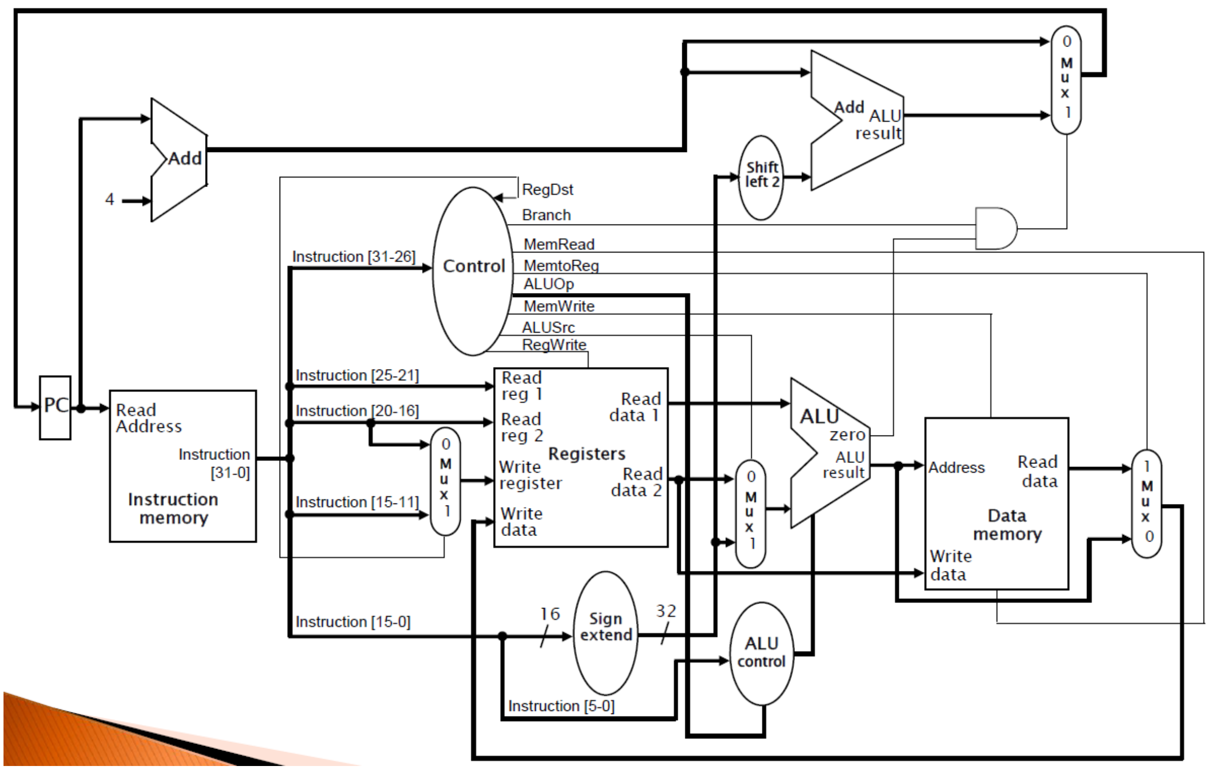
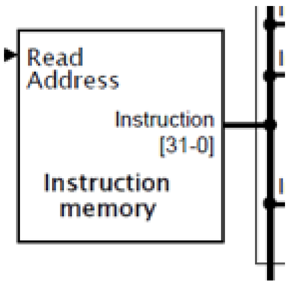
1. INTRODUCTION



This is a Single Cycle Datapath that i used as reference to start my project and i build up on this as i added more instructions. Although this datapath doesn't support all of the instructions as it is, all i had to do was add small units to make them work. For example added shift left 2 unit to shift 26 bit address from jump instruction and then used first 4 bits of PC + 4 to generate the 32 address of jump instruction and put it to a mux with the output of branch/PC mux -at top right- with jump signal as it's selector. This is how i basicly implemented other instructions too. Currently all the instructions in FinalProject.pdf work except Store Byte, Store Conditional, Store Halfword, Jump And Link. Also didn't use ALU control unit but instead generated ALU signals inside the Control Unit.

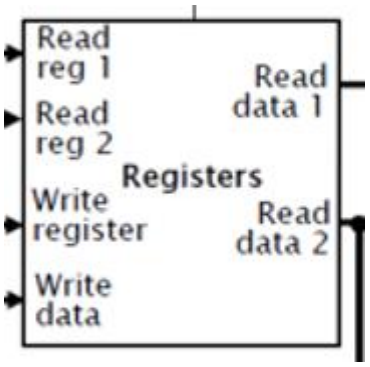
2. METHOD

First of all i want to talk about how things work then i will get into my modules and extra things that i added. Firstly we get the instruction from Instruction memory with program counter being initially 0. Then it parses the instruction to opCode, rs, rt, rd, shamt, immediate, address and funct. And then give opCode and funct to Control Unit as inputs and it generates regDst, aluControl, jump, branch, memRead, memtoReg, memWrite, aluSrc, regWrite signals. After that it calculates the next PC, and chooses register destination. After that, it reads data from Registers unit, sign or zero extends the immediate, generates jump address, selects ALU Unit's second input. After ALU's output, there's a selection of memory destination followed by Memory read/write stage and selection of write_data for write-back stage and then change PC to finish the whole cycle. I will be giving more details in modules about some of the stages of the cycle.



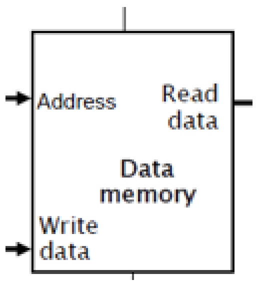
module mips_instr_mem(instruction, program_counter);

Instruction Memory block reads instructions from instruction.mem initially then returns the instruction at given program counter.



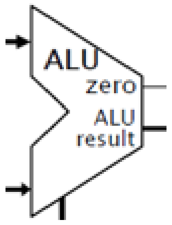
module mips_registers(read_data_1, read_data_2, write_data, read_reg_1, read_reg_2, write_reg, signal_reg_write, clk);

Registers block reads register data from registers.mem and stores in registers. If the regWrite signal is 1 stores write_data in registers else reads data from read_reg_1 and read_reg_2. I've had some problems with writing the data back. In mips core, i call this twice, first with write signal 0 for reading data and second one for writing the data back with regWrite signal. The input write_data was always x and couldn't really figure out why.



module mips_data_mem (read_data, mem_address, write_data, sig_mem_read, sig_mem_write)

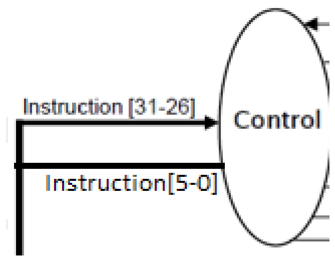
After initially reading data memory from data.mem file to data_mem register, depending on read signal and write signal reads data from data_mem or writes back to data_mem[mem_address]



module mips_alu(outp, zero, data_A, data_B, shamt, ctrl_signal)

ALU Unit gets ctrl_signal from control unit, and makes the operation that the instruction needs. The operations are, And, Or, Add, Sub, Shift left, Shift right, set less than.

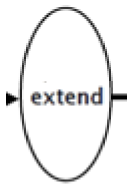
There's also a zero output for branch instructions. If current instruction is branch control signal is 3'b011 which is Sub operation and after that if the result is 0 zero output would be 1 and zero output then goes into and gate with branch signal to see if the branch instruction will be executed.



module mips_control(regDst, aluControl, jump, branch, memRead, memtoReg, memWrite, aluSrc, regWrite, opCode, funct)

Control unit creates all the necessary signals for the given instruction. regDst selects between rt and rd as write register, aluControl is selector for which operation should ALU unit do, jump is the selector between jump address and branch/PC output as i mentioned above, branch selects between PC+4 and Branch address, memRead is for instructions that reads data from memory, memtoReg is for write-back data selection which are ALU output and data read from memory, aluSrc selects second input of the ALU Unit, regWrite signal is 1 when there's a write-back stage for the current instruction.

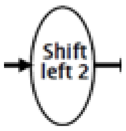
Inputs are opCode and funct field of the instruction. Since the R-type instructions have same signals except aluControl signal and jr instruction, checked if opCode is 0 first, then if the function field is 6'b001000 which is jr instruction, generated correct signals otherwise all the R-type signals are same except aluControl so generated those signals accordingly and when the opCode isn't 0, for every instruction, generated signals one by one. After that, grouping them by their operations which they require in Alu Unit, generated their aluControl signals.



module zeroExtend_16_32(outp, inp)

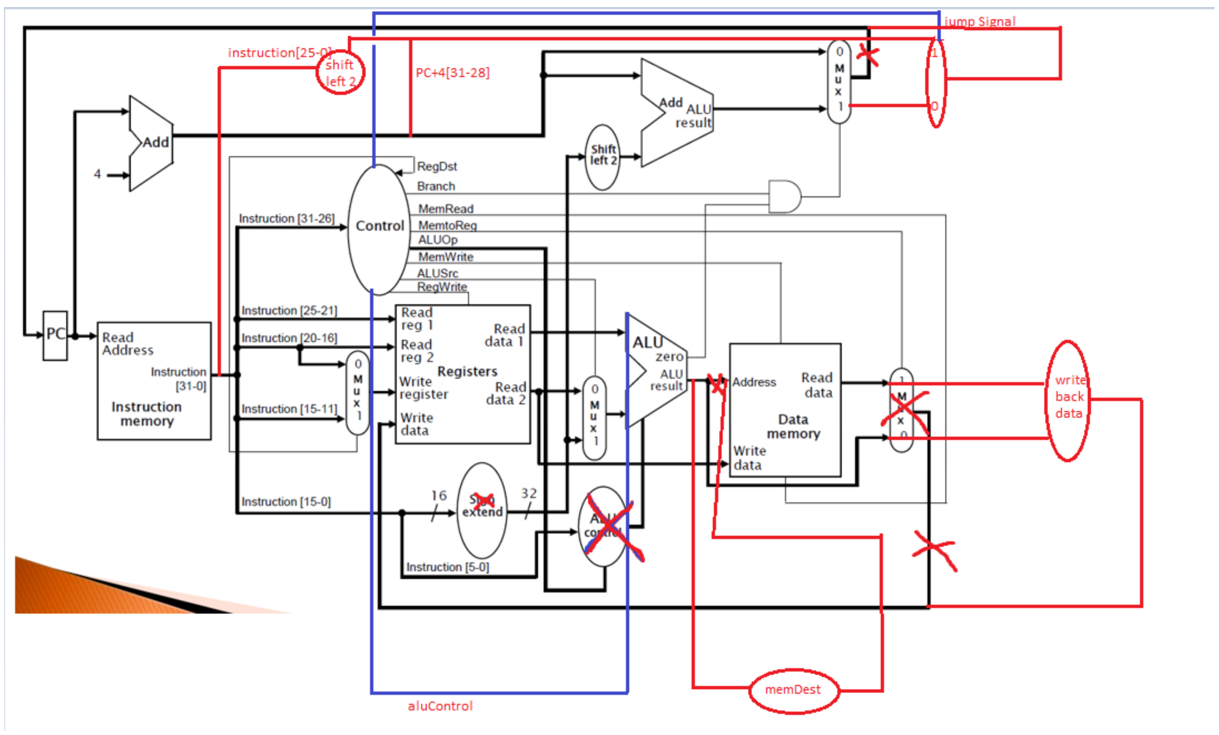
module signExtend_16_32(outp, inp)

signExtend_16_32and zeroExtend_16_32 modules gets immediate as input and sign or zero extends depending on the instruction.



module leftShifter_2bit(outp, inp)

Shift left module is used for J and branch type instructions where address has to be shifted left twice. Reasoning behind shifting is addresses must be 4 or 4s multipliers and we don't store those 2 bits to be able to hold address along with other contents in 32 bit.



module mips_core(clock)

mips_core module is the module that the whole cycle starts and ends. It connects all the modules altogether. Calculates next PC value, fetches instruction from instruction memory, gets control signals from Control Unit, reads register from Registers block, sign and zero extends the immediate, reads/writes from/to Data Memory block, Writes back data to register if necessary, jumps to new PC if necessary and branches to new instruction if necessary.

```verilog
assign memDest = (opCode == 6'b100100 || opCode == 6'b100101 || opCode == 6'b110000 || opCode == 6'b100011 ) ? (read_data_1 + signExtendOutp) : read_data_2;
```

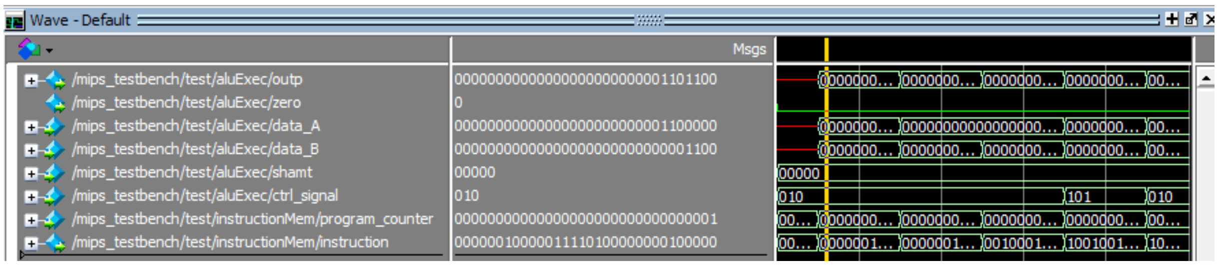
This is what happens inside memDest block, if the instruction is lbu, lhu, ll or lw memDest is read_data_1 + signExtendedImmediate else it's read_data_2.

```verilog
assign write_data = (~(memtoReg)) ? aluOutp :
                     (opCode == 6'b000011) ? PC+2 : //Jal^
                     (opCode == 6'b100100) ? {24'b0, memOutp[7:0]} ://lbu
                     (opCode == 6'b100101) ? {16'b0, memOutp[15:0]} ://lhu
                     (opCode == 6'b001111) ? { immediate, 16'b0 } : //lui
                      memOutp; //ll, lw
```

This is the write back data block and how the output is assigned.

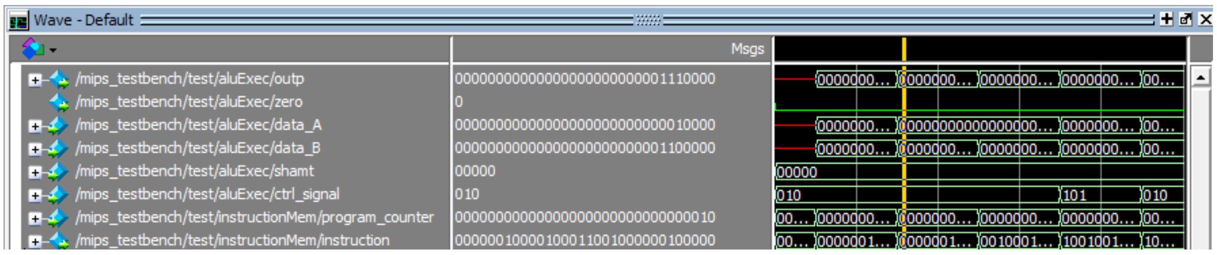
3. Results

3.1 Testbench Results

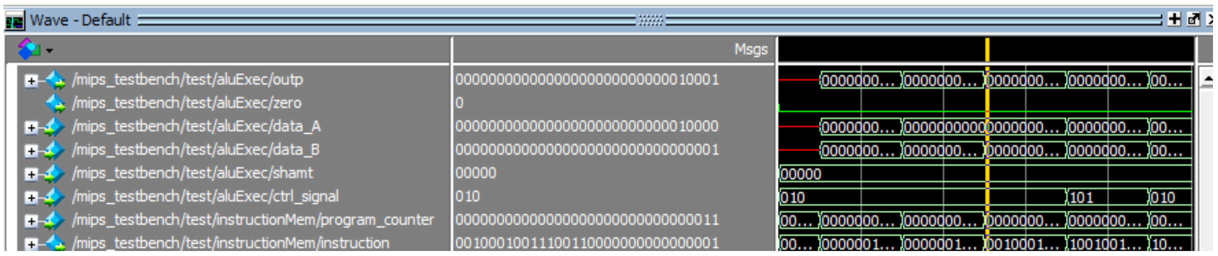


```
Wave - Default
                                                   Msgs
/mips_testbench/test/aluExec/outp                  00000000000000000000000001101100
/mips_testbench/test/aluExec/zero                  0
/mips_testbench/test/aluExec/data_A                00000000000000000000000001100000
/mips_testbench/test/aluExec/data_B                00000000000000000000000000001100
/mips_testbench/test/aluExec/shamt                 00000
/mips_testbench/test/aluExec/ctrl_signal           010
/mips_testbench/test/instructionMem/program_counter 00000000000000000000000000000001
/mips_testbench/test/instructionMem/instruction    00000010000011110100000000100000
```

Add Instruction

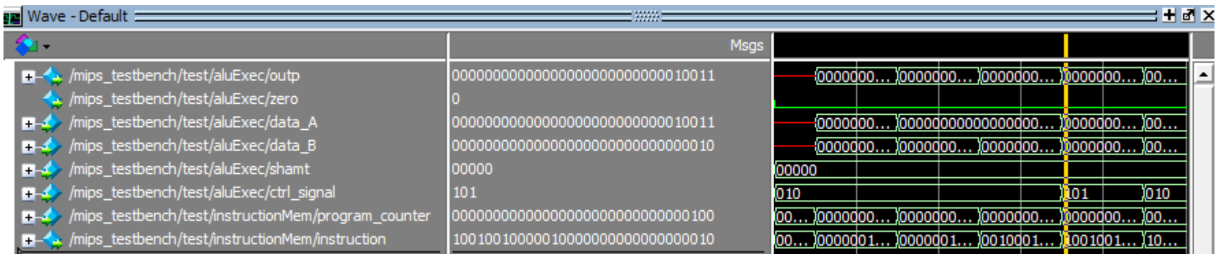


```
Wave - Default
                                                   Msgs
/mips_testbench/test/aluExec/outp                  00000000000000000000000001110000
/mips_testbench/test/aluExec/zero                  0
/mips_testbench/test/aluExec/data_A                00000000000000000000000000010000
/mips_testbench/test/aluExec/data_B                00000000000000000000000001100000
/mips_testbench/test/aluExec/shamt                 00000
/mips_testbench/test/aluExec/ctrl_signal           010
/mips_testbench/test/instructionMem/program_counter 00000000000000000000000000000010
/mips_testbench/test/instructionMem/instruction    00000010000100011001000000100000
```



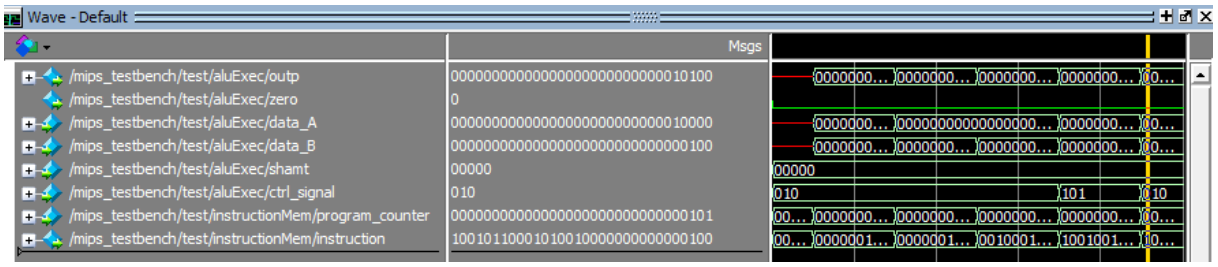
```
Wave - Default
                                                  Msgs
/mips_testbench/test/aluExec/outp                 00000000000000000000000000010001
/mips_testbench/test/aluExec/zero                 0
/mips_testbench/test/aluExec/data_A               00000000000000000000000000010000
/mips_testbench/test/aluExec/data_B               00000000000000000000000000000001
/mips_testbench/test/aluExec/shamt                00000
/mips_testbench/test/aluExec/ctrl_signal          010
/mips_testbench/test/instructionMem/program_counter 00000000000000000000000000000011
/mips_testbench/test/instructionMem/instruction   00100010011100110000000000000001
```

Addi Instruction



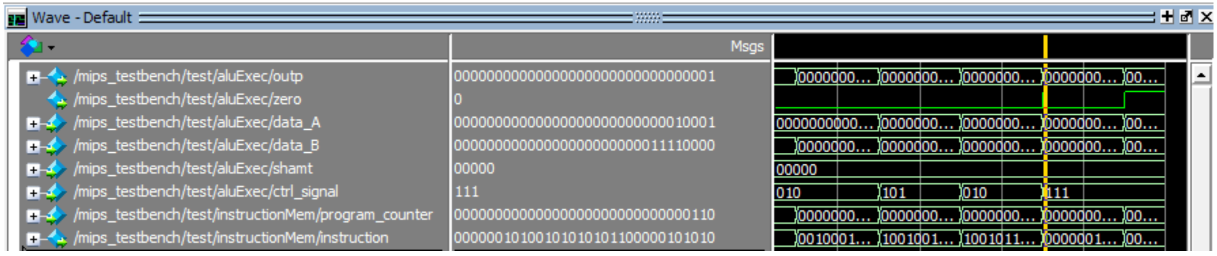
```
Wave - Default
                                                  Msgs
/mips_testbench/test/aluExec/outp                 00000000000000000000000000010011
/mips_testbench/test/aluExec/zero                 0
/mips_testbench/test/aluExec/data_A               00000000000000000000000000010011
/mips_testbench/test/aluExec/data_B               00000000000000000000000000000010
/mips_testbench/test/aluExec/shamt                00000
/mips_testbench/test/aluExec/ctrl_signal          101
/mips_testbench/test/instructionMem/program_counter 00000000000000000000000000000100
/mips_testbench/test/instructionMem/instruction   10010010000100000000000000000010
```

Lbu Instruction



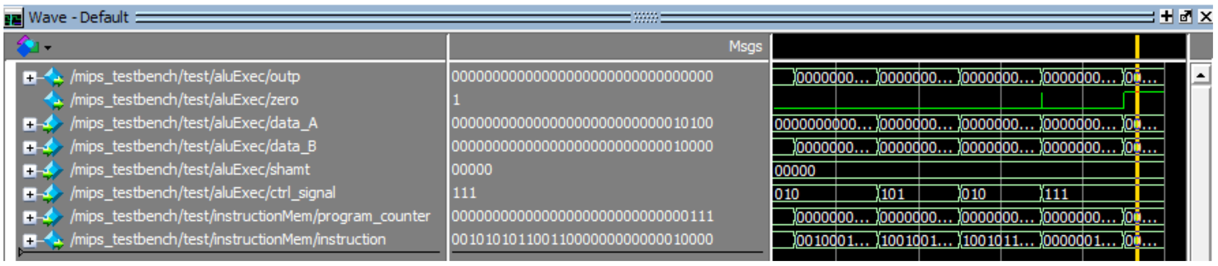
```
Wave - Default
                                                  Msgs
/mips_testbench/test/aluExec/outp                 00000000000000000000000000010100
/mips_testbench/test/aluExec/zero                 0
/mips_testbench/test/aluExec/data_A               00000000000000000000000000010000
/mips_testbench/test/aluExec/data_B               00000000000000000000000000000100
/mips_testbench/test/aluExec/shamt                00000
/mips_testbench/test/aluExec/ctrl_signal          010
/mips_testbench/test/instructionMem/program_counter 00000000000000000000000000000101
/mips_testbench/test/instructionMem/instruction   10010110001010010000000000000100
```

Lhu Instruction



```
Wave - Default
                                                  Msgs
/mips_testbench/test/aluExec/outp                 00000000000000000000000000000001
/mips_testbench/test/aluExec/zero                 0
/mips_testbench/test/aluExec/data_A               00000000000000000000000000010001
/mips_testbench/test/aluExec/data_B               00000000000000000000000011110000
/mips_testbench/test/aluExec/shamt                00000
/mips_testbench/test/aluExec/ctrl_signal          111
/mips_testbench/test/instructionMem/program_counter 00000000000000000000000000000110
/mips_testbench/test/instructionMem/instruction   00000010100101010101100000101010
```

Slt Instruction



```
Wave - Default
                                                  Msgs
/mips_testbench/test/aluExec/outp                 00000000000000000000000000000000
/mips_testbench/test/aluExec/zero                 1
/mips_testbench/test/aluExec/data_A               00000000000000000000000000010100
/mips_testbench/test/aluExec/data_B               00000000000000000000000000010000
/mips_testbench/test/aluExec/shamt                00000
/mips_testbench/test/aluExec/ctrl_signal          111
/mips_testbench/test/instructionMem/program_counter 00000000000000000000000000000111
/mips_testbench/test/instructionMem/instruction   00101010110011000000000000010000
```

Slti Instruction

3.2 Analysis

During tests i realized that if you give registers block regWrite signal, module won’t work properly so as a workaround, i used 1’b0 instead of regWrite and as a result, writing back to register is not working. Also unsigned operations and sc, sh and sb instructions are not supported aswell.

Azmi Utku Sezgin

131044048