
LAB 4: BIT-PAIR RECODED MULTIPLIER

November 15, 2017

Rodrigo Rojas
Clemson University
Department of Electrical and Computer Engineering
rrojas@clemson.edu

Abstract

Laboratory four focuses on creating modular VHDL entities to create a single design with multiple purposes. The laboratory was divided into two parts: Design of a generic bit shift-add multiplier using the bit-pair recoding method discussed in class and the implementation of this design with OpenCL software. The laboratory consisted on designing a generic bit shift-add multiplier to be used in part two and implement it with OpenCL software. The outcome of the design was to be able to implement a VHDL entity with OpenCL software to obtain output of multiplication of two binary numbers in human readable form. The purpose of the laboratory was to reinforce the design on software Quartus II, simulation software ModelSim, programming language VHDL, and the use of OpenCL software.

1 Introduction

The fourth laboratory for ECE 3270, Digital Computing Design, had the purpose of introducing the student to the design of a generic bit shift-add multiplier using bit-pair recoding method using the Quartus II, ModelSim, and OpenCL software as well as the VHDL programming language. Laboratory four was divided into two parts. Part one consisted of the design of a Moore state machine, three shift-register, a register, a multiplexer, and an adder with the purpose of combining all components to output the multiplication of two binary numbers using the bit-pair recoding method. Part two consisted of implementing the design of Part one with the OpenCL software with the purpose of displaying the multiplication of two binary numbers calculated with the bit-pair recoding method in human readable form.

2 Part One: Design of Multiplier

In part one of the laboratory, the design of a shift-add multiplier using the bit-pair method was to be created. The design of the multiplication method was approached through the diagram provided in Laboratory Four Manual (Figure 1)[2]. The multiplication method required multiple components such as an Ripple-Carry Adder, General Register, Shift-Registers, Multiplexer, and Moore state machine.

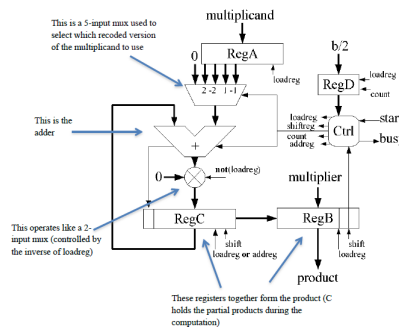


Figure 1: Laboratory Four Design Diagram

2.1 Ripple-Carry Adder

In part one of the laboratory, the design of a Ripple-Carry adder was to be created. Before this could be accomplished, the design of a Full Adder had to be accomplished first because a Ripple-Carry adder is the combination of Full Adders. A Full Adder is a circuit which performs addition of three one-bit binary numbers, two operands, and a carry bit. The Full Adder outputs a sum and a carry bit. The design of the Full Adder regarding logic and VHDL design was retrieved from previous Laboratory One. After the design of a Full Adder, the design of a Ripple-Carry Adder could be performed. The Ripple-Carry adder consisted of a combination of Full Adder circuits in which the carry

bit of each Full Adder is carried on the succeeding most significant Full Adder. The specification of the laboratory four regarding the Ripple-Carry adder consisted of making it a modular entity which could take two binary numbers of n number of bits, therefore it was to be a generic bit Ripple-Carry Adder. The design was made a low-level VHDL entity which was composed of four inputs, two n-bits binary numbers named "registerC" and "bitPair", carry bit named "Cin". Also, it was composed of two outputs, an n-bit output named "sum" corresponded to the two n-bit binary number and other output named "Cout" corresponded to the carry out bit of the sum of the two binary numbers. The architecture of this entity used multiple Full Adder components based on the n-bits of the two binary number inputs. The sum of the Ripple-Carry Adder gave an output of n number of bits of the inputs and was stored in output "sum". The sum of the two n-bit binary numbers output a carry out bit which was stored on output "Cout". It is important to note that output "Cout" was not used through out the Multiplier design because it was not needed in any operation (this will be explained later on the report). Figure 2 displays the VHDL code which contains the logic of using the Full Adder to sum two n-bit binary numbers.

```
-- Adder is responsible to add two numbers passed and returns
-- the sum in output "sum" and the carry bit in output "Cout"
ARCHITECTURE structure_adder OF adder IS
    COMPONENT FullAdder
    PORT (B, A, Cin: IN std_logic;
          Cout, Sum: OUT std_logic);
    END COMPONENT;
    SIGNAL C: std_logic_vector(0 TO n);
BEGIN
    C(0) <= Cin;
    Generate_label:
    FOR i IN 0 TO (n-1) GENERATE
        stage: FullAdder PORT MAP (B => bitPair(i), A => registerC(i),
                                   Cin => C(i), Cout => C(i + 1), Sum => sum(i));
    END GENERATE;
    Cout <= C(n); -- Does not matter
END structure_adder;
```

Figure 2: Ripple-Carry Adder VHDL Code

2.1.1 Ripple-Carry Adder Simulation

After the completion of the Full Adder and Ripple-Carry adder designs, simulation was performed through software, ModelSim. The purpose of the simulation was to confirm that the requirements of the Ripple-Carry Adder were met. The simulation consisted on three cases. Each test case consisted of entering entering different 8-bit binary numbers into inputs "bitPair" and "registerC" and keeping the input "Cin" constant with a value of "0" since it was not needed throughout the multiplication process. All test cases were compared with online Ripple Carry Adder Calculator website provided by instructor in Canvas website[1].

/adder_vhd_tst/Cin	0				
/adder_vhd_tst/bitPair	10110011	00001000	10001010		10110011
/adder_vhd_tst/registerC	10001101	00000101	00000111		10001101
/adder_vhd_tst/sum	01000000	00001101	10010001		01000000
/adder_vhd_tst/Cout	1				

Figure 3: Ripple-Carry Adder Simulation

In the first test case, inputs for "bitPair" and "registerC" were given positive 8-bit binary numbers of values "00001000" (8) and "00000101" (5) correspondingly. The output of this sum was expected to be "00001101" which corresponds to decimal value of 13. Output can be compared with Figure 4. The second test case, inputs for "bitPair" and "registerC" were given positive and negative binary numbers of values "10001010" (-118) and "00000111" (7) correspondingly. The output of this sum was expected to be "10010001" which corresponds to decimal value -111. Output can be compared with Figure 5. The last test case, input for "bitPair" and "registerC" were given negative binary number values "10110011" (-77) and "10001101" (-115) correspondingly. The output of this sum was expected to be "01000000" which corresponds to decimal value -192. Output can be compared with Figure 6.

```

A    0 0 0 0 1 0 0 0 : 8
B    + 0 0 0 0 0 1 0 1 : 5
-----
Sum  0 0 0 0 1 1 0 1 : 13

```

Figure 4: Ripple-Carry Adder Test Case 1

```

A    1 0 0 0 1 0 1 0 : -118
B    + 0 0 0 0 0 1 1 1 : 7
-----
Sum  1 0 0 1 0 0 0 1 : -111

```

Figure 5: Ripple-Carry Adder Test Case 2

```

A    1 0 1 1 0 0 1 1 : -77
B    + 1 0 0 0 1 1 0 1 : -115
-----
Sum  0 1 0 0 0 0 0 0 : -192

```

Figure 6: Ripple-Carry Adder Test Case 3

2.2 Register A

In part one of the laboratory, the design of a generic Register was to be created. Based on the diagram given in the Laboratory Four manual, this corresponded to Register A. The purpose of Register A was to store and maintain the value of the multiplicand value input throughout the multiplication process until the process was completed. The entity of Register A consisted of three inputs. The inputs consisted of a n-bit binary number input which corresponded to the multiplicand, a single bit named "reset" which corresponded to the reset enable pin, single bit input "clock" which corresponded to the synchronization clock throughout the multiplication process, single bit input "loadEnable" which corresponded to the load enable pin which told Register A that the multiplication process had started and made Register A store the value of the multiplicand throughout the multiplication process until the process was finished. Also, Register A contain a single output which was an n-bit binary number which corresponded to the value of the stored multiplicand. The architecture of Register A consisted on a process which constantly checked the value of the "reset" and "clock" values every clock cycle and performed two operations based on the value of "reset" and "loadEnable" inputs. If the value of "reset" input was of value "0", then Register A would clear the value of Register A and store value of "0" in all of its bits and assign this value to output "multiplicand". Otherwise, if the value of input "loadEnable" was of value "1", then Register A would store the value of the multiplicand and would store the value of the multiplicand into output "multiplicand". It should be noted that the multiplicand value was stored only once in Register A, which happened when "loadEnable" was of value "1". After the value of the multiplicand was stored, the input and output of Register A would remain constant, meaning multiplicand value did not change, until the multiplication was finished. Figure 7 displays the VHDL code which contains the logic of Register A.

```
-- Register A will set the register to all 0's if and only if reset value
-- is 0, otherwise, it will load the value of the multiplicand on the register
-- and hold it until all multiplication process is finished
|ARCHITECTURE behavior_registerA OF registerA IS
|BEGIN
|    PROCESS(reset, clock)
|    BEGIN
|        IF reset = '0' THEN
|            multiplicand <= (OTHERS => '0');
|        ELSIF RISING_EDGE(clock) THEN
|            IF loadEnable = '1' THEN
|                multiplicand <= input;
|            END IF;
|        END IF;
|    END PROCESS;
|END behavior_registerA;
```

Figure 7: Register A VHDL Code

2.2.1 Simulation of Register A

After the completion of Register A, simulation was performed through software, ModelSim. The purpose of the simulation was to confirm that the requirements of Register A were met. The simulation consisted of five test cases. Each test case consisted on changing the values of 8-bit binary number "input" and single bit inputs "reset", "clock", and "loadEnable".

/registera_vhd_tst/clock	1	0	1	0	1	0	1
/registera_vhd_tst/loadEnable	1			1		0	1
/registera_vhd_tst/reset	1	U		1		0	1
/registera_vhd_tst/input	-89	U	0		1		0
/registera_vhd_tst/multiplicand	-89	X			8		-89
		X	0		8	0	-89

Figure 8: Register A Simulation

In the first test case, input "reset" and "loadEnable" were given values of "0" and "0" correspondingly. The result of these two inputs loads value "0" into Register A and assign value of "0" to the 8-bits of output "multiplicand" which is correct because Register A is clearing all bits on it's register. The second test case, "reset" and "loadEnable" were given values "1" and "1" correspondingly and input "input" is given value of "8". The result of these three inputs loads value of "8" into Register A and assigns the corresponding value to the 8-bits of output "multiplicand" which is correct because Register A is storing the value of the multiplicand on it's register. In the third case, "reset" and "loadEnable" were given values "0" and "0" correspondingly and input "input" keeps value of "8". The result of these three inputs load value of "0" to 8-bits into Register A and assigns "0" to the 8-bits of output "multiplicand" which is correct because even if there is an input, Register A is told to reset, therefore it needs to clear all bits of its register. The final test case, "reset" and "loadEnable" were given values "1" and "1" correspondingly and input "input" is given value of "-89". The result of these three inputs loads value of "-89" into Register A and assigns the corresponding value to the 8-bits of output "multiplicand" which is correct because Register A is storing the value of the multiplicand on its register.

2.3 Shift-Register C

In part one of the laboratory, the design of a generic shift-register was to be created. Based on the diagram in the Laboratory Four manual, this corresponded to Register C. The purpose of Register C was to store the value of the addition of the current value of Register C and the bit-pair value obtained from the multiplexer. The entity of Register C consisted of eight bit inputs. The inputs consisted of a n-bit binary number named "input" which corresponded to the addition of the previous value stored at Register C and the bit-pair value obtained from the multiplexer, a single bit named "signBit" which corresponded to the extension sign bit sent from the multiplexer, single bit named "clock" which corresponded to the synchronization clock of multiplication process, single bit "reset" which corresponded to the reset enable pin of the state machine, single bit named

"addEnable" which corresponded of enable bit of the addition and shifting process of input "input", single bit input named "loadEnable" which corresponded to enable bit of the loading process of base value of "0". Also, Register C contained a n-bit binary number output named "output" which corresponded to the shifting of n-bit binary number input "input", and a 2-bit value named "bitsToB" which corresponded to the two bits lost of Register C after shifting which will be sent to Register B as output. The architecture of Register C consisted on a process which constantly checked the value of "clock" value very synchronization clock cycle and performed three operations based on the value of inputs "reset", "loadEnable" and "addEnable" inputs. If the value of input "reset", "loadEnable", and "addEnable" were of value "0", "0", and "0" correspondingly, then Register C would clear the value of Register C and store value of "0" in all of its bits and assign this value to output "output" and assign value "00" to output "bitsToB". If the value of inputs "reset", "loadEnable", and "addEnable" were of value "1", "1", and "0" correspondingly, then Register C would load itself with a value of all "0"s by assigning 0's to all bits of output "output" and assigning value "00" to output "bitsToB". Finally, if the value of inputs "reset", "loadEnable", and "addEnable" were of value "1", "0", and "1", then Register C would shift the value of "input" two places to the right and assign that value the output "output" and assign the last three bits of input "input" to output "bitsToB". Figure 9 displays the VHDL code which contains the logic of Register C.

```

}ARCHITECTURE behavior_registerC OF registerC IS
}BEGIN
}  PROCESS(clock)
}    VARIABLE allBits: std_logic_vector((n-1) DOWNT0 0);
}    BEGIN
}      IF reset = '0' THEN -- Reset everything to 0
}        allBits := (OTHERS => '0');
}        bitsToB <= "00";
}      ELSIF RISING_EDGE(clock) THEN
}        IF addEnable = '1' THEN
}          bitsToB <= input(1 DOWNT0 0);
}          FOR i IN 0 TO (n-3) LOOP
}            allBits(i) := input(i+2); -- Shifts bits to the right
}          END LOOP;
}          allBits(n-2) := signBit; -- Stores sign bit in first two bits after addition
}          allBits(n-1) := signBit;
}        ELSIF loadEnable = '1' THEN
}          allBits := (OTHERS => '0');
}          bitsToB <= "00";
}        END IF;
}      END IF;
}      output <= allBits; -- Outputs the final number after shifting
}    END PROCESS;
}END behavior_registerC;

```

Figure 9: Register C VHDL code

2.3.1 Simulation of Shift-Register C

After the completion of Register C, simulation was performed through software, ModelSim. The purpose of the simulation was to confirm that the requirements of Register C were met. The simulation consisted on three test cases. The test cases consisted of changing values of 8-bit binary number "input" and single bit inputs "reset", "signBit", "clock", "loadEnable", and "addEnable" with the purpose of checking that Register C clears all bits, loads base value of all 0's, and shifts the value of "input" two places to the right as well as assigning output "bitsToB" the two bits lost after the shift of input "input".

/registerc_vhd_tst/clock	1	0	1	0	1
/registerc_vhd_tst/reset	1	U	0	1	
/registerc_vhd_tst/loadEnable	0	U	0	1	0
/registerc_vhd_tst/addEnable	1	U	0		1
/registerc_vhd_tst/input	00000110	UUUUUUU			00000110
/registerc_vhd_tst/signBit	1	U			1
/registerc_vhd_tst/bitsToB	10	UU	00		10
/registerc_vhd_tst/output	11000001	UUUUU...	00000000		11000001

Figure 10: Register C Simulation

In the first test case single bit inputs "reset", "loadEnable", and "addEnable" were given values "0", "0", and "0" correspondingly. The result of the three inputs clears n-bit binary number stored in Register C by storing all 0's to all bits in its register by assigning this value to output "output". The second test case, inputs "reset", "loadEnable", and "addEnable" were given values "1", "1", and "0" correspondingly. The result of these three inputs stores value of "0" to n-bit binary number output "output". Finally, in the last test case, inputs "reset", "loadEnable", and "addEnable" were given values "1", "0", and "1" correspondingly and input "input" was given value of "00000110" (6) and input "signBit" a value of "1". The result of these five inputs shifts the value of input "input" two places to the right, store value of "signBit" into the two most significant value of the shifted value, store the shifted and assign value of output "output" and assign the two lost bits after shift to output "bitsToB". The expected result was to be "11000001" which corresponds to the output of the simulation.

2.4 Shift-Register B

In part one of the laboratory, the design of a generic shift-register was to be created. Based on the diagram of the Laboratory Four manual, this corresponded to Register B. The purpose of Register B was to store the value of the original and shifted binary value of the multiplier. The entity of Register B consisted of seven inputs. The inputs consisted of an n-bit binary number named "input" which corresponded to the original value of the multiplier, an n-bit binary number named "inputS" which corresponded to the continued shifted value of original multiplier with the bits passed from Register C on its most significant bits, a two-bit value which corresponded to the two bits sent from Register C, single bit "clock" which corresponded to the synchronization clock of the

multiplication process, a single bit "reset" which corresponded to the reset enable pin of the state machine, single bit named "loadEnable" which corresponded to load enable bit of the loading process of base value input "input", single bit named "shiftEnable" which corresponded to enable bit for shifting process of input "inputS". Also, Register B contained a n-bit binary number output named "output" which corresponded to the shifting of a n-bit binary number input "inputS", and a 3-bit value named "bitsToMul" which corresponded to the last three least significant bits of input "inputS" which will be sent to the multiplexer. The architecture of Register B consisted on a process which constantly checked the value of "clock" and "reset" every synchronization clock cycle and performed three operations based on the value of inputs "reset", "loadEnable", and "shiftEnable" inputs. If the value of the inputs "reset", "loadEnable", and "shiftEnable" were of value "0", "0", and "0" correspondingly, then Register B would clear the value of Register B and store value of "0" in all of its bits and assign this value to output "output", and assign value "000" to output "bitsToMul". If the value of inputs "reset", "loadEnable", and "shiftEnable" were of value "1", "1", and "0" correspondingly, then Register B would assign output "output" the value of input "input" and assign output "bitsToMul" the last two bits of input "input" and single bit value of "0" to its 3 bits correspondingly. If the value of inputs "reset", "loadEnable", and "shiftEnable" were of value "1", "0", and "1" correspondingly, then Register B would shift the value of input "inputS" two places to the right, assign the input "twoBits" to the two most significant bits to the shifted value, assign the finished shifted and sign extended value to output "output", and assign the second bit of input "inputS" to least significant bit of output "bitsToMul" and two least significant bits of shifted value to top two significant bits of output "bitsToMul" which would be sent to the multiplexer for calculations. Figure 11 displays the logic of Register B.

```

ARCHITECTURE behavior_registerB OF registerB IS
BEGIN
    PROCESS(clock, reset)
        VARIABLE allBits: std_logic_vector((n-1) DOWNTO 0);
        BEGIN
            IF reset = '0' THEN -- Reset everything to 0
                allBits := (OTHERS => '0');
                bitsToMul <= "000";
            ELSIF RISING_EDGE(clock) THEN
                IF loadEnable = '1' AND shiftEnable = '0' THEN
                    allBits := input;
                    bitsToMul <= allBits(1 DOWNTO 0) & '0';
                ELSIF shiftEnable = '1' AND loadEnable = '0' THEN
                    bitsToMul(0) <= inputS(1);
                    FOR i IN 0 TO (n-3) LOOP
                        allBits(i) := inputS(i + 2);
                    END LOOP;
                    allBits(n-2) := twoBits(0);
                    allBits(n-1) := twoBits(1);
                    bitsToMul(2 DOWNTO 1) <= allBits(1 DOWNTO 0);
                END IF;
            END IF;
            output <= allBits;
        END PROCESS;
    END behavior_registerB;

```

Figure 11: Register B VHDL code

2.4.1 Simulation of Shift-Register B

After the completion of Register B, simulation was performed through software, ModelSim. The purpose of the simulation was to confirm that the requirements of Register B were met. The simulation consisted on three test cases. The test cases consisted of changing vlaues of two 8-bit binary number inputs "input" and "inputS", two bit input "twoBits", and single bit inputs "reset", "clock", "loadEnable", and "shiftEnable" with the purpose of checking that Register B clears all bits, loads base multiplier value, and shift the value of "inputS" two places to the right as well as assigning output "bitsToMul" the bit lost after shifting as well as the last two least significant bits of the shifted value of "inputS".

/registerb_vhd_tst/clock	1					
/registerb_vhd_tst/reset	1					
/registerb_vhd_tst/loadEnable	0					
/registerb_vhd_tst/shiftEnable	1					
/registerb_vhd_tst/input	00000111	UUUUUUUU		00000111		
/registerb_vhd_tst/inputS	00000111	UUUUUUUU			00000111	
/registerb_vhd_tst/bitsToMul	011	UUU	000		110	011
/registerb_vhd_tst/output	11000001	UUUU...	00000000		00000111	11000001

Figure 12: Register B Simulation

In the first test case single bit inputs "reset", "loadEnable", and "shiftEnable" were given values "0", "0", and "0" correspondingly. The result of the three inputs clears n-bit

binary number stored in Register B by storing all 0's to all bits in its register by assigning this value to output "output". The second test case, inputs "reset", "loadEnable", and "shiftEnable" were given values "1", "1", and "0" correspondingly. The result of these three inputs stores the value of the multiplier to n-bit binary number output "output". Finally, in the last test case, inputs "reset", "loadEnable", and "shiftEnable", were given values "1", "0", and "1" correspondingly and input "inputS" was given value of "00000111" (7) and two-bit input "twoBits" a value of "11". The result of these five inputs shifts the value of input "inputS" two places to the right, store value of "twoBits" into the two most significant value of the shifted value, store the shifted and assign value to output "output" and assign second least bit of input "inputS" to the least significant bit of output "bitsToMul" and the new least significant bits after shift to output "bitsToMul". The expected result was to be "11000001" which corresponds to the output of the simulation.

2.5 Shift-Register D

In part one of the laboratory, the design of a generic shift-register was to be created. Based on the diagram of the Laboratory Four manual, this corresponded to Register D. The purpose of Register D was calculate the number of times the bit-pair add-shift method needed to be completed based on the number of bits of multiplicand and multiplier. The entity of Register D consisted of five inputs. The inputs consisted of a n-bit binary number named "input" which corresponded to a binary number of size $n/2$, single bit "loadEnable" which corresponds to the enable bit for loading the input "input" value into Register D, single bit "countEnable" which corresponds to the enable bit for shifting the input "input" one place to the right, single bit "reset" which corresponds to the reset enable bit for storing all 1's to Register D, and single bit "clock" which corresponds to the synchronization clock throughout the multiplication process. Also, Register C contained a $n/2$ bit binary number output named "output" which corresponded to the shifting of the $n/2$ bit binary number "output". The architecture of Register C consisted on a process which constantly checked the value of "clock" and "reset" every synchronization clock cycle and performed three operations based on the value of inputs "reset", "loadEnable", and "countEnable" inputs. If the value of the inputs "reset", "loadEnable", and "countEnable" were of value "0", "0", and "0" correspondingly, then Register D would clear the value of Register D and store value of "1" in all of its bits and assign this value to output "output". If the value of the inputs "reset", "loadEnable", and "countEnable" were of value "1", "1", and "0" correspondingly, then Register D would assign output "output" the value of all 1's. If the value of inputs "reset", "loadEnable", and "shiftEnable" were of value "1", "0", and "1" correspondingly, then Register D would shift the value of input "input" one place to the right and assign the new shifted value to output "output". Figure 13 shows the logic of Register D

```

]ARCHITECTURE behavior_registerD OF registerD IS
]BEGIN
]  PROCESS(reset, clock)
]    VARIABLE shiftBits: std_logic_vector(((n/2) - 1) DOWNT0 0);
]    BEGIN
]      IF reset = '0' THEN
]        shiftBits := (OTHERS => '1');
]      ELSIF RISING_EDGE(clock) THEN
]        IF loadEnable = '1' THEN
]          shiftBits := (OTHERS => '1');
]        ELSIF countEnable = '1' THEN
]          FOR i IN 0 TO ((n/2)-2) LOOP
]            shiftBits(i) := input(i+1); -- Shifts bits to the right
]          END LOOP;
]          shiftBits((n/2)-1) := '0';
]        END IF;
]      END IF;
]      output <= shiftBits;
]    END PROCESS;
]END behavior_registerD;

```

Figure 13: Register D VHDL code

2.5.1 Simulation of Shift-Register D

After the completion of Register D, simulation was performed through software, ModelSim. The purpose of the simulation was to confirm that the requirements of Register D were met. The simulation consisted of three test cases. The test cases consisted of changing of 4-bit binary number input "input", single bit inputs "loadEnable", "countEnable", "reset", and "clock" with the purpose of checking that Register D clears all bits, loads base value, and shift the value of "input" one place to the right.

/registerd_vhd_tst/clock	1	0	1	0	1	0	1
/registerd_vhd_tst/reset	1	U	0		1		
/registerd_vhd_tst/loadEnable	0	U	0		1		0
/registerd_vhd_tst/countEnable	1	U	0				1
/registerd_vhd_tst/input	1111	UUUU			1111		
/registerd_vhd_tst/output	0111	UUUU	1111				0111

Figure 14: Register D Simulation

In the first test case single bit inputs "reset", "loadEnable", and "countEnable" were given values "0", "0", and "0" correspondingly. The result of the three inputs clears the n/2 bit binary number stored in Register D by storing all 1's to all bits in its register by assigning this value to output "output". The second test case, inputs "reset", "loadEnable", and "countEnable" were given values "1", "1", and "0" correspondingly. The result of these three inputs stores the value of the input "input" to the n/2 bit binary number output "output". Finally, in the last test case, inputs "reset", "loadEnable", and "countEnable", were given values "1", "0", and "1" correspondingly and input "input" was given value of "1111". The result of these five inputs, shifted the value of input

"input" one place to the right, and store a value of 0 on the most significant bit of output "output". The expected result was to be "001" which corresponds to the output of the simulation.

2.6 Multiplexer

In part one of the laboratory, the design of a generic multiplexer was to be created. Based on the diagram of the Laboratory Four manual, this corresponded to a Multiplexer. The purpose of the Multiplexer was to determine the value needed to be summed with value of Register C based on the three last bits from Register B and multiplicand value of Register A. The entity of Multiplexer consisted of two inputs. The inputs consisted of a n-bit binary number named "registerA" which corresponded to the value of the multiplicand, and three-bit input "selectPin" which corresponded to the last three bits sent by Register B. Also, the Multiplexer contained two outputs. A single bit output named "signBit" which contained the sign bit needed to be sent to Register C and n-bit binary number named "output" which corresponds to the converted value of multiplicand which was sent to the Ripple-Carry Adder. The architecture of the Multiplexer consisted on a process which constantly checked the value of input "selectPin" every time the Multiplexer was called and performed five operations. If the value of "selectPin" was "000" or "111" it meant that "output" needed to be set to all 0's and the "signBit" was to be assigned either a "0" or a "1" based on the most significant bit value of the multiplicand value. If the value of "selectPin" was "001" or "010" it meant that the "output" needed to be set to the same value as the multiplicand and the "signBit" was to be set to value of the most significant bit of the multiplicand. If the value of "selectPin" is "011" it meant that the multiplicand input needed to be shifted to the left and then assigned to the Multiplexer "output" and "signBit" was to be set to the most significant bit of the multiplicand bit. If the value of "selectPin" was "110" or "101" it meant that the multiplicand input needed to be converted to its two's complement and set to Multiplexer "output" and set the "signBit" to the most significant bit of the two's complement value of the multiplicand. If the value of "selectPin" was "100" it meant that the multiplicand input needed to be converted to its two's complement, "signBit" was assigned the most significant bit of the two's complement of the multiplicand, then the two's complement of the multiplicand was shifted one space to the left. All decisions of "selectPin" were based from PowerPoint slide provided on Canvas (Figure 15). The logic of the Multiplexer can be seen on Figure 16.

Multiplier bit-pair		Multiplier bit on the right	Multiplicand version to be added	Relation to string of 1's
2^1 $i+1$	2^0 i			
0	0	0	0 x multiplicand	No string
0	0	1	+1 x multiplicand	End of string
0	1	0	+1 x multiplicand	Single 1 (+2 -1)
0	1	1	+2 x multiplicand	End of string
1	0	0	-2 x multiplicand	Beginning of string
1	0	1	-1 x multiplicand	End/beginning of string
1	1	0	-1 x multiplicand	Beginning of string
1	1	1	0 x multiplicand	String of 1's

Figure 15: Multiplexer PowerPoint Values

```

BEGIN
  bitPair <= selectPin;
  PROCESS (bitPair)
    VARIABLE multiplicandValue: std_logic_vector((n-1) DOWNT0 0);
    BEGIN
      IF bitPair = "000" OR bitPair = "111" THEN -- If bitPair is 0
        multiplicandValue := (OTHERS => '0');
        IF registerA(n-1) = '1' THEN
          signBit <= '1';
        ELSIF registerA(n-1) = '0' THEN
          signBit <= '0';
        END IF;
      ELSIF bitPair = "001" OR bitPair = "010" THEN -- If bitPair is 1
        multiplicandValue := registerA;
        signBit <= registerA(n-1);
      ELSIF bitPair = "011" THEN -- If bitPair is 2
        signBit <= registerA(n-1);
        multiplicandValue := registerA((n-2) DOWNT0 0) & '0';
      ELSIF bitPair = "110" OR bitPair = "101" THEN -- If bitPair is -1
        multiplicandValue := NOT registerA;
        multiplicandValue := std_logic_vector(unsigned(multiplicandValue + 1));
        signBit <= multiplicandValue(n-1);
      ELSIF bitPair = "100" THEN -- If bitPair is -2
        multiplicandValue := NOT registerA;
        multiplicandValue := std_logic_vector(unsigned(multiplicandValue + 1));
        signBit <= multiplicandValue(n-1);
        multiplicandValue := multiplicandValue((n-2) DOWNT0 0) & '0';
      END IF;
      output <= multiplicandValue;
    END PROCESS;
  END behavior_multiplexer;

```

Figure 16: Multiplexer VHDL code

2.6.1 Simulation of Multiplexer

After the completion of Multiplexer, simulation was performed through software, Model-Sim. The purpose of the simulation was to confirm that the requirements of Multiplexer were met. The simulation consisted of five test cases. The test cases consisted of keeping

a constant 8-bit binary number input "registerA" of value "00000101" (5) and three-bit input "selectPin" with the purpose of checking that the right conversion was outputted by the Multiplexer based on the value of "selectPin".

/multiplexer_vhd_tst/registerA	00000101	00000101				
/multiplexer_vhd_tst/selectPin	100	000	010	011	110	100
/multiplexer_vhd_tst/signBit	1					
/multiplexer_vhd_tst/output	11110110	00000000	00000101	00001010	11111011	11110110

Figure 17: Multiplexer Simulation

In the first test case input "selectPin" was given value of "000". The expected result of this input was to give "signBit" a value of 0 because the most significant value of the input "registerA" is "0" and the "output" given value of all 0's since "selectPin" is of value "000". The second test case changed value of "selectPin" to value of "010". The expected result of this input was to give "signBit" a value of "0" because the most significant value of the input "registerA" is "0" and the "output" given value should be same as input "registerA" since "selectPin" is of value "010". The third test case changed value of "selectPin" to value of "011". The expected result of this input was to give "signBit" a value of "0" because the most significant value of the input "registerA" is "0" and the "output" given value should be same as input "registerA" shifted one place to the left since "selectPin" is of value "011". The fourth case changed value of "selectPin" to value "110". The expected result of this input was to give "signBit" a value of "1" because the most significant value of the two's complement of input "registerA" is "1" and the "output" given value should be the two's complement of input "registerA" since "selectPin" is of value "110". The last test case changed the value of "selectPin" to value "100". The expected result of this input was to give "signBit" a value of "1" because the most significant value of the two's complement of input "registerA" is "1" and the "output" given value should be the two's complement of input "registerA" shifted one place to the left since "selectPin" is of value "100".

2.7 State Moore Machine: "Ctrl"

In part one of the laboratory, the design of Moore State machine was to be created. The State machine consisted of controlling the state of the bit-pair add-shift multiplication process by determining the current state the machine and use it as base to go to the next state to complete the multiplication process. The State machine contained six possible states: "resetS", "startS", "loadReg", "shiftReg", "addReg", and "done". Each of these states gave different output values for seven outputs which determined what the multiplication process needed to do next. The design of the State Moore Machine was approached in two steps. First, the machine's state transition was determined. The state transitions were calculated by using the diagram provided on the Laboratory Four Manual and determining the next possible state based on the current state of the machine. The calculation can be seen in on Figure 1. Second, software Quartus II was used to

design a low-level entity. The State Machine was mode low-level VHDL entity, which was composed of four inputs and seven outputs. The inputs were composed of four single bit inputs named "clock", "reset", "start", and "timeCount". The otuput consisted of seven single bit outputs named "busy", "load", "shift", "add", "resetO", "countO", and "ovalid". The architecture of the State Machine consisted of an internal signal named "controlState". The purpose of the internal signal was to store the next state of the Moore State machine of the six possible states of the State machine based on the current state of the machine. After obtaining the next state for the machine, the value state of "controlState" was used to determine the value of the seven single bit outputs. The State Machine started it's cycle in the "resetS" state and kept looping to it until the value of input "start" was set to "1" for one clock cycle. After this happened, the state jumped to the "startS" state for one clock cycle, the it jumped to "loadReg" for one clock cycle and set the value of "timeCount" to "1", then it jumped to the "addReg" state for once clock cycle, then it jumped to "shiftReg" for one clock cycle. The next state was determined on the value of input "timeCount". If the value of "timeCount" continued to be "1" it jumped back and forwarded from states "addReg" and "shiftReg". If the value of "timeCount" was changed to a "0", it would jump from state "shiftReg" to state "done". The machine stopped once the "done" state was reached, but it could be changed to state "resetS" if and only if the value of input "reset" was changed to value of "0". The state machine designed on Quartus II software can be seen in Figure 18.

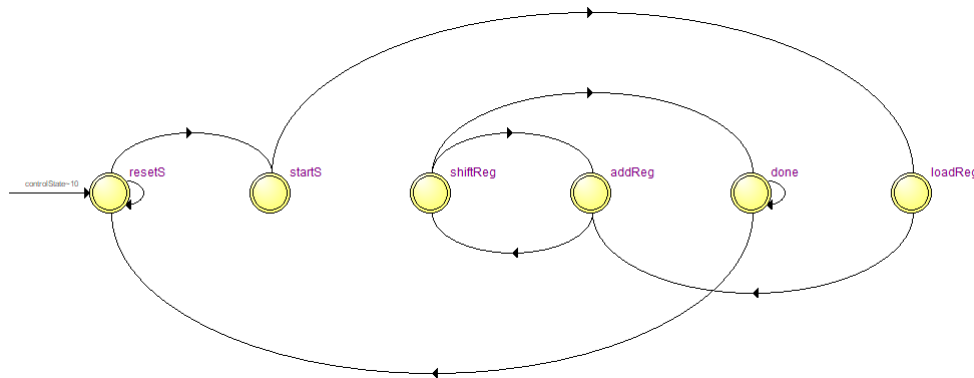


Figure 18: Moore State Machine Diagram

2.7.1 Simulation of Moore State Machine: Ctrl

After the completion of the Moore State Machine, simulation was performed through software, ModelSim. The purpose of the simulation was to confirm that the requirements of the State machine were met. The simulation consisted of the state machine going through each state and returning to it's start state at the end by changing the values inputs "clock", "timeCount", and "start".

/control_vhd_tst/i1/clock	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	
/control_vhd_tst/i1/reset	0	0		1													0	
/control_vhd_tst/i1/start	0	0		1	0													
/control_vhd_tst/i1/timeCount	0	0			1											0		
/control_vhd_tst/i1/busy	0	0			1											0		
/control_vhd_tst/i1/load	0	0			1		0											
/control_vhd_tst/i1/shift	0	0							1		0		1		0			
/control_vhd_tst/i1/add	0	0					1		0		1		0					
/control_vhd_tst/i1/resetO	0	0			1												0	
/control_vhd_tst/i1/countO	0	0					1		0		1		0					
/control_vhd_tst/i1/ovaid	0	0														1	0	
/control_vhd_tst/i1/controlState	resetS	resetS			startS			loadReg		addReg		shiftReg		addReg		shiftReg	done	resetS

Figure 19: Moore State Machine Simulation

The simulation consisted of going through each state and showing that state "resetS" only went to next state "startS" when input "start" was changed to "1", that states "addReg" and "shiftReg" kept looping on each other until the value of input "timeCount" was changed to "0", and that "done" state only went back to "resetS" state when input "reset" was changed to value of "0".

2.8 Top-Level Entity Bitpair

The part one of the laboraotry, the design of the top-level entity was to be created. The top-level entity was called "bitpair" which was responsible for connecting and implementing all components designed: Registers A, B, C, D, Multiplexer, Adder, and Moore State machine, with the purpose of implementation of bit-pair shift-add multiplication method. The design of this top-level entity was approached through the software Quartus II. The bitpair entity was made a top-level entity which contained five inputs. The inputs consisted of four single bit input named "clock" responsible for the synchronization of the multiplication process, "resetrn", responsible to send the reset enable signal, "ivalid" which is responsible for sending a signal to start the multiplication process, and "iready" was not utilized. The other input was a n-bit binary number called "datain" which contained the values of the multiplicand and multiplier. Also, the entity contained three outputs. Two single bit outputs named "oready" responsible for determining if the process was finished or not, and "ovaid" which determines that the output can be set. The other output was a n-bit binary output which contained the result of the multiplication of the multiplicand and multiplier. The architecture of the top-level entity bit pair called all components designed on part one and connected them through 16 signals. Once all components were connected, the State machine "ctrl" was responsible for controlling the whole multiplication process by sending enable bits output to each of the components which determined what step on the multiplication process needed to be done. The logic is displayed on Figure 20.

```

-- State machine ctrl, controls the state in which hardware is operating
ctrl: COMPONENT control PORT MAP (clock => clock, reset => resetn, start => invalid, timeCount => registerD_output(0),
    busy => busyOutput, load => loadOutput, shift => shiftOutput, add => addOutput,
    reset0 => resetOutput, count0 => countOutput, ovalid => ovalid);

-- Register A component, returns the multiplicand value
register_A: COMPONENT registerA PORT MAP (input => datain(((i*2) - 1) DOWNT0 i), reset => resetOutput, clock => clock,
    loadEnable => loadOutput, multiplicand => registerA_output);

-- Register D component, responsible for number of times register C and B shift
register_D: COMPONENT registerD PORT MAP (input => registerD_output, loadEnable => loadOutput, countEnable => countOutput,
    reset => resetOutput, clock => clock, output => registerD_output);

-- Register B component returns last three bits to multiplexer and shifts value to the right two places
register_B: COMPONENT registerB PORT MAP (input => datain(((i-1) DOWNT0 0), inputS => registerB_output, twoBits => registerC_bitsToB, clock => clock, reset => resetOutput,
    loadEnable => loadOutput, shiftEnable => shiftOutput, bitsToMul => registerR_bitsOutput,
    output => registerB_output);

-- Multiplexer responsible to return the "bit pair" value of A based on three bits passed by Register B
multi: COMPONENT multiplexer PORT MAP (registerA => registerA_output, selectPin => registerR_bitsOutput, signBit => multiplexer_signBitOutput,
    output => multiplexerOutput);

-- Register C responsible to hold addition of multiplexer and current value of register as well as passing last two bits to Register B after shifting
register_C: COMPONENT registerC PORT MAP (input => adderSum, signBit => multiplexer_signBitOutput, clock => clock, reset => resetOutput,
    shiftEnable => shiftOutput, loadEnable => loadOutput, addEnable => addOutput, output => registerC_output,
    bitsToB => registerC_bitsToB);

-- Adder component responsible to add current value of Register C and output of multiplexer
adder: COMPONENT adder PORT MAP (Cin => '0', registerC => registerC_output, bitPair => multiplexerOutput, sum => adderSum,
    Cout => adderCout);

```

Figure 20: Port Map of All Components

2.8.1 Simulation of Top-Level Entity Bitpair

After the completion of the Top-Level Entity Bipair, simulation was performed through software, ModelSim. The purpose of the simulation was to confirm that the requirements of the top-level entity were met. Four test cases were simulated which consisted of entering different multiplicand and multiplier values to input "datain". All simulation output was checked with online multiplication calculator provided by the Canvas website[1].

[illegible]

Figure 21: Positive Multiplication Simulation

The first test case consisted of multiplying two positive 8-bit values 6 and 7. The result gives 42 which is the correct output. The shifting of bits, addition, and result can be checked with Figure 22.

A	00000111	7
X	x0000110	6
Shift Only	000000000	
Add A	+00000111	
	000001110	
Shift	0000001110	
Add A	+00000111	
	0000101010	
Shift	00000101010	
Shift Only	000000101010	
Shift Only	0000000101010	
Shift Only	00000000101010	
Shift Only	0000000001010102	

Figure 22: Proof of Positive Multiplication Simulation

[illegible]

Figure 23: Negative Multiplication Simulation

The second test case consisted of multiplying two negative 8-bit values -86 and -102. The result gives positive value 8772 which is the correct output. The shifting of bits, addition, and result can be checked with Figure 24.

A	1 0 1 0 1 0 1 0	-86
X	x 1 0 0 1 1 0 1 0	-102
Shift Only	0 0 0 0 0 0 0 0	
Add A	+ 1 0 1 0 1 0 1 0	
	1 0 1 0 1 0 1 0 0	
Shift	1 1 0 1 0 1 0 1 0 0	
Shift Only	1 1 1 0 1 0 1 0 1 0 0	
Add A	+ 1 0 1 0 1 0 1 0	
	1 0 0 1 0 1 0 0 1 0 0	
Shift	1 1 0 0 1 0 1 0 0 1 0 0	
Add A	+ 1 0 1 0 1 0 1 0	
	0 1 1 1 0 1 0 0 0 1 0 0	
Shift	1 0 1 1 1 0 1 0 0 0 1 0 0	
Shift Only	1 1 0 1 1 1 0 0 0 0 1 0 0	
Shift Only	1 1 1 0 1 1 1 0 1 0 0 0 1 0 0	
Correct	+ 1 0 1 0 1 1 1 0	
	0 1 0 0 0 1 0 0 1 0 0 0 1 0 0 8722	

Figure 24: Proof of Negative Multiplication Simulation

[illegible]

Figure 25: Positive and Negative Multiplication Simulation

The third case consisted of multiplying a positive and negative 8-bit values -102 and 2. The result gives the negative value -204 which is the correct output. The shifting of bits, addition, and result can be checked with Figure 26.

A	1 0 0 1 1 0 1 0	-102
X	x 0 0 0 0 0 0 1 0	2
Shift Only	0 0 0 0 0 0 0 0	
Add A	+ 1 0 0 1 1 0 1 0	
	1 0 0 1 1 0 1 0 0	
Shift	1 1 0 0 1 1 0 1 0 0	
Shift Only	1 1 0 0 1 1 0 1 0 0	
Shift Only	1 1 1 0 0 1 1 0 1 0 0	
Shift Only	1 1 1 1 0 0 1 1 0 1 0 0	
Shift Only	1 1 1 1 1 0 0 1 1 0 1 0 0	
Shift Only	1 1 1 1 1 1 0 0 1 1 0 1 0 0	-204

Figure 26: Proof of Positive and Negative Multiplication Simulation

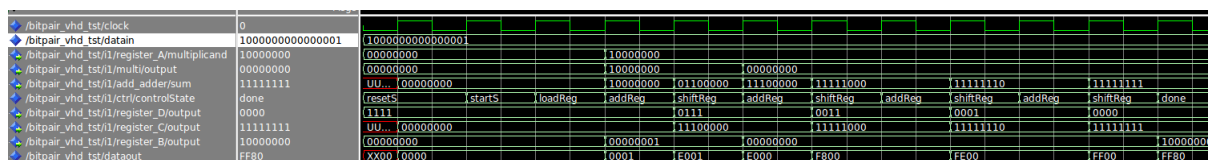


Figure 27: Overflow Multiplication Simulation

3 Part Two: OpenCL Implementation

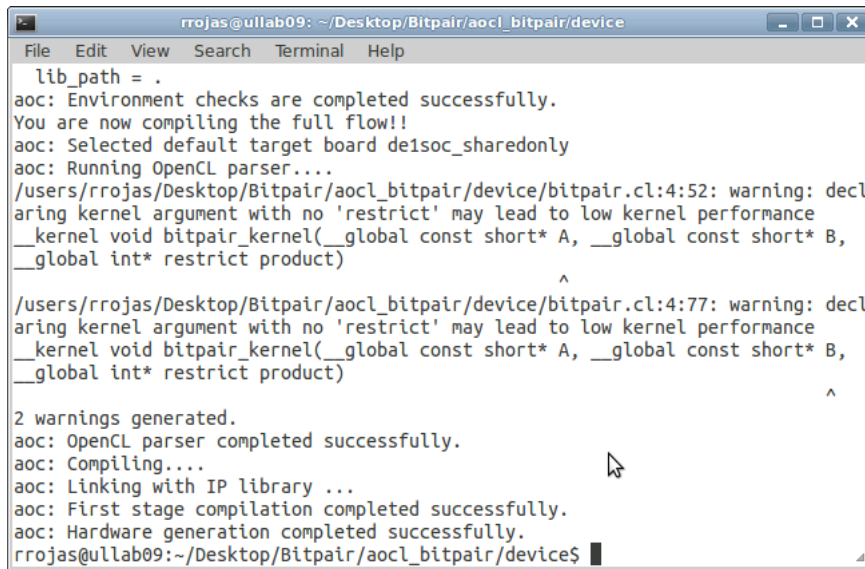
Part two of laboratory four focused on implementing the designed of the bit-pair add-shift multiplication method with the OpenCL software. The design from part one was to be extended to multiply two 16-bit numbers. After the completion of this modification, the multiplier component was to use OpenCL "bitpair.vhd". The input was mapped to input "datain", output to "dataout", as well as pins "ivalid", "ovalid" and "oready". The "ivalid" pin was set high for exactly one clock cycle when "datain" has a mutliplicand and multiplier. The "ovalid" pin was set high whenever the multiplier was not conducting the process of multiplication and is ready to receive the next output. The OpenCL xml file "bitpair_rtl.xml" was modified i the latency to match the clock cycles needed for multiplication process to be done as well as adding all necessary files that the multiplication design uses. After the compilation of OpenCL software was successful, the program was run but it was stuck on an infinite loop. The output of compilation and infinite loop can be seen in Figures 28 and 29.

3.1 Obtaining Tar File

The first part consisted of logging into Canvas and downloading tar file named "Bitpair.tgz" which can be found under "Lab 4 BitPair Multiplier" assignment files. After the file was downloaded, the file was extracted in one of the ULLAB computers located in the laboratory room.

3.2 Compilation of Program and Storage of SD Card

After the modification of the VHD files, compilation of the program could proceed. First, a terminal was opened in the directory "Bitpair/aolc_bitpair" from the extracted "Bitpair.tgz" file. This directory contained file named "Makefile" which was executed by using the command "make", the outcome of executing this command was an executable file named "bitpair". Second, directory of terminal was changed to "Bitpair/aolc_bitpair/device". This directory also contained a file named "Makefile" which was executed by using command "make", the outcome of executing the command was an executable file named "bitpair.aocx". The purpose of compilation was to obtain the files "bitpair" and "bitpair.aocx" with the purpose of executing them on board DE1_SoC board. After the compilation of the VHD files was complete, the executable files "bitpair" and "bitpair.aocx" were transferred to a SD card. The process of transferring the executable files to the SD card could only be completed in a computer running a Linux Operating System. Once the SD card was inserted into the computer, a folder named "lab4" was created in directory "home/root" in which executable files "bitpair" and "bitpair.aocx" were stored at. See Figure 28.

A screenshot of a terminal window titled "rrojas@ullab09: ~/Desktop/Bitpair/aolc_bitpair/device". The terminal shows the output of a compilation process. It starts with "lib_path = ." and "aoc: Environment checks are completed successfully." followed by "You are now compiling the full flow!!". It then shows "aoc: Selected default target board deisoc_sharedonly" and "aoc: Running OpenCL parser...". There are two warnings from the OpenCL parser: one at line 4:52 and another at line 4:77, both stating "warning: declaring kernel argument with no 'restrict' may lead to low kernel performance". After the warnings, it says "2 warnings generated." and "aoc: OpenCL parser completed successfully." followed by "aoc: Compiling....", "aoc: Linking with IP library ...", "aoc: First stage compilation completed successfully.", and "aoc: Hardware generation completed successfully." The prompt "rrojas@ullab09:~/Desktop/Bitpair/aolc_bitpair/device\$" is visible at the bottom.

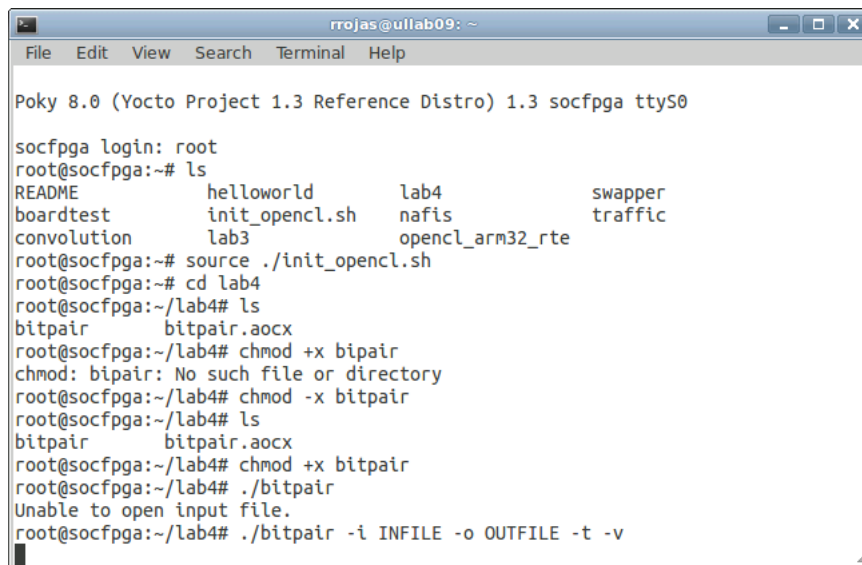
```
rrojas@ullab09: ~/Desktop/Bitpair/aolc_bitpair/device
File Edit View Search Terminal Help
lib_path = .
aoc: Environment checks are completed successfully.
You are now compiling the full flow!!
aoc: Selected default target board deisoc_sharedonly
aoc: Running OpenCL parser...
/users/rrojas/Desktop/Bitpair/aolc_bitpair/device/bitpair.cl:4:52: warning: decl
aring kernel argument with no 'restrict' may lead to low kernel performance
__kernel void bitpair_kernel(__global const short* A, __global const short* B,
__global int* restrict product)
^
/users/rrojas/Desktop/Bitpair/aolc_bitpair/device/bitpair.cl:4:77: warning: decl
aring kernel argument with no 'restrict' may lead to low kernel performance
__kernel void bitpair_kernel(__global const short* A, __global const short* B,
__global int* restrict product)
^
2 warnings generated.
aoc: OpenCL parser completed successfully.
aoc: Compiling....
aoc: Linking with IP library ...
aoc: First stage compilation completed successfully.
aoc: Hardware generation completed successfully.
rrojas@ullab09:~/Desktop/Bitpair/aolc_bitpair/device$
```

Figure 28: OpenCL Compilation

3.3 DE1_SoC Configuration and Program Execution

After the executable files were compiled and stored in an SD card the process of configuring the DE1_SoC board's hardware and software configurations and program execution could proceed. First, the DE1_SoC board was configured to run on an OpenCL configuration. The DE1_SoC board as default runs in the JTAG configuration, to configure it to the OpenCL configuration, switch SW10 on the back of the board had to be set to value

"110101". See Figure 12 for reference. After this was completed, the microSD card could be inserted into the board and upon powering on the board, red LEDs started showing a binary counting sequence. After the hardware OpenCL configuration was completed, the software configuration had to be completed. The software setup consisted on accessing the DE1-SoC board through a ULLAB computer. To do this, a terminal was opened and command "minicom" was entered. This command made the board start initialization. Then, the type of user was prompted and user "root" was entered. After board was accessed, command command "./init_openc1.sh" was entered which configured the board to run OpenCL code. All steps regarding DE1-SoC configuration were based on manual provided by laboratory instructor[3]. After the software configuration was completed and execution of executable files transferred to the SD card could be executed. To accomplish this, directory "root/home/lab4" was accessed. In this directory, executable file "bitpair" and "bitpair.aocx" were located here. To run these executable files, command "chmod +x bitpair" was entered which made file "bitpair" an executable file. After this, command "./bitpair -i INFILE -o OUTFILE -t -v" was entered which executed the file and an output was not obtained, it was stuck on an infinite loop. See Figure 29.



```

rrojas@ullab09: ~
File Edit View Search Terminal Help

Poky 8.0 (Yocto Project 1.3 Reference Distro) 1.3 socfpga ttyS0

socfpga login: root
root@socfpga:~# ls
README          helloworld      lab4             swapper
boardtest       init_openc1.sh  nafis            traffic
convolution     lab3            openc1_arm32_rte
root@socfpga:~# source ./init_openc1.sh
root@socfpga:~# cd lab4
root@socfpga:~/lab4# ls
bitpair          bitpair.aocx
root@socfpga:~/lab4# chmod +x bitpair
chmod: bitpair: No such file or directory
root@socfpga:~/lab4# chmod -x bitpair
root@socfpga:~/lab4# ls
bitpair          bitpair.aocx
root@socfpga:~/lab4# chmod +x bitpair
root@socfpga:~/lab4# ./bitpair
Unable to open input file.
root@socfpga:~/lab4# ./bitpair -i INFILE -o OUTFILE -t -v

```

Figure 29: OpenCL Infinite Execution

4 Conclusion

The purpose of laboratory four was to reinforce the use of the VHDL programming language, design software Quartus II, simulation software ModelSim, and implementation software OpenCL to the student. The outcome of the laboratory four taught the student how to create and design modular entities and implement it with OpenCL software. The laboratory accomplished this by making the student design a bit-pair add-shift

multiplication hardware with multiple components and implementing it with OpenCL software to obtain human readable output as numbers.

References

- [1] Addition. *Computer Arithmetic*. Canvas, 2017.
- [2] Canvas. *BitPair Multiplication.pdf*. Canvas, 2017.
- [3] Canvas. *Compilation and Usage.pdf*. Canvas, 2017.