Assignment 7

Kandula Revanth (21CS10035)

Pola Gnana Shekar (21CS10052)

ALU Design and Simulation:

This Report describes the Design of the ALU in structured Verilog covering the functions:
ADD, SUB, AND, OR, XOR, NOT, SLA, SRA, SRL, ADDI, SUBI, ANDI, ORI, XORI, NOTI, SLAI, SLAI, SRAI, SRLI.

The report also discusses how we can manage other functions like:
LD, ST, LDSP, STSP, BR, BMI, BPL, BZ, PUSH, POP, CALL, RET, MOVE, HALT, NOP.

This report also includes description of test cases used for demonstration of the module which are given in the test bench and how these are modified to work on the FPGA board.

Working of ALU:
The ALU module has:

- Input A: a part of operand_A which is of size 8 bits.
- Input B: a part of operand_B which is of size 8 bits.
- Input clk: clock
- Input ld: control signal for loading each byte of operands.
- Input opLd: control signal for loading op code to know the operation.
- Input exe: control signal which order to execute the operation on operands.
- Input out: control signal which controls the display of output between the choices - higher 16 and lower 16 bits.

- Output res: stores the output (output display on FPGA board).
- Output carryflag: stores if there is any carry after the operation.

The inputs *operand_A* and *operand_B*, each of 32 bits, are given to this module in parts, each byte at a time using A and B as inputs.

A variable *count* is used to assist taking input, when count is 0 take in last byte of operand and then increment the count and take in the next byte and repeat it until you take in all 4 bytes. Count is incremented using the equation:

Count = (Count+1)%4;

Control signal *ld* is set to 1 when the inputs A and B is to be taken and then set to 0 when need to give next input and then again switched back to 1 when ready to take in input and then this process repeats to take in entire operand value.

After this, we take in Op code, for this set the input value on board and then set the control signal *opLd* to 1 and it takes in the op code and then set the signal back to 0.

Now it is time to execute the operation on operands, for this set *exe* to 1 and after some clocks set we can set it back to 0. This completes the execution and stores value in *result* register.

This *result* can be accessed using *res* which provide the lower half of the *result* when control signal *out* is set to 0 and upper half when it is set to 1.

Now Let’s have a look on the function which are being executed

- Function description:

Each function is given a unique 6 bit code and this code are encoded using *parameters*.

ADD: Adds *operand_A* and *operand_B.*

SUB: Subtracts *operand_A* from *operand_B.*

AND: returns bitwise AND of *operand_A* and *operand_B.*

OR: returns bitwise OR of *operand_A* and *operand_B.*

XOR: return bitwise XOR of *operand_A* and *operand_B.*

NOT: return bitwise Negation of *operand_A.*

SLA: Arithematically shifts the *operand_A* to left by 1 or 0 depending the value *operand_B[0].*

SRA: Arithematically shifts the *operand_A* to right by 1 or 0 depending the value *operand_B[0].*

SRL: Logically shifts the *operand_A* to left by 1 or 0 depending the value *operand_B[0].*

For the Immediate value counter parts of these functions, we will have the immediate value in the *operand_B.*

Relation with other functions:

There are some other functions such as LD, ST, LDSP, STSP, BR, BMI, BPL, BZ, PUSH, POP, CALL, RET, MOVE, HALT, NOP.

- Among these some functions has do not use ALU for it's functionality: MOVE, HALT, NOP.
- Other function use ALU to manipulate NPC (next PC) - either to increment or adds some offset:
  BR, BMI, BPL, PUSH, POP, CALL, RET.
- Remaining use ALU to add some offset to memory addresses:
  LD, ST, LDSP, STSP.

Description of test cases in test bench:
Each test case has the following template:

A= [7:0] bits of *operand_A*;

B= [7:0] bits of *operand_B;*

ld = set control signal to 0; (initially)

opLd = set control signal to 0; (initially)

exe = set control signal to 0; (initially) out = set control signal to 0; (initially)

# wait for some time

ld = set it to 1; (takes in A and B).

# wait

ld = set it to 0;

A= [15:8] bits of *operand_A;*

B = [15:8] bits of *operand_B;*

# wait for some time

ld = set it to 1; (takes in A and B).

# wait

ld = set it to 0;

A= [23:16] bits of *operand_A;*

B = [23:16] bits of *operand_B;*

# wait for some time

ld = set it to 1; (takes in A and B).

# wait

ld = set it to 0;

A= [23:16] bits of *operand_A;*

B = [23:16] bits of *operand_B;*

# wait for some time

ld = set it to 1; (takes in A and B).

# wait

ld = set it to 0;

# wait

Take in Op code value using pins assigned to B[5:0] = op code;

#wait

opLd = set to 1; (takes in Op code)

#wait

opLd =   set to 0;

#wait

exe = set to 1: (executes the operation on operands)

#wait

exe = set to 0;

set the *out* value accordingly and #wait for some time and check for output value using *res.*

set *out* to 0 for lower half.

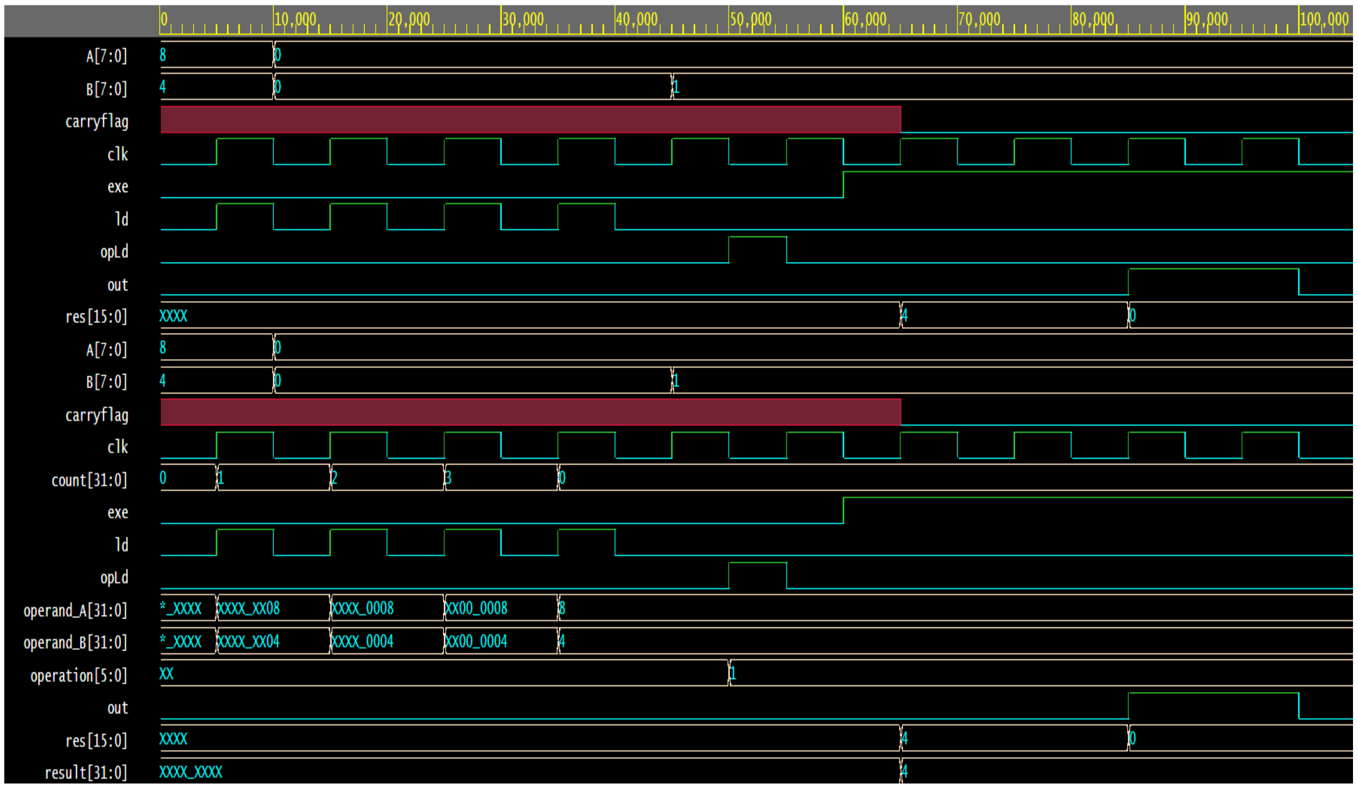
set *out* to 1 for upper half.

FPGA board controls:

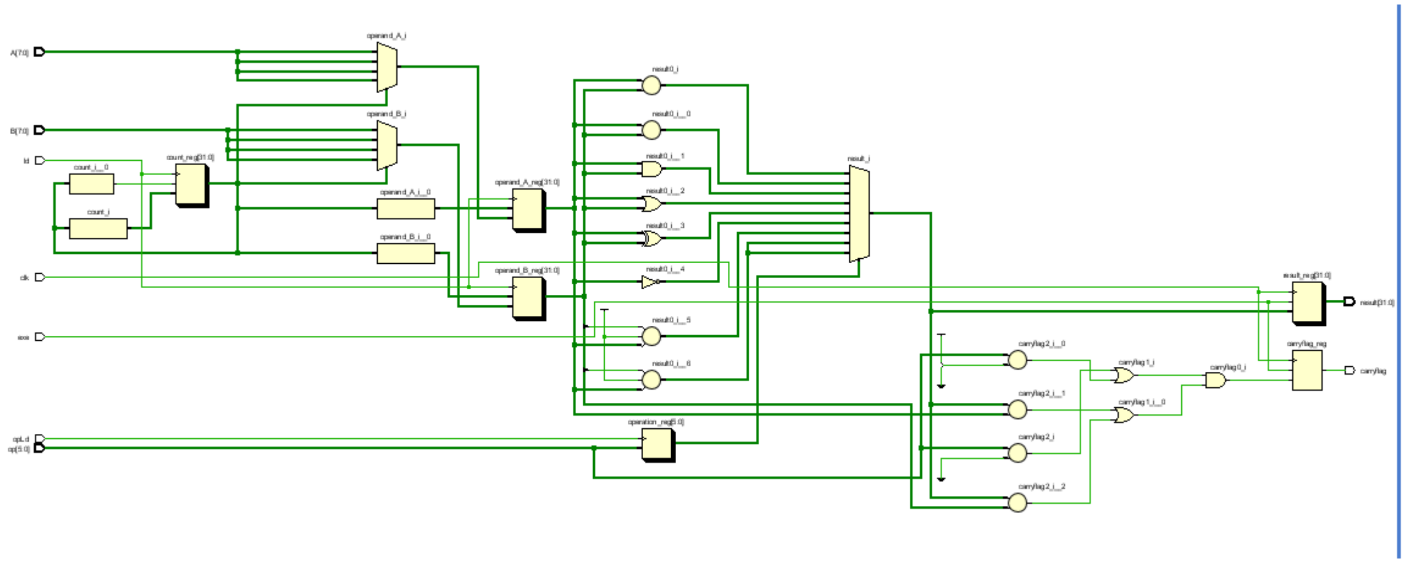
- Take in values of A and B using input pins and ld button
- Input op code using input pins and opLd button
- Execute it using exe button.
- Check output in output LEDs and use out button to switch between upper and lower half of result.

Simulation:

For Finding the subtraction of 8 and 4 :



Schematic Diagram:


A[7:0]
B[7:0]
ld
clk
exe
opLd
op[5:0]
operand_A_i
operand_B_i
count_i__0
count_i
count_reg[31:0]
operand_A_i__0
operand_B_i__0
operand_A_reg[31:0]
operand_B_reg[31:0]
result0_i
result0_i__0
result0_i__1
result0_i__2
result0_i__3
result0_i__4
result0_i__5
result0_i__6
result_i
operation_reg[5:0]
result_reg[31:0]
result[31:0]
carryflag2_i__0
carryflag2_i__1
carryflag2_i
carryflag2_i__2
carryflag1_i
carryflag1_i__0
carryflag0_i
carryflag_reg
carryflag