CONCORDIA UNIVERSITY
Department of Electrical and Computer Engineering
COEN 316 Computer Architecture
Lab 1
Arithmetic and Logic Unit
Fall 2021

Introduction

In this lab, a 32-bit ALU will be designed in VHDL. The ALU will be used as a VHDL component in the CPU which will be designed during the 5 labs. Subsequent labs will provide more details of the CPU. The following VHDL entity specification is to be used for the design of the ALU:

```
library IEEE;
use IEEE.std_logic_1164.all;


entity alu is
port(x, y : in std_logic_vector(31 downto 0);
     -- two input operands
     add_sub : in std_logic ;   -- 0 = add , 1 = sub
     logic_func : in std_logic_vector(1 downto 0 ) ;
     --  00 = AND, 01 = OR , 10 = XOR , 11 = NOR
     func       : in std_logic_vector(1 downto 0 ) ;
     --  00 = lui, 01 = setless , 10 = arith , 11 = logic
     output     : out std_logic_vector(31 downto 0) ;
     overflow   : out std_logic ;
     zero       : out std_logic);
end alu ;
```

The ALU has two 32-bit input operands (ports `x` and `y`) and a 32-bit output port (`output`). There are two sets of control inputs: `func` and `logic_func`. The control input `func` determines which of the four possible operations the ALU (load upper immediate, set less than zero, addition/subtraction, logical) is to be performed. The `logic_func` control input determines which of the for possible logic operations (AND, OR, XOR, NOR) is to be performed. An additional control signal (`add_sub`) is used to choose between addition or subtraction. Arithmetic is performed using 32-bit two's complement notation. Figure 1 gives the block diagram of the ALU and will be used to explain in further detail the operation of the ALU.

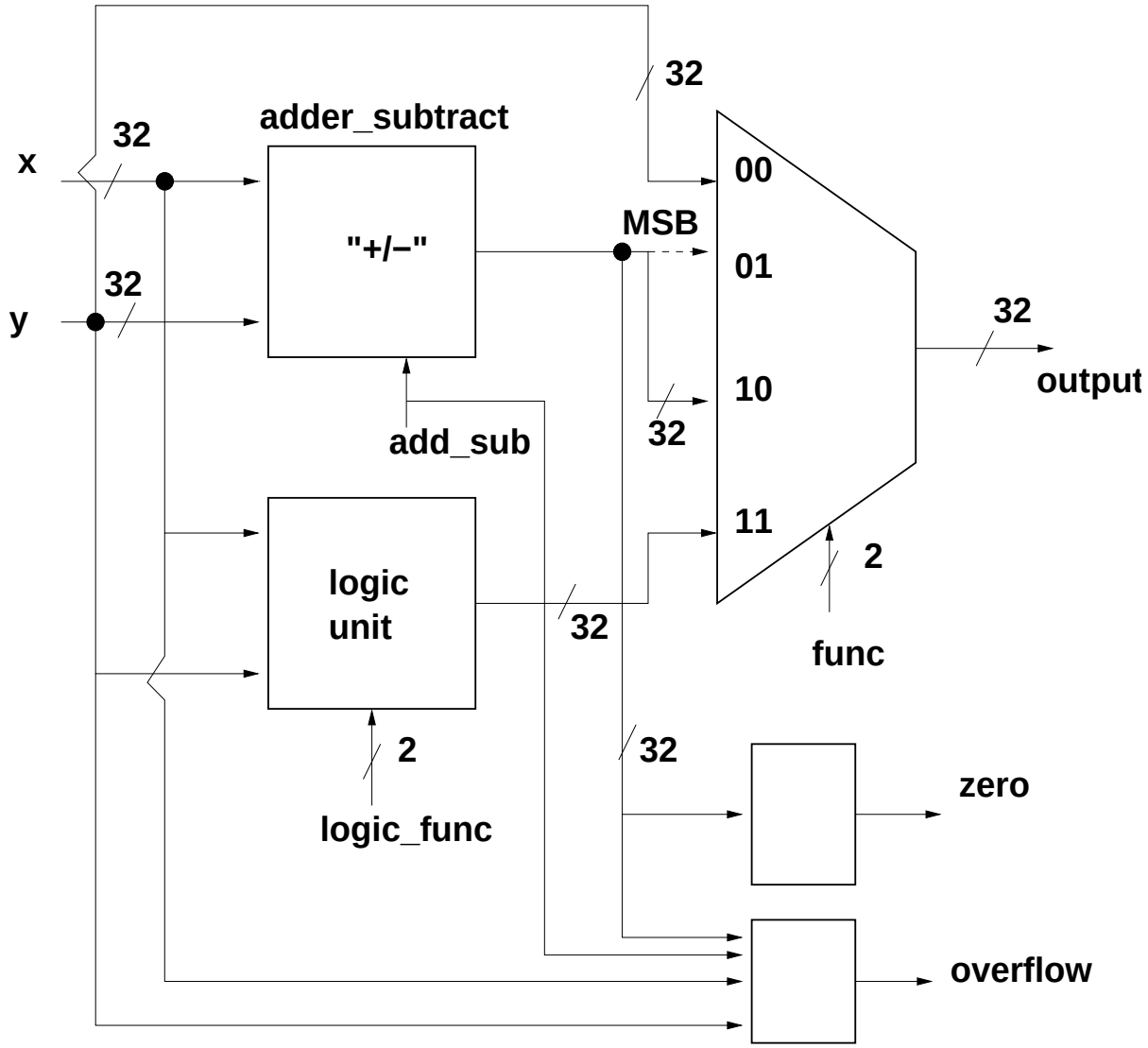

Figure 1: ALU block diagram.

The block labelled “adder_subtract” performs either the addition (`x+y`) or subtraction (`x-y`) in 32-bit two’s complement arithmetic as determined by the value of the control input `add_sub` (‘0’ = addition, ‘1’ = subtraction). The logic unit block performs either `x` AND `y`, `x` OR `y`, `x` XOR `y`, `x` NOR `y` as determined by the 2-bit control input `logic_func` (“00 = AND, “01” = OR, “10” = XOR, “11” = NOR). The multiplexer shown in Figure 1 is used to choose one of four possible values to the `output` port as selected by the 2-bit control input `func`. Table 1 explains the operation of the multiplexer.

Table 1: ALU multiplexer operation.

func	output	Comments
00	y	used to implement the lui instruction
01	"000...MSB" of adder_subttract	used for the slt (set less than zero) instruction
10	output of adder_subtract	x+y if add_sub = '0' x-y if add_sub = '1'
11	output of logic unit	x AND y if logic_func = "00 x OR y if logic_func = "01" x XOR y if logic_func = "10" x NOR y if logic_func = "11"

The MIPS CPU contains within its instruction set the lui (load upper immediate) instruction which loads the upper 16 bits of a specified CPU register with the 16 bit immediate data specified within the instruction with 0s in the low order 16 bits of the specified register. A separate hardware unit (in Lab 5) will be used for this and other related purposes. For the time being, the ALU output is simply the 32 bits present on the y input port whenever func = "00" as specified by the first row of Table 1.

The second row of Table 1 merits a few words of explanation. The MIPS CPU also contains the slt (set less than zero) instruction which is used in conditional branching. The slt instruction sets a specified register to 1 if x < y and 0 otherwise. The condition of whether x < y can be easily checked by performing the operation x - y, and padding the sign bit of the output of the adder_subtract unit with 0s to form a 32-bit number whose value is either 0 or 1. If the condition x < y is true, then the sign of x - y is negative and the most signficant bit (MSB) of the adder_subtract output will be a '1', otherwise it will be a '0'. [1]

The third row simply specifies that the ALU is to perform an arithmetic operation (x+y , or x-y) depending on the value of add_sub. The last row indicates that whenever func = "11", the output of the ALU is one of the four possible logical operations performed on the two inputs (as determined by the value of the logic_func input).

Figure 1 also indicates that the ALU has an output called zero which is to be set to '1' whenever the output of the adder_subtract unit is all 0s and '0' otherwise.

The ALU also has an overflow output which is set to '1' whenever arithmetic overflow (as defined by two's complement arithmetic) occurs. Recall that overflow occurs whenever two positive numbers are added (x + y) and the result is a negative number, or when adding two negative numbers, a positive result is obtained. The possibility of overflow occurring also exists when performing the operation x - y when the two operands are of opposite sign. Thus, one method of detecting overflow is to examine the signs of the two input numbers and the sign of the answer

(**HINT**: construct a truth table containing `x(31)`, `y(31)`, and the sign bit of the adder_subtract output showing which combinations result in oveflow, minimize this table using a K-map to obtain a Boolean equation for the overflow output. Construct a similar table when `add_sub` = 1 indicating the operation `x - y` is to be performed. You will have to choose between the two equations). This is why the overflow block in Figure 1 obtains its inputs from `x`, `y`, and the output of the adder_subtract unit as well as the `add_sub` signal. Note that there are different methods of overflow when performing two's complement arithmetic. Another method is to perform the XOR of the carry-in to the sign bit and the carry-out of the sign bit position. Depending on how you choose to implement the adder_subtract unit in VHDL, the carry-in to the sign bit may not always be accessible. This is also the reason why the entity specification did not explicitly state whether to use the `IEEE.std_logic_signed.all` library or the `IEEE.std_logic_unsigned.all` library. You may choose whichever method of implementing a 32-bit two's complement adder/subtractor in VHDL. The choice of implementation will dictate which library should be used.

Procedure

Design the ALU using VHDL. You may use any style of synthesizable VHDL (i.e. structural with port maps, processes, CSA statements, etc) you wish. Simulate your design with the Modelsim simulator to verify correct functioning for **all** the possible operations supported by the ALU for typical inputs values. Implement the design using the Xilinx Vivado tools and download the .bit file to the Nexys A7 FPGA board. Demonstrate the operation of the ALU to your lab demonstrator.

Board Implementation Procedure

The Nexys A7 development board has insufficient input/outputs to completely implement the full 32-bit design. For this reason, only a **subset** of the `x` and `y` input ports (the low order 4 bits) and a subset of the `output` port (also the low order 4 bits) will be mapped to switches and LEDs of the development board using a Xilinx Design Constraints file (.xdc) . The full 32 bit version of the ALU is to be simulated (and subsequently used as a part of the complete CPU in subsequent labs). In order to physically test the ALU design on the FPGA board, a "board wrapper" VHDL design will be used for implementation with the Xilinx Vivado tools. The entity specification for this "board wrapper" code is:

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_signed.all;

entity alu is
port(x_in, y_in : in std_logic_vector(3 downto 0); -- two input
                                                   -- operands
     add_sub : in std_logic ;   -- 0 = add , 1 = sub
     logic_func : in std_logic_vector(1 downto 0 ) ; --  00 = AND,
      -- 01 = OR , 10 = XOR , 11 = NOR
     func       : in std_logic_vector(1 downto 0 ) ; --  00 = lui,
```

```
      -- 01 = setless , 10 = arith , 11 = logic
     output_out     : out std_logic_vector(3 downto 0) ;
     overflow   : out std_logic ;
    zero       : out std_logic);
end alu ;
```

Create a separate VHDL source code file (name it alu_board.vhd or lab1_board.vhd) containing the above entity specification together with your source code for your ALU design. It will be necessary to declare `x`, `y`, and `output` as **internal** signals (of full 32-bit width) in your "board_wrapper" code as in:

```
signal x,y, output : std_logic_vector(31 downto 0);
```

One can then use the VHDL concatenation operator together with the slice operator to assign a full 32 bit value to the `x` and `y` internal signals (with 0s padded to the high order bits, and the 4 values of each input port to the low order 4 bits) :

```
begin  -- of the architecture


-- assign port inputs to internal signals x and y

x(3 downto 0) <=   x_in(3) & x_in(2) & x_in(1) & x_in(0)  ;
y(3 downto 0) <=   y_in(3) & y_in(2) & y_in(1) & y_in(0)  ;
x(31 downto 4) <= (others => '0');
y(31 downto 4) <= (others => '0');
```

In a similar manner, one may use the slice operator to assign the 4 low-order bits of the 32 bit output internal signal to the output port called output_out :

```
output_out(3 downto 0) <= output(3 downto 0);
```

Requirements

1. Modelsim simulation results for the design.

2. The VHDL source code for the ALU (both for simulation and the "board wrapper" code.)

3. The Xilinx Vivado `runme.log` files for the synthesis and implementation runs. Discuss the importance of any messages or warnings. The `runme.log` files are found within the synth_1 and impl_1 subdirectories of your Vivado project directory. For example:

```
/home/t/ted/VIVADO/COEN316_Nexsys_Board/Lab1/ALU/ALU.runs/synth_1
/home/t/ted/VIVADO/COEN316_Nexsys_Board/Lab1/ALU/ALU.runs/impl_1
```

References

1. Computer Architecture From Microprocessors to Supercomputers, Behrooz Parhami, Oxford University Press, ISBN 0-19-515455-X, 2005.

Ted Obuchowicz
Sept. 19, 2016

Revised: Sept. 10, 2018: elaborated upon overflow detection.
Revised: Sept. 15, 2021: revised for Vivado implementation on Nexys A7 board (use of “board wrapper code to account for mapping of subset of ports).