lab report 4

Andrew Nady

900184042

Question 2

Technical summary:

steps

- Experiment 1, we used code of the last lab and added to it to make the register file. Then we copied the testbench in the report and made the simulation
- Experiment 2,Inside the ALU, we implemented the functionalities of the processor (add, sub, and, or)
- Experiment 3, we made the control unit which controls other things through certain flags. We implemented these flags according to R-format, LW, SW, BEQ
- Experiment 4, ALU control unit which takes the ALUOP from the control unit and controls the ALU to choose which functionality will implement. We used the table to make the ALU selection according to ever function like add and sub

Components:

- Verilog
- VNC

Code functionality:

- Experiment 1, it reads and write to the registers.
- Experiment 2, it implements the functionalities of the processor like add, sub, or, and
- Experiment 3, it controls what will be selected in many parts in the whole circuit like if it is R-format or LW or SW and so on
- Experiment 4,it control the ALU and selects what should be implemented by the ALU, and it is controlled by the Control unit

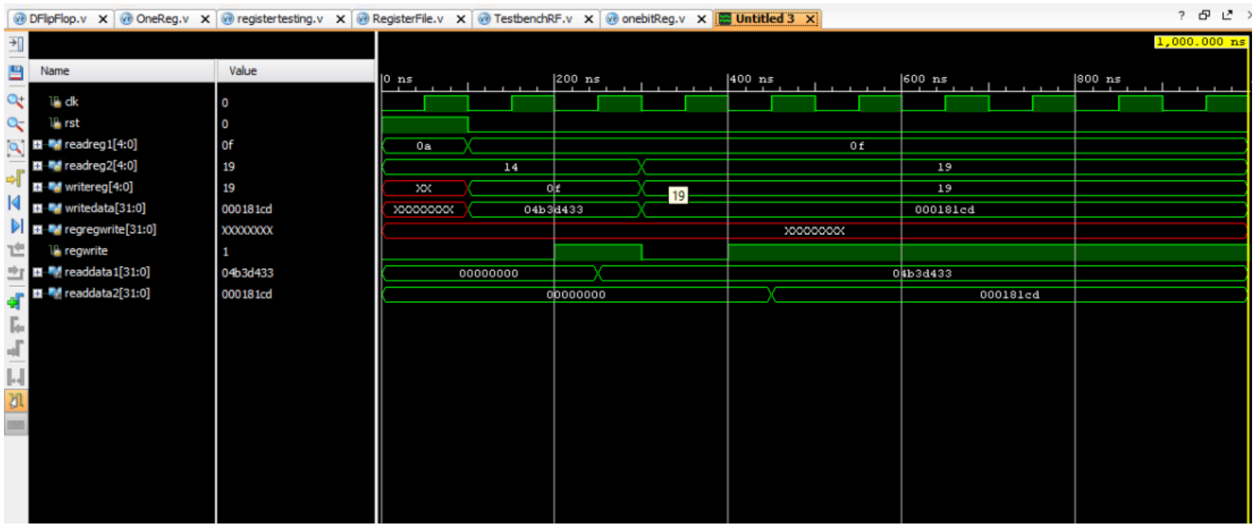
Question 3

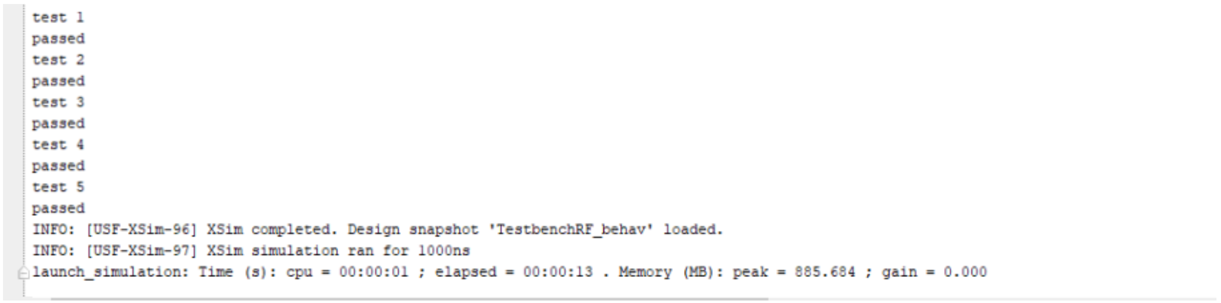
Testbenches and code are included in each experiment folder

Question 4

Simulations:

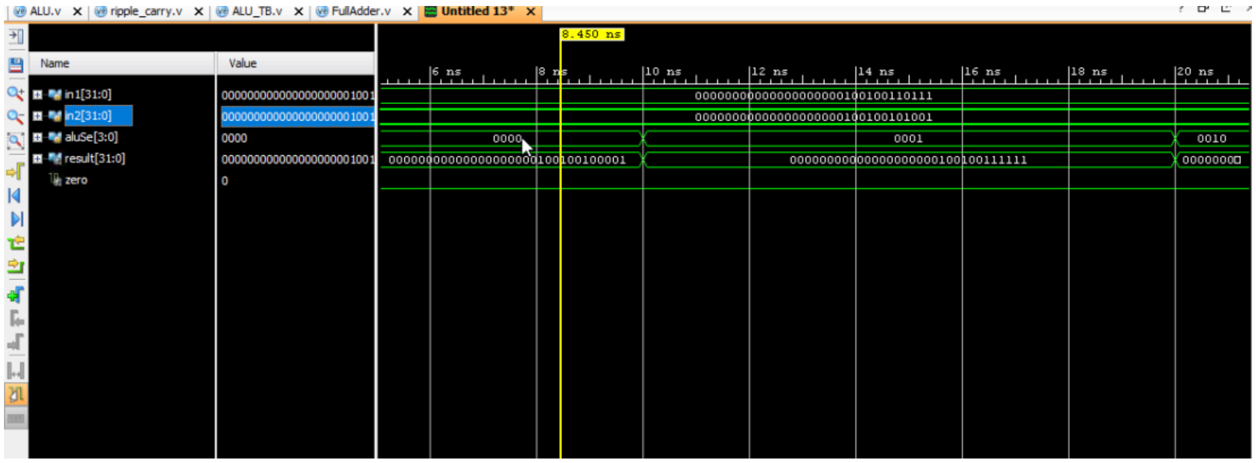
Exp 1:



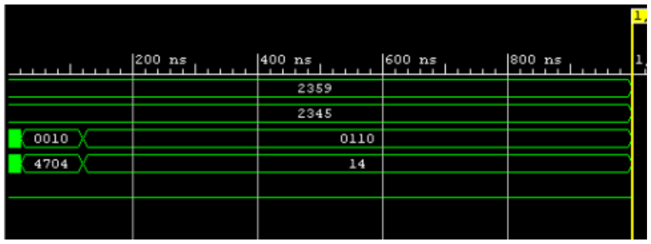


```
test 1
passed
test 2
passed
test 3
passed
test 4
passed
test 5
passed
INFO: [USF-XSim-96] XSim completed. Design snapshot 'TestbenchRF_behav' loaded.
INFO: [USF-XSim-97] XSim simulation ran for 1000ns
launch_simulation: Time (s): cpu = 00:00:01 ; elapsed = 00:00:13 . Memory (MB): peak = 885.684 ; gain = 0.000
```

Exp2:
Or and and



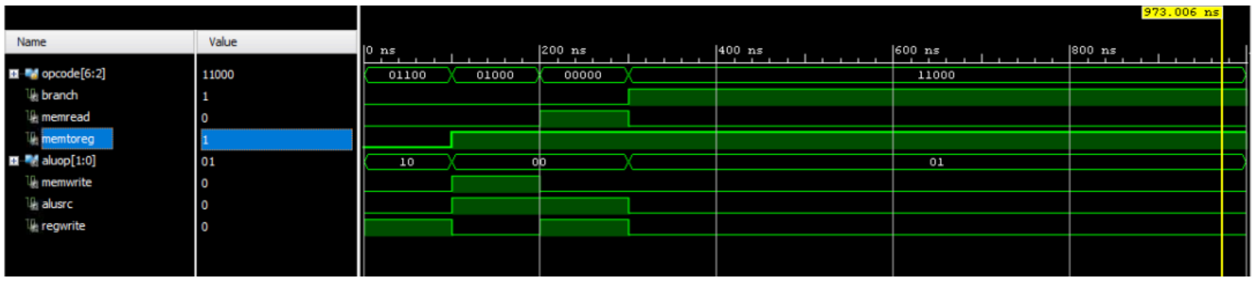
Adding and subtracting



Exp 3:

The results are the same as this table

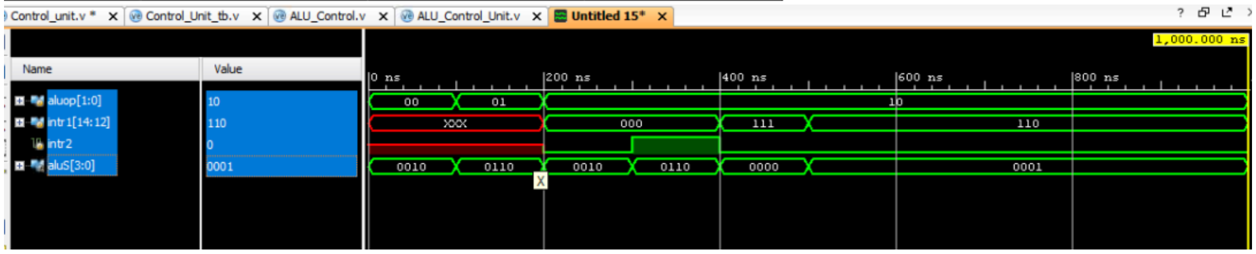
Instruction	Inst[6-2]	Branch	MemRead	MemtoReg	ALUOp	MemWrite	ALUSrc	RegWrite
R-Format	01100	0	0	0	10	0	0	1
LW	00000	0	1	1	00	0	1	1
SW	01000	0	0	X	00	1	1	0
BEQ	11000	1	0	X	01	0	0	0



Ex4:

The results are the same as this table

ALUOp	Inst[14-12]	Inst[30]	ALU Selection
00	X	X	0010 (ADD)
01	X	X	0110 (SUB)
10	000	0	0010 (ADD)
10	000	1	0110 (SUB)
10	111	0	0000 (AND)
10	110	0	0001 (OR)



Question 5

```
`timescale 1ns/1ns
module eightmultione( input [7:0] D, input [2:0] s, output out);
assign out=(D[0]&~s[0]&~s[1]&~s[2])| (D[1]&s[0]&~s[1]&~s[2])|
(D[2]&~s[0]&s[1]&~s[2])| (D[3]&s[0]&s[1]&~s[2])| (D[4]&~s[0]&~s[1]&s[2])|
(D[5]&s[0]&~s[1]&s[2])| (D[6]&~s[0]&s[1]&s[2])| (D[7]&s[0]&s[1]&s[2]);
endmodule
module sixtyfourbit( input [7:0] D [63:0], input [2:0] s[63:0], output
[63:0] out);
genvar i;
generate
for (i=0;i<=63;i=i+1)
begin
eightmultione(  [7:0] D[i], [2:0] s[i], out[i]);
 end
  endgenerate
endmodule
```

Question 6

- to make it simple processor register are much faster than cache memory, and unlike cache memory that store data, the processor register store **instructions** that manipulate data, by instruction i mean (address, opcode, small chuck of data that we need to operate in).
- Registers are controllable, you can store and retrieve information from them. There are very few of them but very fast. A lot of them have very particular uses (Instruction Pointer, Base Pointer, etc) and should not be used by the user.
- Cache is almost completely uncontrollable. You can invalidate it but you cannot explicitly store or retrieve information from it. It is also placed between memory and the CPU, so you don't even know whether it's working or not, unless you do timing comparisons. It is intended to be completely transparent in operations. Also, they can be hierarchical and fairly large (comparatively to registers at least).

So basically, the difference:

registers are:

- Few in number
- Limited in size
- The only things most processors can operate on directly

Cache is:

- Larger in quantity (512 bytes or more)
- Not directly accessible for operations (just a pool between the CPU and main store)
- Extant (32kB+)

Question 7

Code:

```
`timescale 1ns / 1ps

module ALU (
input [31:0] in1, in2,
input [3:0] aluSel,
output reg [31:0] result, output zero,output reg overflow_flag );
wire c1, c2;
wire [31:0]sum;
ripple_carry plus( in1,in2, 0,sum,c1);
wire [31:0]sub;
ripple_carry subt( in1,~in2+1, 0,sub,c2);
integer i;

always @(*)
begin
case(aluSel)
4'b0010:
begin
result= sum;
if(c1==1)
overflow_flag=1;
else
overflow_flag=0;
end

 //2 ;
4'b0110 :
begin
result= sub ;
if(c2==1)
overflow_flag=1;
else
overflow_flag=0;
end    //sub6


4'b0000 : result= in1 & in2  ; //and
4'b0001 :  result= in1 | in2 ;//or
4'b0011:  result= in1 ^ in2 ; //Xor
//logical right
4'b0100:
begin
for(i=0;i<in2;i=i+1)
 result = {1'b0,in1[31:1]};
end


//logical left
```

```
4'b1111:
begin
for(i=0;i<in2;i=i+1)
 result = {in1[30:0],1'b0};
end

//arth right

4'b1100:
begin
for(i=0;i<in2;i=i+1)
 result = {{in1[31]?1'b1:1'b0},in1[31:1]};
end


default : result=0;
endcase
end
  assign zero={result==0?1:0};
endmodule
```