Block RAM Manual

서울대학교 차세대반도체 혁신공유대학

Xuan-Truong Nguyen (응웬트렁)

truongnx@snu.ac.kr

In this manual, we are going to make (1). Single-port RAM and (2). Dual-port RAM.

Let's start with the Single-port RAM which stores 6240 words, each with16 bits.

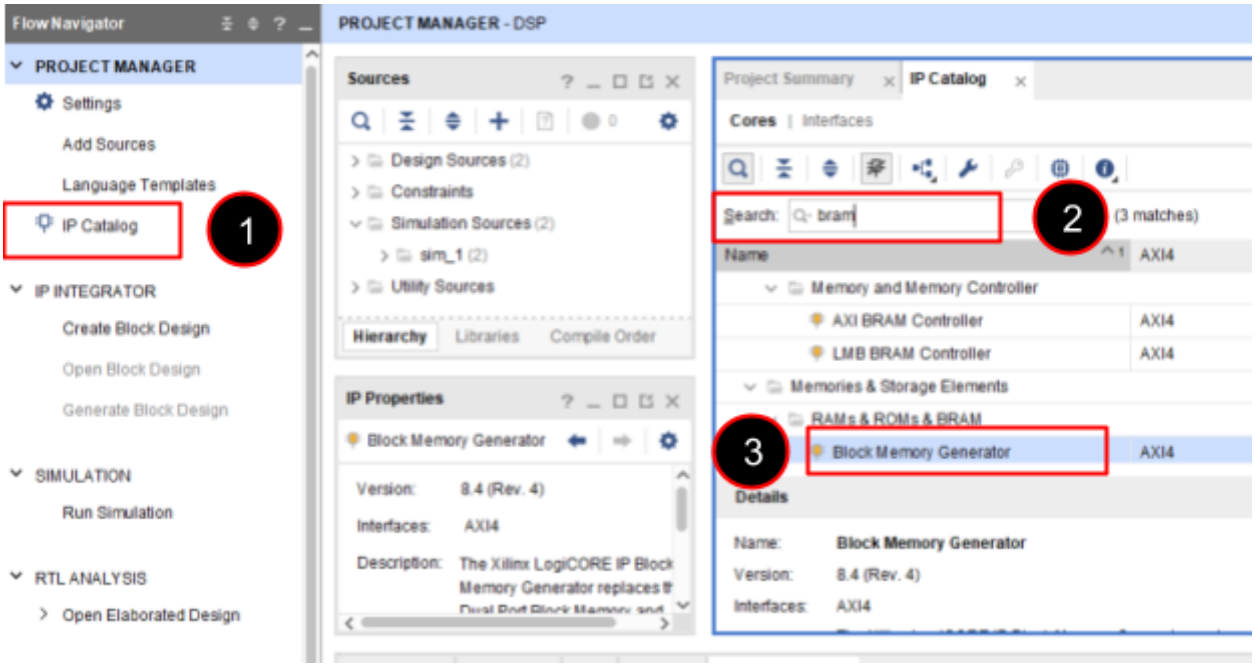
< Overall Process >

Generate a dedicated RAM using Vivado IP generator □ Initialize a RAM using a COE file □ Instantiate the RAM by its wrapper file □ Create test bench

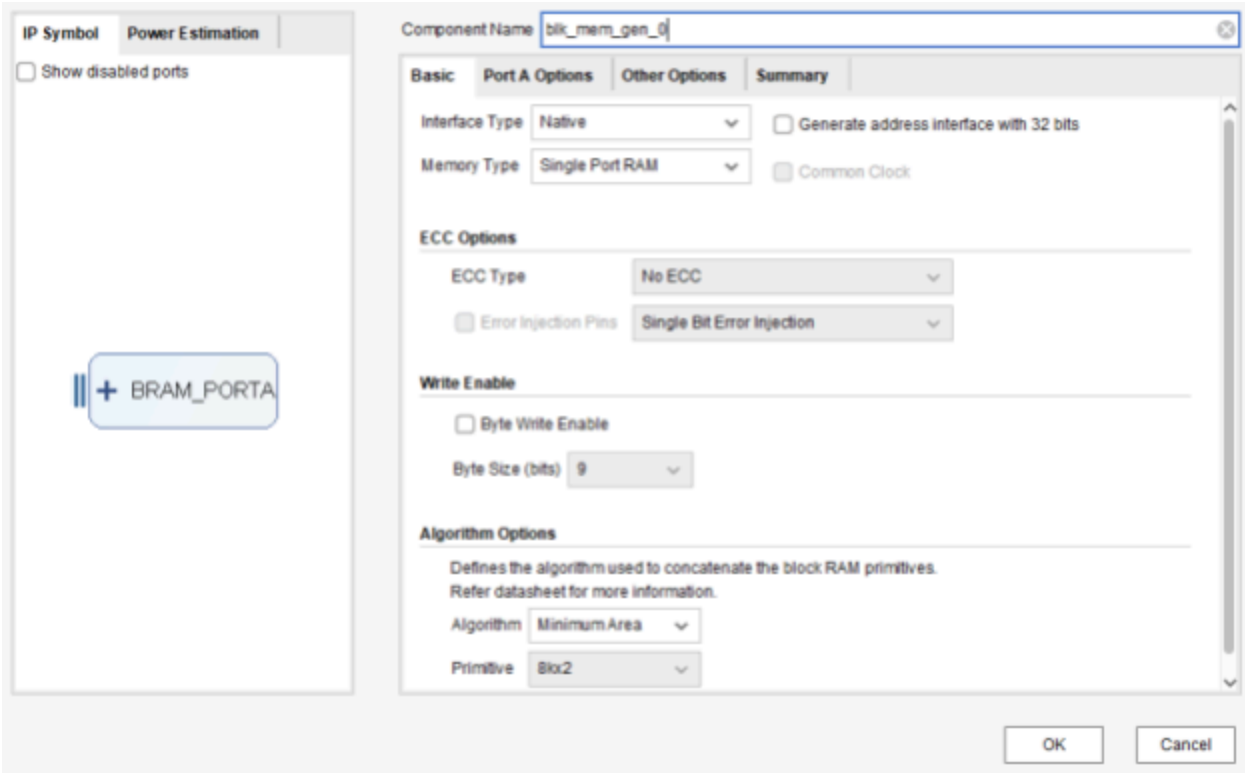
1. Block Memory Generator

We can use **Block Memory Generator** to build a block memory.

Select **IP Catalog** □ Type "bram" on **Search** □ Find and double-click on **Block Memory Generator**



By default, the window for Block Memory Generator is opened as follows. We can set configurations for **"Component Name", "Basic", "Port A (B) Options", "Other Options", and Summary.**

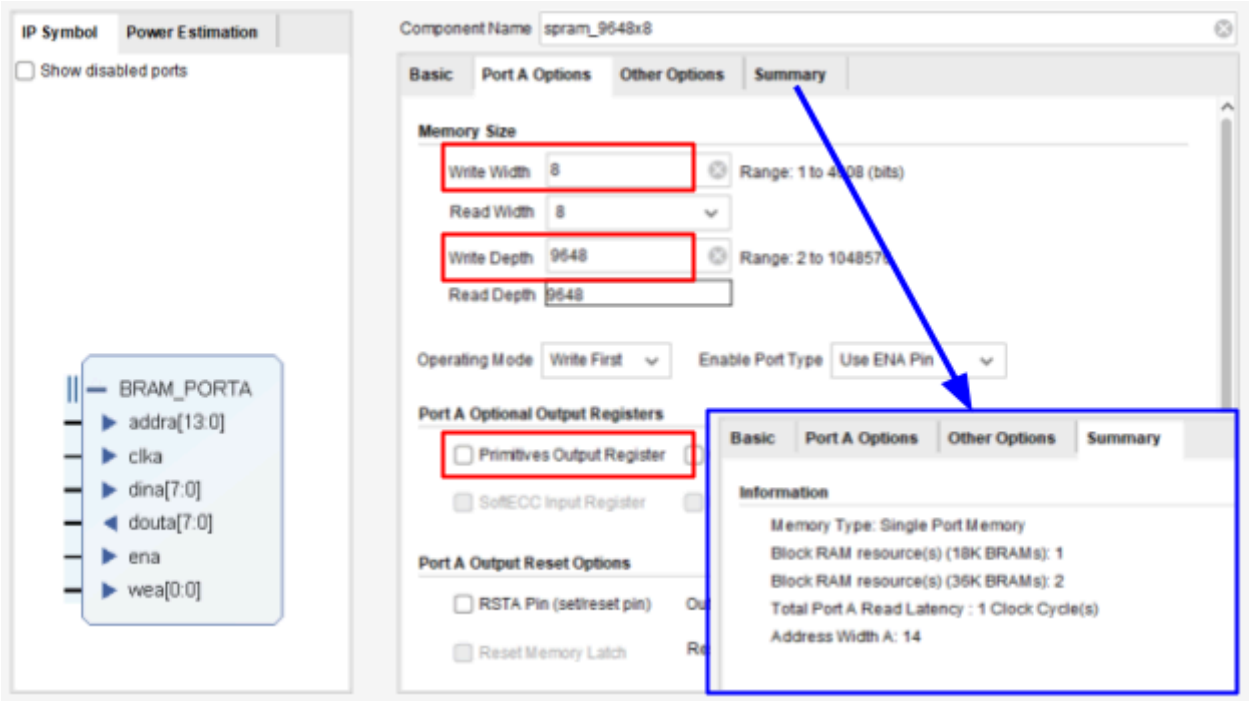


[spram_9648x8]

- We change **Component Name** to “spram_9648x8”(the same name that you declare on your code) and use default settings in Basic to use “**Single Port RAM**”.
- Port A Options are configured as follows:
  - Write/Read Width: 8
  - Write/Read Depth: 9648
  - **Uncheck** ‘**Primitives Output Register’** → A **read** request takes **<u>one cycle</u>**.

    *<u>**NOTE: If you check “**Primitives Output Register**”, **a read request takes two cycles**.</u>*

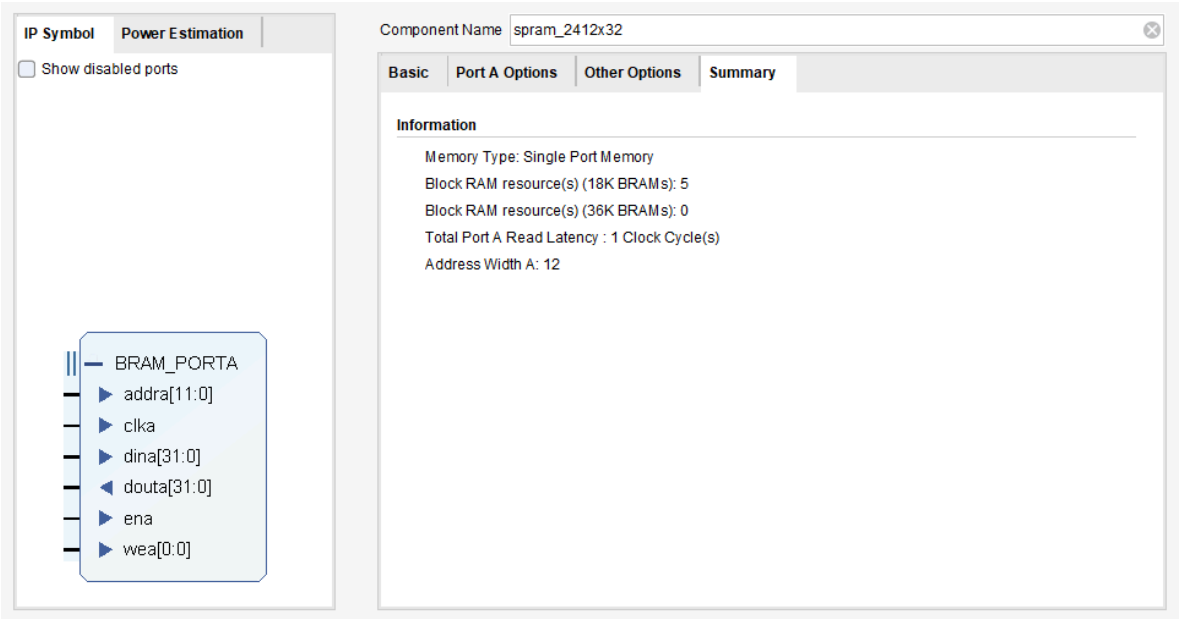
⇒ Now you can see the summarization of the newly configured block RAM: *Single Port Memory, Block Rams resource(s) (one 18K BRAMs, twro 36K BRAMs), one clock cycle for Read Latency, and 14-bit address width.*



Let's build some more.

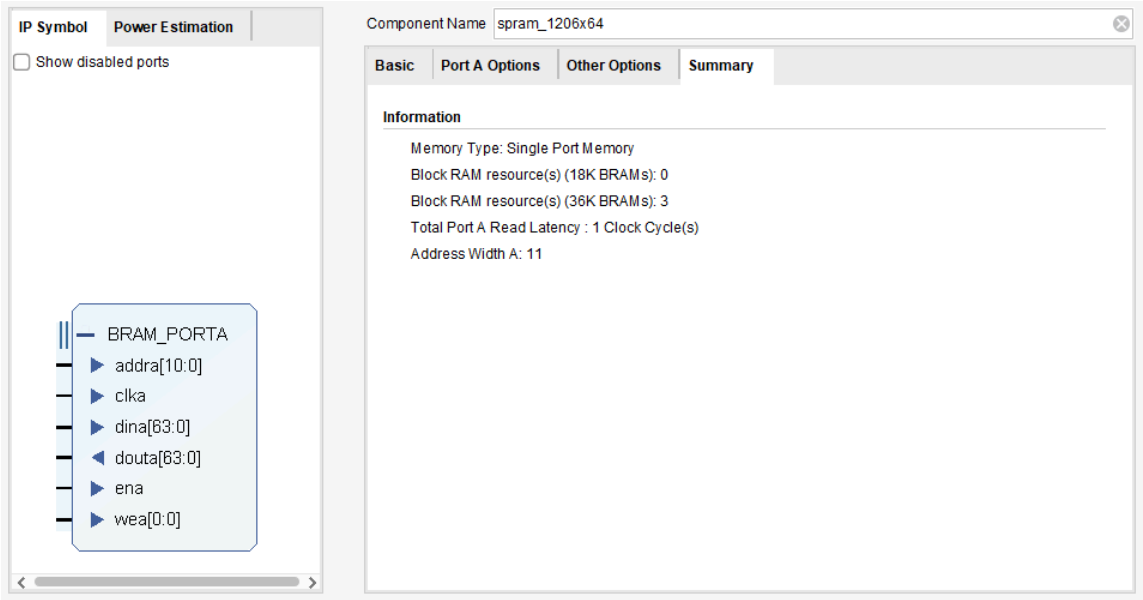
[spram_2412x32]

- **Block RAM resources (18K BRAMs): 5**
- **Block RAM resources (36K BRAMs): 0**



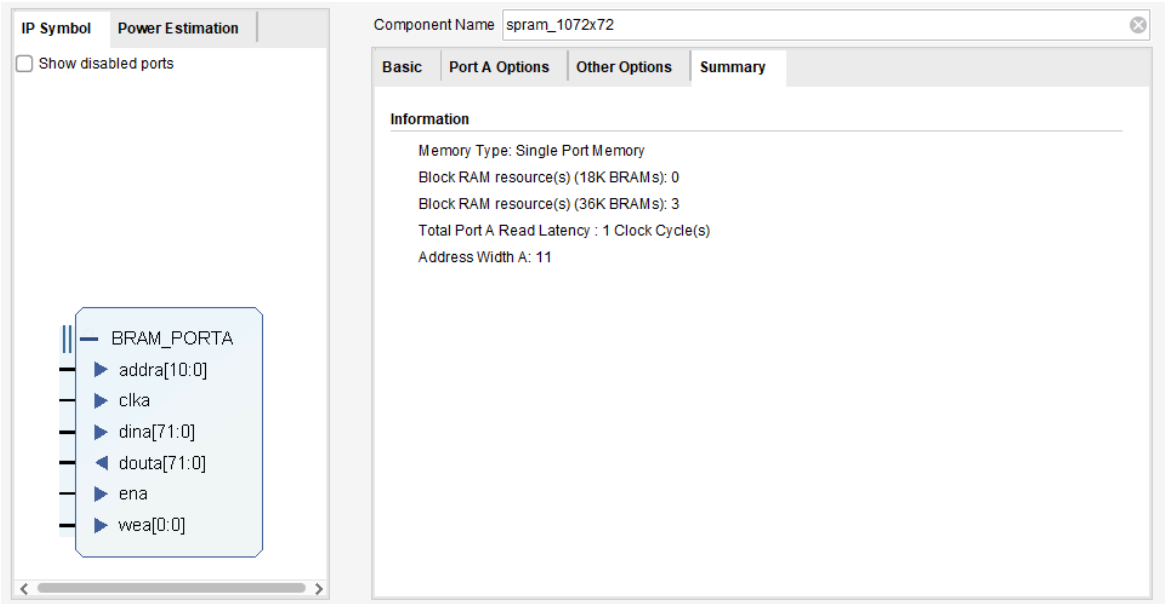
[spram_1206x64]

- **Block RAM resources (18K BRAMs): 0**
- **Block RAM resources (36K BRAMs): 3**



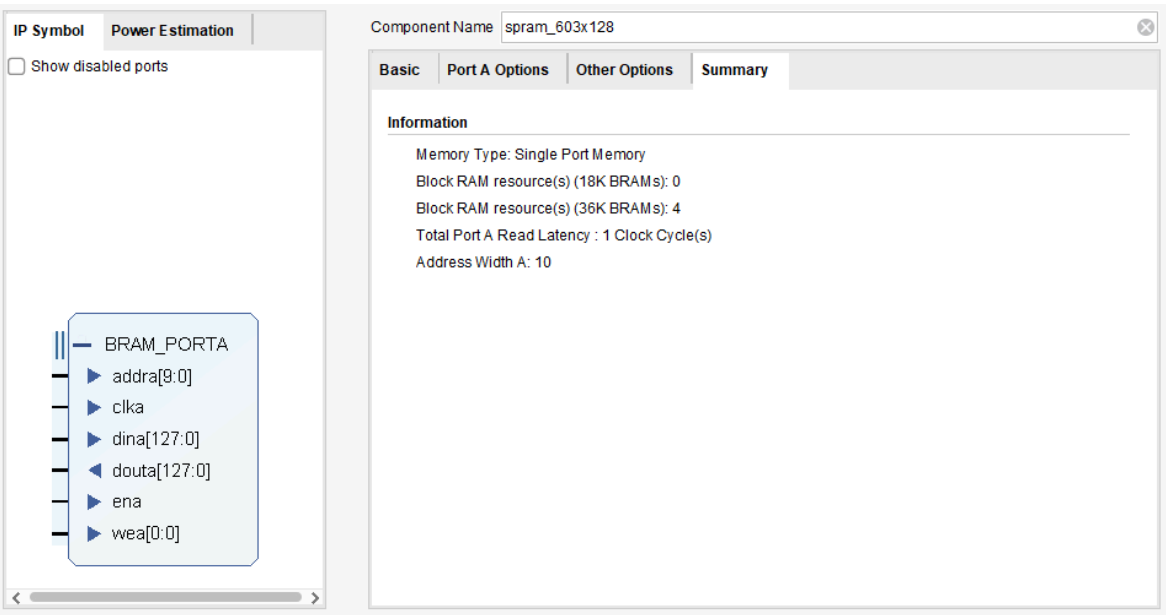
[spram_1072x72]

- **Block RAM resources (18K BRAMs): 0**
- **Block RAM resources (36K BRAMs): 3**



[spram_603x128]

- **Block RAM resources (18K BRAMs): 0**
- **Block RAM resources (36K BRAMs): 4**

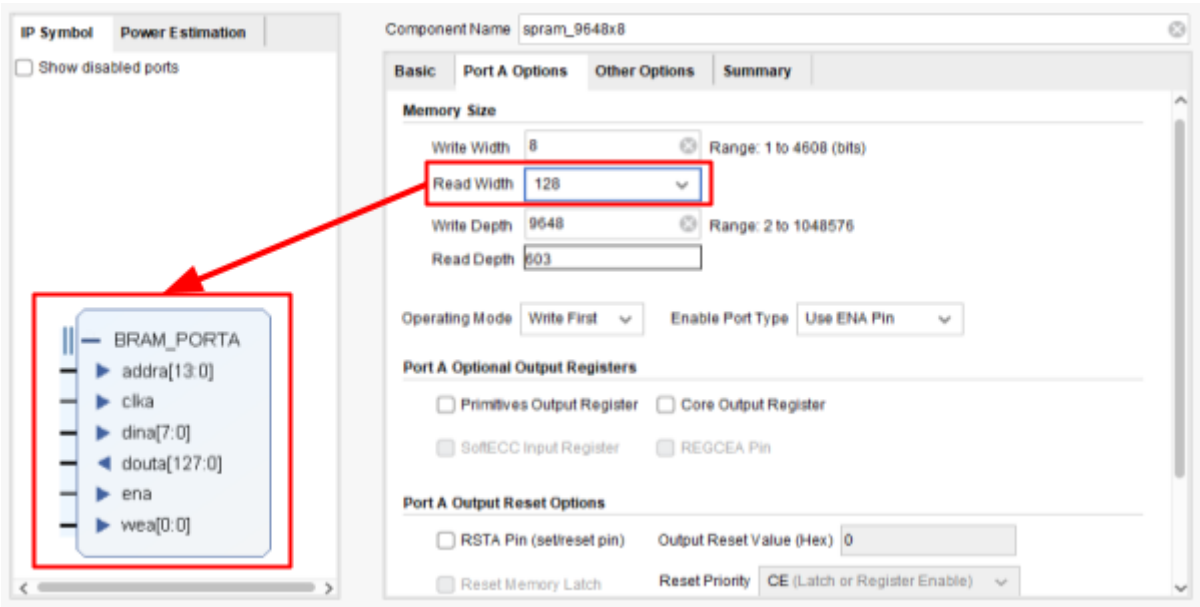


2. Advance Options

(a) Different bit-widths for “read” and “write”

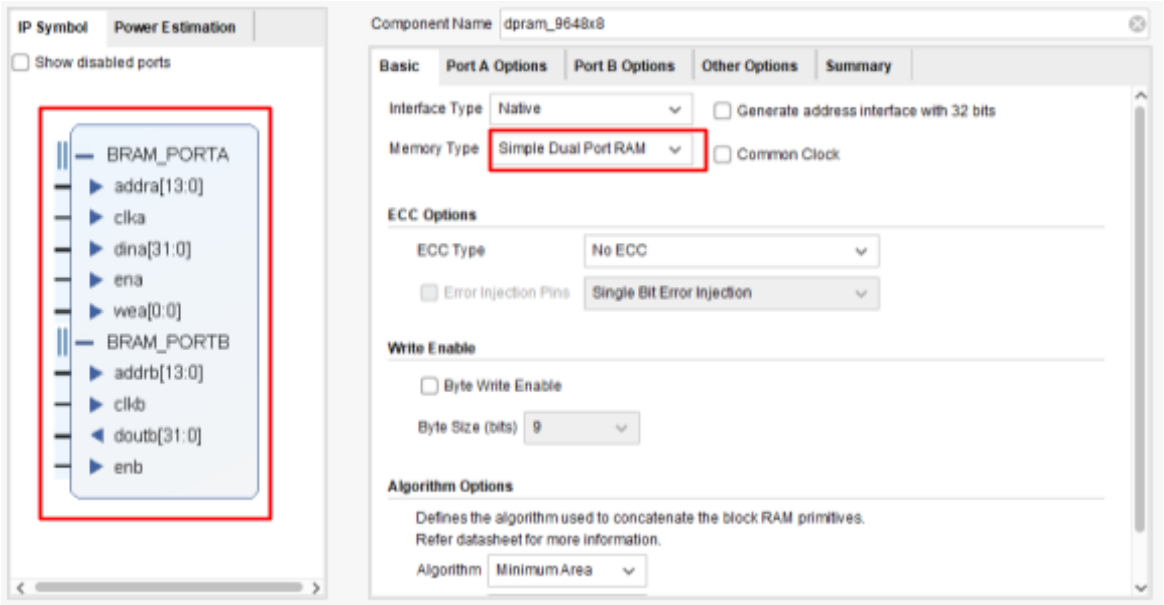
- A motivation for this functionality is that the number of **Read requests is significantly larger** than the number of Write requests. For example, we only **write** or update the filter buffer **once**. Meanwhile, we may **read** the buffer every time we calculate an output pixel.

- We can configure a block memory that has different Write/Read Width. For example, for Write Width = 8, Read Width options are 1, 2, 4, 8, 16, 32, 64, 128. Therefore, **we can increase the bandwidth for Read.** In particular, it only takes **one cycle** to load 128 bits or 16 eight-bit pixels when Read Width = 128.
- This functionality allows us to **increase Read bandwidth without increasing the number of memory instances**. However, if access patterns are irregular or random, we must construct multiple memory instances.



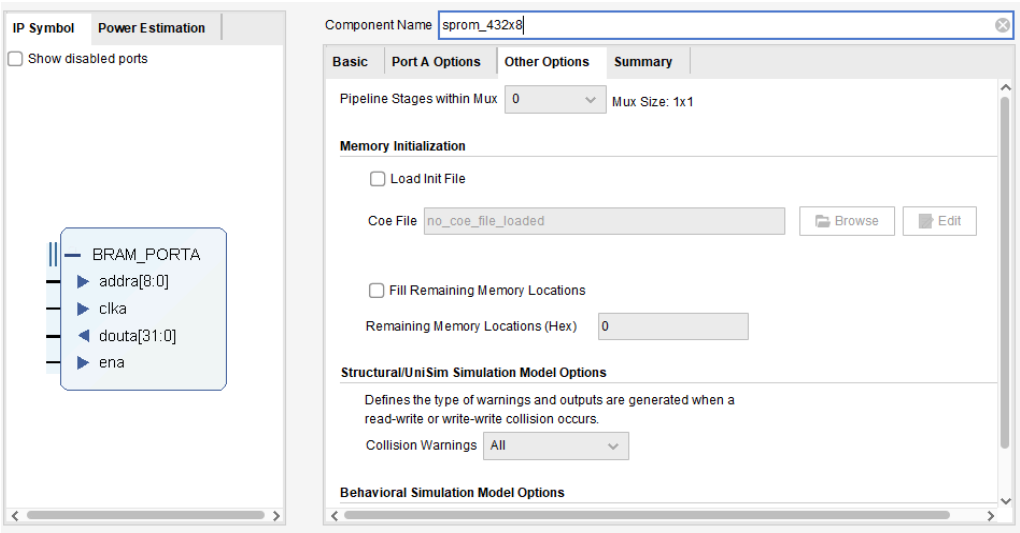
(b) Dual port block memory

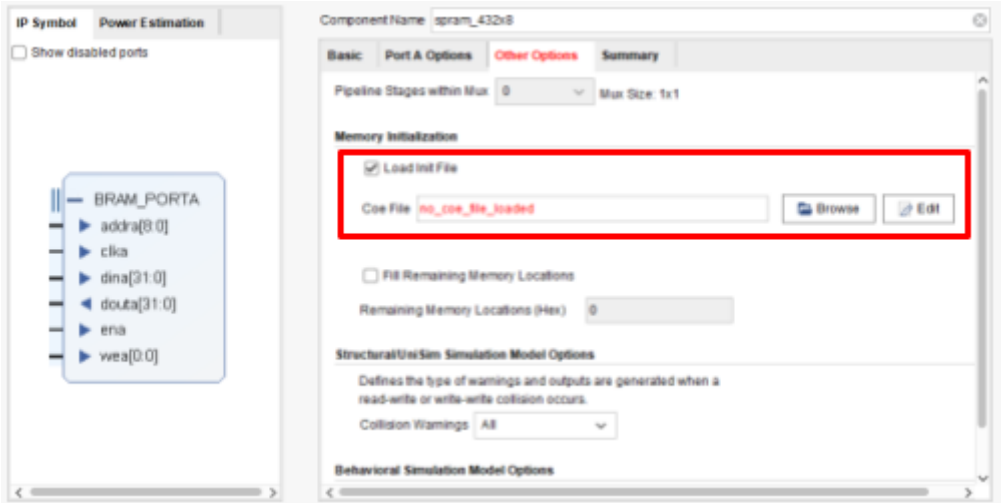
- To increase the bandwidth, we may configure a block memory as “Simple Dual Port RAM” or “True Dual Port RAM”.
  - **Simple Dual Port RAM**: Port A is Write Only, and Port B is Read Only.
  - **True Dual Port RAM**: We can use Read and Write for Ports A and B. In other words, it can work like two independent RAMs. As a result, its resource is two times larger than that of a single-port RAM with the same configuration.
- A dual-port block memory can support **Read and Write requests simultaneously.** For example, a block memory is used to store both input feature maps (IFMs) and output feature maps (OFMs). When computing units request some IFMs from the memory to calculate OFMs, the controller issues Read requests to the block memory. At the same time, the newly computed OFMs can be written to the block memory. To this end, a dual-port block memory can increase the bandwidth.



3. Initialize with COE

A block memory can be configured as ROM or RAM **with an initialized file**. For example, CONV1 has only 432 eight-bit coefficients that are predefined. Then, we can configure it as a single-port ROM or a single-port RAM. Note that we can not update a ROM as it does not have a Write port. (Download and use "./lab1_spram/CONV20_W.coe")





The init file is a COE file as follows.

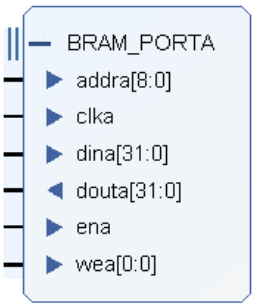
- **Radix**: the radix of the initialized file. Some widely-used radix options are Binary, Decimal, and Hexadecimal (16).
- **Vector**: The initialized vector. For example, if we want to initialize the filter buffer with CONV1's filters, we will make 432 eight-bit lines.
- **Configuration**: To load an init file into a block memory, we must select "Load Init File", and click **Browse** to access a predefined COE file. If we want to change the init file, we can use **Edit.**

CONV20_W.coe

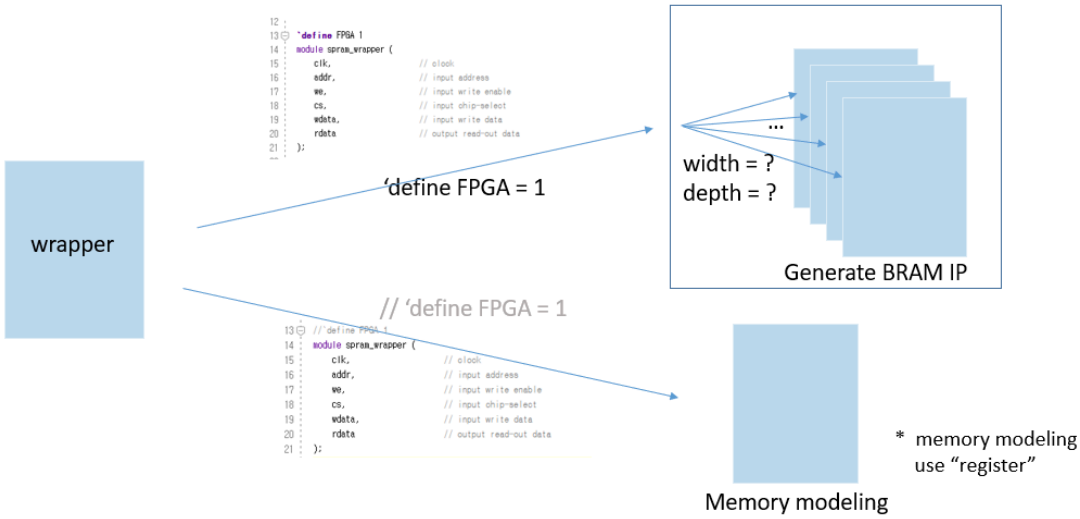
```
memory_initialization_radix=16;
memory_initialization_vector=
fe00fe01    ,
fd01ffff    ,
00000303    ,
00fdfe02    ,
0005ff00    ,
00fefd00    ,
0000fe00    ,
01000000    ,
fdff00fb    ,
01fefffb    ,
ffff0201    ,
00fa0001    ,
010400fd    ,
0500ff00    ,
```

4. Access Block Memory

- To access a block memory, we must generate Read or Write requests. The signals for a block memory (Port A) are as follows:
  - **clka**: Clock. Note that a block memory is synchronous. Therefore, it uses the same clock for port A and port B.
  - **ena**: Enable signal for both Read and Write requests. When there are multiple instances of block memory, ena can be connected to a chip-select (cs) signal.
  - **wea**: Write enable signal. wea = 1 indicates a Write request, while wea=0 refers to a Read request.
  - **addra**: Address for a Read or Write request. The width of addra indicates the depth of a block memory. For example, if addra has 9 bits, the memory has a maximum of 512 words.
  - **dina**: Write data. The data is stored in the memory for a Write request.
  - **douta**: Read data. The data returns to the module that issues a Read request to the memory. It may takes one or two cycles for a Read request.



Let's add a wrapper file**(spram_wrapper.v)** to access the generated block memories. You can access multiple tiles of memory(ex. 128x16 for weights 16x16 for scales/biases) and by using their wrapper. You can choose between two modes of wrapper, one of **FPGA mode(make BRAM IP)**and the other **of simulation mode(use the memory modeling=registers).**



<FPGA mode>

For FPGA mode, you need to define the **DW**(data bitwidth per word) and **AW(**address bit width) and the **DEPTH(**word length).

For example, **Ex)** **DW = 4**

DEPTH = 8

AW = $\log_2$ **DEPTH**
= 3

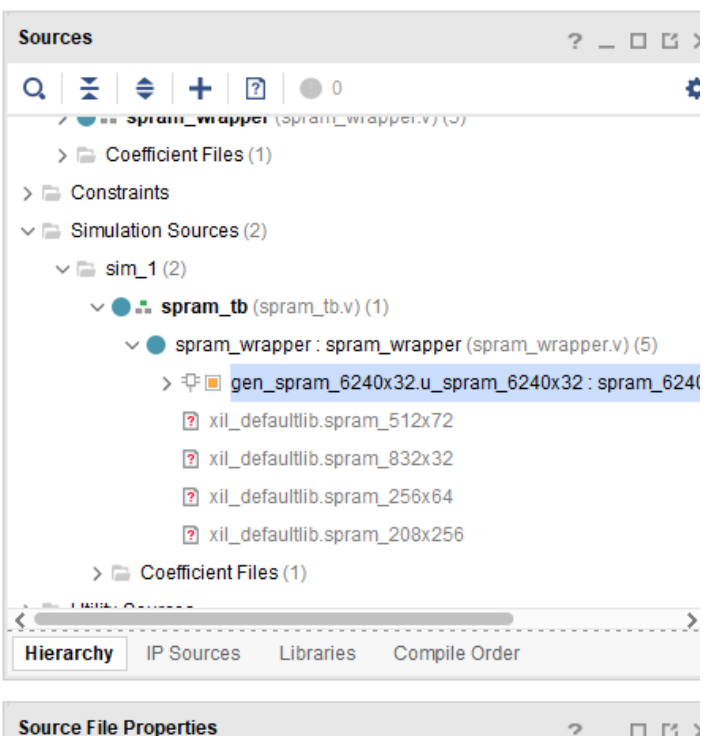
0 1 0 1
0 0 0 1
0 1 0 1
1 1 0 1
0 0 0 1
0 0 1 1
0 1 0 1
0 0 0 1

To customize your BRAM, (a) Set the parameter □ (b) Add the proper “else if~” code to generate the actual model □ (c) Make BRAM IP

```
// output parameters
parameter DW = 32;
parameter AW = 13;
parameter DEPTH = 6240;
parameter N_DELAY = 1;
```

```
else if((DEPTH == 208) && (DW == 256)) begin: gen_spram_208x256
    spram_208x256 u_spram_208x256(
        // write
        .clka(clk),
        .ena(cs),
        .wea(we),
        .addra(addr),
        .dina(wdata),
        // read-out
        .douta(rdata)
    );
end
else if((DEPTH == 6240) && (DW == 32)) begin: gen_spram_6240x32
    spram_6240x32 u_spram_6240x32(
        // write
        .clka(clk),
        .ena(cs),
        .wea(we),
        .addra(addr),
        .dina(wdata),
        // read-out
        .douta(rdata)
    );
end
```

(a) **Set the paramer** in the code (b) code to generate an actual model



(c) Make a BRAM IP and you can find the IP Source inside the wrapper

<Memory modeling mode>

When you are using memory modeling, **just set the parameter** and you can get them customized. Actually, this one also operates like a BRAM IP but it's actually made up of registers.

```
//-----------------------------------------------------------------
// Memory modeling
//
reg [DW-1          :   0]      mem[0:DEPTH-1]; // Memory cell
// Write
always @(posedge clk) begin
    if(cs && we)            mem[addr] <= wdata;
end
// Read
generate
   if(N_DELAY == 1) begin: gen_delay_1
      always @(posedge clk)
         if (cs && !(|we)) rdata_o <= mem[addr];

      assign rdata = rdata_o;
   end
   else begin: gen_delay_n
      reg [N_DELAY*DW-1:0] rdata_r;

      always @(posedge clk)
         if (cs && !(|we)) rdata_r[0*DW+:DW] <= mem[addr];

      always @(posedge clk) begin: delay
         integer i;
         for(i = 0; i < N_DELAY-1; i = i+1)
            if(cs && !(|we))
               rdata_r[(i+1)*DW+:DW] <= rdata_r[i*DW+:DW];
      end
      assign rdata = rdata_r[(N_DELAY-1)*DW+:DW];
   end
endgenerate
```

Make "mem" register (data storage)

Data write to "mem"

Data read from "mem"
: 1 cycle delay

Data read from "mem"
: more than 1 cycle delay
(not used for this example! Just ignore)

Support N cycle data delay
: 1 cycle delay for this example

Assign "mem" output port

Above is the parts of 'spram_wrapper.v,' related to memory modeling.

5. Test bench

Now that the design is completed, lets' move on to the testbench. Download and add **"spram_tb.v"**. The testbench code should test the following operations:

```
// test data set: width =  DW , depth = 16
test_data[0] = 32'h00000000;
test_data[1] = 32'h11111111;
test_data[2] = 32'h22222222;
test_data[3] = 32'h33333333;
test_data[4] = 32'h44444444;
test_data[5] = 32'h55555555;
test_data[6] = 32'h66666666;
test_data[7] = 32'h77777777;
test_data[8] = 32'h88888888;
test_data[9] = 32'h99999999;
test_data[10] = 32'haaaaaaaa;
test_data[11] = 32'hbbbbbbbb;
test_data[12] = 32'hcccccccc;
test_data[13] = 32'hdddddddd;
test_data[14] = 32'heeeeeeee;
test_data[15] = 32'hffffffff;
```

write

```
// ---------- write operation ---------- //
for(i=0;i<16;i=i+1) begin
    #(8*CLK_HALF_CYCLE)
    cs = 1'b1;
    we = 1'b1;
    addr = i;
    wdata = test_data[i];
    #(2*CLK_HALF_CYCLE)
    cs = 1'b0;
    we = 1'b0;
end
```

read

```
// ---------- read operation ---------- //
for(i=0;i<DEPTH;i=i+1) begin
    #(8*CLK_HALF_CYCLE)
    cs = 1'b1;
    addr = i;
    #(4*CLK_HALF_CYCLE)
    cs = 1'b0;
end
```

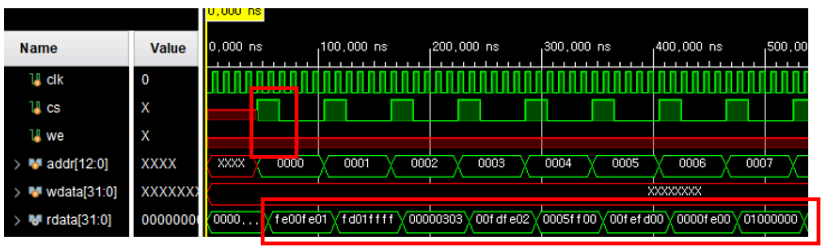
```
// ---------- read operation ---------- //
for(i=0;i<DEPTH;i=i+1) begin
    #(8*CLK_HALF_CYCLE)
    cs = 1'b1;
    addr = i;
    #(4*CLK_HALF_CYCLE)
    cs = 1'b0;
end
```

- **Initialization:** Read the initialized value(already initialized by COE file)
- **Write**: Write the test data set(width = 32, depth = 16)
- **Read**: Read the changed value

After running simulation, observe the waveforms and check if the model works well.

1) Initialization

```
// ---------- read operation ---------- //
for(i=0;i<DEPTH;i=i+1) begin
    #(8*CLK_HALF_CYCLE)
    cs = 1'b1;
    addr = i;
    #(4*CLK_HALF_CYCLE)
    cs = 1'b0;
end
```



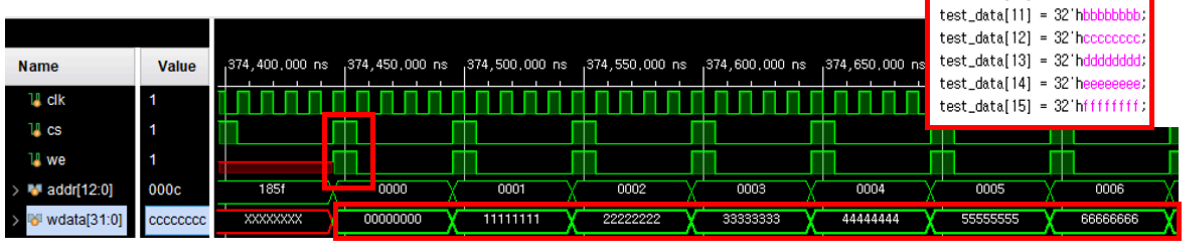
CONV20_W.coe

```
memory_initialization_radix=16;
memory_initialization_vector=
fe00fe01    ,
fd01ffff    ,
00000303    ,
00fdfe02    ,
0005ff00    ,
00fefd00    ,
0000fe00    ,
01000000    ,
fdff00fb    ,
01fefffb    ,
ffff0201    ,
00fa0001    ,
010400fd    ,
0500ff00    ,
```

2) **Write**(width = 32, depth = 16)

```
// ---------- write operation ---------- //
for(i=0;i<16;i=i+1) begin
    #(8*CLK_HALF_CYCLE)
    cs = 1'b1;
    we = 1'b1;
    addr = i;
    wdata = test_data[i];
    #(2*CLK_HALF_CYCLE)
    cs = 1'b0;
    we = 1'b0;
end
```

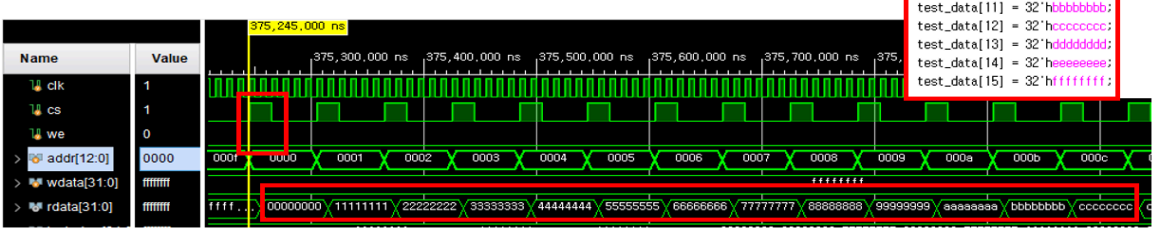
```
// test_data set: width = DW, depth = 16
test_data[0] = 32'h00000000;
test_data[1] = 32'h11111111;
test_data[2] = 32'h22222222;
test_data[3] = 32'h33333333;
test_data[4] = 32'h44444444;
test_data[5] = 32'h55555555;
test_data[6] = 32'h66666666;
test_data[7] = 32'h77777777;
test_data[8] = 32'h88888888;
test_data[9] = 32'h99999999;
test_data[10] = 32'haaaaaaaa;
test_data[11] = 32'hbbbbbbbb;
test_data[12] = 32'hcccccccc;
test_data[13] = 32'hdddddddd;
test_data[14] = 32'heeeeeeee;
test_data[15] = 32'hffffffff;
```



3) Read

```
// ---------- read operation ---------- //
for(i=0;i<DEPTH;i=i+1) begin
    #(8*CLK_HALF_CYCLE)
    cs = 1'b1;
    addr = i;
    #(4*CLK_HALF_CYCLE)
    cs = 1'b0;
end
```

```
// test_data set: width = DW, dept
test_data[0] = 32'h00000000;
test_data[1] = 32'h11111111;
test_data[2] = 32'h22222222;
test_data[3] = 32'h33333333;
test_data[4] = 32'h44444444;
test_data[5] = 32'h55555555;
test_data[6] = 32'h66666666;
test_data[7] = 32'h77777777;
test_data[8] = 32'h88888888;
test_data[9] = 32'h99999999;
test_data[10] = 32'haaaaaaaa;
test_data[11] = 32'hbbbbbbbb;
test_data[12] = 32'hcccccccc;
test_data[13] = 32'hdddddddd;
test_data[14] = 32'heeeeeeee;
test_data[15] = 32'hffffffff;
```



6. Dual-port RAM

Making dual-port RAM is similar with making the single-port one. The key difference is that now we have **port B** along with port A. Using these two ports, we can **simultaneously read and write** the data.

<Ports>

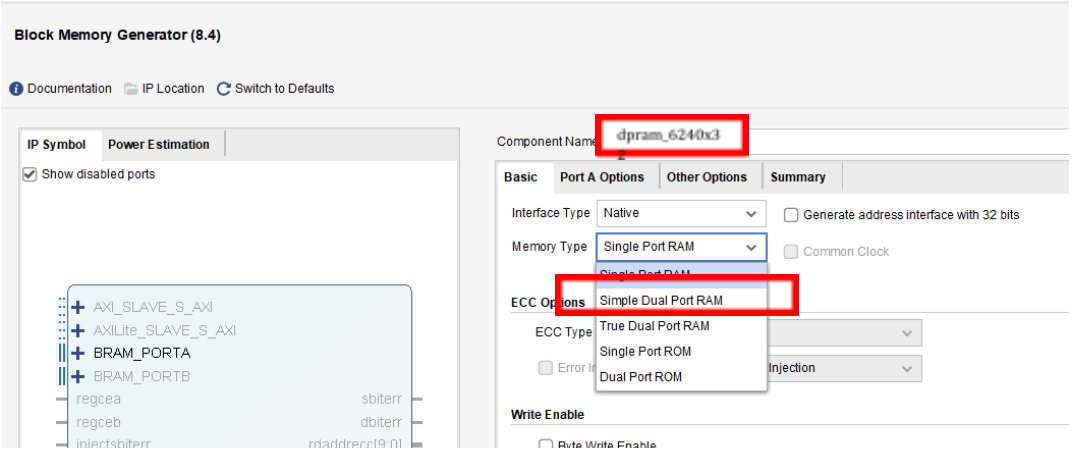
- Port A : Write only
  - Clock (clka)
  - Read/Write enable (ena)
  - Write enable (wea)
  - Address (addra)
  - Write data (dina)
- Port B : Read only
  - Clock (clkb)
  - Read/write enable (enb)
  - Address (addrb)
  - Read data (doutb)

- Read operation
  - If(enb)
    - doutb <= mem[addrb]
- Write operation
  - If(ena & wea)
    - mem[addra] <= dina

* Double-port
- Allow read/write operations at the same time

- (Just like single-port RAM) Generate a dedicated RAM using Vivado IP generator □ Initialize a RAM using a COE file □ Access RAM by wrapper file □ Create test bench

This time, select "**Simple Dual Port RAM**" instead of Single port RAM when you select the memory type.



About the wrapper and test bench, it's almost same with the spram's EXCEPT:

- dpram allow **read/write operations at the same time**!
- dpram **write** the data through **port A**!
- dpram **read** the data through **port B**!

References

[1]. Block RAM Introduction

https://docs.xilinx.com/r/en-US/am007-versal-memory/Block-RAM-Introduction

[2]. 7 Series FPGAs Data Sheet: Overview

https://datasheet.lcsc.com/lcsc/2204011730_XILINX-XC7K70T-1FBG484I_C717942.pdf