ISTANBUL TECHNICAL UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

BLG 222E

COMPUTER ORGANIZATION PROJECT REPORT

PROJECT NO : 2

PROJECT DATE : 30.04.2020

GROUP NO : 31

GROUP MEMBERS:

150180001 : MEHMET ARİF DEMİRTAŞ

150180002 : BARIŞ EMRE MİŞE

150180007 : ÖMER FARUK ÖZKAN

150180009 : ERCE CAN BEKTÜRE

SPRING 2020

Contents

1 INTRODUCTION

In this second project, we implemented an ALU that supports 8-bit arithmetic, logic, shift operations and used it with our registers from the first project to simulate a computer. We will explain our design and give some test cases in this report.

File names are as follows:

- **hw2part1.circ**
- **hw2part2.circ**

Also, these files are included from the first project:

- **part2a.circ**
- **part2b.circ**
- **part2c.circ**

2 DESIGN

This project was divided into two main parts, Part 1 in which we created the Arithmetic Logic Unit and Part 2, which was combining our previous work to create a whole system.

2.1 PART 1

In our ALU design, we used a 2:16 MUX to map our inputs through different functions to OutALU. We have 16 functions in total so we used 4-bit FunSel selection input.

2.1.1 FunSel: 0000 to 0011

This part are made of straightforward operations, A, B, $\bar{A}$, $\bar{B}$. For A and B, we simply connected two lines to each of the inputs. For complements, we used Logisim's inverter gate.

2.1.2 FunSel: 0100 to 0110 - Arithmetic Operations

We implemented three arithmetic operations, in a different subcircuit called 'arithmetic'. We use a full adder inside, and we use the last two bits of FunSel to use this full adder for adding, adding with carry, subtraction using 2's complement. For adding, we did not modified anything. For adding with carry, we supplied carry input to full adder.

For subtraction, we complemented B and gave 1 to carry input of full adder so it became 2's complement.

In addition, if the signs are different, overflow is impossible. If signs are same, output must have the same sign or there is overflow. Also, we can view subtraction as addition, using 2's complement. Using this information, we developed a circuit using XOR gates to detect overflow.

We XOR'ed sign bits of A and B and complemented, XOR'ed sign bits of result and one of the operands, AND'ed these and fed it to the 0 input of MUX. Then we did the same thing using A and B's complement, and fed it to the 1 input of MUX. We connected the selection line of this MUX to a signal that determines which operation we do. We observed that the output of this circuit is overflow state.

2.1.3 Funsel: 0111 to 1001 - Logic Operations

We did our logic operations in a subcircuit called 'logical'. This subcircuit asynchronously does all three logical operations and gives out 3 outputs, we use required outputs in our main circuit. Our logical operations are AND, OR, XOR.

2.1.4 Funsel: 1010 and 1011 - Logic Shifts

We splitted input signals into 8, used 8 2:1 MUX'es and fed each line into the input 0 of MUX below and input 1 of MUX above. We also used another MUX for carry, which we fed A_7 to input 0, and A_0 to input 1. We fed static 0 to empty inputs of MUX'es. We used a common selection line and gave out 8 bit output.

2.1.5 Funsel: 1100 and 1101 - Arithmetic Shifts

There is no carry in arithmetic shifts, unlike logic. Shifting left is adding 0 to the right, which we managed by giving each signal to the MUX one below. Shifting right is copying sign bit, so we fed each signal to the MUX one above and fed A_7 to the last MUX again.

Also, there is overflow in ASL, so we checked for it by using XOR on most significant two bits.

2.1.6 Funsel: 1110 and 1111 - Circular Shifts

Circular shifts do not take input other than carry. Instead, each bit is shifted one below/above and empty bit is taken from carry.

To account for overflow, we XOR'ed bits 6 and 7 and also XOR'ed bit 7 and carry in, and fed them into a MUX. MUX is selected by FunSel, so if it is shift left, it outputs first XOR result, if it is shift right, it outputs second XOR result.

To account for carry, we fed bits 0 and 7 into a MUX and chose carry according to the shift type.

2.1.7 ALU Flag Designs

There are four flags we keep track of in our design. Z, C, N and O.

Z is zero flag, we simply OR all output bits and invert it to decide Z flag.

C is carry flag. Carry is updated in arithmetic operations, LSL, CSR and CSL. All circuits that do these operations output a carry. We feed all carry outputs to a MUX, and this MUX decides which operation is being done, so it outputs that operations carry flag. This output goes to a controlled buffer, which has enable input. We use a decoder and OR gate to select cases where carry is updated, and feed this to enable input.

N is negative flag, we decide it using the MSB of the output and XOR'ing it with 0.

O is overflow flag. Overflow can occur in arithmetic operations, ASL, CSR and CSL. Similar to carry, it is computed in each situation where overflow can occur. A MUX selects relevant cases.

We give out a four bit flag output and connected it to a 4-bit register in 'Main'.

2.2 PART 2

In our second part, we simulated a system with registers, ALU and memory. There are 2 register banks and 1 16-bit register.

ALU output can be written to memory, or it can be input to the registers in part2a or part2b. Registers in part2a can also be take input from part2c, part2b or memory directly. Registers in part2a are the inputs of ALU. ALU also can take inputs from the same sources.

Registers in part2b also takes input from memory, part2c, and ALU. Part2b is the only source for address input of memory.

Part2c register can only be loaded from memory.

Our memory is a RAM with 8-bit addresses and 8-bit data, with separate load and store ports. Its usage is documented in test cases.

3 TEST CASES

3.1 PART 1

In order to add F7 - F3; A = 0111 1111 B = 0011 1111 FunSel = 0100 Result is E6, after rising edge ZCNO error flag is 0011 which means there is overflow.

In order to subtract F1 - 30; A = 0001 1111 B = 0000 0011 FunSel = 0110 Result is C1 and after rising edge ZCNO error flag is 0100 which means there is carry, which means no borrow, correct result.

In order to XOR A and B; A = 0001 1111 B = 0000 0011 FunSel = 1001 Result is 0001 1100 which is correct. ZCNO did not change after rising edge.

In order to LSL A where A is 6d; A = 1011 0110 FunSel = 1010 Result is C6 = 0110 1100, ZCNO changed to 0100 which correctly showed carry.

In order to ASL A where A is 6d; A = 1011 0110 FunSel = 1100 Result is C6 = 0110 1100, ZCNO changed to 0101 from previous state, which correctly showed overflow.

In order to CSL A where A is 7b; A = 1011 0111 FunSel = 1110 Result is C6 = 0110 1111, ZCNO changed to 0101 which correctly showed carry and overflow.

3.2 PART 2

We will try to load two numbers to memory, sum them up, and we will perform ASL on result and write it to the memory and output IR(8-15).

First, we will set A to 22. For this, we set:
FunSel of part2a: 10
RegSel of part2a: 0001
MuxCSel: 1
sel of memory: 1
str of memory: 1
Others are zero. We send clock for required amount of time.

Now, we will move memory forward. We set:
FunSel of part2b: 10
RegSel of part2b: 001
sel of memory: 1
Others are zero. We send clock once, memory is incremented.

Now, we will set B to 0a. For this, we set:
FunSel of part2a: 10
RegSel of part2a: 0010

FunSel of ALU: 0001
str of memory: 1
sel of memory: 1
All others will be zero.

Memory will be incremented by above commands.

A and B will be added. For this, we set:
FunSel of ALU: 0100
MuxCSel: 1
str of memory: 1
sel of memory: 1
All others zero, output is 2c.

Memory will be incremented by above commands.

Load into 2a:
sel of memory: 1
ld of memory: 1
MuxASel: 01
RegSel of part2a: 0100
FunSel: 01
All others will be zero.

Memory will be incremented by above commands.

Use ASL on input A, which is loaded from memory:
OutASel of part2a: 10
MuxCSel: 1
FunSel of ALU: 1100
str of memory: 1
sel of memory: 1
All others are zero.

We will output this from memory to IR(8-15).
sel of memory: 1
ld of memory: 1
!L/H of part2c: 1
EN of part2c: 1
FunSel of part2c: 11