Noah Niedzwiecki

ELEC 5200

Project 5

April 19, 2019

**Introduction**

The focus of this project is the implementation of memory into the current hardware that has been designed in past projects. The following memories have been added:

* Register File with 16 different registers
* Instruction memory with 216 addresses of 16-bit words
* Data memory with 216 addresses of 16-bit words

Through simulation analysis, these different memories were tested in increments.

**Testing**

First, to analyze the newly created register file. A rudimentary program was created in the instruction memory to test the features desired in our register file. First ‘0’ and ‘1’ were loaded into registers A and B respectively. Then, a series of additions were called that stored the result in register A. After each call the PC should increment by one and the instruction read should stay the same since the remaining instruction memory is filled with the add instruction. Referencing figure 1, the ‘READBUSA’ and ‘READBUSB’ signals each represent a specified register value. In figure 1, register A refers to ‘READBUSA’ and register B corresponds with ‘READBUSB’. The ‘WRITEBUS’ signals represents the selected data path to be written back into the register file. Since this is an R type instruction we need the data from the ALU to write to the register file.

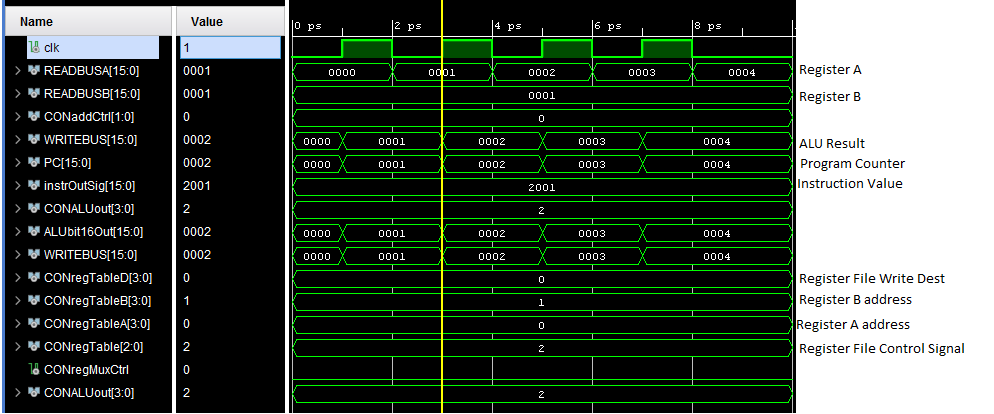


Figure 1: Register File Continuously Being Written to Register A

It can be seen from above that the addition of A and B were stored in register A, seeing how it increments after each instruction call. The ‘instrOutSig’ represent the instruction that was received from the instruction memory. Since *add* has opcode ‘0010’ the MSB of this signal is a two. The following number represents the address to write to, therefore 0. The last two values are the registers being called by the ALU to be operated on, so 0 and 1.

Since the above program did not specifically test the legitimacy of the data memory, another sample program was written to execute a load and store instruction, see figure 2.

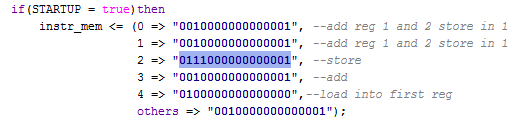


Figure 2: Data Memory Testing Code

As commented in figure 2, the code adds the two registers with addresses one and two together and stores the result in address one. This value is then saved in the data memory with an offset of one from the PC. Then an add instruction is called so that the retrieved value is unique from the value that was saved. Then finally the load retrieved the value that was previously saved into register A. For simplicity, values were manually inserted into the register file so that the mathematical operations could be immediately executed at start. Referencing figure 3 below, register A is originally populated with the value ‘0’, while register B is filled with ‘1’. These two values are added and the result is stored in register A. Again this is done, and at 3 ps the final value of ‘2’ is in register A. At 5ps, the next instruction is read and the value in register A is stored into PC+1 in the data memory. Then at 7ps another add is done so the sum in register A is now 3, then as soon as the next instruction is retrieved on the falling edge of the clock at 8ps, the value stored is written into register A, hence the ‘2’.

To further test the register files, another sample program was written. This time implementing the ‘Load Immediate’ instruction and ‘Branch On Equal’. The code for this test can be seen below in figure 3.

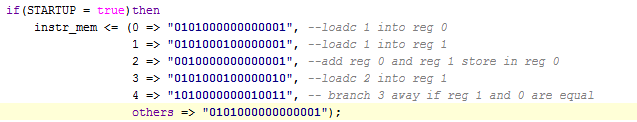


Figure 3: Code for Additional Register File Testing

Instead of manually writing to the register file, the ‘load c’ or load immediate command will be used to populate the register file. In this example, the value ‘1’ is loaded into register A as well as register B. Then these two registers are added together and the result is stored in register A. Since the value ‘2’ should now be in register A, ‘2’ is then loaded into register B so that we can test our branch on equal instruction. The last four bits in the branch instruction decide the value from the PC that should be jumped. Looking at figure 3, it can be seen that the PC should increment by 3. Looking at figure 4, we can see this code in simulation.

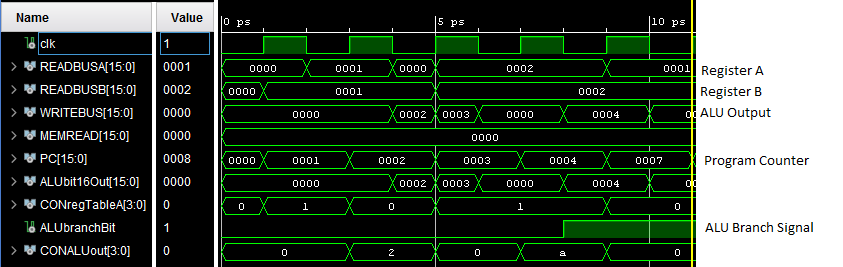


Figure 4: Continuation of Register File Test Simulation

Looking at the program counter transition from 7 to 9ps it can be determined that the branch was indeed taken due to the three value gap in the PC. The branch bit from the ALU is the trigger for the PC calculation hardware so the ALU must complete its analysis before the PC can be incremented.

Lastly, the data memory should be tested, to do so another basic program was derived. The created code can be seen in figure 5.

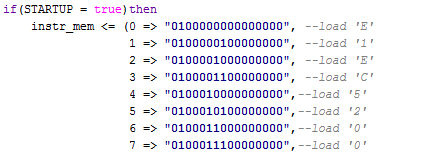


Figure 5: Code to Read from Data Memory and Written to the Register File

Figure 6 shows the code being hard-coded into the data memory:

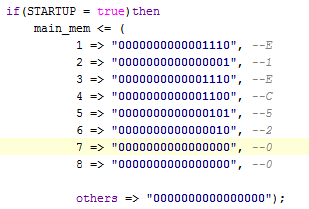
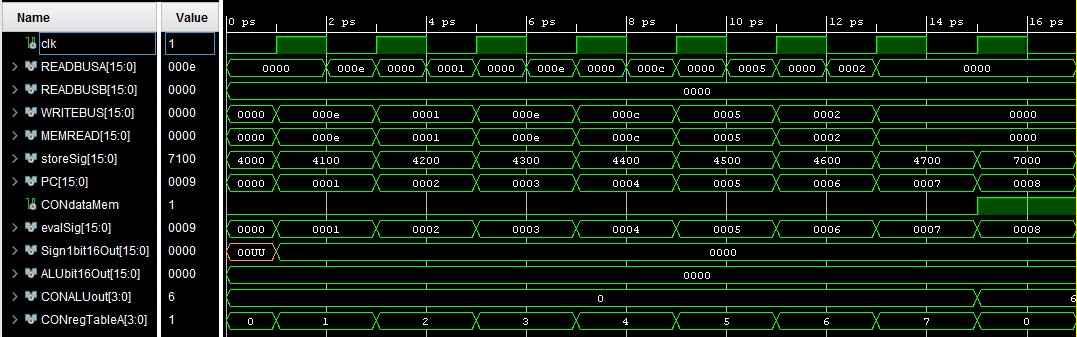


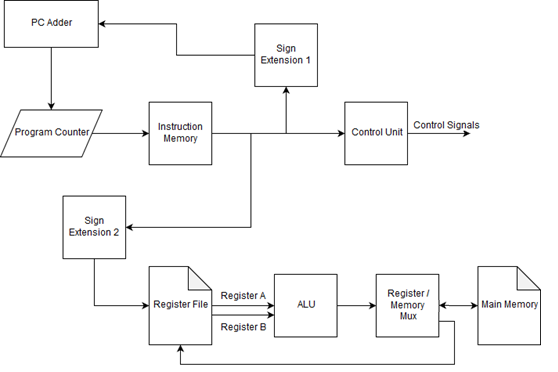
Figure 6: Hard-Coded Data in the Data Memory

This code resulted in the simulation seen in figure 7 on the next page. This simulation shows that the data memory can be directly read from and inserted into the register file. Instead of using a single register, each of the 8 values in the string ‘ELEC5200’ is to be written to registers 0-7. In the instruction memory it can be observed that bits 11 through 8 dictate which of the 16 available registers in the register file should be written to. The last eight bits determine the offset from the PC at which the address in the data memory should be queried. These values remain 0 due to the fact that the PC is incrementing every clock and the stored values are sequential in memory starting at 0.



**Notes**

At the end of this exercise, three functioning memory modules were designed. The most complex of these is the register file since it is having to be written to and read in the same clock cycle. Due to timing errors and general redundancy, several components of the control unit were removed. In particular, the branch bit from the ALU was originally fed into the control block which then controlled the PC addition unit. This was removed and the signal was directly fed into the add unit instead.



General CPU Data Path

**Code Directory**

Main pg 9 – 15

ALU pg 16 – 18

Control pg 19 – 25

Register Mux pg 26

Instruction Memory pg 28 – 29

Data Memory pg 30 – 31

Register File pg 32 – 33

PC pg 34

Sign Extend 2 pg 35

Sign Extend 1 pg 36

**Main 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);

signal EXTENDBUS : STD\_LOGIC\_VECTOR(15 downto 0);

signal LINKBUS : 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;

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;

--INSTR MEM

signal instrOutSig : STD\_LOGIC\_VECTOR(15 downto 0);

-- PC

signal PC : STD\_LOGIC\_VECTOR(15 downto 0);

--Store Signal

signal storeSig : STD\_LOGIC\_VECTOR(15 downto 0);

signal evalSig : STD\_LOGIC\_VECTOR(15 downto 0);

component ADD

port(

branchIn : in STD\_LOGIC;

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(

instruction : in STD\_LOGIC\_VECTOR(15 downto 0);

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);

regFileCtrl : out STD\_LOGIC\_VECTOR(2 downto 0);

storeAddr : out STD\_LOGIC\_VECTOR(15 downto 0);

signCtrl: out STD\_LOGIC;

dataMem : out STD\_LOGIC;

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(

clk : in STD\_LOGIC;

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(15 downto 0));

end component;

component INSTRMEM

port (

clk : in STD\_LOGIC;

readAddr : in STD\_LOGIC\_VECTOR(15 downto 0);

instrOut : out STD\_LOGIC\_VECTOR(15 downto 0));

end component;

component PCREG

port(

clk : in STD\_LOGIC;

addrIn : in STD\_LOGIC\_VECTOR(15 downto 0);

addrOut : out STD\_LOGIC\_VECTOR(15 downto 0));

end component;

component REGFILE

Port (

clk : in STD\_LOGIC;

readAddrA : in STD\_LOGIC\_VECTOR(3 downto 0);

extendIn : in STD\_LOGIC\_VECTOR(15 downto 0);

readAddrB : in STD\_LOGIC\_VECTOR(3 downto 0);

writeAddrA : in STD\_LOGIC\_VECTOR(3 downto 0);

signCtrl : in STD\_LOGIC\_VECTOR(2 downto 0);

valIn : in STD\_LOGIC\_VECTOR(15 downto 0);

linkOut : out STD\_LOGIC\_VECTOR(15 downto 0);

valOutA : out STD\_LOGIC\_VECTOR(15 downto 0);

valOutB : out STD\_LOGIC\_VECTOR(15 downto 0));

end component;

component MAINMEM

Port (

clk : in STD\_LOGIC;

evalAddr : out STD\_LOGIC\_VECTOR(15 downto 0);

storeOff : in STD\_LOGIC\_VECTOR(15 downto 0);

PCin : in STD\_LOGIC\_VECTOR(15 downto 0);

signCtrl : in STD\_LOGIC;

valIn : in STD\_LOGIC\_VECTOR(15 downto 0);

valOut : out STD\_LOGIC\_VECTOR(15 downto 0));

end component;

begin

A1 : ADD port map(branchIn => ALUbranchBit,clk => clk,bit16In => Sign1bit16Out, PCin => PC, REGin => LINKBUS, bit16Out => bit16OutAdd, signCtrl => CONaddCtrl);

--dear lord

A2: CONTROL port map(instruction => instrOutSig, regMuxCtrl => CONregMuxCtrl,

regTableA => CONregTableA, regTableB => CONregTableB, regTableD => CONregTableD, ALUout => CONALUout,

regTable => CONregTable, signCtrl => CONsignCtrl, dataMem => CONdataMem, addCtrl => CONaddCtrl,storeAddr => storeSig);

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

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

A5: SIGN2 port map(instrIn => instrOutSig, bit16Out => EXTENDBUS);

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

A7: INSTRMEM port map(clk => clk, readAddr => PC, instrOut => instrOutSig);

A8 : PCREG port map(clk => clk, addrIn => bit16OutAdd, addrOut => PC);

A9 : REGFILE port map(clk => clk,extendIn => EXTENDBUS, readAddrA => CONregTableA, readAddrB => CONregTableB, writeAddrA => CONregTableD,

signCtrl => CONregTable, valIn => WRITEBUS, linkOut => LINKBUS, valOutA => READBUSA, valOutB => READBUSB);

A10 : MAINMEM port map(clk => clk, storeOff => storeSig, PCin => PC,

signCtrl => CONdataMem, valIn => ALUbit16Out,valOut => MEMREAD,evalAddr => evalSig);

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(15 downto 0));

end ALU;

architecture Behavioral of ALU is

signal bit16Outsig : STD\_LOGIC\_VECTOR(15 downto 0) := "0000000000000000";

signal branchBitsig : STD\_LOGIC := '0';

signal signCtrl1 : STD\_LOGIC\_VECTOR(3 downto 0);

begin

process(signCtrl,clk)begin

signCtrl1 <= signCtrl(15 downto 12);

--if(rising\_edge(clk))then

if(signCtrl1 = "0010") then --add

bit16Outsig <= bit16InA + bit16InB;

elsif(signCtrl1 = "0011") then --sub

bit16Outsig <= bit16InA - bit16InB;

elsif(signCtrl1 = "0110") then --mov/store passes A out to dataMem or registers

bit16Outsig <= bit16InA;

elsif(signCtrl1 = "1010") then

bit16Outsig <= bit16InA + bit16InB;

if(bit16inA - bit16inB = "0000000000000000") then --branch if equal

branchBitsig <= '1';

else

branchBitsig <= '0';

end if;

elsif(signCtrl1 = "1011") then

if(bit16inA - bit16inB = 0) then --branch if not equal

branchBitsig <= '0';

else

branchBitsig <= '1';

end if;

elsif(signCtrl1 = "1100") then

bit16Outsig <= bit16InA or bit16InB; --or

elsif(signCtrl1 = "1101") then

bit16Outsig <= bit16InA and bit16InB; -- and

elsif(signCtrl1 = "1110") then

if(bit16inB - bit16inA > 0) then

branchBitsig <= '1'; --Branch if less than

else

branchBitsig <= '0';

end if;

elsif(signCtrl1 = "1111") then

if(bit16inA - bit16inB > 0) then

branchBitsig <= '1'; --Branch if greater than

else

branchBitsig <= '0';

end if;

else

bit16Outsig <= "0000000000000000";

end if;

--end if;

end process;

bit16Out <= bit16Outsig;

branchBit <= branchBitsig;

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(

instruction : in STD\_LOGIC\_VECTOR(15 downto 0);

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);

regFileCtrl : out STD\_LOGIC\_VECTOR(2 downto 0);

storeAddr : out STD\_LOGIC\_VECTOR(15 downto 0);

signCtrl: out STD\_LOGIC;

dataMem : out STD\_LOGIC;

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 := '0';

signal addCtrlsig: STD\_LOGIC\_VECTOR(1 downto 0) := "00";

signal storeAddrSig : STD\_LOGIC\_VECTOR(15 downto 0);

begin

process(instruction) begin

if(instruction(15 downto 12) = "0000") then --halt

addCtrlsig <= "11";

regTablesig <= "000";

elsif(instruction(15 downto 12) = "0001")then --jump

signCtrlsig <= '1';

addCtrlsig <= "01"; --sign extended

regTablesig <= "000";

elsif(instruction(15 downto 12) = "0010")then --add

regTableDsig <= instruction(11 downto 8);

regTableAsig <= instruction(7 downto 4);

regTableBsig <= instruction(3 downto 0);

ALUOutsig <= "0010";

dataMemsig <= '0'; -- do nothing

addCtrlsig <= "00"; --PC + 1

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

regTablesig <= "010";

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 <= '0'; --do nothing

addCtrlsig <= "00"; --PC + 1

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

regTablesig <= "010";

elsif(instruction(15 downto 12) = "0100")then --load

regTableAsig <= instruction(11 downto 8);

storeAddrSig <= instruction(15 downto 0);

regTablesig <= "001"; --read so regtable looks at input

regTableDsig <= instruction(11 downto 8); --load address

ALUOutsig <= "0000"; --do nothing

dataMemsig <= '0'; --write pcIn address

addCtrlsig <= "00"; --PC + 1

regMuxCtrlsig <= '1'; --MEM TO REG

elsif(instruction(15 downto 12) = "0101")then --load C

regTableAsig <= instruction(11 downto 8);

regTablesig <= "011"; --read sign extend 2

regTableDsig <= instruction(11 downto 8); --load address

ALUOutsig <= "0000"; --do nothing

dataMemsig <= '0'; --do nothing

addCtrlsig <= "00"; --PC + 1

elsif(instruction(15 downto 12) = "0110")then --MOV

regTableDsig <= instruction(11 downto 8); --destination

regTableAsig <= instruction(7 downto 4); --value being moved

ALUOutsig <= "0110"; --pass values from reg to reg mux

dataMemsig <= '0'; --default

addCtrlsig <= "00"; --PC + 1

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

regTablesig <= "010";

elsif(instruction(15 downto 12) = "0111")then --STORE

storeAddrSig <= instruction(15 downto 0);

regTablesig <= "000"; --rtype

regTableAsig <= instruction(11 downto 8);

ALUOutsig <= "0110"; --pass values from reg to reg mux

dataMemsig <= '1'; --write to mem

addCtrlsig <= "00"; --PC + 1

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

elsif(instruction(15 downto 12) = "1000")then --JUMP AND LINK

regTableAsig <= "1111"; --value being stored

ALUOutsig <= "0000"; --pass values from reg to reg mux

dataMemsig <= '0'; --do nothing

signCtrlsig <= '1'; --put 12 bit offset to add

addCtrlsig <= "01"; --sign extended

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

regTablesig <= "100";

elsif(instruction(15 downto 12) = "1001")then --RETURN

regTablesig <= "000"; --do nothing

dataMemsig <= '0'; --do nothing

addCtrlsig <= "10"; -- add puts the register into PC

ALUOutsig <= "0000"; --do nothing

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

elsif(instruction(15 downto 12) = "1010")then --BRANCH IF EQUAL

dataMemsig <='0'; --do nothing

ALUOutsig <= "1010"; --branch if equal

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

signCtrlsig <= '0'; --put 8 bit offset to add

regTablesig <= "000"; --read the 2 registers

addCtrlsig <= "00"; -- PC + 1

elsif(instruction(15 downto 12) = "1011")then --BRANCH IF NOT EQUAL

dataMemsig <= '0'; --do nothing

signCtrlsig <= '0'; --put 8 bit offset to add

ALUOutsig <= "1011"; --branch if equal

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

regTablesig <= "000"; --read the 2 registers

addCtrlsig <= "00"; -- PC + 1

elsif(instruction(15 downto 12) = "1100")then --OR

regTableDsig <= instruction(11 downto 8);

regTableAsig <= instruction(7 downto 4);

regTableBsig <= instruction(3 downto 0);

ALUOutsig <= "1100";

dataMemsig <= '0'; -- do nothing

addCtrlsig <= "00"; --PC + 1

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

regTablesig <= "010"; --read the 2 registers and write

elsif(instruction(15 downto 12) = "1101")then --AND

regTableDsig <= instruction(11 downto 8);

regTableAsig <= instruction(7 downto 4);

regTableBsig <= instruction(3 downto 0);

ALUOutsig <= "1101";

dataMemsig <= '0'; -- do nothing

addCtrlsig <= "00"; --PC + 1

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

regTablesig <= "010";

elsif(instruction(15 downto 12) = "1110")then --BRANCH IF LESS THAN

dataMemsig <= '0'; --do nothing

signCtrlsig <= '0'; --put 8 bit offset to add

ALUOutsig <= "1110"; --branch if equal

regTablesig <= "000"; --read the 2 registers

addCtrlsig <= "00"; -- PC + 1

elsif(instruction(15 downto 12) = "1111")then --BRANCH IF greater THAN

dataMemsig <= '0'; --do nothing

signCtrlsig <= '0'; --put 8 bit offset to add

ALUOutsig <= "1111"; --branch if equal

regTablesig <= "000"; --read the 2 registers

addCtrlsig <= "00"; -- PC + 1

end if;

end process;

storeAddr <= storeAddrSig;

regTableA <= regTableAsig;

regTableB <= regTableBsig;

regTableD <= regTableDsig;

ALUout <= ALUOutsig;

regTable <= regTablesig;

signCtrl <= signCtrlsig;

dataMem <= dataMemsig;

addCtrl <= addCtrlsig;

regMuxCtrl <= regMuxCtrlsig;

end Behavioral;

**Register 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 bit16Outsig : STD\_LOGIC\_VECTOR(15 downto 0) := "0000000000000000";

begin

process(clk, signCtrl,aluIn,memIn)begin

--if rising\_edge(clk) then

if(signCtrl = '0')then

bit16Outsig <= aluIn;

else

bit16Outsig <= memIn;

end if;

--end if;

end process;

bit16Out <= bit16Outsig;

end Behavioral;

**Instruction Memory**

library IEEE;

use IEEE.std\_logic\_1164.all;

use IEEE.std\_logic\_unsigned.all;

entity INSTRMEM is

Port (

clk : in STD\_LOGIC;

readAddr : in STD\_LOGIC\_VECTOR(15 downto 0);

instrOut : out STD\_LOGIC\_VECTOR(15 downto 0));

end INSTRMEM;

architecture Behavioral of INSTRMEM is

subtype WORD is STD\_LOGIC\_VECTOR (15 downto 0);

type MEMORY is array (65535 downto 0) of WORD; -- 2^16 words

signal instr\_mem: MEMORY;

signal instrOutsig : STD\_LOGIC\_VECTOR(15 downto 0);

begin

process(readAddr,clk)

variable R\_ADDR: integer range 0 to 65535;

variable W\_ADDR: integer range 0 to 65535;

variable STARTUP: boolean := true;

begin

if(STARTUP = true)then

instr\_mem <= (0 => "0101000000000001", --loadc 1 into reg 0

1 => "0101000100000001", --loadc 1 into reg 1

2 => "0010000000000001", --add reg 0 and reg 1 store in reg 0

3 => "0101000100000010", --loadc 2 into reg 1

4 => "1010000000010011", -- branch 3 away if reg 1 and 0 are equal

others => "0101000000000001");

STARTUP := false;

else

R\_ADDR := conv\_integer(readAddr);

instrOut <= instr\_mem(R\_ADDR);

end if;

end process;

end Behavioral;

**Data Memory**

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

use IEEE.numeric\_std.ALL;

use ieee.std\_logic\_unsigned.all;

entity MAINMEM is

Port (

clk : in STD\_LOGIC;

storeOff : in STD\_LOGIC\_VECTOR(15 downto 0);

PCin : in STD\_LOGIC\_VECTOR(15 downto 0);

signCtrl : in STD\_LOGIC;

valIn : in STD\_LOGIC\_VECTOR(15 downto 0);

evalAddr : out STD\_LOGIC\_VECTOR(15 downto 0);

valOut : out STD\_LOGIC\_VECTOR(15 downto 0));

end MAINMEM;

architecture Behavioral of MAINMEM is

subtype WORD is STD\_LOGIC\_VECTOR (15 downto 0);

type MEMORY is array (65535 downto 0) of WORD; -- 2^16 words

signal main\_mem: MEMORY;

signal regFileOutA : STD\_LOGIC\_VECTOR(15 downto 0);

signal writeAd : STD\_LOGIC\_VECTOR(15 downto 0);

begin

process(storeOff,clk,PCin,signCtrl)

variable W\_ADDR: integer range 0 to 65535;

variable STARTUP: boolean := true;

begin

if(STARTUP = true)then

main\_mem <= (

1 => "0000000000001110", --E

2 => "0000000000000001", --1

3 => "0000000000001110", --E

4 => "0000000000001100", --C

5 => "0000000000000101", --5

6 => "0000000000000010", --2

7 => "0000000000000000", --0

8 => "0000000000000000", --0

others => "0000000000000000");

STARTUP := false;

else

writeAd <= (storeOff and "0000000011111111") + PCIN;

W\_ADDR := conv\_integer((storeOff and "0000000011111111") + PCIN);

if(signCtrl = '1') then

main\_mem(W\_ADDR) <= valIn; --Write

end if;

valOut <= main\_mem(W\_ADDR);

end if;

end process;

evalAddr <= writeAd;

end Behavioral;

**Register File**

library IEEE;

use IEEE.std\_logic\_1164.all;

use IEEE.std\_logic\_unsigned.all;

entity REGFILE is

Port (

clk : in STD\_LOGIC;

readAddrA : in STD\_LOGIC\_VECTOR(3 downto 0);

readAddrB : in STD\_LOGIC\_VECTOR(3 downto 0);

writeAddrA : in STD\_LOGIC\_VECTOR(3 downto 0);

signCtrl : in STD\_LOGIC\_VECTOR(2 downto 0);

valIn : in STD\_LOGIC\_VECTOR(15 downto 0);

extendIn : in STD\_LOGIC\_VECTOR(15 downto 0);

linkOut : out STD\_LOGIC\_VECTOR(15 downto 0);

valOutA : out STD\_LOGIC\_VECTOR(15 downto 0);

valOutB : out STD\_LOGIC\_VECTOR(15 downto 0));

end REGFILE;

architecture Behavioral of REGFILE is

subtype WORD is STD\_LOGIC\_VECTOR (15 downto 0);

type MEMORY is array (15 downto 0) of WORD; -- 2^16 words

signal reg\_file: MEMORY;

signal regFileOutA : STD\_LOGIC\_VECTOR(15 downto 0);

signal regFileOutB : STD\_LOGIC\_VECTOR(15 downto 0);

begin

process(readAddrA,writeAddrA,readAddrB,clk,signCtrl,valIn,extendIn)

variable R\_ADDRA: integer range 0 to 15;

variable R\_ADDRB: integer range 0 to 15;

variable W\_ADDRA: integer range 0 to 15;

variable STARTUP: boolean := true;

begin

if(STARTUP = true)then

reg\_file <= (0 => "0000000000000000",

1 => "0000000000000000",

2 => "0000000000000000",

others => "0000000000000000");

STARTUP := false;

else

R\_ADDRA := conv\_integer(readAddrA);

W\_ADDRA := conv\_integer(writeAddrA);

R\_ADDRB := conv\_integer(readAddrB);

--if(falling\_edge(clk))then

if(signCtrl = "001") then

reg\_file(W\_ADDRA) <= valIn; --Write to A

elsif(signCtrl = "010") then

reg\_file(W\_ADDRA) <= valIn; --Write to A

elsif(signCtrl = "011") then

reg\_file(W\_ADDRA) <= extendIn; --Write to A

elsif(signCtrl = "100")then

reg\_file(15) <= valIn; -- write to the link

end if;

--end if;

valOutA <= reg\_file(R\_ADDRA);

valOutB <= reg\_file(R\_ADDRB);

end if;

end process;

linkOut <= reg\_file(15);

end Behavioral;

**PC Register**

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

use IEEE.numeric\_std.ALL;

use ieee.std\_logic\_unsigned.all;

entity PCREG is

port(

clk : in STD\_LOGIC;

addrIn : in STD\_LOGIC\_VECTOR(15 downto 0);

addrOut : out STD\_LOGIC\_VECTOR(15 downto 0));

end PCREG;

architecture Behavioral of PCREG is

signal outputSig : STD\_LOGIC\_VECTOR(15 downto 0);

signal startAddr : STD\_LOGIC\_VECTOR(15 downto 0) := "0000000000000000";

begin

process(addrIn)

variable STARTUP: boolean := true;

begin

if(STARTUP = true)then

outputSig <= startAddr;

STARTUP := false;

else

outputSig <= addrIn;

end if;

end process;

addrOut <= outputSig;

end Behavioral;

**Sign Extension 2**

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

use IEEE.numeric\_std.ALL;

use ieee.std\_logic\_unsigned.all;

entity SIGN2 is

port(

instrIn : in STD\_LOGIC\_VECTOR(15 downto 0);

bit16Out: out STD\_LOGIC\_VECTOR(15 downto 0));

end SIGN2;

architecture Behavioral of SIGN2 is

signal bit16OutSig : STD\_LOGIC\_VECTOR(15 downto 0) := "0000000000000000";

begin

process(instrIn)

begin

bit16OutSig <= instrIn and "0000000011111111";

end process;

bit16Out <= bit16OutSig;

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(

clk : in STD\_LOGIC;

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,clk)

begin

if(signCtrl = '0') then

bit16OutSig <= instrIn AND "0000000000001111";

elsif(signCtrl = '1') then

bit16OutSig <= instrIn AND "0000111111111111";

end if;

end process;

bit16Out <= bit16OutSig;

end Behavioral;