Exercise 1

Exercise 6.10 (Harris and Harris Textbook, Chapter 6, Exercise 10)

add $t0, $s0, $s1
lw $t0, 0x20($t7)
addi $s0, $0, −10

Decoded add Instruction:
000000 10000 10001 01000 00000 100000
hex: 0x02114020

Decoded lw Instructions:
100011 10001 01000 0000000000010100
hex: 0x8e280014

Decoded addi Instruction:
001000 00000 10000 1111111111110110
hex: 0x2010fff6

Exercise 6.12 (Harris and Harris Textbook, Chapter 6, Exercise 12)
a) which instructions from Exercise 6.10 are I-type instructions?
addi
lw

b) Sign-extend the 16-bit immediate of each instruction from part (a) so that it becomes a 32-bit immediate.
addi: 11111111111111111111111111110110
lw: 00000000000000000000000000010100

Exercise 6.14 (Harris and Harris Textbook, Chapter 6, Exercise 14)
0x20080000
binary: 001000 00000 01000 0000000000000000
instruction: addi $t0, $0, 0

0x20090001
binary: 001000 00000 01001 0000000000000001
instruction: addi $t1, $0, 1

0x0089502a
binary: 000000 00100 01001 01010 00000 101010
instruction: slt $t2, $a0, $t1

0x15400003
binary: 000101 01010 00000 0000000000000011
instruction: bne $t2, $0, 3

0x01094020
binary: 000000 01000 01001 01000 00000 100000
instruction: add $t0, $t0, $t1

0x21290002
binary: 001000 01001 01001 0000000000000010
instruction: addi $t1, $t1, 2

0x01001020
binary: 000000 01000 00000 00010 00000 100000
instruction: add $v0, $t0, $0

The code above
1. initializes $t0 to 0
2. initializes $t1 to 1
3. compares $a0 to $t1
4. if $a0 is less than $t1, then it jumps to the next instruction
5. if $a0 is greater than or equal to $t1, then it adds $t0 and $t1
6. increments $t1 by 2
7. copies the value of $t0 to $v0

Give three examples from the MIPS architecture of each of the architecture design principles: (1) simplicity favors regularity; (2) make the common case fast; (3) smaller is faster; and (4) good design demands good compromises. Explain how each of your examples exhibits the design principle.

1. simplicity favors regularity
each instruction is 32 bits long and has a specific format. This makes decoding instructions easier and faster. The 32-bit instruction format is also used for the data memory, which makes it easier to access data.

There are only 3 types of instructions: R-type, I-type, and J-type. This makes decoding simpler

Consistent numbers of operands for each instruction type. This makes decoding simpler and hardware design easier.

2. make the common case fast
dedicated register set makes it faster to access registers.

The MIPS architecture is made up os simple and small instructions. This makes it faster to execute instructions.

The simple architecture makes it easier to design a fast pipeline as well as as simpler decoding unit.

3. smaller is faster
The MIPS architecture is made up of simple and small instructions. This means less bits are required to represent each instruction.

The number of registers is small.

the combination of a small register set, and memory makes it easier to design and access large amounts of data.

4. good design demands good compromises MIPS provide support to access data in memory and not just registers. This makes it easier to access large amounts of data, but it also makes it slower.

MIPS is a very simple architecture. This makes it easier to design and implement, but it also makes more complicated designs difficult to implement and requires more smaller instructions to be executed.

MIPS also has a three-format instruction set. This makes decoding instructions slightly more complex than a single format instruction set, but it also makes it more flexible.

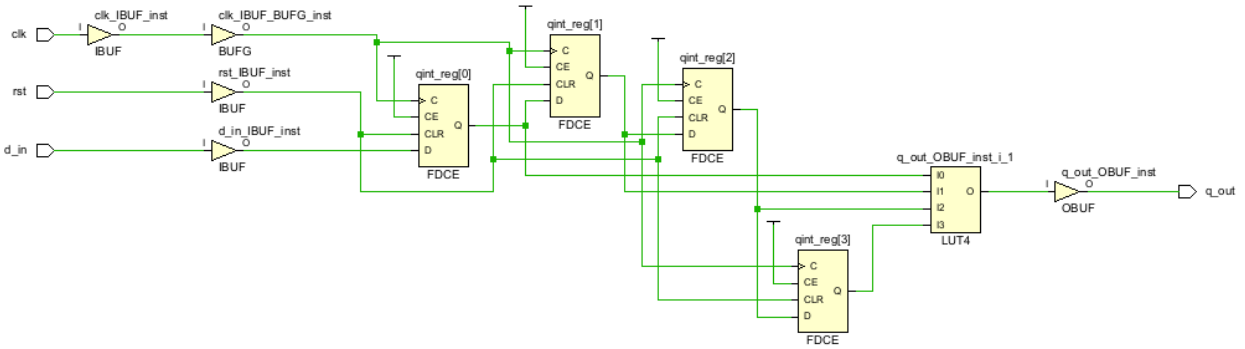
Exercise 2

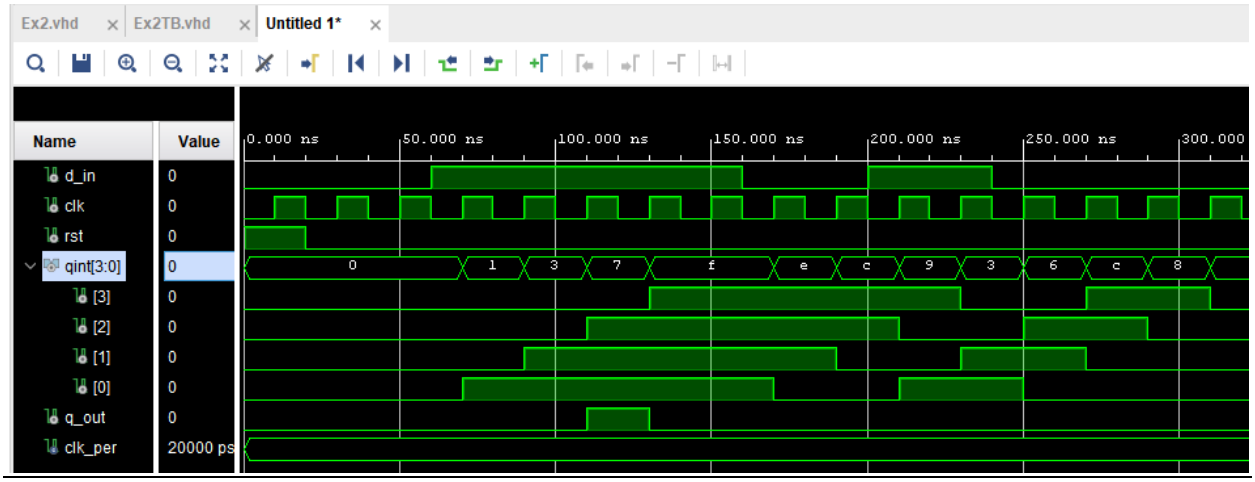
```
library library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity Ex2 is
    port (
        d_in : in std_logic;
        q_out : out std_logic
        clk : in std_logic
        rst : in std_logic
    );
end Ex2;

architecture Behavioral of Ex2 is
    signal qint : std_logic_vector(3 downto 0);

begin

    internal : process (clk, rst)
```

```
        begin
            if (rst = '1') then
                qint <= (others => '0');
            elsif (rising_edge(clk)) then
                qint <= qint(2 downto 0) & d_in;
            end if;
        end process;

    output : process (qint)
        begin
            if (qint = "0111") then
                q_out <= '1';
            else
                q_out <= '0';
            end if;
        end process;

end Behavioral;
```

```
library library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity Ex2TB is
end Ex2TB;

architecture Behavioral of Ex2TB is

    signal d_in, clk, rst : std_logic := '0';
    signal q_out : std_logic;
    constant clk_per : time := 20 ns;

begin

    uut : entity work.Ex2
        port map (
            d_in => d_in,
            clk => clk,
            rst => rst,
            q_out => q_out
        );

    clk <= not clk after clk_per/2;

    stimuli : process
```

```
    begin
        rst <= '1';
        wait until clk = '0';
        assert q_out = '0' report "error1, q_out is " & std_logic'image(q_out) &
" but expected to be 0";

        rst <= '0';
        d_in <= '0';
        wait for 2*clk_per;
        assert q_out = '0' report "error2, q_out is " & std_logic'image(q_out) &
" but expected to be 0";

        d_in <= '1';
        wait for 3*clk_per;
        assert q_out = '1' report "error3, q_out is " & std_logic'image(q_out) &
" but expected to be 1";

        wait for 2*clk_per;
        assert q_out = '0' report "error4, q_out is " & std_logic'image(q_out) &
" but expected to be 0";

        d_in <= '0';
        wait for 2*clk_per;
        assert q_out = '0' report "error5, q_out is " & std_logic'image(q_out) &
" but expected to be 0";

        d_in <= '1';
        wait for 2*clk_per;
        assert q_out = '0' report "error6, q_out is " & std_logic'image(q_out) &
" but expected to be 0";

        d_in <= '0';
        wait for 2*clk_per;
        assert q_out = '0' report "error7, q_out is " & std_logic'image(q_out) &
" but expected to be 0";

        wait;
    end process stimuli;

end Behavioral;
```





Exercise 3

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity Ex3 is
    port (
        sw : in std_logic_vector(1 downto 0);
        bulb : out std_logic;
        sw_out : out std_logic_vector(1 downto 0)
    );

end Ex3;
```

```
architecture Behavioral of Ex3 is
    signal sw_not : std_logic_vector(1 downto 0);
    signal and_int : std_logic_vector(1 downto 0);

begin

    sw_not <= not sw;
    and_int(1) <= sw(0) and sw_not(1);
    and_int(0) <= sw(1) and sw_not(0);
    bulb <= and_int(0) or and_int(1);
    sw_out <= sw;

end Behavioral;
```

```
library ieee;
use ieee.std_logic_1164.all;

entity Ex3TB is
end Ex3TB;

architecture Behavioral of Ex3TB is

    signal sw : std_logic_vector(1 downto 0) := (others => '0');
    signal bulb : std_logic;
    signal sw_out : std_logic_vector(1 downto 0);

begin

    uut : entity work.Ex3
        port map(
            sw => sw,
            bulb => bulb,
            sw_out => sw_out
        );

    stimuli : process
    begin
        sw <= "00";
        wait for 10 ns;
        assert (bulb = '0') report "Bulb should be off when both switches are
off";
        assert (sw_out = sw) report "sw_out should be equal to sw";

        sw <= "01";
```

```
        wait for 10 ns;
        assert (bulb = '1') report "Bulb should be on when switch 1 is on";
        assert (sw_out = sw) report "sw_out should be equal to sw";

        sw <= "10";
        wait for 10 ns;
        assert (bulb = '1') report "Bulb should be on when switch 2 is on";
        assert (sw_out = sw) report "sw_out should be equal to sw";

        sw <= "11";
        wait for 10 ns;
        assert (bulb = '0') report "Bulb should be off when both switches are
on";
        assert (sw_out = sw) report "sw_out should be equal to sw";

        wait;
    end process;

end Behavioral;
```