Lab 4: Arithmetic Logic Unit (ALU)

Introduction

The heart of every computer is an Arithmetic Logic Unit (ALU). This component is the part of a computer that performs arithmetic operations on numbers, e.g. addition, subtraction, etc. This lab use the Verilog language to implement an ALU with 10 functions. Use of the case structure will make this job easy. (Hint)

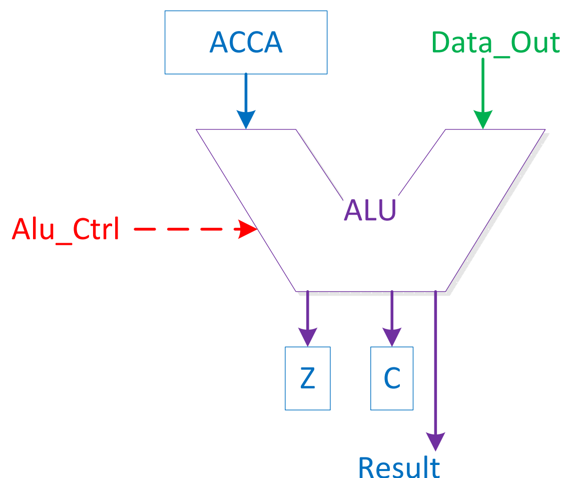

Figure 1: Arithmetic Logic Unit (ALU)

The ALU that will be built (see Figure 1) will perform 10 functions on 8-bit inputs (see Table 1). Please make sure to use the same variable name as the ones defined in this lab. Do **NOT** make your own, as calling the various arithmetic functions later will then be standardized. The ALU will generate an 8-bit result (`Result`), a one bit carry (`C`), and a one bit zero-bit (`Z`). As inputs, the ALU will have three inputs, two 8-bit data lines (`ACCA & Data_Out`), and select variable (`Alu_Ctrl`). The select (`Alu_Ctrl`) will choose which of the 10 ALU functions to execute.

1 Prelab

1.1. Fully read the Lab and Supplement. This lab will have the symbol (⋆⋆) to denote lab book signatures.

1.2. ⋆⋆ Wire (on your personal breadboard) the FPGA board with two sets of DIP switches (12 switches required), and the to 7-Segment displays. (Yes, this will use 26/32 I/O pins) Also, take care not to connect the FPGA two analog pins *This will be worth a signature this week!* Fill out table 2 with the function of each assigned pin in the form: *7-Segment 1 Seg A, 7-Segment 1 Seg B, Dip Switch 0, Dip Switch 1, etc.*

1.3. Fill out Table 1 (Give *unique* values to each instruction.) (**HINT:** use the pre-assigned values from the file "Constants.v" found on the Lab webpage.) How many bits should `Alu_Ctrl` be?

1.4. Write the code to implement the ALU. (See Verilog Supplement for coding various operations using operators in Verilog, and how to use the file "Constants.v") If you need help writing the code, seek assistance! Visit the digital lab during other 231 lab hours or NMT EE help session hours (5:00-8:00 Mon.-Thurs.)

2 Lab

2.1. Using the written code from your prelab, create a Vivado project to implement the ALU.

2.2. Design the ALU using Verilog. **Make sure you deal with any unused bit combinations of the `Alu_Ctrl` lines.** (Hint: review `default` cases in Supplement)

2.3. ⋆⋆ Simulate the ALU and test different combinations of `DATA` and `ACCA`. **Test ALL 12 of the instructions.**

2.4. Create another module (main.v) that will call your ALU module. In this module, have `ACCA` and `Alu_Ctrl` as the external inputs. Output your results on the two 7-segment displays. Hard code `DATA` as `0xAA (1010 1010)`

2.5. ⋆⋆ Program your FPGA with the ALU code.

Table 1: Arithmetic Logic Unit Instructions

Alu_Ctrl	Instruction	Operation (Mnemonic)
	LDDA	Loads ACCA with the value on the Data bus. Z changes to 1 if Result == 0. (Load ACCA from Data)
	ADDA	Adds the value on the Data bus to the value in ACCA and saves the result in ACCA. C is the carry (out) from addition and Z is set if the result is 0. (Add ACCA and Data)
	SUBA	Subtracts the value on the Data bus from the value in ACCA and saves the result in ACCA. C is the carry (in) from subtraction and Z is set if the result is 0. (Subtract value in Data from ACCA)
	ANDA	Perform a bitwise AND of the value on the Data bus with the value in ACCA. Save the result in ACCA. C should be the logical AND of the value on the Data bus with the value in ACCA. Z is set if the result is 0. (AND of ACCA and value on Data)
	ORAA	Perform a bitwise OR of the value on the Data bus with the value in ACCA. Save the result in ACCA. C should be the logical OR of the value on the Data bus with the value in ACCA. Z is set if the result is 0. (OR of ACCA and value on Data)
	COMA	Replace the value in ACCA with its one's complement. C is set to 1 and Z is set if the result is 0. (Compliment ACCA)
	INCA	Increment value in ACCA. Z is set if the result is 0. (INCA ACCA)
	LSLA	Logical shift left of ACCA. C is set to the previous MSB of ACCA and Z is set if the result is 0. (Logical shift left ACCA)
	LSRA	Logical shift right of ACCA. C is set to the previous LSB of ACCA and Z is set if the result is 0. (Logical shift right ACCA)
	ASRA	Arithmetic shift right of ACCA. C is set to the previous LSB of ACCA and Z is set if the result is 0. (Arithmetic shift right ACCA)
	ZERO	Zero the value of ACCA. C is set to 0 and Z is set to 1. (Zero ACCA)
	RST	Reset ACCA to 0xFF. C is set to 0 and Z is set to 0. (Reset ACCA)

Table 2: CMOD Spartan 7 — DIP Pin Assignments

DIP Pin	**FPGA Pin**	Wiring Function/assignment	Wiring Function/assignment	**FPGA Pin**	DIP Pin
1	**L1**			**A4**	48
2	**M4**			**A3**	47
3	**M3**			**B4**	46
4	**N2**			**B3**	45
5	**M2**			**C1**	44
6	**P3**			**B1**	43
7	**N3**			**B2**	42
8	**P1**			**A2**	41
9	**N1**			**C5**	40
—	—	—	—	—	—
—	—	—	—	—	—
—	—	—	—	—	—
—	—	—	—	—	—
—	—	—	—	—	—
—	—	—	—	—	—
16	**P14**			**A33**	33
17	**P15**			**A32**	32
18	**N13**			**J11**	31
19	**N15**			**M13**	30
20	**N14**			**L13**	29
21	**M15**			**J15**	28
22	**M14**			**K14**	27
23	**L15**			**L14**	26
24	**VU**			**GND**	25

3 Supplement: Verilog (3)

3.1 Parameterization

3.1.1 Macros

Listing 1: Macros in Verilog (Constants.v)

```
1 `define Rst_Addr 8'hFF   // Define value as name instead of always using value
```

Listing 2: Macros in Verilog (including Constants.v)

```
1 `timescale 1ns / 1ps
2 `include "Constants.v"
3 module ALU(A, B, C, Data .....);
4 ...
5 assign data = `Rst_Addr; // Use name as value, helps to keep values organized
6 ...                      // Tic is Necessary for calling
7 endmodule
```

3.2 Operators

3.2.1 Ternary Operator

Listing 3: Ternary Operator in Verilog

```
1 assign y = sel ? a : b;
```

If `sel` is true, `y` is assigned to `a`, otherwise it is assigned to `b`.

3.2.2 Concatenation

Listing 4: Concatenation in Verilog

```
1 input a, b, c, d;          // 1-bit inputs
2 output [3:0] Out;          // 4-bit output
3 assign Out = {a, b, c, d} // inputs as each bit of 4-bit Out. eg: Out[3] = a, Out[2] = b, etc.
```

Bits are concatenated using { }.

3.2.3 Comparison

Listing 5: Comparison in Verilog

```
1 if(a > b) begin
2   y = a;
3 end
```

Compare `a` to `b`, if true set `y` equal to `a`. Other comparators are listed in Listing 6.

Listing 6: Comparison Operators

```
1 >      // Greater than
2 <      // Less than
3 >=     // Greater than or equal to
4 <=     // Less than or equal to
5 ==     // Equality
6 ===    // Equality including X and Z
7 !=     // Inequality
8 !==    // Inequality including X and Z
```

3.2.4 Logical Operators

Listing 7: Logical Operators

```
1  !     // Logical negation
2  &&    // Logical and
3  ||    // Logical or
```

3.2.5 Binary Arithmetic Operators

Listing 8: Binary Arithmetic Operators

```
1  +     // Addition
2  -     // Subtraction
3  *     // Multiplication
4  /     // Division (truncated)
5  %     // Modulus
```

3.2.6 Unary Arithmetic Operators

Listing 9: Unary Arithmetic Operators

```
1  -     // Change the sign of the operand
```

3.2.7 Bitwise Operators

Listing 10: Bitwise Operators

```
1  ~     // Bitwise negation
2  &     // Bitwise AND
3  |     // Bitwise OR
4  ^     // Bitwise XOR
5  ~^    // Bitwise XNOR
6  ^~    // Bitwise XNOR (also)
```

3.2.8 Unary Reduction Operators

- Produce a single bit result by applying the operator to all the bits of the operand.

Listing 11: Unary Reduction Operators

```
1  ~     // Bitwise negation
2  &     // Bitwise AND
3  |     // Bitwise OR
4  ‘&    // Reduction NAND
5  ~|    // Reduction NOR
6  ^     // Bitwise XOR
7  ~^    // Bitwise XNOR
8  ^~    // Bitwise XNOR (also)
```

3.2.9 Shift Operators

- Left operand is shifted by the number of bit positions given by the right operand.
- Zeros are used to fill vacated bit positions.

Listing 12: Shift Operators

```
1   <<  // Logical left shift
2   >>  // Logical right shift
```

3.2.10 Precedence

Listing 13: Precedence

```
1  /* Highest Precedence */
2    !, ~
3    *, /, %
4    +, -
5    <<, >>
6    <, <=, >, >=
7    &
8    ^, ^~
9    |
10   &&
11   ||
12   ?:
13 /* Lowest Precedence */
```

```
/////////////////////////////////////////////////////////////////////////////////////////
// -file Constants.v
// -author Christopher Ramirez            <christopher.ramirez@student.nmt.edu>
//                                        <ramirez.christopher.a@gmail.com>
// -Credit to William Brooks (NMT alumni) for the original source files for labs
// -version 1.0
//
// -brief Global definitions of constant values for NMT EE231 Lab (Digital Electronics)
//
// -File Description
//    This file is used to improve consistency and modularity between labs/projects
// Signals/Messages/opcodes that are passed between modules spanning multiple (.v)
// files are defined here for consistency and ease of use during later labs.
//_____________________________________________________________________________________//
//
//
//
`ifndef _CONSTANTS_
`define _CONSTANTS_
//
/////////////////////////////////////////////////////////////////////////////////////////
// -section Default Flags (Inactive; Exception Handling)
// Default states for each module; These choices are left as a recommendation
// for the verilog module developer
// DO NOT ASSUME they have been implemented as such.
// These values are recommended as pre-set default values
//-------------------------------------------------------------------------------------//
//
`define DEFAULT_ADDR  `ADDR_RESET         // < Default Memory address
`define DEFAULT_ALU   `ALU_ZERO           // < Default ALU instruction
`define DEFAULT_DDIR  `DATA_DIRECTION_IN  // < Default Data direction is input (Safer)
`define DEFAULT_LOAD  `REG_INACTIVE       // < Default LOAD control
`define DEFAULT_INC   `REG_INACTIVE       // < Default INCREMENT control
`define DEFAULT_MEMW  `MEM_READ           // < Default Write control
`define DEFAULT_STATE `STATE_RST          // < Default CCU control state
`define RESET_VECTOR  8'hFF               // < Reset vector in Memory map
//_____________________________________________________________________________________//
//
//
//
/////////////////////////////////////////////////////////////////////////////////////////
// -section MUX Selection Choices
// Behavior of Addr_Mux_Select is controlled by these flags
// Specifies which address/data is to be output by the 4:1 MUX
//-------------------------------------------------------------------------------------//
//
`define ADDR_RESET              2'b00 // < Output Reset Address to Address Line
`define ADDR_PC                 2'b01 // < Output contents of PC register to address line
`define ADDR_MAR                2'b11 // < Output contents of MAR register to address line
`define ADDR_IR                 2'b10 // < Output contents of IR register to address line
//_____________________________________________________________________________________//
//
//
//
/////////////////////////////////////////////////////////////////////////////////////////
// -section ALU Instructions (Optional features included)
//
// Provides definition of instructions implemented by ALU
// Instructions are chosen to closely match CPU Instructions
// Implementation of logical operators are encouraged to be bitwise
// Optional instructions are noted, but not required
// In its current state, all values are implemented
//-------------------------------------------------------------------------------------//
//
// Machine State
```

```
`define ALU_ZERO            4'h0  // < Set ACCA to zero
`define ALU_ONES            4'h1  // < Set ACCA to ones
`define ALU_LOAD            4'h2  // < Load value
// Arithmetic Operations
`define ALU_ADDA            4'h3  // < Add value to ACCA
`define ALU_SUBA            4'h4  // < Subtract value from ACCA
// Mathematical Compliments
`define ALU_COMA            4'h5  // <           (1's) Compliment of ACCA
`define ALU_COMA2           4'h6  // < OPTIONAL: (2's) Compliment of ACCA
// Bitwise Logic
`define ALU_ORAA            4'h7  // < OR value with ACCA
`define ALU_XORA            4'h8  // < XOR value with ACCA
`define ALU_ANDA            4'h9  // < AND value with ACCA
// Increment/Decrement
`define ALU_INCA            4'hA  // <           Increment ACCA (by 1)
`define ALU_DECA            4'hB  // < OPTIONAL: Decrement ACCA (by 1)
// Logical Shifts
`define ALU_LSRA            4'hC  // <           Logical Shift Right of ACCA
`define ALU_ASLA            4'hD  // < OPTIONAL: Arithmetic (Ring) Shift Left of ACCA
`define ALU_LSLA            4'hE  // < Logical Shift Left of ACCA
`define ALU_ASRA            4'hF  // < Arithmetic (Ring) Shift Right of ACCA
//____________________________________________________________________________________//
//
//
//
//////////////////////////////////////////////////////////////////////////////////////////
// -section Activity series
// Provides a common definition of active and non-active systems.
// Such terminology allows for responsive programming
//------------------------------------------------------------------------------------//
//
`define REG_ACTIVE    1'b0         // < Defines Registers as active low
`define REG_INACTIVE  ~`REG_ACTIVE  // < Compliment of REG_ACTIVE
`define MEM_WRITE     1'b0         // < (Enable) Writing data to Memory
`define MEM_READ      ~`MEM_WRITE   // < Complement of MEM_WRITE; Do not Write
//____________________________________________________________________________________//
//
//
//////////////////////////////////////////////////////////////////////////////////////////
// -section Finite State Machine (FSM) States
// States chosen to be loosely based on Grey-Code
// Typical Execution Loop {01 -> 11 -> 10} and {01 -> 11}
//------------------------------------------------------------------------------------//
//
`define STATE_RST     2'b00         // < Reset State
`define STATE_FETCH   2'b01         // < Fetch State
`define STATE_EX1     2'b11         // < Execution 1 State
`define STATE_EX2     2'b10         // < Execution 2 State (Not always used)
//
//
//
//////////////////////////////////////////////////////////////////////////////////////////
// -section I/O Port Controls
// Models Bidirectional Port Behavior found on many Microcontrollers
// Port is Bi-Directional w/Write Enable (WE) and Data Direction Control Register (DDCR)
//------------------------------------------------------------------------------------//
//
`define DATA_DIRECTION_IN 1'b1 /**< Treats Port an an Input (from External Sources)*/
`define DATA_DIRECTION_OUT 1'b0 /**< Treats Port as an Output (to External Sources)*/
//____________________________________________________________________________________//
//
//
//
//////////////////////////////////////////////////////////////////////////////////////////
// -section CCU (Computer Control Unit) Instructions (Optional Features Included)
```

```verilog
// Provides definition of instructions implemented by CCU
// Instructions chosen for computational convenience:
//     1. Requirement for EX2 indicated by flag-bit
// Logical operators are encouraged to be bitwise.
// OPTIONAL Instructions are noted as such, but not required
// WARNING: In current state, all addresses are not used
//-----------------------------------------------------------------------------------------//
//
// No operation/Wait
`define INST_NOP      8'h00 // < OPTIONAL: No Operation
// Memory Interaction
`define INST_LDAA_IMM 8'h01 // <           Load ACCA with Immediate Value
// Increment/Decrement (by 1)
`define INST_INCA     8'h02 // <           Increment ACCA (by 1)
`define INST_DECA     8'h03 // < OPTIONAL: Decrement ACCA (by 1)
// Mathematical Compliments
`define INST_COMA     8'h04 // <           (1's) Compliment of ACCA
`define INST_COMA2    8'h05 // < OPTIONAL: (2's) Compliment of ACCA
// Bitwise Shifts
`define INST_LSLA     8'h06 // <           Logical Shift Left of ACCA
`define INST_ASLA     8'h07 // < OPTIONAL: Arithmetic Shift Left of ACCA
`define INST_LSRA     8'h08 // <           Logical Shift Right of ACCA
`define INST_ASRA     8'h09 // <           Arithmetic (Ring) Shift Right of ACCA
// Logical Jumps
`define INST_JMP      8'h0A // < Unconditional Jump to specified address
`define INST_JCC      8'h0B // < Jump if Carry Clear to specified address         (C = 0)
`define INST_JCS      8'h0C // < Jump if Carry Set to specified address           (C = 1)
`define INST_JEQ      8'h0D // < Jump to specified address if Equal               (Z = 1)
`define INST_JNEQ     8'h0E // < OPTIONAL: Jump to specified address if Not Equal (Z = 0)
// Memory Interaction
`define INST_STAA     8'h80 // < Store ACCA at specified address
`define INST_LDAA     8'h81 // < Load ACCA with Value at specified address
// Arithmetic Operations Involving Memory
`define INST_ADDA     8'h82 // < Add value at specified address to ACCA
`define INST_SUBA     8'h83 // < Subtract value at specified address from ACCA
`define INST_CMPA     8'h85 // < Compare ACCA with value at specified address
// Bitwise Logic
`define INST_ORAA     8'h86 // < OR ACCA with value at specified address
`define INST_XORA     8'h87 // < OPTIONAL: (XOR) ACCA with value at specified address
`define INST_ANDA     8'h88 // < AND ACCA with value at specified address
//_________________________________________________________________________________________//
//
//
//
`endif
```