ISTANBUL TECHNICAL UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

BLG 222E

COMPUTER ORGANIZATION
PROJECT REPORT

PROJECT NO : 1

PROJECT DATE : 12.03.2020

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 first project, we implemented registers of 8-bits and 16-bits, used them to make register banks and more complex register structures. We will explain the input/output logic and will give some test cases for our registers in this report.

File names are as follows:

- **part1_8.circ**
- **part1_16.circ**
- **part2a.circ**
- **part2b.circ**
- **part2c.circ**

2 DESIGN

This project was divided into two main parts, Part 1 in which we created the fundamental registers with 8-bit and 16-bit capacity and Part 2, which was also divided into 3 sub-parts, where we used our implementation in Part 1 to design wanted structures.

2.1 PART 1

We have two main files for this part, **part1_8.circ** and **part1_16.circ**.
In file **part1_8.circ**, two circuits can be seen in logisim, **main** and **1_bit_op**. This **1_bit_op** file is made of one D flip-flop, and one 4:1 MUX. We used this MUX to perform wanted functions, which is CLEAR command in $I_0 = 00$, INPUT command in $I_1 = 01$, INCREMENT command in $I_2 = 10$, DECREMENT command in $I_3 = 11$.

CLEAR is connected to the static 0, and INPUT is the user input. We designed INCREMENT and DECREMENT using a identity from the course. Based on the figure above, we derived the following formula for incrementing

$$Bit_n \oplus (Bit_{n-1} \wedge (Bit_{n-2} \wedge (... \wedge Bit_0))) \tag{1}$$

Also based on the same identity, we complemented the idea and came up with the following identity for decrementing, which toggles bit n when right bits are all zero.

$$Bit_n \odot (Bit_{n-1} \vee (Bit_{n-2} \vee (... \vee Bit_0))) \tag{2}$$

0 0 0
0 0 1
0 1 0
0 1 1
1 0 0
1 0 1
1 1 0
1 1 1
0 0 0

As seen on the truth sequence, LSB toggles on every bit.
For every bit that is on the left, when all bits on
the right of it is 1, in the next state
the value of the bit toggles.

In incrementing, we need least significant bit to toggle every clock cycle, for this reason we connected static 1 to the XOR to make it act like an inverter, and connected static 0 to the XNOR to make it act as an inverter. We keep on AND'ing and OR'ing the right bits and giving them to the 1 bit units, and inside them, they make the XOR or XNOR operation.

We used 8 of these single bit units in main to make our 8 bit register neatly, and used 16 of them to implement 16-bit register.

2.2 PART 2

In **part2a.circ**, we used four of our 8 bit registers to make a register bank. We slightly changed the our original register design, by connecting input pins to places where we had static 0, static 1 and clock signal, so we connected these signals to the register bank on the main circuit.
The register on the rightmost is the R_0, and the one on the left most is R_3. Their SELECTION pins are connected to the same 2-bit FunSel input, and their ENABLE pins are connected to a 4 bit input called RegSel through a splitter.

Outputs are fed into 2 multiplexers. Upper MUX controls the OutA, and controlled by OutASel. Lower MUX controls the OutB, and controlled by OutBSel.

In **part2b.circ**, we made an address register bank. We modified our design from 2a, by removing the leftmost register, making RegSel 3-bit, and connecting the 11 condition of MUX'es to the 00 condition, since it is specified that way in the instructions of the projects.

Leftmost register is SP, in the middle there is AR, and on the rightmost, there is PC.

In **part2c.circ**, we made a 16-bit register that used a 8-bit input and another input to load either lower 8-bits of it or the upper 8-bits. Since we need increment and decrement functions to work, we decided to modify our 16-bit register design from part 1.

For this purpose, we used three splitters and a MUX. First splitter splits the output signal into 2, so we can get outputs of upper 8 bits and lower 8 bits on different wires. Then, we use a splitter to combine lower 8-bits with our input, creating a 16-bit signal that has the current state in its lower 8 bits, and new input in its upper 8 bits. We use another splitter to create another 16-bit signal in a similar way, with current state in upper 8 bits and new input in lower 8 bits. We connect both of these into a 16-bit 2:1 MUX, and connect MUX output the the register input. Finally, we connect $!L/H$ signal to the selection of this MUX, so we can upload 8-bit input into 16-bit register without changing the inner structure.

Since the inner structure is not changed, increment and decrement functions work as intended without any further alterations. Only change we had to make was to remap FunSel/selection input, since in part 2c, order of functions was changed. We achieved this by connecting FunSel into the selection of 2-bit 4:1 MUX, whose inputs are constants. We connected these constants using tools from Wiring library in logisim.

3 TEST CASES

3.1 PART 1

PART 1A

In order to load 4A the binary input is: 01001010 Enable is 1, selection is 01. After the rising edge output is loaded.
To increment the number we input 10 to selection, 1 to enable the binary number input is insignificant.

PART 1B

In order to load 42A2 the binary input is: 01000010 10100010 Enable is 1, selection is 01. After the rising edge output is loaded.
To decrement the number we input 11 to selection, 1 to enable the binary number input is insignificant.

3.2 PART 2

PART 2A

In order to load CA to R0 and R2 we set RegSel to 0101 FunSel to 01 the binary input is 11001010.
To increment R2 and R3 we input 1100 to RegSel 10 to FunSel and binary input is insignificant.
To see the output of R2 at OutA and to see the output of R1 at OutB we set OutA Sel to 10 and outB Sel to 01.

PART 2B

To decrement AR and see the output of SP at OutC and to see the output of AR at OutD we set RegSel to 010 FunSel to 11 OutC Sel to 10 OutD Sel to 01.

PART 2C

To load the decimal value 28 to bits 0-7 the input is 00101000 and the !L/H input is 0, enable is 1 FunSel is 11.
To load the decimal value AA to bits 8-15 the input is 10101010 and the !L/H input is 1, enable is 1 FunSel is 11.
To increment 0x00FF we set FunSel to 01, Enable to 1 and the others are insignificant, output is correctly incremented and is 0x0100.

To decrement 0x0000 we set FunSel to 10, Enable to 1 and the others are insignificant, output is correctly incremented and is 0xFFFF, by design it will not go down to negative numbers instead it will return to the maximum possible value.