

[L17]ROM/RAM/FIFO IP测试

前面已经通过大量基础操作大致熟悉了一下ZYNQ的操作,现在说一下一些常用的IP,都是集成在片上的硬核IP.

首先MMCM/PLL这个IP,这个属于调用ZYNQ的片上资源了,具体文档是UG472,其中7010有2个CMT,一个CMT内包含一个MMCM和一个PLL,实际上MMCM功能比PLL强大,所以理解成是PLL的超集,时钟推荐从各种时钟输入引脚输入,当然也可以在本地图线引脚输入,只是效果会比较差.

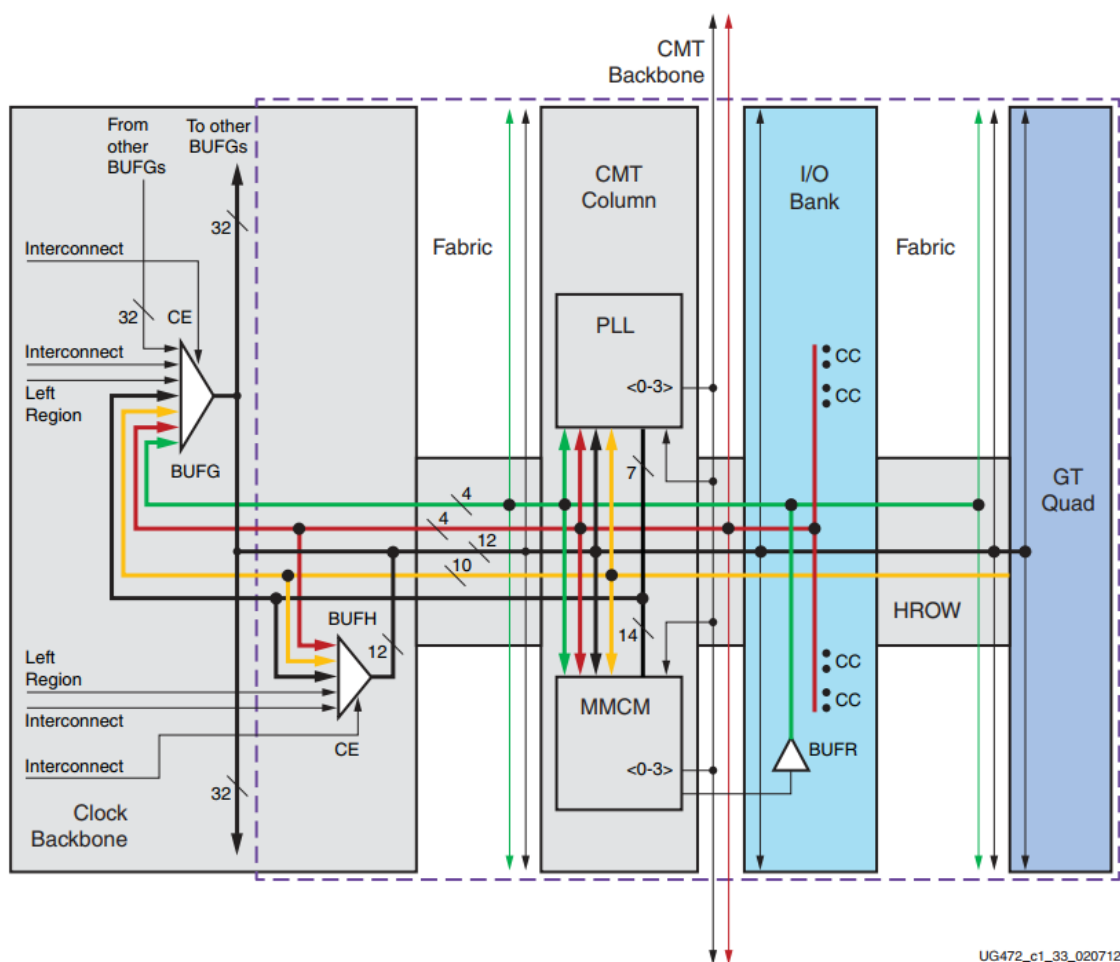


Figure 1-4: BUFG/BUFH/CMT Clock Region Detail

MMCM和PLL结构上稍微有一些差异的地方.

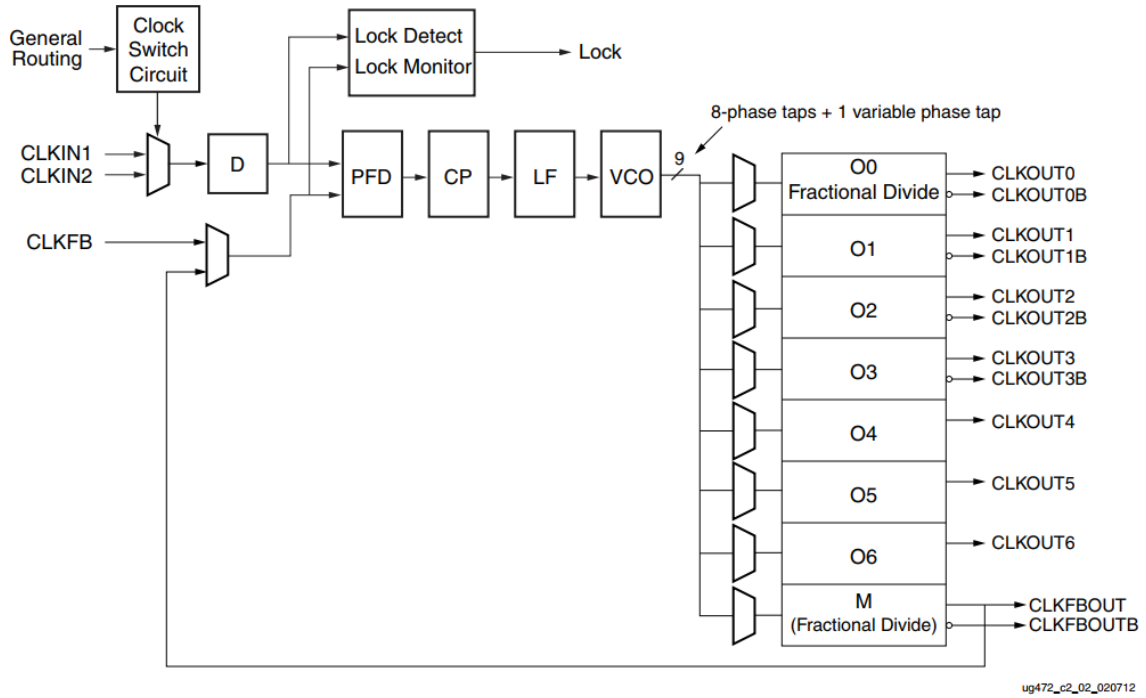


Figure 3-2: Detailed MMCM Block Diagram

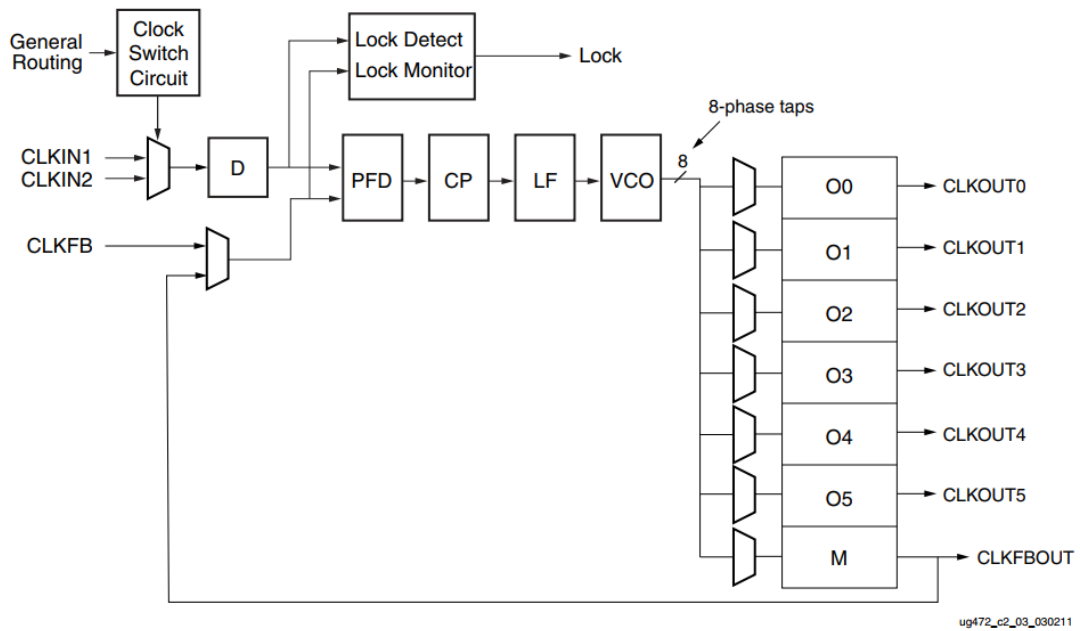


Figure 3-3: Detailed PLL Block Diagram

其中D(Division)是分频器,PFD(Phase Frequency Detector)相位检测,CP(Charge Pump)电荷泵,LF(Loop Filter)环路滤波,VCO(Voltage-controlled oscillator)压控振荡器,O*是输出分频器,其中CLKFB(Clock Feedback)插入M之后,会使得VCO频率是M倍CLKIN,而输出频率就在O*逻辑里,具体的分频结果公式.

$$F_{out} = (F_{in} * M) / (N * O)$$

另外MMCM支持扩频(Spread Spectrum)和动态相移(Dynamic Phase Shift)和额外多一个时钟输出,他们的输出区别不大,都可以驱动多种信号.

Component Name: clk_wiz_0

The phase is calculated relative to the active input clock.

Output Clock	Port Name	Output Freq (MHz)		Phase (degrees)		Duty Cycle (%)		Drives	Max Freq. of buffer
		Requested	Actual	Requested	Actual	Requested	Actual		
<input checked="" type="checkbox"/> clk_out1	clk_out1	50	50.000	0.000	0.000	50	50.0	BUFG	464.037
<input checked="" type="checkbox"/> clk_out2	clk_out2	50	50.000	180	180.000	50	50.0	BUFG	464.037
<input checked="" type="checkbox"/> clk_out3	clk_out3	100	100.000	0.000	0.000	25	25.0	BUFG	464.037
<input checked="" type="checkbox"/> clk_out4	clk_out4	200	200.000	0.000	0.000	75	78.6	BUFG	464.037
<input checked="" type="checkbox"/> clk_out5	clk_out5	200	200.000	0.000	0.000	50.000	50.0	BUFG	464.037
<input checked="" type="checkbox"/> clk_out6	clk_out6	800	700.000	0.000	0.000	50.000	50.0	No buffer	800.000

☐ USE CLOCK SEQUENCING

Clock Sequencing

Output Clock	Sequence Number
clk_out1	1
clk_out2	1
clk_out3	1
clk_out4	1
clk_out5	1
clk_out6	1

Clock Feedback

Source: ☒ Automatic Control On-Chip ☐ Automatic Control Off-Chip ☐ User-Controlled On-Chip ☐ User-Controlled Off-Chip

Signaling: ☒ Single-ended ☐ Differential

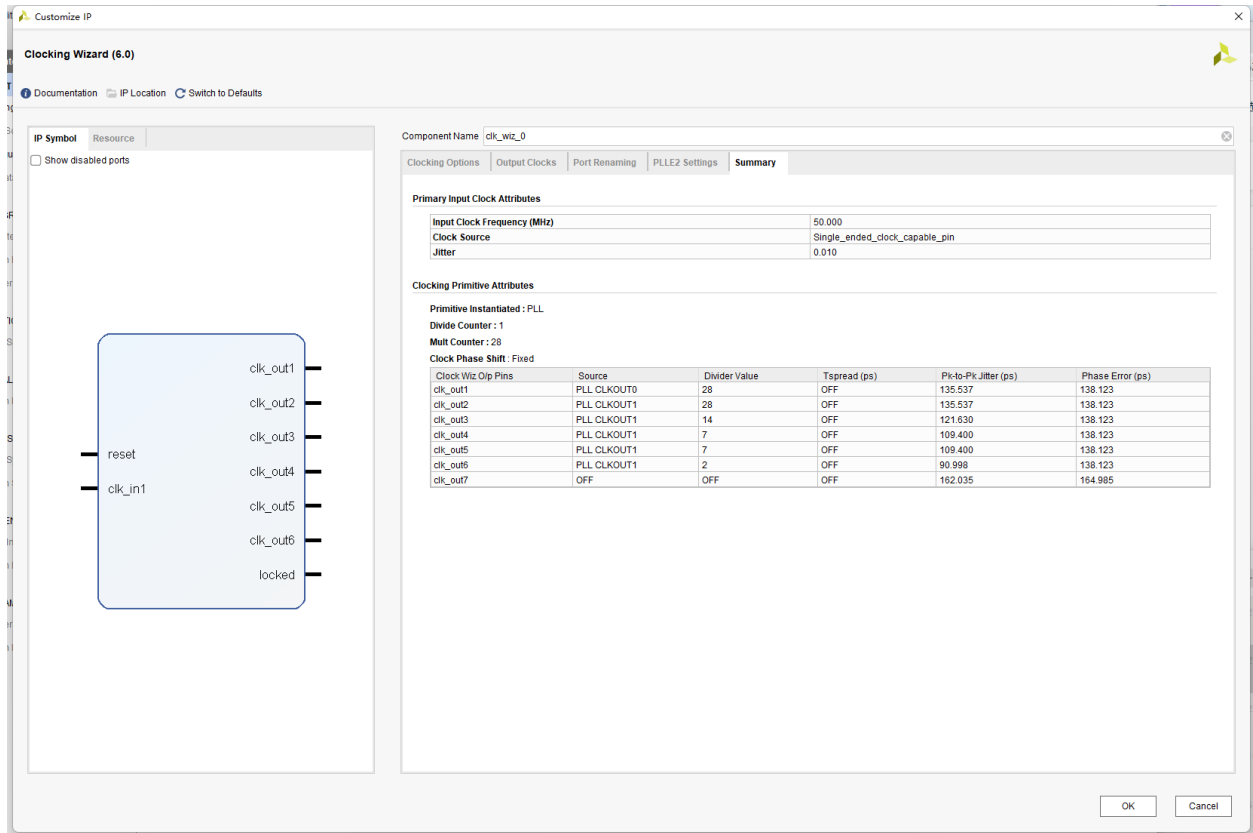
Enable Optional Inputs / Outputs for MMCM/PLL

☒ reset ☐ power_down ☒ locked

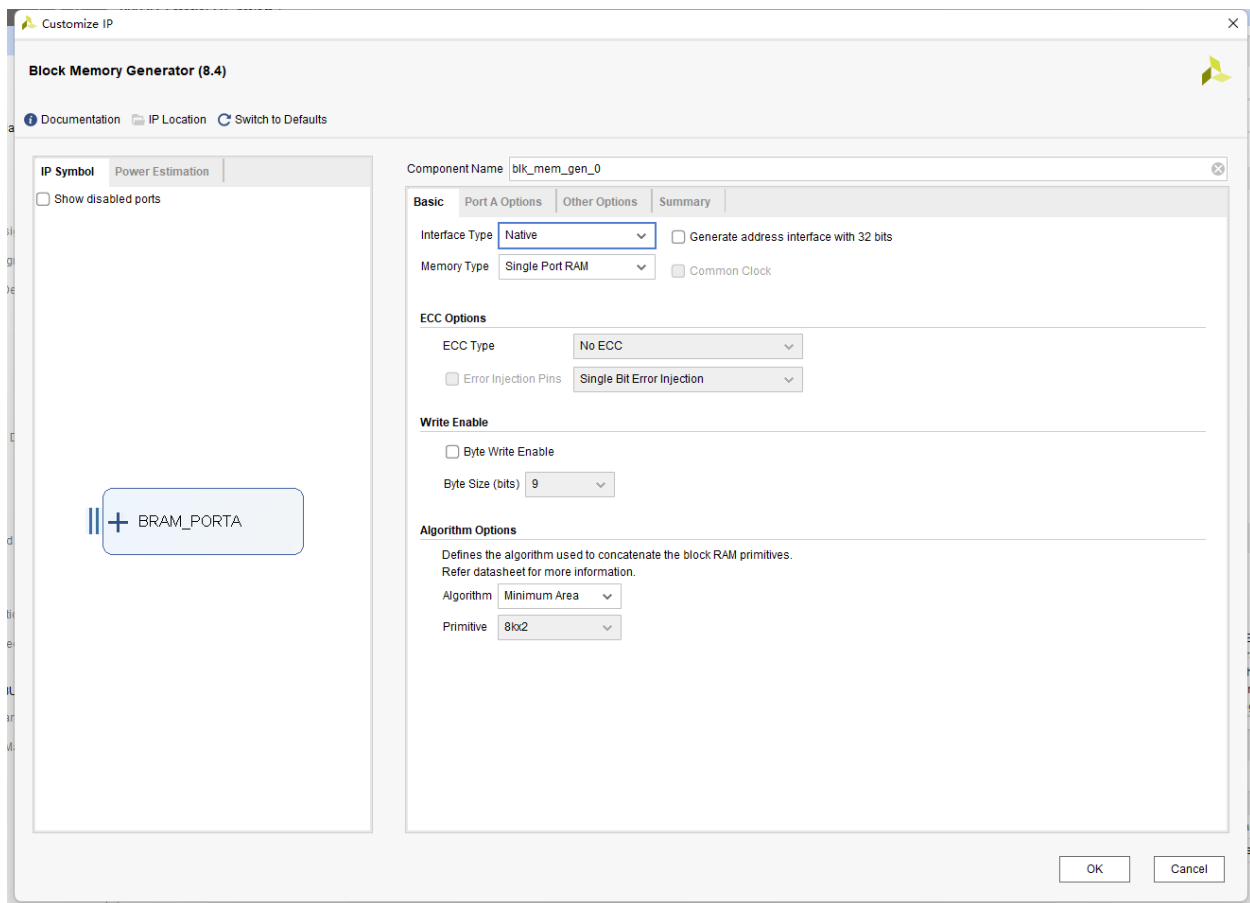
Reset Type

☒ Active High ☐ Active Low

可以得到如下输出结果,只要实例化后就能得到对应的资源.

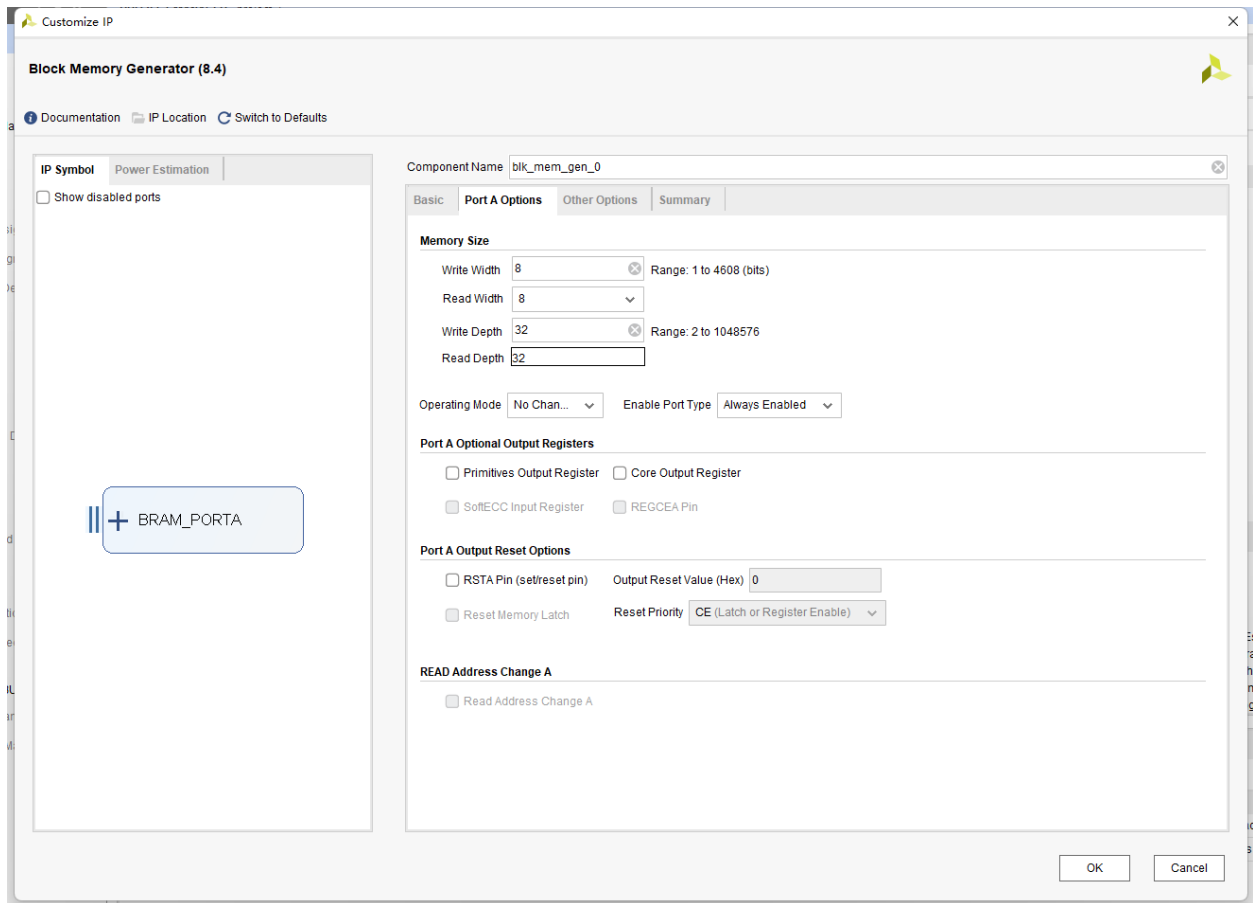


片上储存也是非常常用的IP,文档编号是UG473,这里先说片上RAM,他们和ROM,FIFO都是占用块RAM来实现的,一个块RAM就是36Kbit(比如7010内含2.1Mbit BRAM资源),可以配置成几乎任意位宽度,深度来用,自由度可以说非常高.



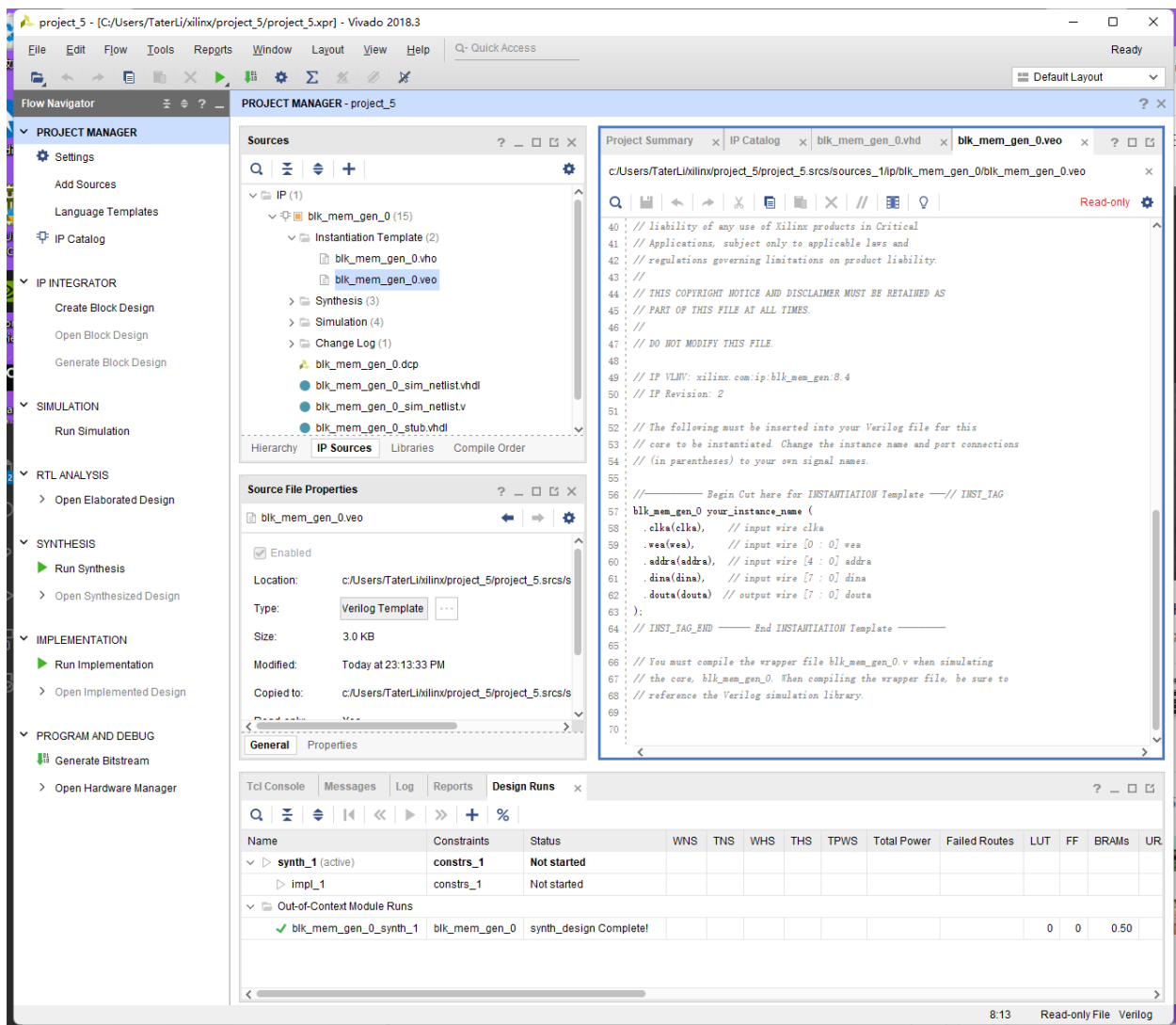
这里主要按着选项来讲一讲.

1. Interface Type:Native (简单的操作) / AXI4 (能挂在AXI总线上)
2. Memory Type:单端口RAM / 双端口RAM / 伪双端口RAM / 单端口ROM / 双端口ROM
3. ECC Type:只有伪双端口RAM能支持.
4. Write Enable:字节写选项,可以把数据的某个字节单独写入到RAM.
5. Algorithm:最小面积 / 功耗 / 固定



读写宽度可以不一样,但是深度是一样的,深度就是能装多少东西.操作模式中支持读优先,写优先,不变三种模式,不变模式不能同时读写,但是更节约资源,写优先模式会写后立即返回数据本身,读优先会写后返回上一次数据.使能端口可以选要和不要,我这里不要了,他是高电平有效的,如果需要也可以绑定在复位引脚上,其他功能可以依据文档再看,目前这么就配置出一个可用的最简RAM块.

生成后去模板复制一个示例出来用.



测试程序(配合ILA).

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 2022/03/05 23:20:09
// Design Name:
// Module Name: demo
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
```

```

//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
/////////////////////////////////////////////////////////////////

module demo(input clk,
            input rst
            );

reg [5:0] cnt;
reg [4:0] addr;
reg [7:0] wdata;
wire [7:0] rdata;

wire we;
assign we=(cnt <= 'd31)?'b1:'b0; // 0~31 写入,32~63 读取

always@(posedge clk or negedge rst)
begin
    if(!rst)
    begin
        cnt <='d0;
    end
    else if(cnt == 'd63)
    begin
        cnt <= 'd0;
    end
    else
    begin
        cnt <= cnt + 'b1;
    end
end

always@(posedge clk or negedge rst)
begin
    if(!rst)
    begin
        wdata <= 'b0;
    end
    else if(cnt < 'd31) // 计算到这里就够了,为什么?看ILA分析.
    begin
        wdata <= wdata + 'b1;
    end
    else
    begin
        wdata <= 'b0;
    end
end

always@(posedge clk or negedge rst)
begin

```



```

        if(!rst)
        begin
            addr <= 'b0;
        end
        else if(cnt == 'd31)
        begin
            addr <= 'b0;
        end
        else
        begin
            addr <= addr + 'b1;
        end
    end
end

blk_mem_gen_0 mem_inst (
    .clka(clk),
    .wea(we),
    .addra(addr),
    .dina(wdata),
    .douta(rdata)
);

ila_0 ila_inst (
    .clk(clk),
    .probe0(clk),
    .probe1(we),
    .probe2(addr),
    .probe3(wdata),
    .probe4(rdata)
);

endmodule

```



ROM也是一样时序,不过只有读没有写,那么内容哪里来呢,那就是COE文件,COE文件格式如下.

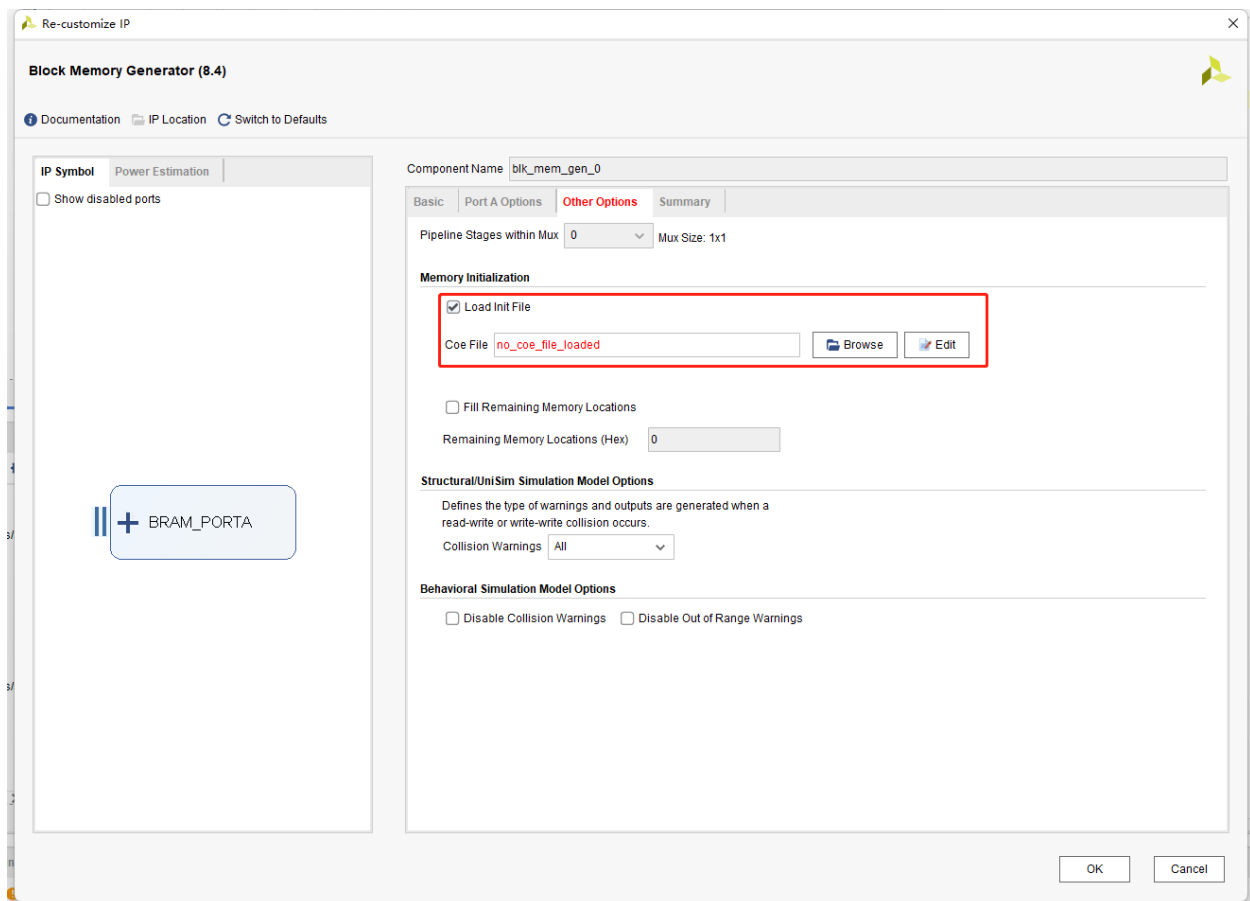
```

memory_initialization_radix=16
memory_initialization_vector=
1,
2,

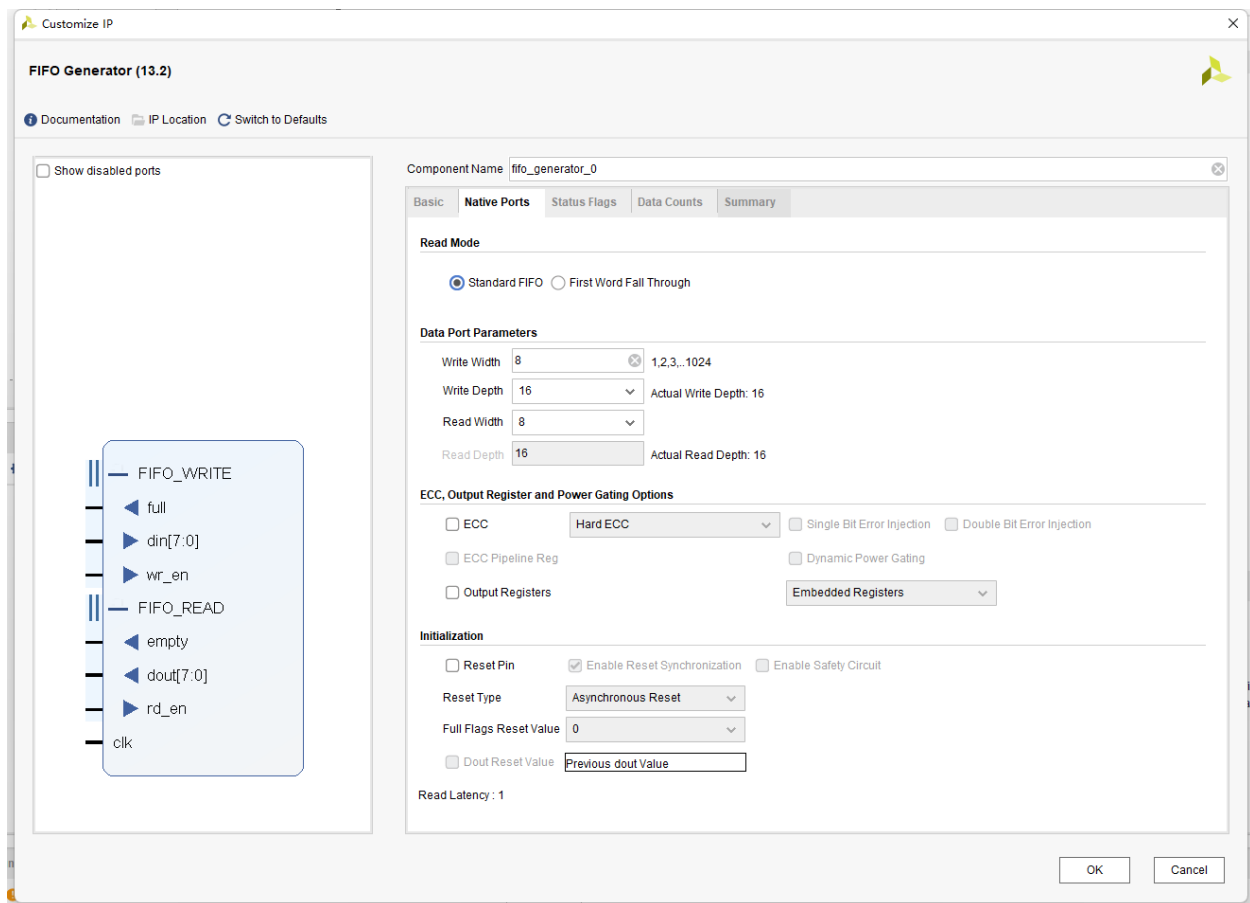
```

```
3,  
4,  
5,  
..  
1c,  
1d,  
1e,  
1f
```

通过初始化文件加载.



FIFO是另一个IP了,时序和连接上有一些不一样,主要是没有了地址,读写完全分离,取而代之是各种标记,比如默认的full和empty,当然也有almost full可以使能.大家打开每个选项卡研究下就知道.



测试代码:

```

`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 2022/03/05 23:20:09
// Design Name:
// Module Name: demo
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

```

```

module demo(input clk,
            input rst
            );

reg [5:0] cnt;
reg [4:0] addr;
reg [7:0] wdata;
wire [7:0] rdata;

wire full;
wire empty;

wire we;
assign we=(cnt <= 'd15)?'b1:'b0; // 0~15 写入,16~31 读取

always@(posedge clk or negedge rst)
begin
    if(!rst)
    begin
        cnt <='d0;
    end
    else if(cnt == 'd31)
    begin
        cnt <= 'd0;
    end
    else
    begin
        cnt <= cnt + 'b1;
    end
end

always@(posedge clk or negedge rst)
begin
    if(!rst)
    begin
        wdata <= 'b0;
    end
    else if(cnt < 'd15) // 计算到这里就够了,为什么?看ILA分析.
    begin
        wdata <= wdata + 'b1;
    end
    else
    begin
        wdata <= 'b0;
    end
end

fifo_generator_0 fifo_inst (
    .clk(clk),          // input wire clk
    .din(wdata),        // input wire [7 : 0] din
    .wr_en(we),         // input wire wr_en
    .rd_en(~we),        // input wire rd_en

```

```

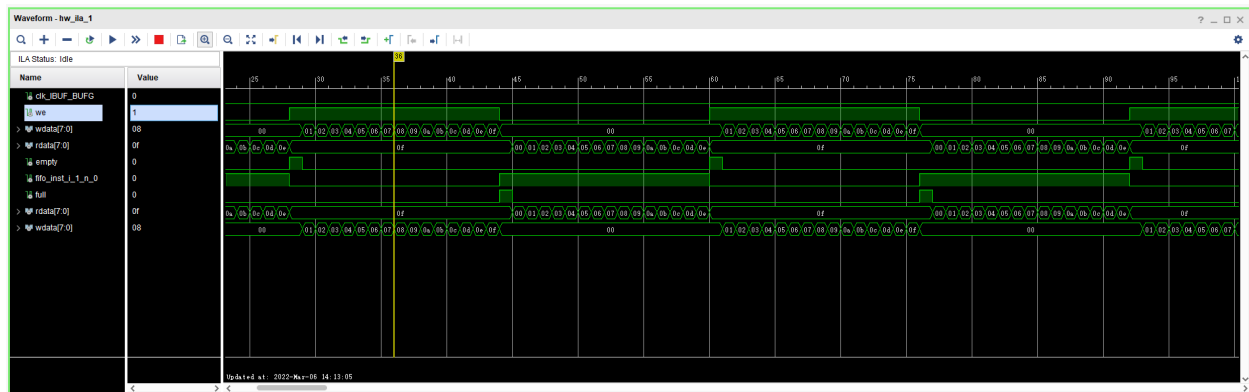
.dout(rdata),    // output wire [7 : 0] dout
.full(full),    // output wire full
.empty(empty)   // output wire empty
);

ila_0 ila_inst (
    .clk(clk), // input wire clk
    .probe0(clk), // input wire [0:0] probe0
    .probe1(we), // input wire [0:0] probe1
    .probe2(~we), // input wire [0:0] probe2
    .probe3(full), // input wire [0:0] probe3
    .probe4(empty), // input wire [0:0] probe4
    .probe5(wdata), // input wire [7:0] probe5
    .probe6(rdata) // input wire [7:0] probe6
);

endmodule

```

ILA测试:



当然,实际使用要配合读取空满标志.