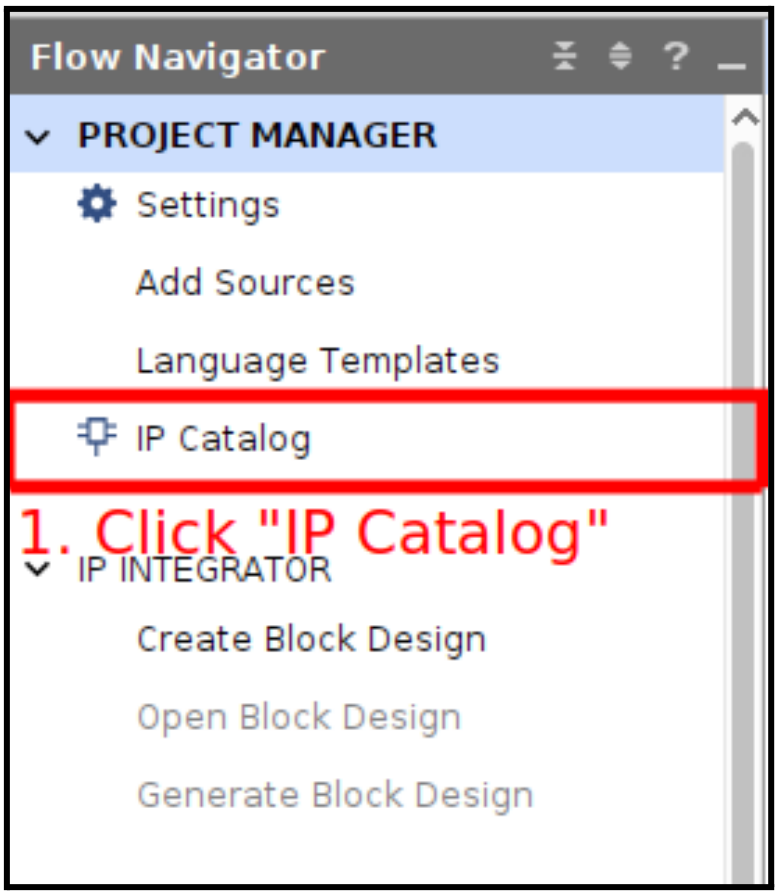
IP Catalog

What is IP Catalog

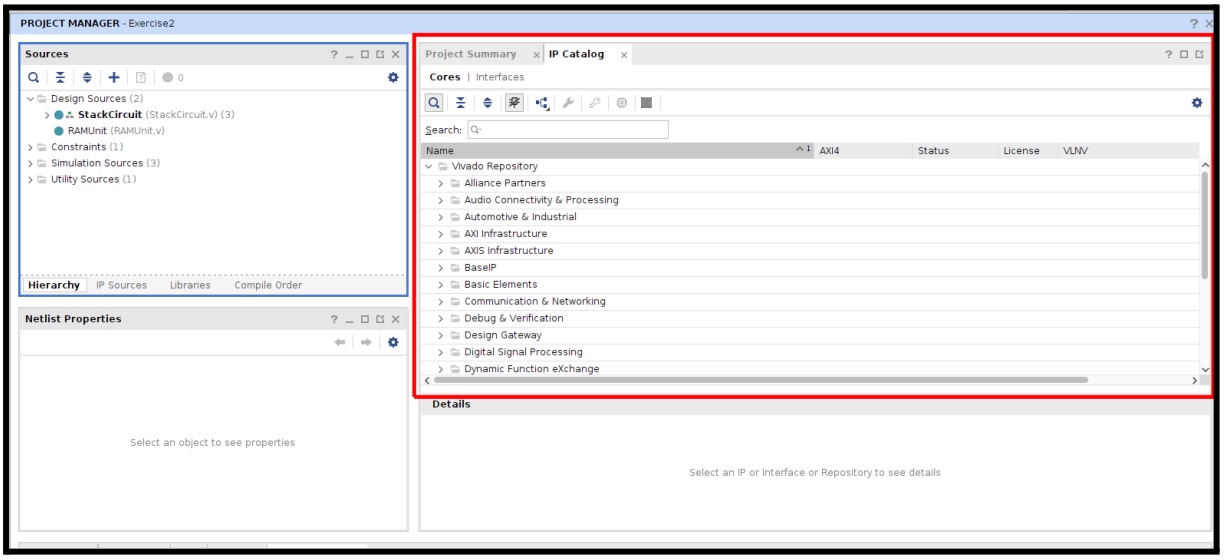
The Xilinx IP Catalog is a feature in Vivado that provides a library of pre-made building blocks, called IP cores, that you can use in your designs. These IP cores save time by letting you use tested and ready-to-use components instead of designing everything from scratch. It's like a toolbox where you can pick and configure parts for your project, such as memory controllers, arithmetic blocks, or communication interfaces.

How to generate an IP to be used in our project

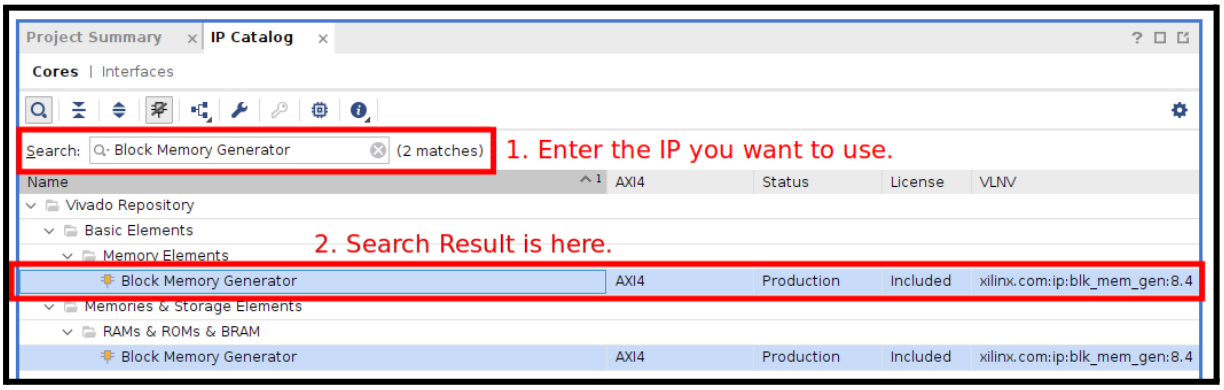
1. Open the IP Catalog.
    a. Click on the "IP Catalog".



b. The IP Catalog will appear.

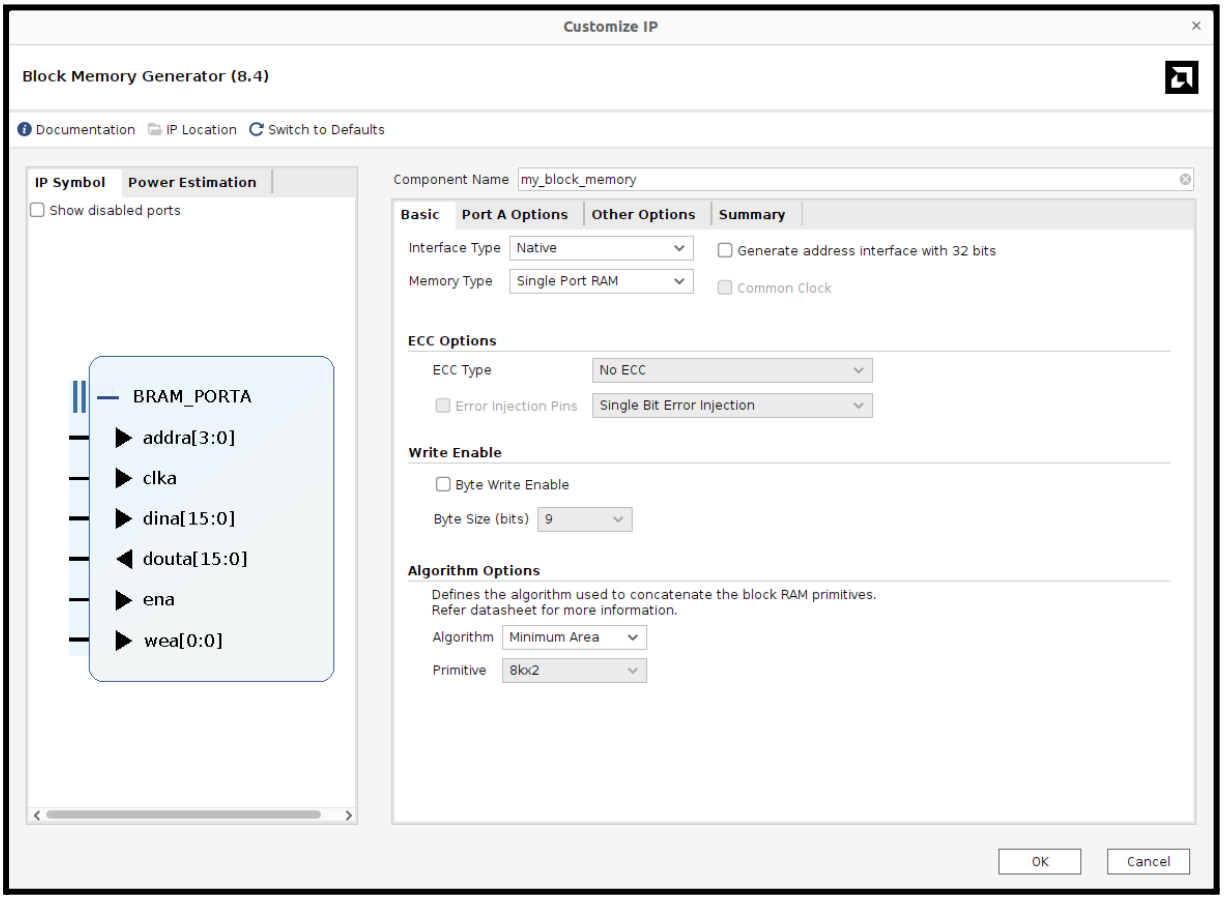


2. Search for the IP you want to use.
   a. For example we are going to search for "Block Memory Generator".



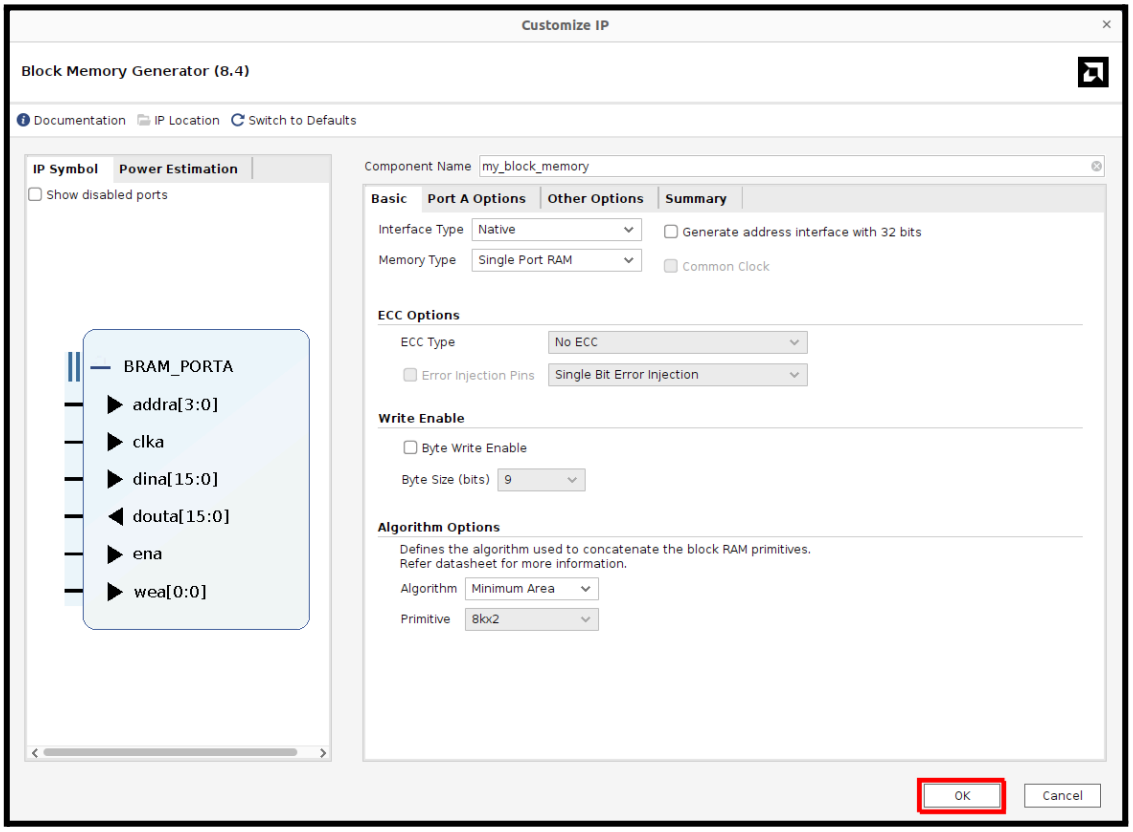
   b. Double click on the "Block Memory Generator".

c. The Customization Window will appear.

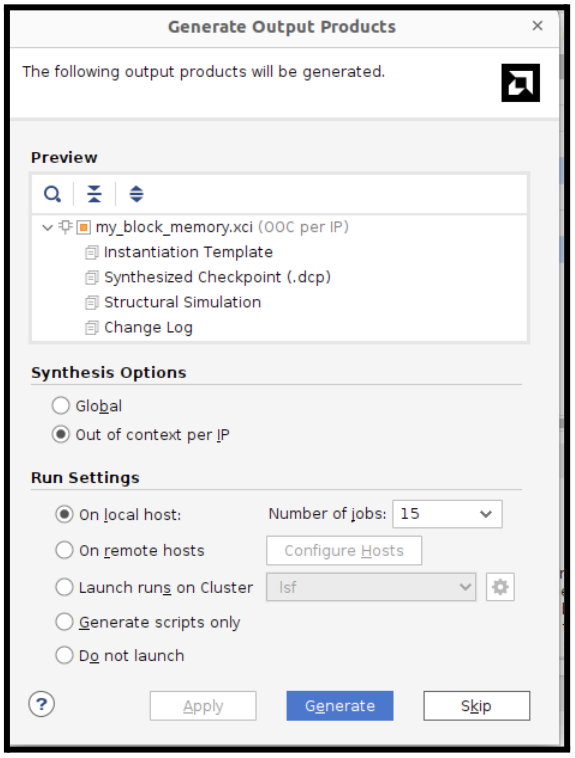


3. Customize your module to suit your needs.

4. After you have finished , Click OK



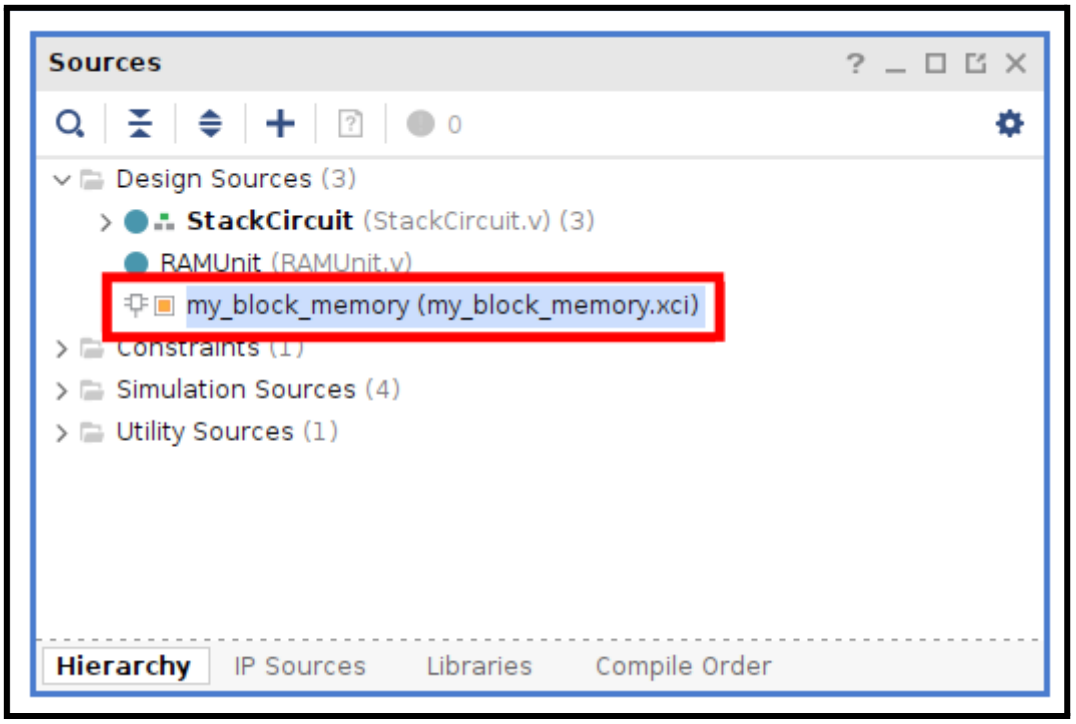
5. The Generate Output Product window will appear, you can click either the 'Generate' or 'Skip' button. The generated IP will be synthesized as an 'Out of Context Module.'



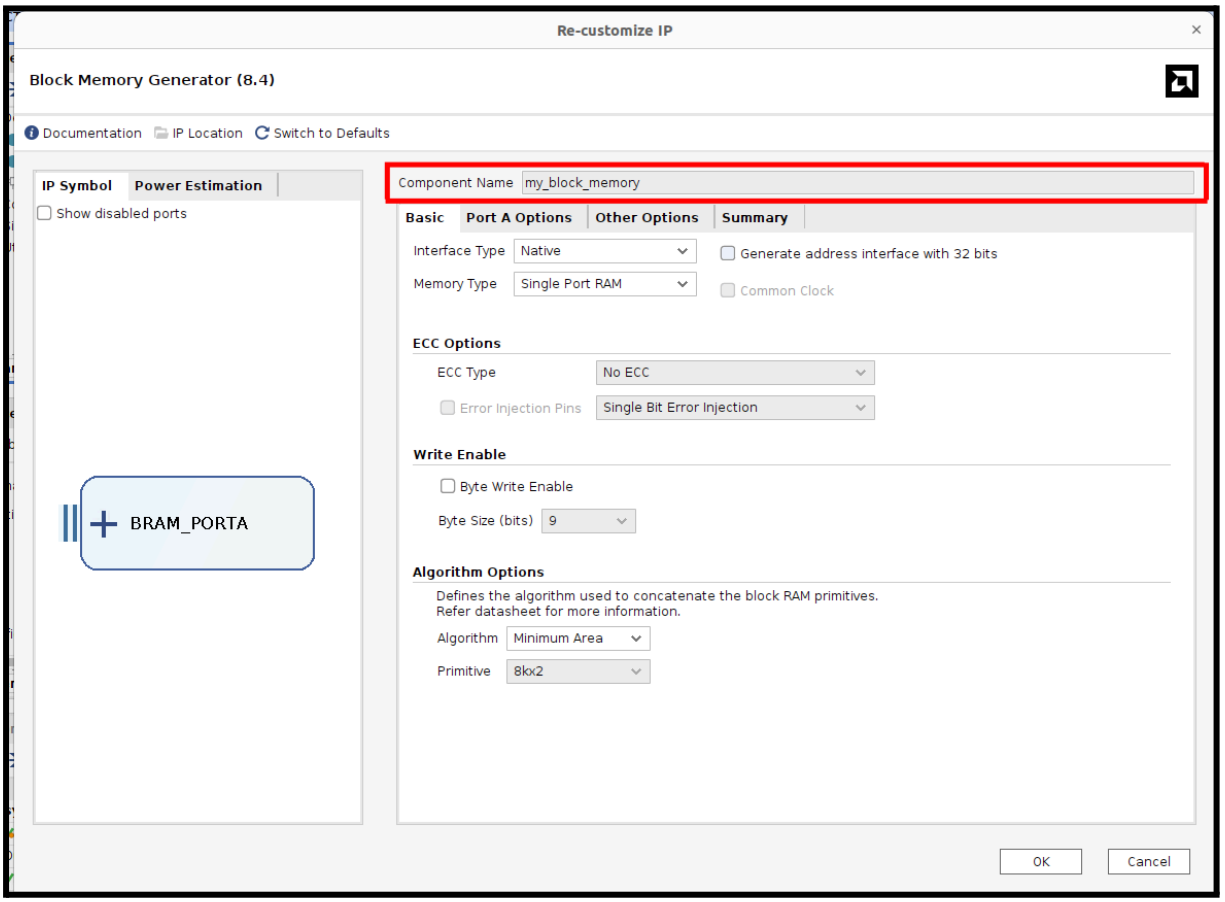
How to use the IP we generated

After generating the IP from the IP Catalog, you can instantiate it in the same way as a standard module that you have written.

1. From the previous step, we have create a “Block Memory Generator” name “my_block_memory”
2. Check the IP name and port.
    a. double click the IP that you have generated.

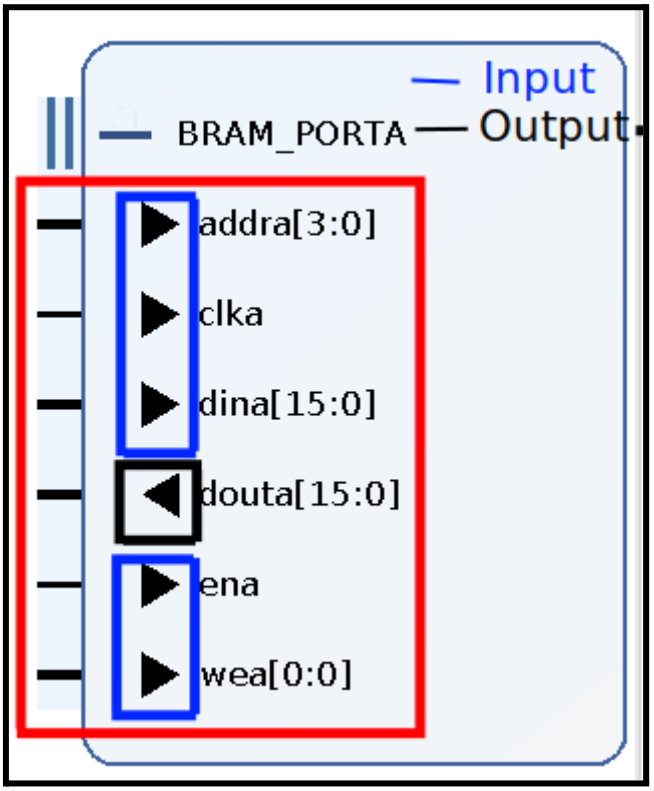


b. This is the component name.



c. On the “IP-Symbol” tap, click the “+” sign to see the module port.

d. The module port name is highlighted with a red rectangle. Ports highlighted in blue represent input ports, while those highlighted in black represent output ports.



3. Go to the module file that you want to include this IP.
4. Add this module to that file.
    a. In this example file, we create a wrapper for the IP. You can use the IP just like any other module you have written.

```
`timescale 1ns / 1ps
module Example (
  input wire [3:0] addra,
  input wire clka,
  input wire [7:0] dina,
  output wire [7:0] douta,
  input wire ena,
  input wire wea
);
 // Instantiate the IP
 my_block_memory my_block_memory_inst (
    .addra(addra),
    .clka (clka),
    .dina (dina),
    .douta(douta),
```

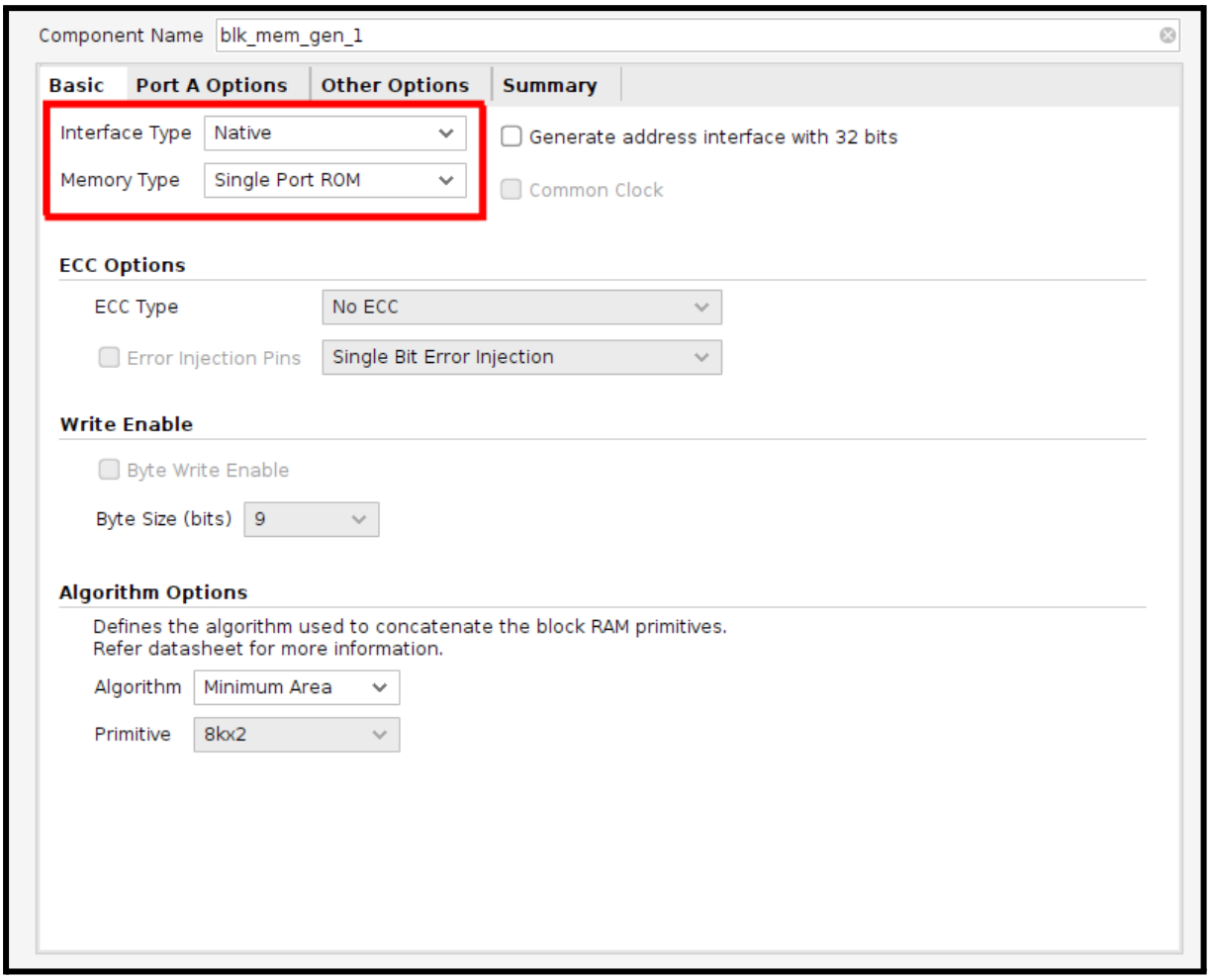
```
    .ena  (ena),
    .wea  (wea)
);
endmodule
```

b. After the heiracle update, it should look like this.

Read Only Memory (ROM)

How to customize ROM (IP Catalog)

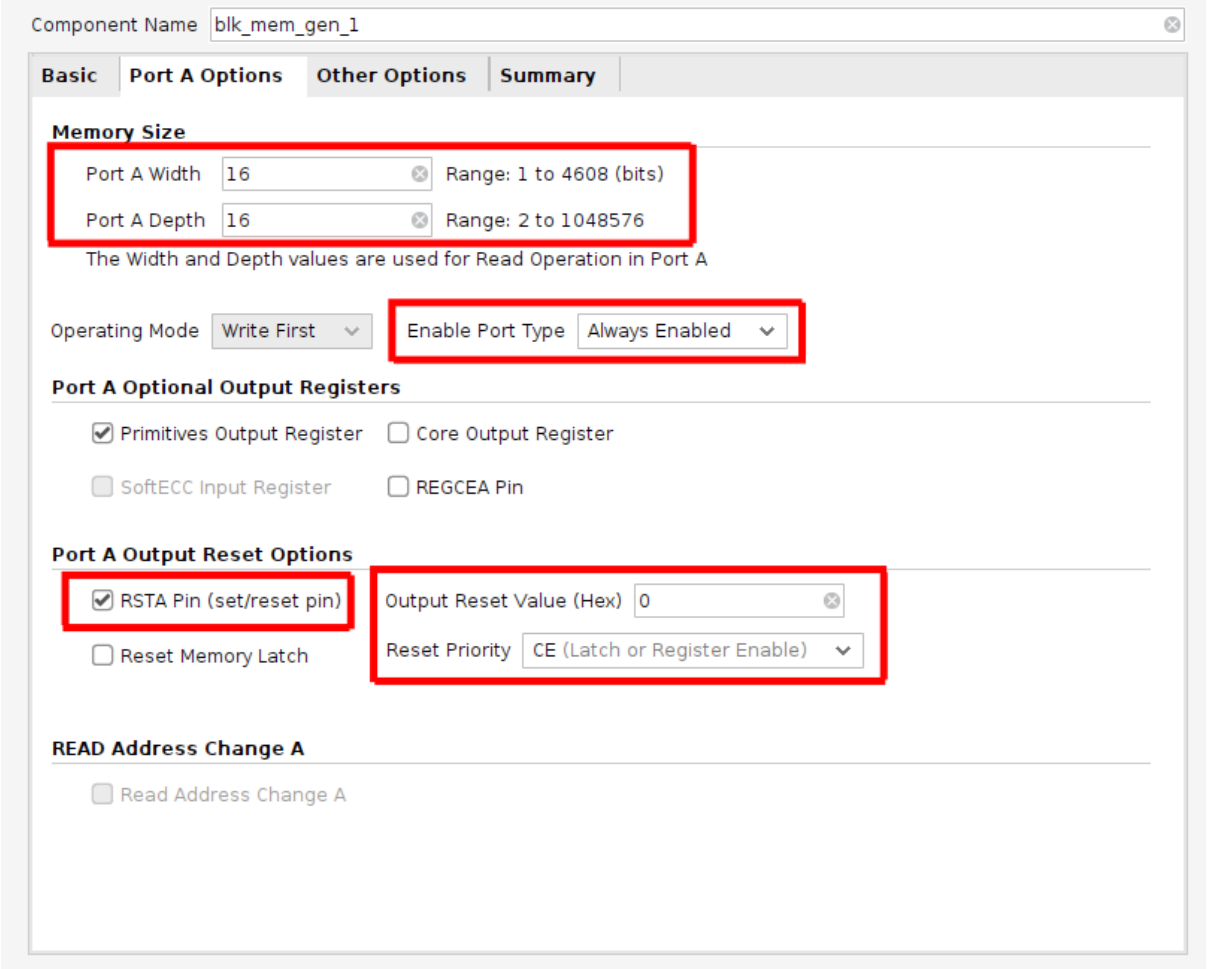
1. Generate the “Block Memory Generator” IP from IP Catalog.
2. On the “Basic” tap, ensure that the Interface Type is “Native” and Memory type is “Single Port ROM”.



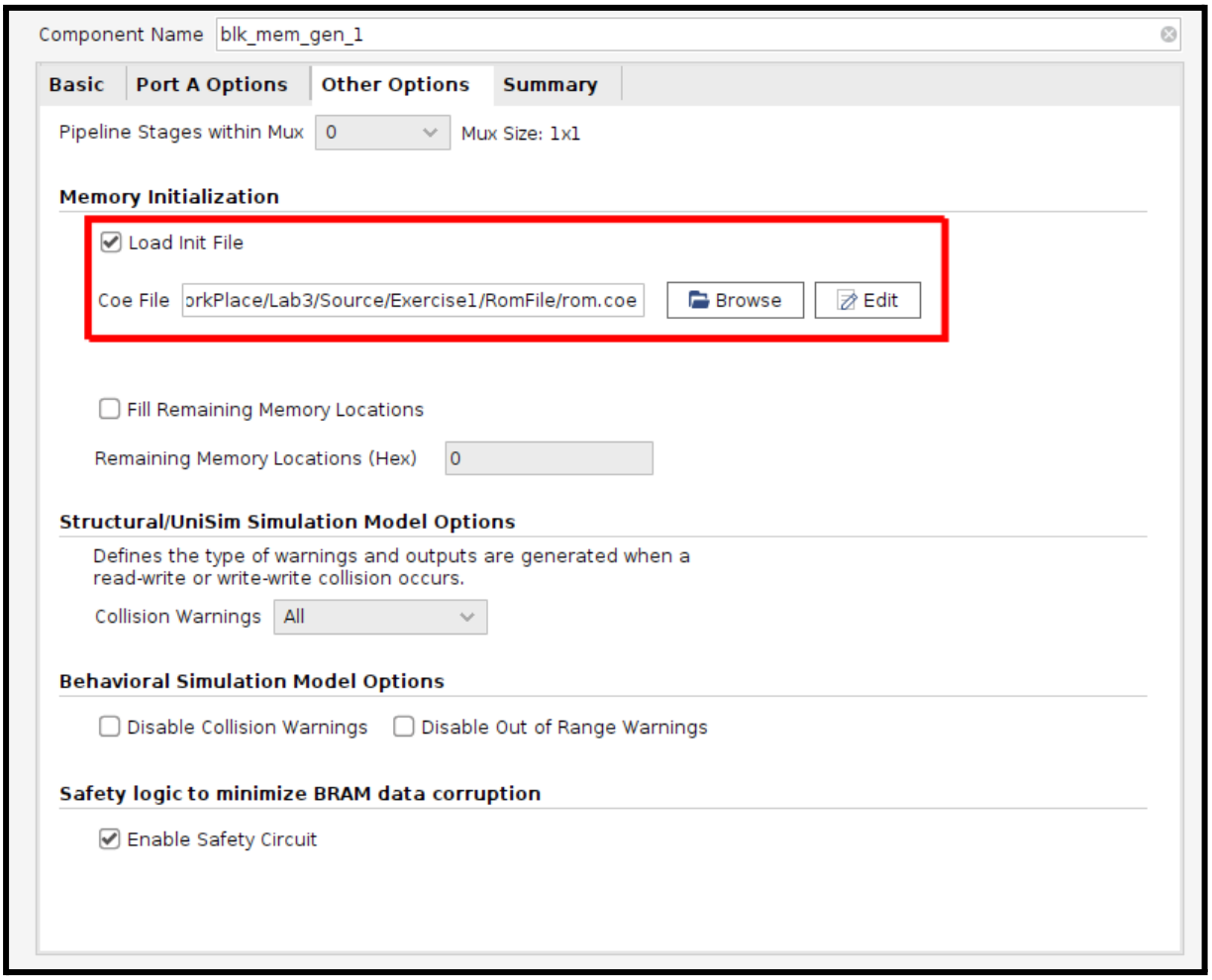
3. On the “Port A Options” tap, ensure that :
    a. The port width and depth are configured based on your requirements.

The port width defines the width of the ROM data output, specifying the number of bits in each data entry.
The port depth defines the number of data entries (or addresses) stored in the ROM.

    b. Enable Port Type is “Always Enabled”. (we read data every clock)
    c. RSTA Pin is enabled and Output Reset Value is 0.
    d. Reset Priority is “CE (Latch or Register Enable)”



4. On the Other Options tap, ensure that you load the appropriate .coe file to initialize the ROM data.



How to initialize ROM value

Path 1: Your own written ROM module

1. Generate a .mem file that contains value for each ROM address.
    a. This is an example of a .mem file for a ROM with a data width of 5 bits and 16 addresses, where the data in each ROM address is the address value plus 1.

```
00001
00010
00011
00100
00101
00110
00111
01000
01001
01010
01011
01100
01101
01110
01111
10000
```

    b. We recommend that you write a program to generate the .mem file. This is the example python program to generate the upper .mem file.

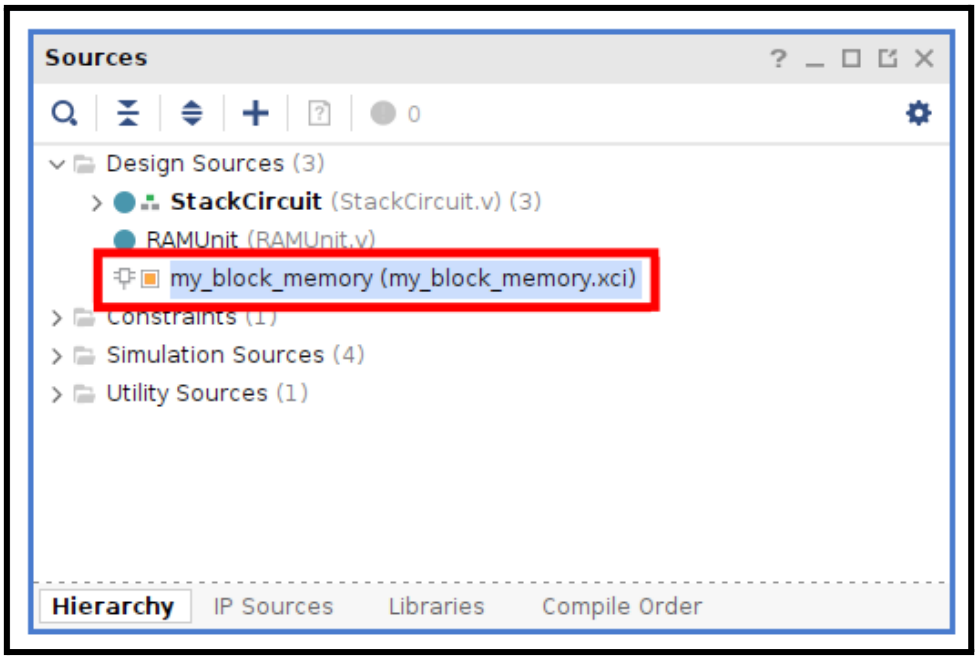
```
file_name = "rom.mem"
with open(file_name, "w") as f:
  for i in range (16):
     for j in [16, 8, 4, 2, 1]:
        f.write("1" if (i+1)&j else "0")
     f.write("\n")
```

2. Add this part of code to your ROM module to initialize the ROM according to the .mem file you have generated.

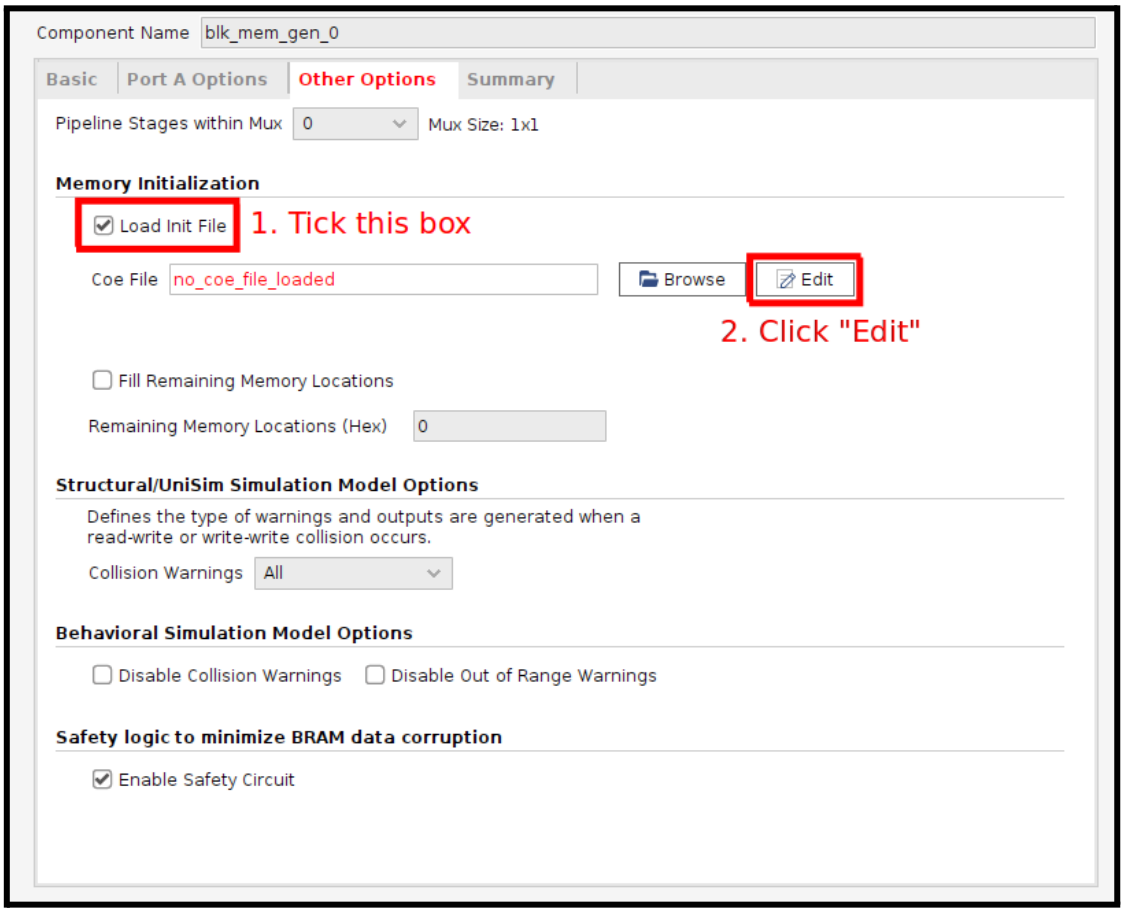
```
reg [7:0] mem[63:0]; // the register that is used to construct ROM
initial begin
  $readmemb("<path to your rom.mem file>/rom.mem",mem);
end
```

Path 2: ROM from IP Catalog

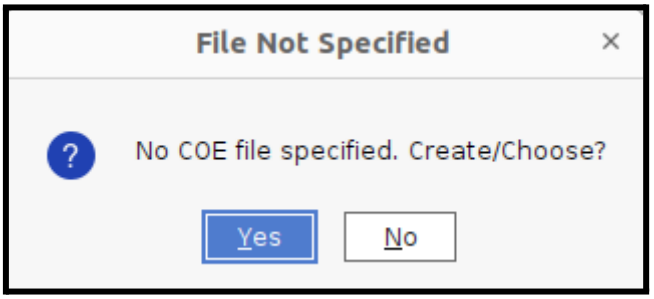
1. Generate a .coe file that contains value for each ROM address.
    a. Open your IP customization by double clicking your generated IP.



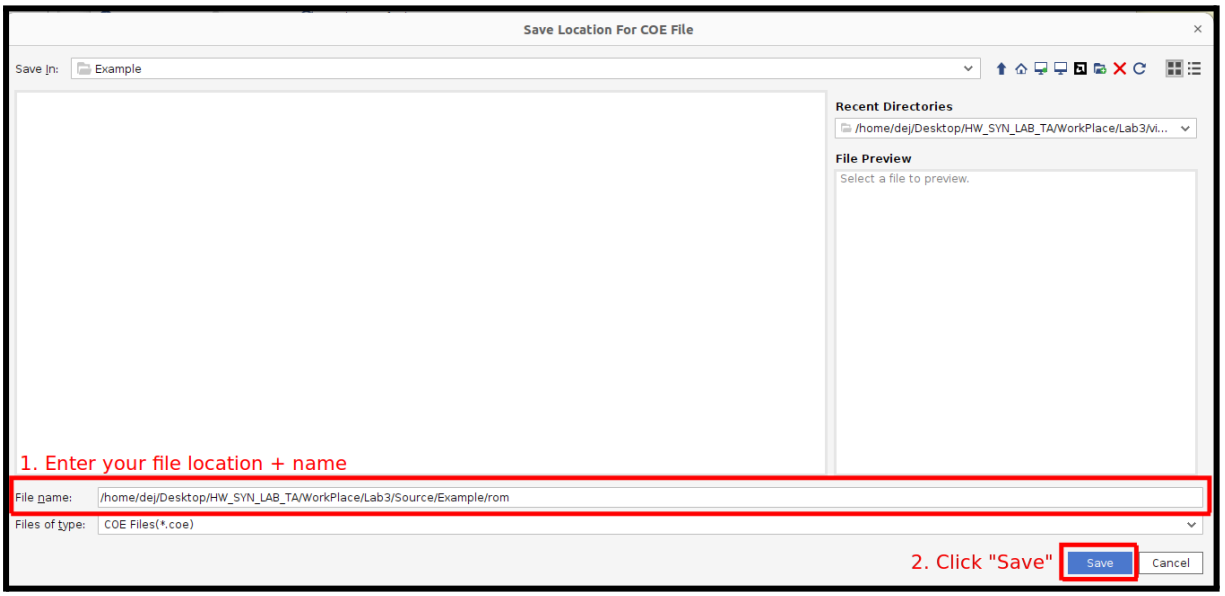
    b. On the “Other Options” tap, tick the “Load init file” and click “Edit”



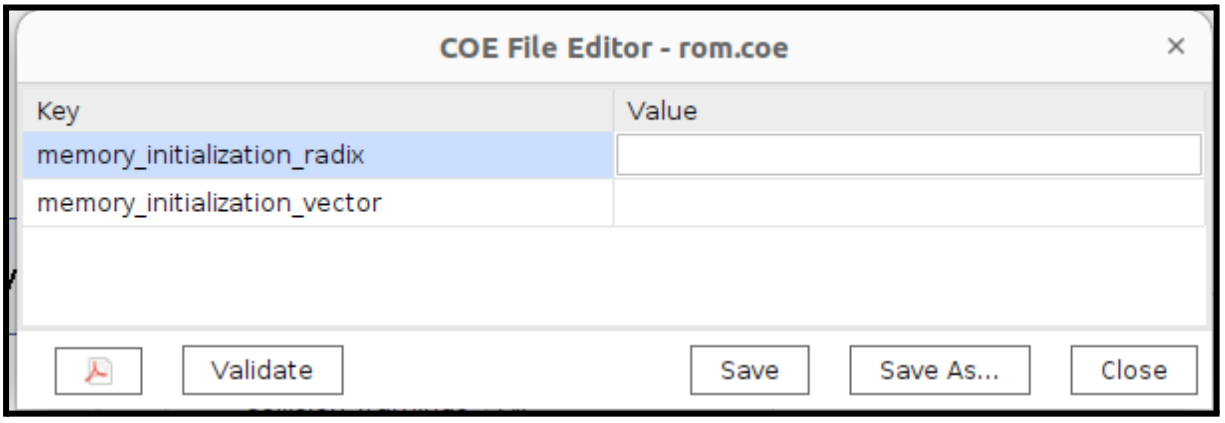
c. This window should pop up, Click “Yes”.



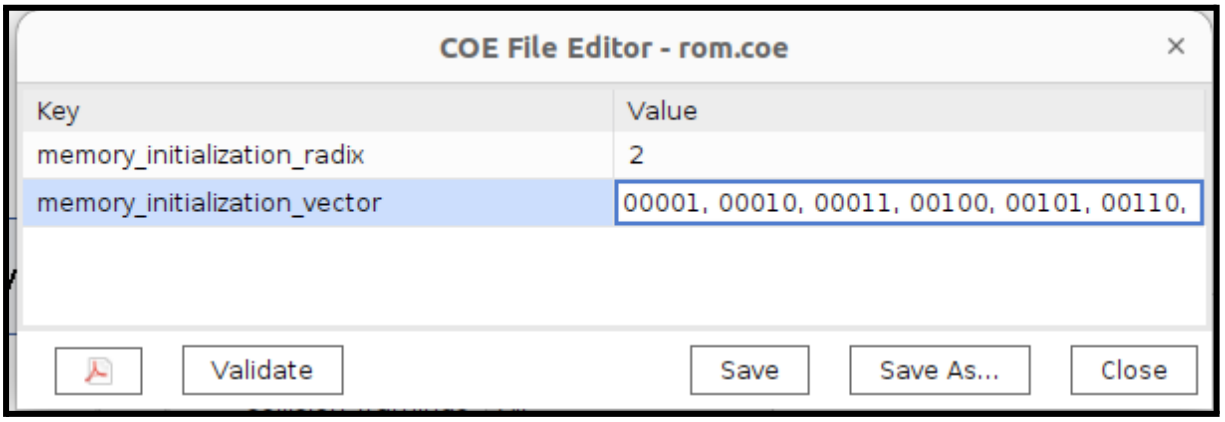
d. Enter your file location and name, Then click “Save”.



e. This Window should pop up.



f. Initialize the radix and vector.
   i. The radix defines the base for input values. If it is set to 2, the input values will be interpreted as binary (base 2).
   ii. The vector is the value of each Address in the ROM.
   iii. Input the radix and vector
   iv. You can click “Validate” to validate your data.
   v. After you have finished, click “Save” then “Close”.



g. For the vector we recommend writing a program to generate it. This is an example of python program that use to generate the above

```
file_name = "rom.mem"
with open(file_name, "w") as f:
  for i in range (16):
    for j in [16, 8, 4, 2, 1]:
      f.write("1" if (i+1)&j else "0")
    f.write(",\n")
```

h. You can then copy the content in the rom.mem then paste it into the memory_initialization_vector.

i. This is an example of a .mem file for a ROM with a data width of 5 bits and 16 addresses, where the data in each ROM address is the address value plus 1.

```
memory_initialization_radix=2;
memory_initialization_vector=00001, 00010, 00011, 00100, 00101, 00110,
00111, 01000, 01001, 01010, 01011, 01100, 01101, 01110, 01111, 10000, ;
```

ROM behavior (IP Catalog)

The DOUTA output of the ROM is driven based on the address input and is updated on the positive edge of the clock. If the RSTA signal is asserted at the positive edge of the clock, the DOUTA will be reset to its initial value (in this case, 0) on the next clock cycle. Figure 1 below shows the timing diagram of the ROM unit from the IP Catalog.

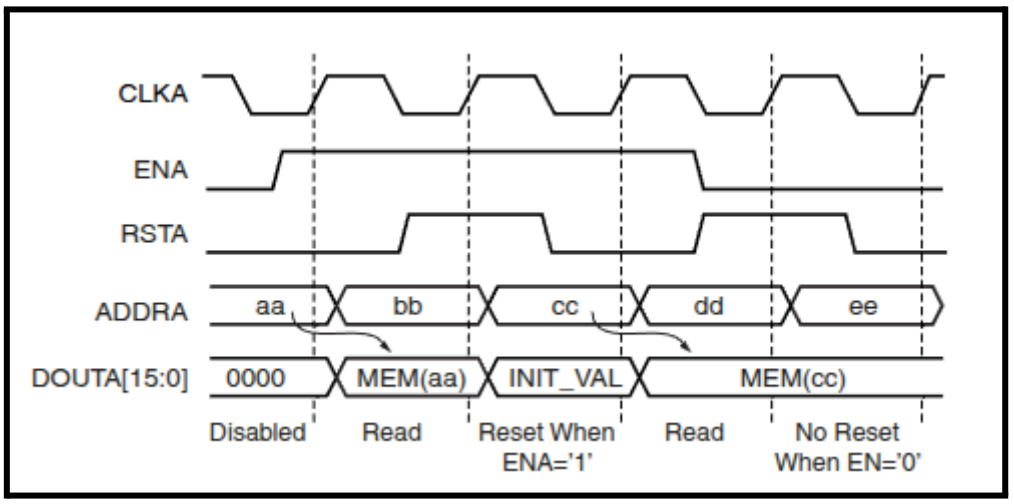


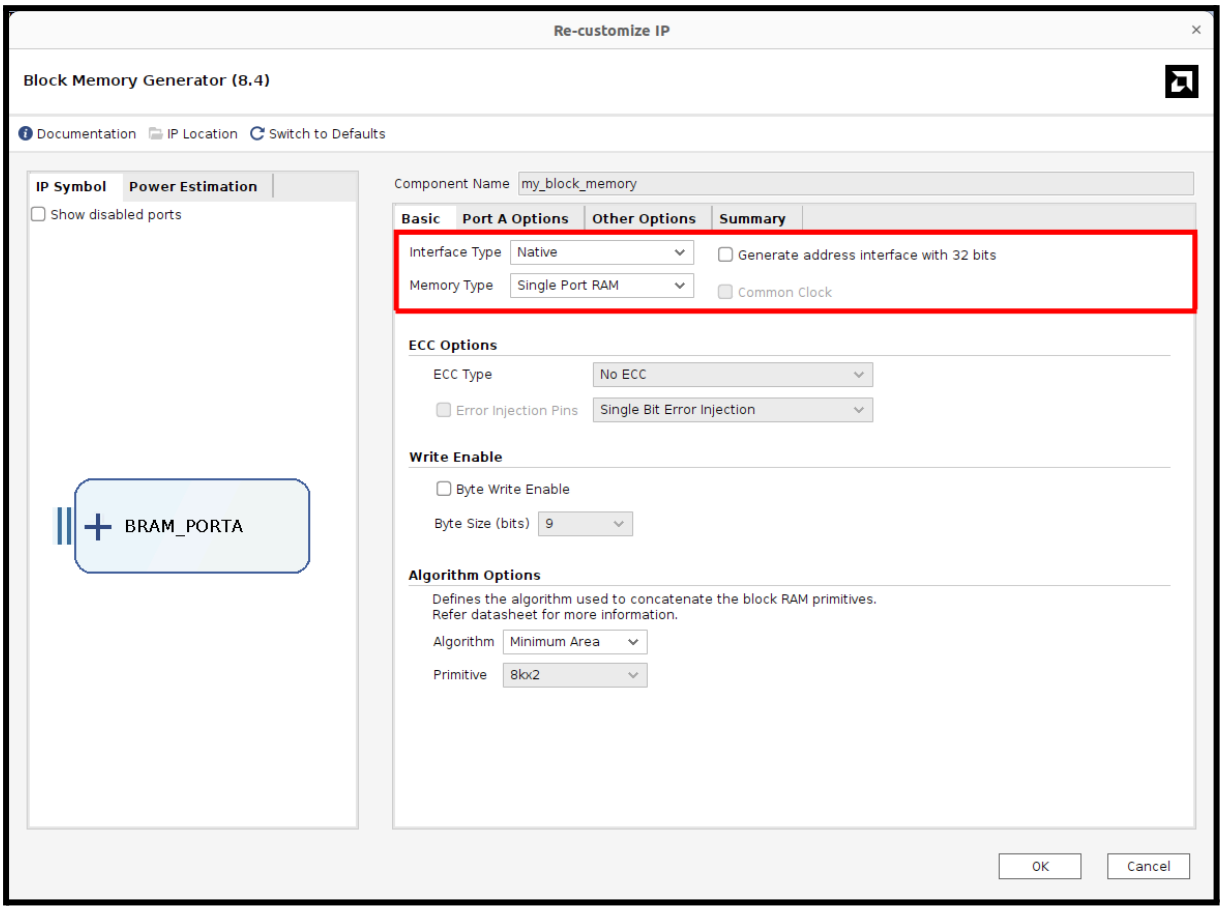
Figure 1 : ROM Timing Diagram

For further information please read this document.

Random-access memory (RAM)

How to customize RAM (IP Catalog)

1. Generate the “Block Memory Generator” IP from IP Catalog.
2. On the “Basic” tap, ensure that the Interface Type is “Native” and Memory type is “Single Port RAM”.

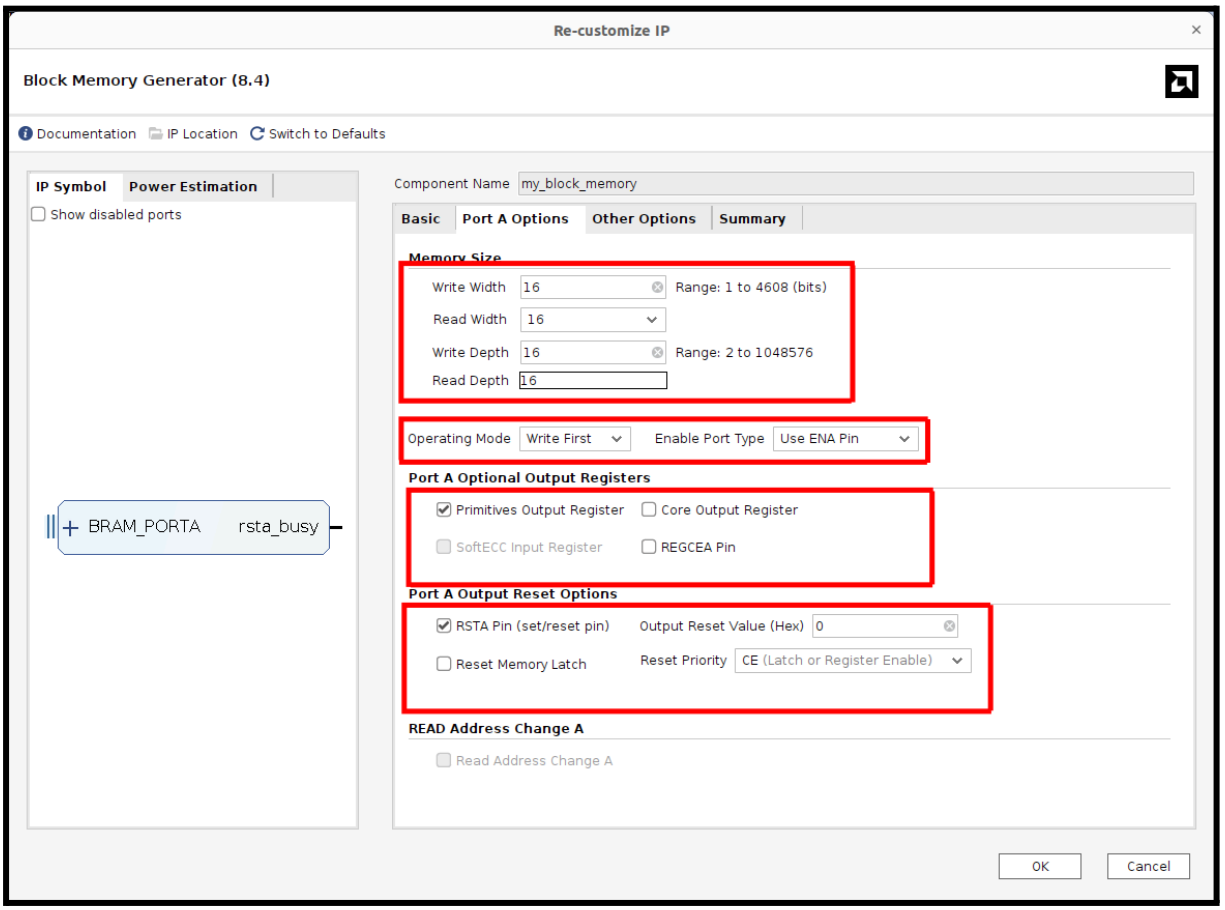


3. On the “Port A Options” tap, ensure that :
    a. The port Write/Read width and depth are configured based on your requirements.

The port width defines the width of the RAM data output, specifying the number of bits in each data entry.

The port depth defines the number of data entries (or addresses) stored in the RAM.

    b. You can use any RAM Operating Mode.
    c. Enable Port Type is “Use ENA Pin”. (we want more control here)
    d. Primitives Output Register is selected.
    e. RSTA Pin is enabled and Output Reset Value is 0.
    f. Reset Priority is “CE (Latch or Register Enable)”



RAM behavior (IP Catalog)

The RAM has two operations: read and write, which are controlled by the ENA and WEA signals. The output DOUTA will only change at the positive edge of the clock. If the ENA signal is not asserted, the RAM is disabled and will not perform any read or write operations (the output DOUTA will remain unchanged). If ENA is asserted but WEA is not, the RAM will only perform a read operation. When both ENA and WEA are asserted, the RAM will perform both read and write operations, but its behavior will depend on its operation mode.

Operation Mode : Write First

In this mode, the output DOUTA will be the same as the input DINA.

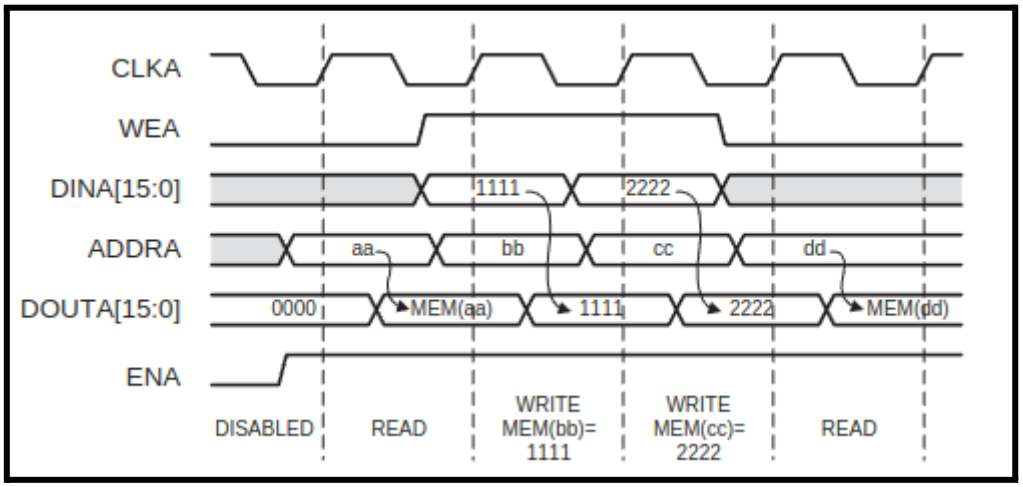


Figure 2 : RAM with Write First Mode Timing Diagram

Operation Mode : Read First

In this mode, the output DOUTA will reflect the old value stored at the address ADDRA before the new data (DINA) is written.

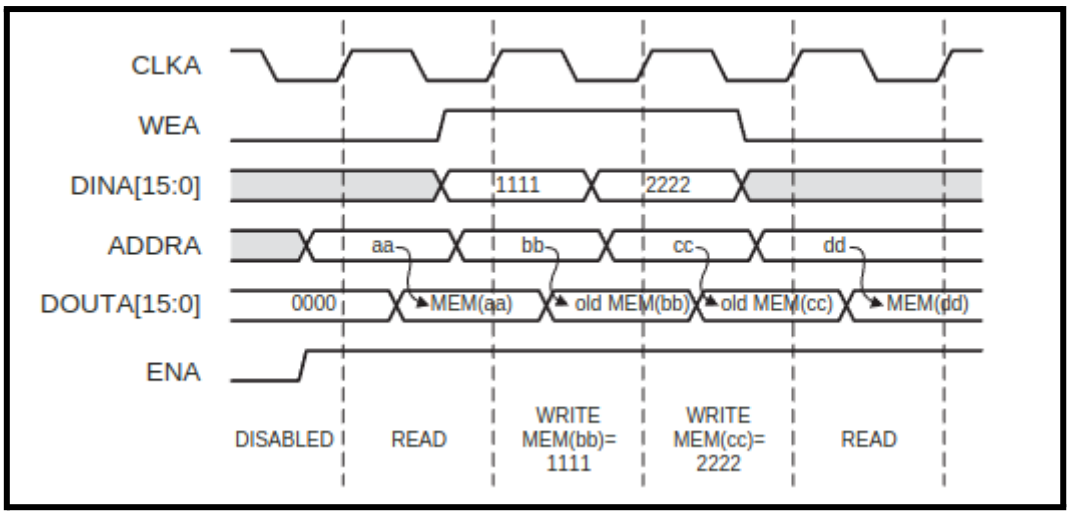


Figure 3 : RAM with Read First Mode Timing Diagram

Operation Mode : No Change

In this mode, the output DOUTA will change only when the RAM performs a read operation (when ENA is asserted but WEA is not).

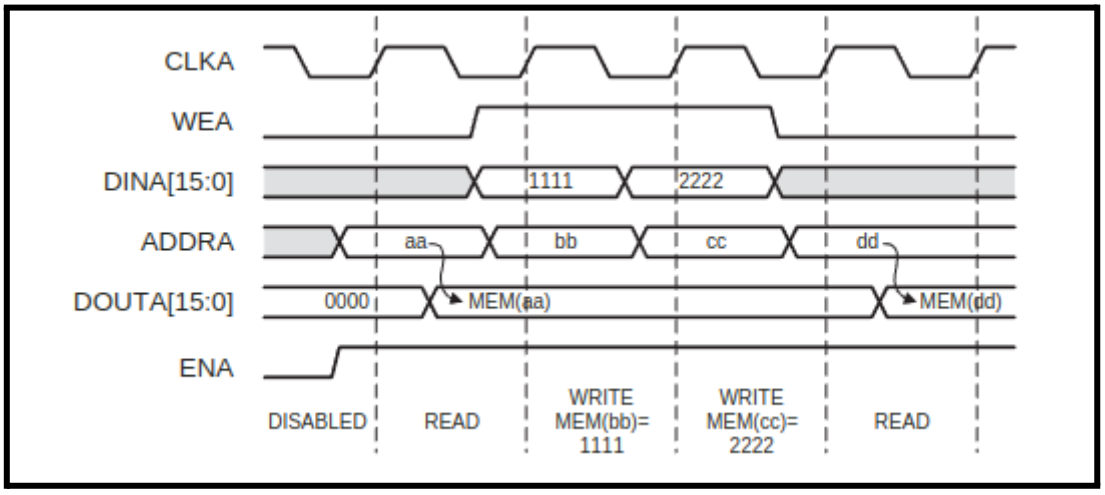


Figure 4 : RAM with No Change Mode Timing Diagram

Reset Behavior

If the RSTA signal is asserted at the positive edge of the clock, the DOUTA will be reset to its initial value (in this case, 0) on the next clock cycle. Figure 1 below shows the timing diagram of the RAM unit from the IP Catalog.

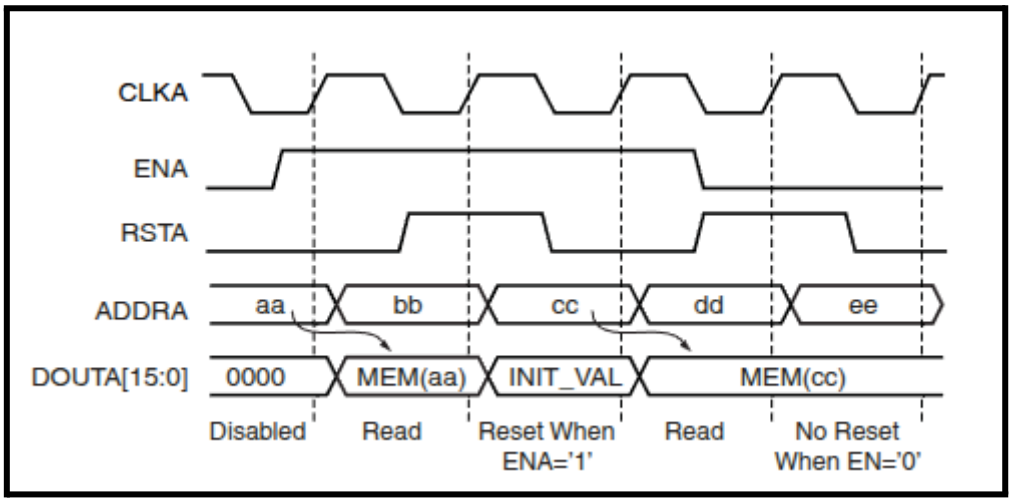


Figure 5 : RAM Timing Diagram

For further information please read this document.

Stack from RAM

What is Stack?

A **stack** is a type of data structure that follows the **Last In, First Out (LIFO)** principle. This means that the last element pushed onto the stack is the first one to be popped off. You can think of it like a stack of plates: the plate placed on top is the first one you'll remove.

There are two primary operations that can be performed on a stack:

1. **Push**: Inserts a new element onto the top of the stack.
2. **Pop**: Removes the top element from the stack."

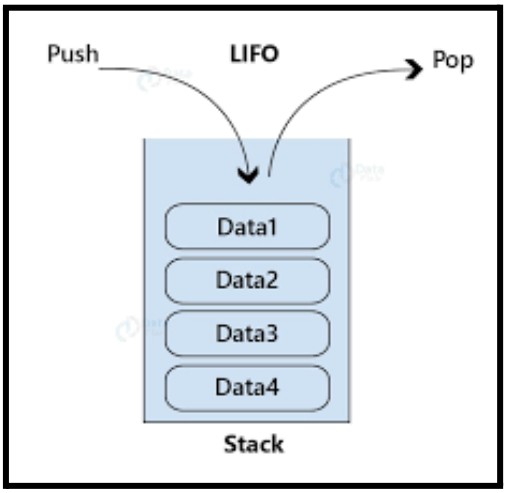


Figure 6 : Stack

Implementation Idea

We borrow an idea from the DataStructure class that implements a stack using an array (since RAM and an array act similarly). To create a stack from RAM, we'll implement a **pointer** that keeps track of the top element in the stack.

Push Operation:

- When a **push** operation occurs, we increment the pointer and store the new data in RAM at the address pointed to by the pointer.

Pop Operation:

- For a **pop** operation, we simply decrement the pointer to remove the top element and adjust the stack's state accordingly.

Reset Operation:

- For a **Reset** operation, we can simply set the pointer to the starting point in our ROM.

Challenges:

The more challenging aspect is controlling the **RAM input signal**. You need to carefully consider the **RAM operation timing diagram** to ensure the operations are synchronized with the clock. Additionally, since the RAM size is fixed, you'll need to manage situations where:

- You try to push more data than the RAM can hold (stack overflow).
- You attempt to pop from an empty stack (stack underflow).