Chip Design 1

Specification of Calculator Project
Part 4: Arithmetic Logic Unit

Version:	1.0
Author:	R. Höller

Copyright Notice

Introduction

This document describes the arithmetic logic unit “alu” which processes the arithmetic/logic operations that are supported by the calculator. The operands OP1 and OP2 as well as the selected arithmetic/logic operation are provided by the calculator control unit. The ALU performs the selected operation and sends back the result as well as any special conditions (calculation error, overflow …) to the calculator control unit. See the distance learning letter “Overview of Calculator Project” for a list of all ports of the ALU. It is also worth having a look at the block diagram again. Finally, it **is very important** that the ALU follows the interface timing shown in the distance learning letter “Calculator Control Unit”!

Arithmetic/Logic Operations

The distance learning letter “Overview of Calculator Project” shows all the arithmetic/logic operations that are supported by the calculator. Both operands “op1_i(11:0)” and “op2_i(11:0)” are treated as unsigned integer values with a data width of 12 bits for all operations. However, you do not have to implement all operations but only the operations that are given by Table 2 of the distance learning letter “Overview of Calculator Project”! If a calculation was started over signal “start_i” while “optype_i(3:0)” selects an operation that is not supported, the ALU forces “error_o” to logic 1 and terminates the calculation by generating a pulse of a single clock cycle on signal “finished_o” (see interface timing in the distance learning letter “Calculator Control Unit”).

Try to implement each single operation as a block of code. A multiplexer selects the appropriate signals “finished_...“, “result(15:0)_...”, “sign_...”, “error_...” and “overflow_...” from all blocks depending on the operation given by “optype_i(3:0)”, and forwards them to the calculator control unit, as shown in Figure 1.

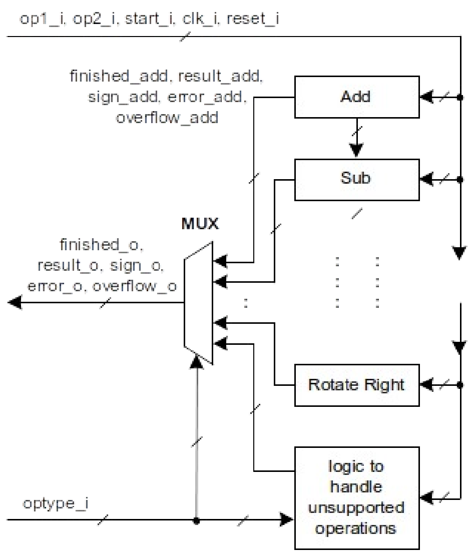
op1_i, op2_i, start_i, clk_i, reset_i
finished_add, result_add, sign_add, error_add, overflow_add
Add
Sub
MUX
finished_o, result_o, sign_o, error_o, overflow_o
Rotate Right
logic to handle unsupported operations
optype_i

Figure 1: ALU Block Diagram

The operations “Add”, “Logical NOT”, “Logical AND”, “Logical OR”, “Logical Ex-OR”, “Rotate Left” and “Rotate Right” can be implemented by using information found in the various lecture slides and distance learning letters of the “Digital System Design” course. Do not forget to generate the pulse on signal “finished_...” when the calculation is finished. Also, signals “error_...”, “overflow_...” and “sign_...” must be handled properly (however, no calculation error or overflow condition may occur and the result cannot be a negative number for these operations).

Implementation details for the remaining arithmetic operations are described in the following.

Operation “Sub”

A subtraction can be performed by using the “-“ operator which is, for example, found in a number of IEEE VHDL packages. However, negative numbers are represented in the two’s complement form and must be converted to absolute values before being forwarded to the calculator control unit. Thus, two cases have to be distinguished:

- Perform the operation “op1_i(11:0) - op2_i(11:0)”. If the result is a positive number, it can be directly forwarded to “result(15:0)_sub” while signal “sign_sub” has to be set to logic 0.
- If “op1_i(11:0) - op2_i(11:0)” is a negative number, you have to (i) invert all bits of the result and (ii) add “1” afterwards which gives the absolute value that can be forwarded to “result_sub(15:0)”. Finally, set “sign_sub” to logic 1.

Do not forget to handle the signals “finished_sub”, “error_sub” and “overflow_sub” (however, no calculation error or overflow condition can occur for the “sub” operation).

Operation “Multiply”

The operation “OP1 * OP2” shall be implemented in the calculator by accumulating OP1 for a number of OP2 times:

$$OP1 * OP2 = \begin{cases} 0, OP2 = 0 \\ \sum_{1}^{OP2} OP1, OP2 > 0 \end{cases}$$

Figure 2 shows a circuitry composed of an adder, a register and a “down counter” which performs the given algorithm. At the begin of the operation (start_i = ’1’), the register is set to zero and the down counter is initialized to “op2_i(11:0)”. Afterwards, the down counter decrements and thus, “op1_i(11:0)” is accumulated with every clock cycle until the down counter reaches zero. Note, that you have to add an additional circuitry to Figure 2 which handles the “finished_mul” signal. Moreover, signal “overflow_mul” must be set to logic 1 once the result exceeds 16 bits. Finally, do not forget to handle the signals “error_mul” and “sign_mul” (however, no negative numbers or calculation errors can occur here).

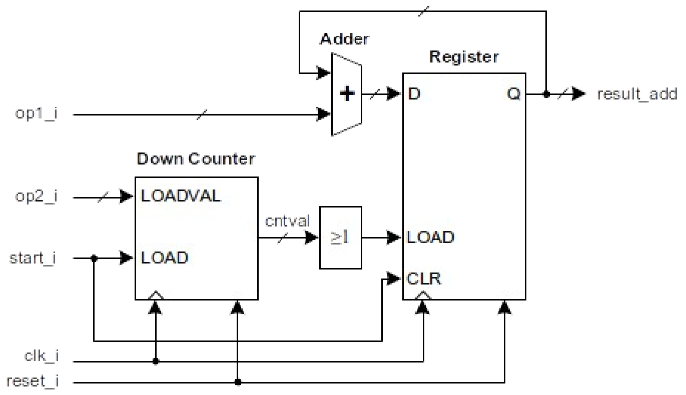

Figure 2: Implementation of Multiply Operation

Note, that other methods exist for implementing the multiply operation using digital circuits. However, the proposed method has the advantage of being simple, consumes less combinatorial resources than, e.g., a pure combinatorial implementation and does not need any technology-dependent resources (thus, it can be ported easily to other FPGA technologies).

Operations “Divide” and “Remainder of Division”

The operation “OP1 / OP2” shall be implemented in the calculator by determining how many times OP2 can be substracted from OP1 before the result becomes negative:

$$\frac{OP1}{OP2}=\begin{cases} undefined, OP2=0 \\ \frac{r}{OP2}, OP2>0, q=0 \\ OP1-\sum_{1}^{q} OP2+\frac{r}{OP2}, OP2>0, q>0 \end{cases}$$

In the expression shown above, “q” should be the integer fraction of the division (quotient) while “r” denotes the remainder.

Figure 3 shows a circuitry composed of a subtractor, a register and an “up counter” which performs the given algorithm. At the begin of the operation (start_i = ’1’), the register is loaded with “op1_i(11:0)”. At the same time, the counter is initialized to zero and will be prepared to start counting in the next clock cycle. Afterwards, the counter increments and the register is loaded with [current register value - op2_i(11:0)] with every clock cycle until the result of the substractor becomes negative (borrow bit of subtractor = logic 1). At this point of time, the operation is finished: The output of the counter shows the integer fraction of the division while the output of the register provides the remainder. Note, that you have to add an additional circuitry to Figure 3 which handles the “finished_...” as well as the “error_...” signal (in the case “division by zero”). Moreover, do not forget to handle the signals “overflow_...” and “sign_...” (however, no overflow or negative numbers may occur during the operation).

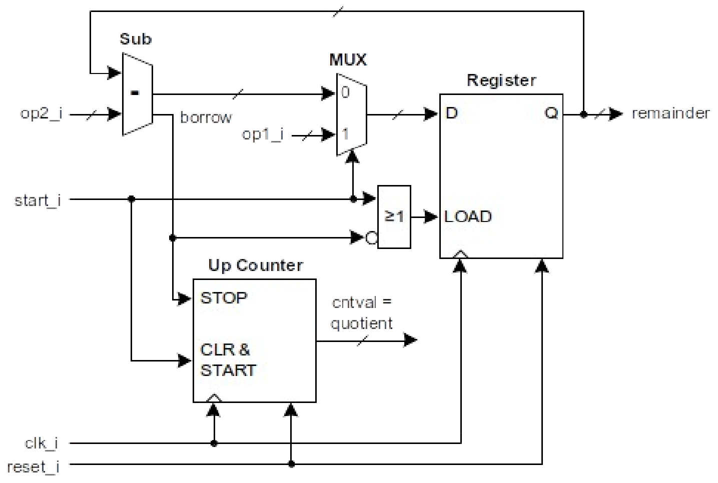

Figure 3: Implementation of Divide Operation

Note, that other methods exist for implementing the divide operation using digital circuits. However, the proposed method has the advantage of being simple.

Operation “Square”

For implementing the square operation we will make use of the simple fact that

$$OP1^2 = OP1 * OP1$$

Therefore, implement the multiply operation as described previously and connect the load value input of the “down counter” to “op1_i(11:0)” rather than to “op2_i(11:0)” (Figure 2). The value of signal “op2_i(11:0)” will be ignored when performing the square operation.

Operation “Square Root”

The proposed implementation of the square root operation is based on an algorithm from the German scientist August Toepler that was invented in the 19th century for the earliest existing mechanical calculators. The algorithm is described in an article entitled „Prof. Toepler‘s Verfahren der Wurzelausziehung mittelst der Thomas‘schen Rechenmaschine“, published in „Polytechnisches Journal, Vierte Reihe, 29. Band, Jahrgang 1866, Augsburg, Verlag J. G. Cotta“:

> *„Den Besitzern und Freunden der Thomas‘schen Rechenmaschine dürfte die Mittheilung eines Verfahrens interessant seyn, welches Prof. Dr. Toepler in Riga für die Ausziehung von Quadratwurzeln mittelst des Arithmometers ersonnen hat und welches vermöge seiner Einfachheit als eine glückliche Bereicherung der theoretischen Hülfsmittel des Maschinen-Rechnens willkommen zu heißen ist.“*

The algorithm works as follows: Subtract all odd numbers (starting with 1) from the radicant until the result becomes negative. The number of subtractions (until the result becomes negative) equals to the integer fraction of the square root. Here is an example:

```
Radicant:  27
         -  1 ⎫
           26 ⎪
         -  3 ⎪
           23 ⎪
         -  5 ⎬ number of subtractions = 5 = integer fraction of
           18 ⎪
         -  7 ⎪
           11 ⎪
         -  9 ⎭
            2
         - 11
           -9 => negative
```

Figure 4 shows a circuitry composed of a subtractor, a register and a counter which performs the given algorithm. At the begin of the operation (start_i = ’1’), the register is loaded with “op1_i(11:0)” which defines the radicant. At the same time, the counter (signal “cntval(n:0)”) is initialized to 1. Starting with the next clock cycle the counter increments by two until the result of the substractor becomes negative (borrow bit of subtractor = logic 1). That way, the counter generates a sequence of odd numbers (1, 3, 5, 7, 9 ...). Once the result of the substractor becomes negative the counter stops and the operation is finished. The integer fraction of $\sqrt{op1\_i(11:0)}$ is equal to the number of subtractions which, in turn, equals to the integer fraction of the last counter value divided by two. In other words, the result is given by signal “cntval(n:1)”.

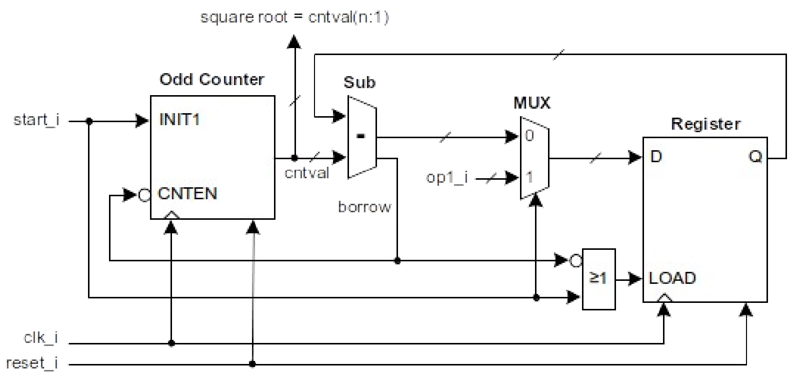

Figure 4: Implementation of Square Root Operation

Note, that you have to add an additional circuitry to Figure 4 which handles the “finished_sro” signal. Moreover, do not forget to handle the signals “error_sro”, “overflow_sro” and “sign_sro” (however, no overflow, calculation errors or negative numbers may occur during the operation). The value of signal “op2_i(11:0)” will be ignored when performing the square root operation.

Operation “Binary Logarithm”

The binary logarithm (“Logarithmus Dualis”) is the logarithm to the base 2

$$lb\,x = \log_2 x$$

In binary arithmetic, the integer fraction of the binary logarithm can be interpreted as the number of bits that are needed to represent a number. Thus, we simply have to find the bit position of the leftmost ‘1’ in a binary number. Here is an example:

	11	10	9	8	7	6	5	4	3	2	1	0
op1_i(11:0)	0	0	0	1	0	1	1	1	0	1	1	0

Since the bit position of the leftmost ‘1’ in “op1_i(11:0)” is 8, the integer fraction of the binary logarithm is 8 in this example.

Finding out the bit position of “op1_i(11:0)” can be done by a combinatorial decoder, for example, by using a “for”-loop in VHDL. Do not forget to handle the “finished_lb”, “error_lb”, “overflow_lb” and “sign_lb” signals (however, no overflow or negative numbers may occur during the operation). The value of signal “op2_i(11:0)” will be ignored when performing the binary logarithm operation.

How to Proceed?

The next steps in the project are as follows:

- Write a VHDL entity for the ALU, name the file, for example, “alu_.vhd” and store it in the “vhdl” sub-folder of your project directory. The entity ports can be found in the distance learning letter “Overview of Calculator Project”. Have a look at the block diagram (which is also included in the distance learning letter “Overview of Calculator Project”) to understand, how the ALU communicates with other units in the design. It is also very important that the ALU follows the interface timing shown in the distance learning letter “Calculator Control Unit” when communicating with the calculator control unit!
- Write a VHDL architecture for the ALU, name the file, for example, “alu_rtl.vhd” and store it in the “vhdl” sub-folder of your project directory. Break down the functionality into smaller sub-blocks (adder, divider, logic to perform “rotate left” operation … result multiplexer) and decide which of these blocks need to be coded as combinatorial logic and which of them need storage elements (registers). You can also create VHDL sub-components for certain pieces of functionality but since the complexity of the ALU is not that high, it is rather recommended that you include everything that is described in this document in a single architecture.
- Create a VHDL configuration if you like to, but this is completely optional!
- Create a VHDL entity/architecture pair (and an optional configuration) for the testbench of the ALU, name the files, for example, “tb_alu_.vhd” and “tb_alu_sim.vhd” and store them in the “tb” sub-folder of your project directory.

- Write “do”-scripts to compile and simulate the ALU as described in the distance learning letter “Introduction to ModelSim-Intel FPGA Starter Edition” and store them in the “sim” sub-folder of your project directory.
- Simulate the ALU using ModelSim and fix all bugs that you find. Do not forget to identify and special cases (result of an operation becomes negative, result exceeds 16 bits, division by zero, etc.)!
- If the ALU was tested successfully, proceed with the distance learning letter “Part 5: Top-level Design”.

Version

Version 1.0	Update to entire document for CHIP1

If you find mistakes or inconsistencies, please report them to the course supervisors via email. Thank you!