CEG 3155: Digital Systems II
(Fall 2023)
Prof. Rami Abielmona
Laboratory #2: *Fixed-Point Arithmetic*

October 01, 2023

1 Objective

The objective of this laboratory is to design and build a fixed-point signed multiplier and divider in VHDL.

Upon completion, the student must be able to:

- Design, realize and test a fixed-point signed multiplier unit;
- Design, realize and test a fixed-point signed divider unit;
- Demonstrate a complete understanding for fixed-point arithmetic.

2 Pre-Lab

Study, implement and simulate in VHDL the adder/subtractor unit presented in this lab (see figure 6). You are taken through the design process in section 3.2, however you will have to perform the coding in RTL and structural VHDL. Refer to section 7 for more guidelines. Show the simulation results of adding and subtracting various number pairs to demonstrate correct functionality of the implementation. Present and explain your work to the TA at the beginning of the session.

3 Introduction to Fixed-Point Arithmetic

Most processors today natively support integer, or **fixed-point**, arithmetic. Some even inherently contain floating-point arithmetic, which is utilized in operations on numbers with fractions, very small numbers, or very large numbers. However, it is important to learn the design process of integer arithmetic before

moving on to floating-point arithmetic. Fixed-point arithmetic is called so, because the *radix point* is fixed and assumed to be to the right of the rightmost digit.

In fixed-point arithmetic, the typical culprits are present: addition, subtraction, multiplication and division. Those operations, along with logical ones (e.g. AND, OR, XOR), compose the **arithmetic logic unit (ALU)** that is at the computational heart of all processors, and in many other digital systems. Addition and subtraction are usually performed using combinational circuits, while multiplication and division are usually realized by sequential circuits. In this lab, we will present the design of an adder/subtractor unit, conceive a design for a multiplier and divider, and eventually realize the arithmetic logic functions of an ALU.

3.1 Review of Number Representations

Numbers can be represented in any base, however we prefer base 2 because it corresponds to the on and off signals utilized inside of a computer. In any number base, the value of the ith digit d is $d * Base^i$. For example, $(10)_2 = (1 * 2^1) + (0 * 2^0) = (2)_{10}$. Note that usually the rightmost bit is the *least significant bit (LSB)*, and the leftmost bit is the *most significant bit (MSB)*.

However, bits are just bits! They have no inherent meaning, and we need to conceive a relationship between bits and numbers. An n-bit number can range from 0 to $2^n - 1$. If the n-bit number is represented as $a_{n-1}a_{n-2}...a_1a_0$, then its unsigned integer value is:

$$A = \sum_{(i=0)}^{(n-1)} 2^i a^i \tag{1}$$

Let us now examine (i.e. review) different ways of representing positive and negative numbers.

3.1.1 Signed-Magnitude

In this representation, we have the following relationships between bits and numbers:

$$A = \sum_{(i=0)}^{(n-2)} 2^i a^i \tag{2}$$

$$A = -\sum_{(i=0)}^{(n-2)} 2^i a^i \tag{3}$$

Equation 2 is used if $a^{n-1} = 0$, while equation 3 is used if $a^{n-1} = 1$. Examples of signed magnitude 3-bit numbers follow:

000 = +0; 001 = +1; 010 = +2; 011 = +3
100 = -0; 101 = -1; 110 = -2; 111 = -3

The MSB is used as the sign bit, and the rest of the bits are used as the magnitude of the number. Problems arise with this type of representation for the following reasons:

- Where do we place the sign bit ?
- An extra step is necessary to update the sign bit
- Both a +0 and a -0 are represented

3.1.2 1's Complement

In this representation, we have the following relationships between bits and numbers:

$$A = -2^{n-1}a^{n-1} + \sum_{(i=0)}^{(n-2)} 2^i a^i \tag{4}$$

$$A = -2^{n-1}a^{n-1} + 1 + \sum_{(i=0)}^{(n-2)} 2^i a^i \tag{5}$$

Equation 4 is used if $a^{n-1} = 0$, while equation 5 is used if $a^{`n-1} = 1$. Examples of 1's complement 3-bit numbers follow:

000 = +0; 001 = +1; 010 = +2; 011 = +3
100 = -3; 101 = -2; 110 = -1; 111 = -0

A negative number is represented by the binary inversion of its positive number representation. Problems arise with this type of representation for the following reasons:

- Adders require an extra step to subtract a number
- Both a +0 and a -0 are represented

3.1.3 2's Complement

In this representation, we have the following relationships between bits and numbers:

$$A = -2^{n-1}a^{n-1} + \sum_{(i=0)}^{(n-2)} 2^i a^i \tag{6}$$

Equation 6 is used if $a^{n-1} = 0$ or if $a^{n-1} = 1$. Examples of 2's complement 3-bit numbers follow:

000 = +0; 001 = +1; 010 = +2; 011 = +3
100 = -4; 101 = -3; 110 = -2; 111 = -1

A negative number is represented by the binary inversion, then an addition of 1, of its positive number representation. This is the best type of representation for the following reasons:

- Operations are easily performed on a digital computer
- No need exists to update the sign bit
- Only one representation for zero (helps in test instructions)

2's complement number representation functions because the unsigned sum of an n-bit number and its negative is 2^n, hence the complement of a 2's complement number x is $2^n - x$. For this, and the reasons listed above, 2's complement has been the overwhelming choice of ALUs since 1985. Other notables of 2's complement notation is its *range*: -2^{n-1} to $2^{n-1}-1$, its *expansion of bit length*: "Add additional bits to the left and fill in with the value of the original sign bit", its *addition overflow rule*: "If two numbers with the same sign bit (both positive or both negative) are added, then overflow occurs if and only if the result has the opposite sign", its *subtraction rule*: "To subtract B from A, take the 2's complement of B and add it to A (e.g. $A - B = A + (-B)$)" and its *subtraction overflow rule*: "Overflow occurs when subtracting a negative number from a positive number and the result is negative, or when subtracting a positive number from a negative number and the result is positive".

3.2 Fixed-Point Addition

Binary addition is similar to decimal addition, except that we are adding 1s and 0s together. The addition of only two bits can be realized using a half-adder, which is a combinational circuit that accepts two binary digits and generates a sum bit and a carry-out bit. The half adder is usually called a (2,2) adder, as it does not have a carry input. The truth table of the 1-bit half adder is shown in Table 1.

a_i	b_i	s_i	c_i
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Table 1: Truth table for a 1-bit half adder

The Karnaugh-map-derived Boolean expressions for the sum and carry output signals can be extracted from the truth table as follows:

$$s_i = \overline{a_i} b_i + a_i \overline{b_i} = a_i \oplus b_i$$
$$c_i = a_i b_i$$

The resulting logic implementations for the sum and carry outputs are shown in figure 1.

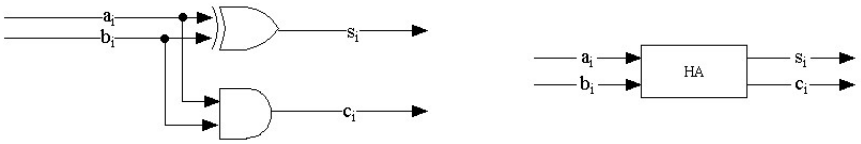


Figure 1: Possible realization of a one-bit half adder

The limitation of the half adder is that it does not accept a carry input, and hence cannot be used in multibit addition which requires a bit from a previous stage to be taken into account: enter the *full-adder*. Also known as the (3,2) adder because it has 3 inputs and 2 outputs, the full adder's truth table is shown in Table 2.

c_i	a_i	b_i	s_i	c_{i+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Table 2: Truth table for a 1-bit full adder

The Karnaugh-map-derived Boolean expressions for the sum and carry output signals can be extracted from the truth table as follows:

$$s_i = \overline{a_i}\,\overline{b_i} c_i + \overline{a_i} b_i \overline{c_i} + a_i \overline{b_i}\,\overline{c_i} + a_i b_i c_i$$
$$c_{i+1} = \overline{a_i} b_i c_i + a_i \overline{b_i} c_i + a_i b_i \overline{c_i} + a_i b_i c_i$$

These expressions can be reduced to the following:

$$s_i = a_i \oplus b_i \oplus c_i$$
$$c_{i+1} = a_i b_i + b_i c_i + a_i c_i$$

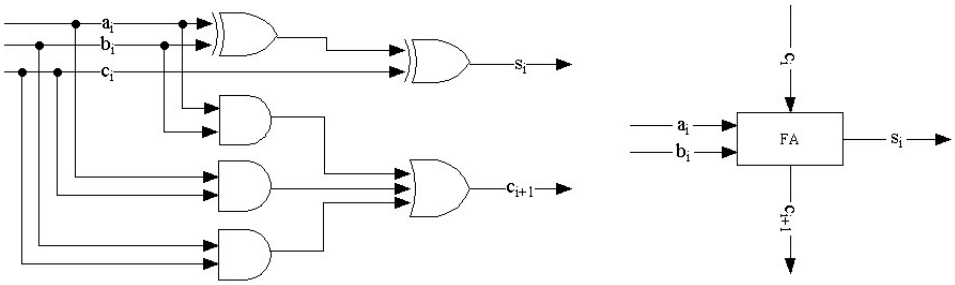


Figure 2: Possible realization of a one-bit full adder

The resulting logic implementations for the sum and carry outputs are shown in figure 2.

Since digital systems design advises to try and re-use efficient and functional circuits in the synthesis of larger more complex circuits, figure 3 (extracted from chapter 5 of our textbook) shows a decomposed full adder circuit, constructed using two half adder components.

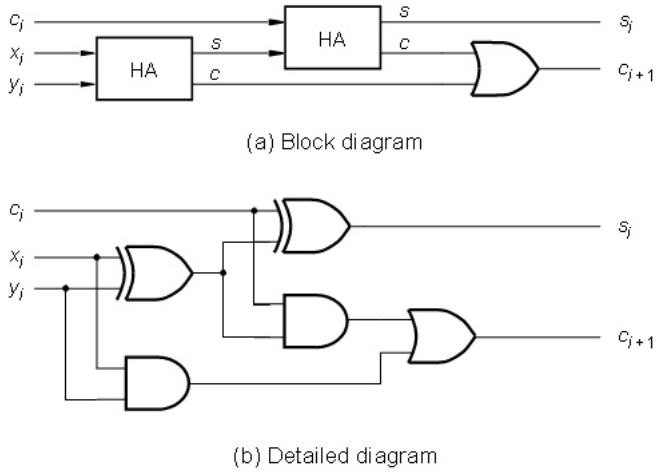


Figure 3: Decomposed full adder circuit realization

3.2.1 Ripple-Carry Adder

In order to perform n-bit addition, one option is to connect the 1-bit full adders designed in the previous section in a cascaded chain of n stages. Hence, for 8-bit addition, we require 8 FA blocks connected as shown in figure 4 (figure 5.6 in the textbook). Note that the carry output signal created on the LSB (c_1) can

ripple all the way to the MSB (c_n). This type of formation causes the structure to be called a *ripple carry adder*. The delay caused by the ripple enforces that this type of adder is not the fastest type, but it is the simplest to construct.

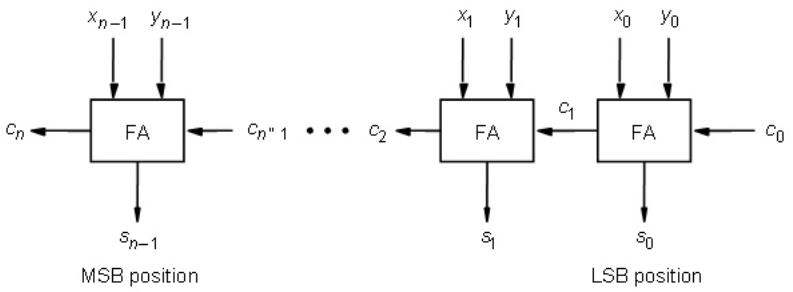


Figure 4: A n-bit ripple carry adder realization

3.3 Fixed-Point Subtraction

Instead of designing half subtractors, full subtractors and ripple borrow subtractors, which is very similar to the addition design process that we followed in section 3.2, we will conceive a subtractor circuit, utilizing the just-designed ripple carry adder. First, it is important to remember the subtraction rule defined towards the end of section 3.1.3, which says that to perform 2's complement subtraction $a-b$, one actually performs $a+(-b)$, while utilizing the 2's complement form of $-b$. Now b and its complement $\overline{b}$ can be provided in many ways, two of which are shown in figure 5. The addition, required to complete the 2's complement conversion, is provided by setting the $bInvert$ or $\overline{Add}/Sub$ control lines to '1'. Hence, we have

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

And that is a big reason why 2's complement number representation is so popular!

Extending the operation to n-bit addition/subtraction, we obtain the adder/subtractor unit shown in figure 6 (figure 5.13 in textbook), which inherently operates on signed numbers. If adding two numbers, we set the $\overline{Add}/Sub$ control line to '0', ensuring that the true (uncomplemented) form of the addend bits are passed through the XOR gates, and the LSB carry-in is zero, whereas if subtracting two numbers, we set the $\overline{Add}/Sub$ control line to '1', ensuring that the complemented form of the addend bits are passed through the XOR gates, and the LSB carry-in is one (in order to perform the addition of 1 required in 2's complement numbers).

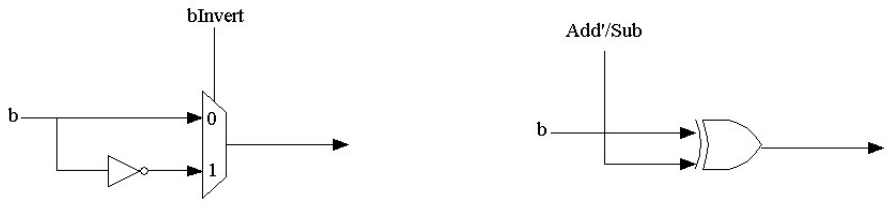


Figure 5: Possible realization of b and $\overline{b}$

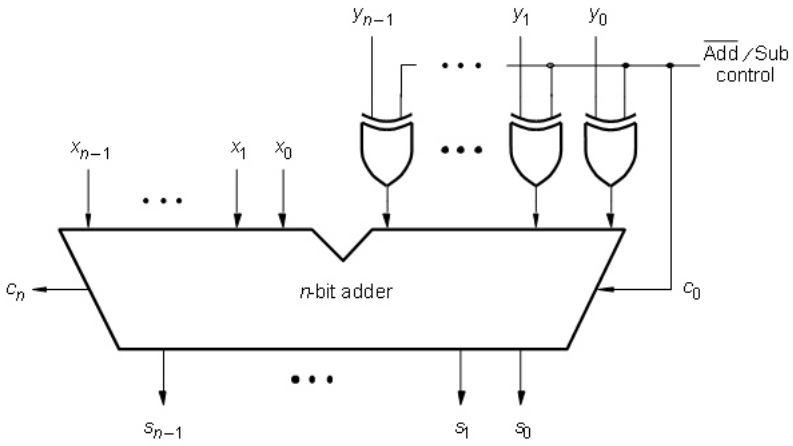
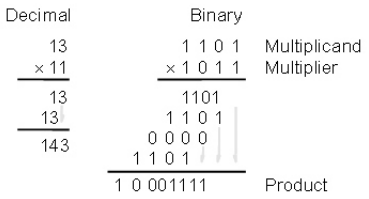


Figure 6: A n-bit adder/subtractor unit realization

Students are referred to chapter 5 of their textbook for a more detailed discussion of number representations and arithmetic circuits, including arithmetic overflows and fast adder realizations (e.g. carry-lookahead adders).

4 Fixed-Point Multiplication

Binary fixed-point multiplication illustrates the design obstacles faced when designing multiplier circuits. There are numerous techniques to accomplish the task, and the choice depends on factors such as speed, area or both. For example, an *array multiplier* is presented in section 5.6.1 of your textbook, but it is too costly in terms of logic area, when a high number of bits are multiplied. For that reason, a *shift-and-add multiplier* is presented in section 10.2.3, which we will be discussing below. Figure 7 (figure 10.15 in the textbook), shows a multiplier algorithm that could be used in the implementation of a shift-and-add multiplier circuit, where A is the multiplicand, B is the multiplier, and P is the product.



(a) Manual method

$P = 0$;
for $i = 0$ to $n - 1$ do
    if $b_i = 1$ then
        $P = P + A$;
    end if;
    Left-shift A ;
end for;

(b) Pseudo-code

Figure 7: A multiplier algorithm

As described in the book:

> The product is formed by a series of addition operations. For each bit i in the multiplier that is 1, we add to the product the value of the multiplicand shifted to the left i times.

Figure 12 (towards the end of this lab) shows a flowchart for the multiplication of two unsigned numbers. In this case, we are assuming that we are multiplying only positive numbers. If the **multiplicand** (usually the first number) or the **multiplier** (usually the second number) is negative, then it is converted to positive and then multiplied. The product is then negated if signs of the operands are opposite. Figure 8 represents one possible realization of a 32-bit shift-and-add multiplier. It is important to note that, in this implementation, since both the multiplicand and multiplier are 32-bits wide, then the product is 64-bits wide, which will require the use of both a 64-bit ALU and a 64-bit multiplicand register (Question: can you think of other ways to reduce the logic and/or routing resources ?). The algorithm starts by setting the product to zero, and the control circuit decides when to shift the multiplicand and multiplier, and when to write (hence, add) the product.

Figures 10.16-10.18 of your textbook take you through the ASM method for the design and implementation of a shift-and-add multiplier. These figures are shown in section 10.2.3 of your textbook, so the student is referred to that section for more detail on the realization, VHDL behavioral implementation, as well as simulation, of the multiplier circuit.

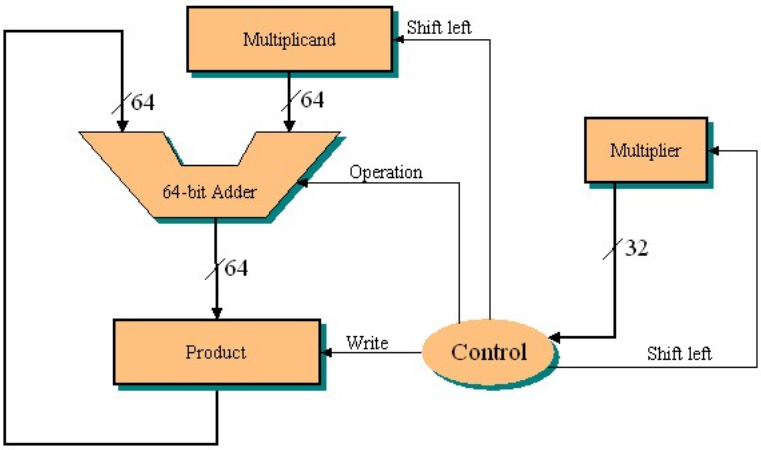


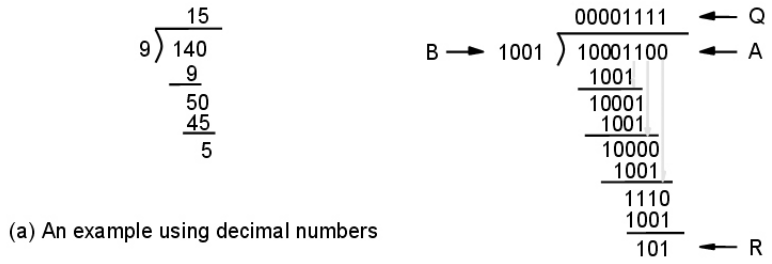
Figure 8: A shift-and-add multiplier circuit realization

5 Fixed-Point Division

Binary fixed-point division illustrates the design obstacles faced when designing divider circuits. There are numerous techniques to accomplish the task, and the choice, again, depends on factors such as speed, area or both. Figure 9 (figure 10.21 in the textbook), shows a divider algorithm that could be used in the implementation of a divider circuit, where A and B are the numbers to divide, Q is the quotient and R is the remainder.
As described in the book:

> `Shift the bits in` A `to the left, one bit at a time, into a shift register` R`. After each shift operation, we compare` R `with` B`. If` $R \geq B$`, a 1 is placed in the appropriate bit position in the quotient and` B `is subtracted from` R`. Otherwise, a 0 bit is placed in the quotient.`

Figure 13 (towards the end of this lab) shows a flowchart for the division of two unsigned numbers. In this case, we are assuming that we are dividing only positive numbers. If the **dividend** (usually the first number) or the **divisor** (usually the second number) is negative, then it is converted to positive and then divided. The quotient is then negated if the signs of the operands are opposite. Figure 10 represents one possible realization of a 32-bit divider. It is important to note that, in this implementation, since both the dividend and the divisor are 32-bits wide, then the quotient is 32-bits wide, but the running remainder register is 64-bits wide (although the remainder is actually 32-bits), which will require the use of both a 64-bit ALU and a 64-bit divisor register, the latter because of the shift right operation (Question: can you think of other ways to



$R = 0$;
for $i = 0$ to $n - 1$ do
    Left-shift $R||A$;
    if $R \geq B$ then
        $q_i = 1$;
        $R = R - B$;
    else
        $q_i = 0$;
    end if;
end for;

(c) Pseudo-code

Figure 9: A divider algorithm

reduce the logic and/or routing resources ?). The algorithm starts by setting the remainder register to the dividend value, and the control circuit decides when to shift the divisor and quotient registers, and when to write (hence, subtract from) the remainder.

Figures 10.22-10.27 of your textbook take you through the ASM method for the design and implementation of a divider. These figures are shown in section 10.2.4 of your textbook, so the student is referred to that section for more detail on the realization, VHDL behavioral implementation, as well as simulation, of the divider circuit.

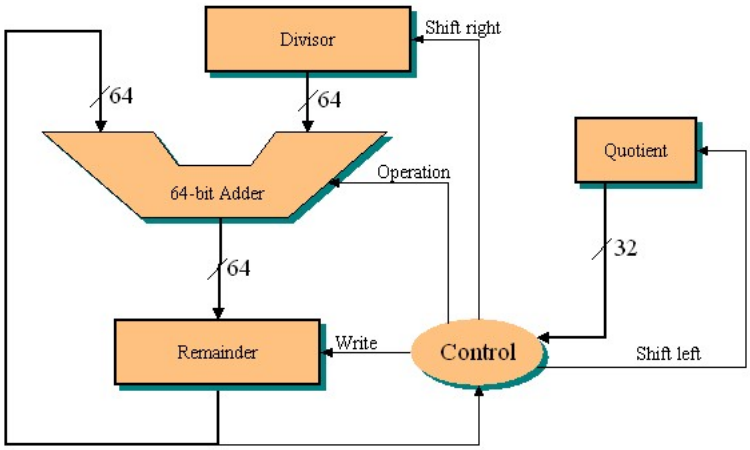


Figure 10: A divider circuit realization

6 Laboratory

In this lab, you will, utilizing the five steps of the ASM method, first proceed to implement the **adder/subtractor circuit** (in the pre-lab) shown in figure 6, then design and implement a **signed multiplier and divider**. Together, all four operations will be integrated in a fixed-point ALU that allows the addition, subtraction, multiplication and division of signed numbers. **Note that the assumption that only positive numbers are input does not hold, hence you may not use the technique of converting negative numbers to positive ones and then multiplying/dividying them**. The input/output specifications of the fixed-point ALU is shown in Table 3. We will be working with **4-bit arithmetic**, hence all operations are 4-bit ones, so as to allow us to easily view the results on the BCD 7-segment decoders. Note that some registers will have to be 8-bits wide, such as the product register for example. Three status signals are required at the output: $CarryOut$ (indicating that there is a carry at the end of the ripple carry adder), $ZeroOut$ (indicating that the current ALU result is equal to zero) and $OverflowOut$ (indicating that the current ALU operation caused an arithmetic overflow as shown in section 5.3.5 of the textbook).

You will have to first come up with the pseudocode that resolves the problems at hand (in this case, the fixed-point signed multiplier and divider). Using the obtained pseudocode, show all of the following for each of the signed multiplier and signed divider: the ASM chart (in graphical form), the datapath, the detailed ASM chart (in graphical form) and finally the control logic using the one-FF-per-state method. After completing the ASM steps, you have to design, realize and test the synchronous digital circuits of the datapath and the control

logic, using the Quartus II software and the DE-2 board from Altera (for more details, see section 7).

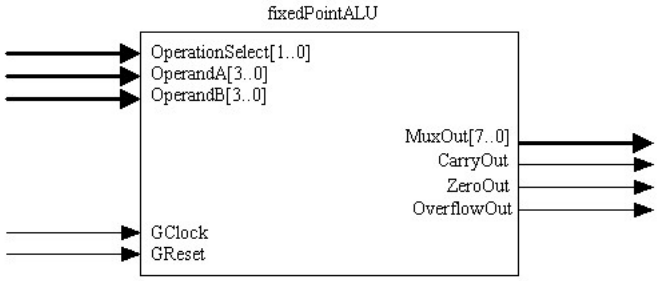


Figure 11: 4-bit fixed-point ALU

Port Type	*Name*	*Description*
Input	GClock	Global clock needed to synchronize the circuitry
Input	GReset	Global reset needed to bring the internals to known states
Input	OperandA[3..0]	Operand A 4-bit input
Input	OperandB[3..0]	Operand B 4-bit input
Input	OperationSelect[1..0]	Selects which arithmetic operation to perform
Output	MuxOut[7..0]	Multiplexer output controlled by OperationSelect[1..0]
Output	CarryOut	The carry (add) or borrow (sub) out status signal
Output	ZeroOut	The zero status signal indicating that the ALU result is zero
Output	OverflowOut	The overflow status signal indicating arithmetic overflow

Table 3: Input/Output Specification

The MuxOut[7..0] and OperationSelect[1..0] output and input are intertwined in the manner shown in Table 4.

6.1 Bonus: CLA instead of RCA

For an **extra 5 marks**, modify your fixed-point ALU design to utilize a 4-bit carry-lookeahead adder (CLA) instead of the ripple carry adder implemented in this lab. You can refer to section 5.4.1 of your textbook for a more detailed discussion on the design of CLAs. Do not show all the components again, but instead, discuss the modifications performed on the components in order to run the modified ALU. If the datapath and control logic have changed, present and discuss the modifications. You have to demonstrate your new design to your TA.

OperationSelect[1..0]	*MuxOut[7..0]*	*Description*
00	('0', '0', '0', '0', SumOut[3..0])	4-bit sum
01	('0', '0', '0', '0', DifferenceOut[3..0])	4-bit difference
10	ProductOut[7..0]	8-bit product
11	(RemainderOut[3..0], QuotientOut[3..0])	4-bit quotient & remainder

Table 4: Output Multiplexer Selection

7 Design Restrictions

- Verilog implementations will not be accepted. Perform all implementations in VHDL code only
- Behavioral level of modeling will not be accepted. Design should be done at the structural level of modeling
- Register Transfer Logic (RTL) design and coding is mandatory
- Use graphical design for the top-level entity, and use your judgement for any other sub-blocks. However, all atomic modules have to be implemented in VHDL (i.e. D flip-flop, 1-bit adder, 1-bit comparator and so on)
- No core instantiations are allowed (i.e. LPMs from Altera or free IP cores from the Internet). All building blocks have to be designed and realized by the group
- The top level entity is given in input/output specification format, but the internals are left up to the group
- The design has to be synchronous and globally reset-able. This means that global clock and reset signals are required in all functional blocks
- Simulate both designs and check your simulation results with your theoretical ones (e.g. add, subtract, multiply and divide two numbers)
- Download the design to the Cyclone chip on your DE-2 boards, and use the on-board switches to input OperationSelect[1..0], and the two 7-segment BCD decoders to demonstrate correct output functionality (MuxOutput[7..0] and three LEDs for the control signals shown in figure 11)
- Each group must demonstrate a working version of the laboratory to the TA, before the due date of the report

8 Report Reminders

- Include timing simulations with explanation for all VHDL source files

- Describe and comment all your VHDL source files
- Include a flowchart representation of your solution to the problem
- Include a block diagram of your solution to the problem
- If using ASM design, include all appropriate charts and paths (control and data)
- If using FSM design, include the state diagram with all appropriate inputs and outputs
- Describe, in your own words, your solution to the problem
- Describe your design obstacles and how they were overcome
- Append all VHDL source code and graphical design files to your report
- Submit a soft copy of all VHDL and graphical design files with your report

9 Acknowledgements

Figures 3, 4, 6, 7 and 9 are taken out of our course textbook: *Fundamentals of Digital Logic with VHDL Design* by Stephen Brown and Zvonko Vranesic. Figures 12 and 13 are taken out of *Computer Organization and Design, The Hardware/Software Interface* by David A. Patterson and John L. Hennessy.

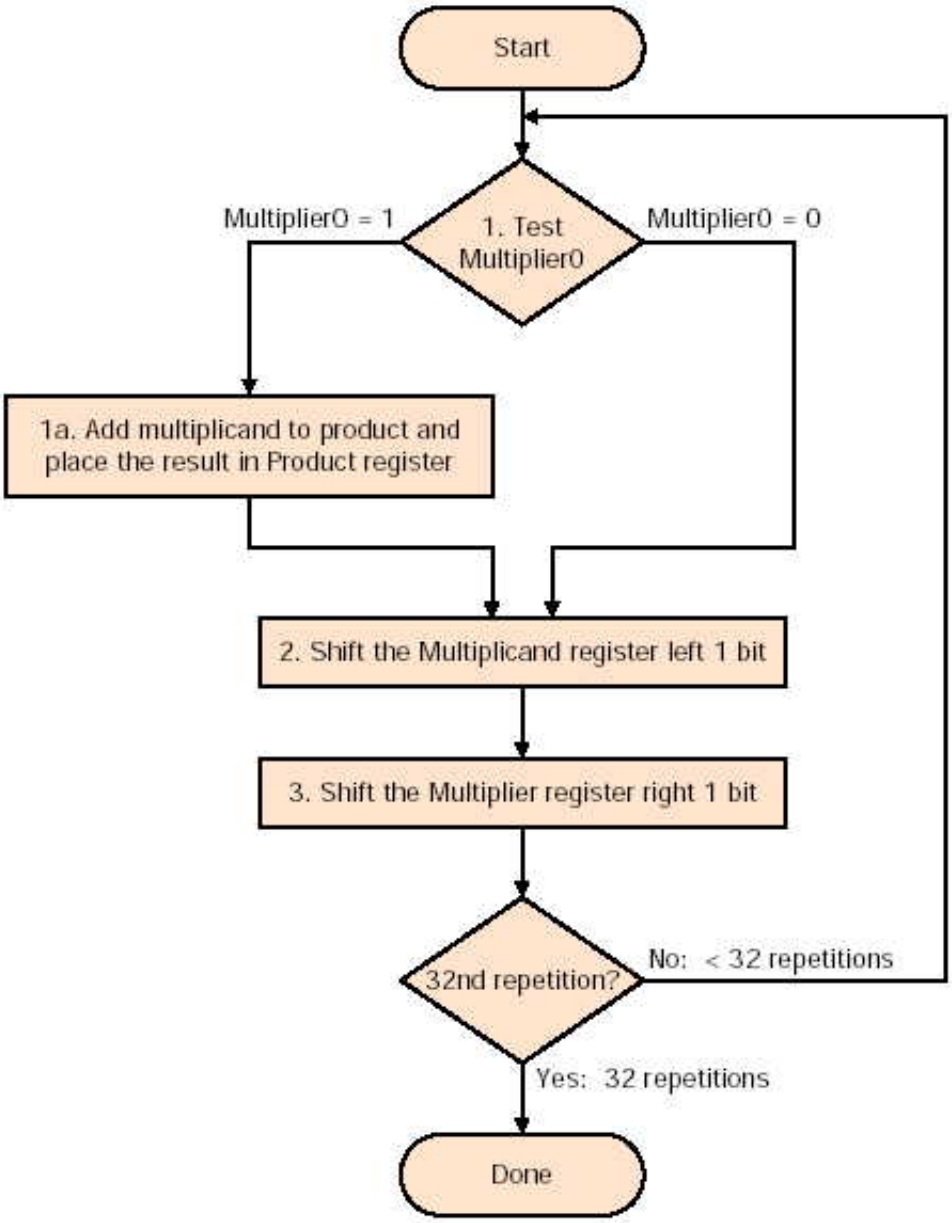


Figure 12: A multiplication operation flowchart

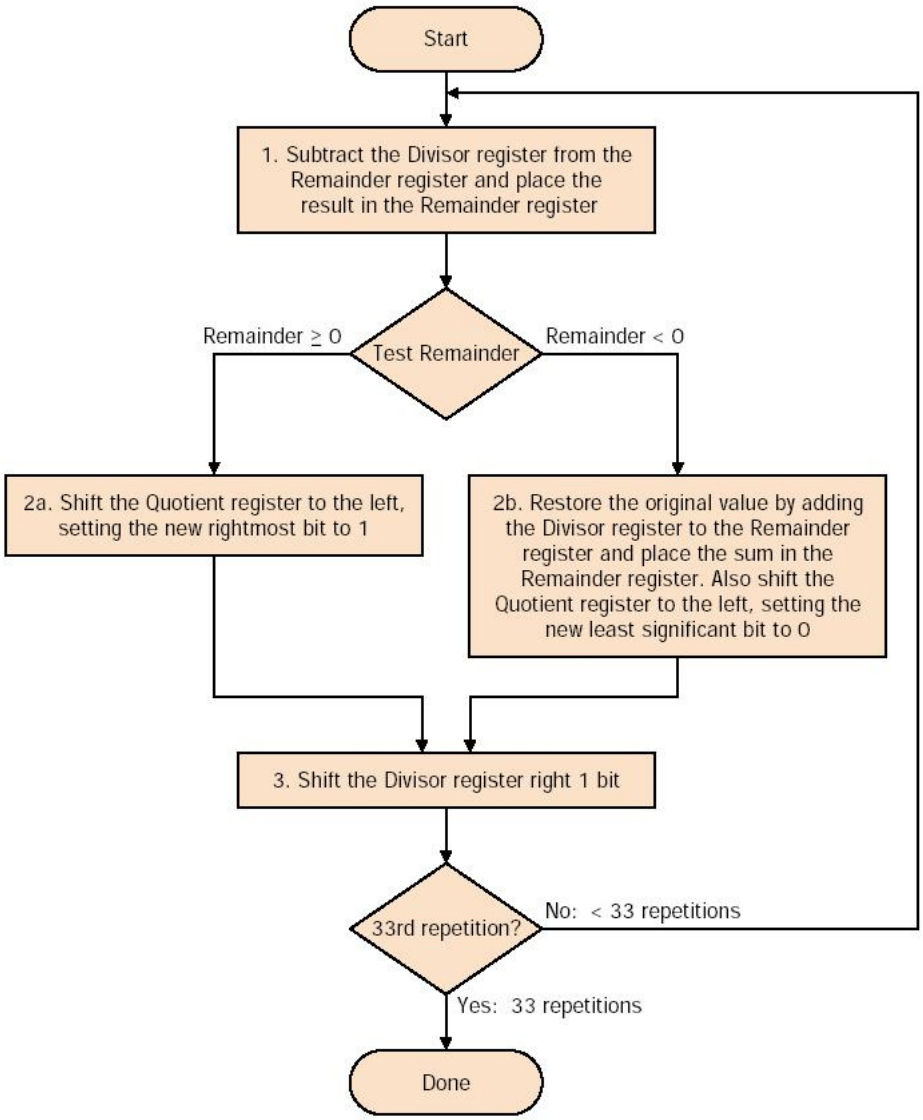


Figure 13: A division operation flowchart