CS230 Project Report

IITB-RISC-22 (multi-cycle processor)

Dhvanit Beniwal, 200050035
Margav Mukeshbhai Savsani 200050072
Pasala Poorna Teja, 200050101
Sartaj Islam, 200050128

DESCRIPTION:

We have designed a multi-cycle processor, IITB-RISC-22, with the given instruction set architecture. There are 8 (16 bit each) registers. There is a memory block of 2^{16} memory elements of 16 bit each, one for every memory address implemented as an entity of its own.

The top level entity, **Top_entity**, has a clock and reset input, and it assembles all the components of the processor in its architecture. Each individual component has its own vhdl file including one for the controller, the memory, the ALU, register file etc.

WORKING (Control):

The controller takes as input a **state,** an opcode (of the instruction) and the carry and zero bits (both from the instruction as **cz** and the carry and zero calculated as **c**, **z**).

The outputs for the controller are a lot of **control signals** (variables ending with "_cs") where the variable name makes it clear which component this signal is going to go to, and the corresponding component (in its own vhdl file) will expect that control signal as its input. The top level entity will take care of this.

The controller also outputs various 1 bit signals (ending with "_write"), where the variable name makes it clear which component it goes to. A **write signal** for some component needs to be **1** if we want the corresponding component to do a **write** operation. The components can implement this easily with a process and an if statement

The controller architecture implements a state machine with 22 states (state 0 is the initial state and after 20 other states there is a reset state). Given any state, we change the control signals appropriately and update the **nextState** signal. The comments in the vhdl file whenever **nextState** is updated illustrate which instruction it refers to (ADD, NDU, LM, JAR etc). These comments may be referred to for details about different states.

At each clock cycle (unless **reset** bit is **1**) a **Top_entity** process assigns **nextState** to **state** so that the controller can process the next state now.

Some major components:

We already mentioned the controller. **Memory.vhd** implements a memory as an array of 16 bit signals (2^{16} of them). It receives a memory address, Data and **M_write** bit as inputs. It outputs the data in that memory address as **M_out** (should it be needed). If the **M_write** bit is **1** it writes the value **Data** into the memory at the given address. This way there are no separate processes for read and write.

The instruction register component (**IR.vhd**) has an input port for the instruction and a 1 bit input port **IR_write**. If this bit is **1** then it processes the 16 bit instruction input (**M_out** because it came from the memory) into opcode, carry-zero bits, RA, RB, RC, etc.

ALU.vhd needs a 2 bit **ALU_control** that selects what it has to do. **00** means addition, **01** means subtraction, **10** means bitwise NAND. But the controller provides a different 2 bit signal **ALU_op** where **10** means "do according to the instruction". **ALU_control.vhd** takes care of this by taking **ALU_op** and giving **ALU_control**. All it needs to do is to look at the third bit of the opcode of the instruction. If it is **1** we know it's NAND and if it's **0** we know it's ADD.

State Flow (for instructions)

All instructions start at state 0, and move to state 1 where the instruction is inferred and sent on the respective next state. All instructions end at state 20. The following are the states used while executing a particular instruction:

- ADD : 1⟶2⟶3⟶4⟶20
- ADC : 1⟶2⟶3⟶4⟶20
- ADZ : 1⟶2⟶3⟶4⟶20
- ADL : 1⟶2⟶5⟶4⟶20
- ADI : 1⟶2⟶6⟶4⟶20
- NDU : 1⟶2⟶3⟶4⟶20
- NDC : 1⟶2⟶3⟶4⟶20
- NDZ : 1⟶2⟶3⟶4⟶20
- LW : 1⟶2⟶6⟶7⟶4⟶20
- SW : 1⟶2⟶6⟶8⟶20
- BEQ : 1⟶2⟶3⟶10⟶20
- LHI : 1⟶4⟶20
- LM : 1⟶9⟶11⟶12⟶13⟶20 (will visit more than once as 13⟶11 etc)
- SM : 1⟶9⟶11⟶12⟶13⟶20 (will visit more than once as 13⟶11 etc)
- JAL : 1⟶14⟶15⟶20
- JLR : 1⟶16⟶17⟶20
- JRI : 1⟶18⟶19⟶20

20⟶0⟶1 (for the next instruction)

Control signals given in each state

The control signals ‘sent’ for each state are as follows:

- State 0:
  - All signals initialised to zero except
  - IR_write = 1
  - ALUReg_write = 1
- State 1:
  - IR_write = 0
  - ALUReg_write = 0
  - carry_write = 0
  - zero_write = 0
  - mux_before_zeroreg_cs = 0
  - pc_write = 1
  - mux_before_pc_cs = 10
- State 2:
  - pc_write <= '0';
  - mux_before_zeroreg_cs <='0';
  - AReg_write <= '1';
  - BReg_write <= '1';
  - mux_before_RFA1_cs <= "01";
  - mux_before_RFA2_cs <= "01";
- State 3:
  - AReg_write <= '0';
  - BReg_write <= '0';
  - mux_before_zeroreg_cs <='0';
  - ALUReg_write<= '1';
  - mux_before_ALUA_cs <= "000";
  - mux_before_ALUB_cs <= "000";
- State 4:
  - carry_write <= '0';
  - zero_write <= '0';
  - imm6 <= '0';
  - mux_before_zeroreg_cs <='0';
  - pc_write <= '0';
  - BReg_write <= '0';
  - ALUReg_write<= '0';
  - MDR_write <= '0';
  - mux_before_MEMA1_cs <= "00";
  - RF_write <= '1';
- State 5:
  - AReg_write <= '0';
  - BReg_write <= '0';
  - mux_before_zeroreg_cs <='0';

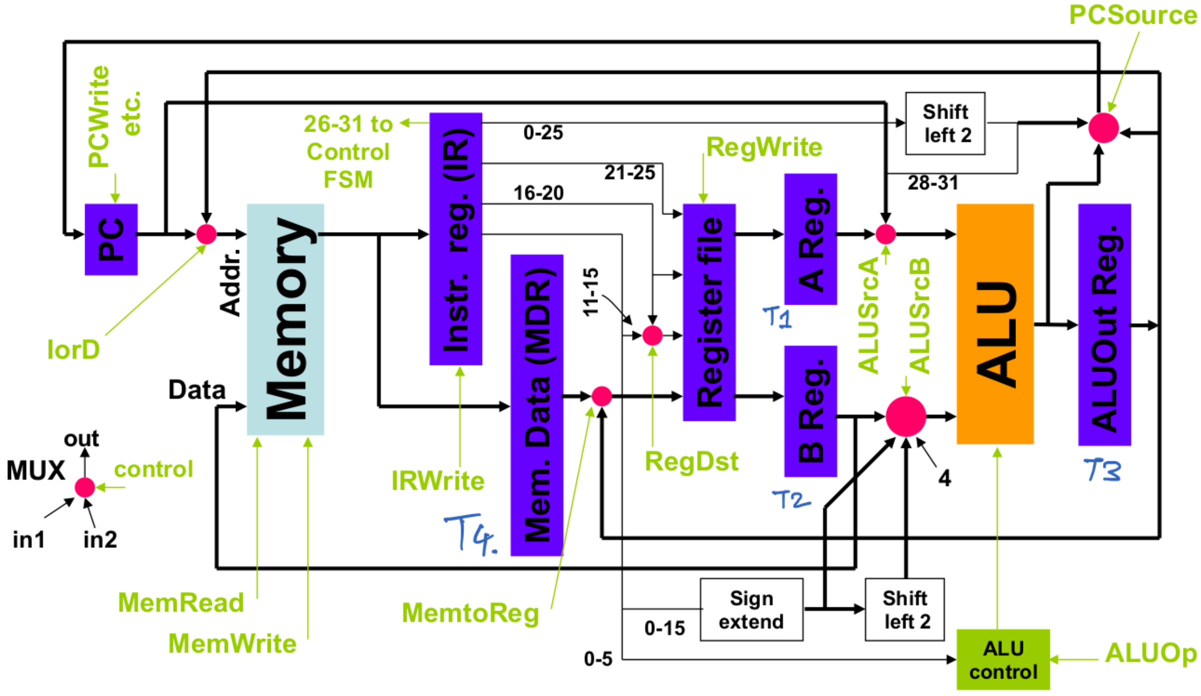
- carry_write <= '1';
    - zero_write <= '1';
    - ALUReg_write<= '1';
    - mux_before_ALUA_cs <= "000";
    - mux_before_ALUB_cs <= "010";
- State 6:
    - AReg_write <= '0';
    - BReg_write <= '1';
    - mux_before_zeroreg_cs <='0';
    - ALUReg_write<= '1';
    - imm6 <= '1';
    - mux_before_RFA2_cs <= "10";
    - mux_before_ALUA_cs <= "000";
    - mux_before_ALUB_cs <= "001";
- State 7:
    - ALUReg_write<= '0';
    - mux_before_zeroreg_cs <='1';
    - carry_write <= '0';
    - zero_write <= '1';
    - imm6 <= '0';
    - BReg_write <= '0';
    - MDR_write <= '1';
    - mux_before_MEMA1_cs <= "10";
- State 8:
    - mux_before_zeroreg_cs <='0';
    - carry_write <= '0';
    - zero_write <= '0';
    - ALUReg_write<= '0';
    - imm6 <= '0';
    - BReg_write <= '0';
    - M_write <= '1';
    - mux_before_MEMA1_cs <= "10";
- State 9:
    - mux_before_zeroreg_cs <='0';
    - carry_write <= '0';
    - zero_write <= '0';
    - pc_write <= '0';
    - BReg_write <= '1';
    - mux_before_RFA2_cs <= "10";
    - LMSM_cs <= "11";
- State 10:
    - mux_before_zeroreg_cs <='0';
    - carry_write <= '0';
    - zero_write <= '0';

- State 11:
  - mux_before_zeroreg_cs <='0';
  - carry_write <= '0';
  - zero_write <= '0';
  - AReg_write <= '0';
  - LMSM_cs <= "10";
- State 12:
  - mux_before_zeroreg_cs <='0';
  - carry_write <= '0';
  - zero_write <= '0';
- State 13:
  - mux_before_zeroreg_cs <='0';
  - carry_write <= '0';
  - zero_write <= '0';
  - LMSM_cs <= "00";
- State 14:
  - mux_before_zeroreg_cs <='0';
  - carry_write <= '0';
  - zero_write <= '0';
  - pc_write <= '0';
  - ALUReg_write<= '1';
  - RF_write <= '1';
  - mux_before_RFD3_cs <= "1000";
  - mux_before_RFA3_cs <= "00";
  - imm6 <= '0';
  - mux_before_ALUA_cs <= "100";
  - mux_before_ALUB_cs <= "001";
- State 15:
  - mux_before_zeroreg_cs <='0';
  - carry_write <= '0';
  - zero_write <= '0';
  - ALUReg_write<= '0';
  - RF_write <= '0';
  - pc_write <= '1';
  - mux_before_pc_cs <="10";
- State 16:
  - mux_before_zeroreg_cs <='0';
  - carry_write <= '0';
  - zero_write <= '0';
  - pc_write <= '0';
  - AReg_write <= '1';
  - RF_write <= '1';
  - mux_before_RFD3_cs <= "1000";
  - mux_before_RFA3_cs <= "00";

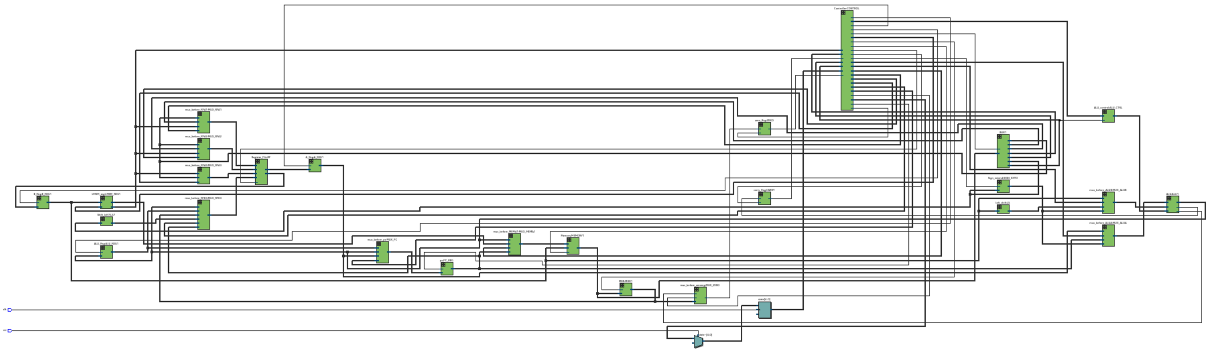
  - mux_before_RFA1_cs <= "01";
- State 17:
  - mux_before_zeroreg_cs <='0';
  - carry_write <= '0';
  - zero_write <= '0';
  - AReg_write <= '0';
  - RF_write <= '0';
  - pc_write <= '1';
  - mux_before_pc_cs <="00";
- State 18:
  - mux_before_zeroreg_cs <='0';
  - carry_write <= '0';
  - zero_write <= '0';
  - pc_write <= '0';
  - AReg_write <= '1';
  - mux_before_RFA1_cs <= "10";
- State 19:
  - mux_before_zeroreg_cs <='0';
  - carry_write <= '0';
  - zero_write <= '0';
  - AReg_write <= '0';
  - pc_write <= '1';
  - imm6 <= '0';
  - mux_before_ALUA_cs <= "000";
  - mux_before_ALUB_cs <= "001";
  - mux_before_pc_cs <="01";
- State 20:
  - mux_before_zeroreg_cs <='0';
  - carry_write <= '0';
  - zero_write <= '0';
  - imm6 <= '0';
  - pc_write <= '0';
  - M_write <= '0';
  - RF_write <= '1';
  - mux_before_RFD3_cs <= "1000";
  - mux_before_RFA3_cs <= "10";

DATAPATH:

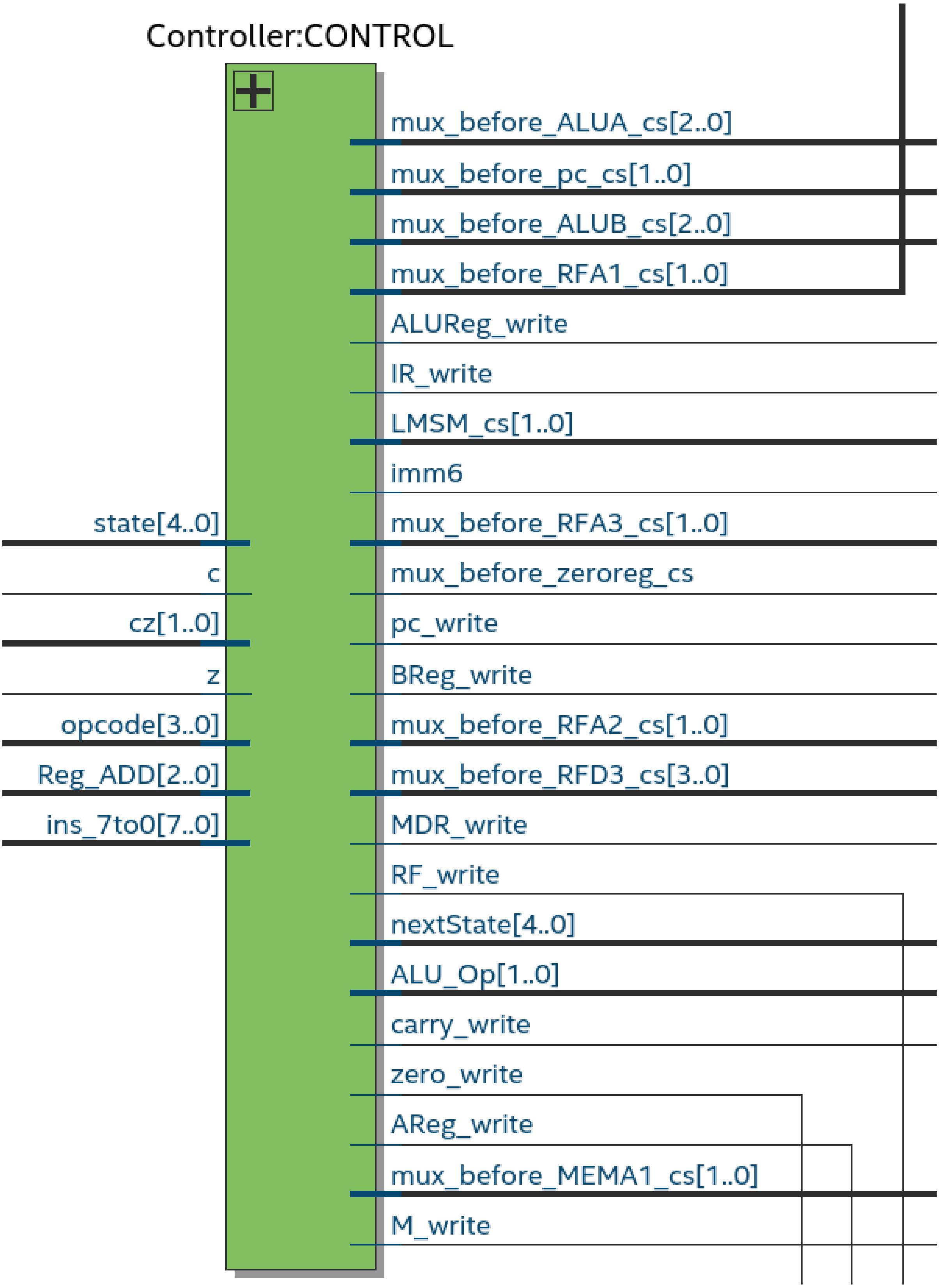
Apart from a few minor changes, the datapath is constructed according to the following diagram, with similar file names so it's easy to follow:



The circuit diagram from our implementation is



The controller in this diagram is the 'tall' box near the top-right.



Controller:CONTROL
state[4..0]
c
cz[1..0]
z
opcode[3..0]
Reg_ADD[2..0]
ins_7to0[7..0]
mux_before_ALUA_cs[2..0]
mux_before_pc_cs[1..0]
mux_before_ALUB_cs[2..0]
mux_before_RFA1_cs[1..0]
ALUReg_write
IR_write
LMSM_cs[1..0]
imm6
mux_before_RFA3_cs[1..0]
mux_before_zeroreg_cs
pc_write
BReg_write
mux_before_RFA2_cs[1..0]
mux_before_RFD3_cs[3..0]
MDR_write
RF_write
nextState[4..0]
ALU_Op[1..0]
carry_write
zero_write
AReg_write
mux_before_MEMA1_cs[1..0]
M_write

File Description

ADD.vhd
This file adds two 16 bit inputs A and B along with a single bit carry input cin and stores the sum in S and the carry output in cout. This uses ripple carry adder to add the signals with the help of ADD_4bit which adds 4 bit numbers in the same fashion.

ALU.vhd
This file performs the alu operation on inputs ALU_A and ALU_B based on the input signal from ALU_control and stores the result in ALU_C and also sets the carry flag ‘c’ and zero flag ‘z’.

ALU_control.vhd
This entity takes inputs as ALU_op and ins_2 which is taken from IR. This provides the output ALU_sel to the input port ALU_control of ALU which in turn decides what operation to perform in ALU.

ALU_Reg.vhd
It has 2 input ports, one is 16 bit ALU_C and the other is single bit ALUReg_write. The output which is result_out is changed here to ALU_C based on the input ALUReg_write.

A_Reg.vhd
This file has 2 input ports, one is 16 bit D1 and the other is AReg_write. According to the signal AReg_write the 16 bit output signal is changed to D1.

B_Reg.vhd
This file has 2 input ports, one is 16 bit D2 and the other is BReg_write. According to the signal BReg_write the 16 bit output signal is changed to D2.

IR.vhd
In this file the 16 bit instruction extracted from the memory is taken as input in M_out and this is then broken to get different outputs which include RA, RB, RC, op_code, cz, ins_8to0, ins_7to0. All these output ports are defined according to the requirement in the problem statement. Also the input port IR_write decides if the value of M_out have to written in the ins signal of this file.

left_shift.vhd
This entity does the left shift operation on the 16 bit input RC and stores the output in the 16 bit output port B.

LMSM_reg.vhd
This entity is primarily used for handling the instructions LM and SM. This is used as an iterator for indexes in Register file and Memory. The Reg_ADD and Mem_ADD are the indices which are initialised by 0 and Mem_ADD_i taken as input and they are increased based on the value of control_sig.

MDR.vhd

This file has 2 input ports, one is 16 bit M_out and the other is MDR_write. According to the signal MDR_write the 16 bit output signal is changed to M_out.

carry_Reg.vhd

This file has 2 input ports, one is single bit ALU_carry and the other is carry_write. According to the signal carry_write the output signal is changed to ALU_carry.

zero_Reg.vhd

This file has 2 input ports, one is single bit ALU_zero and the other is zero_write. According to the signal zero_write the output signal is changed to ALU_zero.

Memory.vhd

This file has an array of Memory which can be accessed through this file only. Read and write actions are performed on memory through this file. It takes a 16 bit index A1 as input and returns the value in M_out. If M_write signal is set then it write the 16 bit Data input to the memory at index A1.

pc.vhd

This file has 2 input ports, one is 16 bit pcin and the other is pc_write. According to the signal pc_write the 16 bit output signal is changed to pcin.

Register_File.vhd

This file has an array of Registers of size 8. This takes 3 bit inputs A1 and A2 which are indices for the Register and the values at these indices are stored in 16 bits output ports D1 and D2 respectively. Also if RF_write signal is set then the value D3 is stored in the register at index A3, both taken as input.

Shift_left7.vhd

This entity takes the 9 bits input imm9 and stores the output after performing a left shift by 7 bits in a 16 bit output port result.

Sign_extend.vhd

This entity performs the sign extension of the input port ins_8to0 and stores the result. Further based on the imm6 input it decides the number of bits by which sign has to be extended.

Top_entity.vhd

This file creates the components of all the entities and maps the signals defined here. This creates a connection between all the ports in the entities and at each cycle our program functions all the required actions defined in the Controller.