Proiect ALU

- Documentatie -

1. Table of contents:

2. Preface & credits

This project was made as an assignment for the CN laboratory in the year 2025 at the Polytechnic University of Timisoara.

All of the typical phases of development were explored thoroughly, those being *the design phase*, *the implementation phase*, and *the testing phase*.

Made by:

Androne Vlad,

Andreas Antoniu,

Indrie Filip-Iulian,

Mihai Toderasc.

A wonderful Learning experience.

3. Project Overview

This chapter offers a simple yet comprehensive description of the project as a whole, as well as an overview of its main objectives, if they were met, and to what degree.

3.1 Description

The project at hand attempts to recreate an *Arithmetic Logic Unit* (**ALU**) typically found inside Central Processing Units (**CPUs**). This component serves, as the name suggests, as a unit that performs mathemetical operations such as *addition*, *subtraction*, *multiplication*, and *division* on a given set of input values, and gives back the result for later use inside the CPU.

This implementation of an ALU uses two serialised 8-bit buses for *input* and *output* respectively, a 2-bit port for the *operation* that is to be performed, and 1-bit ports for the *clock*, *reset*, *begin*, and *end* signals.

Note that the therm “bus” is loosely used here, as registers were actually used to simulate said input and output buses. Furthermore, in the rest of the documentation, we will refer to these as *inbus* for the input bus, and *outbus* for the output bus.

3.2 Objectives

The objectives of this project are as follows, in no particular order:

1. To recreate as faithfully and in as much detail as possible a working ALU that is capable of executing all four of the elementary operations: addition (+), subtraction (-), multiplication (*), and division (/).
2. Implement different and complex algorithms for multiplication and division.
3. Design a full schematic for the hardware designing phase
4. Design all components from scratch in the implementation phase (also known as reinventing the wheel)

5. Overcome any possible complications and iron out any kinks that arise along the way, finishing the objective with a minimal number of bugs.

4. Architecture diagrams

The design phase was solely focused on creating a detailed diagram of the ALU architecture for further reference during the implementation phase, and to help with debugging.

The diagrams were made using MathLab's Simulink library as it provides a technical environment and all the tools needed to create a clean and complete schematic.

[Available upon request]

Each component of the ALU was designed down to the wire, using elementary logic gates. This will be evident further down the line when talking about the following phases.

5. Code listing

This section focuses on showcasing the implementation phase of the project. Here we give a bit of insight into how the actual code is structured, how the different modules are used and why some of the design decisions were made.

5.1 Module structure and use

Complex gates:

Wordgate

```
module b9wordgate(input[8:0] x, input s, output[8:0] y);


9-bit XOR gate
     x – input 9-bit word
```

s – enable bit. decides wether the input gets complemented

y – output 9-bit word

Decoder

```
module dec2_4(input[1:0] c, output d3, d2, d1, d0);
```

2-bit to 4-bit decoder (one hot)

c – input 4-bit code

di – output signals from the decoder, respectively

Multiplexer

```
module b9s_21mux(input[8:0] x, y, input s, output[8:0] z);
```

9-bit double input multiplexer with single bit selection signal

x, y – 9-bit word inputs

s – selection signal

z – selected 9-bit output

Counters:

Simple counter

```
module counter_4b (input clk, rst, output reg [3:0] val);
```

4-bit counter that increments its output by 1 every clock cycle. Possible values of this counter range from 0 (0x0) to 15 (0xF), at which point it resets back to 0.

The *rst* signal sets the output value *val* to 0b0000. *rst* is active on 1.

val retains the value in memory.

clk – clock signal

rst – reset signal

val – output 4-bit value

```
module cnt_47s (input clk, rst, output is4, is8);
```

This counter implements the simple 4-bit counter. It only returns the signals when they reach a certain condition, i.e. when the internal counter reaches a certain value.

The *rst* signal resets the internal counter, as well as the output *is4*, *is8* values. *rst* is active on 1.

clk – clock signal

rst – reset signal

is4 – signal denoting that the internal counter reached the value 4

is8 – signal denoting that the internal counter reached the calue 8

```
module cnt_k (input clk, rst, cnt_up, cnt_down, output is0);
```

This counter implements the simple 4-bit counter. It is used for counting up to a given value (max. 15) and back down to 0. It does this by implementing two counters and checking that the value of the difference between their counts is 0.

The *rst* signal resets the internal counter, as well as the output *is0* value. *rst* is active on 1.

clk – clock signal

rst – reset signal

cnt_up – increments the first internal counter

cnt_down – increments the second internal counter

is0 – signal denoting that the internal counters have counted an equal amount of times

module mod4c(input clk, rst, enable, set0, output[1:0] c);

2-bit counter that counts in modulo 4 increments. This effectively limits the counter's maximum value to the interval (0, 3), inclusively.

The *rst* signal resets the value of the counter, as well as the output *c* value. *rst* is active on 1.

clk – clock signal

rst – reset signal

enable – allows the counter to increment on clock cycles

set0 – sets the output value to 0

c – 2-bit output

module ph4_cnt(input clk, rst, _begin, _end, set0, output ph3, ph2, ph1, ph0);

This counter implements the modulo4 counter. It works similarly to the modulo counter implemented internally, with a couple differences.

The *rst* signal resets the value of the counter, as well as the output *ph3*, *ph2*, *ph1*, and *ph0* values. *rst* is active on 1.

Note that the counter uses a one-hot implementation, meaning that only one output can be active at a time.

clk – clock signal

rst – reset signal

_begin –starts the incrementation of the counter, until receiving the *_end* signal

_end – stops the counter from incrementing

set0 – sets the value 1 to the output *ph0*, and 0 for the rest

phi – outputs of the counter, respectively

Adders

Full adder cell

```
module fac (input x, y, c_in, output z, c_out);
```

The most basic component used for addition.

x, y, c_in – 1-bit inputs

z – sum bit output

c_out – carry bit output

Ripple carry adder

```
module rca #(parameter n = 9)(input[n-1:0] x, y, input c_in,
output[n-1:0] z, output c_out);
```

The simplest adder used for multi-bit operands. By default, it uses 9-bit operands for input and output, but the value can be modified to suit other needs.

Note that because of the syntax of the input / output values, and the internal code of this module, using a value of n=1 might result in a compilation error.

n – variable parameter for size of operands

x, y – n-bit input operands

c_in – 1-bit input operand

z – n-bit output value

c_out – carry output value

Subtractor

module sub_1(input[8:0] x, input enable, output[8:0] y);

Subtracts 1 from a given input. Really

x – 9-bit input operand

enable – bit to subtract

y – 9-bit output value

Complex components

Register

module bit9_reg(input clk, rst, ls, rs, drs, load, oregbl1, oregbl0, oregbr1, oregbr0, setb0, b0, input[8:0] load_bits, output[8:0] write_bits);

Custom made register. Allows for storage of 9-bit values, shifting to the left and right by one or two bits at a time.

Shifting takes bits from an external source. If copying of last bit is wanted, this requires external logic. One can also leverage the option to directly set the value of the last stored bit inside of the register.

The *rst* signal resets the value stored inside the register. *rst* is active on 1.

clk – clock signal

rst – reset signal

ls – left shift signal

rs – right shift signal

drs – double right shift signal

oregbli – bits that are to be read during a right shift operation, respectively

oregbri - bits that are to be read during a left shift operation, respectively

setb0 – when active, sets the 0th bit to the given value

b0 – value to set the 0th bit to

load_bits – 9-bit input value to be stored in the register

write_bits – 9-bit output value to be read from the register

Control unit

```
module control_unit(input clk, rst, _begin, b7, b6, a8, q2,
cnt0, cnt4, cnt8, amsb3ne, qlsb3ne, input[1:0]op, output l1, l2,
cl3, add, sub, ls, rs, drs, w1, w2, cnt_incr, load_qs, output
reg[1:0] div_op, output div_correction, div_hold_init,
div_hold_fin, ds_conv_sub, cuend, output[3:0] phase, output[2:0]
cycle);
```

The main component that does the heavy lifting. It implements different components mentioned above, and gives most of the signals required for the correct functioning of the ALU.

Note that the *phase* and *cycle* output values are used for visualisation purposes only, and have no functional use inside the program, although functionality could be atributed.

The *rst* signal propagates downwards. *rst* is active on 1.

clk – clock signal

rst – reset signal

_begin – signal that notifies different systems to activate

(a|b|q)i – the ith stored bit in the A, B, or Q register, respectively

cnti – signal that a counter has reached the value i

amsb3ne – signal active if the 3 most significant bits in the A register are not equal

qlsb3ne - signal active if the 3 least significant bits in the Q register are not equal

op – 2-bit operation code (00 +, 01 -, 10 *, 11 /)

li – signal for loading the ith register

cl3 – conditional load of third register

add – addition signal

sub – subtraction signal

ls – left shift signal

rs – right shift signal

drs – double right shift signal

wi – signal to write the ith register's values

cnt_incr – signal to increase a counter's value

load_qs – used for division when working with the Q and Q' registers

div_op – saves the operation to be performed, while performing the division operation

div_correction – enables correction step in the division operation

div_hold_init, div_hold_fin – stop the phase counter to shift and unshift registers, respectively, in the division operation

ds_conv_sub – used in the normalisation step, in the division operation

cuend – control unit end signal

phase – (display only) phase code

cycle – (display only) cycle code

Arithmetic logic unit

```
module alu(input[7:0] in, input[1:0] op, input _begin, clk, rst,
output [7:0]out, output _end);
```

The actual ALU unit module representation. It has all been leading up to this.

The input and output are serialised. This limits the ALU to only accepting or delivering one byte of information at a time. It also requires the designer to be careful of the order the inputs are stored in memory.

The *rst* signal propagates downwards. *rst* is active on 1.

in – 8-bit serial input bus

op – 2-bit operation code

_begin – signal that tells the ALU to begin operating

clk – clock signal

rst – reset signal

out – 8-bit serial output bus

_end – signal denoting the termination of the operation process

6. Functionality

This section focuses on the tests that were performed on the finished ALU module during the testing phase. Each one was passed successfully and yielded expected results. Additional information can be found in the next subsections.

6.1 Testbench

A number of testbenches were used during the testing, but only a more generalised one was kept. It tests all the operations with different inputs.

[Available upon request]

6.2 Waveforms

A program that parses the output from the Icarus Verilog Compiler was developed, with the aim to convert the text to a visible waveform.

[Available upon request]

7. Conclusion

This ALU project successfully met all its intended objectives, demonstrating a solid understanding of digital logic and hardware design.

Despite a short implementation and testing window, the final system performed all four basic arithmetic operations—addition, subtraction, multiplication, and division—correctly and reliably. Building each component from scratch allowed us to explore key design principles, and thorough testing confirmed the ALU's functionality.

Overall, the project was a valuable and rewarding learning experience in digital systems development.