

EE 471 Lab 3 Report

Designing an ALU

Working with an SRAM, Reg File, I/O, and C Stuff Cont....

Lab performed by:

David Zhu

Rebecca Chu

Phongsakorn

Table of Contents

ABSTRACT:	2
INTRODUCTION:	2
DISCUSSION OF THE PROJECT:	2
Design Specification:	2
Design Procedure:	3
System Description:	3
Software Implementation:	5
Hardware Implementation:	8
TEST PLAN:	12
TEST SPECIFICATION:	12
TEST CASES:	14
ANALYSIS OF RESULTS:	15
ERROR ANALYSIS:	19
SUMMARY AND CONCLUSION:	19
APPENDIX:	20
Intergration of SRAM, Register File, MDR, MAR and ALU	20
C Code: Pointers	49

ABSTRACT:

For this project we created the ALU component of the computer. Our ALU can perform the no-op, add, subtract, and, or, xor, set-less-than, and shift left logical operations.

To test our ALU we implemented it onto the Cyclone-II chip and used switches and manual testing as well as signal tap output waveforms to confirm the correctness of its functionality.

Upon testing our design we were able to confirm that our ALU is capable of performing the functionalities required of it.

INTRODUCTION:

The task for this project was to create an ALU for our computer and our approach to the creation of this component was to first plan out the functionalities and create the modules that would perform the functions one at a time.

Once the modules for each function were created, we put them inside the ALU module and the ALU module would be in charge of taking in the inputs and translating them into the function that would be performed. Lastly, the ALU would use the respective module for the function and perform that said function.

We used verilog to write all of the modules for our ALU, and using signal tap and the quartus environment we tested out module, and in the end confirming that it functions correctly.

In the last part of the project, we used the C language to learn more about C variables and their addresses.

DISCUSSION OF THE PROJECT:

Design Specification:

The requirements of the ALU are that it performed a number of functions. The functions required are:

- no-op,
- addition,
- subtraction,
- and,
- or,
- xor,
- set-less-than,
- shift left logical.

In addition to performing the required functions, the ALU must also perform all of these in a single clock cycle. The functions themselves must also be created using dataflow or structural level verilog code.

The final ALU module must be able to:

- load data and instructions from the SRAM
- transfer data from the SRAM to the register file
- be able to read and interpret the instruction given to it
- perform the requested function
- store the result in the register file

Design Procedure:

When designing our ALU we first started off by making individual modules for each one of the supported functions. In order to design each and every one of them, we first decided on a design pattern that would allow the function to be performed in a single clock cycle. The shifter in particular was made using a barrel shifter design in order to meet the single clock cycle requirement. Once the design was decided on we proceeded to writing them in verilog.

Once the modules were created we began testing them to ensure that they correctly performed the function. To test them we created test benches in bugHunter and used the bugHunter environment to display the outputs of the modules. When a bug was detected we debugged them in order to ensure correctness.

Once we were sure that the individual function modules were correct, we integrated them together into one module which would be the ALU.

The ALU module is tasked with receiving inputs from the user and decoding the instructions given to it and performing the correct requested function. We wrote the ALU such that it was able to receive an instruction, interpret it, and perform the action that is requested. In order to test our ALU, we first created a state machine that would be loaded onto the Cyclone-II chip and be tested using switch inputs and LED light outputs. Once we were able to confirm correct functionality of the ALU, we continued onto integrating it with the SRAM and register file.

We started first by establishing how the ALU would interact with the SRAM and register file. Once that was established we wrote a state machine that would be able to perform the functions and correctly move data and instructions to and from register file and SRAM. The correctness of the integration was confirmed using signal tap.

System Description:

ALU:

inputs - run, clk, rst, A, B, control

outputs - zero, carryOut, overflow, negative, result

The ALU takes the inputs to perform the function on. The operands used for the operations are A and B, each a 32 bit 2's complement number, and the function to perform is specified by the control input, which is an unsigned number from 0 to 7. The clk is used to clock the module, and rst is used to perform a reset on the module, clearing all data. The run signal is used to tell the ALU to perform the function on the current operands; a 0 indicates to perform the computation.

The ALU has outputs which include certain flags and the result from the function computation. The flags indicate information as follows:

- zero - the zero flag, which is given a value of 1 when the result of the function is zero and 0 otherwise
- carryOut - the carry out flag, which has a value of 1 when there is a carryout as a result of the function being performed, otherwise it is 0
- overflow - the overflow flag, this flag is given a value of 1 when the function performed causes an overflow of data in the result and 0 otherwise.
- negative - negative flag, flagged with a 1 when the result is negative and 0 otherwise

The ALU performs a number of operations and they are as follows:

- no-op: does no operation and always returns a result of 0
- add: adds B to A
- subtract: subtracts B from A
- and: performs a bitwise “and” on A and B
- or: performs a bitwise “or” on A and B
- xor: performs a bitwise “xor” on A and B
- set less than: returns a result of 1 when A is less than B and 0 otherwise
- shift left logical: performs a logical shift on A to the left by number of bits specified by the value of B. The maximum number of bits the function shifts is 3.

The only timing constraint on the ALU is that it must perform the functions each in a single cycle of the clock. In order to meet the time constraints of the single clock cycle, we use carry look ahead for the addition and subtraction operation. Also we used barrel shifter to do the shift left operation.

Error handling:

No explicit error handling functionality is built into our ALU and therefore if the inputs to the module were erroneous then the final output and results that are outputted are not well defined. That being said it is advised that the user follow the input guidelines closely and use only inputs that have been well defined by our design in the system description section above.

Software Implementation:

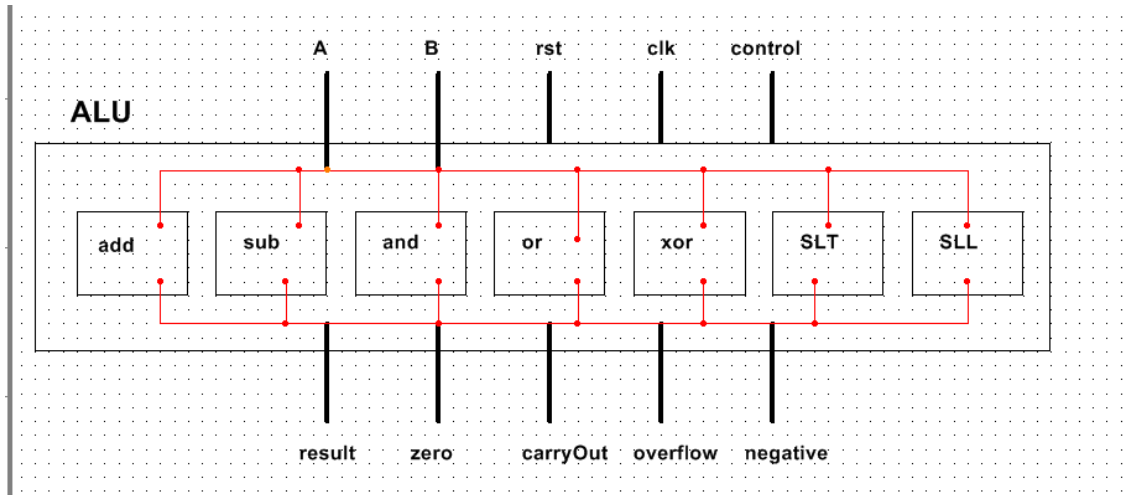


Figure 1.4.1: Block diagram of ALU

As can be seen in figure 1.4.1 above, our overall ALU design is a single module which is the ALU, and inside the ALU are individual modules, one each, for each of the functions that we must perform.

The adder module performs the addition operation on operands A and B. The subtract module performs a subtraction operation on A and B, namely subtracting B from A. The logicAnd module performs a bitwise logical and on A and B. Likewise, the logicOr module performs a logical or on A and B. Similarly, the logicXor module performs a logical xor on the operands A and B. The SLT module performs a “set less than” operations on A and B. Namely, it sets the result to be 1 if A is less than B and 0 otherwise. The barrel shifter module performs a logical shift left on A by a certain number of bits which is specified by the value of B. The maximum number of bits that it can shift is 3. Additionally, in order to meet specifications the shifter was designed using a barrel shifter.

When the ALU module is used, it takes in the operands A and B, and a 3 bit control. It takes the 3 bit control input and uses it to interpret which function to perform, the exact function to perform is decided as follows:

- 0 : no op
- 1 : add
- 2 : subtract
- 3 : and
- 4 : or
- 5 : xor
- 6 : set less than
- 7 : shift left logical

Once deciding which function was selected, the ALU will pass the values of A and B into the respective module to perform the function.

Once the specific module has performed its function, a result will be returned to the ALU. This result is then outputted back to the user. Additionally, the ALU will trigger certain flags to have a value of 1 depending on whether or not certain conditions were met. Together with the result, the flag values are also outputted back to the user.

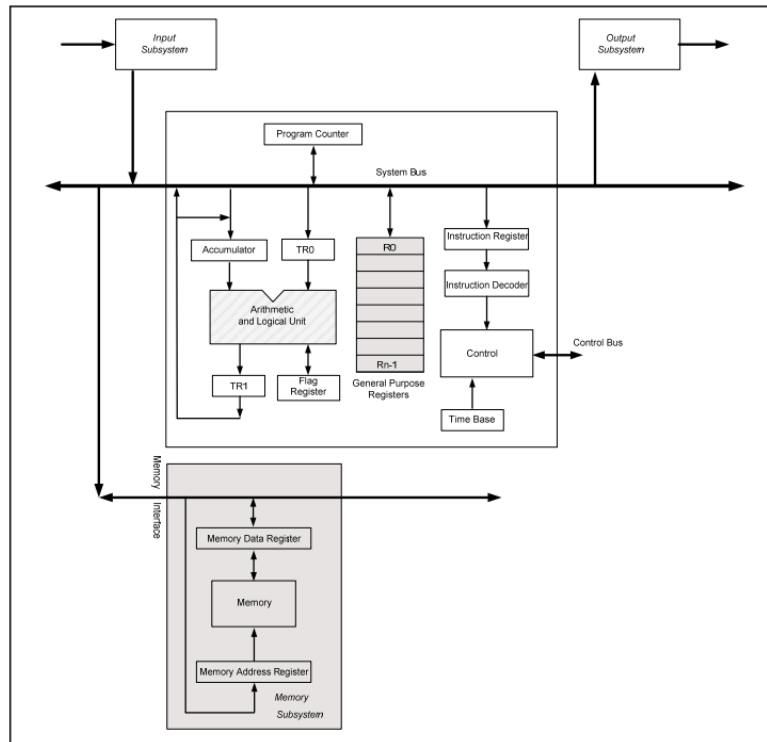


Figure 1.4.2: Overall integration of the system

Hardware Implementation:

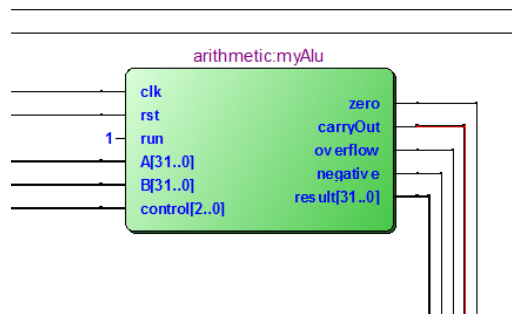


Figure 1.5.1: Inputs and output of ALU

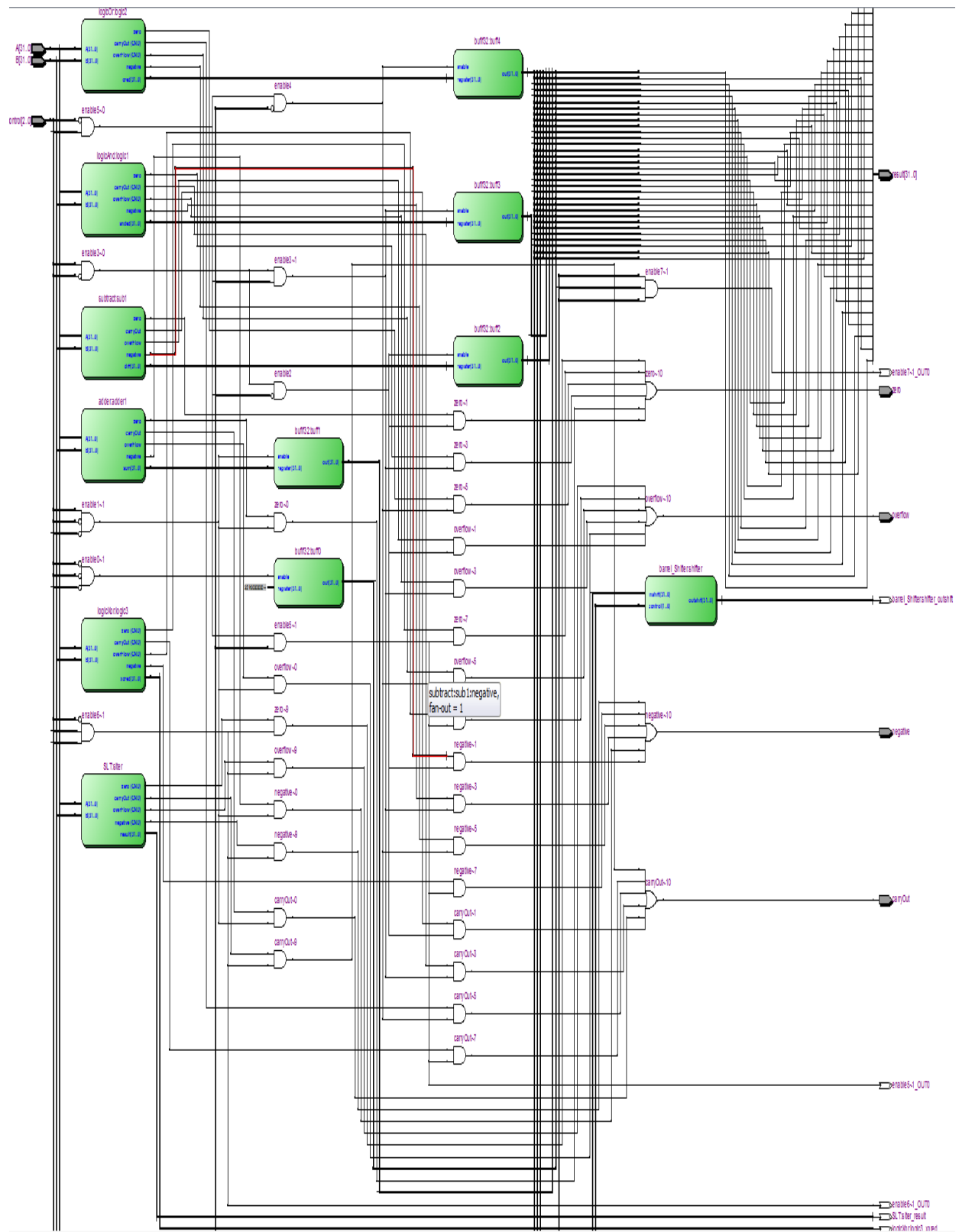


Figure 1.5.2: The connection of operations inside ALU

Switches that we implemented on DE1 board:

SW3...SW0

Enter four hex digits as the data for operands A and B.

SW6...SW4

Specify the operation / control code.

- 3 of 8 - SW9... SW8

Specify whether operand A or B is to be entered or the result is to be displayed.

KEY [0]

Interpreted by the system as an ENTER. When the ENTER is pressed, the ALU will read the state of the input switches and respond accordingly.

KEY[1]

The RUN command, direct the ALU to perform the specified operation and display the results.

Hex3...Hex0

Display the operands or the results of the operation in hex, on the, on the DE1 board.

carryOut	Output	PIN_U21	6	B6_N1	3.3-V LV...default)	24mA (default)	
ce	Output	PIN_AB5	8	B8_N1	3.3-V LV...default)	24mA (default)	
clk	Input	PIN_D12	3	B3_N0	3.3-V LV...default)	24mA (default)	
data_bus[15]	Bidir	PIN_U8	8	B8_N1	3.3-V LV...default)	24mA (default)	
data_bus[14]	Bidir	PIN_V8	8	B8_N1	3.3-V LV...default)	24mA (default)	
data_bus[13]	Bidir	PIN_W8	8	B8_N1	3.3-V LV...default)	24mA (default)	
data_bus[12]	Bidir	PIN_R9	8	B8_N0	3.3-V LV...default)	24mA (default)	
data_bus[11]	Bidir	PIN_U9	8	B8_N0	3.3-V LV...default)	24mA (default)	
data_bus[10]	Bidir	PIN_V9	8	B8_N1	3.3-V LV...default)	24mA (default)	
data_bus[9]	Bidir	PIN_W9	8	B8_N0	3.3-V LV...default)	24mA (default)	
data_bus[8]	Bidir	PIN_Y9	8	B8_N0	3.3-V LV...default)	24mA (default)	
data_bus[7]	Bidir	PIN_AB9	8	B8_N0	3.3-V LV...default)	24mA (default)	
data_bus[6]	Bidir	PIN_AA9	8	B8_N0	3.3-V LV...default)	24mA (default)	
data_bus[5]	Bidir	PIN_AB8	8	B8_N0	3.3-V LV...default)	24mA (default)	
data_bus[4]	Bidir	PIN_AA8	8	B8_N0	3.3-V LV...default)	24mA (default)	
data_bus[3]	Bidir	PIN_AB7	8	B8_N1	3.3-V LV...default)	24mA (default)	
data_bus[2]	Bidir	PIN_AA7	8	B8_N1	3.3-V LV...default)	24mA (default)	
data_bus[1]	Bidir	PIN_AB6	8	B8_N1	3.3-V LV...default)	24mA (default)	
data_bus[0]	Bidir	PIN_AA6	8	B8_N1	3.3-V LV...default)	24mA (default)	
lb	Output	PIN_Y7	8	B8_N1	3.3-V LV...default)	24mA (default)	
negative	Output	PIN_U22	6	B6_N1	3.3-V LV...default)	24mA (default)	
oe	Output	PIN_T8	8	B8_N1	3.3-V LV...default)	24mA (default)	
overflow	Output	PIN_V22	6	B6_N1	3.3-V LV...default)	24mA (default)	
sram_we	Output	PIN_AA10	8	B8_N0	3.3-V LV...default)	24mA (default)	
top_start	Input	PIN_L22	5	B5_N1	3.3-V LV...default)	24mA (default)	
top_state[4]	Output				3.3-V LV...default)	24mA (default)	
top_state[3]	Output				3.3-V LV...default)	24mA (default)	
top_state[2]	Output				3.3-V LV...default)	24mA (default)	
top_state[1]	Output				3.3-V LV...default)	24mA (default)	
top_state[0]	Output				3.3-V LV...default)	24mA (default)	
ub	Output	PIN_W7	8	B8_N1	3.3-V LV...default)	24mA (default)	
zero	Output	PIN_V21	6	B6_N1	3.3-V LV...default)	24mA (default)	
<<new node>>							
rst	Input	PIN_L21	5	B5_N1	3.3-V LV...default)	24mA (default)	
sramOut[31]	Output				3.3-V LV...default)	24mA (default)	
sramOut[30]	Output				3.3-V LV...default)	24mA (default)	

Figure 1.5.3: Pin assignment for the DE1 board

Logic Equations

ENABLE from Controls:

```

assign enable0 = ~control[2] & ~control[1] & ~control[0];
assign enable1 = ~control[2] & ~control[1] & control[0];
assign enable2 = ~control[2] & control[1] & ~control[0];
assign enable3 = ~control[2] & control[1] & control[0];
assign enable4 = control[2] & ~control[1] & ~control[0];
assign enable5 = control[2] & ~control[1] & control[0];
assign enable6 = control[2] & control[1] & ~control[0];
assign enable7 = control[2] & control[1] & control[0];

```

Negative Flag

assign negative = (negative0 & enable1) | (negative1 & enable2) | (negative2 & enable3) |
(negative3 & enable4) | (negative4 & enable5) | (negative5 & enable6) ;
assign overflow = (overflow0 & enable1) | (overflow1 & enable2) | (overflow2 & enable3) |
(overflow3 & enable4) | (overflow4 & enable5) | (overflow5 & enable6) ;
assign carryOut = (carryOut0 & enable1) | (carryOut1 & enable2) | (carryOut2 & enable3) |
(carryOut3 & enable4) | (carryOut4 & enable5) | (carryOut5 & enable6) ;
assign zero = (zero0 & enable1) | (zero1 & enable2) | (zero2 & enable3) | (zero3 &
enable4) | (zero4 & enable5) | (zero5 & enable6)

Overflow Flag

Overflow1 = Co31^Co30

Adder

For Carry out bits

Co1 = Co0 & (A1^B1) | (A1 & B1)

For Sum

Sum1 = A1^B^Co0;

For comprehensive logic equations for adder, please see the Appendix

And

And1 = A1 & B1

For comprehensive logic equations for And, please see the Appendix

Or

Or1 = A1 | B1

For comprehensive logic equations for Or, please see the Appendix

TEST PLAN:

For the ALU, to test whether the design is working as expected, each of the 8 operations needs to be tested to see if their inputs produce the results according to their control. Of which, each operation's flags also need to be checked to see if the output is correct. The system should also recognize and deal with signed numbers using two's complement as negative numbers.

TEST SPECIFICATION:

Enter in a set of two 16 bit binary numbers, A and B, and with selected control, the output should be the result according to the control with relevant flags set. The shown inputs are arbitrary numbers to see the result and flags.

- Input A: 16'd1 //1
 - Input B: 16'b1111111111111111 //-1 (in two's complement)
- Addition:
 - Control: 001
 - Result: 0
 - Flags:
 - negative: 1
 - carry out: 1
 - overflow: 0
 - zero: 1
 - Constraints: can only enter 16 bit inputs, complete in one clock cycle
- Subtraction:
 - Control: 010
 - Result: -2
 - Flags:
 - negative: 1
 - carry out: 1
 - overflow: 0
 - zero: 0
 - Constraints: can only enter 16 bit inputs, complete in one clock cycle
- AND:
 - Control: 011
 - Result: 0000000000000001
 - Flags:
 - negative: 0
 - carry out: 0
 - overflow: 0
 - zero: 0
 - Constraints: can only enter 16 bit inputs, complete in one clock cycle
- OR:
 - Control: 100
 - Result: 1111111111111111
 - Flags:
 - negative: 1
 - carry out: 0
 - overflow: 0
 - zero: 0
 - Constraints: can only enter 16 bit inputs, complete in one clock cycle
- XOR:
 - Control: 101

- Result: 1111111111111110
- Flags:
 - negative: 1
 - carry out: 0
 - overflow: 0
 - zero: 0
- Constraints: can only enter 16 bit inputs, complete in one clock cycle
- SLT:
 - Control: 110
 - Result: 0
 - Flags:
 - negative: X
 - carry out: X
 - overflow: X
 - zero: X
 - Constraints: can only enter 16 bit inputs, complete in one clock cycle
- SLL:
 - Control: 111
 - Result: 1111111111111110
 - Flags:
 - negative: 1
 - carry out: 0
 - overflow: 0
 - zero: 0
 - Constraints: can only enter 16 bit inputs, complete in one clock cycle
 - Limit: B input cannot exceed 3, and has to be positive integers

TEST CASES:

There are two ways to test if the system works. The first one is to put the code onto the DE1 board through quartus with the required pin assignment as detailed in the specs. Test the cases with the following steps:

- Select one of the eight controls and switch to it in SW 6-4
- Switch SW9...SW8 both to zero
- Enter in the first input (i.e. input A) as specified in the Test Specification. Enter the first least four significant digits first in SW3...0. Press enter in Key [0]. Repeat to most significant four bits of input A
- Switch SW9...SW8 to 0 and 1 respectively to enter input B
- Enter in the first input (i.e. input B) as specified in the Test Specification. Enter the first least four significant digits first in SW3...0. Press enter in Key [0]. Repeat to most significant four bits of input B

- Following these steps, the result should be shown on the display in HEX. Check the binary equivalent of this hex to see if they are correct as anticipated.
- To test the limits, see what happens when enter more than 3 for input B for SLL for testing false conditions.

The second way is to write a test bench for each function of the ALU and run it in bugHunter. This should result in produce a waveform and monitor each flag bits and the result from each control.

ANALYSIS OF RESULTS:

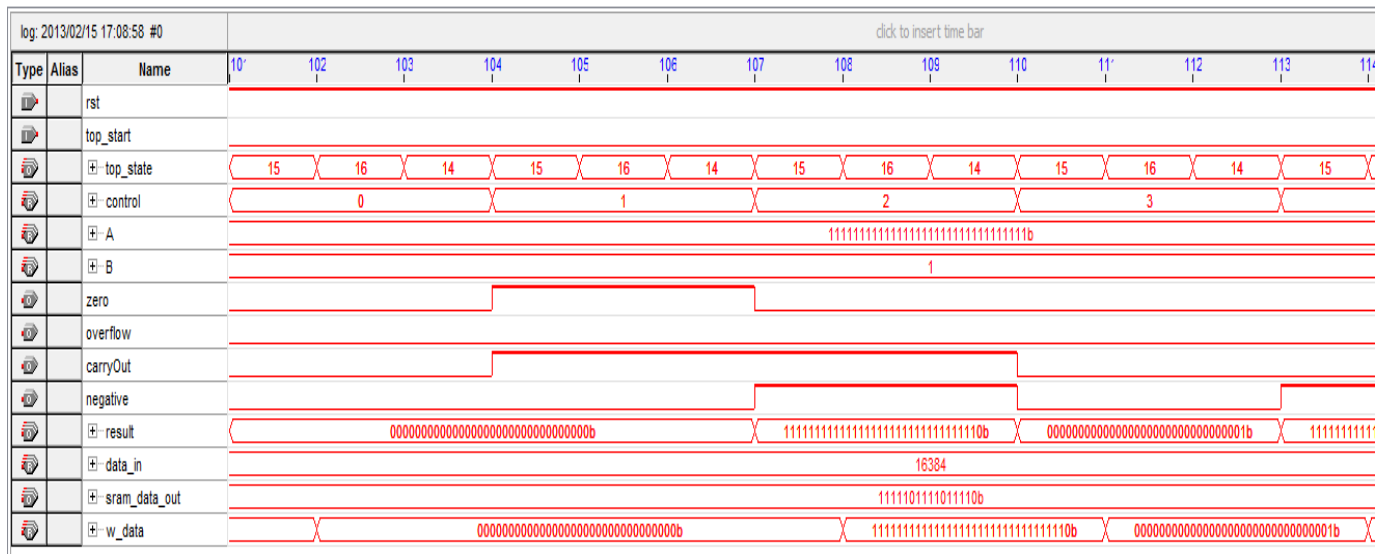


Figure 5.1: Signal tap waveform of operation 0-3.

From Figure 5.1, the result which is the sum of the operation proves that the operations work correctly. The control indicated the operation that is being performed.

The control:

- 0 : no op
- 1 : add
- 2 : subtract
- 3 : and
- 4 : or
- 5 : xor
- 6 : set less than
- 7 : shift left logical

w_data, shows that the result is being store in the new registered file. data_in shows that the data is being passed into SRAM. Ainput is -1 and Binput is +1

- $-1 + 1 = 0$. This would flag zero and carryout
- $-1 \text{ Sub } 1 = -2$ This would flag carryOut and negative.
- $-1 \text{ and } 1 = \dots 000001$

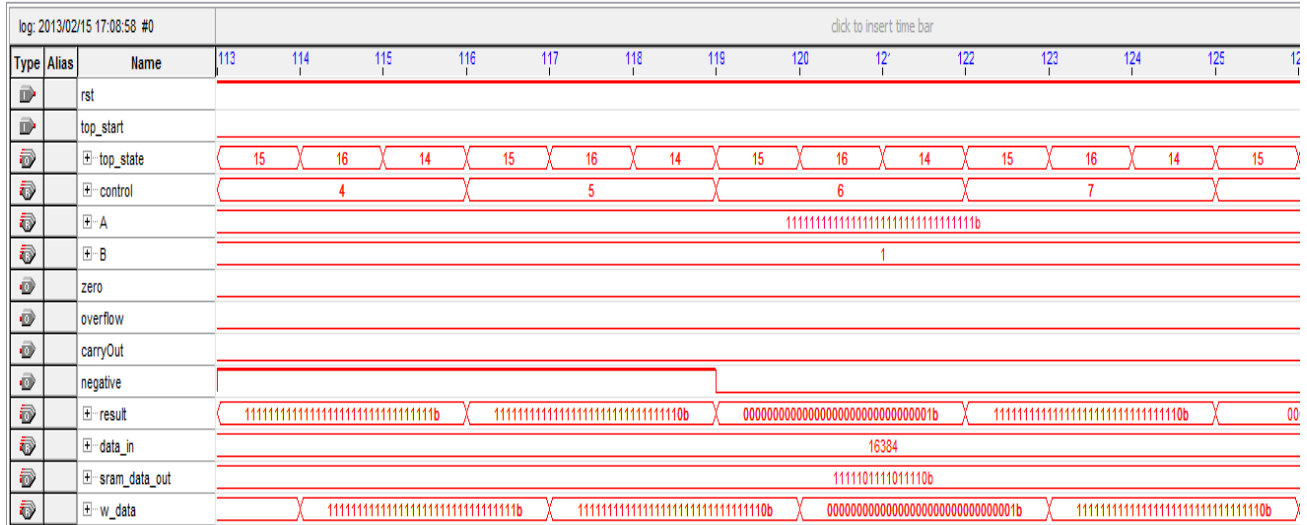


Figure 5.2: Signal tap waveform of operation 4-7.

- $-1 \text{ or } 1$ would result in $\dots 11111111$ with negative flag
- $-1 \text{ xor } 1$ would result in $\dots 111111110$ with negative flag
- $-1 \text{ SLT } 1$ would result in 1 because -1 is lesser than 1 .
- $-1 \text{ shift by } 1 \text{ bit}$ would result in $\dots 111111110$

The following waveforms are test results from bugHunter for ALU. Figure 5.3 to figure 5.8 displays the operations of adder, subtract, and, or, xor, and SLT with different test inputs A and B to see if the result and flags are set right.

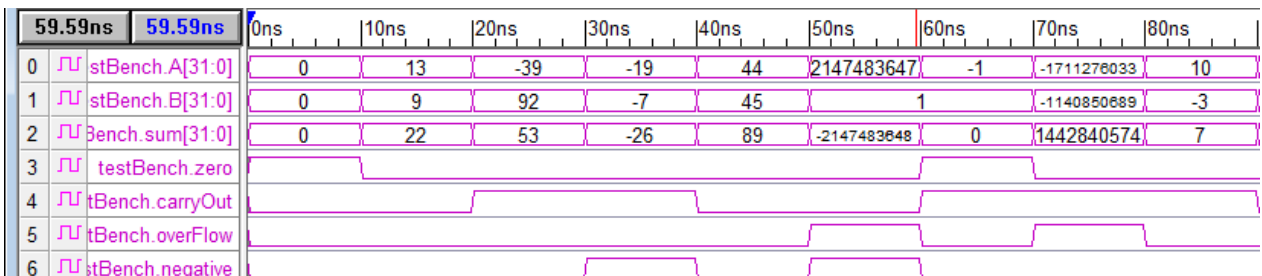


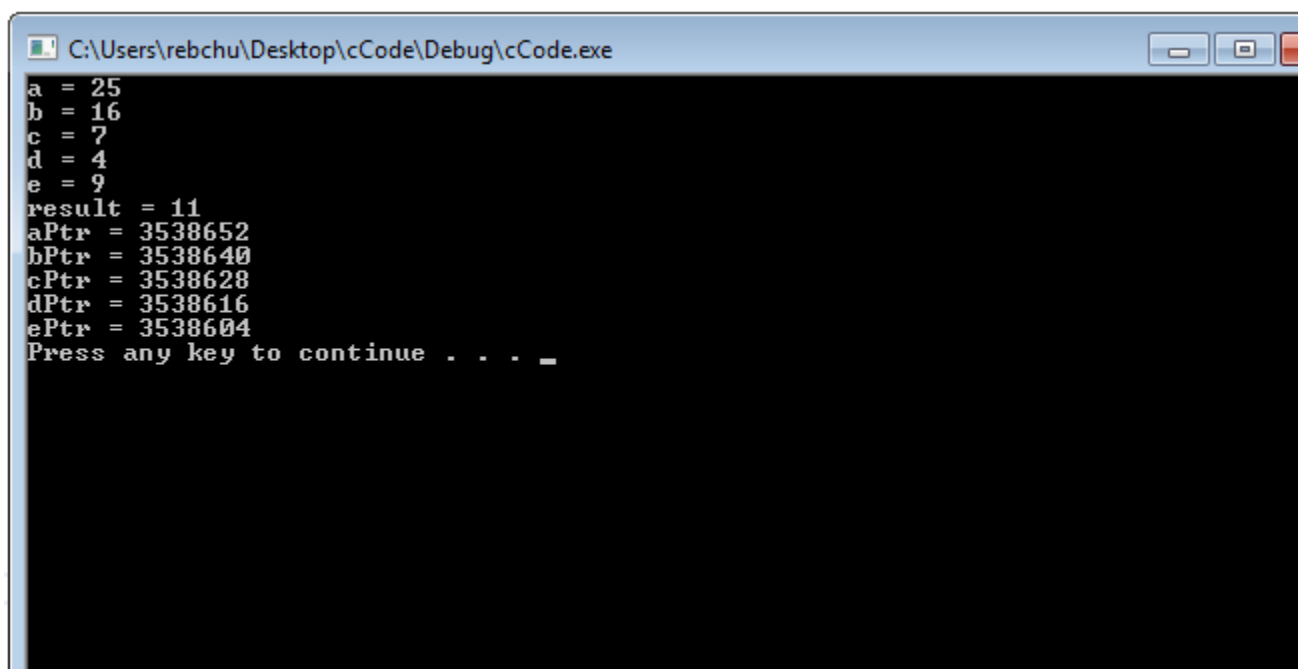
Figure 5.3: BugHunter waveform for adder operation

C Code Pointers

For the C code, the pointer variables of xPtr and yPtr just after the declaration are nothing because no address has been assigned to it.

I think having `int z = *yPtr` and `float w = *xPtr` instead will make z equal to the integer equivalent of float y, and w equal to the float equivalent of int x. In other words, z will have the value of 3 (truncated decimals) and w will have 9.000 (with decimals).

For the second part of the C code, the operation with the c code yields the following result as shown in the screenshot below:

A screenshot of a Windows command prompt window titled "C:\Users\rebchu\Desktop\cCode\Debug\cCode.exe". The window displays the output of a C program. The output shows the values of variables a, b, c, d, and e, followed by the value of result, and then the addresses stored in pointer variables aPtr, bPtr, cPtr, dPtr, and ePtr. The addresses are 3538652, 3538640, 3538628, 3538616, and 3538604 respectively. The prompt "Press any key to continue . . . _" is visible at the bottom.

```
C:\Users\rebchu\Desktop\cCode\Debug\cCode.exe
a = 25
b = 16
c = 7
d = 4
e = 9
result = 11
aPtr = 3538652
bPtr = 3538640
cPtr = 3538628
dPtr = 3538616
ePtr = 3538604
Press any key to continue . . . _
```

Figure 5.9: C code screenshot displaying pointer operation

ERROR ANALYSIS:

Throughout the project, we have encountered many problems with the DE1 board and quartus, when debugging the projects. One of the biggest challenges is the internal access time error. We were able to compile the program in Bug Hunter but not in quartus. We thought that the error might be due to the quartus program itself.

After hours of debugging the program and finding the error, we have identified that it was some problems with the “include”, which we used to link the module together. After we removed the include and added the entire sub module in the library of quartus II we were able to compile the program successfully.

SUMMARY AND CONCLUSION:

All in all, our ALU is able to perform all the required operations. We integrated successfully our ALU with SRAM, register file, MDR and MAR. Upon testing the ALU with switches, LEDS and signal tap, we were able to confirm that our ALU was capable of performing the required functionalities.

Although we faced some difficulties with the internal error message we were able to figure out the problem by doing line by line debugging of code. After figuring out the error, we were able to integrate the ALU successfully. Also, we have learnt more about pointer and addresses in the last part of the project using C code.

APPENDIX:

Intergration of SRAM, Register File, MDR, MAR and ALU

```
module IntegrateALU(clk, rst, top_start, top_state, sram_data_out, addr_out, result,
                   data_bus, sram_rw, sram_we, ce, oe, ub, lb, sram_state,
                   reg_state, zero, carryOut, overflow, negative, regIn, sramOut);

    //general parameters
    input clk; // input clk for timing
    input rst; // active low reset
    input top_start; // start signal for the top level module
    reg sram_done, reg_done, reg_start, reg_we; // for sram and register; 1 = done, 0 = not
done
    reg [4:0] w_sel;
    reg [31:0] A, B;
    reg [2:0] control;
    reg [31:0] w_data;

    output reg [4:0] top_state;

    /*reg [25:0] tBase;
    always@(posedge clk) tBase = tBase + 1'b1; 0*/

    ////////////
    //SRAM i/o
    ////////////
    reg sram_start; // 0 = start, 1 = stop
    output reg sram_rw; // input to determine to read or to write; 1 = read, 0 = write

    output [15:0] sram_data_out; // data from the SRAM reg for data to be read out to LED's
    output [4:0] addr_out; // org addr_out | addr bits that goes to the sram
    inout [15:0] data_bus; // 16-bit data bus to transfer data between interface and SRAM

    reg [15:0] data_in; // org data_in| 16 bit data input bus
    wire [15:0] instr_code; // org data_in| 16 bit data input bus
    reg [15:0] instr_code1;
    reg [2:0] opcode = 0;
    reg [4:0] addA_sram = 5'd1;
    reg [4:0] addB_sram = 5'd2;
    reg [4:0] addA_reg = 5'd1;
    reg [4:0] addB_reg = 5'd2;
```

```

reg [4:0] addResult_reg = 5'd3;
reg [4:0] addIS = 5'd3;
reg [31:0] dataA_in = /*32'd7;*/32'b111111111111111111;
reg [31:0] dataB_in = /*32'd3;*/32'd1;
reg [17:0] addr_in;
output [31:0] result;
reg [31:0] sramIn;
reg [31:0] dataA_in1, dataB_in1;

output zero, carryOut, overflow, negative;

output sram_we; // active low write control
output ce; // active low chip select
output oe; // active low output enable
output ub; // active low upper byte select
output lb; // active low lower byte select

output [4:0] sram_state;
output [3:0] reg_state;

//MAR, MDR
wire [17:0] mar_addr_out;
reg [17:0] mar_addr_in;
reg mar_en = 1;
output [31:0] regIn, sramOut;

parameter IDLE = 0, INITIALSRAM = 1, WRITEA1 = 2, WRITEA2 = 3, WRITEB1 = 4,
WRITEB2 = 5, WRITEB3 = 6,
WRITEI1 = 7, WRITEI2 = 8, FINISH = 9, INITIALREG = 10,
SIGNEXTEND = 11, WRITEREGA = 12,
WRITEREGB = 13, COMPUTE = 14, STORE = 15, CHANGEOP = 16,
READSRAMA1 = 17, READSRAMA2 = 18,
READSRAMA3 = 19, READSRAMB1 = 20, READSRAMB2 = 21,
READSRAMB3 = 22, READSRAMIS1 = 23,
READSRAMIS2 = 24, READSRAMIS3 = 25;

instruction myInstr(opcode, addA_sram, addB_sram, instr_code);

decoderIS myDecoder(instr_code1, opcode_out, addA_sram_out, addB_sram_out);

sram_inter mySram(clk, rst, sram_start, sram_rw, ce, sram_we,
oe, ub, lb, sram_done, data_bus, addr_in, addr_out,
sram_state, data_in, sram_data_out);

```

```

register_inter myRegister(reg_we, clk, rst, reg_start, reg_done, r_sel1, r_sel2, w_sel,
                        r_data1, r_data2, w_data, reg_state);

arithmetic myAlu(result, zero, carryOut, overflow, negative, A , B,
                control, clk, rst, run); // run ALU

mar myMar(mar_addr_in, mar_addr_out, clk, rst, mar_en);
mdr myMdr(clk, rst, sram_rw, regIn, regOut, sramIn, sramOut, out);

always@(posedge clk or negedge rst) // idol state
begin
    if(rst == 0)
        begin
            // general signals
            reg_start = 1; // these might be throwing things off so
check these by commenting out if things are awry
            sram_start = 1;
            top_state = IDLE;

        end
    else
        begin
            case(top_state)
                IDLE:begin
                    if(top_start == 0)
                        begin
                            top_state = INITIALSRAM;
                        end
                    else
                        begin
                            top_state = IDLE;
                        end
                end

                INITIALSRAM:begin
                    sram_start = 0;          // start sram
                    sram_rw = 0; // 0 = write
                    top_state = WRITEA1;
                end

                //put in A
                WRITEA1:begin
                    top_state = WRITEA2;

```

```

        end

WRITEA2:begin
    mar_addr_in = addA_sram;
    addr_in = mar_addr_out;

    data_in = dataA_in;

    if(sram_state == 5) // state w4 of sram
        begin
            top_state = WRITEB1;
        end
    else
        begin
            top_state = WRITEA2;
        end
    end
end

//put in B

WRITEB1:begin
    top_state = WRITEB2;
end

WRITEB2:begin
    mar_addr_in = addB_sram;
    addr_in = mar_addr_out;

    data_in = dataB_in;

    if(sram_state == 5) // state w4 of sram
        begin
            top_state = WRITEB3;
        end
    else
        begin
            top_state = WRITEB2;
        end
    end
end

WRITEB3:begin
    top_state = WRITEI1;
end

```



```

WRITEI1:begin
    top_state = WRITEI2;
end

//put in instruction
WRITEI2:begin
    mar_addr_in = addIS;
    addr_in = mar_addr_out;

    //addr_in = addIS;
    data_in = instr_code;

    if(sram_state == 5) // state w4 of sram
        begin
            top_state = FINISH;
        end
    else
        begin
            top_state = WRITEI2;
        end
    end
end

FINISH:begin
    begin
        sram_done = 1; // finished

        sram_start = 1; // turn off

        top_state = INITIALREG;
    end
end

//decode and put stuff in register
INITIALREG:begin
    reg_done = 0;
    sram_start = 0; // turn on sram
    reg_start = 0; // turn on register
    sram_rw = 1; // 1 = read from sram
    reg_we = 1; // 1 = write to register

    mar_en = 1;

```

writing

sram

```

        if(sram_state == 6) // state w5 of sram
            begin
                sram_done = 0; // dropping
the signal now is to give correct

                // timing to acquire correct operations\

                top_state = INITIALREG;
            end
        else if(sram_state == 11) // state R5 of sram
            begin
                //sram_done = 0;
                top_state = READSRAMA1;
            end
        else
            begin
                top_state = INITIALREG;
            end
        end
    // read A from sram
    READSRAMA1:begin
        mar_addr_in = addA_sram;
        //sramIn = sram_data_out;
        top_state = READSRAMA2;
    end

    READSRAMA2:begin
        addr_in = mar_addr_out;
        sramIn = sram_data_out;
        top_state = READSRAMA3;
    end

    READSRAMA3:begin
        dataA_in1 = regIn;
        top_state = READSRAMB1;
    end

    // read B from sram
    READSRAMB1:begin
        mar_addr_in = addB_sram;
        //sramIn = sram_data_out;
        top_state = READSRAMB2;
    end

    READSRAMB2:begin

```

```

                                addr_in = mar_addr_out;
                                sramIn = sram_data_out;
                                top_state = READSRAMB3;
                                end

                                READSRAMB3:begin
                                    dataB_in1 = regIn;
                                    top_state = READSRAMIS1;
                                end

                                // read Instructions from sram
                                READSRAMIS1:begin
                                    mar_addr_in = addIS;
                                    //sramIn =
sram_data_out;
                                    top_state =
                                READSRAMIS2;
                                end

                                READSRAMIS2:begin
                                    addr_in =
mar_addr_out;
                                    sramIn =
sram_data_out;
                                    top_state =
                                READSRAMIS3;
                                end

                                READSRAMIS3:begin
                                    instr_code1 = regIn;
                                    top_state =
SIGNEXTEND;
                                end

                                SIGNEXTEND: begin
                                    if(dataA_in1[15] == 0)
                                        begin
                                            dataA_in1[31:16] = 16'd0;
                                        end
                                    else if (dataA_in1[15] == 1)
                                        begin
                                            dataA_in1[31:16] =
16'b1111111111111111;/// high Z?

```



```

        w_data = result;

        top_state = CHANGEOP;
    end

    CHANGEOP:begin
        if(opcode == 8)
            begin
                top_state = IDLE;
            end
        else
            begin
                opcode = opcode + 1;
                addResult_reg =
                    addResult_reg + 1;
                top_state =
                    COMPUTE;
            end
        end

        default: begin
            top_state = IDLE;
        end
    endcase
end

endmodule

//MDR
module mdr(clk, rst, rw, regIn, regOut, sramIn, sramOut, out);

    input clk, rst, rw;
    input [31:0] regOut;
    input [31:0] sramIn;
    output [31:0] regIn;
    output [31:0] sramOut;
    output [31:0] out;
    wire [31:0] d;

    //bufif32([32]out, [32]register, enable)

```

```

bufif32 bufsramIn(d, sramIn , rw);

bufif32 bufregIn(regIn, out , rw&clk);

bufif32 bufsramOut(sramOut, out , ~rw&clk);

bufif32 bufregOut(d, regOut , ~rw);

reg32 myreg32(d, clk, rst, out, ~clk);

//bufif32 buf1(out, sramOut, ~rw);
//bufif32 buf2(out, regIn, rw);

// reading from SRAM to regfile rw = 1;
// writing from regfile to SRAM rw = 0;
endmodule


//MAR
module mar(addr_in, addr_out, clk, rst, en);
    input addr_in, clk, rst, en;
    output addr_out;

    reg32 addr_reg(addr_in, clk, rst, addr_out, en);

endmodule


//instruction
module instruction(opcode, addA, addB, instruction);
    input [2:0] opcode;
    input [4:0] addA, addB;
    output [15:0] instruction;           //instruction SRAM containing op code, &A, &B

    assign instruction [15:13] = opcode[2:0];    //3 bit for op code
    assign instruction [12:8] = addA;            //5 bit for address A
    assign instruction [7:3] = addB;             //5 bit for address B

endmodule


//decoder
module decoderIS(IS, opcode, addA, addB);

```

```

input [15:0] IS;
output [2:0] opcode;
output [4:0] addA;
output [4:0] addB;

assign opcode = IS[15:13];
assign addA = IS[12:8];
assign addB = IS[7:3];

endmodule

//SRAM Model
module sram_inter(clk, rst, sram_start, sram_rw, ce, sram_we, oe, ub, lb, sram_done,
                  data_bus, addr_in, addr_out, sram_state, data_in, data_out);

input clk; // input clk for timing
input rst; // active low reset
input sram_start; // active low sram_start signal
input sram_rw; // input to determine to read or to write; 1 = read, 0 = write
input sram_done; // signal to tell sram to stop reading/writing
input [15:0] data_in; // 16 bit input data
input [17:0] addr_in; // 18 bit input address

output reg ce; // active low chip select
output reg sram_we; // active low write control
output reg oe; // active low output enable
output reg ub; // active low upper byte select
output reg lb; // active low losram_wer byte select
output reg [17:0] addr_out; // 18-bit address bus
inout [15:0] data_bus; // 16-bit data bus to transfer data betwsram_ween interface and SRAM
output [15:0] data_out; // reg for data to be read out to LED's

wire [15:0] data_out = data_bus;

parameter IDLE = 0, READY = 1, W1 = 2, W2 = 3, W3 = 4, W4 = 5, W5 = 6;
parameter R1 = 7, R2 = 8, R3 = 9, R4 = 10, R5 = 11, R6 = 12, R7 = 13, R8 = 14;
output reg [4:0] sram_state;

always@(posedge clk or negedge rst)

begin
    if(rst == 0)
        begin

```

```

        ce = 1;
        oe = 1;
        sram_we = 1;
        ub = 1;
        lb = 1;
        addr_out = addr_in;
        sram_state = IDLE;

    end

    else

    begin
        case(sram_state)

            IDLE:begin

                ce = 1;
                oe = 1;
                sram_we = 1;
                ub = 1;
                lb = 1;
                if(sram_start == 0)
                begin
                    sram_state = READY;
                end
                else
                begin
                    sram_state = IDLE;
                end
            end

            READY:begin
                if(sram_rw == 0)
                begin
                    //addr_out = addr_in;
                    sram_state = W1; // go to write sram_state
                end
                else if(sram_rw == 1)
                begin
                    sram_state = R1; // go to read sram_state
                end
                else
                begin
                    sram_state = READY;
                end
            end
        endcase
    end
end

```



```

        end
    end
end
/*
W1:begin
    ce <= 0;
    sram_we <= 0;
    ub <= 0;
    lb <= 0;
    sram_state <= W3;
end

W3:begin
    if(sram_done == 1)
        begin
            sram_state <= W4;
        end
    else
        begin
            addr_out <= addr_in;
            sram_state <= W3;
        end
    end
end

W4:begin
    ce <= 1;
    sram_we <= 1;
    ub <= 1;
    lb <= 1;
    sram_state = IDLE;
end*/
W1:begin
    addr_out = addr_in;
    sram_state = W2;
end

W2:begin
    ce = 0;
    sram_we = 0;
    ub = 0;
    lb = 0;
    sram_state = W3;
end

W3:begin

```

```

        sram_state = W4;
    end

W4:begin
    ce = 1;
    sram_we = 1;
    ub = 1;
    lb = 1;
    //data_out <= 127-addr;
    sram_state = W5;
end

W5:begin
    addr_out = addr_in;

    if(sram_done == 1)
        begin
            sram_state = IDLE;
        end
    else
        begin
            sram_state = W1;
        end
    end
end

R1:begin
    addr_out = addr_in;
    sram_state = R2;
end

R2:begin
    ce = 0;
    sram_state = R3;
end

R3:begin
    lb = 0;
    ub = 0;
    sram_state = R4;
end

R4:begin
    oe = 0;
    sram_state = R5;
end

```

```

        end

        R5:begin
            if(sram_done == 1)
                begin
                    sram_state = R6;
                end
            else
                begin
                    addr_out = addr_in;
                    sram_state = R5;
                end
            end
        end
        R6: begin
            ce = 1;
            sram_state = R7;
        end
        R7: begin
            oe = 1;
            sram_state = R8;
        end
        R8: begin
            lb = 1;
            ub = 1;
            sram_state = IDLE;
        end
        default:begin
            sram_state = IDLE;
        end
    endcase

end

end

assign data_bus = oe ? data_in : 16'bZ;

endmodule

//register file interface
module register_inter(reg_we, clk, rst, reg_start, reg_done, r_sel1, r_sel2, w_sel,

```

```

        r_data1, r_data2, w_data, reg_state);

parameter IDLE = 0, READY = 1, WRITE = 2, READ = 3;

output reg [3:0] reg_state;
input [4:0]r_sel1, r_sel2, w_sel;
input [31:0]w_data;
input clk, rst, reg_start, reg_done;

input reg_we; // 0 = read, 1 = write;
output [31:0]r_data1, r_data2;

register2 myRegister(w_sel, reg_we, clk, rst, w_data, r_data1, r_data2, r_sel1, r_sel2);

always@(posedge clk or negedge rst) // idol state
begin
    if(rst == 0)
    begin
        reg_state <= IDLE;
    end
    else
    begin
        case(reg_state)
            IDLE:begin
                if(reg_start == 0)
                begin
                    reg_state <= READY;
                end
                else
                begin
                    reg_state <= IDLE;
                end
            end
            READY:begin
                if(reg_we == 0)
                begin
                    reg_state <= READ;
                end
                else if(reg_we == 1)
                begin
                    reg_state <= WRITE;
                end
            end
            else

```

```

        begin
            reg_state <= READY;
        end
    end

    WRITE:begin
        if(reg_done == 1) // done writing
        begin
            reg_state <= IDLE;
        end
        else
        begin
            reg_state <= WRITE;
        end
    end

    READ:begin
        if(reg_done == 1) // done reading
        begin
            reg_state <= IDLE;
        end
        else
        begin
            reg_state <= READ;
        end
    end

    default:begin
        reg_state <= IDLE;
    end

endcase

end

end

endmodule

```

```

module register2(w_sel, reg_we, clk, rst, w_data, r_data1, r_data2, r_sel1, r_sel2);

    input [4:0] w_sel, r_sel1, r_sel2;
    input reg_we, clk, rst;

```

```

input [31:0]w_data;
output[31:0] r_data1, r_data2;

wire [31:0] bo0, bo1, bo2, bo3, bo4, bo5, bo6, bo7, bo8, bo9, bo10, bo11,
bo12, bo13, bo14, bo15, bo16, bo17, bo18, bo19, bo20, bo21,
bo22, bo23, bo24, bo25, bo26, bo27, bo28, bo29, bo30, bo31;

wire [31:0] out0, out1, out2, out3,out4,out5,out6,out7,out8,out9,out10,out11,

out12,out13,out14,out15,out16,out17,out18,out19,out20,out21,out22,

out23,out24,out25,out26,out27,out28,out29,out30,out31;

wire [31:0] hot_code; // one-hot-code of the 5 bit register select
wire a0, a1, a2, a3, a4, a5, a6, a7, a8, a9, a10, a11, a12, a13, a14, a15, a16,
a17, a18, a19, a20, a21, a22, a23, a24, a25, a26, a27, a28, a29, a30,
a31;
// decoding a one hot code
decoder decode1(hot_code, w_sel);

and and0(a0, reg_we, hot_code[0]), and1(a1, reg_we, hot_code[1]), and2(a2, reg_we,
hot_code[2]);
and and3(a3, reg_we, hot_code[3]), and4(a4, reg_we, hot_code[4]), and5(a5, reg_we,
hot_code[5]);
and and6(a6, reg_we, hot_code[6]), and7(a7, reg_we, hot_code[7]), and8(a8, reg_we,
hot_code[8]);
and and9(a9, reg_we, hot_code[9]);

and and10(a10, reg_we, hot_code[10]), and11(a11, reg_we, hot_code[11]), and12(a12,
reg_we, hot_code[12]);
and and13(a13, reg_we, hot_code[13]), and14(a14, reg_we, hot_code[14]), and15(a15,
reg_we, hot_code[15]);
and and16(a16, reg_we, hot_code[16]), and17(a17, reg_we, hot_code[17]), and18(a18,
reg_we, hot_code[18]);
and and19(a19, reg_we, hot_code[19]);

and and20(a20, reg_we, hot_code[20]), and21(a21, reg_we, hot_code[21]), and22(a22,
reg_we, hot_code[22]);
and and23(a23, reg_we, hot_code[23]), and24(a24, reg_we, hot_code[24]), and25(a25,
reg_we, hot_code[25]);
and and26(a26, reg_we, hot_code[26]), and27(a27, reg_we, hot_code[27]), and28(a28,
reg_we, hot_code[28]);
and and29(a29, reg_we, hot_code[29]);

```

```
and and30(a30, reg_we, hot_code[30]), and31(a31, reg_we, hot_code[31]);
```

```
//read
```

```
mux mux1(r_data1, r_sel1, out0, out1, out2, out3,out4,out5,out6,out7,out8,out9,out10,
```

```
out11,out12,out13,out14,out15,out16,out17,out18,out19,out20,out21,out22,  
out23,out24,out25,out26,out27,out28,out29,out30,out31);
```

```
mux mux2(r_data2, r_sel2, out0, out1, out2, out3,out4,out5,out6,out7,out8,out9,out10,
```

```
out11,out12,out13,out14,out15,out16,out17,out18,out19,out20,out21,out22,  
out23,out24,out25,out26,out27,out28,out29,out30,out31);
```

```
// use and_gate outputs as enables for bufifs that take input
```

```
// of the w_data and output to the respective register
```

```
reg32 r0(bo0, clk, rst, out0, a0);
```

```
reg32 r1(bo1, clk, rst, out1, a1);
```

```
reg32 r2(bo2, clk, rst, out2, a2);
```

```
reg32 r3(bo3, clk, rst, out3, a3);
```

```
reg32 r4(bo4, clk, rst, out4, a4);
```

```
reg32 r5(bo5, clk, rst, out5, a5);
```

```
reg32 r6(bo6, clk, rst, out6, a6);
```

```
reg32 r7(bo7, clk, rst, out7, a7);
```

```
reg32 r8(bo8, clk, rst, out8, a8);
```

```
reg32 r9(bo9, clk, rst, out9, a9);
```

```
reg32 r10(bo10, clk, rst, out10, a10);
```

```
reg32 r11(bo11, clk, rst, out11, a11);
```

```
reg32 r12(bo12, clk, rst, out12, a12);
```

```
reg32 r13(bo13, clk, rst, out13, a13);
```

```
reg32 r14(bo14, clk, rst, out14, a14);
```

```
reg32 r15(bo15, clk, rst, out15, a15);
```

```
reg32 r16(bo16, clk, rst, out16, a16);
```

```
reg32 r17(bo17, clk, rst, out17, a17);
```

```
reg32 r18(bo18, clk, rst, out18, a18);
```

```
reg32 r19(bo19, clk, rst, out19, a19);
```

```
reg32 r20(bo20, clk, rst, out20, a20);
```

```
reg32 r21(bo21, clk, rst, out21, a21);
```

```
reg32 r22(bo22, clk, rst, out22, a22);
```

```
reg32 r23(bo23, clk, rst, out23, a23);
```

```

reg32 r24(bo24, clk, rst, out24, a24);
reg32 r25(bo25, clk, rst, out25, a25);
reg32 r26(bo26, clk, rst, out26, a26);
reg32 r27(bo27, clk, rst, out27, a27);
reg32 r28(bo28, clk, rst, out28, a28);
reg32 r29(bo29, clk, rst, out29, a29);
reg32 r30(bo30, clk, rst, out30, a30);
reg32 r31(bo31, clk, rst, out31, a31);

```

```

bufif32 bufif00(bo0, w_data, a0);
bufif32 bufif01(bo1, w_data, a1);
bufif32 bufif02(bo2, w_data, a2);
bufif32 bufif03(bo3, w_data, a3);
bufif32 bufif04(bo4, w_data, a4);
bufif32 bufif05(bo5, w_data, a5);
bufif32 bufif06(bo6, w_data, a6);
bufif32 bufif07(bo7, w_data, a7);
bufif32 bufif08(bo8, w_data, a8);
bufif32 bufif09(bo9, w_data, a9);
bufif32 bufif10(bo10, w_data, a10);
bufif32 bufif11(bo11, w_data, a11);
bufif32 bufif12(bo12, w_data, a12);
bufif32 bufif13(bo13, w_data, a13);
bufif32 bufif14(bo14, w_data, a14);
bufif32 bufif15(bo15, w_data, a15);
bufif32 bufif16(bo16, w_data, a16);
bufif32 bufif17(bo17, w_data, a17);
bufif32 bufif18(bo18, w_data, a18);
bufif32 bufif19(bo19, w_data, a19);
bufif32 bufif20(bo20, w_data, a20);
bufif32 bufif21(bo21, w_data, a21);
bufif32 bufif22(bo22, w_data, a22);
bufif32 bufif23(bo23, w_data, a23);
bufif32 bufif24(bo24, w_data, a24);
bufif32 bufif25(bo25, w_data, a25);
bufif32 bufif26(bo26, w_data, a26);
bufif32 bufif27(bo27, w_data, a27);
bufif32 bufif28(bo28, w_data, a28);
bufif32 bufif29(bo29, w_data, a29);
bufif32 bufif30(bo30, w_data, a30);
bufif32 bufif31(bo31, w_data, a31);

```


endmodule

```
module mux(out, sel, reg0, reg1, reg2, reg3, reg4, reg5, reg6,  
           reg7, reg8, reg9, reg10, reg11, reg12, reg13, reg14, reg15,  
           reg16, reg17, reg18, reg19, reg20, reg21, reg22, reg23, reg24,  
           reg25, reg26, reg27, reg28, reg29, reg30, reg31);
```

```
    output [31:0] out;
```

```
    input [4:0] sel;
```

```
    input [31:0] reg0, reg1, reg2, reg3, reg4, reg5, reg6,
```

```
           reg7, reg8, reg9, reg10, reg11, reg12, reg13, reg14, reg15,
```

```
           reg16, reg17, reg18, reg19, reg20, reg21, reg22, reg23, reg24,
```

```
           reg25, reg26, reg27, reg28, reg29, reg30, reg31;
```

```
    wire [31:0] hot_code;
```

```
    decoder decoder01(hot_code, sel);
```

```
    bufif32 bufif00(out, reg0, hot_code[0]);
```

```
    bufif32 bufif01(out, reg1, hot_code[1]);
```

```
    bufif32 bufif02(out, reg2, hot_code[2]);
```

```
    bufif32 bufif03(out, reg3, hot_code[3]);
```

```
    bufif32 bufif04(out, reg4, hot_code[4]);
```

```
    bufif32 bufif05(out, reg5, hot_code[5]);
```

```
    bufif32 bufif06(out, reg6, hot_code[6]);
```

```
    bufif32 bufif07(out, reg7, hot_code[7]);
```

```
    bufif32 bufif08(out, reg8, hot_code[8]);
```

```
    bufif32 bufif09(out, reg9, hot_code[9]);
```

```
    bufif32 bufif10(out, reg10, hot_code[10]);
```

```
    bufif32 bufif11(out, reg11, hot_code[11]);
```

```
    bufif32 bufif12(out, reg12, hot_code[12]);
```

```
    bufif32 bufif13(out, reg13, hot_code[13]);
```

```
    bufif32 bufif14(out, reg14, hot_code[14]);
```

```
    bufif32 bufif15(out, reg15, hot_code[15]);
```

```
    bufif32 bufif16(out, reg16, hot_code[16]);
```

```
    bufif32 bufif17(out, reg17, hot_code[17]);
```

```
    bufif32 bufif18(out, reg18, hot_code[18]);
```

```
    bufif32 bufif19(out, reg19, hot_code[19]);
```

```
    bufif32 bufif20(out, reg20, hot_code[20]);
```

```
    bufif32 bufif21(out, reg21, hot_code[21]);
```

```
    bufif32 bufif22(out, reg22, hot_code[22]);
```

```
    bufif32 bufif23(out, reg23, hot_code[23]);
```

```
    bufif32 bufif24(out, reg24, hot_code[24]);
```

```

    bufif32 bufif25(out, reg25,hot_code[25]);
    bufif32 bufif26(out, reg26,hot_code[26]);
    bufif32 bufif27(out, reg27,hot_code[27]);
    bufif32 bufif28(out, reg28,hot_code[28]);
    bufif32 bufif29(out, reg29,hot_code[29]);
    bufif32 bufif30(out, reg30,hot_code[30]);
    bufif32 bufif31(out, reg31,hot_code[31]);

```

```

endmodule

```

```

module arithmetic(result, zero, carryOut, overflow, negative, A, B, control, clk, rst, run);

```

```

    output zero;
    output carryOut;
    output overflow;
    output negative;
    output [31:0]result;
    wire [31:0]tempResult, diff,andResult,orResult,xorResult,sltResult,sllResult;

```

```

    input run;
    input clk;
    input rst;
    input[31:0] A;
    input[31:0] B;
    input[2:0] control;
    //input[1:0] controlB;

```

```

    parameter NOP = 3'b000;
    parameter ADD = 3'b001;
    parameter SUB = 3'b010;
    parameter AND = 3'b011;
    parameter OR = 3'b100;
    parameter XOR = 3'b101;
    parameter SLT = 3'b110;
    parameter SLL = 3'b111;

```

```

    wire zero0,zero1,zero2,zero3,zero4,zero5;
    wire carryOut0,carryOut1,carryOut2,carryOut3,carryOut4,carryOut5;
    wire negative0,negative1,negative2,negative3,negative4,negative5;
    wire overflow0,overflow1,overflow2,overflow3,overflow4,overflow5;

```

```

    wire[31:0] A, B;

```

```

bufif32 buff0(result, 32'b0, enable0);
bufif32 buff1(result, tempResult, enable1);
bufif32 buff2(result, diff, enable2);
bufif32 buff3(result, andResult, enable3);
bufif32 buff4(result, orResult, enable4);
bufif32 buff5(result, xorResult, enable5);
bufif32 buff6(result, sltResult, enable6);
bufif32 buff7(result, sllResult, enable7);

```

```

assign enable0 = ~control[2] & ~control[1] & ~control[0];
assign enable1 = ~control[2] & ~control[1] & control[0];
assign enable2 = ~control[2] & control[1] & ~control[0];
assign enable3 = ~control[2] & control[1] & control[0];
assign enable4 = control[2] & ~control[1] & ~control[0];
assign enable5 = control[2] & ~control[1] & control[0];
assign enable6 = control[2] & control[1] & ~control[0];
assign enable7 = control[2] & control[1] & control[0];

```

```

assign negative = (negative0 & enable1) | (negative1 & enable2) | (negative2 & enable3) |
    (negative3 & enable4) | (negative4 & enable5) | (negative5 & enable6) ;
assign overflow = (overflow0 & enable1) | (overflow1 & enable2) | (overflow2 & enable3) |
    (overflow3 & enable4) | (overflow4 & enable5) | (overflow5 & enable6) ;
assign carryOut = (carryOut0 & enable1) | (carryOut1 & enable2) | (carryOut2 & enable3) |
    (carryOut3 & enable4) | (carryOut4 & enable5) | (carryOut5 & enable6) ;
assign zero = (zero0 & enable1) | (zero1 & enable2) | (zero2 & enable3) | (zero3 &
    enable4) | (zero4 & enable5) | (zero5 & enable6) ;

```

```

//assign negative = negative0 & enable1 ;

```

```

adder adder1(A, B, zero0, carryOut0, overflow0, negative0, tempResult);

```

```

subtract sub1(A, B, zero1, carryOut1, overflow1, negative1, diff);

```

```

logicAnd logic1(A, B, zero2, carryOut2, overflow2, negative2, andResult);

```

```

logicOr logic2(A, B, zero3, carryOut3, overflow3, negative3, orResult);

```

```

logicXor logic3(A, B, zero4, carryOut4, overflow4, negative4, xorResult);

```

```

barrel_Shifter shifter(sllResult, A, B);

```

```
SLT slter(A, B, zero5, carryOut5, overflow5, negative5, sltResult);
```

```
Endmodule
```

```
module adder (A, B, zero, carryOut, overFlow, negative, sum);  
    input [31:0] A, B;  
    output [31:0] sum;  
    output zero, carryOut, overFlow, negative;  
    wire [31:0] Co;
```

```
    assign Co[0] = 0 & (A[0]^B[0]) | (A[0]&B[0]);  
    assign Co[1] = Co[0] & (A[1]^B[1]) | (A[1]&B[1]);  
    assign Co[2] = Co[1] & (A[2]^B[2]) | (A[2]&B[2]);  
    assign Co[3] = Co[2] & (A[3]^B[3]) | (A[3]&B[3]);  
    assign Co[4] = Co[3] & (A[4]^B[4]) | (A[4]&B[4]);  
    assign Co[5] = Co[4] & (A[5]^B[5]) | (A[5]&B[5]);  
    assign Co[6] = Co[5] & (A[6]^B[6]) | (A[6]&B[6]);  
    assign Co[7] = Co[6] & (A[7]^B[7]) | (A[7]&B[7]);  
    assign Co[8] = Co[7] & (A[8]^B[8]) | (A[8]&B[8]);  
    assign Co[9] = Co[8] & (A[9]^B[9]) | (A[9]&B[9]);  
    assign Co[10] = Co[9] & (A[10]^B[10]) | (A[10]&B[10]);  
    assign Co[11] = Co[10] & (A[11]^B[11]) | (A[11]&B[11]);  
    assign Co[12] = Co[11] & (A[12]^B[12]) | (A[12]&B[12]);  
    assign Co[13] = Co[12] & (A[13]^B[13]) | (A[13]&B[13]);  
    assign Co[14] = Co[13] & (A[14]^B[14]) | (A[14]&B[14]);  
    assign Co[15] = Co[14] & (A[15]^B[15]) | (A[15]&B[15]);  
    assign Co[16] = Co[15] & (A[16]^B[16]) | (A[16]&B[16]);  
    assign Co[17] = Co[16] & (A[17]^B[17]) | (A[17]&B[17]);  
    assign Co[18] = Co[17] & (A[18]^B[18]) | (A[18]&B[18]);  
    assign Co[19] = Co[18] & (A[19]^B[19]) | (A[19]&B[19]);  
    assign Co[20] = Co[19] & (A[20]^B[20]) | (A[20]&B[20]);  
    assign Co[21] = Co[20] & (A[21]^B[21]) | (A[21]&B[21]);  
    assign Co[22] = Co[21] & (A[22]^B[22]) | (A[22]&B[22]);  
    assign Co[23] = Co[22] & (A[23]^B[23]) | (A[23]&B[23]);  
    assign Co[24] = Co[23] & (A[24]^B[24]) | (A[24]&B[24]);  
    assign Co[25] = Co[24] & (A[25]^B[25]) | (A[25]&B[25]);  
    assign Co[26] = Co[25] & (A[26]^B[26]) | (A[26]&B[26]);  
    assign Co[27] = Co[26] & (A[27]^B[27]) | (A[27]&B[27]);  
    assign Co[28] = Co[27] & (A[28]^B[28]) | (A[28]&B[28]);  
    assign Co[29] = Co[28] & (A[29]^B[29]) | (A[29]&B[29]);  
    assign Co[30] = Co[29] & (A[30]^B[30]) | (A[30]&B[30]);  
    assign Co[31] = Co[30] & (A[31]^B[31]) | (A[31]&B[31]);
```

```

    assign sum[0] = A[0]^B[0]^0;
    assign sum[1] = A[1]^B[1]^Co[0];
    assign sum[2] = A[2]^B[2]^Co[1];
    assign sum[3] = A[3]^B[3]^Co[2];
    assign sum[4] = A[4]^B[4]^Co[3];
    assign sum[5] = A[5]^B[5]^Co[4];
    assign sum[6] = A[6]^B[6]^Co[5];
    assign sum[7] = A[7]^B[7]^Co[6];
    assign sum[8] = A[8]^B[8]^Co[7];
    assign sum[9] = A[9]^B[9]^Co[8];
    assign sum[10] = A[10]^B[10]^Co[9];
    assign sum[11] = A[11]^B[11]^Co[10];
    assign sum[12] = A[12]^B[12]^Co[11];
    assign sum[13] = A[13]^B[13]^Co[12];
    assign sum[14] = A[14]^B[14]^Co[13];
    assign sum[15] = A[15]^B[15]^Co[14];
    assign sum[16] = A[16]^B[16]^Co[15];
    assign sum[17] = A[17]^B[17]^Co[16];
    assign sum[18] = A[18]^B[18]^Co[17];
    assign sum[19] = A[19]^B[19]^Co[18];
    assign sum[20] = A[20]^B[20]^Co[19];
    assign sum[21] = A[21]^B[21]^Co[20];
    assign sum[22] = A[22]^B[22]^Co[21];
    assign sum[23] = A[23]^B[23]^Co[22];
    assign sum[24] = A[24]^B[24]^Co[23];
    assign sum[25] = A[25]^B[25]^Co[24];
    assign sum[26] = A[26]^B[26]^Co[25];
    assign sum[27] = A[27]^B[27]^Co[26];
    assign sum[28] = A[28]^B[28]^Co[27];
    assign sum[29] = A[29]^B[29]^Co[28];
    assign sum[30] = A[30]^B[30]^Co[29];
    assign sum[31] = A[31]^B[31]^Co[30];

    assign carryOut = Co[31];

    assign negative = sum[31];
    assign zero = (sum == 32'd0)? 1: 0;
    assign overFlow = Co[31]^Co[30];
endmodule

module mux_structural(
    mux_out    ,
    sel        ,
    A          ,

```

B

);

input A, B, sel ;

output mux_out;

not not1(nsel, sel);

and and1(p1, B, sel);

and and2(p2, A , nsel);

or or1(mux_out, p2, p1);

endmodule

// 0,1,2,3 bits left barrel shifter

//

module barrel_Shifter(outshift, inshift, control);

output[31:0] outshift;

input [31:0] inshift;

input [1:0] control;

wire [31:0] outMux;

mux_structural mux0_0 (outMux[0] , control[1] , inshift[0], 0);

mux_structural mux0_1 (outMux[1] , control[1] , inshift[1], 0);

mux_structural mux0_2 (outMux[2] , control[1] , inshift[2], inshift[0]);

mux_structural mux0_3 (outMux[3] , control[1] , inshift[3], inshift[1]);

mux_structural mux0_4 (outMux[4] , control[1] , inshift[4], inshift[2]);

mux_structural mux0_5 (outMux[5] , control[1] , inshift[5], inshift[3]);

mux_structural mux0_6 (outMux[6] , control[1] , inshift[6], inshift[4]);

mux_structural mux0_7 (outMux[7] , control[1] , inshift[7], inshift[5]);

mux_structural mux0_8 (outMux[8] , control[1] , inshift[8], inshift[6]);

mux_structural mux0_9 (outMux[9] , control[1] , inshift[9], inshift[7]);

mux_structural mux0_10(outMux[10], control[1] , inshift[10], inshift[8]);

mux_structural mux0_11(outMux[11], control[1] , inshift[11], inshift[9]);

mux_structural mux0_12(outMux[12], control[1] , inshift[12], inshift[10]);

mux_structural mux0_13(outMux[13], control[1] , inshift[13], inshift[11]);

mux_structural mux0_14(outMux[14], control[1] , inshift[14], inshift[12]);

mux_structural mux0_15(outMux[15], control[1] , inshift[15], inshift[13]);

```

mux_structural mux0_16(outMux[16], control[1], inshift[16], inshift[14]);
mux_structural mux0_17(outMux[17], control[1], inshift[17], inshift[15]);
mux_structural mux0_18(outMux[18], control[1], inshift[18], inshift[16]);
mux_structural mux0_19(outMux[19], control[1], inshift[19], inshift[17]);
mux_structural mux0_20(outMux[20], control[1], inshift[20], inshift[18]);
mux_structural mux0_21(outMux[21], control[1], inshift[21], inshift[19]);
mux_structural mux0_22(outMux[22], control[1], inshift[22], inshift[20]);
mux_structural mux0_23(outMux[23], control[1], inshift[23], inshift[21]);
mux_structural mux0_24(outMux[24], control[1], inshift[24], inshift[22]);
mux_structural mux0_25(outMux[25], control[1], inshift[25], inshift[23]);
mux_structural mux0_26(outMux[26], control[1], inshift[26], inshift[24]);
mux_structural mux0_27(outMux[27], control[1], inshift[27], inshift[25]);
mux_structural mux0_28(outMux[28], control[1], inshift[28], inshift[26]);
mux_structural mux0_29(outMux[29], control[1], inshift[29], inshift[27]);
mux_structural mux0_30(outMux[30], control[1], inshift[30], inshift[28]);
mux_structural mux0_31(outMux[31], control[1], inshift[31], inshift[29]);

```

```

mux_structural mux1_0(outshift[0], control[0], outMux[0], 0);
mux_structural mux1_1(outshift[1], control[0], outMux[1], outMux[0]);
mux_structural mux1_2(outshift[2], control[0], outMux[2], outMux[1]);
mux_structural mux1_3(outshift[3], control[0], outMux[3], outMux[2]);
mux_structural mux1_4(outshift[4], control[0], outMux[4], outMux[3]);
mux_structural mux1_5(outshift[5], control[0], outMux[5], outMux[4]);
mux_structural mux1_6(outshift[6], control[0], outMux[6], outMux[5]);
mux_structural mux1_7(outshift[7], control[0], outMux[7], outMux[6]);
mux_structural mux1_8(outshift[8], control[0], outMux[8], outMux[7]);
mux_structural mux1_9(outshift[9], control[0], outMux[9], outMux[8]);
mux_structural mux1_10(outshift[10], control[0], outMux[10], outMux[9]);
mux_structural mux1_11(outshift[11], control[0], outMux[11], outMux[10]);
mux_structural mux1_12(outshift[12], control[0], outMux[12], outMux[11]);
mux_structural mux1_13(outshift[13], control[0], outMux[13], outMux[12]);
mux_structural mux1_14(outshift[14], control[0], outMux[14], outMux[13]);
mux_structural mux1_15(outshift[15], control[0], outMux[15], outMux[14]);
mux_structural mux1_16(outshift[16], control[0], outMux[16], outMux[15]);
mux_structural mux1_17(outshift[17], control[0], outMux[17], outMux[16]);
mux_structural mux1_18(outshift[18], control[0], outMux[18], outMux[17]);
mux_structural mux1_19(outshift[19], control[0], outMux[19], outMux[18]);
mux_structural mux1_20(outshift[20], control[0], outMux[20], outMux[19]);
mux_structural mux1_21(outshift[21], control[0], outMux[21], outMux[20]);
mux_structural mux1_22(outshift[22], control[0], outMux[22], outMux[21]);
mux_structural mux1_23(outshift[23], control[0], outMux[23], outMux[22]);
mux_structural mux1_24(outshift[24], control[0], outMux[24], outMux[23]);
mux_structural mux1_25(outshift[25], control[0], outMux[25], outMux[24]);
mux_structural mux1_26(outshift[26], control[0], outMux[26], outMux[25]);

```

```

mux_structural mux1_27(outshift[27], control[0] , outMux[27], outMux[26] );
mux_structural mux1_28(outshift[28], control[0] , outMux[28], outMux[27] );
mux_structural mux1_29(outshift[29], control[0] , outMux[29], outMux[28] );
mux_structural mux1_30(outshift[30], control[0] , outMux[30], outMux[29] );
mux_structural mux1_31(outshift[31], control[0] , outMux[31], outMux[30] );

```

```

endmodule

```

```

module logicAnd (A, B, zero, carryOut, overFlow, negative, anded);

```

```

    input [31:0] A, B;

```

```

    output [31:0] anded;

```

```

    output zero, carryOut, overFlow, negative;

```

```

    assign anded[0] = A[0] & B[0];

```

```

    assign anded[1] = A[1] & B[1];

```

```

    assign anded[2] = A[2] & B[2];

```

```

    assign anded[3] = A[3] & B[3];

```

```

    assign anded[4] = A[4] & B[4];

```

```

    assign anded[5] = A[5] & B[5];

```

```

    assign anded[6] = A[6] & B[6];

```

```

    assign anded[7] = A[7] & B[7];

```

```

    assign anded[8] = A[8] & B[8];

```

```

    assign anded[9] = A[9] & B[9];

```

```

    assign anded[10] = A[10] & B[10];

```

```

    assign anded[11] = A[11] & B[11];

```

```

    assign anded[12] = A[12] & B[12];

```

```

    assign anded[13] = A[13] & B[13];

```

```

    assign anded[14] = A[14] & B[14];

```

```

    assign anded[15] = A[15] & B[15];

```

```

    assign anded[16] = A[16] & B[16];

```

```

    assign anded[17] = A[17] & B[17];

```

```

    assign anded[18] = A[18] & B[18];

```

```

    assign anded[19] = A[19] & B[19];

```

```

    assign anded[20] = A[20] & B[20];

```

```

    assign anded[21] = A[21] & B[21];

```

```

    assign anded[22] = A[22] & B[22];

```

```

    assign anded[23] = A[23] & B[23];

```

```

    assign anded[24] = A[24] & B[24];

```



```

assign anded[25] = A[25] & B[25];
assign anded[26] = A[26] & B[26];
assign anded[27] = A[27] & B[27];
assign anded[28] = A[28] & B[28];
assign anded[29] = A[29] & B[29];
assign anded[30] = A[30] & B[30];
assign anded[31] = A[31] & B[31];

```

```

assign carryOut = 0;
assign overFlow = 0;
assign negative = anded[31];
assign zero = (anded == 0)? 1:0;

```

endmodule

```

module logicOr (A, B, zero, carryOut, overFlow, negative, ored);

```

```

    input [31:0] A, B;
    output [31:0] ored;
    output zero, carryOut, overFlow, negative;

```

```

    assign ored[0] = A[0] | B[0];
    assign ored[1] = A[1] | B[1];
    assign ored[2] = A[2] | B[2];
    assign ored[3] = A[3] | B[3];
    assign ored[4] = A[4] | B[4];
    assign ored[5] = A[5] | B[5];
    assign ored[6] = A[6] | B[6];
    assign ored[7] = A[7] | B[7];
    assign ored[8] = A[8] | B[8];
    assign ored[9] = A[9] | B[9];
    assign ored[10] = A[10] | B[10];
    assign ored[11] = A[11] | B[11];
    assign ored[12] = A[12] | B[12];
    assign ored[13] = A[13] | B[13];
    assign ored[14] = A[14] | B[14];
    assign ored[15] = A[15] | B[15];
    assign ored[16] = A[16] | B[16];
    assign ored[17] = A[17] | B[17];
    assign ored[18] = A[18] | B[18];
    assign ored[19] = A[19] | B[19];
    assign ored[20] = A[20] | B[20];
    assign ored[21] = A[21] | B[21];
    assign ored[22] = A[22] | B[22];
    assign ored[23] = A[23] | B[23];

```

```

    assign ored[24] = A[24] | B[24];
    assign ored[25] = A[25] | B[25];
    assign ored[26] = A[26] | B[26];
    assign ored[27] = A[27] | B[27];
    assign ored[28] = A[28] | B[28];
    assign ored[29] = A[29] | B[29];
    assign ored[30] = A[30] | B[30];
    assign ored[31] = A[31] | B[31];

    assign carryOut = 0;
    assign overFlow = 0;
    assign negative = ored[31];
    assign zero = (ored == 0)? 1:0;

endmodule

```

C Code: Pointers

```

int main(void)
{
    //declare type integer variables
    int A = 25;
    int B = 16;
    int C = 7;
    int D = 4;
    int E = 9;

    //declare type integer result
    int result;

    //declare and define five variables of type pointer to integer
    int* aPtr = &A;
    int* bPtr = &B;
    int* cPtr = &C;
    int* dPtr = &D;
    int* ePtr = &E;

    int a = *aPtr;
    int b = *bPtr;
    int c = *cPtr;
    int d = *dPtr;
    int e = *ePtr;
}

```

```

result = ((a-b)*(c+d))/e;

//result computation through its pointer
result = (((*aPtr)-(*bPtr))*((*cPtr)+(*dPtr)))/(*ePtr);

printf("a = %d\n", a);
printf("b = %d\n", b);
printf("c = %d\n", c);
printf("d = %d\n", d);
printf("e = %d\n", e);
printf("result = %d\n", result);

//display the memory address

printf("aPtr = %d\n", aPtr);
printf("bPtr = %d\n", bPtr);
printf("cPtr = %d\n", cPtr);
printf("dPtr = %d\n", dPtr);
printf("ePtr = %d\n", ePtr);

system("pause");

return 0;

}

```