Unit II

Arithmetic Operations

ALU - Addition and subtraction – Multiplication – Division – Floating Point operations – Subword parallelism.

2.1 Arithmetic and logic unit (ALU)

- An arithmetic logic unit (ALU) represents the fundamental building block of the central processing unit of a computer.
- An ALU is a digital circuit used to perform arithmetic and logic operations.

What is an ALU?

- An arithmetic logic unit (ALU) is a digital circuit used to perform arithmetic and logic operations. It represents the fundamental building block of the central processing unit (CPU) of a computer.
- Modern CPUs contain very powerful and complex ALUs. In addition to ALUs, modern CPUs contain a control unit (CU).
- Most of the operations of a CPU are performed by one or more ALUs, which load data from input registers. A register is a small amount of storage available as part of a CPU.
- The control unit tells the ALU what operation to perform on that data and the ALU stores the result in an output register. The control unit moves the data between these registers, the ALU, and memory.

Symbol of ALU:

- Result lines provide result of the chosen function applied to values of A and B. Since this ALU operates on 32-bit operands, it is called 32-bit ALU.
- Zero output indicates if all Result lines have value 0
- Overflow indicates integer overflow of add and subtract functions; for unsigned integers, this overflow indicator does not provide any useful information
- Carry out indicates carry out and unsigned integer overflow.

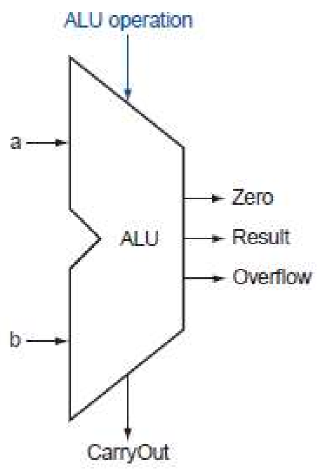


Figure : The symbol used to represent an ALU

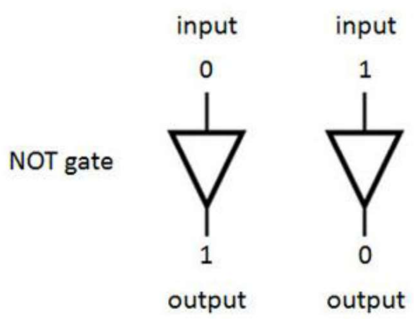
ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set on less than
1100	NOR

Figure: The values of the three ALU control lines Bnegate and Operation and the corresponding ALU operations.

How an ALU Works

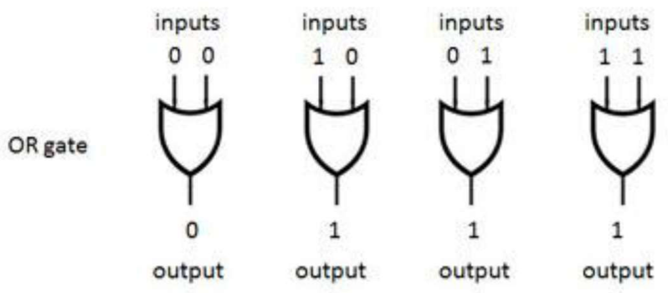
- An ALU performs basic arithmetic and logic operations. Examples of arithmetic operations are addition, subtraction, multiplication, and division. Examples of logic operations are comparisons of values such as NOT, AND, and OR.
- All information in a computer is stored and manipulated in the form of binary numbers, i.e. 0 and 1. Transistor switches are used to manipulate binary numbers since there are only two possible states of a switch: open or closed. An open transistor, through which there is no current, represents a 0. A closed transistor, through which there is a current, represents a 1.

- Operations can be accomplished by connecting multiple transistors. One transistor can be used to control a second one in effect, turning the transistor switch on or off depending on the state of the second transistor. This is referred to as a gate because the arrangement can be used to allow or stop a current.
- The simplest type of operation is a NOT gate. This uses only a single transistor. It uses a single input and produces a single output, which is always the opposite of the input. This figure shows the logic of the NOT gate.



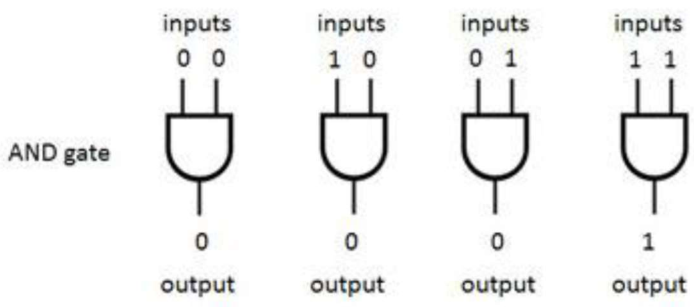
How a NOT gate processes binary data

- Other gates consist of multiple transistors and use two inputs.
- The OR gate results in a 1 if either the first or the second input is a 1.
- The OR gate only results in a 0 if both inputs are 0.
- This figure shows the logic of the OR gate.



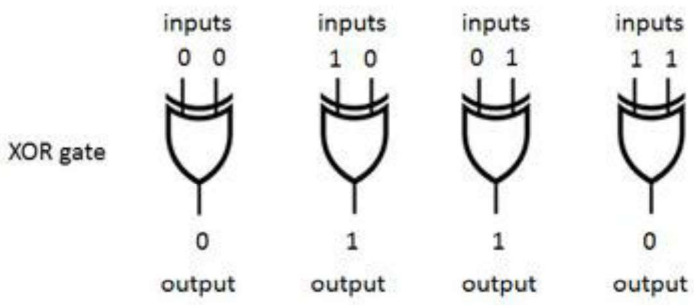
How an OR gate processes binary data

- The AND gate results in a 1 only if both the first and second input are 1s.
- This figure shows the logic of the AND gate.



How an AND gate processes binary data

- The XOR gate, also pronounced X-OR gate, results in a 0 if both the inputs are 0 or if both are 1. Otherwise, the result is a 1. This figure shows the logic of the XOR gate.



How an XOR gate processes binary data.

- The arithmetic and logic unit (ALU) performs necessary addition, subtraction, multiplication, division, shifting, and logical operations.
- It requires one or two operands to operate and produces a result.
- The complexity of an ALU is determined by the way in which its arithmetic instructions are realized.
- Simple ALUs perform fixed point addition and subtraction, word based logic operations; can be realized by combinational circuits.The arithmetic logic unit (ALU) is the brain of the computer, the device that performs the arithmetic operations like addition and subtraction or logical operations like AND and OR. The MIPS word is 32 bits wide, so we need a 32-bit-wide ALU.
- Let's assume that we will connect 32 1-bit ALUs to create the desired ALU. Therefore start by constructing a 1-bit ALU.

2.1.1 A 1-Bit ALU

- The 1-bit logical unit for AND and OR looks like below figure.
- The multiplexer on the right then selects a AND b or a OR b, depending on whether the value of Operation is 0 or 1.
- The line that controls the multiplexor is shown in color to distinguish it from the lines containing data.
- Notice that we have renamed the control and output lines of the multiplexer to give them names that reflect the function of the ALU. The next function to include is addition.
- An adder must have two inputs for the operands and a single-bit output for the sum. There must be a second output to pass on the carry, called CarryOut.
- Since the CarryOut from the neighbor adder must be included as an input, we need a third input. This input is called CarryIn.

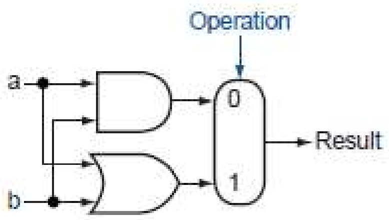


Figure :The 1-bit logical unit for AND and OR.

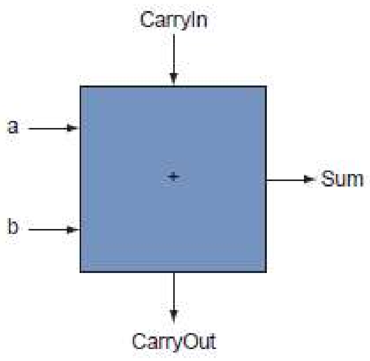


Figure :A 1-bit adder.

- This adder is called a full adder; it is also called a (3,2) adder because it has 3 inputs and 2 outputs.

- An adder with only the a and b inputs is called a (2,2) adder or half adder.

Inputs			Outputs		Comments
a	b	CarryIn	CarryOut	Sum	
0	0	0	0	0	$0 + 0 + 0 = 00_{two}$
0	0	1	0	1	$0 + 0 + 1 = 01_{two}$
0	1	0	0	1	$0 + 1 + 0 = 01_{two}$
0	1	1	1	0	$0 + 1 + 1 = 10_{two}$
1	0	0	0	1	$1 + 0 + 0 = 01_{two}$
1	0	1	1	0	$1 + 0 + 1 = 10_{two}$
1	1	0	1	0	$1 + 1 + 0 = 10_{two}$
1	1	1	1	1	$1 + 1 + 1 = 11_{two}$

Figure : Input and output specification for a 1-bit adder.

- We can express the output functions CarryOut and Sum as logical equations, and these equations can in turn be implemented with logic gates.
- We can turn this truth table into a logical equation:

CarryOut = (b · CarryIn) + (a · CarryIn) + (a · b) + (a · b · CarryIn)

- If a · b · CarryIn is true, then all of the other three terms must also be true, so we can leave out this last term corresponding to the fourth line of the table. We can thus simplify the equation to

CarryOut = (b · CarryIn) + (a · CarryIn) + (a · b)

Inputs		
a	b	CarryIn
0	1	1
1	0	1
1	1	0
1	1	1

Figure: Values of the inputs when CarryOut is a 1.

- Figure below shows that the hardware within the adder for CarryOut consists of three AND gates and one OR gate.
- The three AND gates correspond exactly to the three parenthesized terms of the formula above for CarryOut, and the OR gate sums the three terms.
- The Sum bit is set when exactly one input is 1 or when all three inputs are 1.
- The Sum results in a complex Boolean equation.

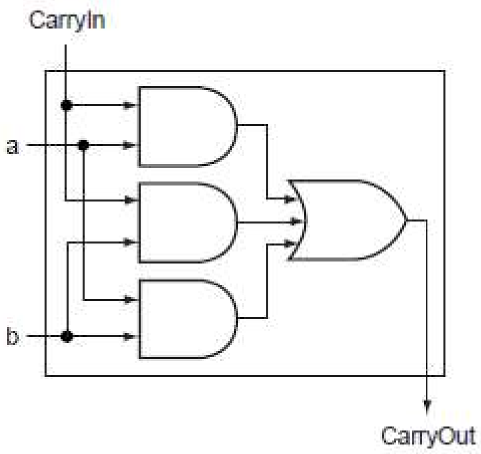


Figure: Adder hardware for the carry out signal.

- Figure below shows a 1-bit ALU derived by combining the adder with the earlier components.
- Sometimes designers also want the ALU to perform a few more simple operations, such as generating 0.
- The easiest way to add an operation is to expand the multiplexer controlled by the Operation line and, for this example, to connect 0 directly to the new input of that expanded multiplexer.

Sum = (a · b · CarryIn) + (a · b · CarryIn) + (a · b · CarryIn) + (a · b · CarryIn)

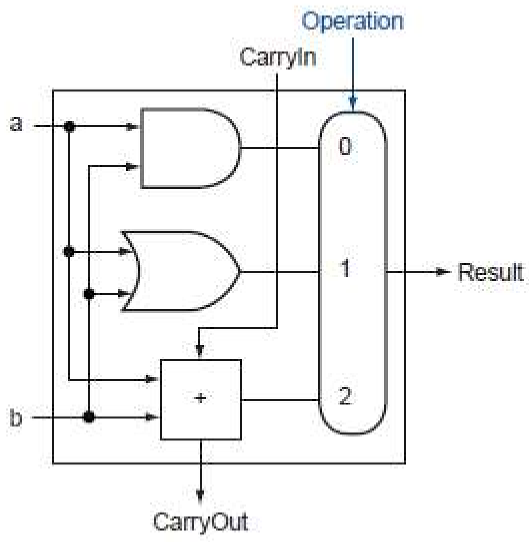


Figure: A 1-bit ALU that performs AND, OR, and addition.

2.1.2 A 32-Bit ALU

- The full 32-bit ALU is created by connecting adjacent "black boxes.
- A single carry out of the least significant bit (Result0) can ripple all the way through the adder, causing a carry out of the most significant bit (Result31).
- Hence, the adder created by directly linking the carries of 1-bit adders is called a ripple carry adder.
- Subtraction is the same as adding the negative version of an operand, and this is how adders perform subtraction. Recall that the shortcut for negating a two's complement number is to invert each bit (sometimes called the one's complement) and then add 1.

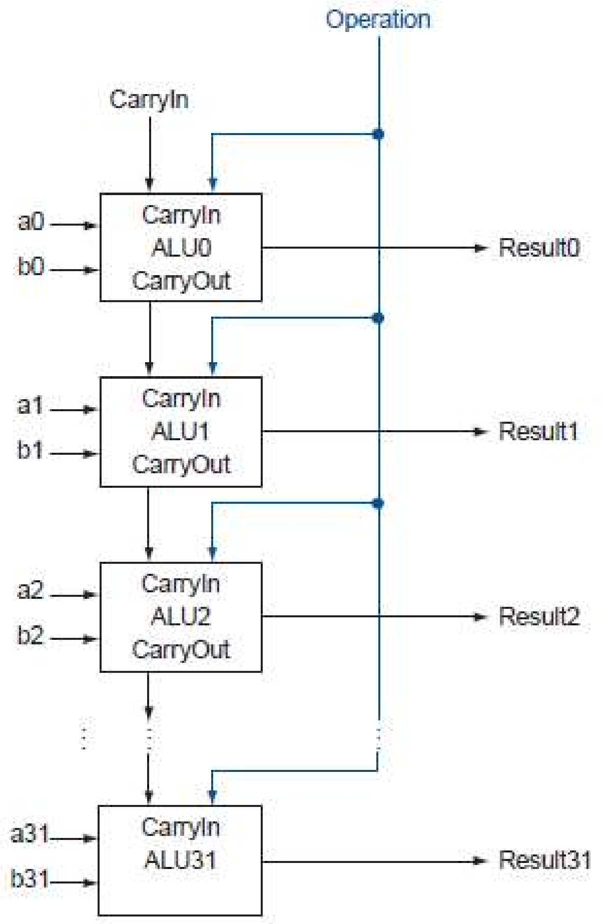


FIGURE B.5.7 A 32-bit ALU constructed from 32 1-bit ALUs.

- To invert each bit, we simply add a 2:1 multiplexor that chooses between b and b'.
- Suppose we connect 32 of these 1-bit ALUs. The added multiplexer gives the option of b or its inverted value, depending on Binvert, but this is only one step in negating a two's complement number.
- The least significant bit still has a CarryIn signal, even though its unnecessary for addition.
- If we set this CarryIn to 1 instead of 0. The adder will then calculate a + b + 1.
- By selecting the inverted version of b, we can perform subtraction process that can obtain as

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

-
- A MIPS ALU also needs a NOR function. Instead of adding a separate gate for NOR, we can reuse much of the hardware already in the ALU, like we did for subtract. The insight comes from the following truth about NOR:

$$(\overline{a + b}) = \bar{a} \cdot \bar{b}$$

- That is, NOT (a OR b) is equivalent to NOT a AND NOT b. This fact is called DeMorgan's theorem and is explored in the exercises in more depth.
    - Since we have AND and NOT b, we only need to add NOT a to the ALU.

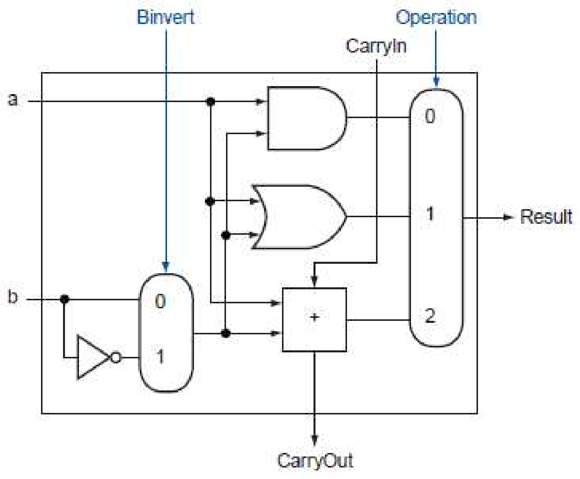


Figure: A 1-bit ALU that performs AND, OR, and addition on a and b or a and b.

- By selecting b (Binvert = 1) and setting CarryIn to 1 in the least significant bit of the ALU, we get two's complement subtraction of b from a instead of addition of b to a.

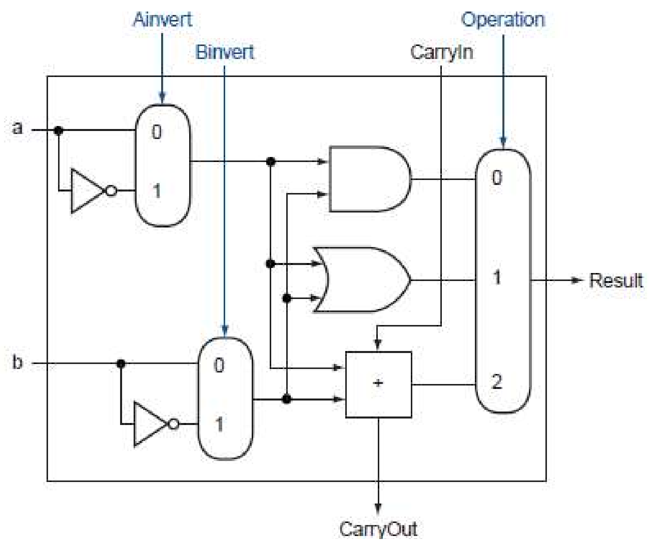


FIGURE B.5.9 A 1-bit ALU that performs AND, OR, and addition on a and b or a and b.

- By selecting a (Ainvert = 1) and b (Binvert = 1), we get a NOR b instead of a AND b.

2.1.3 Tailoring the 32-Bit ALU to MIPS

- These four operations—add, subtract, AND, OR—are found in the ALU of almost every computer, and the operations of most MIPS instructions can be performed by this ALU.
- But the design of the ALU is incomplete.One instruction that still needs support is the set on less than instruction (slt).
- Recall that the operation roduces 1 if rs < rt, and 0 otherwise. Consequently, slt will set all but the least significant bit to 0, with the least significant bit set according to the comparison.
- For the ALU to perform slt, we first need to expand the three-input multiplexer in below figure to add an input for the slt result.

- We call that new input Less and use it only for slt. The top drawing of below figure shows the new 1-bit ALU with the expanded multiplexor.
- From the description of slt above, we must connect 0 to the Less input for the upper 31 bits of the ALU, since those bits are always set to 0. What remains to consider is how to compare and set the least significant bit for set on less than instructions. What happens if we subtract b from a? If the difference is negative, then a < b since

$$\mathbf{(a - b) < 0 \Rightarrow ((a - b) + b) < (0 + b)}$$
$$\mathbf{\Rightarrow a < b}$$

- We want the least significant bit of a set on less than operation to be a 1 if a < b; that is, a 1 if a – b is negative and a 0 if it's positive.
- The result corresponds exactly to the sign bit values: 1 means negative and 0 means positive. Following this line of argument, we need only connect the sign bit from the adder output to the least significant bit to get set on less than.
- Unfortunately, the Result output from the most significant ALU bit in the top of below figure for the slt operation is not the output of the adder; the ALU output for the slt operation is obviously the input value Less.
- Thus, we need a new 1-bit ALU for the most significant bit that has an extra output bit: the adder output.
- The bottom drawing of below figure shows the design, with this new adder output line called Set, and used only for slt. As long as we need a special ALU for the most significant bit, we added the overflow detection logic since it is also associated with that bit.
- Alast, the test of less than is a little more complicated than just described because of overflow, as we explore in the exercises.
- The below figure shows the 32-bit ALU. Notice that every time we want the ALU to subtract, we set both CarryIn and Binvert to 1.
- For adds or logical operations, we want both control lines to be 0. We can therefore simplify control of the ALU by combining the CarryIn and Binvert to a single control line called Bnegate.

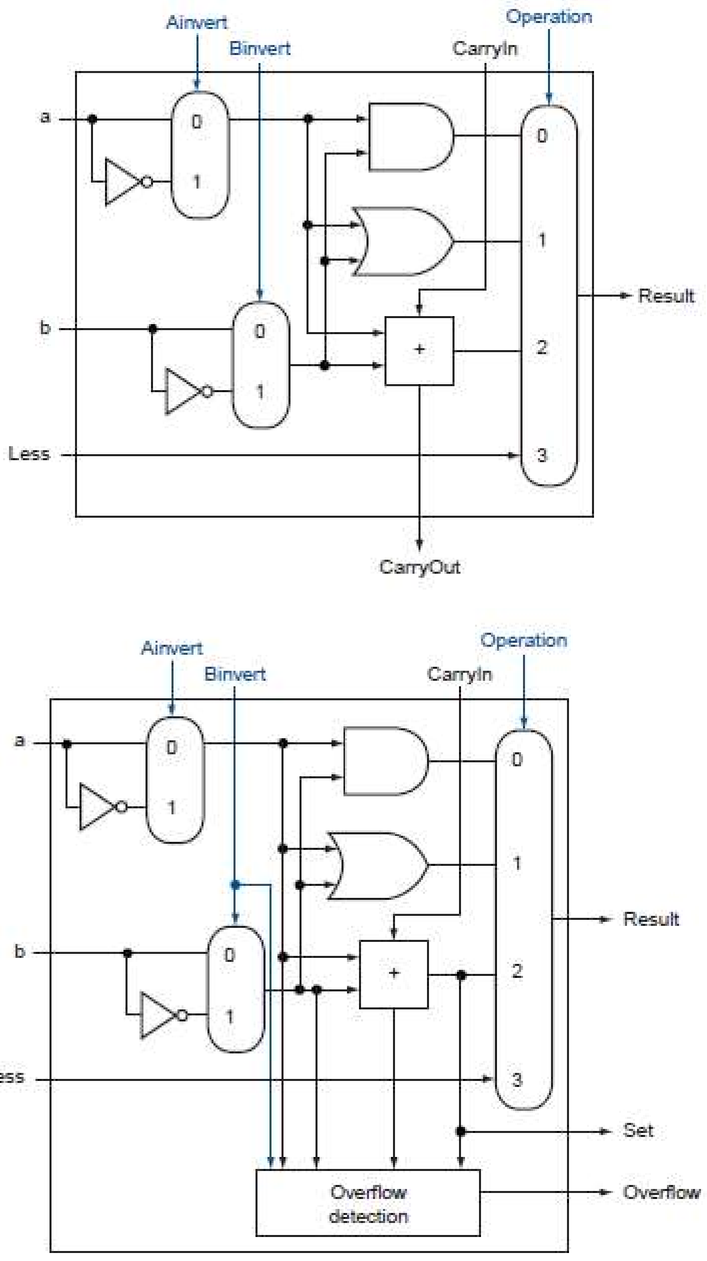


FIGURE B.5.10 (Top) A 1-bit ALU that performs AND, OR, and addition on a and b or b, and (bottom) a 1-bit ALU for the most significant bit.

- The top drawing includes a direct input that is connected to perform the set on less than operation; the bottom has a direct output from the adder for the less than comparison called Set.
- The below figure A 32-bit ALU constructed from the 31 copies of the 1-bit ALU in the top of bthe above figure and one 1-bit ALU in the bottom of that figure.
- Less inputs are connected to 0 except for the least significant bit, which is connected to the Set output of the most significant bit.
- If the ALU performs a – b and we select the input 3 in the multiplexor in Figure B.5.10, then Result = 0 . . . 001 if a < b, and Result = 0 . . . 000 otherwise.

- To further tailor the ALU to the MIPS instruction set, we must support conditional branch instructions.
- These instructions branch either if two registers are equal or if they are unequal.
- The easiest way to test equality with the ALU is to subtract b from a and then test to see if the result is 0 since

$$\mathbf{(a - b = 0) \Rightarrow a = b}$$

- Thus, if we add hardware to test if the result is 0, we can test for equality.
- The simplest way is to OR all the outputs together and then send that signal through an inverter:

$$\text{Zero} = \overline{(\text{Result31} + \text{Result30} + \ldots + \text{Result2} + \text{Result1} + \text{Result0})}$$

- The below figure shows the revised 32-bit ALU. We can think of the combination of the 1-bit Ainvert line, the 1-bit Binvert line, and the 2-bit Operation lines as 4- bit control lines for the ALU, telling it to perform add, subtract, AND, OR, or set on less than.

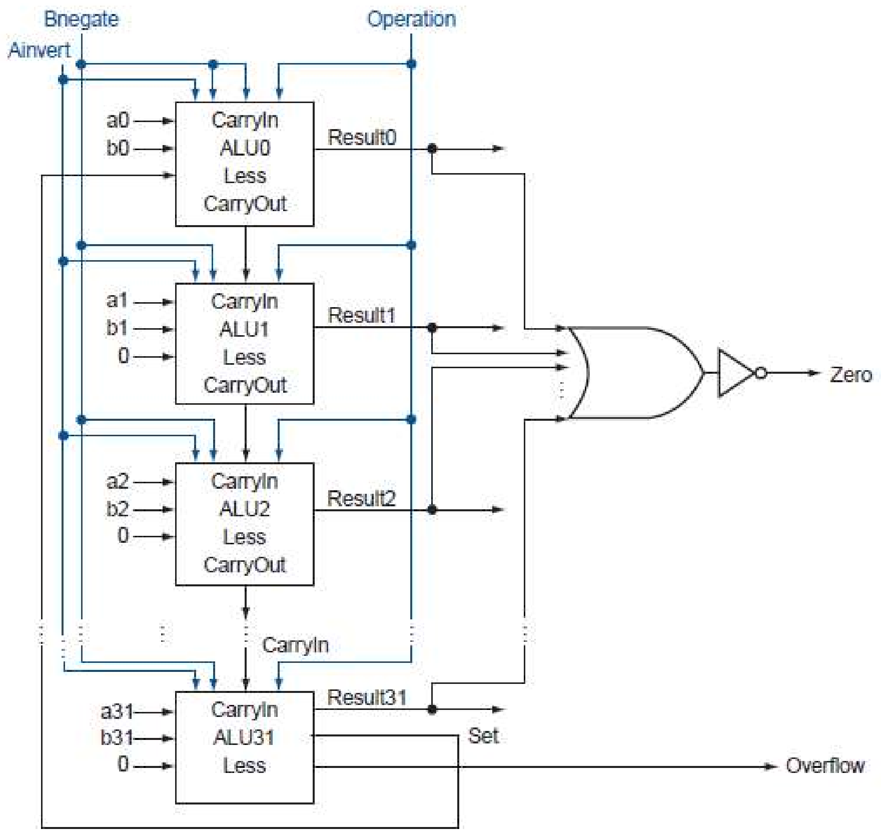


Figure : The final 32-bit ALU. This adds a Zero detector to Figure B.5.11.

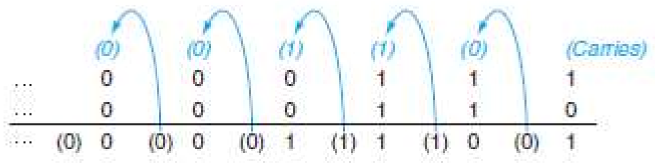
2.2 Addition and Subtraction

Addition operation:

- Digits are added bit by bit from right to left, with carries passed to the next digit to the left.
- Adding 6_{ten} to 7_{ten} in binary

```
  0000 0000 0000 0000 0000 0000 0000 0111two = 7ten
+ 0000 0000 0000 0000 0000 0000 0000 0110two = 6ten
= 0000 0000 0000 0000 0000 0000 0000 1101two = 13ten
```

Figure: Binary addition, showing carries from right to left.



- The rightmost bit adds 1 to 0, resulting in the sum of this bit being 1 and the carry out from this bit being 0. Hence, the operation for the second digit to the right is 0 + 1 + 1.
- This generates a 0 for this sum bit and a carry out of 1.
- The third digit is the sum of 1 + 1 + 1, resulting in a carry out of 1 and a sum bit of 1.
- The fourth bit is 1 + 0 + 0, yielding a 1 sum and no carry.

Overflow:

- Overflow occurs when the result from an operation cannot be represented with the available hardware, in this case a 32-bit word.

When can overflow occur in addition?

- When adding operands with different signs, overflow cannot occur. The reason is the sum must be no larger than one of the operands.
- For example, -10 + 4 = -6. Since the operands fit in 32 bits and the sum is no larger than an operand, the sum must fit in 32 bits as well.
- Therefore, no overflow can occur when adding positive and negative operands.
- Adding or subtracting two 32-bit numbers can yield a result that needs 33 bits to be fully expressed.
- The lack of a 33rd bit means that when overflow occurs, the sign bit is set with the *value* of the result instead of the proper sign of the result.
- Since we need just one extra bit, only the sign bit can be wrong.
- Hence, overflow occurs when adding two positive numbers and the sum is negative, or vice versa. This means a carry out occurred into the sign bit.
- Overflow occurs in subtraction when we subtract a negative number from a positive number and get a negative result, or when we subtract a positive

number from a negative number and get a positive result. This means a borrow occurred from the sign bit.

- The below figure shows the combination of operations, operands, and results that indicate an overflow.

Operation	Operand A	Operand B	Indicating Overflow
A+B	≥ 0	≥ 0	< 0
A+B	< 0	< 0	≥ 0
A-B	≥ 0	< 0	< 0
A-B	< 0	≥ 0	≥ 0

- We have just seen how to detect overflow for two's complement numbers in a computer.
- Unsigned integers are commonly used for memory addresses where overflows are ignored.
- The computer designer must therefore provide a way to ignore overflow in some cases and to recognize it in others.
- The MIPS solution is to have two kinds of arithmetic instructions to recognize the two choices:
    1. Add (add), add immediate (addi), and subtract (sub) cause exceptions on overflow.
    2. Add unsigned (addu), add immediate unsigned (addiu), and subtract unsigned (subu) do *not* cause exceptions on overflow.

- MIPS detects overflow with an exception, also called an interrupt on many computers. An exception or interrupt is essentially an unscheduled procedure call.
- The address of the instruction that overflowed is saved in a register, and the computer jumps to a predefined address to invoke the appropriate routine for that exception.
- The interrupted address is saved so that in some situations the program can continue after corrective code is executed.

For signed addition:

```
addu $t0, $t1, $t2                # $t0 = sum, but don't trap
xor  $t3, $t1, $t2                # Check if signs differ
slt   $t3, $t3, $zero             # $t3 = 1 if signs differ
bne $t3, $zero, No_overflow       # $t1, $t2 signs ≠, so no overflow
xor $t3, $t0, $t1                 # signs =; sign of sum match too?
# $t3 negative if sum sign different
slt $t3, $t3, $zero               # $t3 = 1 if sum sign different
bne $t3, $zero, Overflow          # All three signs ≠; go to overflow
```

For unsigned addition :

($t0 = $t1 + $t2), the test is

```
addu $t0, $t1, $t2                # $t0 = sum
nor $t3, $t1, $zero               # $t3 = NOT $t1
# (2's comp – 1: 232 – $t1 – 1)
sltu $t3, $t3, $t2                # (2^32 – $t1 – 1) < $t2
#if 2^32 – 1 < $t1 + $t2
bne $t3,$zero,Overflow            # if(2^32–1 < $t1 + $t2) go to
                                  overflow
```

2.3 Multiplication

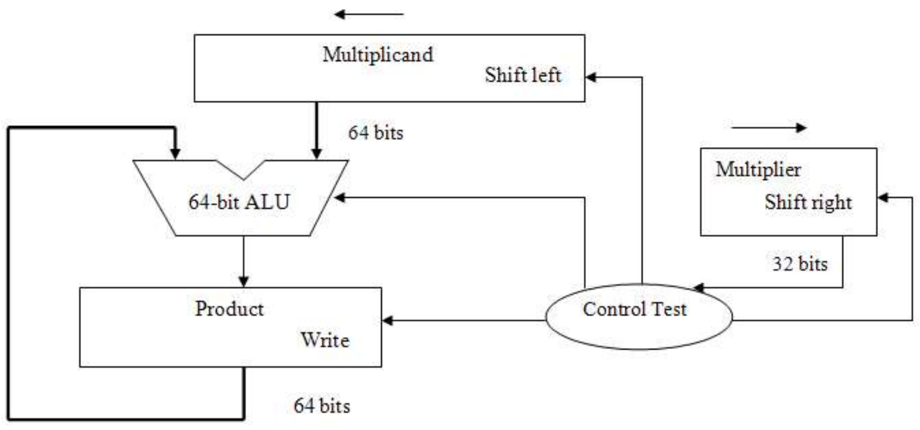
- The multiplication of binary number is done in the same way of decimal number.

```
     1101ten        Multiplicand
X    1011ten        Multiplier
     1101
    1101
   0000
  1101
 10001111ten        Product
```

- The first operand is called the multiplicand and the second the multiplier. The final result is called the product.

- To take the digits of the multiplier one at a time from right to left, multiplying the multiplicand by the single digit of the multiplier and shifting the intermediate product one digit to the left of the earlier intermediate products.
- The number of digits in the product is larger than the number in either the multiplicand or the multiplier.
- In fact, if we ignore the sign bits, the length of the multiplication of an n-bit multiplicand and an m-bit multiplier is a product that is n + m bits long.
- That is, n + m bits are required to represent all possible products. Hence, like add, multiply must cope with overflow because we frequently want a 32-bit product as the result of multiplying two 32-bit numbers.

2.3 .1 First Version of Multiplication

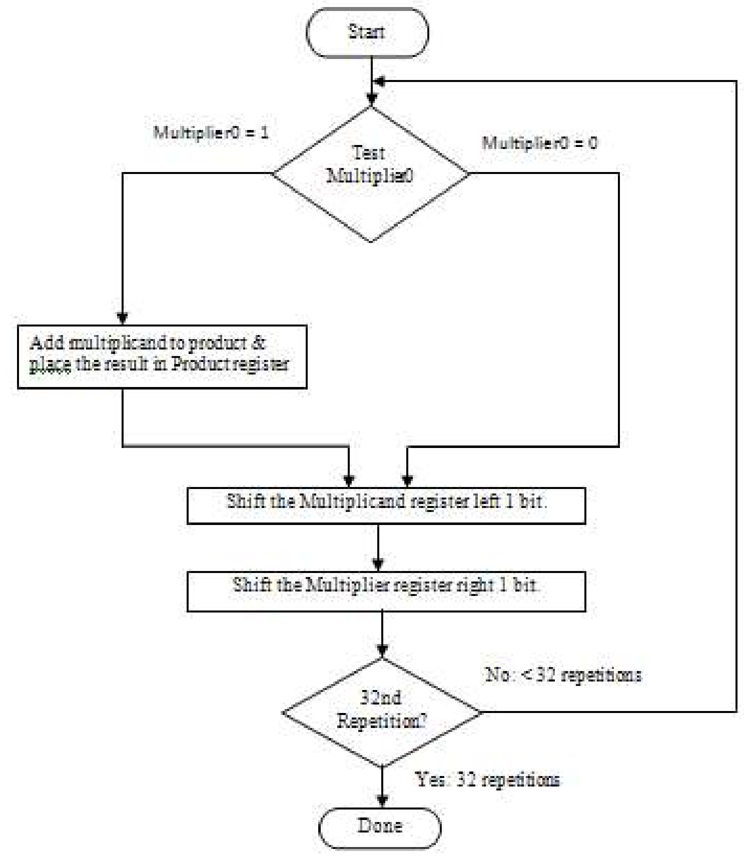


- The Multiplicand register, ALU, and the Product register are 64 bits and the Multiplier with 32 bits.
- The 32-bit multiplicand starts in the right half of the Multiplicand register, and is shifted left 1 bit on each step.
- The Multiplier is shifted in the opposite direction in each step.
- Control test is to decide when to shift the Multiplicand and Multiplier registers and when to write new values into the Product register.

Steps:

1. Initialize the register product to 0.If the least significant bit of multiplier is 1, then add the multiplicand to the product.

2. Then goto step 2,3.,if 0 then goto step 2,3.
3. Shift the Multiplicand register left 1 bit.
4. Shift the Multiplier register right 1 bit.
5. Repeat the three steps for 32 times.


Start
Multiplier0 = 1
Test Multiplier0
Multiplier0 = 0
Add multiplicand to product & place the result in Product register
Shift the Multiplicand register left 1 bit.
Shift the Multiplier register right 1 bit.
No: < 32 repetitions
32nd Repetition?
Yes: 32 repetitions
Done

First Multiplication: 0010*0011

Iteration	Step	Multiplier	Multiplicand	Product
0	Initial	0011	0000 0010	0000 0000
1	1a:M[0]=1→Prod(r)=Prod (r)+ Mcand 2: Left Shift the Multiplicand 3: Right Shift the Multiplier	0011 0011 **0001**	0000 0010 **0000 0100** 0000 0100	0000 **0010** 0000 0010 0000 0010
2	1a:M[0]=1→Prod(r)=Prod (r)+ Mcand	0001 0001	0000 0100 **0000 1000**	0000 **0110** 0000 0110

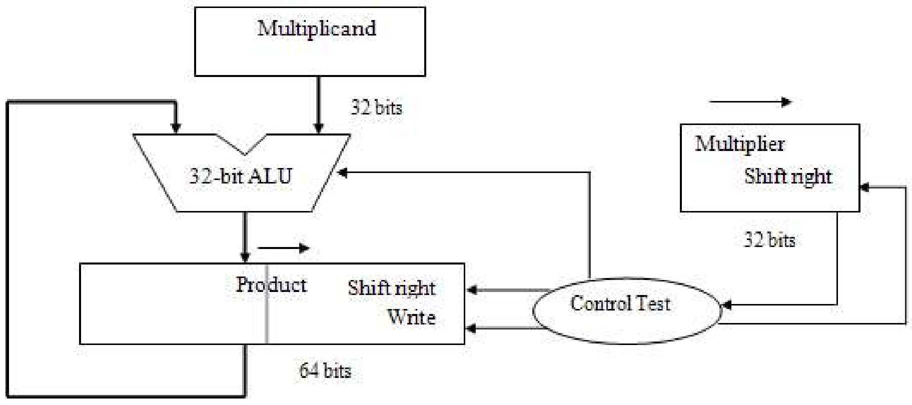
	2: Left Shift the Multiplicand	**0000**	0000 1000	0000 0110
	3: Right Shift the Multiplier			
3	1a:M[0]=0→ No Operation	0000	0000 1000	0000 0110
	2: Left Shift the Multiplicand	0000	**0001 0000**	0000 0110
	3: Right Shift the Multiplier	**0000**	0001 0000	0000 0110
4	1a:M[0]=0→ No Operation	0000	0001 0000	0000 0110
	2: Left Shift the Multiplicand	0000	**0010 0000**	0000 0110
	3: Right Shift the Multiplier	**0000**	0010 0000	0000 0110

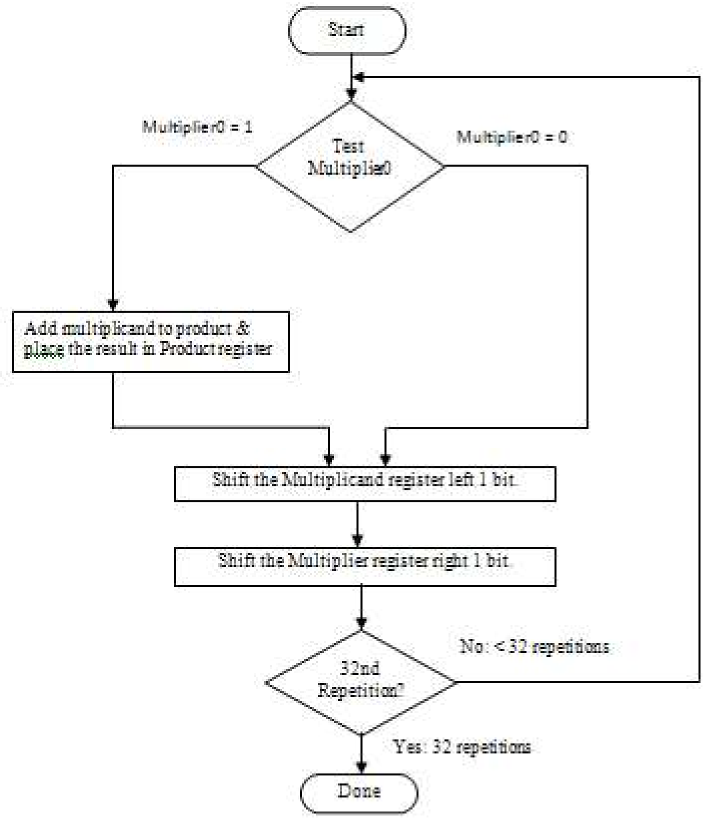
2.3.2 Second Version of Multiplication

- The Multiplicand register, ALU, and the Multiplier register are 32 bits and the Product register with 64 bits. Here the product is shifted right.
- Control test is to decide when to shift the Product and Multiplier registers and when to write new values into the Product register.

Steps:

1. Initialize the register product to 0.If the Least significant bit of multiplier is 1, then add the multiplicand to the left half of the product and place the result in the left half of the product register, then goto step 2,3.,if 0 then goto step 2,3.
2. Shift the Product register right1 bit.
3. Shift the Multiplier register right 1 bit.
4. Repeat the three steps for 32 times.




Start
Test Multiplier0
Multiplier0 = 1
Multiplier0 = 0
Add multiplicand to product & place the result in Product register
Shift the Multiplicand register left 1 bit
Shift the Multiplier register right 1 bit
32nd Repetition?
No: < 32 repetitions
Yes: 32 repetitions
Done

Second Multiplication:0010 * 0011

Iteration	Step	Multiplier	Multiplicand	Product
0	Initial	0011	0010	0000 0000
1	1a:M[0]=1→Prod(l)=Prod	0011	0010	**0010** 0000
	(l)+ Mcand	0011	0010	**0001 0000**
	2:Right Shift the Product	**0001**	0010	0001 0000
	3: Right Shift the Multiplier			
2	1a:M[0]=1→ Prod(l)=Prod	0001	0010	**0011** 0000
	(l)+ Mcand	0001	0010	**0001 1000**
	2:Right Shift the Product	**0000**	0010	0001 1000
	3: Right Shift the Multiplier			
3	1a:M[0]=0→No Operation	0000	0010	0001 1000
	2:Right Shift the Product	0000	0010	**0000 1100**
	3: Right Shift the Multiplier	**0000**	0010	0000 1100

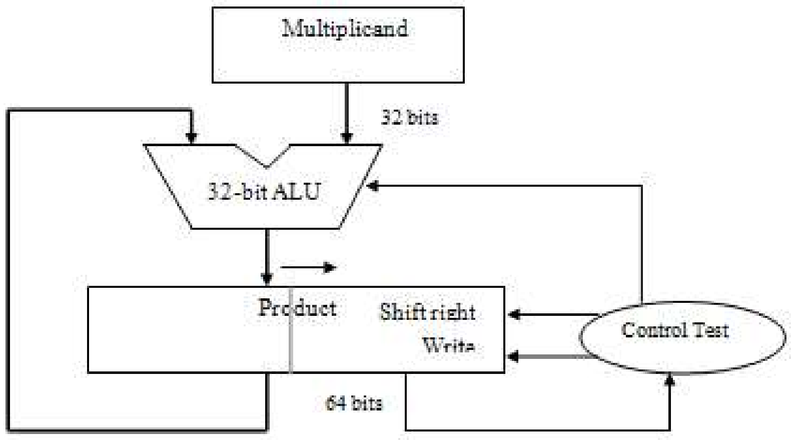
4	1a:M[0]= 0→No Operation	0000	0010	0000 1100
	2:Right Shift the Product	0000	0010	**0000 0110**
	3: Right Shift the Multiplier	**0000**	0010	0000 0110

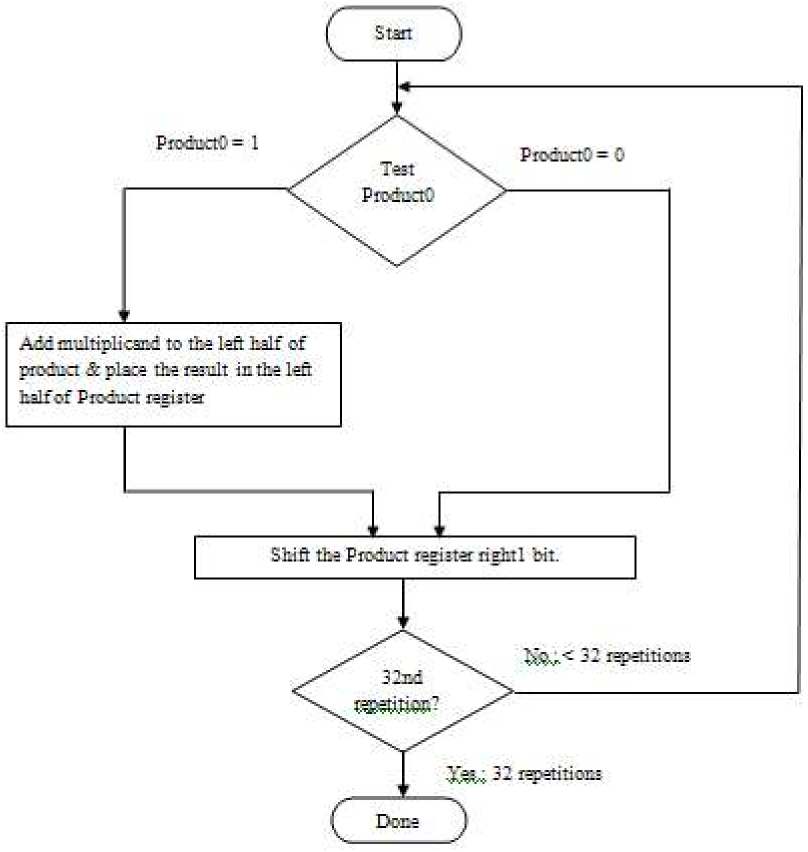
2.3.3 Third Version of Multiplication

- The Multiplicand register and ALU are 32 bits and the Product register with 64 bits. Here the product is shifted right.
- Here Multiplier register has removed and placed in right half of Product register.
- Control test is to decide when to shift the Product registers and when to write new values into the Product register.

Steps:

1. If the Least significant bit of product is 1, then add the multiplicand to the left half of the product and place the result in the left half of the product register, then goto step 2.,if 0 then goto step 2.
2. Shift the Product register right1 bit.
3. Repeat the three steps for 32 times.




Start
Test Product0
Product0 = 1
Product0 = 0
Add multiplicand to the left half of product & place the result in the left half of Product register
Shift the Product register right1 bit.
32nd repetition?
No.: < 32 repetitions
Yes.: 32 repetitions
Done

Third Multiplication:0010*0011

Iteration	Step	Multiplicand	Product
0	Initial	0010	0000 0011
1	1a:P[0]=1→Prod(l)=Prod (l)+ Mcand 2:Right Shift the Product	0010 0010	**0010** 0011 **0001 0001**
2	1a:P[0]=1→Prod(l)=Prod (l)+ Mcand 2:Right Shift the Product	0010 0010	**0011** 0001 **0001 1000**
3	1b:P[0]=0→ No Operation 2:Right Shift the Product	0010 0010	0001 1000 **0000 1100**
4	1b:P[0]=0→ No Operation 2:Right Shift the Product	0010 0010	**0000 1100** 0000 0110

2.3.4 Signed Multiplication

- The easiest way to understand how to deal with signed numbers is to first convert the multiplier and multiplicand to positive numbers and then remember the original signs.
- The algorithms should then be run for 31 iterations, leaving the signs out of the calculation.
- The algorithm can be used for infinite digits, and we are representing them with 32 bits.
- Hence, the shifting steps would need to extend the sign of the product for signed numbers.
- When the algorithm completes, the lower word would have the 32-bit product.

2.3.5 Faster Multiplication

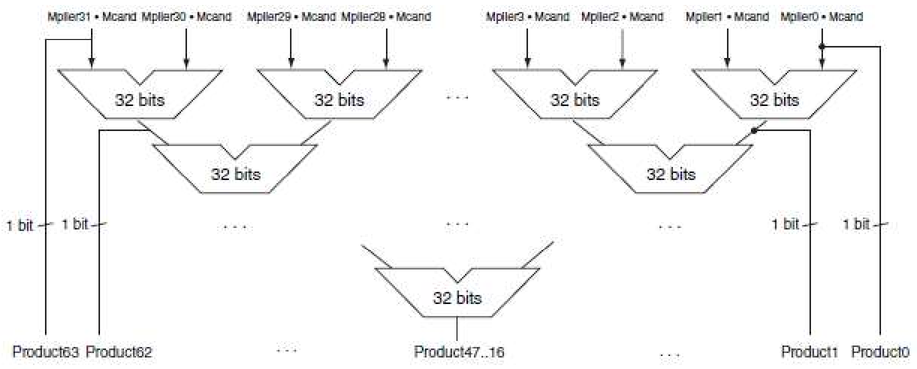


Figure: Faster Multiplication hardware.

- Moore's law has provided so much more in resources that hardware designers can now build much faster multiplication hardware.
- Whether the multiplicand is to be added or not is known at the beginning of the multiplication by looking at each of the 32 multiplier bits.
- Faster multiplications are possible by providing one 32-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.

- Instead of waiting for 32 add times, we wait just the $\log_2$ (32) or five 32-bit add times. In fact, multiply can go even faster than five add times because of the use of carry save adders.

2.3.6 Multiply in MIPS

- MIPS provide a separate pair of 32-bit registers to contain the 64-bit product, called *Hi* and *Lo*.
- To produce a properly signed or unsigned product, MIPS has two instructions: multiply (mult) and multiply unsigned multu).
- To fetch the integer 32-bit product, the programmer uses *move from lo* (mflo).
- The MIPS assembler generates a pseudo instruction for multiply that specifies three general purpose registers, generating mflo and mfhi instructions to place the product into registers.

2.4 Division

- The reciprocal operation of multiply is divide, an operation that is even less frequent and even more quirky. It even offers the opportunity to perform a mathematically invalid operation: dividing by 0.
- The example is dividing 1,001,010ten by 1000ten:

```
                      1001ten     Quotient
Divisor 1000ten    1001010ten     Dividend
                  -1000
                     10
                     101
                     1010
                    -1000
                       10ten      Remainder
```

- Divide's two operands, called the dividend and divisor, and the result, called the quotient, are accompanied by a second result, called the remainder. Here is another way to express the relationship between the components:

Dividend =Quotient x Divisor + Remainder

- Check whether the remainder is smaller than the divisor. Infrequently, programs use the divide instruction just to get the remainder, ignoring the quotient.
- Binary numbers contain only 0 or 1, so binary division is restricted to these two choices, thereby simplifying binary division.
- So it's easy to figure out how many times the divisor goes into the portion of the dividend:

Dividend:

- It is a number being divided.

Divisor:

- It is a number that the dividend is divided by.

Quotient:

- It is a primary result of a division; a number that when multiplied by the divisor and added to the remainder produces the dividend.

Remainder:

- It is a secondary result of a division; a number that when added to the product of the quotient and the divisor produces the dividend.

Division Hardware

2.4 .1 First version of division

- The Divisor register, ALU, and Remainder register are all 64 bits wide, with only the Quotient register being 32 bits.
- The 32-bit divisor starts in the left half of the Divisor register and is shifted right 1 bit each iteration.
- The remainder is initialized with the dividend. Control decides when to shift the Divisor and Quotient registers and when to write the new value into the Remainder register.

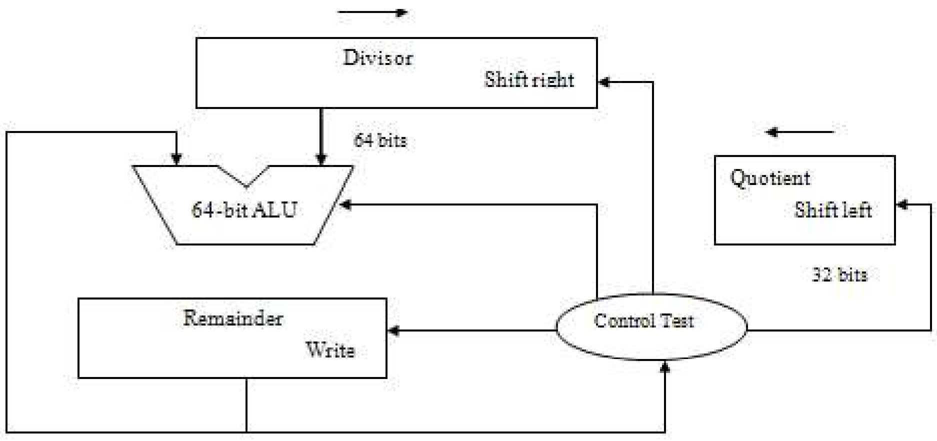


Figure: First version of the division hardware.

Division: 0111 by 0010

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->Rem(l)+Div,Left shift Q,Q0=0	**0000**	0010 0000	**0000** 0111
	3:Right Shift Divisor	0000	**0001 0000**	0000 0111
2	1:Rem=Rem-Div	0000	0001 0000	**1111 0111**
	2b:Rem<0->Rem(l)+Div,Left shift Q,Q0=0	**0000**	0001 0000	**0000** 0111
	3:Right Shift Divisor	0000	**0000 1000**	0000 0111
3	1:Rem=Rem-Div	0000	0000 1000	**1111 1111**
	2b:Rem<0->Rem(l)+Div,Left shift Q,Q0=0	**0000**	0000 1000	**0000** 0111
	3:Right Shift Divisor	0000	**0000 0100**	0000 0111
4	1:Rem=Rem-Div	0000	0000 0100	**0000 0011**
	2b:Rem>=0-> Left shift Q,Q0=1	**0001**	0000 0100	0000 0011
	3:Right Shift Divisor	0001	**0000 0010**	0000 0011
5	1:Rem=Rem-Div	0001	0000 0010	**0000 0001**

	2b:Rem>=0-> Left shift Q,Q0=1	**0011**	0000 0010	0000 0001
	3:Right Shift Divisor	0011	**0000 0001**	0000 0001

Division Algorithm

- We start with the 32-bit Quotient register set to 0.
- Each iteration of the algorithm needs to move the divisor to the right one digit, so we start with the divisor placed in the left half of the 64-bit Divisor register and shift it right 1 bit each step to align it with the dividend.
- The Remainder register is initialized with the dividend.

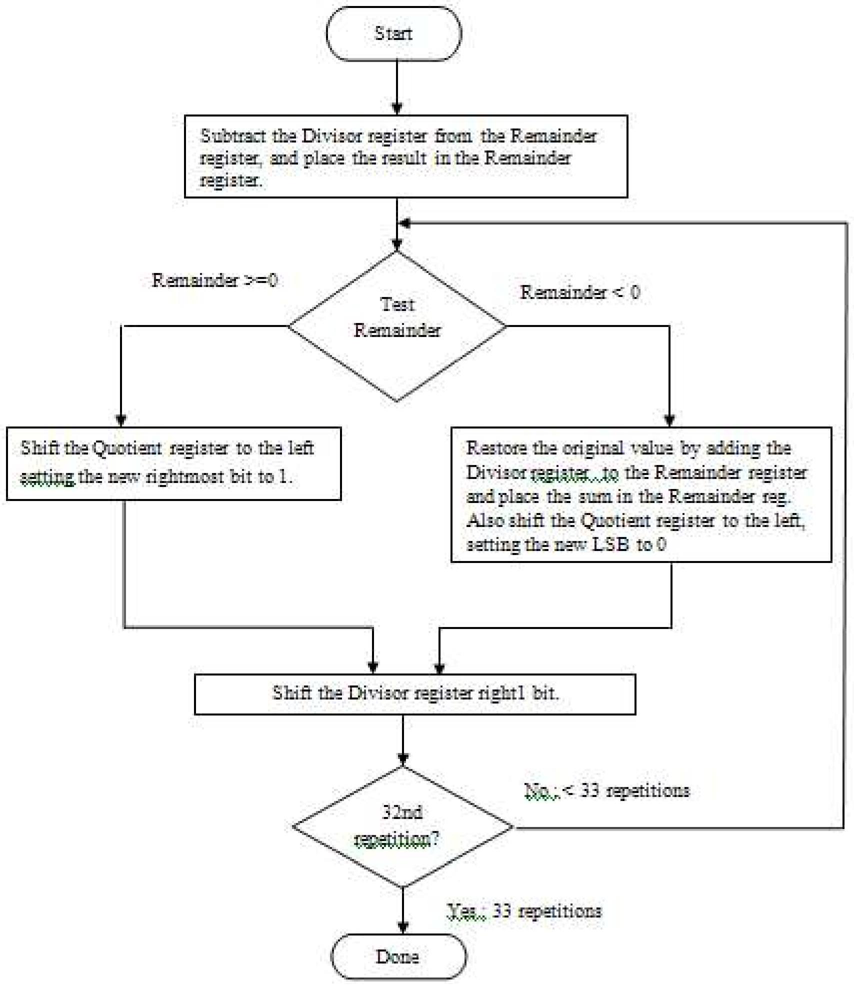

Start
Subtract the Divisor register from the Remainder register, and place the result in the Remainder register.
Remainder >=0
Test Remainder
Remainder < 0
Shift the Quotient register to the left setting the new rightmost bit to 1.
Restore the original value by adding the Divisor register to the Remainder register and place the sum in the Remainder reg. Also shift the Quotient register to the left, setting the new LSB to 0
Shift the Divisor register right1 bit.
32nd repetition?
No: < 33 repetitions
Yes: 33 repetitions
Done

Figure : A division algorithm.

Step 1:

- If the remainder is positive, the divisor did go into the dividend, so step 2a generates a 1 in the quotient.

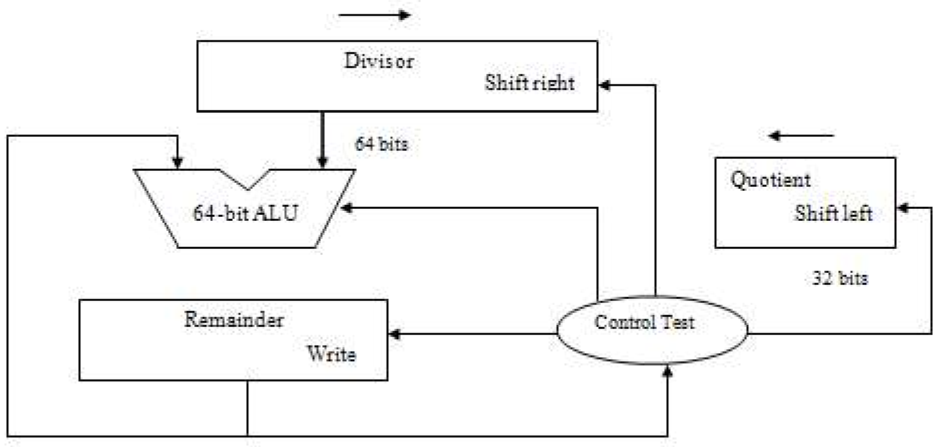
Step 2:

- A negative remainder after step 1 means that the divisor did not go into the dividend, so step 2b generates a 0 in the quotient and adds the divisor to the remainder, thereby reversing the subtraction of step 1.

Step 3:

- The final shift, in step 3, aligns the divisor properly, relative to the dividend for the next iteration. These steps are repeated 33 times.

2.4.2 Second version of division



2.4.3 Third version of division

- The Divisor register, ALU, and Quotient register are all 32 bits wide, with only the Remainder register left at 64 bits.
- Compared to first version, the ALU and Divisor registers are halved and the remainder is shifted left.
- This version also combines the Quotient register with the right half of the Remainder register.

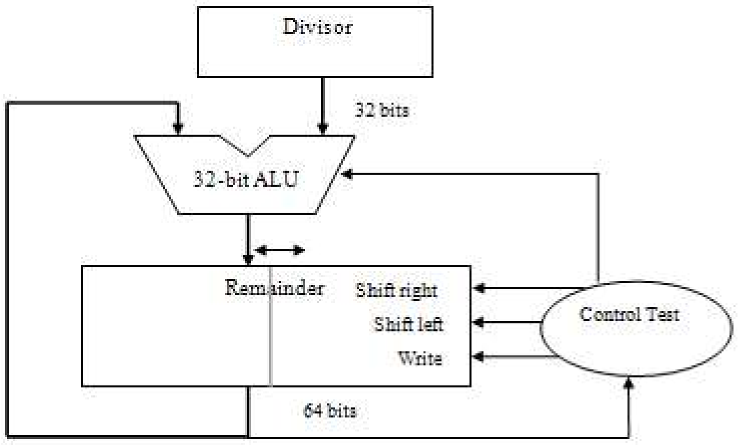
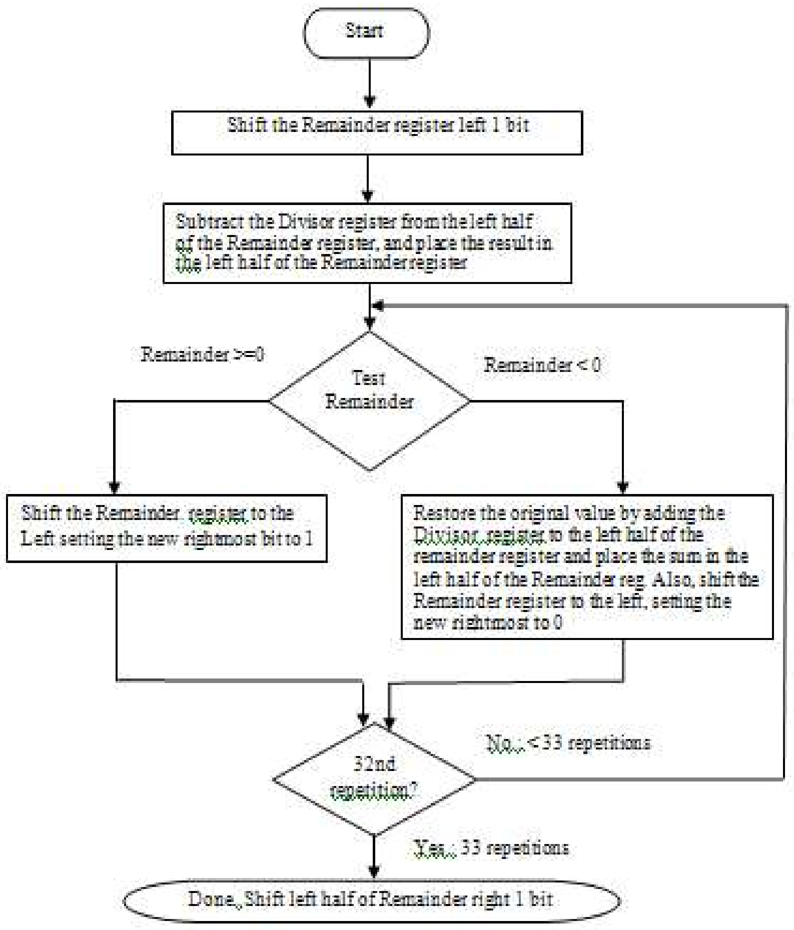


FIGURE 3.12 An improved version of the division hardware.


Start
Shift the Remainder register left 1 bit
Subtract the Divisor register from the left half of the Remainder register, and place the result in the left half of the Remainder register
Remainder >=0
Test Remainder
Remainder < 0
Shift the Remainder register to the Left setting the new rightmost bit to 1
Restore the original value by adding the Divisor register to the left half of the remainder register and place the sum in the left half of the Remainder reg. Also, shift the Remainder register to the left, setting the new rightmost to 0
No.: <33 repetitions
32nd repetition?
Yes.: 33 repetitions
Done. Shift left half of Remainder right 1 bit

Division: 0111 by 0010

Iteration	Step	Divisor	Remainder
0	Initial Values 1:Left shift Rem	0010	0000 0111 **0000 1110**
1	2:Rem(l)=Rem(l)-Div 3b:Rem<0->Rem(l)+Div, Left shift the Rem,R0=0	0010	**1110** 1110 **0000** 1110 **0001 1100**
2	2:Rem(l)=Rem(l)-Div 3b:Rem<0->Rem(l)+Div, Left shift the Rem,R0=0	0010	**1111** 1100 **0001** 1100 **0011 1000**
3	2:Rem(l)=Rem(l)-Div 3a:Rem>=0-> Left shift the Rem,R0=1	0010	**0001** 1000 **0011** 0001
4	2:Rem(l)=Rem(l)-Div 3a:Rem>=0-> Left shift the Rem,R0=1	0010	**0001** 0001 **0010** 0011
5	Right shift the Rem(l)	0010	**0001** 0011

2.4.4 Signed Division

- The simplest solution is to remember the signs of the divisor and dividend and then negate the quotient if the signs disagree.
- Thus the signed division algorithm negates the quotient if the signs of the operands are opposite and makes the sign of the non-zero remainder match the dividend.

2.4.5 Faster 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 32 partial products immediately.

- There are techniques to produce more than one bit of the quotient per step.
- The SRT division technique tries to guess several quotient bits per step, using a table lookup based on the upper bits of the dividend and remainder.
- This technique has subsequent steps to correct wrong guesses. 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.
- The accuracy of this fast method depends on having proper values in the look-up-table.

2.4.6 Divide in MIPS

- The same sequential hardware can be used for both multiply and divide.
- The only requirement is a 64-bitregister that can shift left or right and a 32-bit ALU that adds or subtracts.
- Hence, MIPS uses the 32-bit Hi and 32-bit Lo registers for both multiply and divide.
- To handle both signed integers and unsigned integers, MIPS have two instructions: divide (div) and divide unsigned (divu).
- The MIPS assembler allows divide instructions to specify three registers, generating the mflo or mfhi instructions to place the desired result into a general-purpose register.

2.5 Floating Point:

- In addition to signed and unsigned integers, programming languages support numbers with fractions, which are called reals in mathematics. Here are some examples for real numbers:
    1. 3.14159265. . .ten (pi)
    2. 2.71828. . .ten (*e*)

3. 0.000000001_{ten} or $1.0_{ten} \times 10^{-9}$ (seconds in a nanosecond)
4. $3{,}155{,}760{,}000_{ten}$ or $3.15576_{ten} \times 10^{9}$ (seconds in a typical century)

- From the above values notice that in the last two cases, the number didn't represent a small fraction, but the two numbers are bigger than 32-bit signed integer. So the alternative notation for the last two numbers is called scientific notation.

Scientific notation:

- It is a notation which has a single digit to the left of the decimal point. A number in scientific notation that has no leading 0s is called a **normalized** number.
- For example, $1.0_{ten} \times 10^{-9}$ is in normalized scientific notation, but $0.1_{ten} \times 10^{-8}$ and $10.0_{ten} \times 10^{-10}$ are not. Binary numbers can be represented in normalized form.
- To keep a binary number in normalized form, we need a base that we can increase or decrease by exactly the number of bits the number must be shifted to have one nonzero digit to the left of the decimal point.
- Computer arithmetic that supports such numbers is called **floating point.**

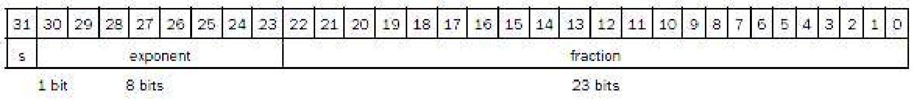
Floating point:

- Computer arithmetic represents numbers in which the binary point is not fixed called Floating point.
- Floating point represents numbers in which the binary point is not fixed.
- A standard scientific notation for real in normalized form has three advantages.
  1. It simplifies exchange of data that includes floating-point numbers.
  2. It simplifies the floating-point arithmetic algorithms to know that numbers will always be in this form.
  3. It increases the accuracy of the numbers that can be stored in a word.

2.5.1 Floating-Point Representation

- A designer of a floating-point representation must find a compromise between the size of the fraction and the size of the exponent.

- Because a fixed word size means we must take a bit from one to add a bit to the other.
- Increasing the size of the fraction enhances the precision of the fraction.
- Increasing the size of the exponent increases the range of numbers that can be represented.
- Floating-point numbers are usually a multiple of the size of a word.
- The representation of a MIPS floating-point number is shown below,



- where *s* is the sign of the floating-point number (1 meaning negative), exponent is the value of the 8-bit exponent field and fraction is the 23-bit number.
- This representation is called sign and magnitude, since the sign has a separate bit from the rest of the number.
- In general, floating-point numbers are generally of the form

$$(-1)^S \text{ x F x } 2^E$$

- F involves the value in the fraction field and E involves the value in the exponent
- These sizes of exponent and fraction give MIPS computer arithmetic an extraordinary range.
- Thus, overflow interrupts can occur in floating-point arithmetic as well as in integer arithmetic.

Overflow (floating-point):

- Overflow means the exponent is too large to be represented in the exponent field.
- It is a situation in which a positive exponent becomes too large to fit in the exponent field.

Underflow (floating-point):

- Underflow occurs when the negative exponent is too large to fit in the exponent field.

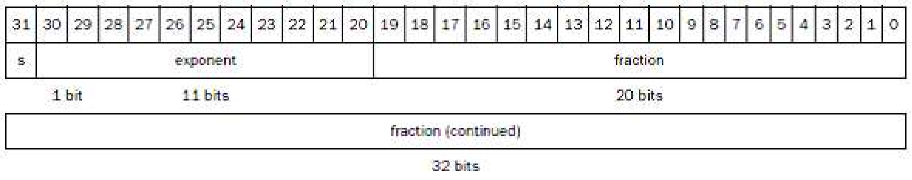
- It is a situation in which a negative exponent becomes too large to fit in the exponent field.
- Both underflow and overflow condition will occur in floating-point arithmetic.
- In C this number is called double, and operations on doubles are called double precision floating-point arithmetic; single precision floating point is the name of the earlier format.

Double precision:

- A floating point value represented in two 32-bit words.

Single precision:

- A floating point value represented in a single 32-bit word.



- The representation of a double precision floating-point number takes two MIPS words, as shown below, where s is still the sign of the number, exponent is the value of the 11-bit exponent field, and fraction is the 52-bit number in the fraction.
- Its advantage is its greater precision because of the larger fraction.

IEEE 754 floating-point Standard:

- This standard has greatly improved both the ease of porting floating-point programs and the quality of computer arithmetic.
- To pack even more bits into the significant, IEEE 754 makes the leading 1 bit of normalized binary numbers implicit.
- Hence, the number is actually 24 bits long in single precision (implied 1 and a 23-bit fraction), and 53 bits long in double precision (1 + 52).
- Example for invalid operations, such as 0/0 or subtracting infinity from infinity. This symbol is NaN, for Not a Number.
- Special symbol to represent unusual events (∞).

Floating-point representation:

- Floating-point representation of IEEE 754 performs integer comparisons, especially for sorting.
- Sign in the most significant bit allows a quick test of less than, greater than, or equal to 0.
- Placing the exponent before the significand also simplifies sorting of floating point numbers using integer comparison instructions.
- Because numbers with bigger exponents look larger than numbers with smaller exponents, as long as both exponents have the same sign.
- Negative exponents create a challenge to simplified sorting.
- If we use two's complement or any other notation in which negative exponents have a 1 in the most significant bit of the exponent field, a negative exponent will look like a big number.
- For example, $1.0_{two} \times 2^{-1}$ would be represented as

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	1	1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 . . .																						

- Remember that the leading 1 is implicit in the significand, so the value $1.0_{two} \times 2^{+1}$ would look like the smaller binary number.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 . . .																						

Biased notation:

- It is a convention way to represent the most negative exponent as $00\ldots00_{two}$ and the most positive as $11\ldots11_{two}$.
- The bias is the number subtracted from the normal, unsigned representation to determine the real value.
- IEEE 754 uses a bias of 127 for single precision, so −1 is represented by the bit pattern of the value $-1 + 127_{ten}$, or $126_{ten} = 0111\ 1110_{two}$, and +1 is represented by 1 + 127, or $128_{ten} = 1000\ 0000_{two}$.
- Biased exponent means that the value represented by a floating-point number is really

$$(-1)^S \text{ x } (1 + \text{Fraction}) \text{ x } 2^{(\text{Exponent} - \text{Bias})}$$

- The exponent bias for double precision is 1023.

Floating-Point Representation:

Example:1

Show the IEEE 754 binary representation of the number -0.75_{ten} in single and double precision.

Solution:

The number -0.75_{ten} is also written as

$-3/4_{ten}$ or $-3/2^2_{ten}$

It is also represented by the binary fraction

$-11_{two}/2^2_{ten}$ or -0.11_{two}

In scientific notation, the value is

$-0.11_{two} \text{ x } 2^0$

and in normalized scientific notation, it is

$-1.1_{two} \text{ x } 2^{-1}$

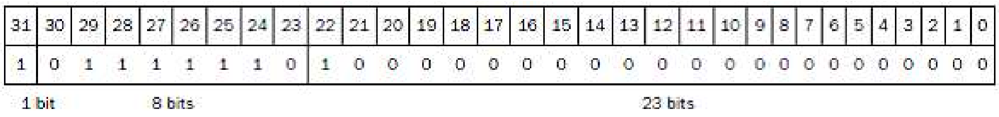
The general representation for a single precision number is

$(-1)^S \text{ x } (1 + \text{Fraction}) \text{ x} 2^{(\text{Exponent} - 127)}$

When we subtract the bias 127 from the exponent of $-1.1_{two} \text{ x} 2^{-1}$, the result is

$(-1)^1 \text{ x}(1 + .1000\ 0000\ 0000\ 0000\ 0000\ 000_{two}) \text{ x } 2^{(126 - 127)}$

The single precision binary representation of -0.75_{ten} is then

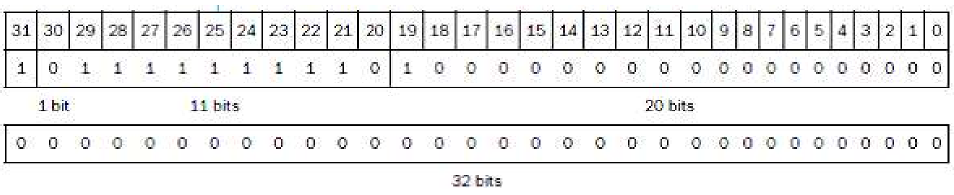


31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

1 bit 8 bits 23 bits

The double precision representation is

$(-1)^1 \text{ x } (1 + .1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two}) \text{ x } 2^{(1022-1023)}$



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

1 bit 11 bits 20 bits

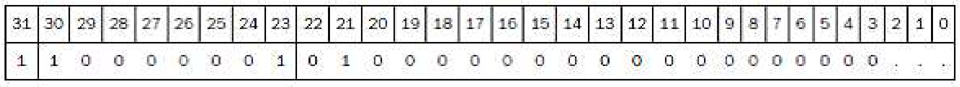
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

32 bits

Converting Binary to Decimal Floating Point:

Example:2

What decimal number is represented by this single precision float?



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	.	.	.

Answer:

The sign bit is 1, the exponent field contains 129, and the fraction field contains 1 x 2^{-2} = 1/4, or 0.25. Using the basic equation,

$(-1)^S$ x (1 + Fraction) x $2^{(\text{Exponent} - \text{Bias})}$

= $(-1)^1$ x (1 + 0.25) x $2^{(129-127)}$

= -1 x 1.25x 2^2

= -1.25 x 4

= -5.0

2.5.2 Floating-Point Addition

Let's add two numbers in scientific notation and illustrate the problems in floating-point addition:

9.999_{ten} x10^1

1.610_{ten} x 10^{-1}.

Assume that we can store only four decimal digits of the significand and two decimal digits of the exponent.

Step 1:

- To add these numbers properly, we must align the decimal point of the number that has the smaller exponent.
- Hence, we need a form of the smaller number, 1.610_{ten} x 10^{-1}, that matches the larger exponent.
- There are multiple representations of an un-normalized floating-point number in scientific notation:

1.610_{ten} x 10^{-1} = 0.1610_{ten} x10^0 = 0.01610_{ten} x 10^1

- These numbers matches the exponent of the larger number, 9.999_{ten} x10^1.
- Thus the first step shifts the significand of the smaller number to the right until its corrected exponent matches that of the larger number.
- But we can represent only four decimal digits so, after shifting, the number is really:

  0.016_{ten} x 10^1

Step 2:

- Perform the addition of the significands:

$$\begin{array}{r} 9.999_{ten} \\ \underline{+\ 0.016_{ten}} \\ 10.015_{ten} \end{array}$$

- The sum is 10.015_{ten} x 10^1.

Step 3:

This sum is not in normalized scientific notation, so we need to adjust it:

10.015_{ten} x 10^1 = 1.0015_{ten} x 10^2

shifting to the right, but

- If one number were positive and the other negative, we need to left shifts to get normalized form.
- Whenever the exponent is increased or decreased, we must check for overflow or underflow.
- that is, we must make sure that the exponent still fits in its field.

Step 4:

- Since we assumed that the significand can be only four digits long (excluding the sign), we must round the number.
- The number 1.0015_{ten} x10^2 is rounded to four digits in the significand to 1.002_{ten} x 10^2.
- The sum may not be normalized and we would need to perform step 3 again.

Decimal Floating-Point Addition:

Example:3

Add two numbers 0.5_{ten} and -0.4375_{ten} in binary using floating-Point Addition algorithm.

Answer:

- Check whether the two numbers are in normalized notation,
- Assuming 4 bits of precision:

$0.5_{ten} = 1/2_{ten} = 1/2^1_{ten}$

$= 0.1_{two} = 0.1_{two} \times 2^0 = 1.000_{two} \times 2^{-1}$

$-0.4375_{ten} = -7/16_{ten} = -7/2^4_{ten}$

$= -0.0111_{two} = -0.0111_{two} \times 2^0 = -1.110_{two} \times 2^{-2}$

Now we follow the algorithm:

Step 1:

- The significand of the number with the lesser exponent ($-1.11_{two} \times 2^{-2}$) is shifted right until its exponent matches the larger number:

$-1.110_{two} \times 2^{-2} = -0.111_{two} \times 2^{-1}$

Step 2:

- Add the significands:

$1.000_{two} \times 2^{-1} + (-0.111_{two} \times 2^{-1}) = 0.001_{two} \times 2^{-1}$

Step 3:

- Normalize the sum, checking for overflow or underflow:

$0.001_{two} \times 2^{-1} = 0.010_{two} \times 2^{-2} = 0.100_{two} \times 2^{-3}$

$= 1.000two \times 2^{-4}$

- Since $127 \geq -4 \geq -126$, there is no overflow or underflow.

Step 4:

- Round the sum

$1.000_{two} \times 2^{-4}$

- The sum already fits exactly in 4 bits, so there is no change to the bits due to rounding.
- This sum is then

$1.000_{two} \times 2^{-4} = 0.0001000_{two} = 0.0001_{two}$

$= 1/24_{ten} = 1/16_{ten} = 0.0625_{ten}$

- This sum is what we would expect from adding 0.5_{ten} to -0.4375_{ten}.

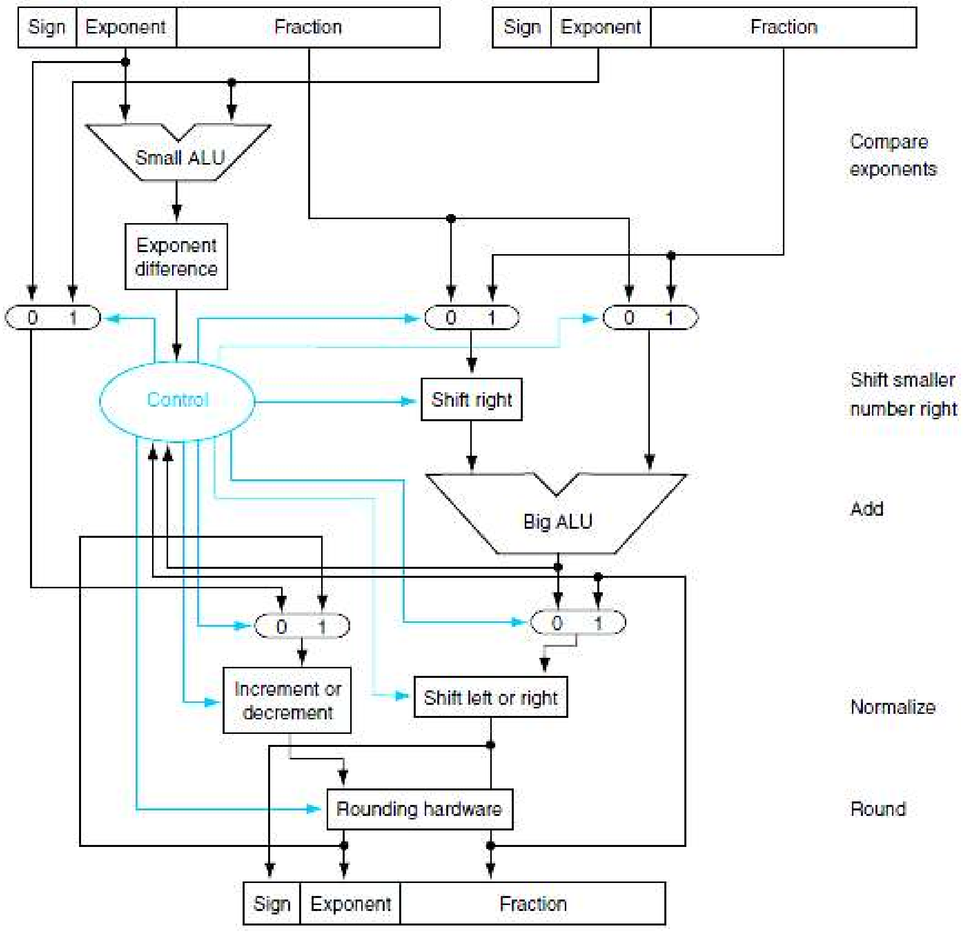
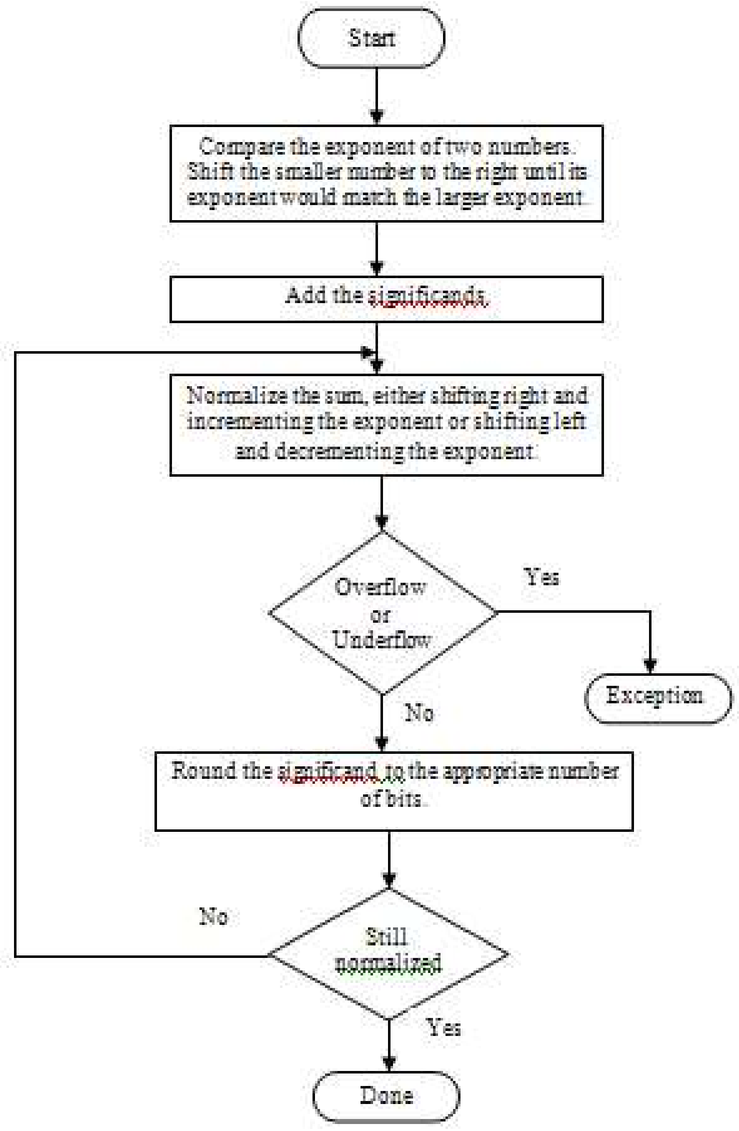


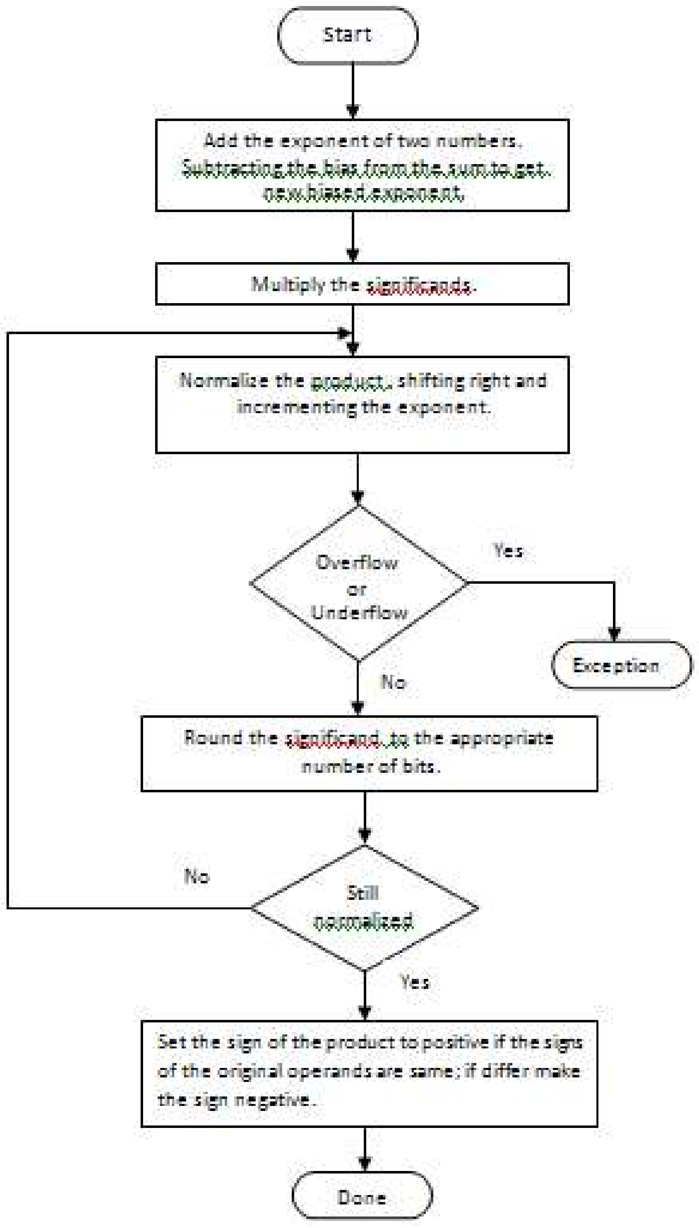
Figure: Block diagram of an arithmetic unit dedicated to floating-point addition.

- First, the exponent of one operand is subtracted from the other using the small ALU to determine which is larger and by how much.
- The multiplexers select the larger exponent, the significand of the smaller number, and the significand of the larger number. The smaller significand is shifted right, and then the significands are added together using the big ALU.
- The normalization step then shifts the sum left or right and increments or decrements the exponent.
- Rounding then creates the final result, which may require normalizing again to produce the final result.

Floating-Point Addition:


Start
Compare the exponent of two numbers. Shift the smaller number to the right until its exponent would match the larger exponent.
Add the significands.
Normalize the sum, either shifting right and incrementing the exponent or shifting left and decrementing the exponent.
Overflow or Underflow
Yes
Exception
No
Round the significand to the appropriate number of bits.
No
Still normalized
Yes
Done

2.5.3 Floating-Point Multiplication


Start
Add the exponent of two numbers. Subtracting the bias from the sum to get new biased exponent.
Multiply the significands.
Normalize the product, shifting right and incrementing the exponent.
Overflow or Underflow
Yes
Exception
No
Round the significand to the appropriate number of bits.
Still normalized
No
Yes
Set the sign of the product to positive if the signs of the original operands are same; if differ make the sign negative.
Done

Example:4

Multiply the numbers 0.5_{ten} and -0.4375_{ten}, using floating-Point algorithm.\

Answer:

In binary, the task is multiplying $1.000_{two} \times 2^{-1}$ by $-1.110_{two} \times 2^{-2}$.

Step 1:

- Adding the exponents without bias:

  $-1 + (-2) = -3$

or, using the biased representation:

$(-1 + 127) + (-2 + 127) - 127 = (-1 - 2)+(127 + 127 - 127)$

$= -3 + 127 = 124$

Step 2:

- Multiplying the significands:

$$\begin{array}{r} 1.000_{two} \\ \underline{\times\ 1.110_{two}} \\ 0000 \\ 1000 \\ 1000 \\ \underline{1000} \\ 1110000_{two} \end{array}$$

- The product is $1.110000_{two} \times 2^{-3}$, but we need to keep it to 4 bits, so it is $1.110_{two} \times 2^{-3.}$

Step 3:

- Now we check the product to make sure it is normalized, and then check the exponent for overflow or underflow. The product is already normalized and, since $127 \geq -3 \geq -126$, there is no overflow or underflow.

Step 4:

- Rounding the product makes no change:

  $1.110_{two} \times 2^{-3}$

Step 5:

- Since the signs of the original operands differ, make the sign of the product negative. Hence the product is

  $-1.110_{two} \times 2^{-3}$

- Converting to decimal to check our results:

$-1.110_{two} \times 2^{-3} = -0.001110_{two} = -0.00111_{two}$

$= -7/25_{ten} = -7/32_{ten} = -0.21875_{ten}$

- The product of 0.5_{ten} and -0.4375_{ten} is indeed -0.21875_{ten}.

2.5.4 Floating-Point Instructions in MIPS

- MIPS supports the IEEE 754 single precision and double precision formats with these instructions:
    1. Floating-point addition, single (add.s) and addition, double (add.d)
    2. Floating-point subtraction, single (sub.s) and subtraction, double (sub.d)
    3. Floating-point multiplication, single (mul.s) and multiplication, double (mul.d)
    4. Floating-point division, single (div.s) and division, double (div.d)
    5. Floating-point comparison, single (c.x.s) and comparison, double (c.x.d), where x may be equal (eq), not equal (neq), less than (lt), less than or equal (le), greater than (gt), or greater than or equal (ge)
    6. Floating-point branch, true (bc1t) and branch, false (bc1f)

- Floating-point comparison sets a bit to true or false, depending on the comparison condition, and a floating-point branch then decides whether or not to branch, depending on the condition.
- The MIPS designers decided to add separate floating-point registers called \$f0, \$f1, \$f2, . . .and used either for single precision or double precision.
- Hence, they included separate loads and stores for floating-point registers: lwc1 and swc1.

Advantages:

1. It has separate register, bandwidth is high.
2. More bits can be stored in the instruction format.

Disadvantages:

1. Floating-point has separated the registers that increase the number of instructions needed to execute a program.
2. It must have separate set of data transfer instructions to move data between floating-point registers and memory.

2.5.5 Accurate Arithmetic

- Floating-point numbers are normally approximations for a number they can't really represent.
- The reason is that an infinite variety of real numbers exists between, say, 0 and 1, but no more than 2^{53} can be represented exactly in double precision floating point.
- The best we can do is get the floating point representation close to the actual number. Thus, IEEE 754 offers several modes of rounding to pick the desired approximation.
- If every intermediate result had to be truncated to the exact number of digits, there would be no opportunity to round. IEEE 754, therefore, always keeps 2 extra bits on the right during intermediate additions, called **guard** and **round**..

Guard:

- The first of two extra bits kept on the right during intermediate calculations of floating-point numbers; used to improve rounding accuracy.

Round:

- Method to make the intermediate floating-point result fit the floating-point format; the goal is typically to find the nearest number that can be represented in the format.

Sticky bit :

- A bit used in rounding in addition to guard and round that is set whenever there are nonzero bits to the right of the round bit.

Fused multiply add:

- A floating-point instruction that performs both a multiply and an add, but rounds only once after the add.

2.6 Sub word Parallelism

- Parallelism will improve the performance of computer. Many application needs the performance high compared to small applications.
- For example consider the graphical and multimedia applications.
- Every processor has its own graphic display, support the graphics operations many transistors are used.
- Many graphic system originally used 8 bit to represent each of the three primary colors plus 8 bit for a location of a pixel.
- The primary colors are RGB- Red, Green, Blue and it has to be represented using 8 bits.
- In addition graphic display, speakers and microphones for teleconferencing. Audio samples need more than 8 bit of precision, but 16 bits are sufficient. Storing bytes and half word in the memory take up less space.
- Every processor has special support to access these data, but due to the infrequency of arithmetic operations on these data sizes in typical integer programs, there were little supports beyond data transfer.
- The rising popularity of multimedia application led to arithmetic instruction that support narrow operation that can easily operate in parallel.
- Many graphics and audio applications will perform the same operation again and again.
- Parallelism is used to perform simultaneous operations on vectors of data used in specific applications.
- Parallelism perform simultaneous arithmetic or logic operation on short vectors (smaller word) of following values:
    1. Sixteen 8 bit operands
    2. Eight 16 bit operands
    3. Four 32 bit operands
    4. Two 64 bit operands.

- The parallelism must occur within a wide word, in case it extends means that is called as sub word parallelism.
- Sub word parallelism is also known as data level parallelism or vector or SIMD parallelism.
- For example, ARM added more than 100 instructions in the NEON multimedia instruction extension to support sub word parallelism.
- These techniques can be used in ARMv7 or ARMv8 processor.
- It added 256 bytes of new register for NEON that can be viewed as
    1. 32 registers 8 byte wide
    2. 16 registers 16 byte wide.
- NEON supports all the sub word data types except 64-bit floating point numbers:
    1. 8-bit, 16-bit, 32-bit, 64-bit signed and unsigned integer
    2. 32-bit floating point number

Data transfer	**Arithmetic**	**Logical/ Compare**
VLDR, F32	VADD,F32,VADD{LW}{S8, U8,S16,U16,S32,U32}	VAND 64, VAND,128
VSTR,F32	VSUB,F32,VSUB{LW}{S8, U8,S16,U16,S32,U32}	VORR 64, VORR,128
VLD{1,2,3,4}{ 18,16,132}	VMUL,F32,VMULL{S8,U8, S16,U16,S32,U32}	VEOR 64, VEOR,128
VST{1,2,3}{18 ,16,132,}	VMLA,F32,VMLAL{S8,U8, S16,U16,S32,U32}	VBIC 64, VBIC,128
VMOV}{18,16 ,132, 32}#imm	VMLS,F32,VMLSL{S8,U8,S 16,U16,S32,U32}	VORN 64, VORN ,128
VMVN}{18,16 ,132, 32}#imm	VMAX,{S8,U8,S16,U16,S32, U32,F32}	VCEQ{18,116,132,F32
VMOV{164,1 28}	VMIN,{S8,U8,S16,U16,S32, U32, F32}	VCGE{S8,U8,S16,U16,S32, U32, F32}
VMVN{164,1 28}	VABS,{ S8,U8, S32, F32} VNEG,{S8,U8,S16,U16,S32,	VCGT{S8,U8,S16,U16,S32, U32, F32}

	S64,U64,}
	VSHL,{S8,U8,S16,U16,S32,
	S64, U64}
	VSHR,{S8,U8,S16,U16,S32,
	S64, U64}
		VCLE{S8,U8,S16,U16,S32,
		U32, F32}
		VCLT{S8,U8,S16,U16,S32,
		U32, F32}
		VTST{ 118,116,1132}

- { } is used to show optional variations of the basic operations.
- {S16, U8, 8} stands for signed and unsigned 8-bit integers.
- {S16, U16, 16} stands for signed and unsigned 16-bit integers.
- {S32, U32, 32} stands for signed and unsigned 32-bit integers.
- {F32} stands for signed and unsigned 32-bit floating point numbers of which 4 fit in a 128 bit register.

Types of Division

1. **Restoring-division**
2. **Non-Restoring-division**

Restoring-division

Algorithm

- Shift *A* and Q left one binary position.
  Subtract *Mt* from *A,* and place the answer back in *A.*
- If the sign of *A* is I, set *qo* to 0 and add *M* back to *A,(I.e restore A);* otherwise set *qo* to 1.

Non-Restoring-division

Algorithm

- If the sign of A is 0, shift A and Q left one binary position and subtract M from A; otherwise, shift A and Q left and add M to A.
- Now, if the sign of A is 0, set q_0 to 0, otherwise set q_0 to 0.
- Finally, If the sign of A is 1, add M to A.

Initially	0 0 0 0 0	1 0 0 0	
	0 0 0 1 1		
Shift	0 0 0 0 1	0 0 0 _	
Subtract	$\underline{1\ 1\ 1\ 0\ 1}$		First Cycle
Set q0	**1** 1 1 1 0		
Restore	$\underline{\quad\ 1\ 1}$		
	0 0 0 0 1	0 0 0 $\underline{0}$	
Shift	0 0 0 1 0	0 0 $\underline{0}$ _	
Subtract	$\underline{1\ 1\ 1\ 0\ 1}$		Second Cycle
Set q0	**1** 1 1 1 1		
Restore	$\underline{\quad\ 1\ 1}$		
	0 0 0 1 0	0 0 $\underline{0}$ $\underline{0}$	
Shift	0 0 1 0 0	0 $\underline{0}$ $\underline{0}$ _	
Subtract	$\underline{1\ 1\ 1\ 0\ 1}$		Third Cycle
Set q0	**0** 0 0 1 0	0 $\underline{0}$ $\underline{0}$ $\underline{1}$	
Shift	0 0 0 1 0	$\underline{0}$ $\underline{0}$ $\underline{1}$ _	
			Fourth Cycle
Subtract	$\underline{1\ 1\ 1\ 0\ 1}$		
Set q0	**1** 1 1 1 1		
Restore	$\underline{\quad\ 1\ 1}$		
	0 0 0 1 0	$\underline{0}$ $\underline{0}$ $\underline{1}$ $\underline{0}$	
	Reminder	Quotient	