CPU-Lite (Single-Cycle CPU)

Name: Jonathan J. Rodriguez

CSC343 – Computer Organization Lab, Spring 2020

Instructor: Izidor Gertner

The City College of New York

Table of Contents

Objective

The objective of this laboratory exercise is to create a single-cycle CPU-Lite that will work like a regular processor. It will feature 32 4-byte registers, Control Module, Instruction Memory module, several multiplexers, Data Memory, Extended Module, and Program Counter to work like a full-fledged CPU. All instructions will work under one clock-cycle. We will also have to adjust that clock cycle period in order to fit all of the component executions properly. All operations are based on the MIPS architecture.

Board Specifications & Software Used

1. Altera DE2 Board (Quartus II 13.0 sp1)
2. Model Sim

Design Specifications & Functionality

2-To-1 Mux

C:\Users\JRodr\Dropbox\Spring 2020 CCNY\CSC343 - Computer Organization Lab\RODR_2TO1MUX.vhd (/rodr_testbenchcpu/uut/RODR_Mu

```
Ln#
 1  library ieee;
 2  use ieee.std_logic_1164.all;
 3  use ieee.std_logic_arith.all;
 4  use ieee.std_logic_unsigned.all;
 5
 6  entity RODR_2TO1MUX is
 7
 8    generic(nbits: natural);
 9    port( RODR_A: in std_logic_vector(nbits-1 downto 0);
10      RODR_B: in std_logic_vector(nbits-1 downto 0);
11      RODR_SEL : in std_logic;
12      RODR_OUT : Out std_logic_vector(nbits-1 downto 0));
13
14 end RODR_2TO1MUX;
15
16 Architecture LogicFunction of RODR_2TO1MUX is
17 begin
18   process(RODR_SEL, RODR_A, RODR_B)
19   begin
20     case RODR_SEL is
21
22     when '0' =>
23       RODR_OUT <= RODR_A;
24     when others =>
25       RODR_Out <= RODR_B;
26     end case;
27   end process;
28 end LogicFunction;
```

Figure 1: 2-to-1 Mux

In this figure shown above is the 2-to-1 mux that will be used in 4 instances throughout the system (this will be explained later). Within the entity, we take in two inputs **RODR_A** and **RODR_B** along with a select signal that will choose between the two, named **RODR_SEL**. The one selected will then be pushed out of the mux, named **RODR_OUT**.

This mux is used in four places:

1. Program Counter Control (PCC): To decide whether to not increment the PCC by 4 bytes or by4 + 32-bit extended immediate from the instruction register.
2. ALU Control Select: Decide whether or not the ALU will take the data from the second register address or as the 32-bit extended immediate from the instruction register.
3. Register Destination Select: Decide whether to insert the address of the RT or RD register for inputting data based on the ALU functionality.
4. Writing to Register Select : Decide whether to take the first register's data from the register array with the second register's data or the data from the Data Memory Module.

ALU

C:\Users\JRodr\Dropbox\Spring 2020 CCNY\CSC343 - Computer Organization Lab\RODR_ALU.vhd (/rodr_testbenchcpu/uut/RC

```
Ln#
 1  LIBRARY ieee ;
 2  USE ieee.std_logic_1164.all ;
 3  USE ieee.std_logic_unsigned.all;
 4
 5    ENTITY RODR_ALU IS
 6    generic(nbits : natural := 32);
 7    PORT(RODR_IN1    : in std_logic_vector(nbits-1 downto 0);
 8         RODR_IN2    : in std_logic_vector(nbits-1 downto 0);
 9         RODR_ALUCtr : in std_logic_vector (1 downto 0);
10         RODR_OUT    : out std_logic_vector(nbits-1 downto 0));
11    END RODR_ALU ;
```

Figure 2: ALU Entity

Shown above is the ALU ports. There is only one instance of this and will house the necessary operations for all the functionality of the system. It will take in two vectors of 32-bit size, named **RODR_IN1** and **RODR_IN2**. It will also take in a control signal that will tell it when to add, subtract, or do an or immediate function, named **RODR_ALUCtr**. The output will then be the result of **RODR_ALUCtr**, named **RODR_OUT.**

```
13  ARCHITECTURE LogicFunction OF RODR_ALU IS
14
15  signal tempOut : std_logic_vector (nbits -1 downto 0) := (others => '0');
16  Begin
17
18    process(RODR_IN1, RODR_IN2, RODR_ALUCtr, tempOut)
19    begin
20
21    case RODR_ALUCtr is
22      when "00" => -- ADD
23        tempOut <= RODR_IN1 + RODR_IN2;
24      when "01" => -- SUB
25        tempOut <= RODR_IN1 - RODR_IN2;
26      when "10" => -- ORI
27        tempOut <= RODR_IN1 OR RODR_IN2;
28      when others =>
29        tempOut <= (others => '0');
30    end case;
31
32    RODR_Out <= tempOut;
33  end process;
34
35  END LogicFunction;
```

Figure 3: ALU Architecture

The architecture shown above is simple. Based on the control signal **RODR_ALUCTR**, it will then tell us whether to add, subtract, or do an or immediate operation. In any other case, the output will be all 0's. The result will then get outputted from **tempOut** to **RODR_Out**.

Control Module

```
C:\Users\JRodr\Dropbox\Spring 2020 CCNY\CSC343 - Computer Organization Lab\RODR_Control.vhd (/rodr_testbenchcpu/uut/RODR_CTRL) - Default
Ln#
 1  LIBRARY ieee ;
 2  USE ieee.std_logic_1164.all ;
 3  USE ieee.std_logic_unsigned.all;
 4
 5    ENTITY RODR_Control IS
 6    generic(nbits : natural := 32);
 7    PORT(RODR_IN : in std_logic_vector(nbits-1 downto 0);
 8        RODR_Zero : in std_logic_vector (nbits -1 downto 0);
 9        RODR_ALUCtr    : out std_logic_vector (1 downto 0); -- Control (ADD, SUB, ORI)
10        RODR_MemWr     : out std_logic; -- Control deciding when to write into a register or not
11        RODR_MemtoReg  : out std_logic; -- Control Deciding when to take data from Data Memory into register
12        RODR_RegWr     : out std_logic; -- Control deciding whether to write or not to a register
13        RODR_ExtOp     : out std_logic; -- Control deciding whether to zero extend or sign extend
14        RODR_ALUSrc    : out std_logic; -- Control Deciding between busB from RegAr OR immExtend32
15        RODR_RegDst   : out std_logic; -- Control Deciding between RT and RD register
16        RODR_PCSrc     : out std_logic);
17    END RODR_Control ;
18
```

Figure 4: Control Entity

Shown above is the control operation that will provide for all the functionality for all modules in the system. It will take in the information from the Instruction Memory module and take the opcodes/FUNCT and decide what flags to output for their right execution.

```
19  ARCHITECTURE LogicFunction OF RODR_Control IS
20
21    signal RODR_OPCODE : std_logic_vector(5 downto 0);
22    signal RODR_RS : std_logic_vector(4 downto 0);
23    signal RODR_RT: std_logic_vector(4 downto 0);
24    signal RODR_RD: std_logic_vector(4 downto 0);
25    signal RODR_SHAMT: std_logic_vector(4 downto 0);
26    signal RODR_FUNCT: std_logic_vector(5 downto 0);
27    signal RODR_IMMEDIATE : std_logic_vector(15 downto 0);
28  Begin
29    RODR_OPCODE <= RODR_IN(nbits-1 downto nbits-6);
30    RODR_RS <= RODR_IN(nbits-7 downto nbits-11);
31    RODR_RT <= RODR_IN(nbits-12 downto nbits-16);
32    RODR_RD <= RODR_IN(nbits-17 downto nbits-21);
33    RODR_SHAMT <= RODR_IN(nbits-22 downto nbits-26);
34    RODR_FUNCT <= RODR_IN(nbits-27 downto nbits-32);
35    RODR_IMMEDIATE <= RODR_IN(nbits-17 downto nbits-32);
```

Figure 5: Control Signals

Shown above are the control sub signals that are used that parse the input, **RODR_IN**. The Opcode portion is the first 6 bits of the input and supports functionality for the Immediate instructions sets. RS, RT, and RD are the addresses to specific registers within the register file. **RODR_SHAMT** is used for determining the shift amount for a register. **RODR_FUNCT** is used to support functionality for the register-to-register instruction sets. **RODR_Immediate** are the last 16 bits and is used for Immediate instruction sets.

```
37  process(RODR_OPCODE, RODR_RS, RODR_RT, RODR_RD, RODR_SHAMT, RODR_FUNCT, RODR_IMMEDIATE, RODR_Zero)
38  variable ALUctr : std_logic_vector (1 downto 0) := (others => '0');
39  variable MemWr, MemtoReg, RegWr, ExtOp, ALUSrc, RegDst, PCSrc : std_logic := '0';
40  begin
41    if(RODR_OPCODE = "000000") then
42      case RODR_FUNCT is
43        when "100000" => -- ADD (20 HEX)
44          ALUctr := "00";
45          MemWr := '0';
46          MemtoReg := '0';
47          RegWr := '1';
48          ExtOp := '0';
49          ALUSrc := '0';
50          RegDst := '1';
51          PCSrc := '0';
52        when "100010" => -- SUB (22 HEX)
53          ALUctr := "01";
54          MemWr := '0';
55          MemtoReg := '0';
56          RegWr := '1';
57          ExtOp := '0';
58          ALUSrc := '0';
59          RegDst := '1';
60          PCSrc := '0';
```

Figure 6: Control Process

The control will help with understanding which operation will be done first. Because the opcode shows all 0s, we are then using the register-to-register instruction sets. This will house the **ADD** and **SUB** instructions. **ALUCtr** was specified in *figure 3* and is a two-bit instruction that controls whether it will add (“00”), sub (“01”), or ORI (“10”). **MemWr** will control whether we will be writing to the Data Memory module (‘1’) or not (‘0’). **MemtoReg** will go to a mux and will determine whether we will be extracting the data from the output of the ALU or from the Data Memory module. **RegWr** is a control signal that will determine whether the input to the register array will write to the register specified as the destination. **ExtOp** will control whether to zero-extend or sign-extend the 16-bit immediate value into 32-bits. **ALUSrc** goes to a mux and

controls whether to take the second data signal from the register array or the immediate value. **RegDst** determines to either take the destination address that goes into the register array as the RT or RD register. **PCSrc** will go to another mux for the program counter and will determine whether or not the program counter will increment by 4 bytes or by 4 + the immediate number.

```
61          when others =>
62            ALUctr := "11";
63            MemWr := '0';
64            MemtoReg := '0';
65            RegWr := '0';
66            ExtOp := '0';
67            ALUSrc := '0';
68            RegDst := '0';
69            PCSrc := '0';
70        end case;
71      else
72        case RODR_OPCODE is
73          when "001101" => -- ori (13 HEX)
74            ALUctr := "10";
75            MemWr := '0';
76            MemtoReg := '0';
77            RegWr := '1';
78            ExtOp := '1';
79            ALUSrc := '1';
80            RegDst := '0';
81            PCSrc := '0';
82          when "100011" => -- lw (23 HEX)
83            ALUctr := "00";
84            MemWr := '0';
85            MemtoReg := '1';
86            RegWr := '1';
87            ExtOp := '1';
88            ALUSrc := '1';
```

Figure 7: Control Process 2

In case the opcode does not equal all 0s, it will then start the Immediate instruction set controls. Here we do or immediate, load word, store word, branch not equal (BNE) and branch equal (BEQ).

```
89          RegDst := '0';
90          PCSrc := '0';
91        when "101010" => -- sw (2C HEX)
92          ALUctr := "00";
93          MemWr := '1';
94          MemtoReg := '0';
95          RegWr := '0';
96          ExtOp := '1';
97          ALUSrc := '1';
98          RegDst := '0';
99          PCSrc := '0';
100       when "000100" => -- BEQ (4 HEX)
101         ALUctr := "01";
102         MemWr := '0';
103         MemtoReg := '0';
104         RegWr := '0';
105         ExtOp := '0';
106         ALUSrc := '0';
107         RegDst := '0';
108         if(RODR_Zero = "00000000000000000000000000000000") then
109           PCSrc := '1';
110         else
111           PCSrc := '0';
112         end if;
113       when "000101" => -- BNE (5 HEX)
114         ALUctr := "01";
115         MemWr := '0';
116         MemtoReg := '0';
```

Figure 8: Control Process 3

```
116         MemtoReg := '0';
117         RegWr := '0';
118         ExtOp := '0';
119         ALUSrc := '0';
120         RegDst := '0';
121         if(RODR_Zero = "00000000000000000000000000000000") then
122           PCSrc := '1';
123         else
124           PCSrc := '0';
125         end if;
126       when others =>
127         ALUctr := "11";
128         MemWr := '0';
129         MemtoReg := '0';
130         RegWr := '0';
131         ExtOp := '0';
132         ALUSrc := '0';
133         RegDst := '0';
134         PCSrc := '0';
135     end case;
136   end if;
137   RODR_ALUctr <= ALUCtr;
138   RODR_MemWr <= MemWr;
139   RODR_MemtoReg <= MemtoReg;
140   RODR_RegWr <= RegWr;
```

Figure 9: Control Process 4

```
126         when others =>
127           ALUctr := "11";
128           MemWr := '0';
129           MemtoReg := '0';
130           RegWr := '0';
131           ExtOp := '0';
132           ALUSrc := '0';
133           RegDst := '0';
134           PCSrc := '0';
135       end case;
136     end if;
137     RODR_ALUctr <= ALUCtr;
138     RODR_MemWr <= MemWr;
139     RODR_MemtoReg <= MemtoReg;
140     RODR_RegWr <= RegWr;
141     RODR_ExtOp <= ExtOp;
142     RODR_ALUSrc <= AL/rodr_testbenchcpu/uut/ExtOp
143     RODR_RegDst <= R0
144     RODR_PCSrc <= PCSrc;
145   end process;
146
147   END LogicFunction;
```

Figure 10: Control Process 5

At the very end we put these signals from the process into the output of the control module to be sent out to various places in the CPU system.

Immediate Extender

```
1   library ieee;
2   use ieee.std_logic_1164.all;
3   use ieee.numeric_std.all;
4
5   entity RODR_Extender is
6
7     generic(nbits: natural := 16);
8     port( RODR_IN: in std_logic_vector(nbits-1 downto 0);
9       RODR_SEL: in std_logic;
10      RODR_OUT : Out std_logic_vector(nbits*2-1 downto 0));
11
12  end RODR_Extender;
13
14  architecture LogicFunction of RODR_Extender is
15  begin
16
17  process(RODR_IN, RODR_SEL)
18  begin
19    if(RODR_SEL = '0') then  -- OR IMMEDIATE
20      RODR_Out <= std_logic_vector(resize(unsigned(RODR_IN), RODR_OUT'length));
21    else          -- LOAD and STORE Instructions
22      RODR_Out <= std_logic_vector(resize(signed(RODR_IN), RODR_OUT'length));
23    end if;
24  end process;
25  end LogicFunction;
```

Figure 11: Immediate Extender Code

Shown above is the immediate extender. It will always do this process, but its output will only be used when doing immediate operations and **ALUSrc** = ‘1’. It takes the input from the Instruction Memory module’s last 16-bits. **RODR_SEL** will come from the **ExtOp** control signal and will determine whether to extend the 16-bits to 32-bit unsigned or signed. **RODR_OUT** will then take that selection as its output.

Instruction Memory Module

```
1   LIBRARY ieee ;
2   USE ieee.std_logic_1164.all ;
3   USE ieee.numeric_std.all;
4
5     ENTITY RODR_InstrMem IS
6     generic(nbits : natural := 32);
7     PORT(RODR_PCSource : in std_logic_vector (nbits-1 downto 0);
8         RODR_CLK : in std_logic;
9         RODR_Out : out std_logic_vector (nbits-1 downto 0));
10    END RODR_InstrMem ;
11
```

Figure 12: Instruction Memory Module

The instruction memory module has the instruction register operations that will be sent to both the control module, register array, and the extender module. **RODR_PCSource** comes from the mux that controls whether to use the normal program counter signal or to branch to a different instruction within the instruction module. **RODR_CLK** is then used to control this operation and will then send its output using **RODR_Out**.

```
12  ARCHITECTURE LogicFunction OF RODR_InstrMem IS
13
14  type RODR_Memory is array (0 to 31) of std_logic_vector(nbits-1 downto 0);
15
16  signal RODR_InstructionMem : RODR_Memory := (
17    "00000000000000001000100000100000",  -- Add $r2, $r0, $r1
18    "00000000000000001000100000100010",  -- Sub $r2, $r0, $r1
19    "00110100000000001000000000000010",  -- ori $r1, $r0, 1
20    "10001100000000001000000000000001",  -- lw $r1, $r0, 1
21    "10101000000000001000000000000001",  -- sw $r1, $r0, 1
22    "00010000000000001000000000000001",  -- BEQ $r0, $r1, 1
23    "00010100000000001000000000000001",   -- BNE $r0, $r1, 1
24    others=> (others=>'0'));
25  Begin
26
27  process(RODR_PCSource, RODR_CLK)
28  begin
29    if(Rising_Edge(RODR_CLK)) then
30      RODR_Out <= RODR_InstructionMem(to_integer(unsigned(RODR_PCSource(2 downto 0))));
31    end if;
32  end process;
33
34
35  END LogicFunction;
```

Figure 13: Instruction Memory Architecture

Shown above is the architecture. In order to house the instruction register output, we have to create a new type called **RODR_Memory**. This will be 32 slots long and will each have 32-bits to be assigned within them. Once that has been created, we create a new signal called **RODR_InstructionMem** with this type and start to store instruction sets that we want to run throughout the program. We will then run a process to determine which one of those instructions will be executed based on the program counter sent within this module.

Program Counter Module

```
1   LIBRARY ieee ;
2   USE ieee.std_logic_1164.all ;
3   USE ieee.std_logic_unsigned.all;
4
5     ENTITY RODR_PC IS
6     generic(nbits : natural := 32);
7     PORT(RODR_IN  : in std_logic_vector (nbits-1 downto 0);
8         RODR_CLK : in std_logic;
9         RODR_OUT : out std_logic_vector (nbits-1 downto 0));
10    END RODR_PC ;
11
12  ARCHITECTURE LogicFunction OF RODR_PC IS
13
14    begin
15
16    process(RODR_IN, RODR_CLK)
17    variable PCNext : std_logic_vector(nbits-1 downto 0) := (others => '0');
18    begin
19      if(Rising_Edge(RODR_CLK)) then
20          PCNext := RODR_IN;
21          RODR_Out <= PCNext;
22      end if;
23    end process;
24
25  END LogicFunction;
```

Figure 14: Program Counter

Shown above is the program counter module. This will be used to control the instructions in the Instruction Memory Module. **RODR_IN** will take in signals from a mux that will determine whether the program counter goes up by 4 or 4 + the immediate value (will be shown later). **RODR_CLK** is the clock that will control when the program counter changes. **RODR_Out** will then be the output of the new program counter value. The architecture is very simple as the process only stores the inputted value and then outputs it when the rising edge of the clock comes in.

Register Array Module

```
1   LIBRARY ieee ;
2   USE ieee.std_logic_1164.all;
3   USE ieee.std_logic_unsigned.all;
4   use IEEE.numeric_std.ALL;
5
6     ENTITY RODR_RegAr IS
7     generic(nbits : natural := 32);
8     PORT(RODR_SrcAddr    : IN STD_LOGIC_VECTOR (4 downto 0);
9         RODR_SrcAddr2   : IN STD_Logic_Vector (4 downto 0);
10        RODR_DstAddr    : IN STD_Logic_vector (4 downto 0);
11        RODR_Data       : in std_logic_vector (31 downto 0);
12        RODR_RegWr      : in std_logic;
13        RODR_Clock      : in std_logic;
14         RODR_OUT1       : OUT STD_LOGIC_VECTOR (nbits-1 downto 0);
15         RODR_OUT2       : OUT STD_LOGIC_VECTOR (nbits-1 downto 0));
16  END RODR_RegAr ;
17
```

Figure 15: Register Array Module

The register array module controls all 32 registers that can be used to input and output from to be used for computation in the ALU. It will take in two source addresses (**RODR_SrcAddr** & **RODR_SrcAddr2**), and one destination address (**RODR_DstAddr)**. It will also take in data once the ALU computation has completed and goes through the last mux (**RODR_Data)**. **RODR_RegWr** is a control signal from the Control Module that will determine if the destination register will be written to or not. **RODR_Clock** will be used to output the correct data through **RODR_OUT1** and **RODR_OUT2**.

```
18  ARCHITECTURE LogicFunction OF RODR_RegAr IS
19  type RODR_RegInput is array (0 to 31) of std_logic_vector(nbits-1 downto 0);
20  signal r : RODR_RegInput := ("00000000000000000000000000000011", "00000000000000000000000000000010","00000000000000000000000000000000", ot
21  begin
22
23  process(RODR_SrcAddr, RODR_SrcAddr2, RODR_DstAddr, RODR_RegWr, RODR_Clock, RODR_Data)
24  variable srcreg1, srcreg2, destreg : std_logic_vector (4 downto 0) := (others => '0');
25  variable Data : std_logic_vector (31 downto 0) := (others => '0');
26  variable regwr : std_logic := '0';
27  Begin
28    srcreg1 := RODR_SrcAddr;
29    srcreg2 := RODR_SrcAddr2;
30    destreg := RODR_DstAddr;
31    regwr := RODR_RegWr;
32    Data := RODR_Data;
33
34    if(rising_edge(RODR_Clock)) then
35      RODR_OUT1 <= r(to_integer(unsigned(srcreg1)));
36      RODR_OUT2 <= r(to_integer(unsigned(srcreg2)));
37      case regwr is
38        When '0' =>
39        When '1' =>
40          r(to_integer(unsigned(destreg))) <= Data;
41        When others =>
42      end case;
43    end if;
44  end process;
45  END LogicFunction;
```

Figure 16: Register Array Architecture

All the register array modules will then go through the architecture shown above. We also have created a new type here called **RODR_RegInput** that will store and retrieve data from this register array. We then create a signal **r** that will use that data type to initialize those registers. Inside the process, we simply output the data from the registers selected as the two source addresses. However, if the **RODR_RegWr** control signal is ‘1’, we have to write from the input **RODR_Data** into the destination register.

Master File

```
1   LIBRARY ieee ;
2   USE ieee.std_logic_1164.all ;
3   USE ieee.std_logic_unsigned.all;
4
5   ENTITY RODR_MasterFile IS
6     generic(nbits : natural := 32);
7     PORT(RODR_Clock     : in std_logic;
8         RODR_ALUResult : out std_logic_vector(nbits-1 downto 0);
9         RODR_PCOutput  : out std_logic_vector(nbits-1 downto 0);
10        RODR_RegResult : out std_logic_vector(nbits-1 downto 0));
11    END RODR_MasterFile ;
12
```

Figure 17: Master File

The master file will have all the components before and contain all the connections between them. For porting, we just control the clock signal (**RODR_Clock**) which will be done in our testbench file. The output will then be the ALU result (**RODR_ALUResult**), the Program Counter output (**RODR_PCOutput**), and the Register result that goes into the Register Array Module (**RODR_RegResult**).

```
13  ARCHITECTURE LogicFunction OF RODR_MasterFile IS
14
15    Component RODR_2TO1MUX is
16      generic(nbits: natural := 32);
17      port( RODR_A: in std_logic_vector(nbits-1 downto 0);
18        RODR_B: in std_logic_vector(nbits-1 downto 0);
19        RODR_SEL : in std_logic;
20        RODR_OUT : Out std_logic_vector(nbits-1 downto 0));
21
22    end Component;
23
24    Component RODR_InstrMem IS
25      generic(nbits : natural := 32);
26      PORT(RODR_PCSource : in std_logic_vector (nbits-1 downto 0);
27          RODR_CLK : in std_logic;
28           RODR_Out : out std_logic_vector (nbits-1 downto 0));
29    END Component;
30
31    Component RODR_PC IS
32      generic(nbits : natural := 32);
33      PORT(RODR_IN  : in std_logic_vector (nbits-1 downto 0);
34          RODR_CLK : in std_logic;
35           RODR_OUT : out std_logic_vector (nbits-1 downto 0));
36    END Component;
37
```

Figure 18: Components within the Master File

Shown above are all the components previously created based on the other sections.

```
38    Component RODR_Control IS
39      generic(nbits : natural := 32);
40      PORT(RODR_IN : in std_logic_vector(nbits-1 downto 0);
41        RODR_Zero : in std_logic_vector (nbits -1 downto 0);
42        RODR_ALUCtr    : out std_logic_vector (1 downto 0); -- Control (ADD, SUB, ORI)
43        RODR_MemWr     : out std_logic; -- Control deciding when to write into a register or not
44        RODR_MemtoReg  : out std_logic; -- Control Deciding when to take data from Data Memory into register
45        RODR_RegWr     : out std_logic; -- Control deciding whether to write or not to a register
46        RODR_ExtOp     : out std_logic; -- Control deciding whether to zero extend or sign extend
47        RODR_ALUSrc    : out std_logic; -- Control Deciding between busB from RegAr OR immExtend32
48        RODR_RegDst   : out std_logic; -- Control Deciding between RT and RD register
49        RODR_PCSrc     : out std_logic);
50    END Component;
51
52    Component RODR_ALU IS
53    generic(nbits : natural := 32);
54    PORT(RODR_IN1    : in std_logic_vector(nbits-1 downto 0);
55        RODR_IN2    : in std_logic_vector(nbits-1 downto 0);
56        RODR_ALUCtr : in std_logic_vector (1 downto 0);
57        RODR_OUT    : out std_logic_vector(nbits-1 downto 0));
58    END Component ;
59
60    Component RODR_DataMem IS
61      generic(nbits : natural := 32);
62      PORT(RODR_WrEN    : in std_logic;
63        RODR_Addr    : in std_logic_vector(nbits-1 downto 0);
64        RODR_DataIN  : in std_logic_vector(nbits-1 downto 0);
65        RODR_Clock   : in std_logic;
66        RODR_DataOUT : out std_logic_vector(nbits-1 downto 0));
67    END Component ;
```

Figure 19: Components within the Master File

```
69      Component RODR_RegAr IS
70      generic(nbits : natural := 32);
71      PORT(RODR_SrcAddr     : IN STD_LOGIC_VECTOR (4 downto 0);
72          RODR_SrcAddr2    : IN STD_Logic_Vector (4 downto 0);
73          RODR_DstAddr     : IN STD_Logic_vector (4 downto 0);
74          RODR_Data        : in std_logic_vector (31 downto 0);
75          RODR_RegWr       : in std_logic;
76          RODR_Clock       : in std_logic;
77           RODR_OUT1        : OUT STD_LOGIC_VECTOR (nbits-1 downto 0);
78           RODR_OUT2        : OUT STD_LOGIC_VECTOR (nbits-1 downto 0));
79      END Component;
80
81      Component RODR_Extender is
82      generic(nbits: natural := 16);
83      port( RODR_IN: in std_logic_vector(nbits-1 downto 0);
84        RODR_SEL: in std_logic;
85        RODR_OUT : Out std_logic_vector(nbits*2-1 downto 0));
86
87      end Component;
88
89      signal ProgramCounter : std_logic_vector (31 downto 0) := (Others => '0');
90      signal PC1 : std_logic_vector (31 downto 0) := (others => '0');
91      signal PC2 : std_logic_vector (31 downto 0) := (others => '0');
92      signal PCOut : std_logic_vector (31 downto 0) := (others => '0');
93
94      signal InstMemOut : std_logic_vector(31 downto 0) := (others => '0');
95
96      signal RDAddress : std_logic_vector(4 downto 0) := (others => '0');
97      signal RTAddress : std_logic_vector(4 downto 0) := (others => '0');
98      signal RSAddress : std_logic_vector(4 downto 0) := (others => '0');
```

Figure 20: Components within Master File + signals

Shown here is more components. We then show all the signals that go into these components. We create them here as they will be used when we start to port map all the components and connect them properly.

```
99      signal RegDst : std_logic   := '0';
100     signal PCSrc : std_logic := '0';
101     signal RegDestAddr : std_logic_vector(4 downto 0) := (others => '0');
102     signal busW : std_logic_vector (31 downto 0) := (others => '0');
103     signal RegWr : std_logic := '0';
104     signal busA : std_logic_vector(31 downto 0) := (others => '0');
105     signal busB : std_logic_vector(31 downto 0) := (others => '0');
106
107     signal ImmExtend32 : std_logic_vector(31 downto 0) := (others => '0');
108     signal ALUSrc : std_logic := '0';
109     signal ALUbusB : std_logic_vector (31 downto 0) := (others => '0');
110
111     signal Imm16 : std_logic_vector(15 downto 0) := (others => '0');
112     signal ExtOp : std_logic := '0';
113
114     signal ALUCtr : std_logic_vector (1 downto 0) := (others => '0');
115     signal ALUOut : std_logic_vector (31 downto 0) := (others => '0');
116
117     signal MemWr : std_logic := '0';
118     signal DataMemOut : std_logic_vector(31 downto 0) := (others => '0');
119
120     signal MemtoReg : std_logic := '0';
121
122     Begin
123
124     Imm16 <= InstMemOut(15 downto 0);
125
126     RDAddress <= InstMemOut(nbits-17 downto nbits-21);
127     RTAddress <= InstMemOut(nbits-12 downto nbits-16);
128     RSAddress <= InstMemOut(nbits-7 downto nbits-11);
```

Figure 21: More Signals in Master File + Signal Addressing

We have more signals above for port mapping. After the “Begin” keyword, we start to initialize some of them. For **Imm16**, we take the last 15 bits from the instruction memory module (signal called **InstMemOut**). We then start to parse out the RD, RT, and RS addresses from that signal as well and put them into their own respective signals.

```
129
130     RODR_PCMux    : RODR_2TO1MUX      PORT MAP (PC1, PC2, PCSrc, ProgramCounter);
131     RODR_ProC     : RODR_PC        PORT MAP (ProgramCounter, RODR_Clock, PCOut);
132
133     RODR_InstMem     : RODR_InstrMem                PORT MAP (PCOut, RODR_Clock, InstMemOut);
134     RODR_CTRL     : RODR_Control      PORT MAP (InstMemOut, ALUOut, ALUCtr, MemWr, MemtoReg, RegWr, ExtOp, ALUSrc, RegDst, PCSrc);
135
136     RODR_MuxRegDest : RODR_2TO1MUX  GENERIC MAP (5) PORT MAP (RDAddress, RTAddress, RegDst, RegDestAddr);
137     RODR_RegArray   : RODR_RegAR                    PORT MAP (RSAddress, RTAddress, RegDestAddr, busW, RegWr, RODR_Clock, busA, busB);
138
139
140     RODR_MuxExtSEL  : RODR_2TO1MUX                  PORT MAP (busB, ImmExtend32, ALUSrc, ALUbusB);
141     RODR_Extend     : RODR_Extender                 PORT MAP (Imm16, ExtOp, ImmExtend32);
142
143     RODR_ALU1       : RODR_ALU                      PORT MAP (busA, ALUbusB, ALUCtr, ALUOut);
144
145     RODR_DatMem     : RODR_DataMem                  PORT MAP (MemWr, ALUOut, busB , RODR_Clock, DataMemOut);
146
147     RODR_MuxEnd     : RODR_2TO1MUX                  PORT MAP (ALUOut, DataMemOut, MemtoReg, busW);
148
149     process(PC1, PC2, PCOut, ImmExtend32)
150     begin
151       PC1 <= PCOut + 1;
152       PC2 <= PC1 + ImmExtend32;
153     end process;
154
155     RODR_ALUResult <= ALUOut;
156     RODR_PCOutput <= PCOut;
157     RODR_RegResult <= busW;
158 end LogicFunction;
```

Figure 22: Port Map + Processes within Master File

Shown above is the port mapping structure for all the components. We start with **RODR_PCMux**, which is the controller to the program counter. **PC1** and **PC2** are the selections to either increment the counter by 4 or 4 + the immediate value. It uses **PCSrc** to decide and then output using **ProgramCounter**. This then goes into the program counter component to get stored and outputted. **RODR_CTRL** will then take the instruction from the instruction memory module and start outputting a bunch of flags towards other components within the system. The register array will then get a mux for its destination address as we use **RegDst** to determine whether the destination register either goes to RD or RT register. We then have **RODR_MuxExtSEL** that will control the second signal the ALU will take in (second data from register array or immediate value). We then also have a data memory module, called **RODR_DatMem** that will be used for loading and storing data using the address provided into it with the **MemWr** = ‘1’. Lastly, we have a mux that will either take the result from the ALU or the data from Data Memory Module.

Testbench

```
1   LIBRARY ieee ;
2   USE ieee.std_logic_1164.all ;
3   USE ieee.std_logic_unsigned.all;
4
5     ENTITY RODR_TestbenchCPU IS
6        generic(N : natural := 32);
7     END RODR_TestbenchCPU ;
```

Figure 23: Testbench

Shown above is the testbench entity portion. We do not need any ports as we are using this to test our code. We only have a generic value of 32 that will be passed out to ensure that everything we are using is 32-bits long.

```
9   ARCHITECTURE LogicFunction OF RODR_TestbenchCPU IS
10    signal Clock : std_logic := '0';
11    signal ALUResult : std_logic_vector(N-1 downto 0) := (others => '0');
12    signal PCOutput : std_logic_vector(N-1 downto 0) := (others => '0');
13    signal RegResult: std_logic_vector(N-1 downto 0) := (others => '0');
14
15
16    Component RODR_MasterFile IS
17     generic(nbits : natural := 32);
18     PORT(RODR_Clock    : in std_logic;
19        RODR_ALUResult : out std_logic_vector(nbits-1 downto 0);
20        RODR_PCOutput  : out std_logic_vector(nbits-1 downto 0);
21        RODR_RegResult : out std_logic_vector(nbits-1 downto 0));
22    END Component;
23  Begin
24
25    uut : RODR_MasterFile PORT MAP (Clock, ALUResult, PCOutput, RegResult);
26
27    process
28    begin
29        Wait for 1 ns;
30        Clock <= '1';
31        wait for 1 ns;
32        Clock <= '0';
33  end process;
34
35  END LogicFunction;
```

Figure 24: Testbench Architecture

Shown above is the testbench architecture. We will be controlling the clock using this file. The outputs that will be shown in simulation is the **Clock**, **ALUResult**, **PCOutput**, and **RegResult**. We then call only one component, the **RODR_MasterFile** that has all our components put together into one. We then port map that in this file. In the end, we start a process that will only clock the process and continue to do so until we stop the program.

Conclusion

In this laboratory exercise, I have learned how to create a single-cycle CPU. We have done this by taking the Opcode/FUNCT portion of the instruction memory and using the control module that will decide the operations of all the other modules in the system. They all work under one clock signal which is amazing as there are so many signals that need to be processed

before the next rising edge comes. I learned that with the use of the program counter, it will change between the add, sub, ori, lw, sw, BNE, and BEQ. The addition will add the two data that goes inside the ALU and add them. Subtraction will then go inside the ALU and subtract them. Or Immediate then takes the value from the RS register and the immediate value and the output will be done via an OR gate. Loading and storing words shows how to input and output from the Data Memory Module. Lastly, Branch equal and Branch not equal will move to a different location using the program counter and increment it to the next instruction. Overall, this experiment was amazing, and I hope to take this information into the future.