ECGR 3183: Computer Organization - Control Units

Mikey Neal

April 27, 2021

Contents

Main Control Unit

VHDL Code

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity Main_Control is
    port(   UPPER_11_OPCODE : in std_logic_vector(10 downto 0);
            Reg2Loc : out std_logic;
            ALUSrc : out std_logic;
            MemtoReg : out std_logic;
            RegWrite : out std_logic;
            MemRead : out std_logic;
            MemWrite : out std_logic;
            Branch : out std_logic;
            UncondBranch : out std_logic;
            ALU_OP : out std_logic_vector(1 downto 0));
end Main_Control;

architecture Behavioral of Main_Control is
begin
    process (UPPER_11_OPCODE)
    begin
        if (UPPER_11_OPCODE(10 downto 5) = "000101")
            or (UPPER_11_OPCODE(10 downto 4) = "1011010") then -- CBNZ, CBZ, or B
            Reg2Loc      <= '1';
            ALUSrc       <= '0';
            --MemtoReg   <= '1';
            RegWrite     <= '0';
            MemRead      <= '0';
            MemWrite     <= '0';
            Branch       <= '1';
            UncondBranch <= NOT(UPPER_11_OPCODE(10));
            ALU_OP       <= "01";
        elsif (UPPER_11_OPCODE(10 downto 0) = "11111000000") then -- STUR
            Reg2Loc      <= '1';
            ALUSrc       <= '1';
            --MemtoReg   <= '1';
            RegWrite     <= '0';
            MemRead      <= '0';
            MemWrite     <= '1';
            Branch       <= '0';
            UncondBranch <= '0';
            ALU_OP       <= "00";
        elsif (UPPER_11_OPCODE(10 downto 0) = "11111000010") then -- LDUR
            --Reg2Loc    <= '1';
            ALUSrc       <= '1';
            MemtoReg     <= '1';
            RegWrite     <= '1';
            MemRead      <= '1';
            MemWrite     <= '0';
```

```
            Branch       <= '0';
            UncondBranch <= '0';
            ALU_OP       <= "00";
        else
            Reg2Loc      <= '0';
            ALUSrc       <= '0';
            MemtoReg     <= '0';
            RegWrite     <= '1';
            MemRead      <= '0';
            MemWrite     <= '0';
            Branch       <= '0';
            UncondBranch <= '0';
            ALU_OP       <= "10";
        end if;
    end process;
end Behavioral;
```

Explanation

The Main Control Unit takes in the upper 11 bits of an instruction (denoted as "UPPER_11_OPCODE") and outputs control lines for the rest of the CPU (denoted as "Reg2Loc", "ALUSrc", "MemtoReg", "RegWrite", "MemRead", "MemWrite", "Branch", "UncondBranch", and "ALU_OP").

Output

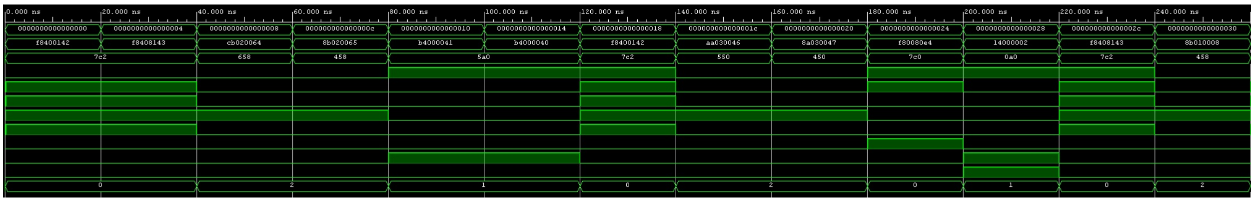

Figure 1: Graph of the output for the Main Control Unit using the Instruction Memory

ALU Control Unit

VHDL Code

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity ALU_Control is
    port(   ALU_OP : in std_logic_vector(1 downto 0);
            UPPER_11_OPCODE : in std_logic_vector(10 downto 0);
            ALU_Code : out std_logic_vector(3 downto 0));
end ALU_Control;

architecture Behavioral of ALU_Control is
begin
    process (UPPER_11_OPCODE, ALU_OP)
    begin
        if    (UPPER_11_OPCODE(10 downto 0) = "10001010000") then
            ALU_Code <= "0000"; -- AND
        elsif (UPPER_11_OPCODE(10 downto 0) = "10101010000") then
            ALU_Code <= "0001"; -- ORR
        elsif (UPPER_11_OPCODE(10 downto 0) = "11111000000")
            or (UPPER_11_OPCODE(10 downto 0) = "11111000010")
            or (UPPER_11_OPCODE(10 downto 0) = "10001011000") then
            ALU_Code <= "0010"; -- STUR, LDUR, or ADD
        elsif (UPPER_11_OPCODE(10 downto 0) = "11001011000") then
            ALU_Code <= "0110"; -- SUB
        elsif (UPPER_11_OPCODE(10 downto 5) = "000101")
            or (UPPER_11_OPCODE(10 downto 4) = "1011010") then
            ALU_Code <= "0111"; -- B, CBZ, or CBNZ
        end if;
    end process;
end Behavioral;
```

Explanation

The ALU Control Unit takes in the upper 11 bits of an instruction (denoted as “UPPER_11_OPCODE”) and outputs the ALU type Operation Code (denoted as “ALU_Code”). It also takes in the “ALU_OP” from the main control, but it is unused as the “UPPER_11_OPCODE” can more easily be used to determine the instruction being used.

Output

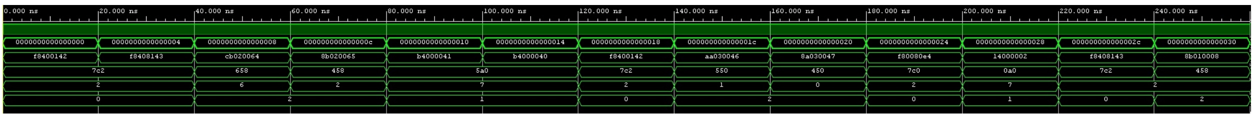

Figure 2: Graph of the output for the ALU Control Unit using the Instruction Memory

CPU Units from the Datapath Elements Project

Program Counter

VHDL Code

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity Program_Counter is
    Port ( Clock : in STD_LOGIC;
           PC_INPUT : in STD_LOGIC_VECTOR (63 downto 0);
           PC_OUTPUT : out STD_LOGIC_VECTOR (63 downto 0));
end Program_Counter;

architecture Behavioral of Program_Counter is

begin
    process (Clock)
    begin
        if rising_edge(Clock) then
            PC_OUTPUT <= PC_INPUT;
        end if;
    end process;
end Behavioral;
```

Explanation

On the rising edge of the clock (denoted as “Clock”), the output (denoted as “PC_OUTPUT”) will be set equal to the current input (denoted as “PC_INPUT”).

Instruction Memory

VHDL Code

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity INSTUCTION_RAM is
    port(write_enable, read_enable : in std_logic;
         address : in std_logic_vector(63 downto 0);
         data_in : in std_logic_vector(31 downto 0);
         data_out : out std_logic_vector(31 downto 0));
end INSTUCTION_RAM;

architecture Behavioral of INSTUCTION_RAM is

type memory is array(0 to 255) of std_logic_vector(7 downto 0);
signal ram : memory := (0  => "01000010",
                        1  => "00000001",
                        2  => "01000000",
                        3  => "11111000",
                        4  => "01000011",
                        5  => "10000001",
                        6  => "01000000",
                        7  => "11111000",
                        8  => "01100100",
                        9  => "00000000",
                        10 => "00000010",
                        11 => "11001011",
                        12 => "01100101",
                        13 => "00000000",
                        14 => "00000010",
                        15 => "10001011",
                        16 => "01000001",
                        17 => "00000000",
                        18 => "00000000",
                        19 => "10110100",
                        20 => "01000000",
                        21 => "00000000",
                        22 => "00000000",
                        23 => "10110100",
                        24 => "01000010",
                        25 => "00000001",
                        26 => "01000000",
                        27 => "11111000",
                        28 => "01000110",
                        29 => "00000000",
                        30 => "00000011",
                        31 => "10101010",
                        32 => "01000111",
                        33 => "00000000",
                        34 => "00000011",
```

```
                        35 => "10001010",
                        36 => "11100100",
                        37 => "10000000",
                        38 => "00000000",
                        39 => "11111000",
                        40 => "00000010",
                        41 => "00000000",
                        42 => "00000000",
                        43 => "00010100",
                        44 => "01000011",
                        45 => "10000001",
                        46 => "01000000",
                        47 => "11111000",
                        48 => "00001000",
                        49 => "00000000",
                        50 => "00000001",
                        51 => "10001011",
                        others => "00000000");
begin
    dw : process(write_enable, address, data_in) is
        begin
            if write_enable = ’1’ then
                ram(to_integer(unsigned(address)) + 0) <= data_in( 7 downto  0);
                ram(to_integer(unsigned(address)) + 1) <= data_in(15 downto  8);
                ram(to_integer(unsigned(address)) + 2) <= data_in(23 downto 16);
                ram(to_integer(unsigned(address)) + 3) <= data_in(31 downto 24);
            end if;
    end process dw;
    dr : process(read_enable, address) is
        begin
            if read_enable = ’1’ then
                data_out( 7 downto  0) <= ram(to_integer(unsigned(address)) + 0);
                data_out(15 downto  8) <= ram(to_integer(unsigned(address)) + 1);
                data_out(23 downto 16) <= ram(to_integer(unsigned(address)) + 2);
                data_out(31 downto 24) <= ram(to_integer(unsigned(address)) + 3);
            end if;
    end process dr;
end Behavioral;
```

Explanation

When the write enable (denoted as “write_enable”) is true, it will take the current input (denoted as “data_in”) and set the memory (in variable “ram”) at the address (denoted as “address”) to that value. When the read enable (denoted as “read_enable”) is true, it will take the memory (in variable “ram”) at the address (denoted as “address”) and output that value (through the signal “data_out”).

Output

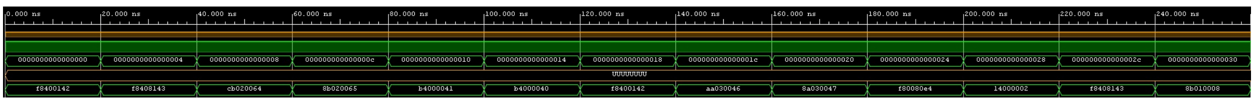

Figure 3: Graph of the output for the Instruction Memory

Register File

VHDL Code

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity REGISTER_FILE is
    port(write_enable : in std_logic;
         address_in, address_out_1, address_out_2 : in std_logic_vector(4 downto 0);
         data_in : in std_logic_vector(63 downto 0);
         data_out_1, data_out_2 : out std_logic_vector(63 downto 0));
end REGISTER_FILE;

architecture Behavioral of REGISTER_FILE is

type memory is array(0 to 31) of std_logic_vector(63 downto 0);
signal ram : memory := (
    others => "0000000000000000000000000000000000000000000000000000000000000000");
begin
    process(write_enable, address_in, data_in, address_out_1, address_out_2) is
        begin
            if write_enable = '1' then
                if NOT(address_in = "11111") then
                    ram(to_integer(unsigned(address_in))) <= data_in;
                end if;
            end if;
                data_out_1 <= ram(to_integer(unsigned(address_out_1)));
                data_out_2 <= ram(to_integer(unsigned(address_out_2)));
    end process;
end Behavioral;
```

Explanation

When the write enable (denoted as “write_enable”) is true, it will take the current input (denoted as “data_in”) and set the memory (in variable “ram”) at the input address (denoted as “address_in”) to that value, unless the address is XZR, then it will do nothing. It will take the memory (in variable “ram”) at the output addresses (denoted as “address_out_1” and “address_out_2”) and output those values (through the signal “data_out_1” and “data_out_2” respective to the address_out numbers).

Output

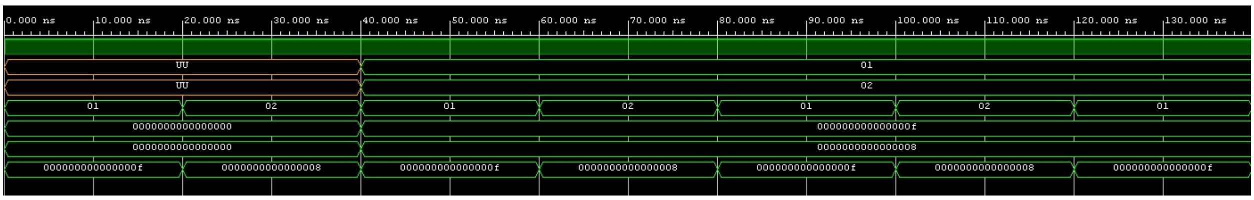

Figure 4: Graph of the output for the Register File

ALU

VHDL Code

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity ALUv2 is
    Port ( A_in : in STD_LOGIC_VECTOR (63 downto 0);
           B_in : in STD_LOGIC_VECTOR (63 downto 0);
           ALU_Out : out STD_LOGIC_VECTOR (63 downto 0);
           ALU_Flags : out STD_LOGIC_VECTOR (3 downto 0);
           OPCODE : in STD_LOGIC_VECTOR (3 downto 0));
end ALUv2;
architecture Behavioral of ALUv2 is

begin
    Process (OPCODE, A_in, B_in)
        variable Temp : STD_LOGIC_VECTOR (63 downto 0);
        variable Temp_ALU_OUT : STD_LOGIC_VECTOR (63 downto 0);
        variable Temp_ALU_FLAGS : STD_LOGIC_VECTOR (3 downto 0);
        variable Temp2 : STD_LOGIC;
    begin
    ALU_Flags(0) <= '0';
    ALU_Flags(1) <= '0';
        if (OPCODE = "0010") or (OPCODE = "0110") then
            if (OPCODE = "0110") then
                Temp_ALU_FLAGS(1) := '1';
                Temp2 := '1';
                Temp := NOT(B_in);
            else
                Temp_ALU_FLAGS(1) := '0';
                Temp2 := '0';
                Temp := (B_in);
            end if;
            for i in 0 to 63 loop
                Temp_ALU_FLAGS(0) := Temp_ALU_FLAGS(1);
                Temp_ALU_OUT(i) := (Temp(i) xor A_in(i)) xor Temp_ALU_FLAGS(1);
                Temp_ALU_FLAGS(1) := (Temp(i) and A_in(i)) or
                    (Temp_ALU_FLAGS(1) and Temp(i)) or
                    (Temp_ALU_FLAGS(1) and A_in(i));
            end loop;
            ALU_Flags(0) <= Temp_ALU_FLAGS(0) xor Temp_ALU_FLAGS(1);
            ALU_Flags(1) <= Temp2 xor Temp_ALU_FLAGS(1);
        elsif(OPCODE = "0000") then
            Temp_ALU_OUT := B_in and A_in;
        elsif(OPCODE = "0001") then
            Temp_ALU_OUT := B_in or A_in;
        elsif(OPCODE = "0111") then
            Temp_ALU_OUT := B_in;
        elsif(OPCODE = "1100") then
```

```
            Temp_ALU_OUT := B_in nor A_in;
        else
            Temp_ALU_OUT :=
                "0000000000000000000000000000000000000000000000000000000000000000";
        end if;
        if (Temp_ALU_OUT =
            "0000000000000000000000000000000000000000000000000000000000000000") then
            ALU_Flags(2) <= '1';
        else
            ALU_Flags(2) <= '0';
        end if;
        ALU_Flags(3) <= Temp_ALU_OUT(63);
        ALU_OUT <= Temp_ALU_OUT;
    end Process;
end Behavioral;
```

Explanation

The ALU takes two inputs, "A_in" and "B_in", and acts on them based on the operation code (denoted as "OPCODE"), these codes were given in the assignment. The ALU outputs several flags in the "ALU_Flags" array, the array is formatted {Negative Flag, Zero Flag, Carry Flag, Overflow Flag}. The ALU Output is through "ALU_Out".

Output

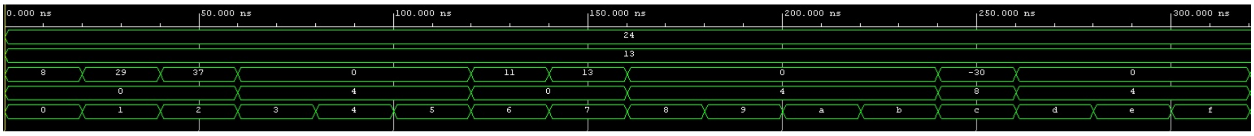

Figure 5: Graph of the output for the ALU

Data Memory

VHDL Code

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity Data_RAM is
    port(write_enable, read_enable : in std_logic;
         address, data_in : in std_logic_vector(63 downto 0);
         data_out : out std_logic_vector(63 downto 0));
end Data_RAM;

architecture Behavioral of Data_RAM is
    type memory is array(0 to 255) of std_logic_vector(7 downto 0);
    signal ram : memory := (0  => "01000010",
                            1  => "00000001",
                            2  => "01000000",
                            3  => "11111000",
                            4  => "01000011",
                            5  => "10000001",
                            6  => "01000000",
                            7  => "11111000",
                            8  => "01100100",
                            9  => "00000000",
                            10 => "00000010",
                            11 => "11001011",
                            12 => "01100101",
                            13 => "00000000",
                            14 => "00000010",
                            15 => "10001011",
                            16 => "01000001",
                            17 => "00000000",
                            18 => "00000000",
                            19 => "10110100",
                            20 => "01000000",
                            21 => "00000000",
                            22 => "00000000",
                            23 => "10110100",
                            24 => "01000010",
                            25 => "00000001",
                            26 => "01000000",
                            27 => "11111000",
                            28 => "01000110",
                            29 => "00000000",
                            30 => "00000011",
                            31 => "10101010",
                            32 => "01000111",
                            33 => "00000000",
                            34 => "00000011",
                            35 => "10001010",
                            36 => "11100100",
```

```
                              37 => "10000000",
                              38 => "00000000",
                              39 => "11111000",
                              40 => "00000010",
                              41 => "00000000",
                              42 => "00000000",
                              43 => "00010100",
                              44 => "01000011",
                              45 => "10000001",
                              46 => "01000000",
                              47 => "11111000",
                              48 => "00001000",
                              49 => "00000000",
                              50 => "00000001",
                              51 => "10001011",
                              others => "00000000");

begin
    dw : process(write_enable, address, data_in) is
        begin
            if write_enable = '1' then
                ram(to_integer(unsigned(address)) + 0) <= data_in( 7 downto  0);
                ram(to_integer(unsigned(address)) + 1) <= data_in(15 downto  8);
                ram(to_integer(unsigned(address)) + 2) <= data_in(23 downto 16);
                ram(to_integer(unsigned(address)) + 3) <= data_in(31 downto 24);
                ram(to_integer(unsigned(address)) + 4) <= data_in(39 downto 32);
                ram(to_integer(unsigned(address)) + 5) <= data_in(47 downto 40);
                ram(to_integer(unsigned(address)) + 6) <= data_in(55 downto 48);
                ram(to_integer(unsigned(address)) + 7) <= data_in(63 downto 56);
            end if;
    end process dw;
    dr : process(read_enable, address) is
        begin
            if read_enable = '1' then
                data_out( 7 downto  0) <= ram(to_integer(unsigned(address)) + 0);
                data_out(15 downto  8) <= ram(to_integer(unsigned(address)) + 1);
                data_out(23 downto 16) <= ram(to_integer(unsigned(address)) + 2);
                data_out(31 downto 24) <= ram(to_integer(unsigned(address)) + 3);
                data_out(39 downto 32) <= ram(to_integer(unsigned(address)) + 4);
                data_out(47 downto 40) <= ram(to_integer(unsigned(address)) + 5);
                data_out(55 downto 48) <= ram(to_integer(unsigned(address)) + 6);
                data_out(63 downto 56) <= ram(to_integer(unsigned(address)) + 7);
            end if;
    end process dr;
end Behavioral;
```

Explanation

When the write enable (denoted as “write_enable”) is true, it will take the current input (denoted as “data_in”) and set the memory (in variable “ram”) at the address (denoted as “address”) to that value. When the read enable (denoted as “read_enable”) is true, it will take the memory (in variable “ram”) at the address (denoted as “address”) and output that value (through the signal “data_out”).

Output

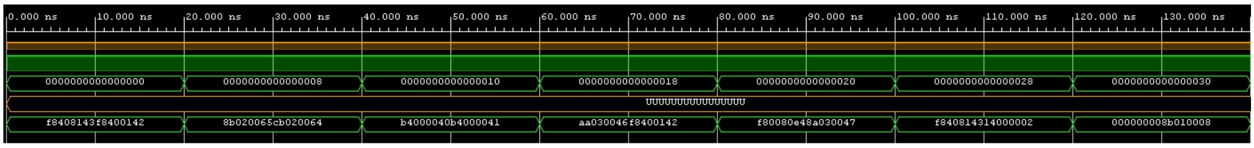

Figure 6: Graph of the output for the Data Memory

Sign Extend

VHDL Code

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity Sign_Extend is
    Port ( INST_In : in STD_LOGIC_VECTOR (31 downto 0);
           IMM_Out : out STD_LOGIC_VECTOR (63 downto 0));
end Sign_Extend;

architecture Behavioral of Sign_Extend is
begin
    process(INST_In)
    begin
        if (INST_In(31 downto 26) = "000101") then -- B
            IMM_Out(31 downto 0) <= INST_In;
            for i in 26 to 63 loop
                IMM_Out(i) <= INST_In(25);
            end loop;
        elsif (INST_In(31 downto 25) = "1011010") then -- CBZ and CBNZ
            IMM_Out(18 downto 0) <= INST_In(23 downto 5);
            for i in 19 to 63 loop
                IMM_Out(i) <= INST_In(23);
            end loop;
        elsif (INST_In(31 downto 21) = "11111000000") or
            (INST_In(31 downto 21) = "11111000010") then -- LOAD and STORE
            IMM_Out(8 downto 0) <= INST_In(20 downto 12);
            for i in 9 to 63 loop
                IMM_Out(i) <= INST_In(20);
            end loop;
        end if;
    end process;
end Behavioral;
```

Explanation

This component takes an instruction in (denoted as "INST_In"), and if it is a "B", "CBZ", "CBNZ", "LDUR", or "STUR" operation, it will extract the Immediate value, sign extend, and output to "IMM_Out".

Output

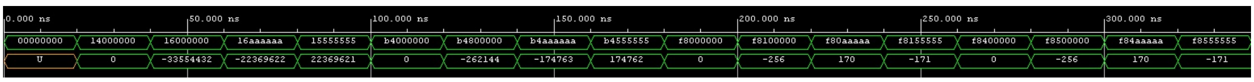

Figure 7: Graph of the output for the Sign Extend