Noah Niedzwiecki
ELEC 5200
CPU Project 4
April 5, 2019

Introduction:

This project continues the verification and testing of the Niedzwiecki ISA 16-bit CPU as detailed by project portions one through three. To recap, each instruction is comprised of 16-bits. The first four of which are the instruction's opcode. The functionality of the remaining instruction bits depend on the opcode. If R-type instruction, the next eight bits determine the registers to be operated on. If J-type instruction, the remaining twelve bits determine the program counter offset to jump to and the previous program counter address is stored in register F. If I-type, the next eight bits determine the registers to evaluate when considering branching; the last four are the bits that will be added to the program counter. Figure 1 below shows the datapath for CPU

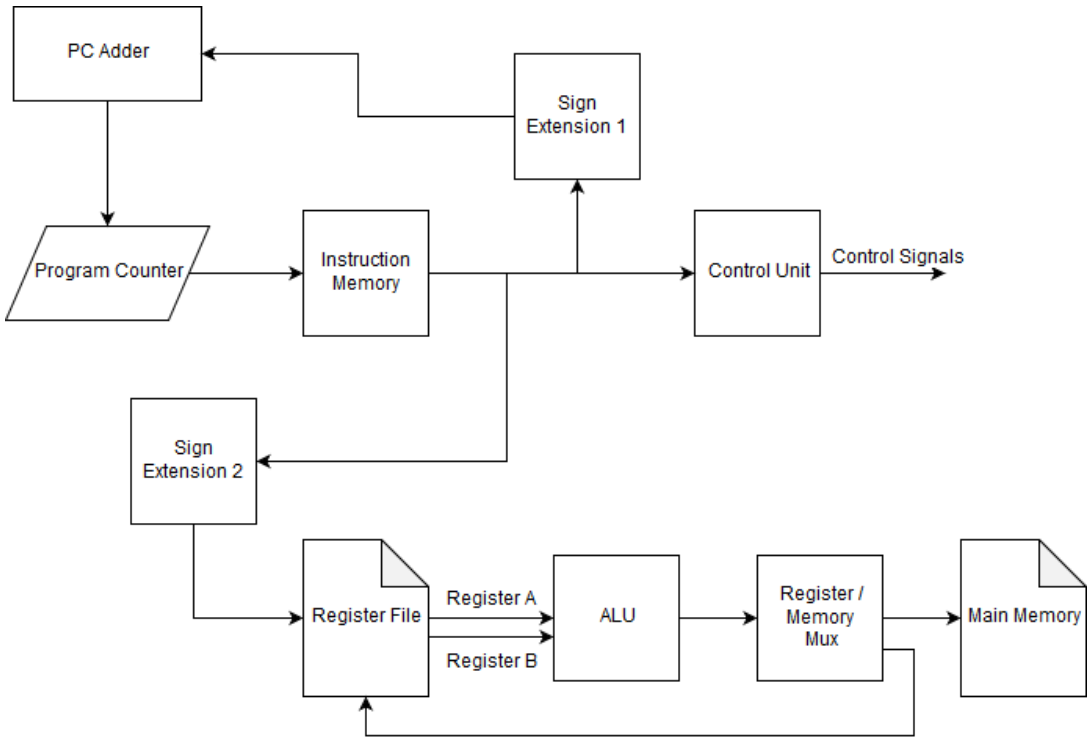

Figure 1: CPU Datapath Diagram

Instruction Verification:

In this verification process, the register file and memories will be simulated by a variety of signals that can be seen in the following waveform images: WRITEBUS, READBUSA, and READBUSB. WRITEBUS will take the place of the output of register F for branches and the data from the main memory. READBUSA and READBUSB replace the data output from the register file.

HALT: With opcode 0000b, this instruction cancels cpu operation. To do this, the adding circuit block outputs address 0x0000 to the program counter which effectively stops all operation. Figure 2 shows the PC at 0xFFFF, at the rising edge of clk at 1ps, the output of bit16addCtrl is 0x0000. This signal represents the output of the PC Adder to be written to the program counter register.

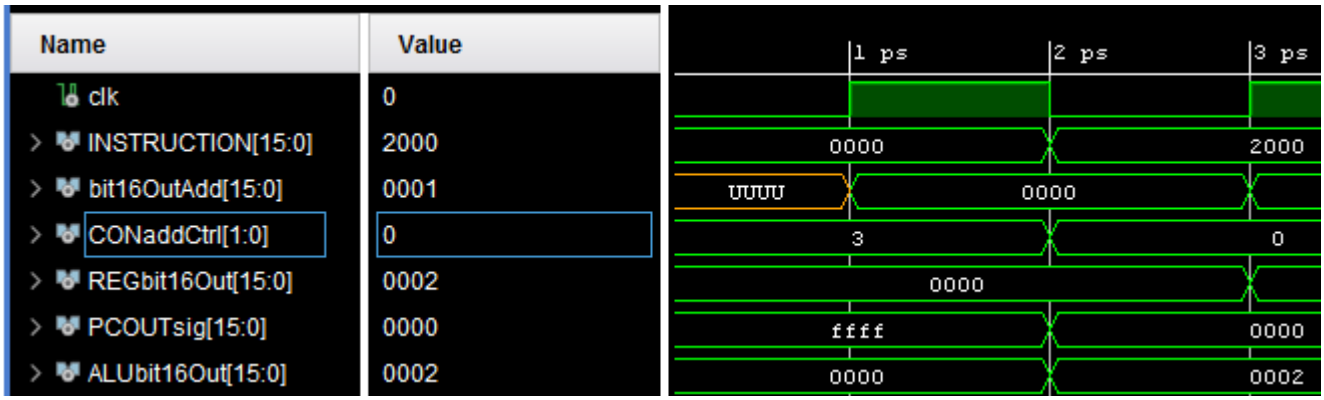

Figure 2: Halt Instruction Setting the PC to 0x0000

JUMP: With opcode 0001b, this instruction adds a 12 bit offset to the program counter. To do this, the 4 MSB of the instruction set the control unit so that the Address Mux selects the 12 bits from the instruction and sends them to sign extension 1. The 16 bits resulting value sent to the adder circuit which adds the program counter and the offset together and stores it into the program counter. Figure 3 shows the PC Adder output changing to the current PC value(PCOUTsig) to the last twelve bits specified in the instruction.

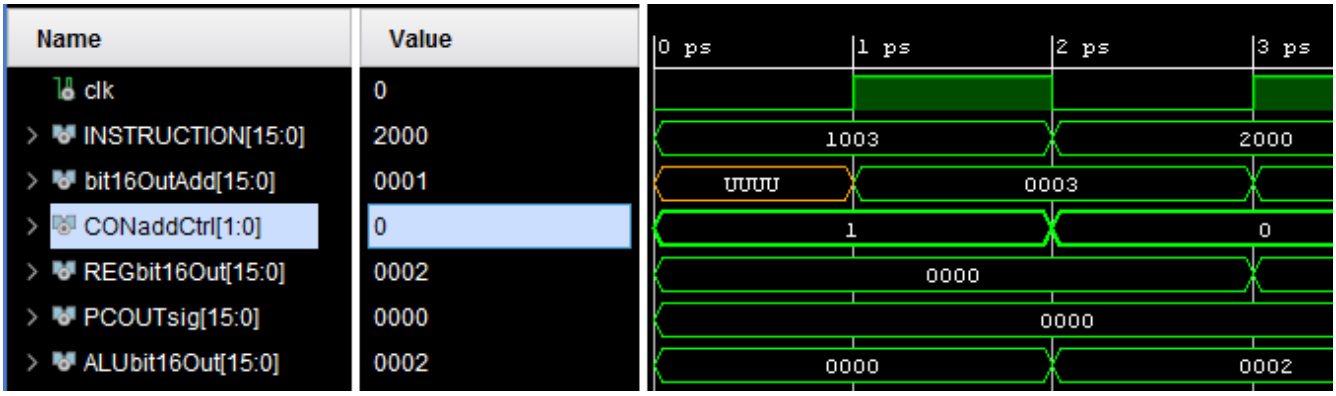

Figure 3: Jump and Link Instruction Offsetting the PC by 0x0003

ADD: With opcode 0010b, this instruction adds 2 registers together and stores the sum in another register. The control unit realizes the opcode and determines which registers need to be accessed. Each register table holds 4 registers so 2 bits of the control line going into these tables act as the address. The Arithmetic Logic Unit has internal buffers which allow for serial data input from the registers in case the 2 referenced registers are in the same table. The control unit enables the ALU and selects which of the 4 arithmetic commands to execute on the input. The output is connected to the mem/reg mux which would then send it to the register mux which would then write to the proper register. Once the ALU operation concludes the control unit would then enable writing on the destination register and provide the necessary address. Figure 4 below shows the sum of READBUSA and READBUSB being output by the Register/Memory Mux.

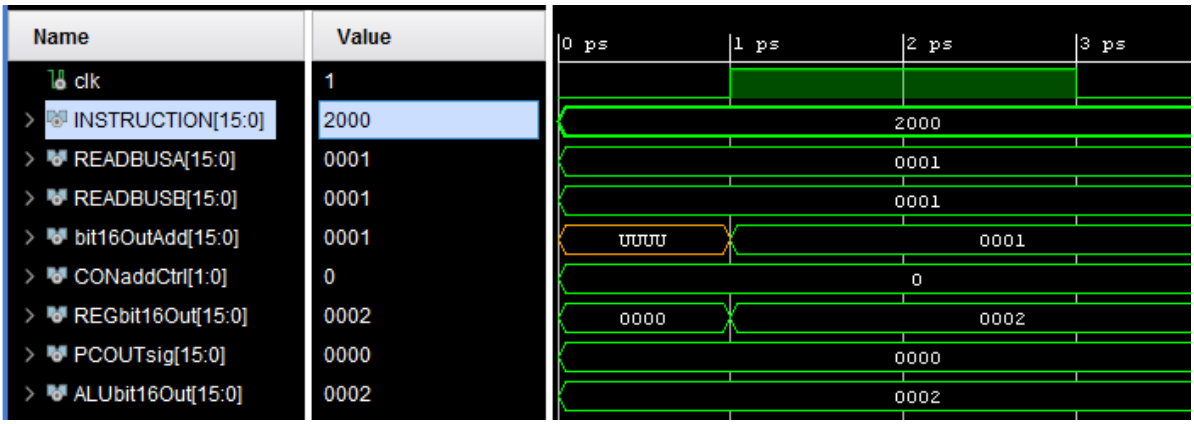

Figure 4: Add Instruction Summing the Two READBUS Values

SUB: with opcode 0011b, this instruction executes the exact same way as add in terms of control unit flow. However, this operation sends single bit to the control unit to signify it a subtraction instruction results in a zero or negative value. Figure 5 below shows the difference of READBUSA and READBUSB being output by the Register/Memory Mux.

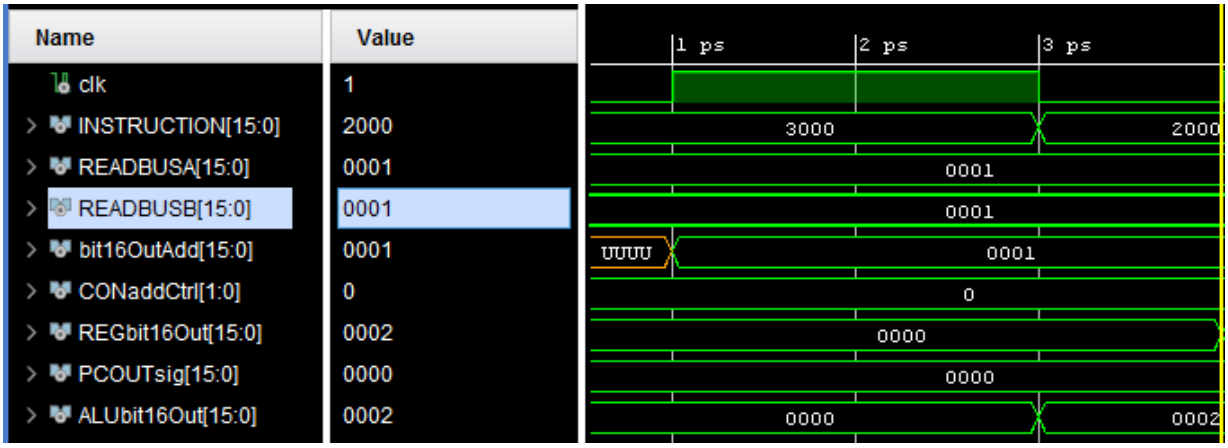

Figure 5: Subtraction Instruction Subteracting the Two READBUS Values

LOAD: with opcode 0100b, this instruction populates a specified register with the contents of a memory address at the program counter plus an 8 bit offset. To do this, the 8 offset bits are sent through the address mux to the sign extender and eventually added with the PC. Instead of being loaded into the PC, this value is sent to the data memory module which loads the address value onto the register mux to be sent to the appropriate register file location. Figure 6 shows the MEMREAD being output by the register/memory mux.

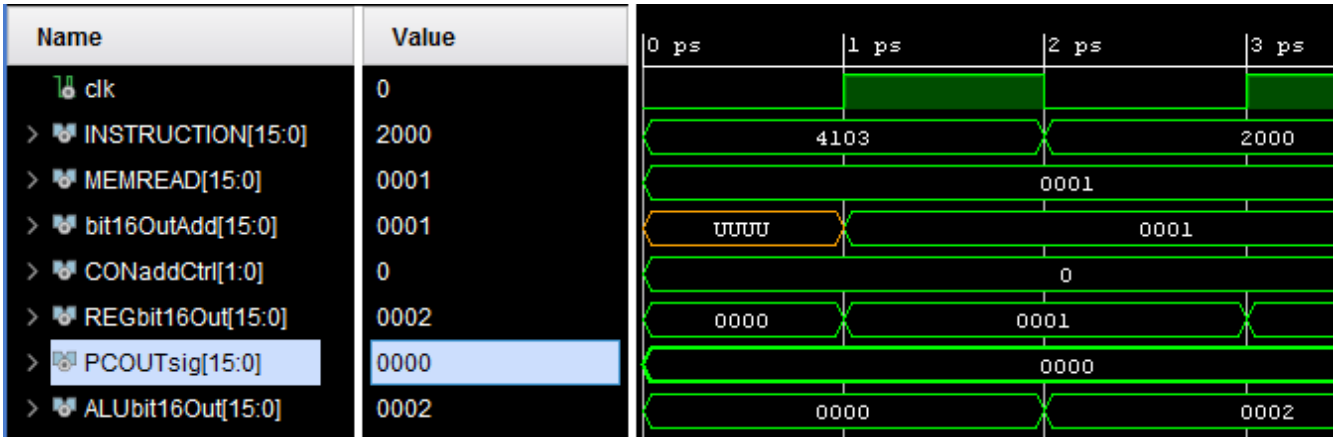

Figure 6: Load Instruction Reading MEMREAD Bus into the Register/Memory Mux

LOADC: with opcode 0101b, this instruction loads a register with an 8 bit immediate value. The 8 bits are sign extended with the sign extension 2 module and the result is then loaded into the proper register. Again, the register write and address are specified by the control unit signal. Figure 7 shows destination register 0x0000(CONregTableD) being selected by bits 11-8 of the instruction. CONregTable is the control signal coming from the control unit which specifies a write in the register file.

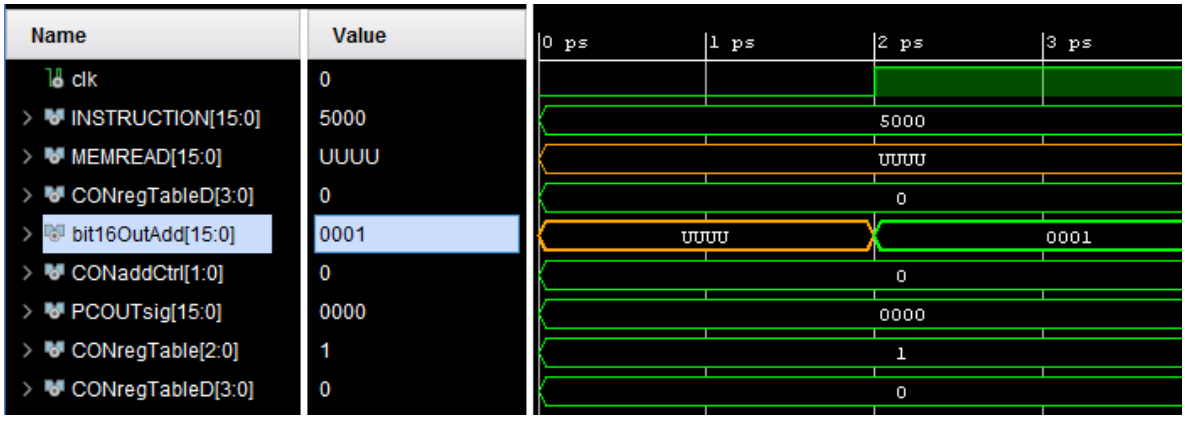

Figure 7: LOADC Instruction Preparing Instruction Bits to be Written into the Register File

MOVE: with opcode 0110b, this instructions copies the value of one register into another. For this to take place, the register being read is loaded into the ALU. Then sent to the register mux through the mem/register mux selector. The register being written to is enabled for such a function and the write is complete. Figure 8 shows READBUSA being written through the ALU to the register/memory mux to be written to the CONregTableD register.

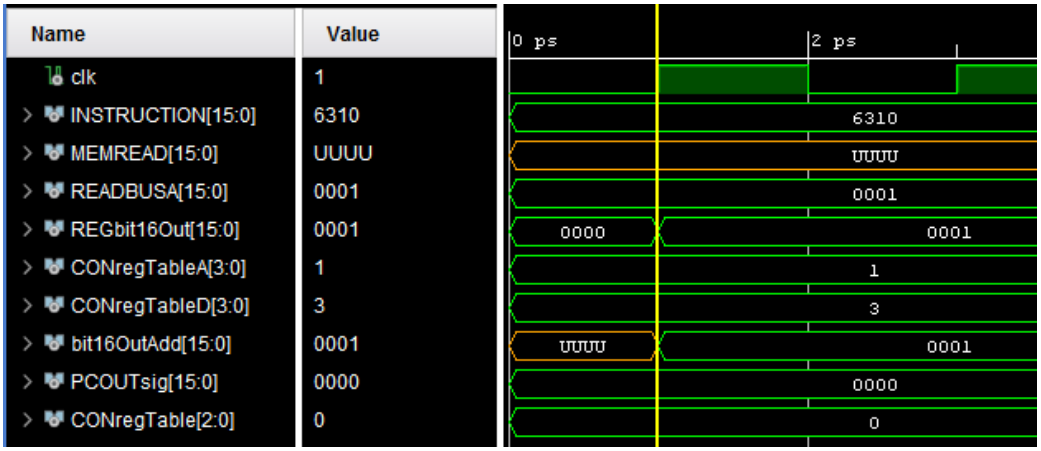

Figure 8: MOVE Instruction Writing READBUSA Data Through to the Register File

STORE: with opcode 0111b, this instruction takes a register's contents and saves it in the data memory at a location 8 bits offset from the PC. First the register is put into the ALU and sent to he mem/reg MUX to the data memory. To determine the data memory location, the 8 bit offset is sent through the address mux to sign extension 1. The value is added with the PC and the result is connected to the data memory module as the address to write to. Figure 9 shows the value of READBUSA being written through the ALU to the register/memory mux to be written to memory.

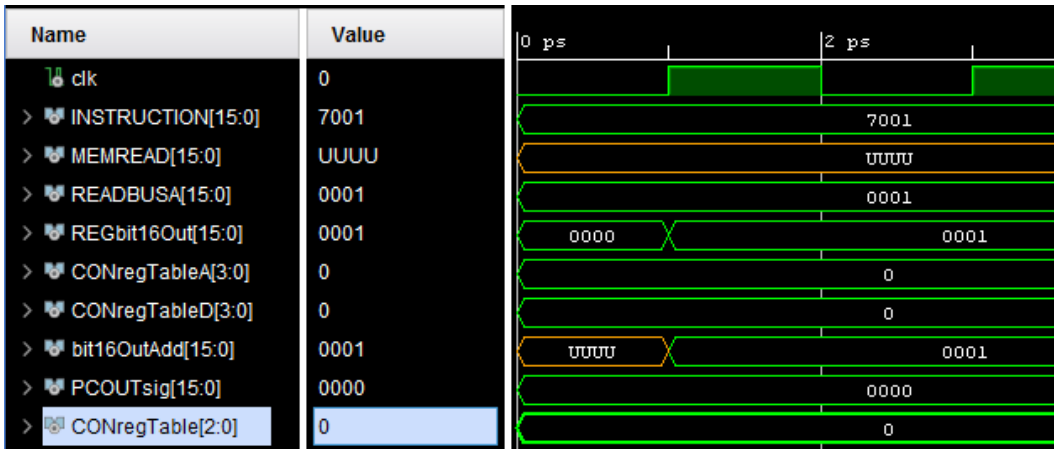

Figure 9: STORE Instruction Writing READBUSA Data Through the Register/Memory Mux

JUMP&LINK: with opcode 1000b, this instruction moves the current PC value to register rP and loads a new PC value according to the 12-bit immediate PC offset. The 8 bits are sent through the address mux to sign extension 1 and then added to the PC. To save the program counter, register table D allows for the data on the program counter line to be written to the last register available. Reg Table D is the only register table that has a line connected to the program counter for it is the only register that should be used to save the PC. Figure 10 shows the PC incrementing by the amount specified in the last twelve bits of the instruction.

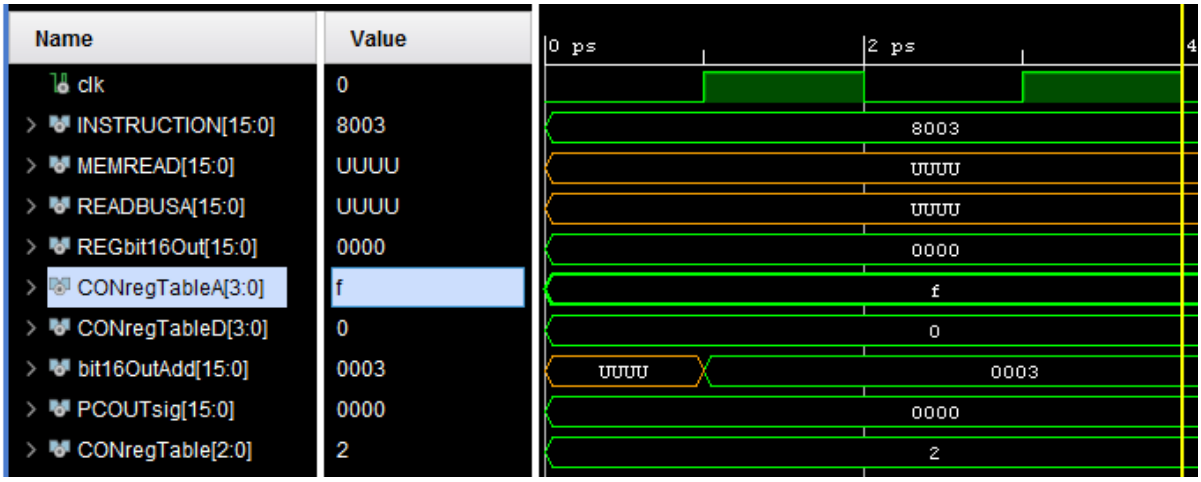

Figure 10: Jump&Link Instruction Jumping Ahead Three Bits

RETURN: with opcode 1001b, this instruction moves the address saved in register rP and pushes it to the program counter. To do this, register rP is output to the ADD unit where 0 is then added and shifted onto the program counter. The control unit shuts the ALU off so that the value is not altered in any way. A control signal going into the ADD unit identifies this as the one time that the data should be unmodified and added with 0. Figure 11 Shows the value specified in the WRITEBUS, which symbolizes register F's contents, being added to the current PC value.

Figure 11: Return Writing the WRITEBUS' Value to the PC

BRANCH IF EQUAL: with opcode 1010b, this instruction evaluates two register values and determines if they are equal by subtracting the two and sending the appropriate signal to the control unit. If the difference between the two register values is zero then the JUMP&LINK sequence will be executed. Figure 12 shows that READBUSA and READBUSB are equal and the branch will be taken.

The PC value will increment by the four bits specified in the instruction and the branchBit signal from the ALU going high.

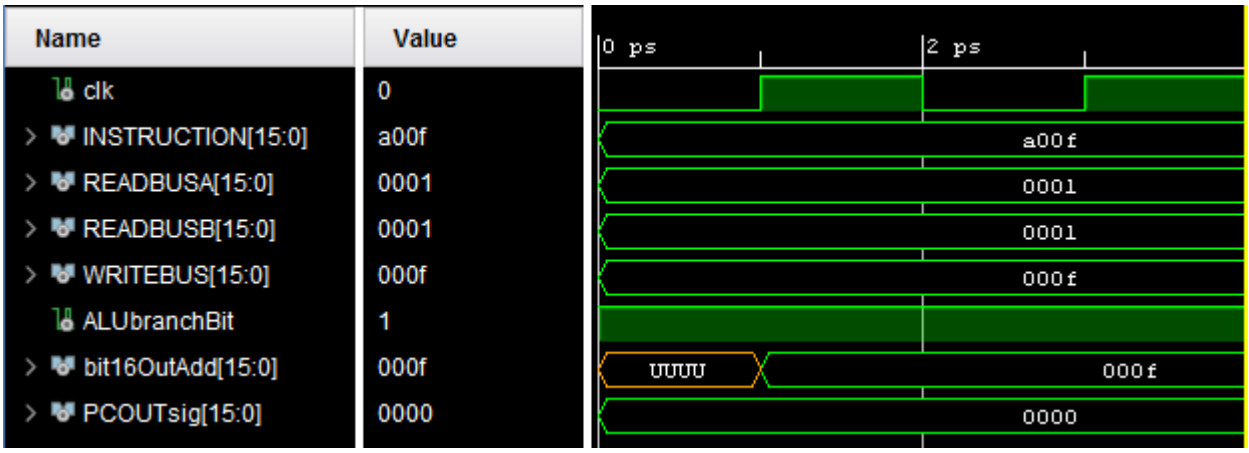

Figure 12: Branch if Equal Instruction Taking the Branch and Incrementing the PC

If the two busses are not equal then the branch bit will remain low and the PC will be incremented by one. This can be seen in figure 13.

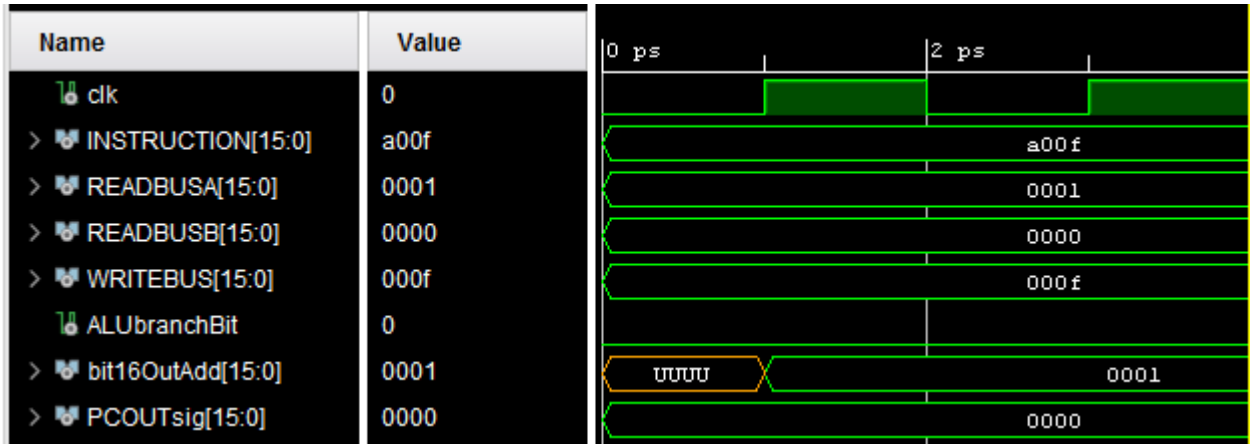

Figure 13: Branch if Equal Instruction Not Taking the Branch and Incrementing the PC by One

BRANCH IF NOT EQUAL: with opcode 1011b, this instruction evaluates two register values and determines if they are equal by subtracting the two and sending the appropriate signal to the control unit. If the difference between the two register values is zero then the JUMP&LINK sequence will not be executed. Figure 14 shows that the two READBUS values are equal and the branch is not taken.

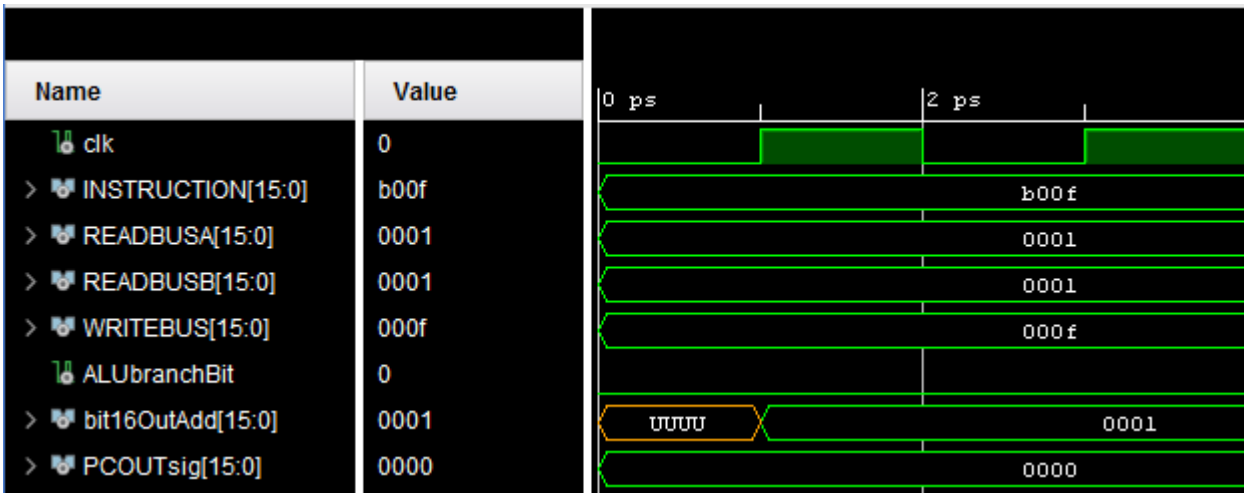

Figure 12: Branch if Not Equal Instruction Not Taking the Branch and Incrementing the PC by One

If the two busses are not equal the branch is then taken. This can be seen below in figure 13.

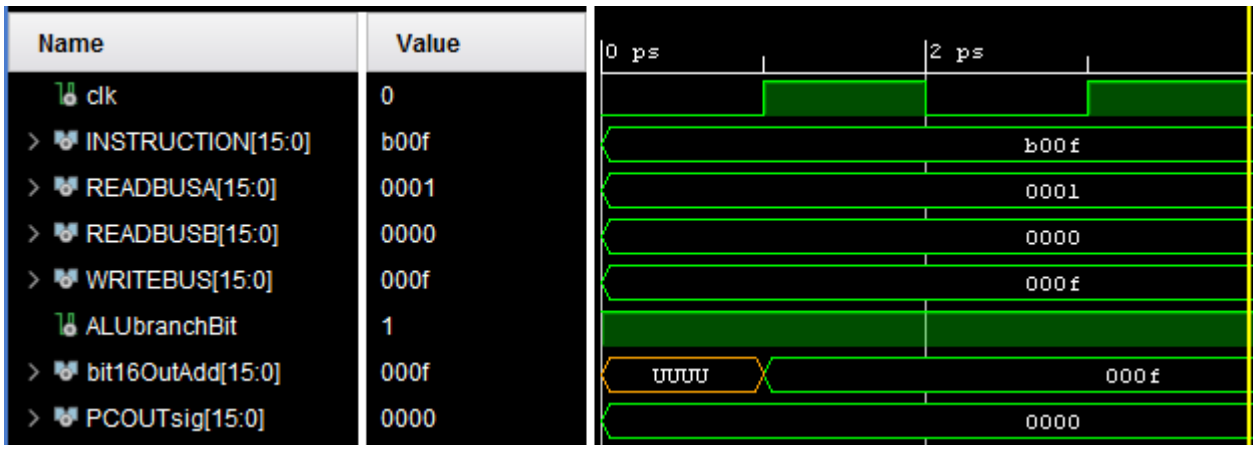

Figure 13: Branch if Not Equal Instruction Taking the Branch and Incrementing the PC by 0x000F

OR: with opcode 1100b, this instruction performs a bitwise OR on two registers' values and saves the result the specified address. This operation uses all the same modules as ADD instruction, but the control unit instructs the ALU to perform the bitwise operation. Figure 14 shows the result of the two busses being bitwise OR'd.

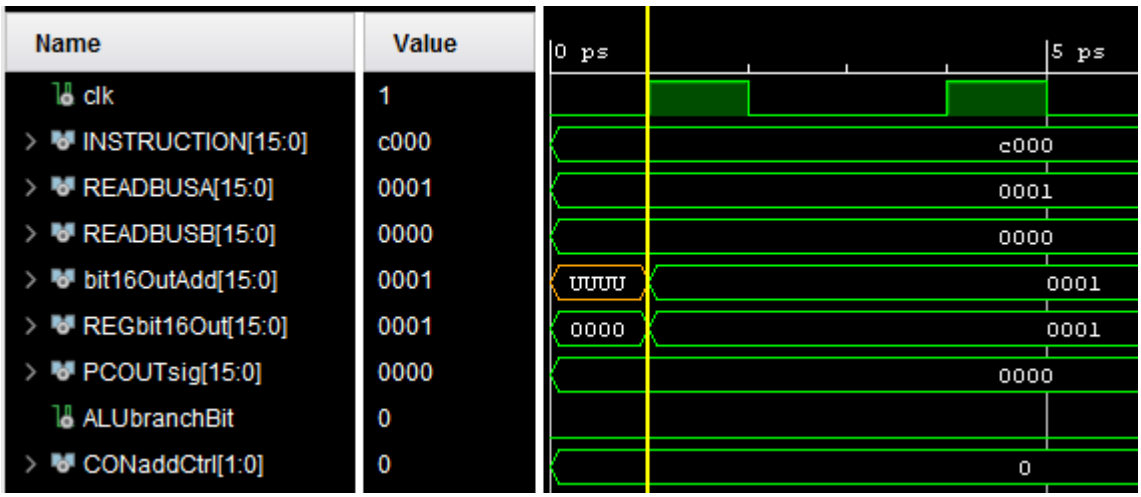

Figure 14: Or Instruction Operating on the Two Bus Values and Outputting the Resulting 0x0001

AND: with opcode 1101b, this instruction uses the same components as OR, but the ALU is instructed to perform the bitwise AND operation on the two register operands. Figure 15 shows the output of REGbit16Out being 0x0001 when 0x0003 and 0x0001 being AND'd together.

Figure 14: And Instruction Operating on the Two Bus Values and Outputting the Resulting 0x0001

BRANCH LESS THAN: with opcode 1110b, this instruction behaves the exact same as the BRANCH IF EQUAL command. If the reference register is smaller than the second register then the result will be zero or negative when the two are subtracted. The control unit is informed by the ALU and the branch takes place. Figure 15 shows that READBUSA is less than READBUSB and the branch is taken as a result.

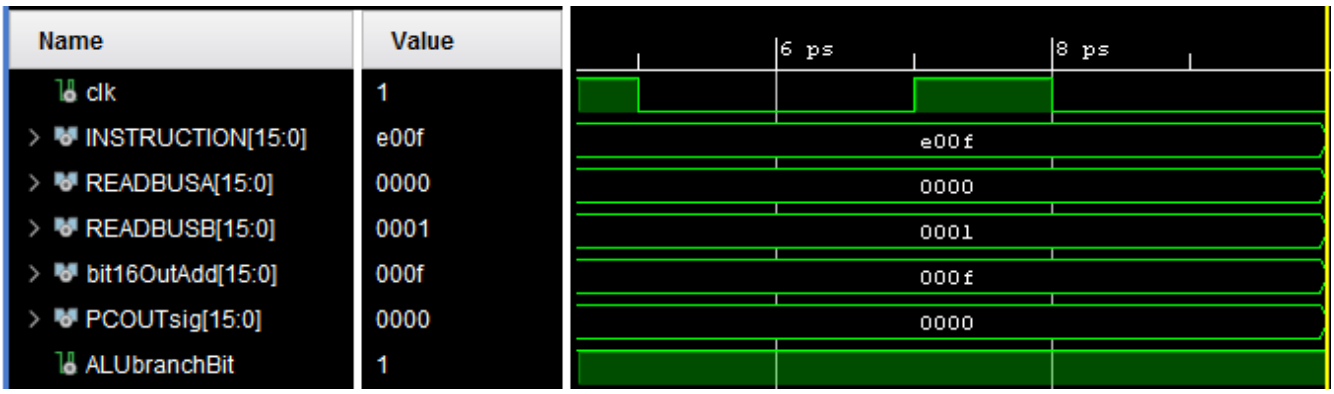

Figure 15: Branch Less Than Taking the Branch When A is less than B

BRANCH GREATER THAN: with opcode 1111b, this instructions is the exact same as BRANCH IF LESS THAN, but it only occurs if the ALU provides the control unit a non-activated signal line. Figure 16 shows the result when READBUSA is larger than READBUSB.

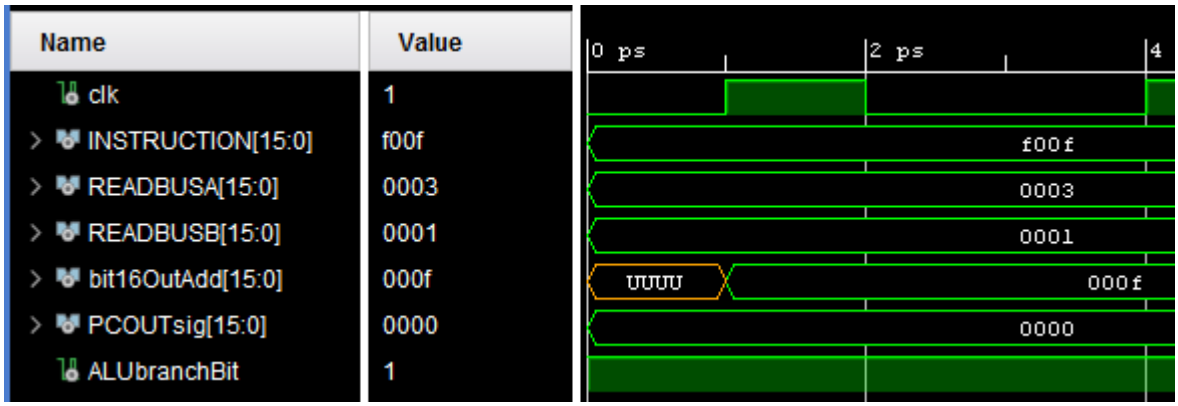

Figure 16: Branch Greater Than Taking the Branch When A is Greater Than B

To simulate a small program, the output of the register/memory mux is written to WRITEBUS so that it can be evaluated by the other components in the datapath. In this mock program, two values are added together, the bitwise and'd, then finally, if the result is not equal to 0x0002, a branch will take place. The first two values to be added are 0x0001 and 0x0001, then these values are and'd with 0x0003. The result should be 0x0001, which is then compared to 0x0002. The result should be that they are equal and the branch is not taken. The final PC value should finalize at 3. This simulation can be seen in figure 17.

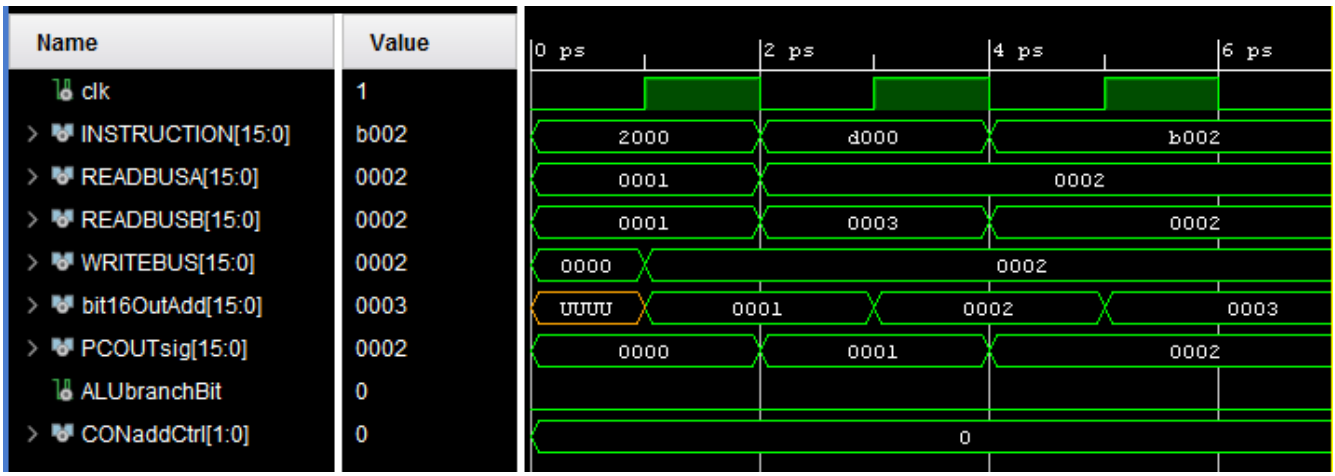

Figure 17: Three Instruction Program Test Pass

With everything tested, the final RTL model can be evaluated. Because no values are being written directly to a register file, Sign Extension 2 is not being used. The result can be seen below in figure 18.

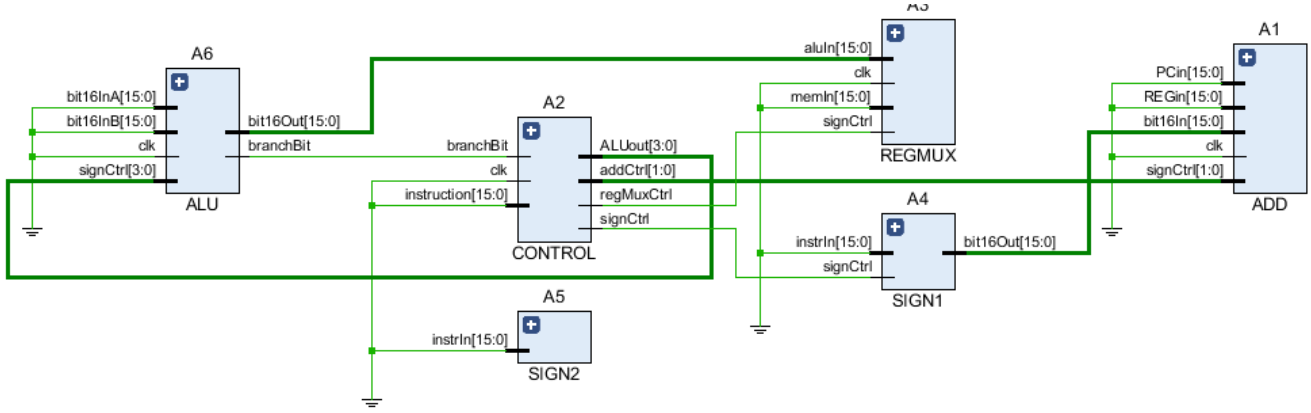

Figure 18: RTL Model of the Verified CPU Model

Top Level Code:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.ALL;
use ieee.std_logic_unsigned.all;


entity main is

end main;

architecture Behavioral of main is
signal clk : std_logic;
signal READBUSA : STD_LOGIC_VECTOR(15 downto 0);
signal READBUSB : STD_LOGIC_VECTOR(15 downto 0);

signal MEMREAD : STD_LOGIC_VECTOR(15 downto 0);

signal INSTRUCTION : STD_LOGIC_VECTOR(15 downto 0);

signal WRITEBUS : STD_LOGIC_VECTOR(15 downto 0);


--Add
signal bit16OutAdd : STD_LOGIC_VECTOR(15 downto 0);

--Control
signal CONregMuxCtrl : STD_LOGIC;
signal CONregTableA : STD_LOGIC_VECTOR(3 downto 0);
signal CONregTableB : STD_LOGIC_VECTOR(3 downto 0);
signal CONregTableD : STD_LOGIC_VECTOR(3 downto 0);
signal CONALUout : STD_LOGIC_VECTOR(3 downto 0);
signal CONregTable :  STD_LOGIC_VECTOR(2 downto 0);
signal CONsignCtrl : STD_LOGIC;
signal CONdataMem : STD_LOGIC_VECTOR(1 downto 0);
signal CONaddCtrl : STD_LOGIC_VECTOR(1 downto 0);

--RegMux
signal REGbit16Out : STD_LOGIC_VECTOR(15 downto 0);

--Sign1
signal Sign1bit16Out : STD_LOGIC_VECTOR(15 downto 0);

--Sign2
signal Sign2bit16Out : STD_LOGIC_VECTOR(15 downto 0);

--PC
signal PCOUTsig : STD_LOGIC_VECTOR(15 downto 0);

--ALU
signal ALUbit16Out : STD_LOGIC_VECTOR(15 downto 0);
signal ALUbranchBit : STD_LOGIC;
```

```
component ADD
    port(
        clk : in std_logic;
        bit16In : in STD_LOGIC_VECTOR(15 downto 0);
        PCin : in STD_LOGIC_VECTOR(15 downto 0);
        REGin : in STD_LOGIC_VECTOR(15 downto 0);
        bit16Out : out STD_LOGIC_VECTOR(15 downto 0);
        signCtrl : in STD_LOGIC_VECTOR(1 downto 0));
end component;

component CONTROL
    port(
        clk : in STD_LOGIC;
        instruction : in STD_LOGIC_VECTOR(15 downto 0);
        branchBit : in STD_LOGIC;
        regMuxCtrl : out STD_LOGIC;
        regTableA : out STD_LOGIC_VECTOR(3 downto 0);
        regTableB : out STD_LOGIC_VECTOR(3 downto 0);
        regTableD : out STD_LOGIC_VECTOR(3 downto 0);
        ALUout : out STD_LOGIC_VECTOR(3 downto 0);
        regTable : out STD_LOGIC_VECTOR(2 downto 0);
        signCtrl: out STD_LOGIC;
        dataMem : out STD_LOGIC_VECTOR(1 downto 0);
        addCtrl: out STD_LOGIC_VECTOR(1 downto 0));
end component;

component REGMUX
port(
        signCtrl : in STD_LOGIC;
        bit16Out : out STD_LOGIC_VECTOR(15 downto 0);
        aluIn : in STD_LOGIC_VECTOR(15 downto 0);
        memIn : in STD_LOGIC_VECTOR(15 downto 0));
end component;

component SIGN1
    port(
    instrIn: in STD_LOGIC_VECTOR(15 downto 0);
    bit16Out: out STD_LOGIC_VECTOR(15 downto 0);
    signCtrl: in STD_LOGIC);
end component;

component SIGN2
 port(
   instrIn : in STD_LOGIC_VECTOR(15 downto 0);
   bit16Out: out STD_LOGIC_VECTOR(15 downto 0));
end component;



component ALU
port(
```

```
        bit16InA : in STD_LOGIC_VECTOR(15 downto 0);
        bit16InB : in STD_LOGIC_VECTOR(15 downto 0);
        bit16Out : out STD_LOGIC_VECTOR(15 downto 0); -- this goes to either datamem or regtable
        branchBit : out STD_LOGIC;
        signCtrl : in STD_LOGIC_VECTOR(3 downto 0));
end component;

begin
A1 : ADD port map(clk => clk,bit16In => Sign1bit16Out, PCin => PCOUTsig, REGin => WRITEBUS,
bit16Out => bit16OutAdd, signCtrl => CONaddCtrl);


--dear lord
A2: CONTROL port map(clk => clk,instruction => INSTRUCTION, branchBit => ALUbranchBit, regMuxCtrl
=> CONregMuxCtrl,
regTableA => CONregTableA, regTableB => CONregTableB, regTableD => CONregTableD, ALUout =>
CONALUout,
regTable => CONregTable, signCtrl => CONsignCtrl, dataMem => CONdataMem, addCtrl => CONaddCtrl);


A3: REGMUX port map(signCtrl => CONregMuxCtrl, bit16Out => REGbit16Out, aluIn => ALUbit16Out,
memIn => MEMREAD);

A4: SIGN1 port map(instrIn => INSTRUCTION, bit16Out => Sign1bit16Out, signCtrl => CONsignCtrl);

A5: SIGN2 port map(instrIn => INSTRUCTION, bit16Out => Sign2bit16Out);

A6: ALU port map(bit16InA => READBUSA, bit16InB => READBUSA, bit16Out => ALUbit16Out, branchBit
=> ALUbranchBit, signCtrl => CONALUout);


end Behavioral;
```

Top Level TestBench:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.ALL;
use ieee.std_logic_unsigned.all;



entity mainBench is
--  Port ( );
end mainBench;

architecture Behavioral of mainBench is
signal clk : std_logic;
signal READBUSA : STD_LOGIC_VECTOR(15 downto 0);
signal READBUSB : STD_LOGIC_VECTOR(15 downto 0);

signal MEMREAD : STD_LOGIC_VECTOR(15 downto 0);

signal INSTRUCTION : STD_LOGIC_VECTOR(15 downto 0);

signal WRITEBUS : STD_LOGIC_VECTOR(15 downto 0);



--Add
signal bit16OutAdd : STD_LOGIC_VECTOR(15 downto 0);

--Control
signal CONregMuxCtrl : STD_LOGIC;
signal CONregTableA : STD_LOGIC_VECTOR(3 downto 0);
signal CONregTableB : STD_LOGIC_VECTOR(3 downto 0);
signal CONregTableD : STD_LOGIC_VECTOR(3 downto 0);
signal CONALUout : STD_LOGIC_VECTOR(3 downto 0);
signal CONregTable :  STD_LOGIC_VECTOR(2 downto 0);
signal CONsignCtrl : STD_LOGIC;
signal CONdataMem : STD_LOGIC_VECTOR(1 downto 0);
signal CONaddCtrl : STD_LOGIC_VECTOR(1 downto 0);

--RegMux
signal REGbit16Out : STD_LOGIC_VECTOR(15 downto 0);

--Sign1
signal Sign1bit16Out : STD_LOGIC_VECTOR(15 downto 0);

--Sign2
signal Sign2bit16Out : STD_LOGIC_VECTOR(15 downto 0);

--PC
signal PCOUTsig : STD_LOGIC_VECTOR(15 downto 0);
```

```
--ALU
signal ALUbit16Out : STD_LOGIC_VECTOR(15 downto 0);
signal ALUbranchBit : STD_LOGIC;

component ADD
    port(
        clk : in std_logic;
        bit16In : in STD_LOGIC_VECTOR(15 downto 0);
        PCin : in STD_LOGIC_VECTOR(15 downto 0);
        REGin : in STD_LOGIC_VECTOR(15 downto 0);
        bit16Out : out STD_LOGIC_VECTOR(15 downto 0);
        signCtrl : in STD_LOGIC_VECTOR(1 downto 0));
end component;

component CONTROL
    port(
        clk : in STD_LOGIC;
        instruction : in STD_LOGIC_VECTOR(15 downto 0);
        branchBit : in STD_LOGIC;
        regMuxCtrl : out STD_LOGIC;
        regTableA : out STD_LOGIC_VECTOR(3 downto 0);
        regTableB : out STD_LOGIC_VECTOR(3 downto 0);
        regTableD : out STD_LOGIC_VECTOR(3 downto 0);
        ALUout : out STD_LOGIC_VECTOR(3 downto 0);
        regTable : out STD_LOGIC_VECTOR(2 downto 0);
        signCtrl: out STD_LOGIC;
        dataMem : out STD_LOGIC_VECTOR(1 downto 0);
        addCtrl: out STD_LOGIC_VECTOR(1 downto 0));
end component;

component REGMUX
port(
        clk : in STD_LOGIC;
        signCtrl : in STD_LOGIC;
        bit16Out : out STD_LOGIC_VECTOR(15 downto 0);
        aluIn : in STD_LOGIC_VECTOR(15 downto 0);
        memIn : in STD_LOGIC_VECTOR(15 downto 0));
end component;

component SIGN1
    port(
    instrIn : in STD_LOGIC_VECTOR(15 downto 0);
    bit16Out: out STD_LOGIC_VECTOR(15 downto 0);
    signCtrl: in STD_LOGIC);
end component;

component SIGN2
 port(
   instrIn : in STD_LOGIC_VECTOR(15 downto 0);
   bit16Out: out STD_LOGIC_VECTOR(15 downto 0));
end component;
```

```
component ALU
port(
	clk : in STD_LOGIC;
	bit16InA : in STD_LOGIC_VECTOR(15 downto 0);
	bit16InB : in STD_LOGIC_VECTOR(15 downto 0);
	bit16Out : out STD_LOGIC_VECTOR(15 downto 0); -- this goes to either datamem or regtable
	branchBit : out STD_LOGIC;
	signCtrl : in STD_LOGIC_VECTOR(3 downto 0));
end component;

begin


A1 : ADD port map(clk => clk,bit16In => Sign1bit16Out, PCin => PCOUTsig, REGin => WRITEBUS,
bit16Out => bit16OutAdd, signCtrl => CONaddCtrl);


--dear lord
A2: CONTROL port map(clk => clk, instruction => INSTRUCTION, branchBit => ALUbranchBit, regMuxCtrl
=> CONregMuxCtrl,
regTableA => CONregTableA, regTableB => CONregTableB, regTableD => CONregTableD, ALUout =>
CONALUout,
regTable => CONregTable, signCtrl => CONsignCtrl, dataMem => CONdataMem, addCtrl => CONaddCtrl);


A3: REGMUX port map(clk => clk,signCtrl => CONregMuxCtrl, bit16Out => REGbit16Out, aluIn =>
ALUbit16Out, memIn => MEMREAD);

A4: SIGN1 port map(instrIn => INSTRUCTION, bit16Out => Sign1bit16Out, signCtrl => CONsignCtrl);

A5: SIGN2 port map(instrIn => INSTRUCTION, bit16Out => Sign2bit16Out);

A6: ALU port map(clk => clk,bit16InA => READBUSA, bit16InB => READBUSB, bit16Out => ALUbit16Out,
branchBit => ALUbranchBit, signCtrl => CONALUout);

process begin

------HALT
clk <= '0';
INSTRUCTION <= "0000000000000000"; --halt
PCOUTsig <= "1111111111111111"; --PC value
wait for 1ps;
clk <= '1';
wait for 1ps;
assert bit16OutAdd =  "0000000000000000"
report "ADD CNTRL HALT IS BAD"
severity ERROR;

--JUMP
clk <= '0';
PCOUTsig <= "0000000000000000"; --PC value
INSTRUCTION <= "0001000000000011"; --jump
```

```
wait for 1ps;
clk <= '1';
wait for 1ps;

assert bit16OutAdd = "0000000000000011"
report "ADD CNTRL JUMP IS BAD"
severity ERROR;

--ADD
clk <= '0';
PCOUTsig <= "0000000000000000"; --PC value
READBUSA <= "0000000000000001";
READBUSB <= "0000000000000001";
INSTRUCTION <= "0010000000000000";
wait for 1ps;
clk <= '1';
wait for 1ps;
assert ALUbit16Out = "0000000000000010"
report "ADD CNTRL ALU IS BAD"
severity ERROR;
wait for 1ps;
assert ALUbit16Out = REGbit16Out
report "ADD CNTRL REGMUX IS BAD"
severity ERROR;

--SUB
clk <= '0';
PCOUTsig <= "0000000000000000"; --PC value
READBUSA <= "0000000000000001";
READBUSB <= "0000000000000001";
INSTRUCTION <= "0011000000000000";
wait for 1ps;
clk <= '1';
wait for 1ps;
assert ALUbit16Out = "0000000000000000"
report "SUB CNTRL ALU IS BAD"
severity ERROR;
wait for 1ps;
assert ALUbit16Out = REGbit16Out
report "SUB CNTRL REGMUX IS BAD"
severity ERROR;

--LOAD
INSTRUCTION <= "0100000100000011";
PCOUTsig <= "0000000000000000"; --PC value
MEMREAD <= "0000000000000001";
clk <= '0';
wait for 1ps;
clk <= '1';
wait for 1ps;

assert CONregTableD = INSTRUCTION(11 downto 8)
report "LOAD CNTRL DEST ADDR IS BAD"
```

```
severity ERROR;

assert REGbit16Out = "0000000000000001"
report "LOAD CNTRL REGMUX IS BAD"
severity ERROR;

--LOADC
INSTRUCTION <= "0101000000000000";
PCOUTsig <= "0000000000000000"; --PC value
clk <= '0';
wait for 2ps;
clk <= '1';
wait for 2ps;
assert CONregTable = "001"
report "REG CNTRL IS BAD"
severity ERROR;

assert CONregTableD = INSTRUCTION(11 downto 8)
report "LOADC CNTRL DEST ADDR IS BAD"
severity ERROR;

--MOV
INSTRUCTION <= "0110001100010000";
READBUSA <= "0000000000000001";--input values
PCOUTsig <= "0000000000000000"; --PC value
clk <= '0';
wait for 1ps;
clk <= '1';
wait for 1ps;

assert CONregTableD = "0011"
report "MOV CNTRL DEST ADDR IS BAD"
severity ERROR;

assert CONregTableA = "0001"
report "MOV CNTRL SRC ADDR IS BAD"
severity ERROR;

assert REGbit16Out = "0000000000000001"
report "MOV CNTRL REGMUX IS BAD"
severity ERROR;

--STORE
INSTRUCTION <= "0111000000000001";
PCOUTsig <= "0000000000000000"; --PC value
READBUSA <= "0000000000000001";--input values
clk <= '0';
wait for 1ps;
clk <= '1';
wait for 1ps;
assert ALUbit16Out = "0000000000000001"
report "STORE CNTRL ALU IS BAD"
severity ERROR;
```

```
assert REGbit16Out = "0000000000000001"
report "STORE CNTRL REGMUX IS BAD"
severity ERROR;


--JUMP AND LINK
clk <= '0';
PCOUTsig <= "0000000000000000"; --PC value
INSTRUCTION <= "1000000000000011"; --JUMP and LINK
wait for 1ps;
clk <= '1';
wait for 1ps;

assert bit16OutAdd = "0000000000000011"
report "ADD CNTRL JUMP IS BAD"
severity ERROR;

--RETURN
clk <= '0';
PCOUTsig <= "0000000000000000"; --PC value
INSTRUCTION <= "1001000000000000"; --RETURN
WRITEBUS <= "1111111111111111";
wait for 1ps;
clk <= '1';
wait for 1ps;

assert bit16OutAdd = "1111111111111111"
report "ADD CNTRL RETURN IS BAD"
severity ERROR;

--Branch if Equal --- branch bit not being evaluated
clk <= '0';
PCOUTsig <= "0000000000000000"; --PC value
INSTRUCTION <= "1010000000001111"; --RETURN
WRITEBUS <= "0000000000001111";

READBUSA <= "0000000000000001";
READBUSB <= "0000000000000001";

wait for 1ps;
clk <= '1';
wait for 1ps;

assert bit16OutAdd = "0000000000001111"
report "ADD CNTRL RETURN IS BAD"
severity ERROR;

PCOUTsig <= "0000000000000000"; --PC value
INSTRUCTION <= "1010000000001111"; --RETURN
WRITEBUS <= "0000000000001111";
clk <= '0';
READBUSA <= "0000000000000001";
```

```
READBUSB <= "0000000000000000";

wait for 1ps;
clk <= '1';
wait for 1ps;

assert bit16OutAdd = "0000000000010000"
report "ADD CNTRL RETURN IS BAD"
severity ERROR;

--branch if not equal
clk <= '0';
PCOUTsig <= "0000000000000000"; --PC value
INSTRUCTION <= "1011000000001111"; --RETURN
WRITEBUS <= "0000000000001111";

READBUSA <= "0000000000000001";
READBUSB <= "0000000000000001";

wait for 1ps;
clk <= '1';
wait for 1ps;

assert bit16OutAdd = "0000000000000001"
report "ADD CNTRL RETURN IS BAD"
severity ERROR;

wait for 1ps;
clk <= '0';
READBUSA <= "0000000000000001";
READBUSB <= "0000000000000000";

wait for 1ps;
clk <= '1';
wait for 1ps;

assert bit16OutAdd = "0000000000001111"
report "ADD CNTRL RETURN IS BAD"
severity ERROR;

--OR
clk <= '0';
PCOUTsig <= "0000000000000000"; --PC value
INSTRUCTION <= "1100000000000000"; --RETURN
READBUSA <= "0000000000000001";
READBUSB <= "0000000000000000";
wait for 1ps;
clk <= '1';
wait for 1ps;
clk <= '0';
wait for 1ps;
assert REGbit16Out = "0000000000000001"
report "OR CNTRL RETURN IS BAD"
```

```
severity ERROR;

----AND
clk <= '0';
PCOUTsig <= "0000000000000000"; --PC value
INSTRUCTION <= "1101000000000000"; --RETURN
READBUSA <= "0000000000000011";
READBUSB <= "0000000000000001";
wait for 1ps;
clk <= '1';
wait for 1ps;
clk <= '0';
wait for 1ps;
assert REGbit16Out = "0000000000000001"
report "OR CNTRL RETURN IS BAD"
severity ERROR;


--BRANCH IF LESS THAN
clk <= '0';
PCOUTsig <= "0000000000000000"; --PC value
INSTRUCTION <= "1110000000001111"; --RETURN
READBUSA <= "0000000000000000";
READBUSB <= "0000000000000001";

wait for 1ps;
clk <= '1';
wait for 1ps;
clk <= '0';
wait for 1ps;

assert bit16OutAdd = "0000000000001111"
report "ADD CNTRL RETURN IS BAD"
severity ERROR;

--BRANCH GREATER THAN
clk <= '0';
PCOUTsig <= "0000000000000000"; --PC value
INSTRUCTION <= "1111000000001111"; --RETURN
READBUSA <= "0000000000000011";
READBUSB <= "0000000000000001";

wait for 1ps;
clk <= '1';
wait for 1ps;
clk <= '0';
wait for 1ps;

assert bit16OutAdd = "0000000000001111"
report "ADD CNTRL RETURN IS BAD"
severity ERROR;

---CHANGED REGMUX OUTPUT TO WRITE BUS
```

```
clk <= '0';
PCOUTsig <= "0000000000000000"; --PC value
INSTRUCTION <= "0010000000000000"; --add
READBUSA <= "0000000000000001";
READBUSB <= "0000000000000001";
wait for 1ps;
clk <= '1';

wait for 1 ps;

clk <= '0';
PCOUTsig <= bit16OutAdd;
READBUSA <= WRITEBUS;
READBUSB <= "0000000000000011";
INSTRUCTION <= "1101000000000000"; --and
wait for 1ps;
clk <= '1';

wait for 1ps;

clk <= '0';
PCOUTsig <= bit16OutAdd;
READBUSA <= WRITEBUS;
READBUSB <= "0000000000000010";
INSTRUCTION <= "1011000000000010"; --BNEQ
wait for 1ps;
clk <= '1';
wait for 1ps;
clk <= '0';
wait for 1ps;
clk <= '1';
wait for 1ps;
assert ALUbranchBit = '0'
report "FAILED"
severity ERROR;


end process;
end Behavioral;
```

Control:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.ALL;
use ieee.std_logic_unsigned.all;

entity CONTROL is
    port(
        clk : in STD_LOGIC;
        instruction : in STD_LOGIC_VECTOR(15 downto 0);
        branchBit : in STD_LOGIC;
        regMuxCtrl : out STD_LOGIC;
        regTableA : out STD_LOGIC_VECTOR(3 downto 0);
        regTableB : out STD_LOGIC_VECTOR(3 downto 0);
        regTableD : out STD_LOGIC_VECTOR(3 downto 0);
        ALUout : out STD_LOGIC_VECTOR(3 downto 0);
        regTable : out STD_LOGIC_VECTOR(2 downto 0);

        signCtrl: out STD_LOGIC;
        dataMem : out STD_LOGIC_VECTOR(1 downto 0);
        addCtrl: out STD_LOGIC_VECTOR(1 downto 0));


end CONTROL;

architecture Behavioral of CONTROL is
signal regTableAsig : STD_LOGIC_VECTOR(3 downto 0) := "0000";
signal regTableBsig : STD_LOGIC_VECTOR(3 downto 0) := "0000";
signal regTableDsig : STD_LOGIC_VECTOR(3 downto 0) := "0000";
signal ALUOutsig : STD_LOGIC_VECTOR(3 downto 0) := "0000";
signal regTablesig : STD_LOGIC_VECTOR(2 downto 0) := "000";
signal signCtrlsig: STD_LOGIC := '0';
signal regMuxCtrlsig: STD_LOGIC := '0';
signal dataMemsig: STD_LOGIC_VECTOR(1 downto 0) := "00";
signal addCtrlsig: STD_LOGIC_VECTOR(1 downto 0) := "00";

begin

process(instruction,branchBit) begin
    if(instruction(15 downto 12) = "0000") then --halt
        addCtrlsig <= "11";
    elsif(instruction(15 downto 12) = "0001")then --jump
        signCtrlsig <= '1';
        addCtrlsig <= "01"; --sign extended

    elsif(instruction(15 downto 12) = "0010")then --add
        regTablesig <= "000";
        regTableDsig <= instruction(11 downto 8);
        regTableAsig <= instruction(7 downto 4);
        regTableBsig <= instruction(3 downto 0);
        ALUOutsig <= "0010";
        dataMemsig <= "11"; -- do nothing
        addCtrlsig <= "00"; --PC + 1
```

```
    regMuxCtrlsig <= '0'; --ALU TO REG

elsif(instruction(15 downto 12) = "0011")then --sub
    regTablesig <= "000";
    regTableDsig <= instruction(11 downto 8);
    regTableAsig <= instruction(7 downto 4);
    regTableBsig <= instruction(3 downto 0);
    ALUOutsig <= "0011";
    dataMemsig <= "11"; --do nothing
    addCtrlsig <= "00"; --PC + 1
    regMuxCtrlsig <= '0'; --ALU TO REG

elsif(instruction(15 downto 12) = "0100")then --load
    regTablesig <= "000"; --read so regtable looks at input
    regTableDsig <= instruction(11 downto 8); --load address
    ALUOutsig <= "0000"; --do nothing
    dataMemsig <= "10"; --write pcIn address
    addCtrlsig <= "00"; --PC + 1
    regMuxCtrlsig <= '1'; --MEM TO REG

elsif(instruction(15 downto 12) = "0101")then --load C
    regTablesig <= "001"; --read sign extend 2
    regTableDsig <= instruction(11 downto 8); --load address
    ALUOutsig <= "0000"; --do nothing
    dataMemsig <= "11"; --do nothing
    addCtrlsig <= "00"; --PC + 1

elsif(instruction(15 downto 12) = "0110")then --MOV
    regTablesig <= "000"; --rtype
    regTableDsig <= instruction(11 downto 8); --destination
    regTableAsig <= instruction(7 downto 4); --value being moved
    ALUOutsig <= "0110"; --pass values from reg to reg mux
    dataMemsig <= "11"; --default
    addCtrlsig <= "00"; --PC + 1
    regMuxCtrlsig <= '0'; --ALU TO REG

elsif(instruction(15 downto 12) = "0111")then --STORE
    regTablesig <= "000"; --rtype
    regTableAsig <= instruction(7 downto 4); --value being stored
    ALUOutsig <= "0110"; --pass values from reg to reg mux
    dataMemsig <= "01"; --write to mem
    addCtrlsig <= "00"; --PC + 1
    regMuxCtrlsig <= '0'; --ALU TO REG

elsif(instruction(15 downto 12) = "1000")then --JUMP AND LINK
    regTablesig <= "010"; --save PC
    regTableAsig <= "1111"; --value being stored
    ALUOutsig <= "0000"; --pass values from reg to reg mux
    dataMemsig <= "11"; --do nothing
    signCtrlsig <= '1'; --put 12 bit offset to add
    addCtrlsig <= "01"; --sign extended
    --regMuxCtrlsig <= '0'; --ALU TO REG
```

```
elsif(instruction(15 downto 12) = "1001")then --RETURN
    regTablesig <= "000"; --do nothing
    dataMemsig <= "11"; --do nothing
    addCtrlsig <= "10"; -- add puts the register into PC
    ALUOutsig <= "0000"; --do nothing
    dataMemsig <= "11"; --do nothing
    regMuxCtrlsig <= '0'; --ALU TO REG

 elsif(instruction(15 downto 12) = "1010")then --BRANCH IF EQUAL
    regTablesig <= "000"; --do nothing
    dataMemsig <= "11"; --do nothing
    ALUOutsig <= "1010"; --branch if equal
    dataMemsig <= "11"; --do nothing
    regMuxCtrlsig <= '0'; --ALU TO REG
    signCtrlsig <= '0'; --put 8 bit offset to add

    if(branchBit = '1')then
       addCtrlsig <= "10"; -- add puts the sign extend into PC
    else
       addCtrlsig <= "00"; -- PC + 1
    end if;

elsif(instruction(15 downto 12) = "1011")then --BRANCH IF NOT EQUAL
    regTablesig <= "000"; --do nothing
    dataMemsig <= "11"; --do nothing
    signCtrlsig <= '0'; --put 8 bit offset to add
    ALUOutsig <= "1011"; --branch if equal
    dataMemsig <= "11"; --do nothing
    regMuxCtrlsig <= '0'; --ALU TO REG

    if(branchBit = '1')then
       addCtrlsig <= "01"; -- add puts the sign extend into PC
    else
       addCtrlsig <= "00"; -- PC + 1
    end if;

elsif(instruction(15 downto 12) = "1100")then --OR
    regTablesig <= "000";
    regTableDsig <= instruction(11 downto 8);
    regTableAsig <= instruction(7 downto 4);
    regTableBsig <= instruction(3 downto 0);
    ALUOutsig <= "1100";
    dataMemsig <= "11"; -- do nothing
    addCtrlsig <= "00"; --PC + 1
    regMuxCtrlsig <= '0'; --ALU TO REG

 elsif(instruction(15 downto 12) = "1101")then --AND
   regTablesig <= "000";
   regTableDsig <= instruction(11 downto 8);
   regTableAsig <= instruction(7 downto 4);
   regTableBsig <= instruction(3 downto 0);
   ALUOutsig <= "1101";
   dataMemsig <= "11"; -- do nothing
```

```
      addCtrlsig <= "00"; --PC + 1
      regMuxCtrlsig <= '0'; --ALU TO REG

   elsif(instruction(15 downto 12) = "1110")then --BRANCH IF LESS THAN
         regTablesig <= "000"; --do nothing
         dataMemsig <= "11"; --do nothing
         signCtrlsig <= '0'; --put 8 bit offset to add
         ALUOutsig <= "1110"; --branch if equal
         dataMemsig <= "11"; --do nothing

         if(branchBit = '1')then
            addCtrlsig <= "01"; -- add puts the sign extend into PC
         else
            addCtrlsig <= "00"; -- PC + 1
         end if;

      elsif(instruction(15 downto 12) = "1111")then --BRANCH IF greater THAN
            regTablesig <= "000"; --do nothing
            dataMemsig <= "11"; --do nothing
            signCtrlsig <= '0'; --put 8 bit offset to add
            ALUOutsig <= "1111"; --branch if equal
            dataMemsig <= "11"; --do nothing

            if(branchBit = '1')then
               addCtrlsig <= "01"; -- add puts the sign extend into PC
            else
               addCtrlsig <= "00"; -- PC + 1
            end if;

   end if;
end process;

regTableA <= regTableAsig;
regTableB <= regTableBsig;
regTableD <= regTableDsig;
ALUout <= ALUOutsig;
regTable <= regTablesig;
signCtrl <= signCtrlsig;
dataMem <= dataMemsig;
addCtrl <= addCtrlsig;
regMuxCtrl <= regMuxCtrlsig;


end Behavioral;
```

Add Unit:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.ALL;
use ieee.std_logic_unsigned.all;

entity ADD is
   port(
      clk : in STD_LOGIC;
      bit16In : in STD_LOGIC_VECTOR(15 downto 0);
      PCin : in STD_LOGIC_VECTOR(15 downto 0);
      REGin : in STD_LOGIC_VECTOR(15 downto 0);
      bit16Out : out STD_LOGIC_VECTOR(15 downto 0);
      signCtrl : in STD_LOGIC_VECTOR(1 downto 0));
end ADD;

architecture Behavioral of ADD is
signal bit16OutSig : STD_LOGIC_VECTOR(15 downto 0);
begin

CTRL: process(clk,signCtrl,REGin,PCin)
   begin
   if rising_edge(clk) then
      if(signCtrl = "00") then
         bit16OutSig <= PCin + "0000000000000001";--PC = PC+1
      elsif(signCtrl = "01") then
         bit16OutSig <= PCin + bit16In;--from signextend1

      elsif(signCtrl = "10") then
         bit16OutSig <= REGin;--Branch value
      else
         bit16OutSig <= "0000000000000000"; --HALT
      end if;

   end if;
   end process;
   bit16Out <= bit16OutSig;

end Behavioral;
```

ALU:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.ALL;
use ieee.std_logic_unsigned.all;

entity ALU is
    port(
        clk: in std_logic;
        bit16InA : in STD_LOGIC_VECTOR(15 downto 0);
        bit16InB : in STD_LOGIC_VECTOR(15 downto 0);
        bit16Out : out STD_LOGIC_VECTOR(15 downto 0); -- this goes to either datamem or regtable
        branchBit : out STD_LOGIC;
        signCtrl : in STD_LOGIC_VECTOR(3 downto 0));

end ALU;

architecture Behavioral of ALU is
signal bit16Outsig : STD_LOGIC_VECTOR(15 downto 0) := "0000000000000000";
signal branchBitsig : STD_LOGIC := '0';
begin
process(signCtrl)begin

        if(signCtrl = "0010") then --add
            bit16Outsig <= bit16InA + bit16InB;
        elsif(signCtrl = "0011") then --sub
            bit16Outsig <= bit16InA - bit16InB;

        elsif(signCtrl = "0110") then --mov/store passes A out to dataMem or registers
            bit16Outsig <= bit16InA;

        elsif(signCtrl = "1010") then
            if(bit16inA - bit16inB = "0000000000000000") then --branch if equal
                branchBitsig <= '1';
            else
                branchBitsig <= '0';
            end if;
        elsif(signCtrl = "1011") then
            if(bit16inA - bit16inB = 0) then --branch if not equal
                branchBitsig <= '0';
            else
                branchBitsig <= '1';
            end if;

        elsif(signCtrl = "1100") then
            bit16Outsig <= bit16InA or bit16InB; --or

        elsif(signCtrl = "1101") then
            bit16Outsig <= bit16InA and bit16InB; -- and

        elsif(signCtrl = "1110") then
            if(bit16inB - bit16inA > 0) then
                branchBitsig <= '1';  --Branch if less than
```

```
            else
                branchBitsig <= '0';
            end if;
        elsif(signCtrl = "1111") then
            if(bit16inA - bit16inB > 0) then
                branchBitsig <= '1';  --Branch if greater than
            else
                branchBitsig <= '0';
            end if;
        else
            bit16Outsig <= "0000000000000000";
        end if;

end process;

bit16Out <= bit16Outsig;
branchBit <= branchBitsig;
end Behavioral;
```

Register/Memory Mux:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.ALL;
use ieee.std_logic_unsigned.all;

entity REGMUX is
    port(
        clk : in STD_LOGIC;
        signCtrl : in STD_LOGIC;
        bit16Out : out STD_LOGIC_VECTOR(15 downto 0);
        aluIn : in STD_LOGIC_VECTOR(15 downto 0);
        memIn : in STD_LOGIC_VECTOR(15 downto 0));
end REGMUX;

architecture Behavioral of REGMUX is
signal bi16Outsig : STD_LOGIC_VECTOR(15 downto 0) := "0000000000000000";
begin

process(clk, signCtrl)begin
    if rising_edge(clk) then
        if(signCtrl = '0')then
            bi16Outsig <= aluIn;
        else
            bi16Outsig <= memIn;
        end if;
    end if;

end process;
bit16Out <= bi16Outsig;

end Behavioral;
```

Sign Extension 1:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.ALL;
use ieee.std_logic_unsigned.all;

entity SIGN1 is
    port(
        instrIn: in STD_LOGIC_VECTOR(15 downto 0);
        bit16Out: out STD_LOGIC_VECTOR(15 downto 0);
        signCtrl: in STD_LOGIC);
end SIGN1;

architecture Behavioral of SIGN1 is
signal bit16OutSig : STD_LOGIC_VECTOR(15 downto 0) := "0000000000000000";
begin
CTRL: process(signCtrl)
    begin
    if(signCtrl = '0') then
        bit16OutSig <= instrIn and "0000000011111111";

    elsif(signCtrl = '1') then
        bit16OutSig <= instrIn and "0000111111111111";
    end if;
    end process;
    bit16Out <= bit16OutSig;
end Behavioral;
```