جمهوری اسلامی ایران
وزارت علوم تحقیقات و فناوری

Project Report

Behavioral & Structural VHDL Implementation of a ROM

BY

Mr. Kiani, Esfandiar

Student ID: 9957122

Department of Computer Engineering

Sheikhbahaee University

Supervisor

Mr. Tabei, Mehdi

Winter 2023

1. Abstraction

This report delves into the design and implementation of Read-Only Memory (ROM) using two distinct design paradigms within the VHDL (VHSIC Hardware Description Language) framework: Behavioral and Structural modeling. The Behavioral model plays a pivotal role in verifying the logic and functionality of the ROM, whereas the Structural model provides detailed insight into the actual hardware configuration, facilitating a seamless transition from a virtual model to a physical realization.

Extensive simulations have been carried out to validate the performance, accuracy, and effectiveness of the ROM designs. The results affirm that both implementations successfully meet the required specifications, each with its inherent strengths and trade-offs in terms of complexity, simulation speed, and resource utilization.

This report aims to provide a comprehensive reference for scholars in the field, offering a clear example of how VHDL can be employed to tackle complex digital design projects from two fundamentally different but complementary perspectives. It is a testament to the versatility and power of VHDL in modeling and simulating digital circuits, pivotal for both educational purposes and the advancement of digital design and integrated circuitry.

2. Introduction

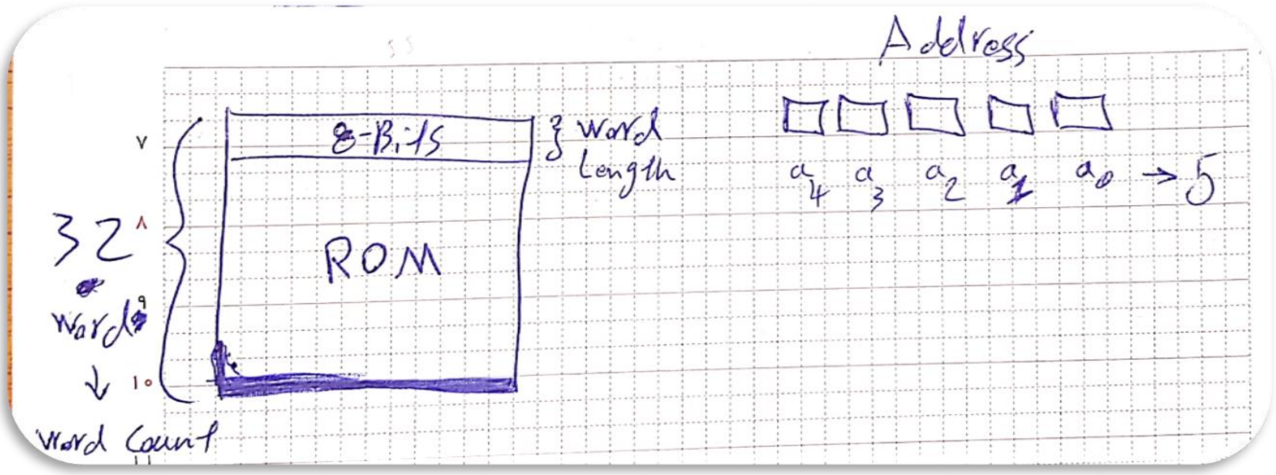
Read-Only Memory (ROM) is a type of non-volatile storage medium that is utilized widely in electronic systems to permanently store data. ROM retains its contents regardless of the presence or absence of power, making it an essential component for preserving firmware, bootloader code, and system-level functions across reboots and power cycles. Its applications are diverse, ranging from simple device control found in microwaves and washing machines to complex code storage in computers, smartphones, airplanes, and space shuttles required during system boot-up.

The implementation of ROM using VHDL furnishes designers with a powerful tool for capturing complex digital logic in an abstract, yet precise manner. VHDL not only allows for the synthesis of physical hardware but is also an instrumental asset in the testing and verification processes of its simulation capabilities. It supports a range of abstraction levels, from behavioral modeling, which emphasizes functionality, to structural modeling, focusing on the actual physical representation of circuits. For ROM design, this level of abstraction is especially beneficial as it allows designers to validate logic performance before moving to hardware prototypes.

3. Design Methodology

i. Hardware Design

The design methodology initially aimed to develop a single-phase process for creating a representation of a Read-Only Memory (ROM) device with a capacity of 32 words by 8 bits. This means that the internal structure of the memory is comprised of 32 eight-bit words, which can also be understood as 1-byte words. Consequently, for addressing the individual memory cells (words), a 5-bit address line is required.



ii. Initial Data Specification

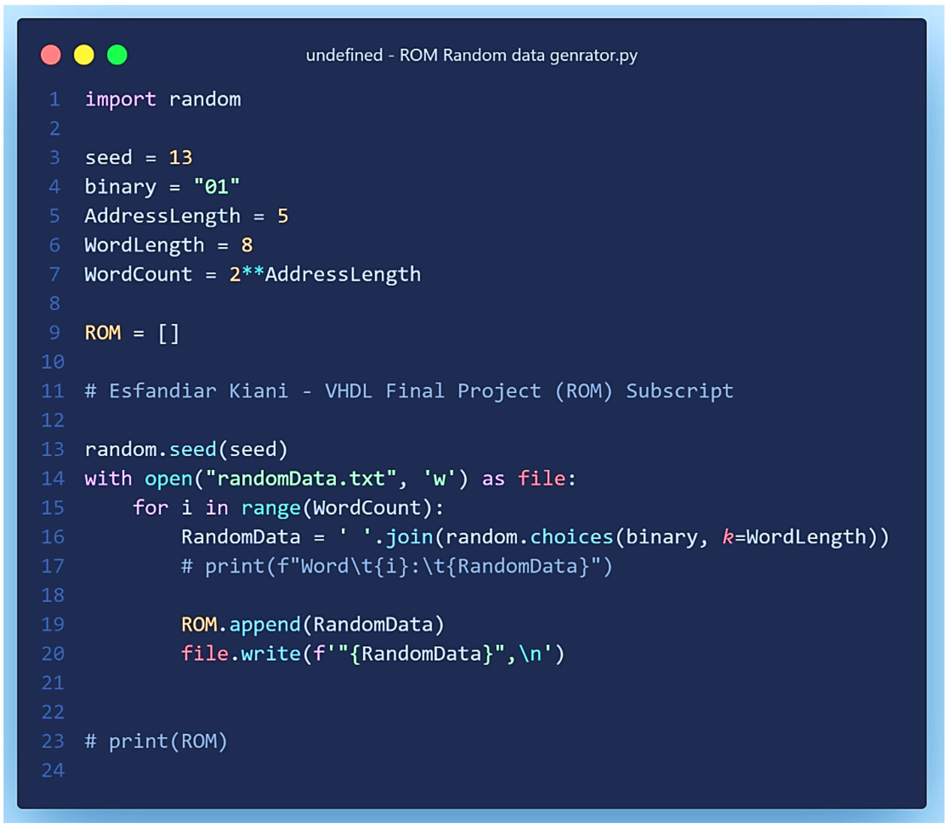
Within the domain of ROM design, a defining attribute is the intrinsic restriction against post-manufacturing data modification within the memory cells. ROM is engineered to maintain data in a static state, rendering it immutable during regular operational conditions. The process of data retrieval is inherently passive, involving selective activation of associated word lines coupled with the actuation of sense amplifiers, consequentially resulting in the output of pre-stored data.

This absence of write capability inherently streamlines the automation process in design execution. The initial dataset is generated by using Python programming language, which is renowned for its power and utility in computational tasks. Python facilitates the creation of this dataset through a deterministic random generation process, tailored for integration and applicability within VHDL frameworks.

The methodical approach adopted ensures that the Python-generated data posits a flexible format, enhancing adaptability and ensuring efficacious application throughout the VHDL design trajectory. The culmination of this procedural approach is manifested in the generation of a standardized output file, the specifications of which will be delineated subsequently.

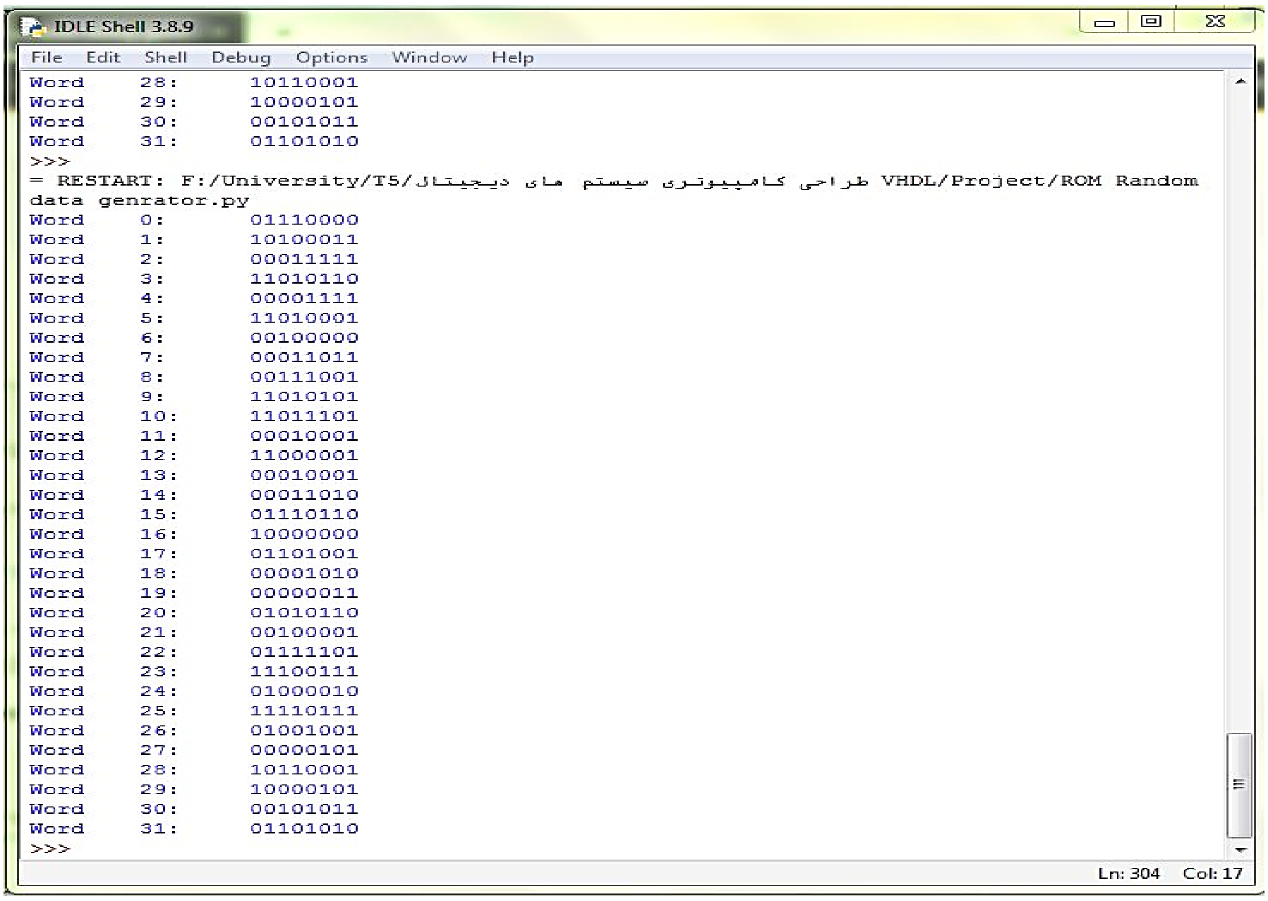
First of all, I need to import 'random' library to generate random (semi-random) data. After that, I define some variables such as seed (to give a specific answer every single time that I run the script), the binary alphabet, and the ROM size.

Also define an empty list datatype to store generated data together, before regenerating random data by using the 'random.choice()' method and write them into a file which is named 'randomData.txt' in the same directory.



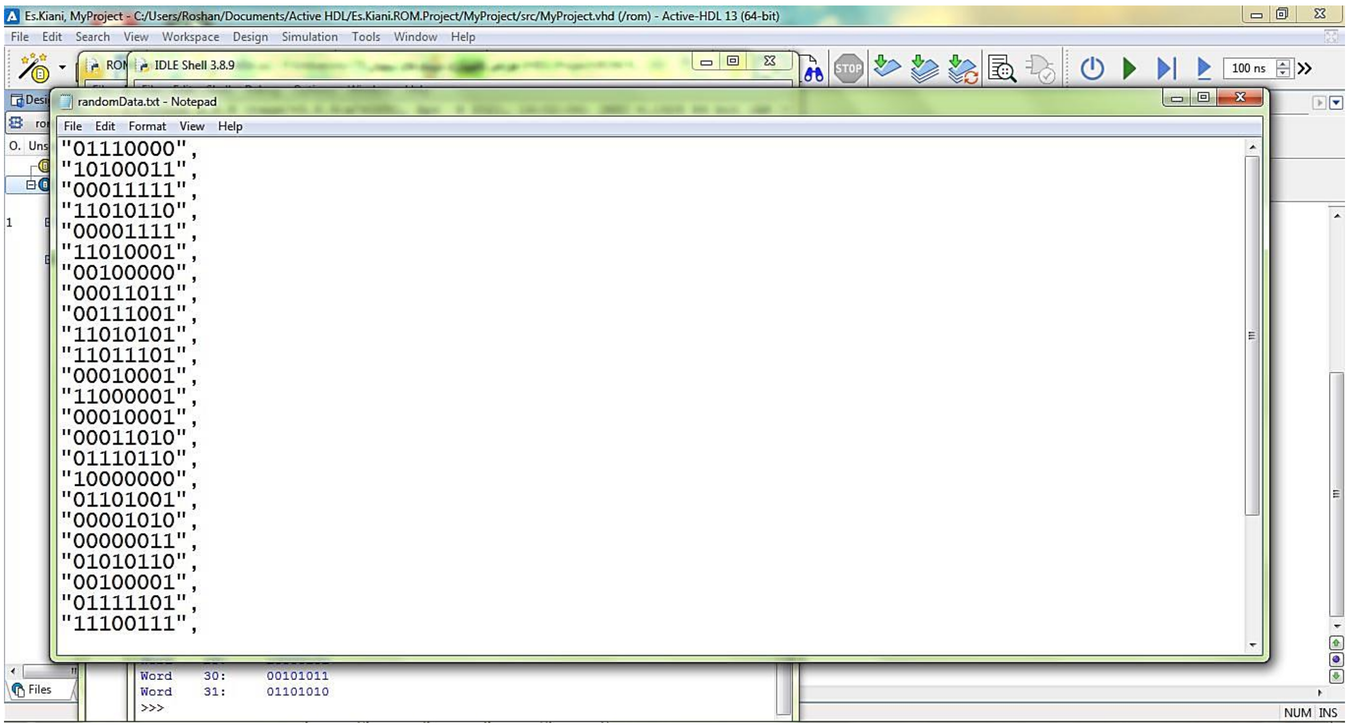
```python
import random

seed = 13
binary = "01"
AddressLength = 5
WordLength = 8
WordCount = 2**AddressLength

ROM = []

# Esfandiar Kiani - VHDL Final Project (ROM) Subscript

random.seed(seed)
with open("randomData.txt", 'w') as file:
    for i in range(WordCount):
        RandomData = ' '.join(random.choices(binary, k=WordLength))
        # print(f"Word\t{i}:\t{RandomData}")

        ROM.append(RandomData)
        file.write(f'"{RandomData}",\n')


# print(ROM)

```

The standart output of the script is:



```
Word    28:     10110001
Word    29:     10000101
Word    30:     00101011
Word    31:     01101010
>>>
= RESTART: F:/University/T5/طراحی کامپیوتری سیستم های دیجیتال VHDL/Project/ROM Random
data genrator.py
Word    0:      01110000
Word    1:      10100011
Word    2:      00011111
Word    3:      11010110
Word    4:      00001111
Word    5:      11010001
Word    6:      00100000
Word    7:      00011011
Word    8:      00111001
Word    9:      11010101
Word    10:     11011101
Word    11:     00010001
Word    12:     11000001
Word    13:     00010001
Word    14:     00011010
Word    15:     01110110
Word    16:     10000000
Word    17:     01101001
Word    18:     00001010
Word    19:     00000011
Word    20:     01010110
Word    21:     00100001
Word    22:     01111101
Word    23:     11100111
Word    24:     01000010
Word    25:     11110111
Word    26:     01001001
Word    27:     00000101
Word    28:     10110001
Word    29:     10000101
Word    30:     00101011
Word    31:     01101010
>>>
```

And the ‘randomData.txt’ file content is:


Es.Kiani, MyProject - C:/Users/Roshan/Documents/Active HDL/Es.Kiani.ROM.Project/MyProject/src/MyProject.vhd (/rom) - Active-HDL 13 (64-bit)
File Edit Search View Workspace Design Simulation Tools Window Help
IDLE Shell 3.8.9
randomData.txt - Notepad
File Edit Format View Help
100 ns
NUM INS

```
"01110000",
"10100011",
"00011111",
"11010110",
"00001111",
"11010001",
"00100000",
"00011011",
"00111001",
"11010101",
"11011101",
"00010001",
"11000001",
"00010001",
"00011010",
"01110110",
"10000000",
"01101001",
"00001010",
"00000011",
"01010110",
"00100001",
"01111101",
"11100111",
```

```
Word    30:    00101011
Word    31:    01101010
>>>
```

4. Implementation

After defining the architecture and overall structure of memory arrays and the corresponding bit generators (initial data), the process moved to the hardware design phase using VHDL language.

In addition to the standard inputs and outputs, such as the address and data lines, a control input named ‘enable,’ was incorporated. This active-low input, enables the output to transition between logical 0/1 (High and Low) and High-Impedance (Z) states. Similarly, the structural dimensions were designed with relative parameterization and reference-based, allowing for scalability in the overall design.

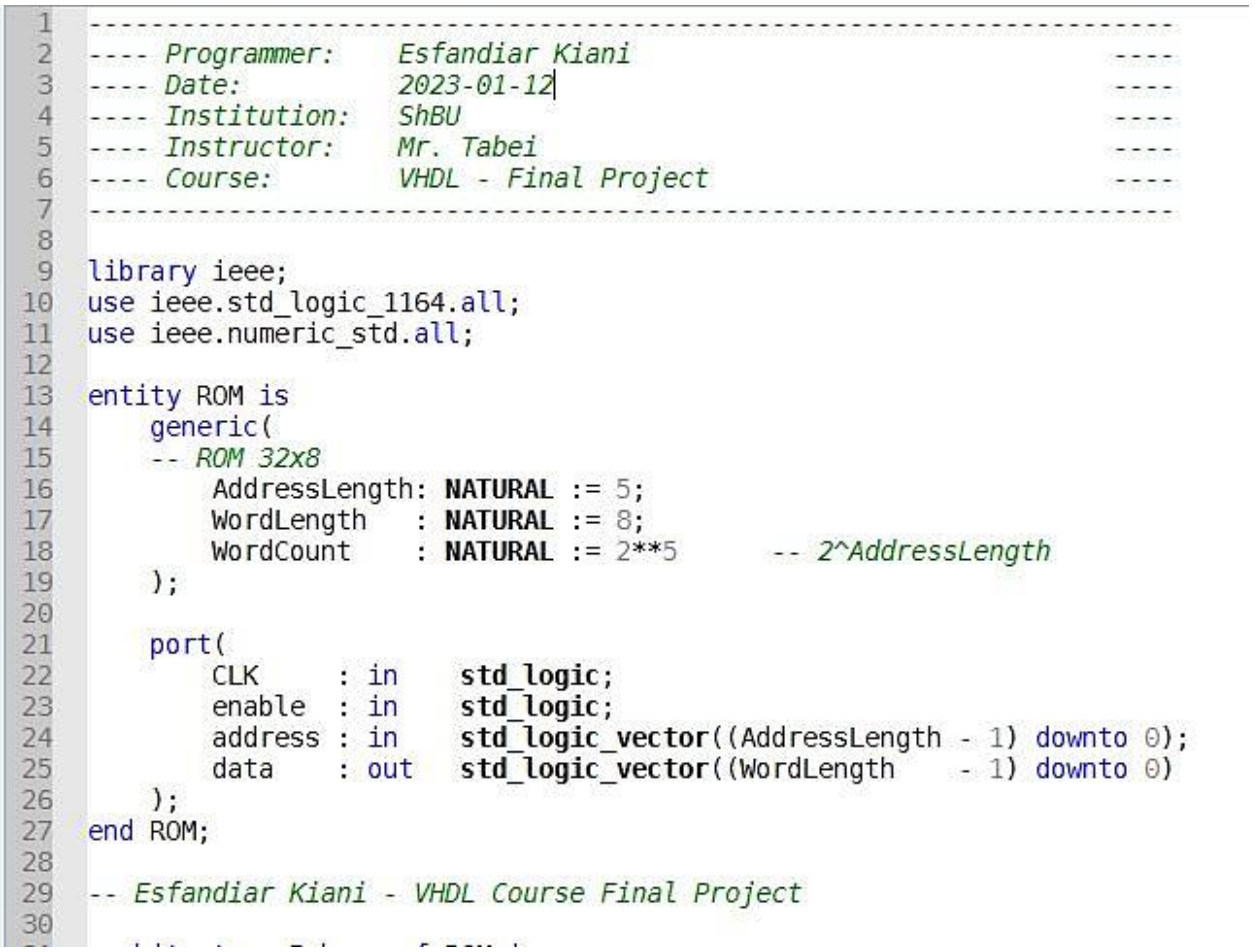
This flexibility ensures that changes to the memory array size can be implemented with ease, and it is only the initial data that needs to be adjusted according to the redefined size of the memory cells.

The memory is designed to positive-edge triggered, operating with a delay of one clock cycle. This implies a reliable and stable production of the output after any alterations in the input, becoming visible on the following clock edge.

i. Behavioral

In the VHDL code, after the header part, I used the ‘ieee’ library and imported ‘std_logic_1164’ and ‘numeric’ module.

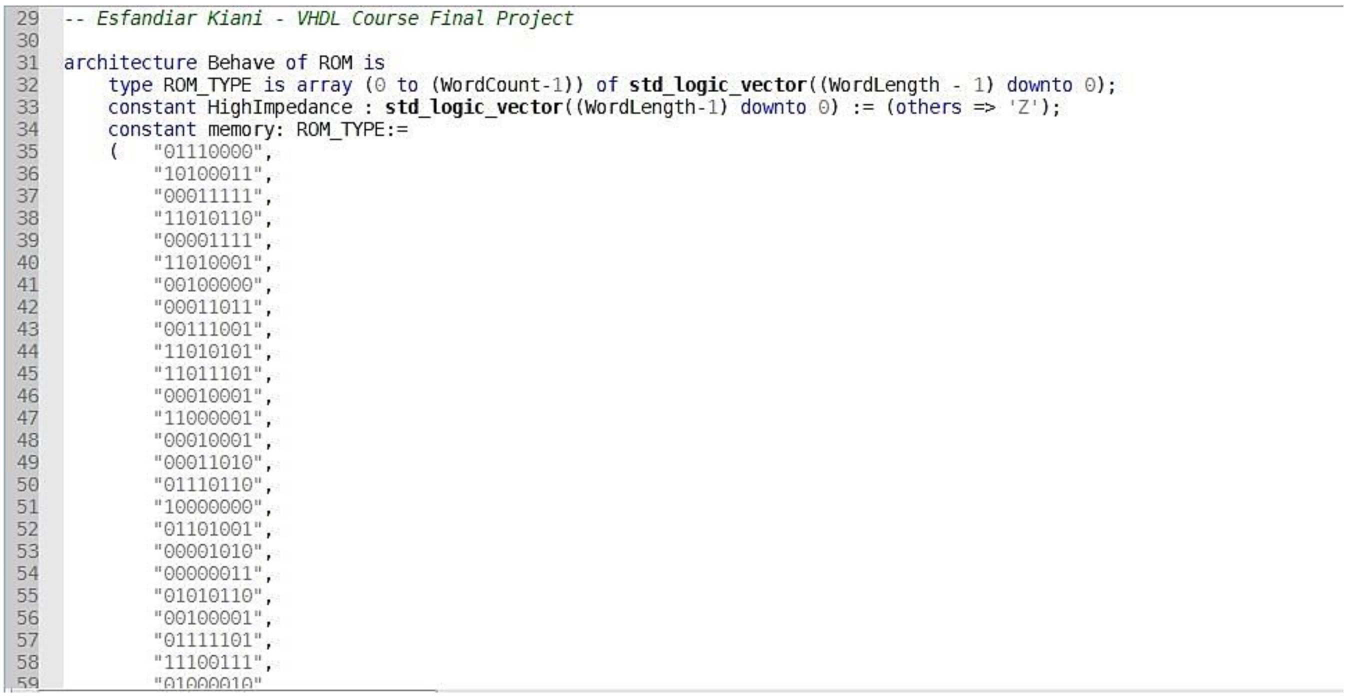
After that, I have to define one or more entities that I want to use them. As you can see in the following code, I use generic datatype to define my ROM size in some variables and define 4 specific ports for clock pulse (CLK), enable, address line (ROM input), and the data line (output).



```vhdl
-------------------------------------------------------------------
---- Programmer:    Esfandiar Kiani                            ----
---- Date:          2023-01-12                                 ----
---- Institution:   ShBU                                       ----
---- Instructor:    Mr. Tabei                                  ----
---- Course:        VHDL - Final Project                       ----
-------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity ROM is
    generic(
    -- ROM 32x8
        AddressLength: NATURAL := 5;
        WordLength   : NATURAL := 8;
        WordCount    : NATURAL := 2**5      -- 2^AddressLength
    );

    port(
        CLK     : in    std_logic;
        enable  : in    std_logic;
        address : in    std_logic_vector((AddressLength - 1) downto 0);
        data    : out   std_logic_vector((WordLength    - 1) downto 0)
    );
end ROM;

-- Esfandiar Kiani - VHDL Course Final Project

```

However, the main part of a VHDL program is the architecture section which we should represent the functionality of our circuit to the machine. In this part, I need to define an 8-bit array (vector) for each memory word, and a second array (not vector) to include all memory word arrays inside it (2D array).

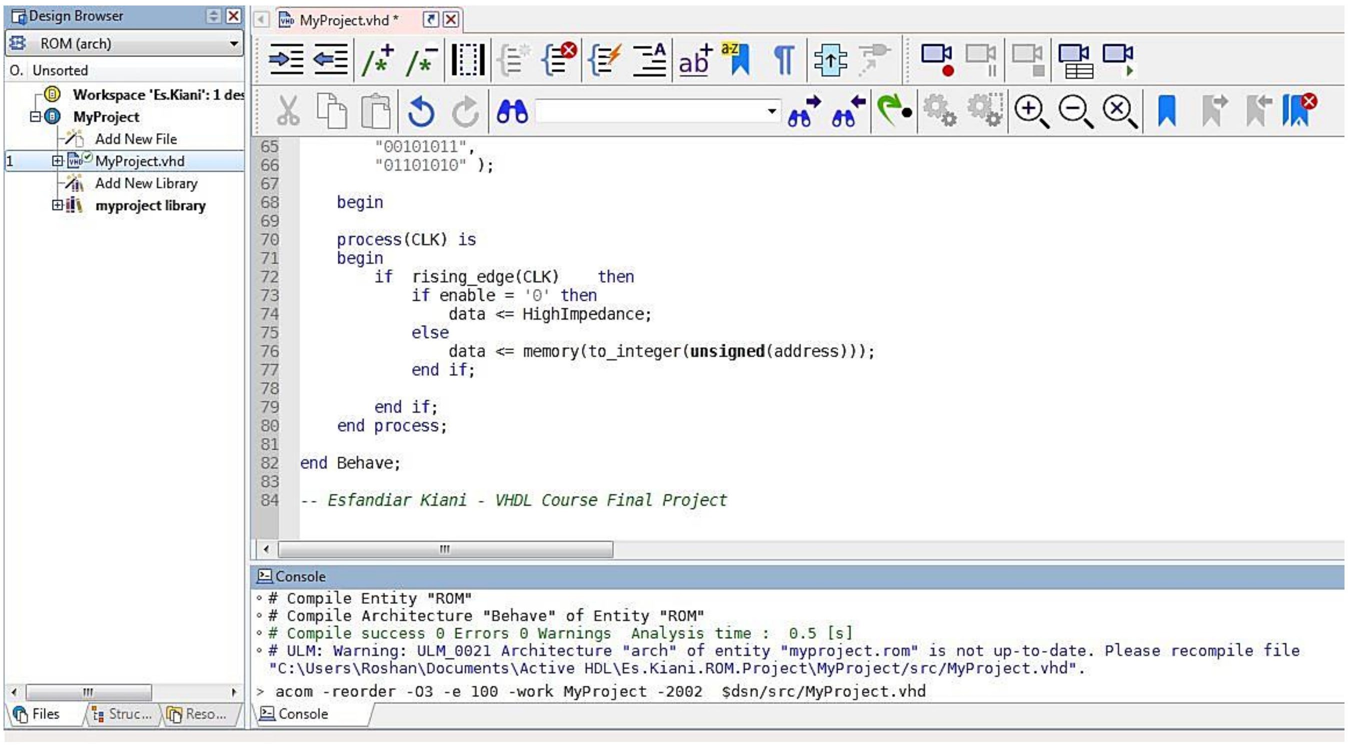
Therefore, to do that, I define a new datatype named 'ROM_TYPE', which is actually a 32-cell array of 8-bit vectors. Now I can store the previously generated data in these cells.



```
-- Esfandiar Kiani - VHDL Course Final Project

architecture Behave of ROM is
    type ROM_TYPE is array (0 to (WordCount-1)) of std_logic_vector((WordLength - 1) downto 0);
    constant HighImpedance : std_logic_vector((WordLength-1) downto 0) := (others => 'Z');
    constant memory: ROM_TYPE:=
    (   "01110000",
        "10100011",
        "00011111",
        "11010110",
        "00001111",
        "11010001",
        "00100000",
        "00011011",
        "00111001",
        "11010101",
        "11011101",
        "00010001",
        "11000001",
        "00010001",
        "00011010",
        "01110110",
        "10000000",
        "01101001",
        "00001010",
        "00000011",
        "01010110",
        "00100001",
        "01111101",
        "11100111",
        "01000010"
```

At this stage, the only missing part is the output functionality of the ROM, which is based on the CLK pulse and the input address.

We can do it by using a 'process' that monitor the CLK and every time it is changed, run their body. I consider a standard logic for my ROM. In every clock rising-edge, if the 'enable' input is equal to '0', the output is going to the High-Impedance mode (Z); otherwise, it shows the data stored in the given address on the address lines.



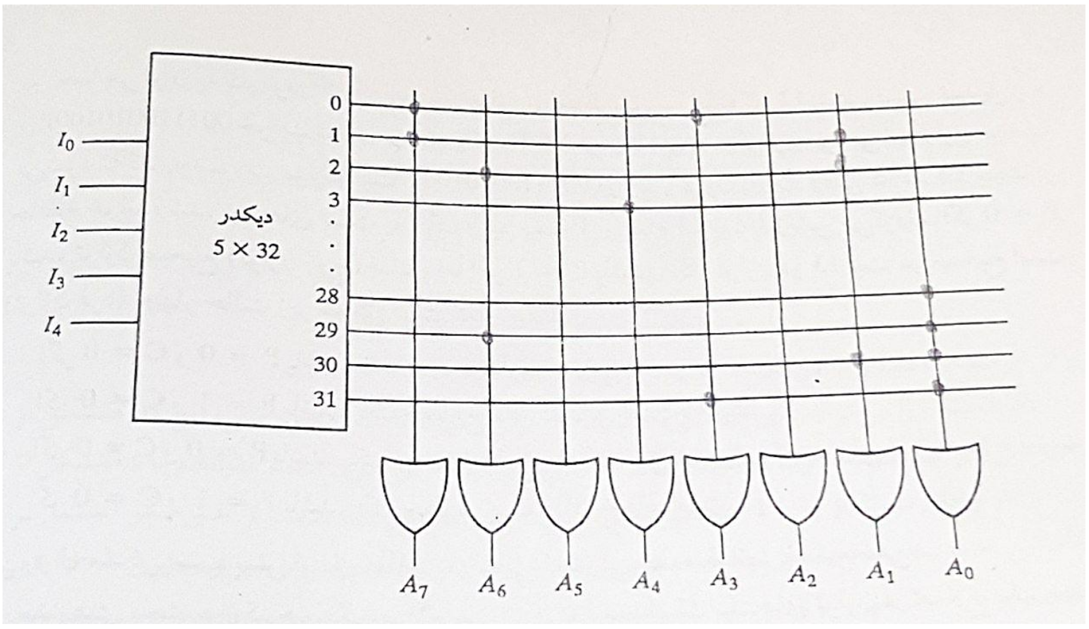
```
        "00101011",
        "01101010" );

    begin

    process(CLK) is
    begin
        if  rising_edge(CLK)    then
            if enable = '0' then
                data <= HighImpedance;
            else
                data <= memory(to_integer(unsigned(address)));
            end if;

        end if;
    end process;

end Behave;

-- Esfandiar Kiani - VHDL Course Final Project
```

```
# Compile Entity "ROM"
# Compile Architecture "Behave" of Entity "ROM"
# Compile success 0 Errors 0 Warnings  Analysis time :  0.5 [s]
# ULM: Warning: ULM_0021 Architecture "arch" of entity "myproject.rom" is not up-to-date. Please recompile file
 "C:\Users\Roshan\Documents\Active HDL\Es.Kiani.ROM.Project\MyProject/src/MyProject.vhd".
> acom -reorder -O3 -e 100 -work MyProject -2002  $dsn/src/MyProject.vhd
```

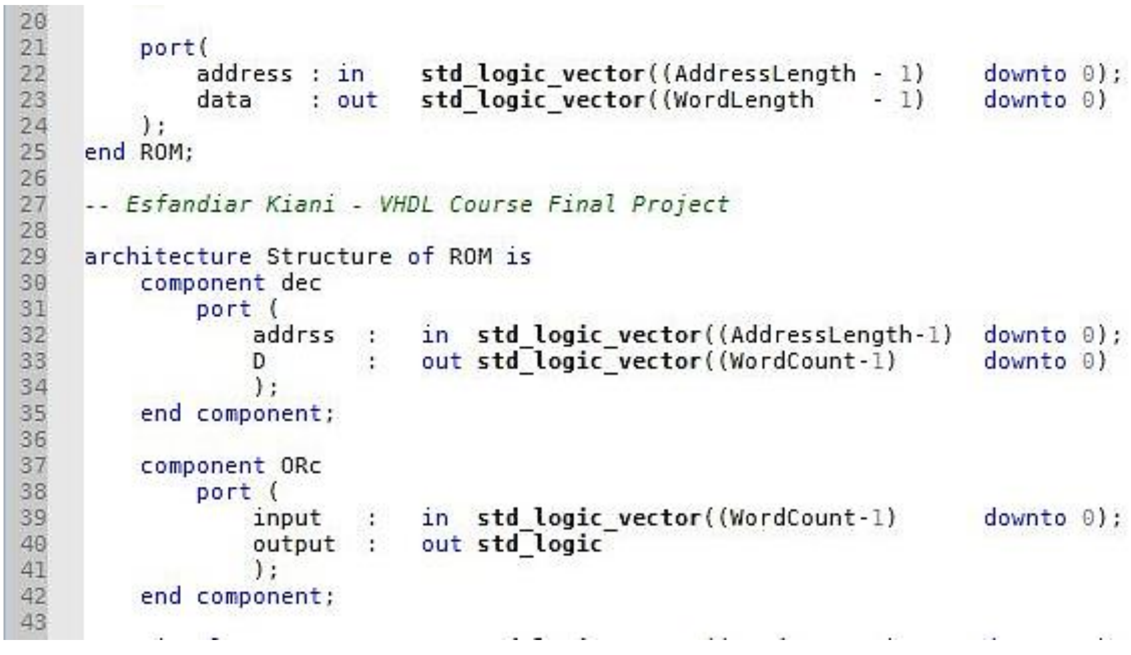
ii. Structural

After detailing the architecture, the memory array was also implemented structurally. This approach reutilizes the predefined dimensions for memory cells, applying the same initial random-generated data used in the behavioral implementation (refer to page-4) during the structural design.

In the structural method, an in-depth understanding of the internal circuitry of the memory array is essential to establish connections among various components explicitly in the code. Consequently, for the internally designed memory array, the following internal structure was taken into account



When detailing the internal circuitry of a memory array in a structural description, it is first necessary to define the various elements of the circuit as components. Subsequently, the connections between these components and the related input and output ports must be specified. Therefore, with reference to the internal architecture of the memory array, it becomes evident that there is a requirement for two distinct components: the decoder (‘dec’ is stand for ‘decoder’) and some logical ‘OR’ gates (‘Orc’ is stand for ‘OR Component’). These components must be appropriately sized to match the dimensions required by the memory array.


```
    port(
        address : in    std_logic_vector((AddressLength - 1)    downto 0);
        data    : out   std_logic_vector((WordLength    - 1)    downto 0)
    );
end ROM;

-- Esfandiar Kiani - VHDL Course Final Project

architecture Structure of ROM is
    component dec
        port (
            addrss   :   in  std_logic_vector((AddressLength-1)  downto 0);
            D        :   out std_logic_vector((WordCount-1)      downto 0)
            );
    end component;

    component ORc
        port (
            input    :   in  std_logic_vector((WordCount-1)      downto 0);
            output   :   out std_logic
            );
    end component;
```

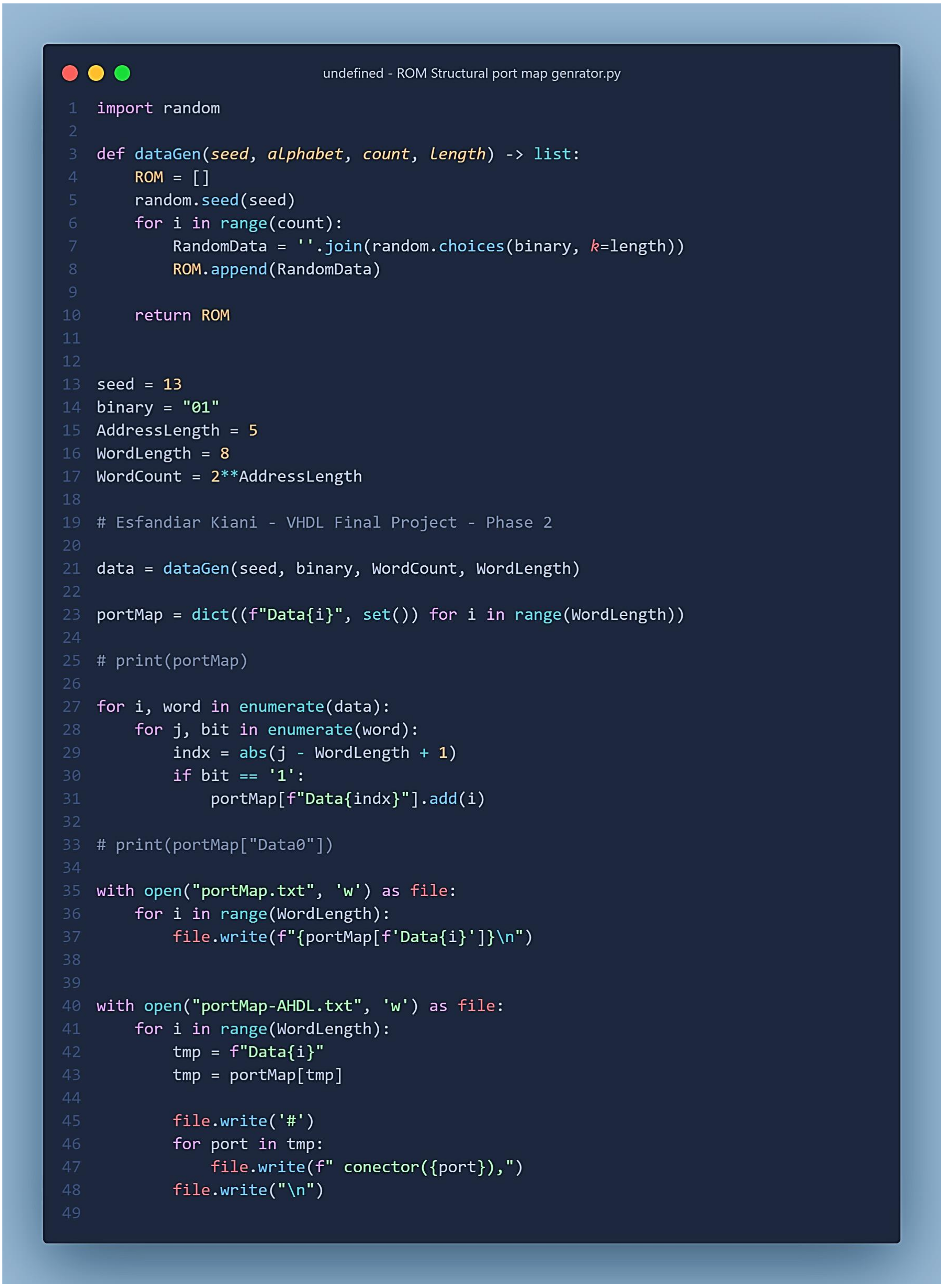
In this design approach for a ROM memory, the decoder selection lines correspond to the memory address lines, and the memory output represents the initial data.

During the manufacturing process, connecting the decoder's output to the memory output occurs for the bits of each word where the value equals 1. (For a clearer understanding, refer to page 10).

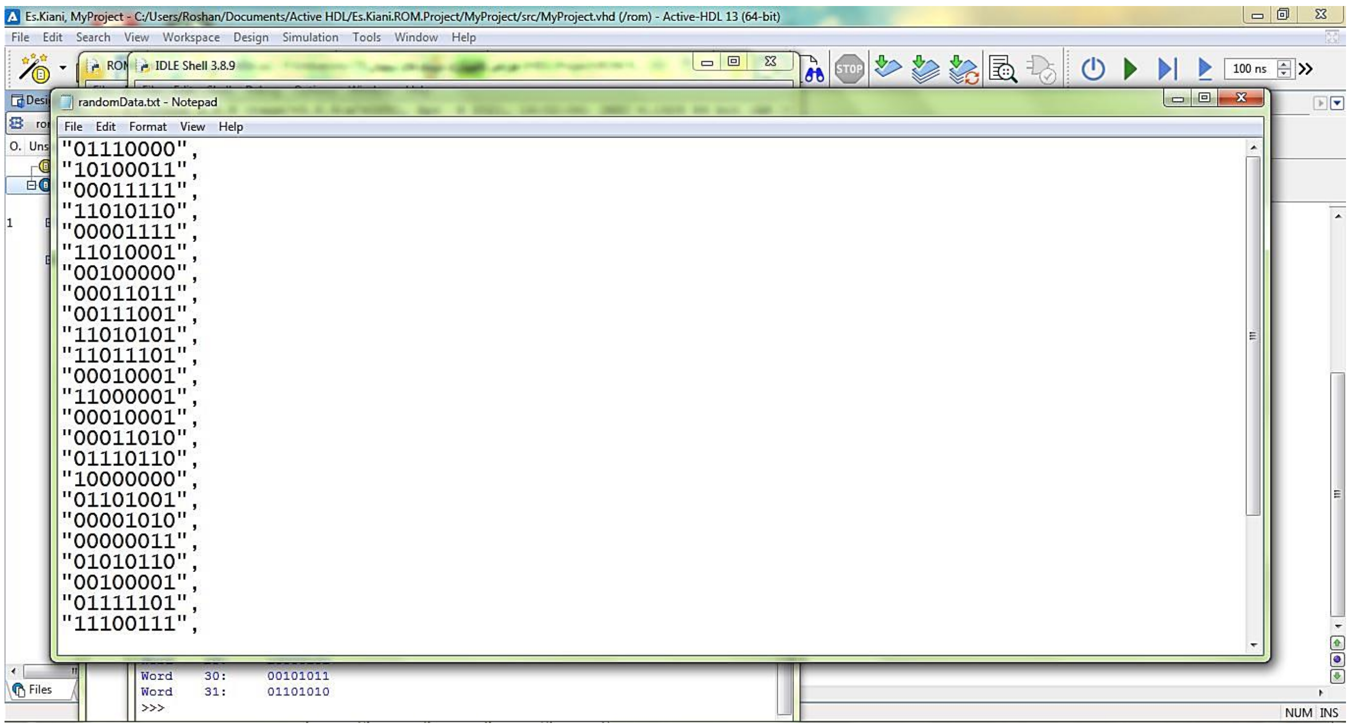
In essence, at any given moment, only one of the decoder's output lines will carry a logical '1', indicating a selected word from the memory. Let's assume in this word, the first and last bits have the value '1', and the rest are '0'. We need to connect this particular decoder output line to the corresponding memory output bits '0' and '7' and leave the rest unconnected for the memory output to match the intended word.

To systematically scale and execute this task, it is necessary to know which words in the initial data have '1' in their first-bit position to connect the decoder's output lines to the OR gate in bit position '1' of the memory output. This process is repeated for all the bits until completion. Accordingly, I modified the output functionality of the Python script that was initially written to generate random data. The revised script now systematically processes all words and for each of the eight memory output bits, it compiles a list. This list comprehensively enumerates the words from the memory where the bit value is equal to 1.

As anticipated, the script's output aligned perfectly with the results of manual calculations and, consistent with the initial data, demonstrated complete accuracy. This concurrence signifies that we can now proceed to codify these connections within the VHDL code. Hence, we are well-positioned to structurally design the internal circuitry of the ROM memory with success assured at this juncture.

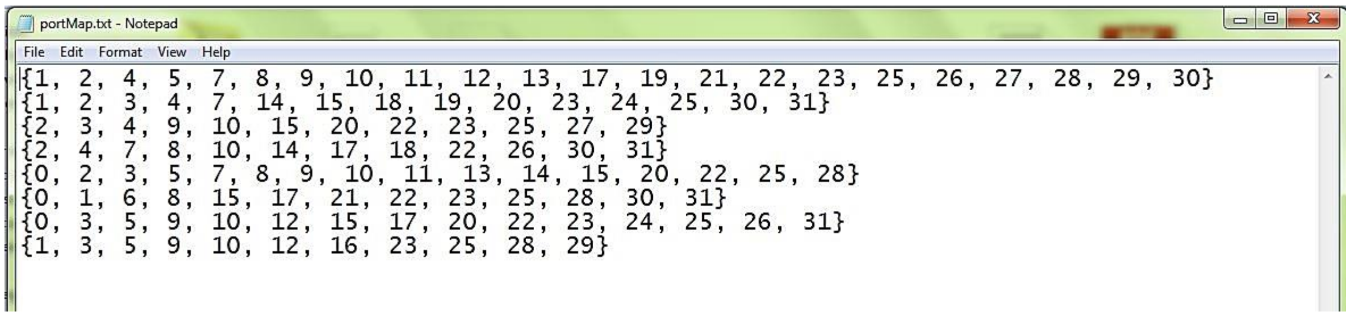


```python
import random

def dataGen(seed, alphabet, count, length) -> list:
    ROM = []
    random.seed(seed)
    for i in range(count):
        RandomData = ''.join(random.choices(binary, k=length))
        ROM.append(RandomData)

    return ROM


seed = 13
binary = "01"
AddressLength = 5
WordLength = 8
WordCount = 2**AddressLength

# Esfandiar Kiani - VHDL Final Project - Phase 2

data = dataGen(seed, binary, WordCount, WordLength)

portMap = dict((f"Data{i}", set()) for i in range(WordLength))

# print(portMap)

for i, word in enumerate(data):
    for j, bit in enumerate(word):
        indx = abs(j - WordLength + 1)
        if bit == '1':
            portMap[f"Data{indx}"].add(i)

# print(portMap["Data0"])

with open("portMap.txt", 'w') as file:
    for i in range(WordLength):
        file.write(f"{portMap[f'Data{i}']}\n")


with open("portMap-AHDL.txt", 'w') as file:
    for i in range(WordLength):
        tmp = f"Data{i}"
        tmp = portMap[tmp]

        file.write('#')
        for port in tmp:
            file.write(f" conector({port}),")
        file.write("\n")

```


Es.Kiani, MyProject - C:/Users/Roshan/Documents/Active HDL/Es.Kiani.ROM.Project/MyProject/src/MyProject.vhd (/rom) - Active-HDL 13 (64-bit)
File Edit Search View Workspace Design Simulation Tools Window Help
IDLE Shell 3.8.9
randomData.txt - Notepad
File Edit Format View Help
100 ns
Files
NUM INS

```
"01110000",
"10100011",
"00011111",
"11010110",
"00001111",
"11010001",
"00100000",
"00011011",
"00111001",
"11010101",
"11011101",
"00010001",
"11000001",
"00010001",
"00011010",
"01110110",
"10000000",
"01101001",
"00001010",
"00000011",
"01010110",
"00100001",
"01111101",
"11100111",
```

```
Word    30:     00101011
Word    31:     01101010
>>>
```



```
{1, 2, 4, 5, 7, 8, 9, 10, 11, 12, 13, 17, 19, 21, 22, 23, 25, 26, 27, 28, 29, 30}
{1, 2, 3, 4, 7, 14, 15, 18, 19, 20, 23, 24, 25, 30, 31}
{2, 3, 4, 9, 10, 15, 20, 22, 23, 25, 27, 29}
{2, 4, 7, 8, 10, 14, 17, 18, 22, 26, 30, 31}
{0, 2, 3, 5, 7, 8, 9, 10, 11, 13, 14, 15, 20, 22, 25, 28}
{0, 1, 6, 8, 15, 17, 21, 22, 23, 25, 28, 30, 31}
{0, 3, 5, 9, 10, 12, 15, 17, 20, 22, 23, 24, 25, 26, 31}
{1, 3, 5, 9, 10, 12, 16, 23, 25, 28, 29}
```

To better understand the output produced in this part:

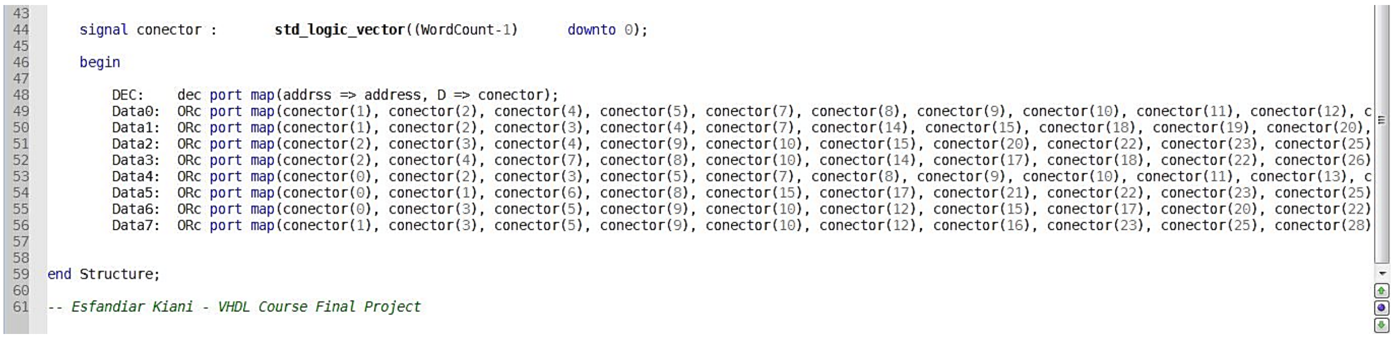
The output file has 8 lines, each level representing one bit of the memory output (our memory has 8-bit words).

Output: $O_1O_2O_3O_4O_5O_6O_7O_8$

The first line represents O_1, the second line represents O_2, …

And in each line, we see some numbers between 0 and 31. For instance, it means the words number 1 and 2, have '1' value on their O_1 bit.

Now, by having this data I can write a structural code for my ROM. I pass the decoder outputs into a conector signal, and connect the relative outputs of RAM to the coresponding bits of the conector signal to generate the correct data when it is turned on.



```

    signal conector :       std_logic_vector((WordCount-1)      downto 0);

    begin

        DEC:    dec port map(addrss => address, D => conector);
        Data0:  ORc port map(conector(1), conector(2), conector(4), conector(5), conector(7), conector(8), conector(9), conector(10), conector(11), conector(12), c
        Data1:  ORc port map(conector(1), conector(2), conector(3), conector(4), conector(7), conector(14), conector(15), conector(18), conector(19), conector(20),
        Data2:  ORc port map(conector(2), conector(3), conector(4), conector(9), conector(10), conector(15), conector(20), conector(22), conector(23), conector(25)
        Data3:  ORc port map(conector(2), conector(4), conector(7), conector(8), conector(10), conector(14), conector(17), conector(18), conector(22), conector(26)
        Data4:  ORc port map(conector(0), conector(2), conector(3), conector(5), conector(7), conector(8), conector(9), conector(10), conector(11), conector(13), c
        Data5:  ORc port map(conector(0), conector(1), conector(6), conector(8), conector(15), conector(17), conector(21), conector(22), conector(23), conector(25)
        Data6:  ORc port map(conector(0), conector(3), conector(5), conector(9), conector(10), conector(12), conector(15), conector(17), conector(20), conector(22)
        Data7:  ORc port map(conector(1), conector(3), conector(5), conector(9), conector(10), conector(12), conector(16), conector(23), conector(25), conector(28)


end Structure;

-- Esfandiar Kiani - VHDL Course Final Project
```