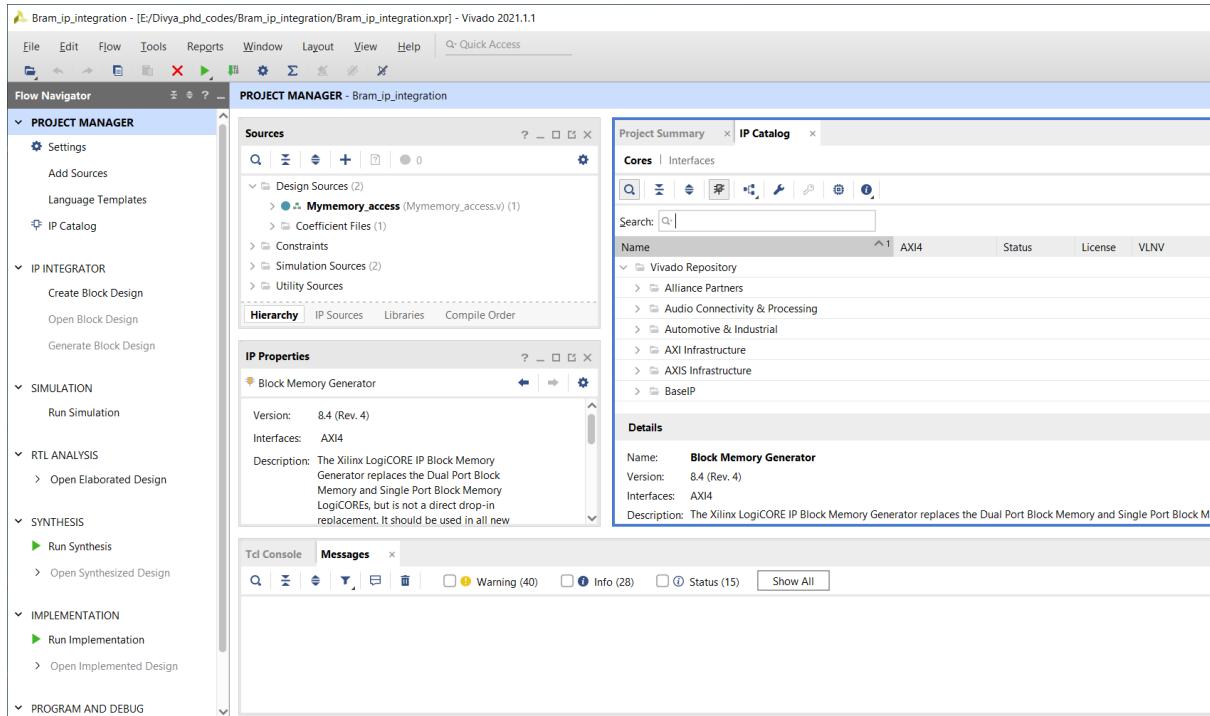


BRAM IP Integration

In Vivado , BRAM IP can be added to your design by following steps:

1.After creating a project, click on **IP Catalog** from the Flow Navigator pane.



2.From IP Catalog select Block Memory Generator IP as shown below:

The screenshot shows the Xilinx IP Catalog interface. The search bar at the top contains the text "blo". Below the search bar is a table with columns: Name, Status, License, and VLVN. The table lists several IP cores under categories like Memory Elements, Digital Signal Processing, and Building Blocks. One core, "Block Memory Generator", is highlighted in blue. Its details are shown in the "Details" section below the table.

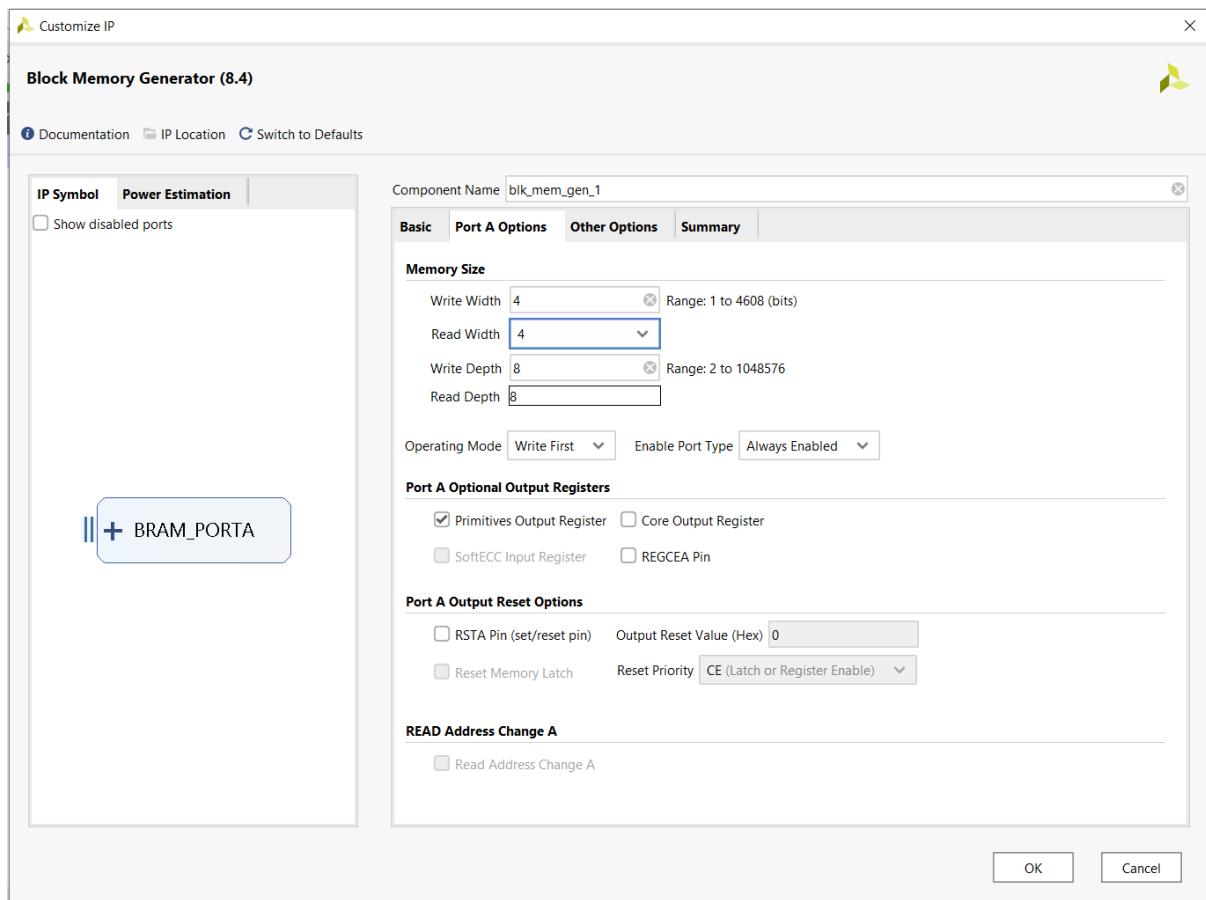
Name	Status	License	VLVN
Block Memory Generator	Production	Included	xilinx.com:ip:blk_mem_gen:8.4
Complex Multiplier	AXI4-Stream	Production	Included
CORDIC	AXI4-Stream	Production	Included

Details

Name: **Block Memory Generator**
Version: 8.4 (Rev. 4)
Interfaces: AXI4
Description: The Xilinx LogiCORE IP Block Memory Generator replaces the Dual Port Block Memory and Single Port Block Memory LogiCOREs, but is

3.Customize the IP according to the required design. Below , it is customized to create a Single Port Bram with width=4 and depth=8.

The screenshot shows the "Customize IP" dialog for the "Block Memory Generator (8.4)". The dialog has tabs for "IP Symbol", "Power Estimation", and "Basic". The "Basic" tab is selected. In the "Basic" tab, the component name is set to "blk_mem_gen_1". Under "Interface Type", "Native" is selected. Under "Memory Type", "Single Port RAM" is selected. Under "ECC Options", "No ECC" is selected. Under "Write Enable", "Byte Write Enable" is checked with a byte size of 9. Under "Algorithm Options", the algorithm is set to "Minimum Area" and the primitive is set to "8lx2". On the left, a tree view shows the port connections: BRAM_PORTA (addr[3:0], clka, dina[15:0], douta[15:0], ena, wea[0:0]). At the bottom right are "OK" and "Cancel" buttons.



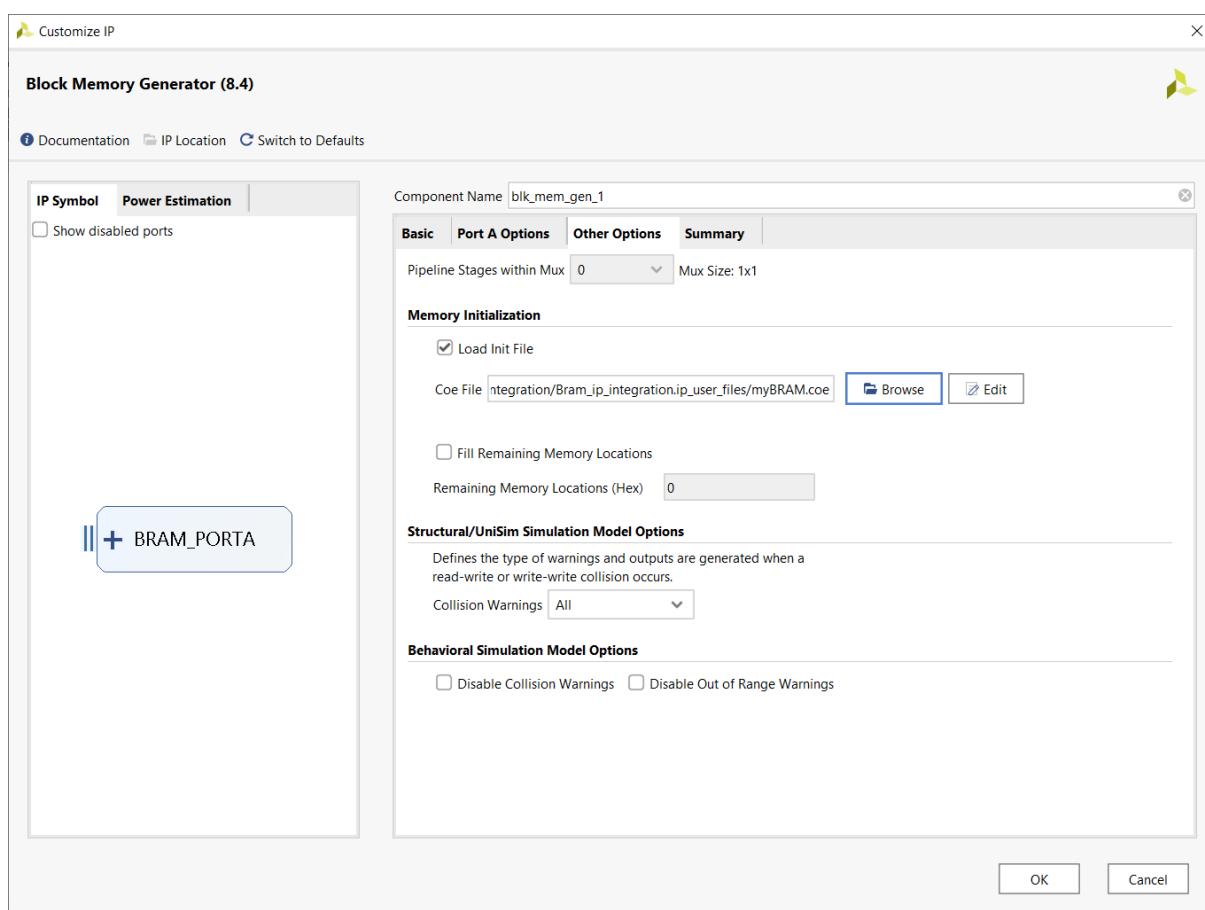
4. The initial data in the BRAM can be loaded either by a mem or a coe file. Below, it is loaded using a coe file.

myBRAM.coe - Notepad

File Edit Format View Help

```
memory_initialization_radix=10;
memory_initialization_vector=0 1 2 3 4 5 6 7 ;
```

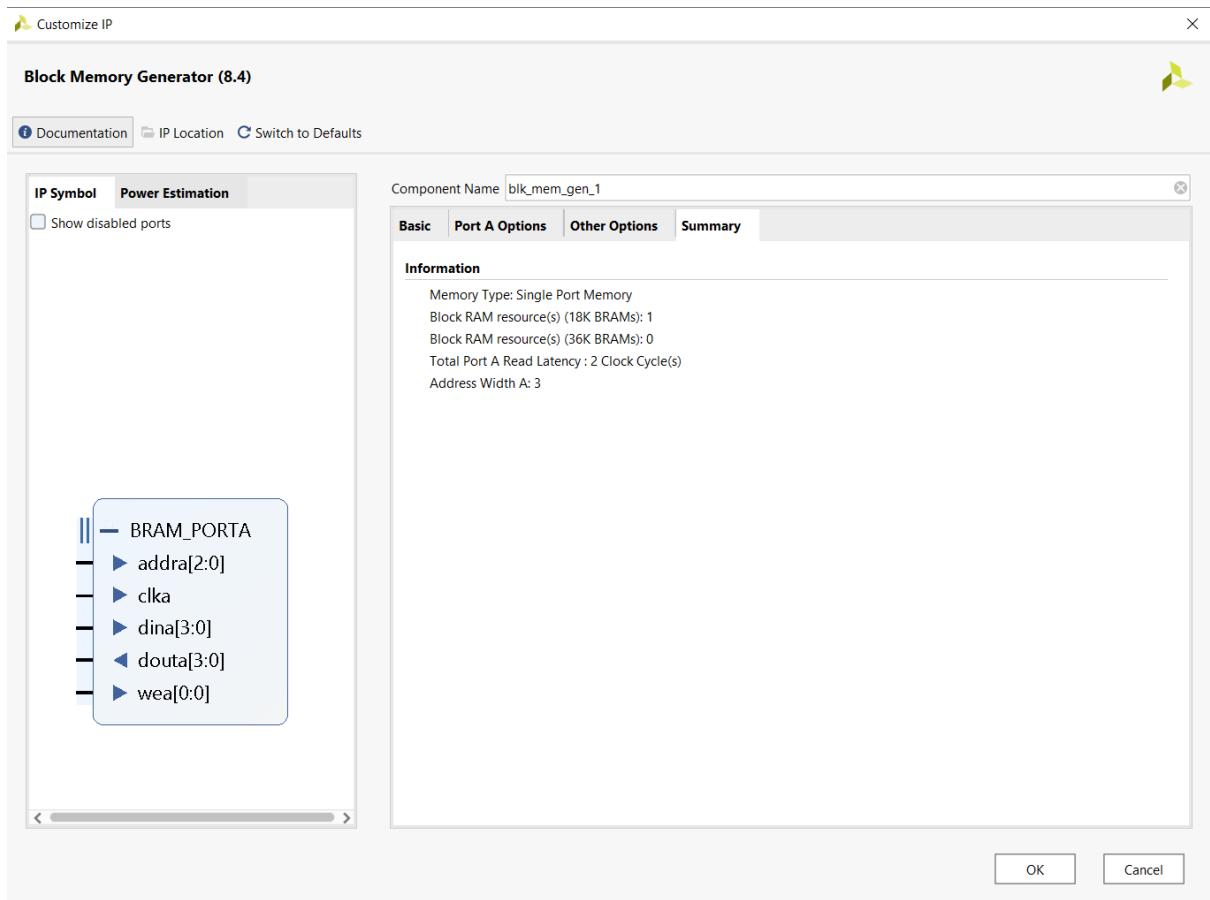
Ln 1, Col 1 100% Windows (CRLF) UTF-8



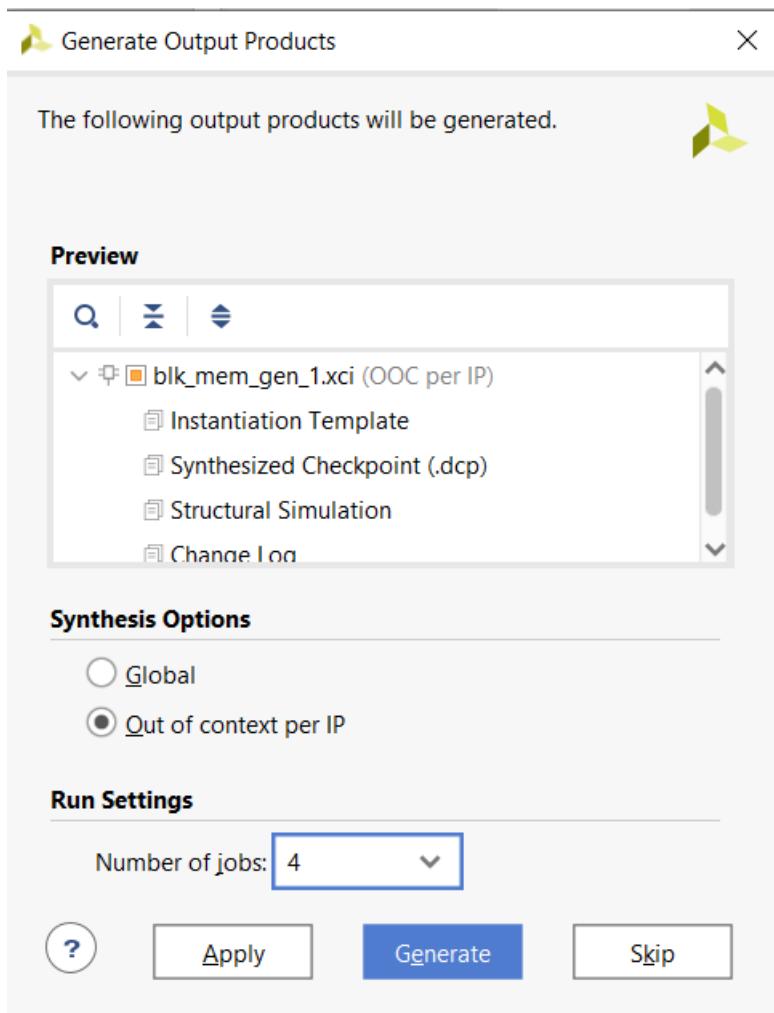
COE File Editor - myBRAM.coe

Key	Value
memory_initialization_radix	10
memory_initialization_vector	0 1 2 3 4 5 6 7

5.The summary of the customized IP can be seen as follows:



6. After clicking on OK , Generate Output Products dialogue box will appear.
Click on Generate.



7. Now the generated BRAM of desired type, width and depth can be instantiated in the main module as follows:

The screenshot shows the Xilinx Vivado Project Manager interface. On the left, the 'PROJECT MANAGER' window displays a 'Sources' tree with a 'Design Sources' section containing a 'mem_access' item. On the right, the 'mem_access.v' file is open in the 'Project Summary' editor. The code is a Verilog module named 'mem_access' that instantiates a 'blk_mem_gen_1' component. The component has several input ports: .clka, .wea, .addr, .dina, and .douta. The 'Source File Properties' panel on the left shows the file is enabled and its location is E:/Divya_phd_codes/Bram_ip_integration/Bram_ip_in. The code in the editor is as follows:

```

17 // Revision 0.01 - File Created
18 // Additional Comments:
19 //
20 //////////////////////////////////////////////////////////////////
21
22 module mem_access(
23 );
24
25 );
26
27 blk_mem_gen_1 uut (
28     .clka(clka), // input wire clka
29     .wea(1'b0), // input wire [0 : 0] wea
30     .addr(4'b010), // input wire [3 : 0] address
31     .dina(dina), // input wire [3 : 0] dina
32     .douta(douta)); // output wire [3 : 0] douta
33
34 endmodule
35

```

Q1. You are required to design a Verilog module which has three Block RAMs:

- **RAM_A** → 16-bit wide, depth 128 (addresses 0–127).
- **RAM_B** → 16-bit wide, depth 256 (addresses 0–255).
- **RAM_F** → 1-bit wide, depth 256 (addresses 0–255).

The design must perform the following tasks:

Part A: Reading Data and Control

1. Read data sequentially from **RAM_A[0...127]**.
2. Read data in reverse order from **RAM_B[127...0]**.
3. For each index i , read the **mode bit** from **RAM_F[i]** (addresses 0–127).

If mode = 0:

$$\text{Result}[i] = \text{RAM_A}[i] + \text{RAM_B}[127-i]$$

If mode = 1:

$$\text{Result}[i] = \text{RAM_A}[i] - \text{RAM_B}[127-i]$$

Part B: Writing Results

1. Store the **computed result** into **RAM_B[128+i]** for each i (addresses 128–255).
2. Detect **overflow or underflow**:
 - For addition, overflow occurs if the result exceeds 16 bits (carry out of MSB).
 - For subtraction, underflow occurs if the result is negative (borrow).
3. Write the detected **overflow/underflow flag (1-bit)** into **RAM_F[128+i]** (addresses 128–255).

Part C: Processing Sequence

- Repeat the operation for all 128 values.
- At the end of the process:
 - **RAM_B [128...255]** contains results.
 - **RAM_F [128...255]** contains corresponding overflow/underflow flags.
- Write a testbench to show the outcomes.