted left and then B is added. How is it possible? Let's argue:

- In flowchart for restoring method, when A<B, we restore A by operation A-B+B. Next tine in a loop, this number is shifted left (multiplied by 2) and B subtracted again, which gives: 2(A - B + B) – B = **2A-B**.
- In Nonrestoring method, we leave A-B as it is. Next time around the loop, the number is shifted left and B is added: 2(A-B)+B = **2A-B** (same as above).

Exercises: textbook ch 10 → 10.5, 10.9, 10.10, 10.15

10.5 solution



Transfer Augend sign into T_S.
Then add: $AC \leftarrow AC + BR$
A_S will have sign of sum.

Boolean function for circuit:

$V = T_S' B_S' A_S + T_S B_S A_S'$

Truth Table for combin. circuit

T_S	B_S	A_S	V	
0	0	0	0	
0	0	1	1	change of sign
0	1	0	0	quantities
0	1	1	0	subtracted
1	0	0	0	
1	0	1	0	
1	1	0	1	change of sign
1	1	1	0	

10.9 and 10.10 solution: do it yourself

10.15 solution: