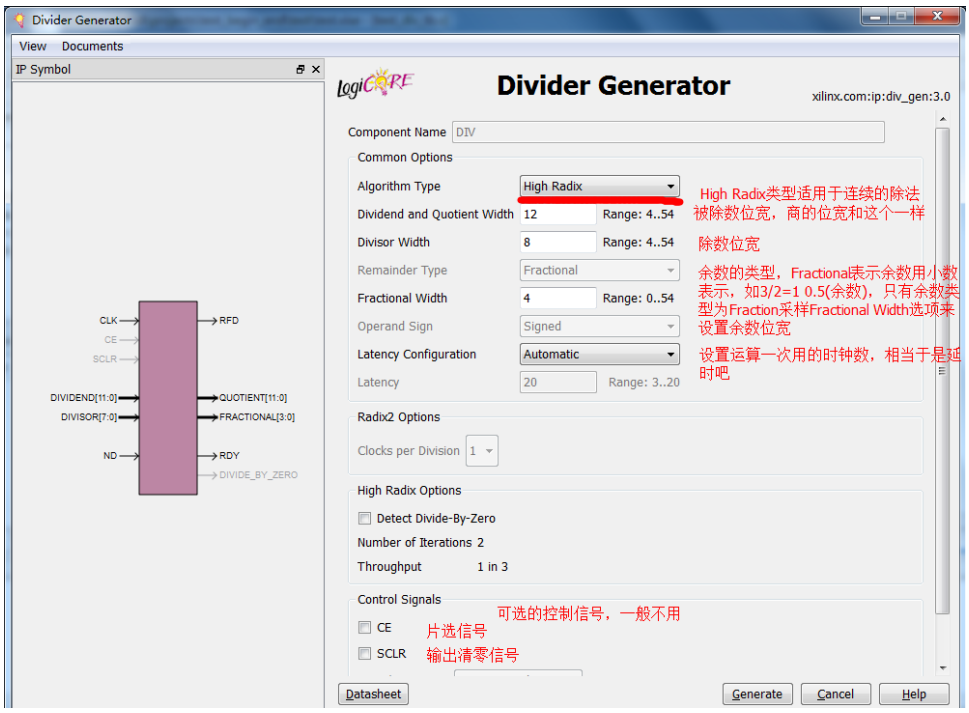
ISE IP 核使用说明（14.5）

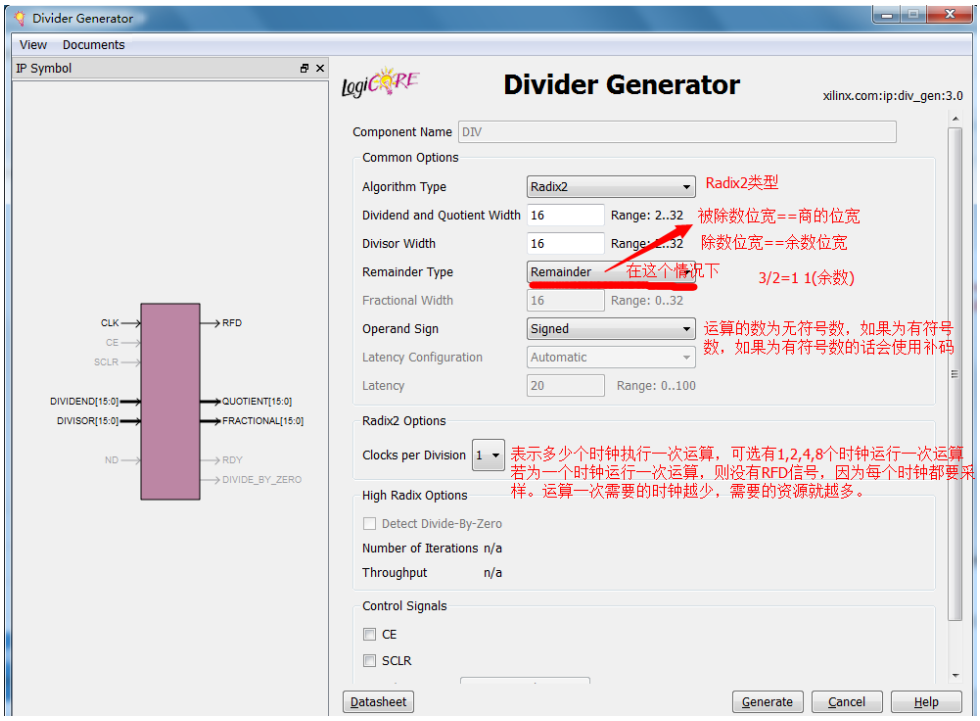
1.除法器.....1
2． CORDIC –IP 核.....4
3.CORIDC-sin/cos.....6
4.CORDIC-SQRT.....10
5.Block Memory.....13
6.Shift-Register.....21
7.ACC 累加器.....25
8.复数乘法器.....27
9.乘法器.....28
10.FFT.....30
11.FIFO.....33

1.除法器

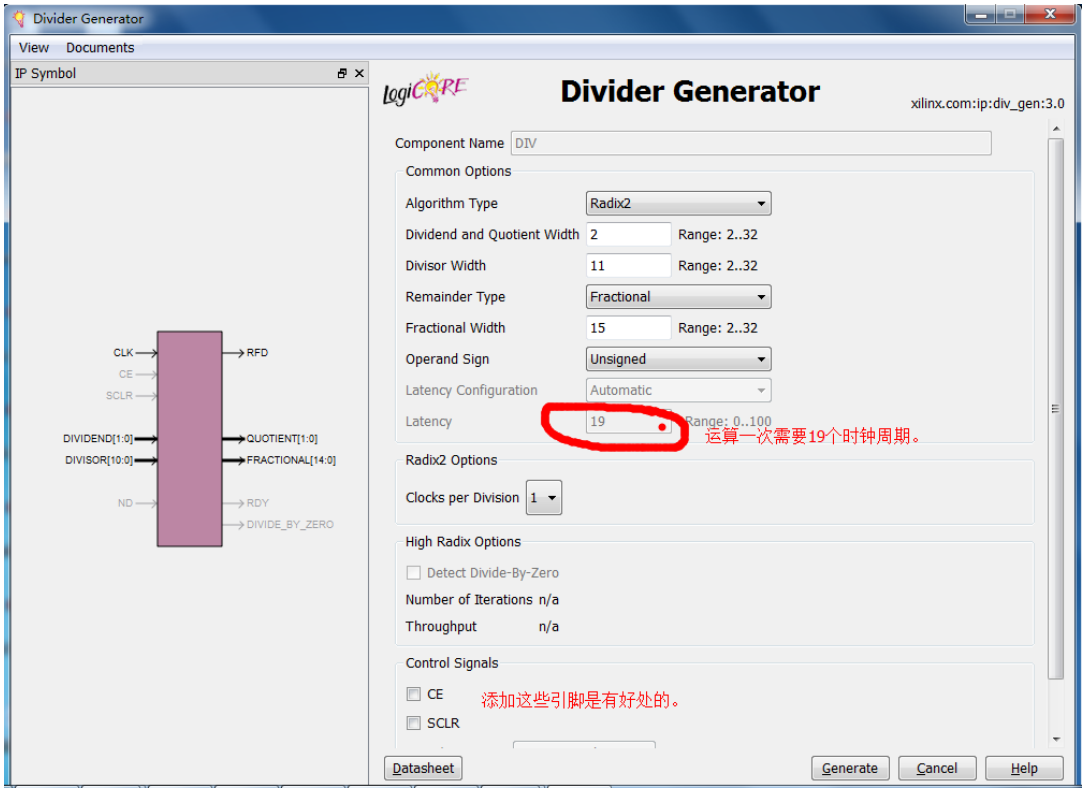
1.设置



使用 High Radix 类型（这样会多出 RDY 和 ND 两个握手信号方便连续的除法）



上面有个地方说错了，Clocks per Division 不是运算需要的时间，Latency 才是需要的时钟数



Radix2 类型，没有 RDY 信号反馈运算是否完成（可能需要对时钟计数来确定是否运算完成,所用时钟是确定值）。

1，除法器内核设置为 Radix2 时，小数位包含符号位已经补码化，小数位可以按照如下方式接在整数位后面:

```
wire [7:0] qv_q_cp;
assign   qv_q_cp=fv_q[26]?(qv_q[7:0]-1'b1):qv_q[7:0];
wire [33:0] v_q_div;
```

assign v_q_div= {qv_q_cp,fv_q[25:0]};

assign xk_re_tmp = xk_re_quoti[26]?{xk_re_quoti-1'b1,xk_re_frac[8:0]}:{xk_re_quoti,xk_re_frac[8:0]};

因为整数部分，如果为负数的时候是取反加一为对应的值，加上小数部分后，整数部分只做取反运算了，所以通过减一来实现对应的加一操作。

当设置为 High Radix 时，小数位也补码化但不包含符号位，可以直接接在整数位后面。

1.小数最高位为符号位，当商为 0 的时候，xk_re_quoti 都是全 0，不管小数部分的正负，所以应该用小数部分的最高为作为判断条件-----

2.因为整数部分，如果为负数的时候是取反加一为对应的值，加上小数部分后，整数部分只做取反运算了，所以通过减一来实现对应的加一操作。

3.所以组合的方式为：assign xk_re_tmp = xk_re_frac[9]?{xk_re_quoti-1'b1,xk_re_frac[8:0]}:{xk_re_quoti,xk_re_frac[8:0]};//2015.4.2

2.模块信号的作用

DIV div (

.clk(clk), // input clk --时钟信号，运算周期可以使一个周期也可以是多个周期

.nd(nd), // input nd---用来提示 IP 核，有新的数据输入，请采样。这是一个新数据的反馈信号

.rdy(rdy), // output rdy 表示运算完成，请取走结果

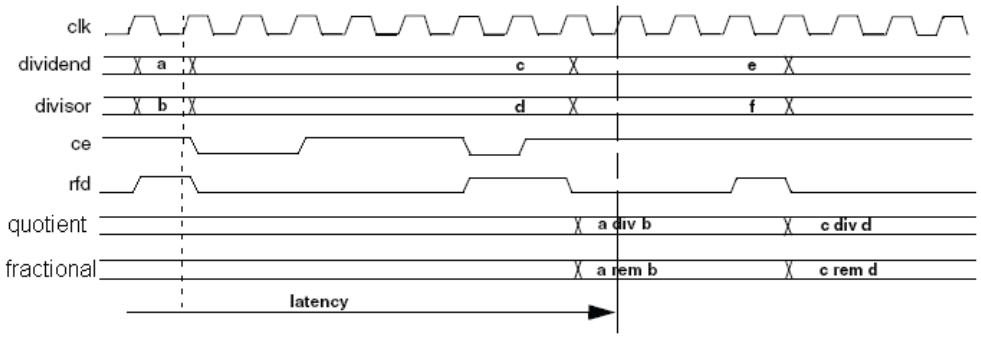
.rfd(rfd), // output rfd--高电平表示采样，表示已经采样。需要新的数据输入。这是一个新数据的请求信号。

.dividend(a), // input [11 : 0] dividend ---被除数

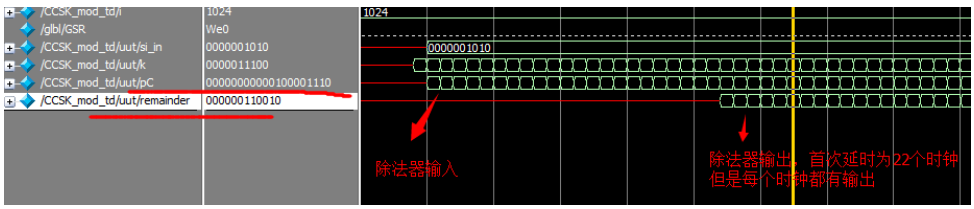
.divisor(b), // input [7 : 0] divisor ---------除数

.quotient(c), // output [11 : 0] quotient --商

.fractional(d)); // output [3 : 0] fractional 余数 若 Remainder Type 为 Remainder 则 :3/2 余数为 1(整数) 若为 Fractional 则：3/2 余数为 0.5(小数，用 4 位二进制数表示为：1000=1*2^-1+.....)



由这个图，好像是要运算完成后才能输入数据，但是测试发现，每个时钟都可以输入的，只是首次延时为 latency



由仿真结果看只有在 RDY 为高的时候，输出结果才是正确的。如果错过恰当的时间去取数据可能取到的是错误的数据，所以要子啊 RDY 一为高就要取输出的数据

2． CORDIC –IP 核

CORDIC:就是“广义的坐标旋转数字运算”

CORIDC 支持的运算：1.极坐标与直角坐标的转换

2.三角函数，sin,cos

3.sinh,cons

4.Asin,Asinh

5.平方根

CORIDC 的输出量化误差：1.由于输入的量化噪声的影响，误差为 1/2LSB

2.由于内部运算的误差，这个误差可以由输入的位宽的增大来减小。

3.两种误差影响都是在小输入的时候影响较为明显，大输入的时候几乎没有影响。

CORIDC architectural configurations（核结构配置）：Parallel，Word Serial（并行和字串行）

Word Serial:完成一次运算要多个时钟，消耗的资源比较少。Nbit 的输出需要 N 个时钟周期才能完成计算，只用了一个 shift Add_Sub stage（一个运算模块），通过反馈计算多位的数据，没计算一位需要反馈 N 次，作用需要 N 个时钟才有一个有效的输出。

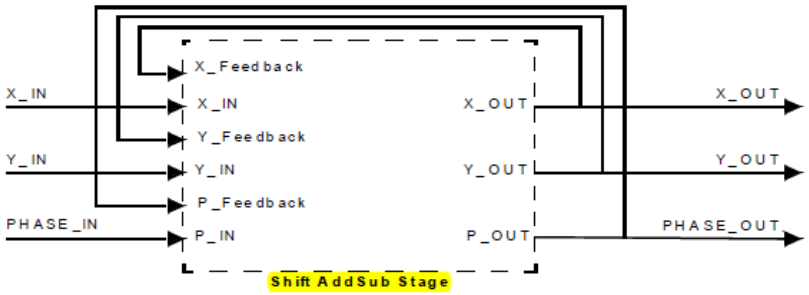


Figure 6: **Word Serial Architecture Configuration**

Word Serial Timing

每隔一个周期才采一次样，要使用ND,RFD信号

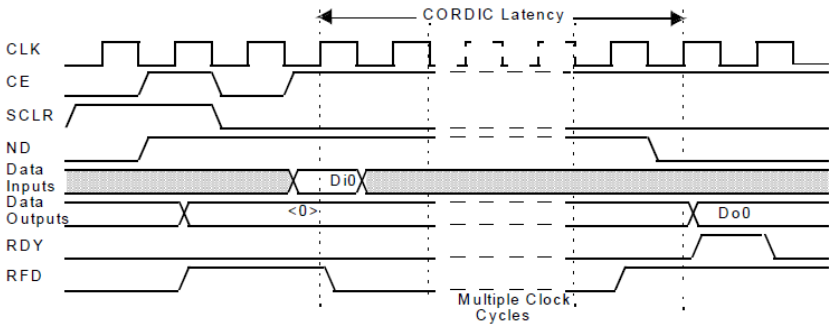


Figure 8: **Control Signal Timing Diagram (Word Serial Architecture)**

Parallel:可以实现单时钟实现一次运算，但是要消耗大量的资源。由下图可知，如果输出的一个 Nbit 的数据需要 N 个串行的 Shift Add_Sub Stage，所以从输入第一个数据到接收到第一个数据的输出需要 N 个时钟的延时，但是接下来的每个时钟都要数据输出，相当于是使用了流水线。

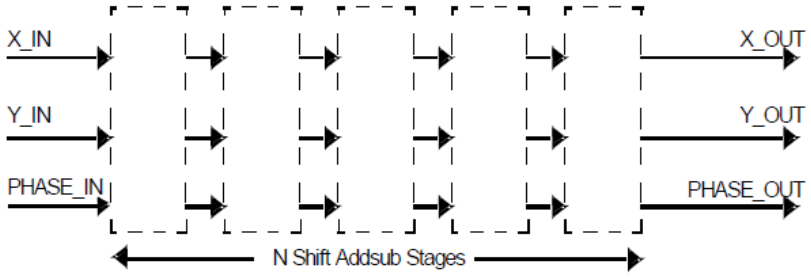


Figure 7: **Parallel Architectural Configuration**

Parallel Architecture Timing

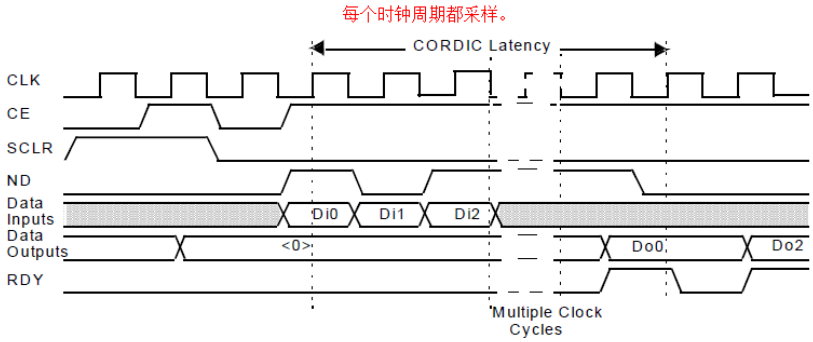


Figure 9: **Control Signal Timing Diagram (Parallel Architecture)**

CORIDIC IP 的数据表示形式。

数据信号

1.在函数配置为 Rotate,Translate，sin,cos,Atan 时，X_IN,Y_IN,X_OUT,Y_OUT 数据信号的使用的是 2 定点二进制的补码，且用 2 位来表示整数，其余的为来表示小数。

Using a 10-bit word width, +1 and -1 are represented as:

"0100000000" => 01.00000000 => +1.0

"1100000000" => 11.00000000 => - 1.0

2.对于平方根函数，输入 X_IN，和输出 X_OUT，可以表示为无符号小数和无符号整数形式。若为 Unsigned Fractional(无符号小数），则 0<=X_IN<+2，即输入输出都为小于 2 的正数，若为 Usigned Integer，则 0<=X_IN<2^N，即输入输出都为整数，没有小数部分（小数部分被舍弃）。且当为 Usigned Fractional 的时候，数据固定第 1bit 为整数，其他的 bit 为小数部分

Unsigned Fractional

1100000000=>1.100000000=>1.5

Unsigned Interger

0000001000=>0000001000=>8

相位信号

PHASE_IN and PHASE_OUT 使用的而是定点的二进制补码，且用 3bit 位来表示整数，其余的为来表示小数。（有符号数）

若相位设置为“Radians”

-Pi <= (PHASE_IN) <= Pi

In 2Q7, or Fix10_7, format values, +Pi and -Pi are represented as:

"01100100100" => 011.00100100 => +3.14

"10011011100" => 100.11011100 => - 3.14

若相位设置为“Scaled Radians”(归一化的相位)

-1 <= (PHASE_IN) <= +1

In 2Q7, or Fix10_7 format values, +1 and -1 are represented as:

"0010000000" => 001.0000000 => +1.0

"1110000000" => 111.0000000 => - 1.0

Q Numbers Format---一定是有符号数，无符号数用 UFix 表示。

一个符号位接着 Xbit 整数和 Nbit 小数，数据总长度为 1+x+N.如 1Q7:"0010000000" => 001.0000000 => +1.0

也可以用 FIX 来表示：Fix(1+X+N)_N.即 Fix(总位数)_小数位数。Q15 表示没有整数位有一个符号位和 15 个小数位。用 FIX 表示为：Fix(16)_15.

映射不同的数据格式（默认为上面的数据格式，要映射其他数据格式要手动配置）

Rotate, Translate, Sin, Cos and Atan，把数据信号映射到可选的数据格式（即固定的整数位数的小数格式），输入数据 X_IN 映射到可选的数据格式上。若输入输出位宽相同，则默认的输出也会映射到与输入相同的数据格式上（实际计算结果也就是与输入位宽相同)若输出的位宽比输入的位宽小，则会相应的减少输出数据的小数位(在实际计算结果下去掉最后面的小数位），输出位宽小于输入位宽可能找出输出数据产生较大的误差。

如输出实际为 FIX10_8: 00.10101001

若输出位宽设为 8 则输出数据格式

为 Fix8_6:00.1010110,去掉了最好两位。

Square Root Functional Configuration

CORE Generator GUI and Parameters（核生成器用户界面和参数）

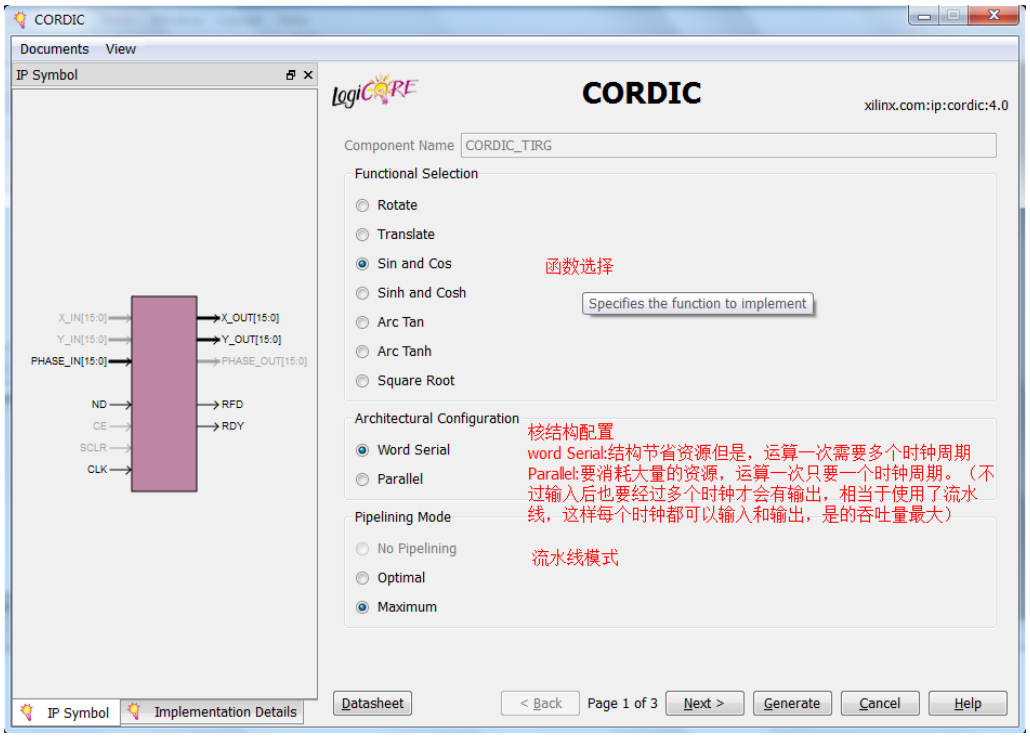
粗旋转，即把输入输出数据的范围有第一象（-pi/4~pi/4）限扩展到了这个圆平面(-pi~pi),在运算 Vector rotation, Vector translation, Sin and Cos, and ArcTan 时这个功能是默认打开的，但在运算 Sinh and Cosh, ArcTanh, and Square Root 时不需要这个功能.

3.CORIDC-sin/cos

粗旋转模块可以使输入是整个角平面。这个功能是默认开启的，用户也可以手动取消。

输入角度 Pin 是一个二进制的定点补码，用 3bit 来表示整数，7bit 来表示小数。输出也是一个二进制补码，2bit 来表示整数，余下比特来表示小数。

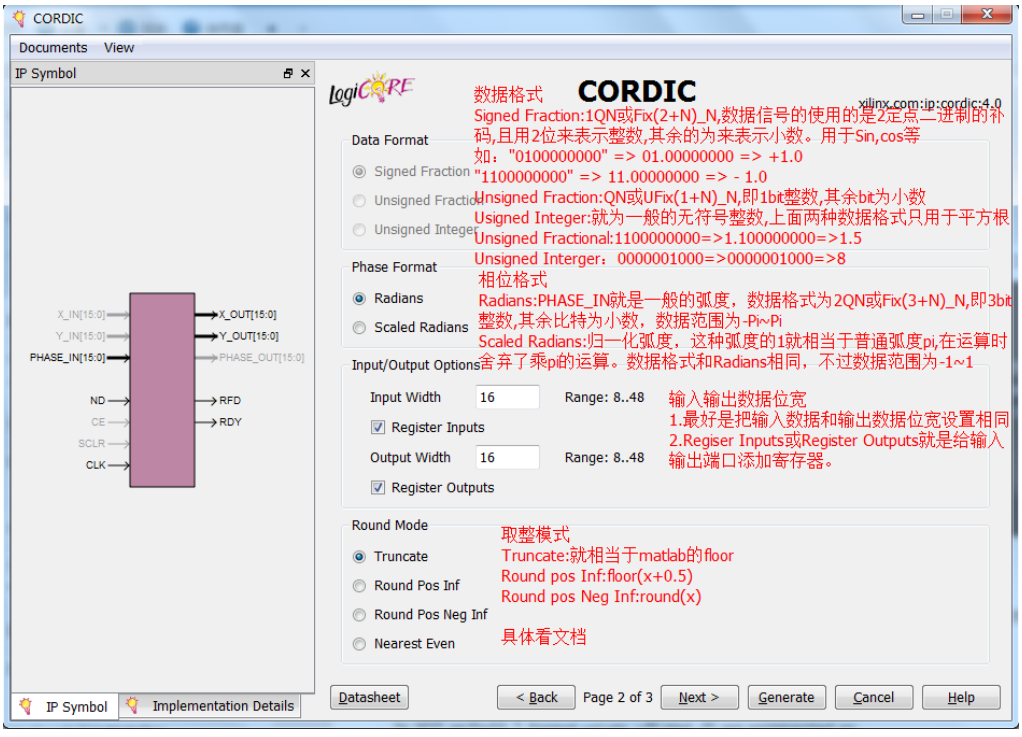
1.核生成配置



Parallel 吞吐量为每个时钟输入和输出一个数据，但是首次延时为输入相位的位宽数。

Parallel 时运算一次需要的时钟数

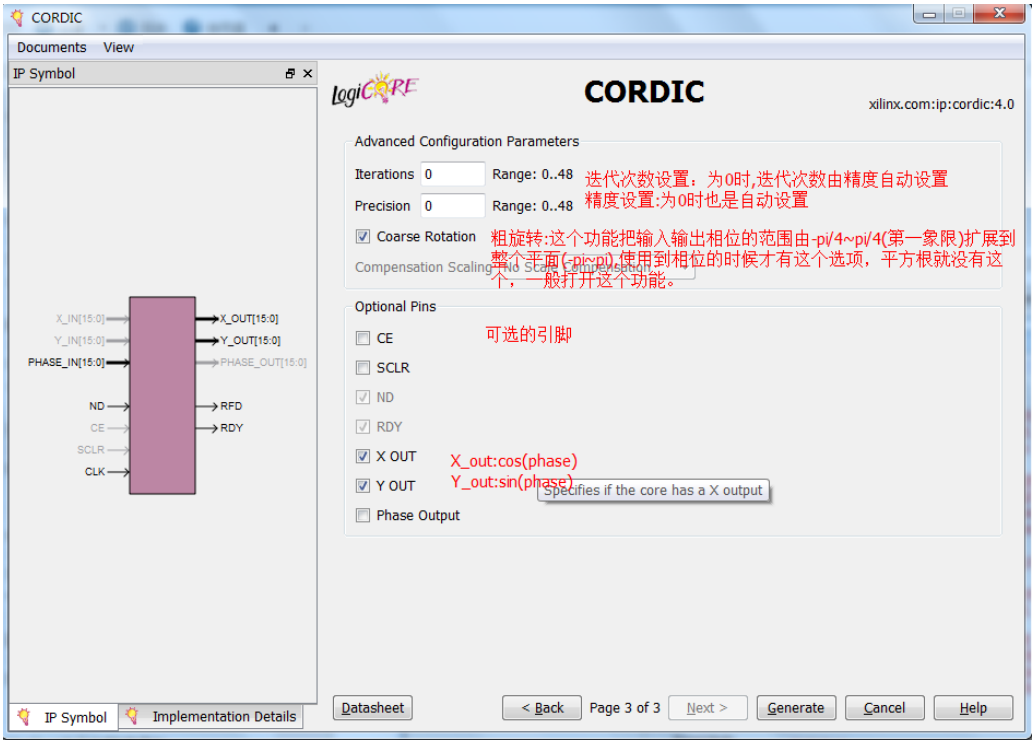
A parallel CORDIC core with N bit output width has a latency of N cycles and produces a new output every cycle. The implementation size of a parallel CORDIC core is directly proportional to the internal precision times the number of iterations. Paralle结果的运算的时钟周期数为输入相位的位宽数



- **Round Mode:** The CORDIC core provides four rounding modes. Table 19 illustrates the behavior of the different Rounding modes.
  - **Truncate:** The X_OUT, Y_OUT, and PHASE_OUT outputs are truncated.
  - **Positive Infinity:** The X_OUT, Y_OUT, and PHASE_OUT outputs are rounded such that 1/2 is rounded up (towards positive infinity). It is equivalent to the MATLAB function floor(x+0.5).
  - **Pos Neg Infinity:** The outputs X_OUT, Y_OUT, and PHASE_OUT are rounded such that 1/2 is rounded up (towards positive infinity) and -1/2 is rounded down (towards negative infinity). It is equivalent to the MATLAB function round(x).
  - **Nearest Even:** The X_OUT, Y_OUT, and PHASE_OUT outputs are rounded toward the nearest even number such that a 1/2 is rounded down and 3/2 is rounded up.

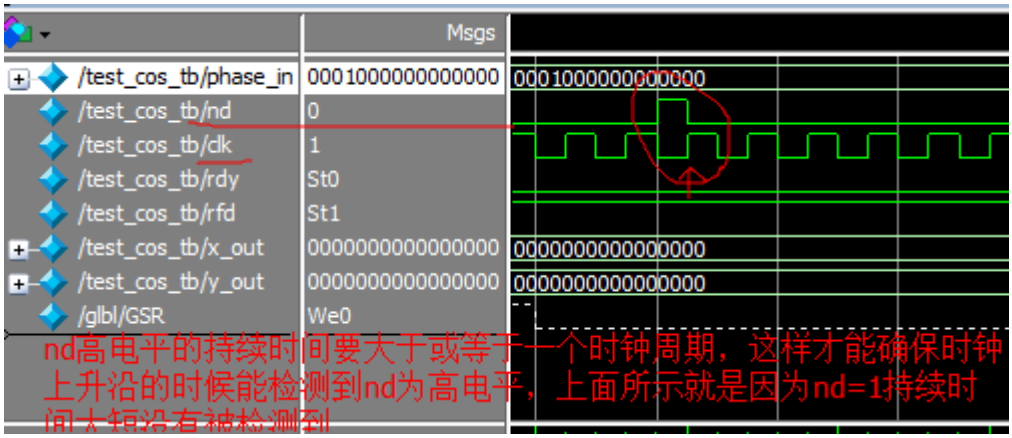
Table 19: **Rounding Modes**

	Truncate	Pos Neg Infinity	Positive Infinity	Nearest Even
1.50	1	2	2	2
1.00	1	1	1	1
0.50	0	1	1	0
0.25	0	0	0	0
0.00	0	0	0	0
- 0.25	-1	0	0	0
- 0.50	-1	-1	0	-1
- 0.75	-1	-1	-1	-1



3.模块使用说明

```
CORDIC_TIRG your_instance_name (
    .phase_in(phase_in), // input [15 : 0] phase_in--相位角输入，使用了归一化的相位,1/2 表示 pi/2
    .nd(nd), // input nd----------------------------外部数据发送者通知 IP 核,有新的数据输入,请采样
    .x_out(x_out), // output [15 : 0] x_out----------cos(phase_in)
    .y_out(y_out), // output [15 : 0] y_out----------sin(phase_in)
    .rdy(rdy), // output rdy-------------------------计算完成信号，通知数据接收者取走结果
    .rfd(rfd), // output rfd---------------------IP 核请求新的输入数据,请求外部数据发送者发送新的输入数据
    .clk(clk) // input clk--------------------------同步信号。。
);
```

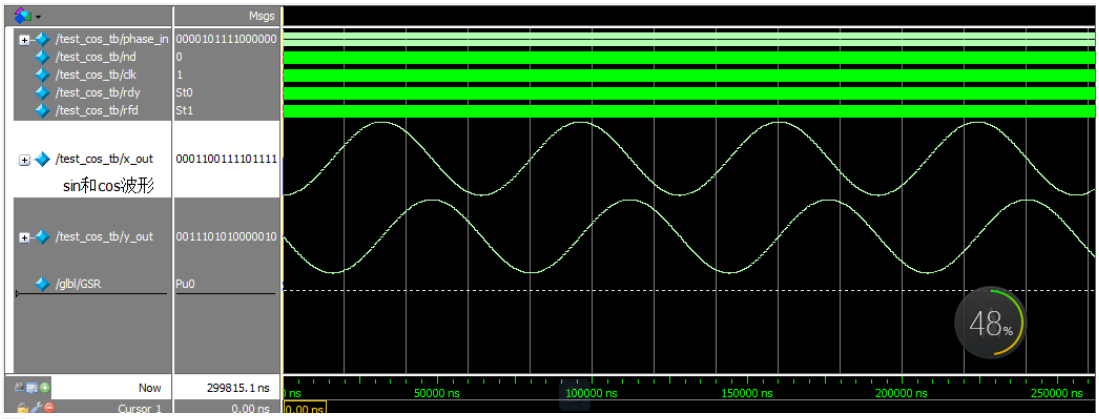


还要就是模块开始使用的时候要延时一段时间，再把 nd 设为 nd=1 才能被采样的。

```
    initial begin
        nd = 0;
        clk = 0;
        forever #5 clk = ~clk;
    end
    initial begin
        #100 phase_in = 16'b0001_0000_0000_0000;//=>000.1=>0.5,注意要加上16'b否则为十进制数了。
        nd = 1;
        #10 nd = 0;                                  //nd=1的持续时间要大于一个时钟周期
    end
endmodule
```

延时了一段时间再送数据,开始运算的是要延时一下接下来的发送数据就根据RFD

nd持续时间大于一个时钟周期



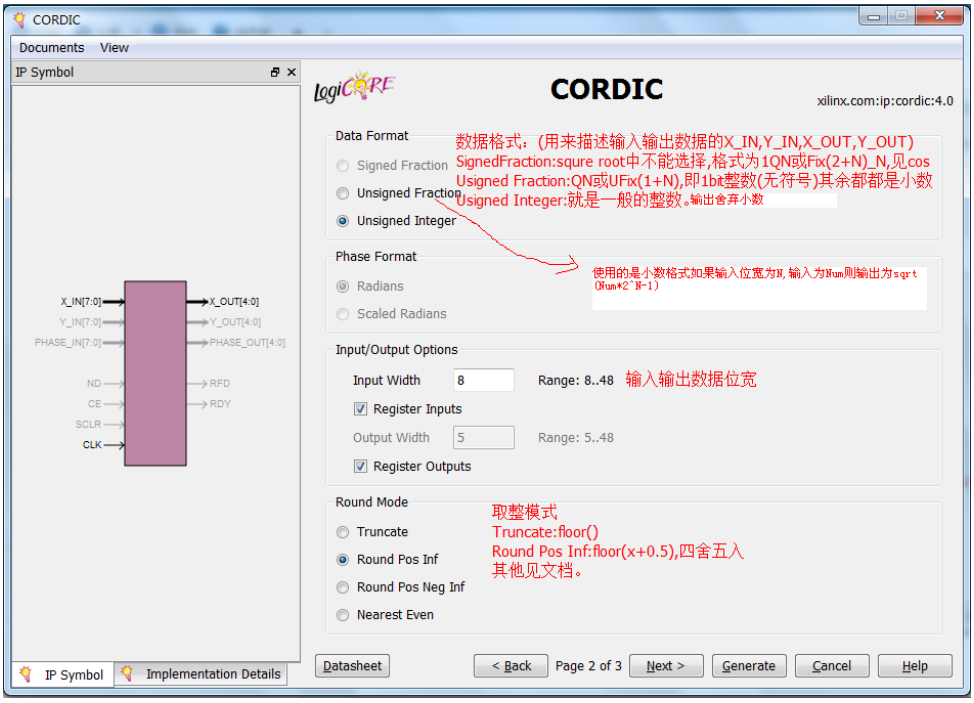

```
    initial begin
        phase_in = 16'b1110_0000_0000_0000; //-pi
        nd = 0;
        clk = 0;
        forever #5 clk = ~clk;
    end
    /*initial begin
        #100 phase_in = 16'b0001_0000_0000_0000;//=>000.1=>0.5,注意要加上16'b否则为十进制数了。
        nd = 1;
        #10 nd = 0;                                  //nd=1的持续时间要大于一个时钟周期
    end*/

    always@(posedge rfd)                        //当rfd上升沿的时候，表示IP核需要输入新的数据
        begin
            #100 phase_in = phase_in + 16'b0000_0100_0000;//步进
            if(phase_in == 16'b0010_0000_0000_0000)//为pi的时候sin(-pi)=sin(pi),cos(-pi)=cos(pi)
                phase_in = 16'b1110_0000_0000_0000;
            else
                phase_in = phase_in;
            nd = 1;
            #10 nd = 0;
        end
endmodule
```

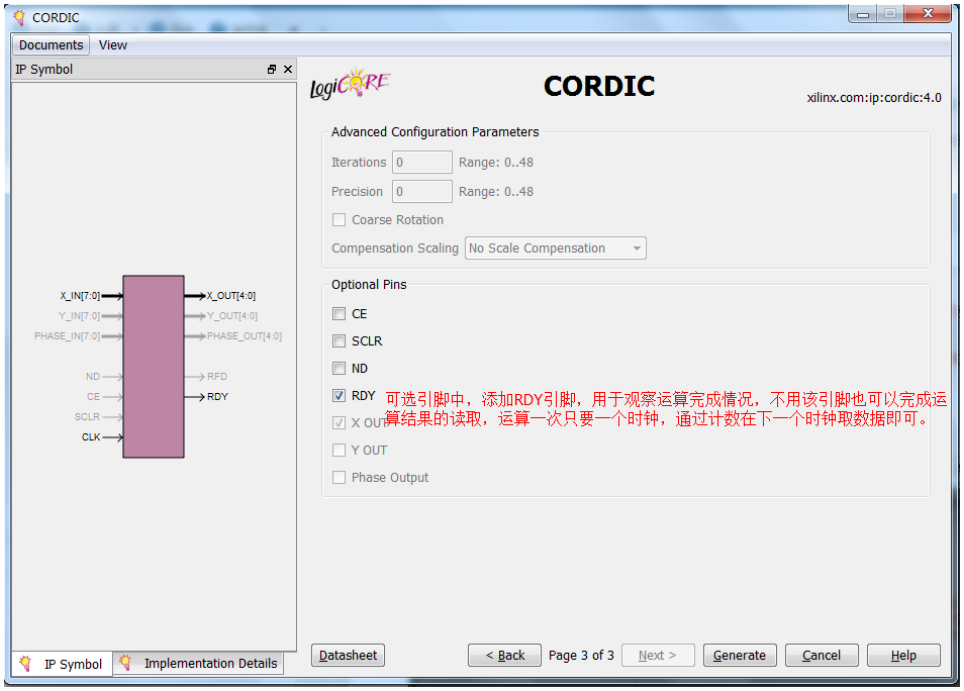
2QN，-1的补码

这里的数据用的是补码

4.CORDIC-SQRT

1.GUI 设置





2.模块使用说明

CORDIC_Sqrt your_instance_name (

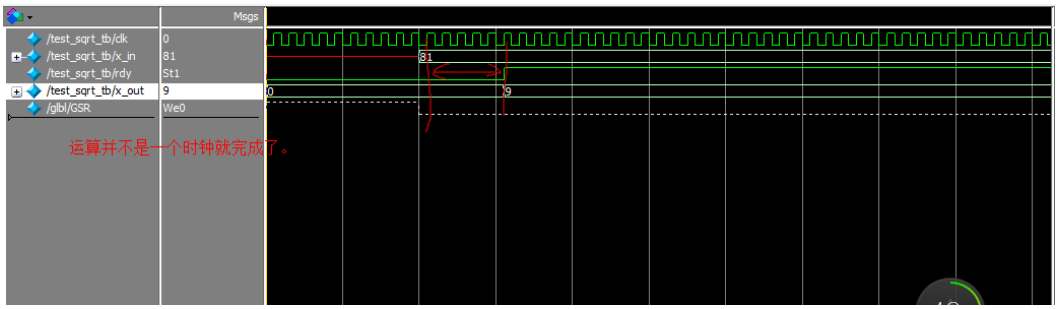
.x_in(x_in), // input [7 : 0] x_in--------输入数据，要与 GUI 中设置的数据格式相同

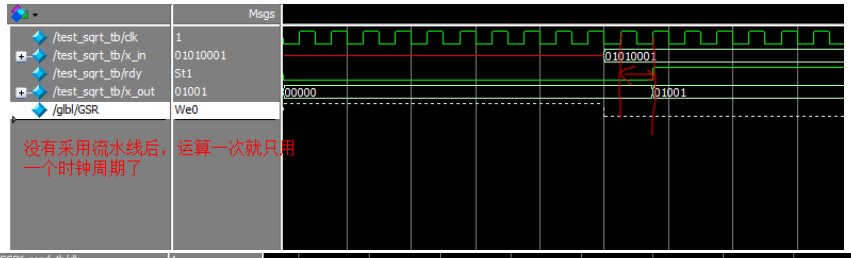
.x_out(x_out), // output [4 : 0] x_out-输出数据

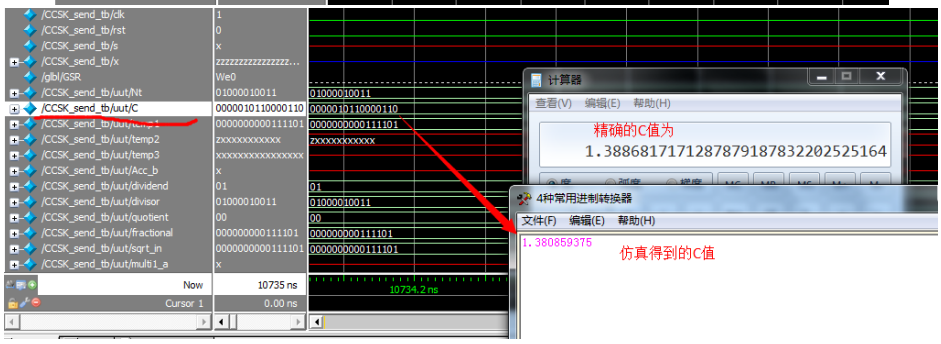
.rdy(rdy), // output rdy----------计算是否完成的状态信号，完成后为高电平

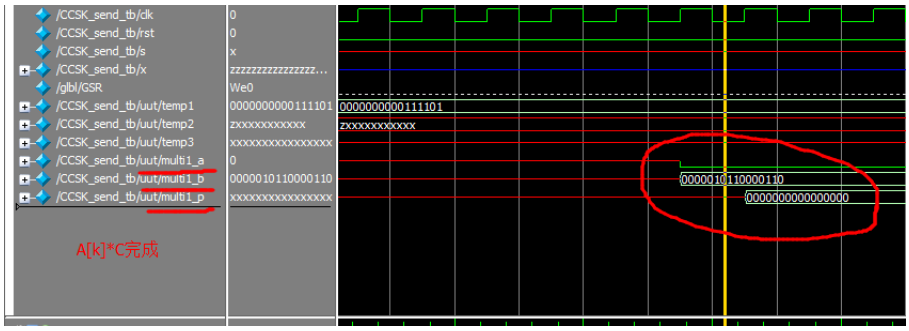
.clk(clk) // input clk----------时钟

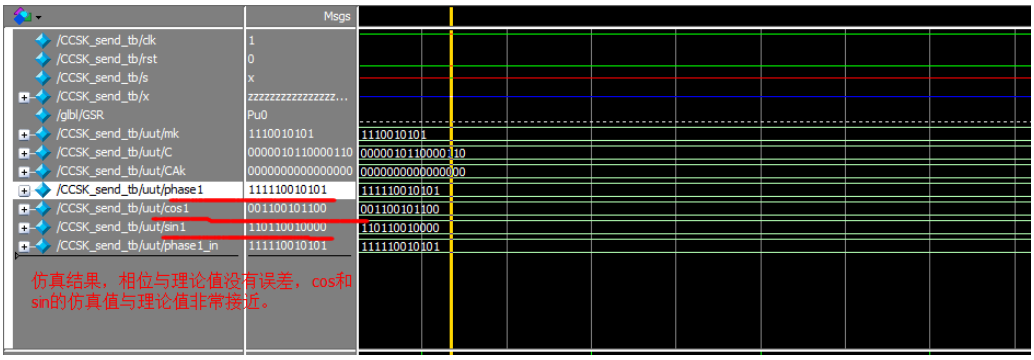
);

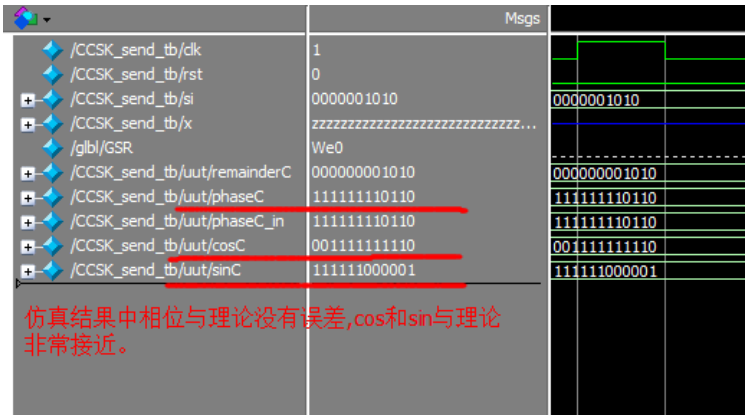

Msgs
/test_sqrt_tb/clk
/test_sqrt_tb/x_in
/test_sqrt_tb/rdy
/test_sqrt_tb/x_out
/glbl/GSR
0
81
St1
9
We0
运算并不是一个时钟就完成了。


Msgs
/test_sqrt_tb/clk
/test_sqrt_tb/x_in
/test_sqrt_tb/rdy
/test_sqrt_tb/x_out
/glbl/GSR
1
01010001
St1
01001
We0
00000
没有采用流水线后，运算一次就只用一个时钟周期了


/CCSK_send_tb/clk
/CCSK_send_tb/rst
/CCSK_send_tb/s
/CCSK_send_tb/x
/glbl/GSR
/CCSK_send_tb/uut/Nt
/CCSK_send_tb/uut/C
/CCSK_send_tb/uut/temp2
/CCSK_send_tb/uut/temp3
/CCSK_send_tb/uut/Acc_b
/CCSK_send_tb/uut/dividend
/CCSK_send_tb/uut/divisor
/CCSK_send_tb/uut/quotient
/CCSK_send_tb/uut/fractional
/CCSK_send_tb/uut/sqrt_in
/CCSK_send_tb/uut/multi1_a
Now
10735 ns
Cursor 1
0.00 ns
10734.2 ns
计算器
查看(V) 编辑(E) 帮助(H)
精确的C值为
1.38868171712878791878322025251 64
4种常用进制转换器
文件(F) 编辑(E) 帮助(H)
1.380859375
仿真得到的C值


/CCSK_send_tb/clk
/CCSK_send_tb/rst
/CCSK_send_tb/s
/CCSK_send_tb/x
/glbl/GSR
/CCSK_send_tb/uut/temp1
/CCSK_send_tb/uut/temp2
/CCSK_send_tb/uut/temp3
/CCSK_send_tb/uut/multi1_a
/CCSK_send_tb/uut/multi1_b
/CCSK_send_tb/uut/multi1_p
A[k]*C完成


Msgs
/CCSK_send_tb/clk
/CCSK_send_tb/rst
/CCSK_send_tb/s
/CCSK_send_tb/x
/glbl/GSR
/CCSK_send_tb/uut/mk
/CCSK_send_tb/uut/C
/CCSK_send_tb/uut/CAk
/CCSK_send_tb/uut/phase1
/CCSK_send_tb/uut/cos1
/CCSK_send_tb/uut/sin1
/CCSK_send_tb/uut/phase1_in
Pu0
1110010101
001100101100
110110010000
仿真结果，相位与理论值没有误差，cos和sin的仿真值与理论值非常接近。



5.Block Memory

宽度 x 深度

存储空间分配算法

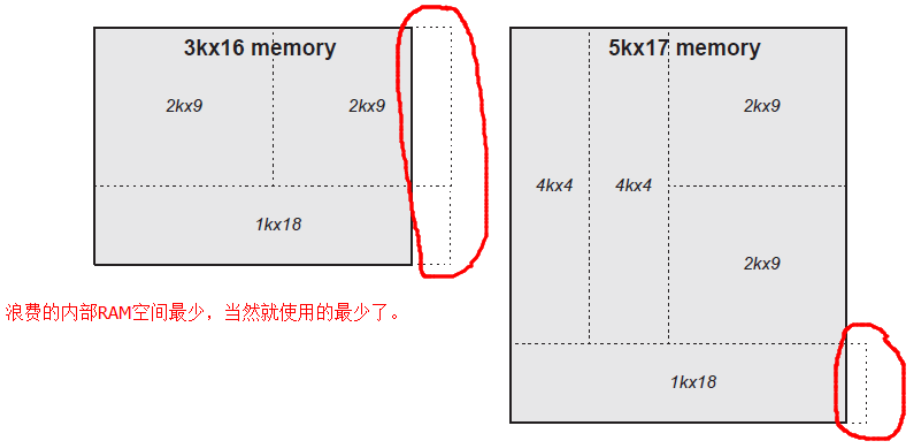
Selectable Memory Algorithm

The Block Memory Generator core arranges block RAM primitives according to one of three algorithms: the minimum area algorithm, the low power algorithm and the fixed primitive algorithm.

使用最少的RAM基本单元

使内部容易读取，

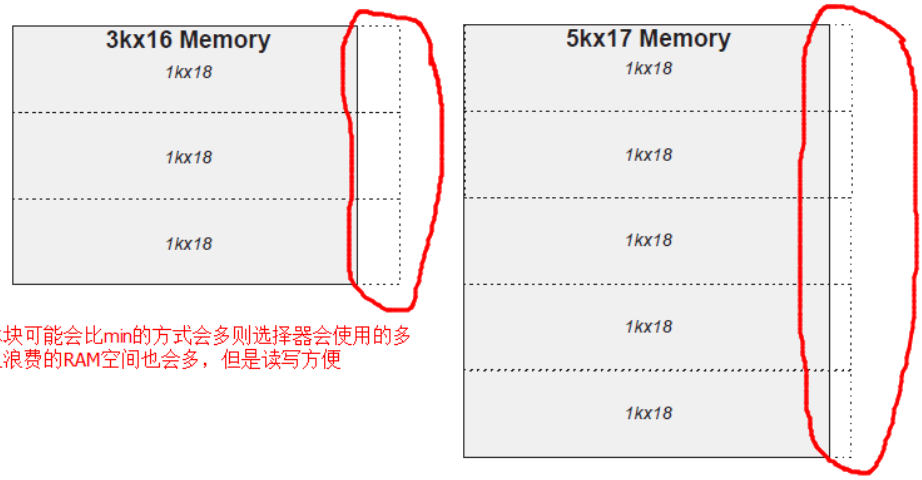
使用固定的RAM基本单元来生成需要的RAM



浪费的内部RAM空间最少，当然就使用的最少了。

Figure 3-6: **Examples of the Minimum Area Algorithm**

Minimum Area 使用的块也是最少的，当然使用的数据选择器也是最少的。



基本块可能会比min的方式会多则选择器会使用的多，且浪费的RAM空间也会多，但是读写方便

Figure 3-7: **Examples of the Low Power Algorithm**

LOW Power 这种方式读写一次的时候会操作最少的基本块

这种分配方式要求用户，知道自己要创建的RAM使用这种分配方式会是什么样的结构，否则配置出来的会很浪费，如下

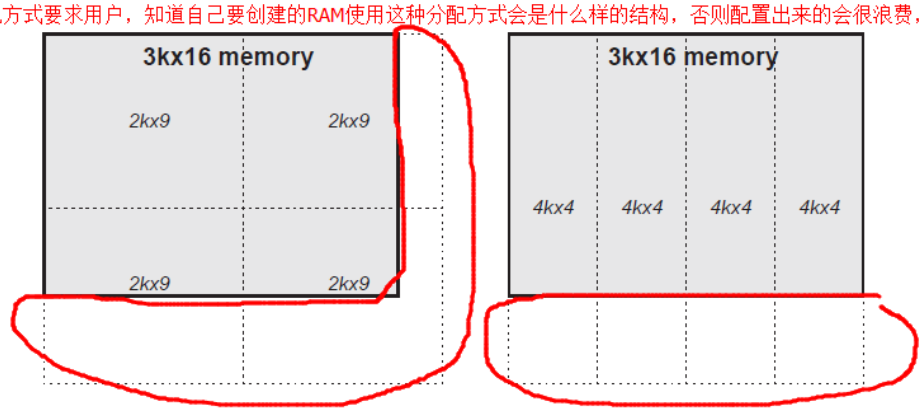


Figure 3-8: **Examples of the Fixed Primitive Algorithm**

3 种读写操作模式

- **Write First Mode**: In WRITE_FIRST mode, the input data is simultaneously written into memory and driven on the data output, as shown in Figure 3-9. This transparent mode offers the flexibility of using the data output bus during a Write operation on the same port.

输入数据的时候，同时也可以输出数据，并且这里是输入数据先于输出数据，所以输出的数据为刚刚新输入的数据

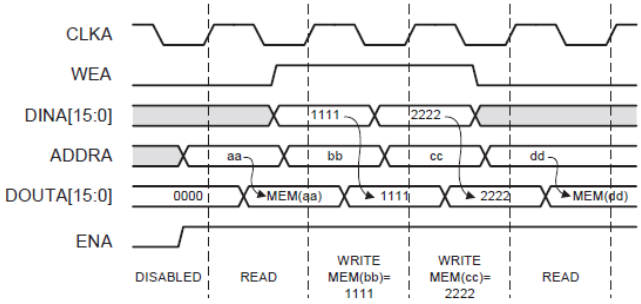


Figure 3-9: **Write First Mode Example**

- **Read First Mode**: In READ_FIRST mode, data previously stored at the Write address appears on the data output, while the input data is being stored in memory. This Read-before-Write behavior is illustrated in Figure 3-10.

读数据的同时写数据，这里读数据优先于写数据，所以读出了的数据是原先的数据，而不是刚刚写入的数据

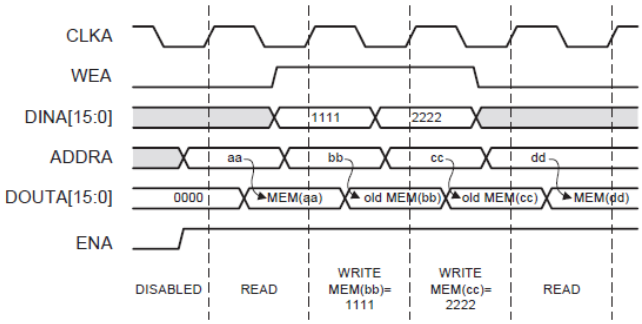


Figure 3-10: **Read First Mode Example**

时钟周期中间准备数据时钟上升沿时候读取数据，是在时钟上升沿读取数据

- **No Change Mode**: In NO_CHANGE mode, the output latches remain unchanged during a Write operation. As shown in Figure 3-11, the data output is still the previous Read data and is unaffected by a Write operation on the same port.

写数据的时候不允许读数据

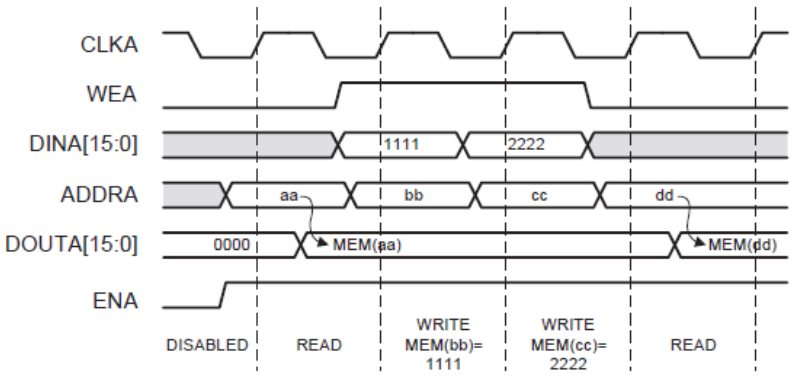


Figure 3-11: **No Change Mode Example**

采样两个端口输入输出。

Table 3-2: **Read-to-Write Aspect Ratio Example Ports**

Interface	Data Width	Memory Depth
Port A Write	64	512
Port A Read	16	2048
Port B Write	256	128
Port B Read	32	1024

A的输入输出端口的位宽和深度不一样，但是总的容量是一样的，B端口也是这样。A、B端口操作的是同一个RAM区

字节输入 byte writes

Byte-Write Example

Consider a Single-port RAM with a data width of 24 bits, or 3 bytes with byte size of 8 bits. The Write enable bus, WEA, consists of 3 bits. Figure 3-14 illustrates the use of byte-writes and shows the contents of the RAM at address 0. Assume all memory locations are initialized to 0.

WEA指定输入信号DINA中的那几个字节作为输入到内部ram,addra指定输入那个RAM单元

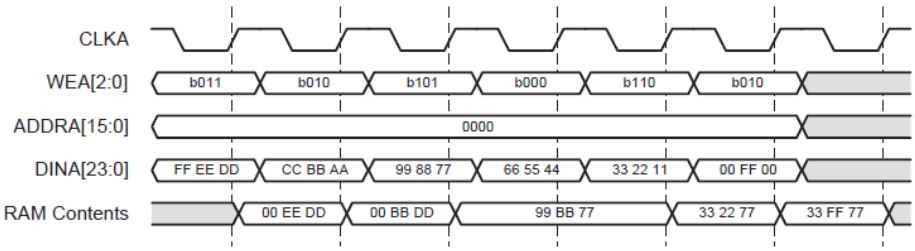


Figure 3-14: **Byte-Write Example**

A,B 端口同时对一个地址写的冲突问题

Collisions and Synchronous Clocks: General Guidelines

Synchronous clocks cause a number of special case collision scenarios, described below.

- **Synchronous Write-Write Collisions**: A Write-Write collision occurs if both ports attempt to Write to the same location in memory. The resulting contents of the memory location are unknown. Note that Write-Write collisions affect memory content, as opposed to Write-Read collisions which only affect data output.
- **Using Byte-Writes**: When using byte-writes, memory contents are not corrupted when separate bytes are written in the same data word. RAM contents are corrupted only when both ports attempt to Write the same byte. Figure 3-15 illustrates this case. Assume ADDRA = ADDRB = 0.

当A,B端口同时向一个地址写数据的时候，会发生冲突，使得地址空间的内容为未知。

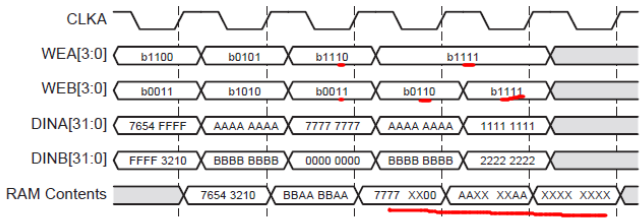


Figure 3-15: **Write-Write Collision Example**

对同一个地址同时写和读的冲突

- **Synchronous Write-Read Collisions**: A synchronous Write-Read collision may occur if a port attempts to Write a memory location and the other port reads the same location. While memory contents are not corrupted in Write-Read collisions, the validity of the output data depends on the Write port operating mode.
  - If the Write port is in READ_FIRST mode, the other port can reliably read the old memory contents.
  - If the Write port is in WRITE_FIRST or NO_CHANGE mode, data on the output of the Read port is invalid.
  - In the case of byte-writes, only bytes which are updated will be invalid on the Read port output.

Figure 3-16 illustrates Write-Read collisions and the effects of byte-writes. DOUTB is shown for when port A is in WRITE_FIRST mode and READ_FIRST mode. Assume ADDRA = ADDRB =

0, port B is always reading, and all memory locations are initialized to 0. The RAM contents are never corrupted in Write-Read collisions.

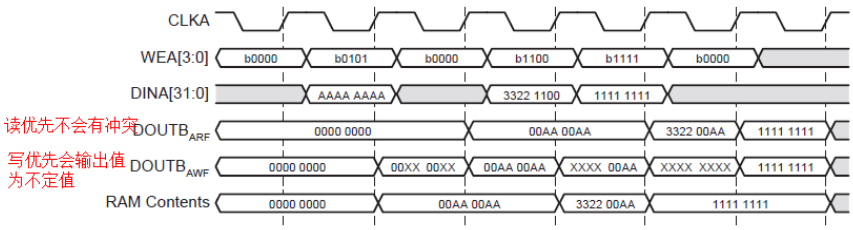


Figure 3-16: **Write-Read Collision Example**

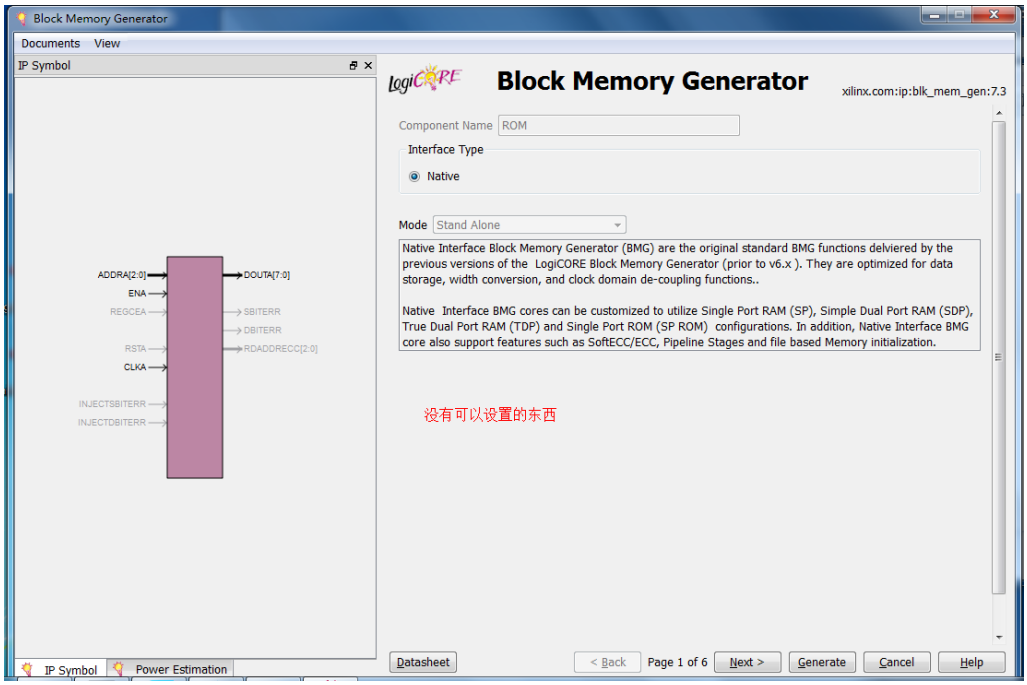
对于 simple Dual-Port Rame

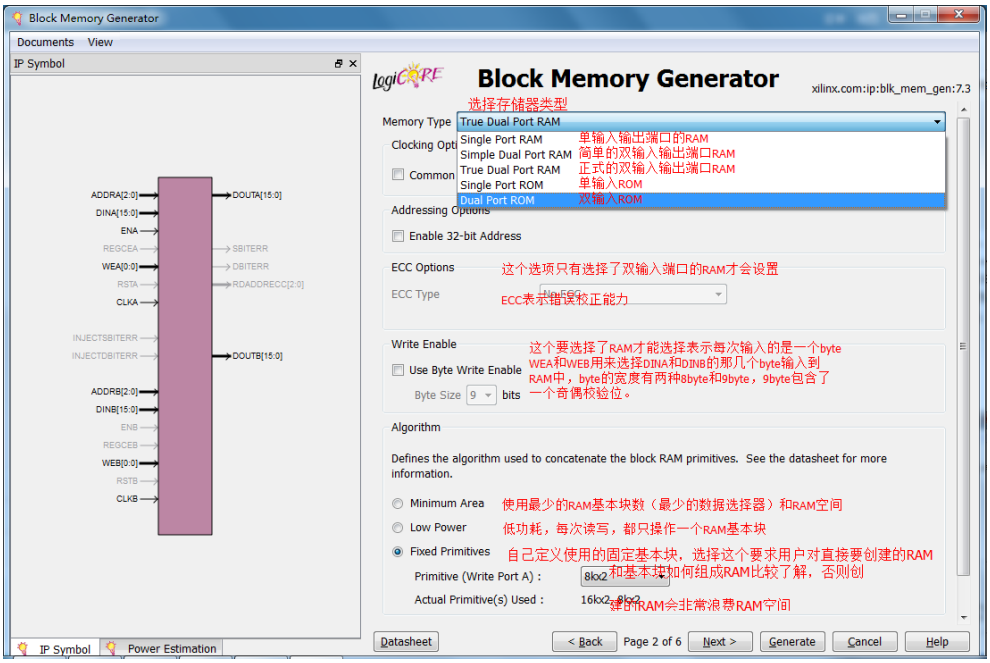
Collisions and Simple Dual-port RAM

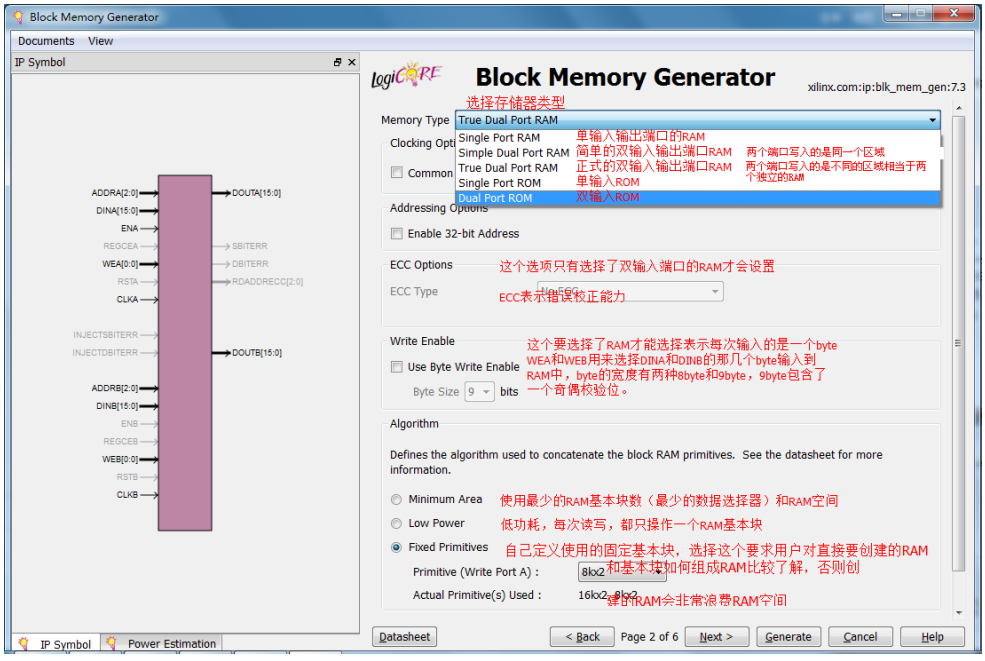
对于Simple Dual-port Ram没有操作模式的选项，但是会自动的选择READ_FIRST后WRITE_FIRST由设备的类型和时钟配置

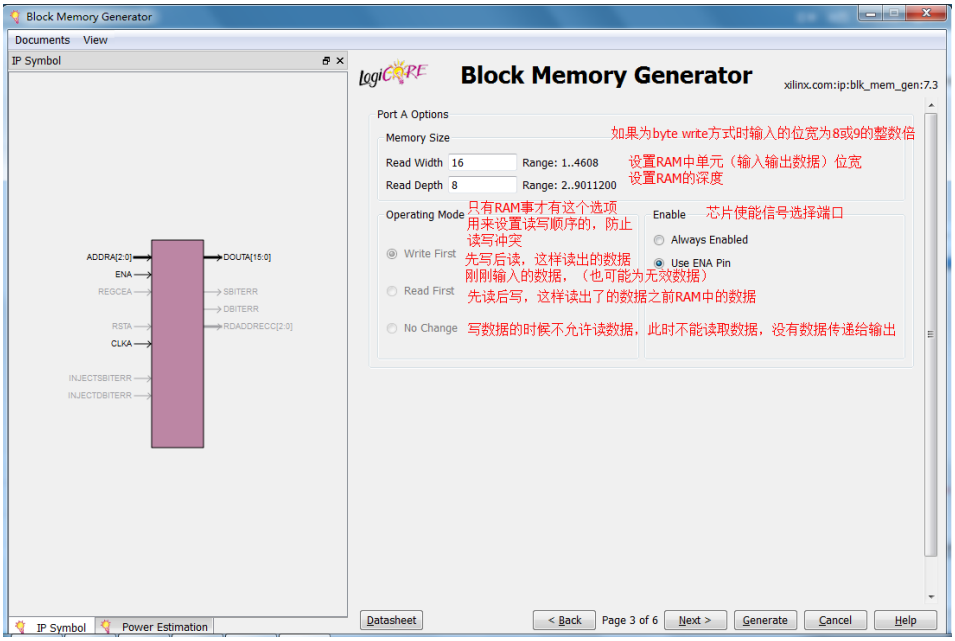
For Simple Dual-port RAM, the operating modes are not selectable, but are automatically set to either READ_FIRST or WRITE_FIRST depending on the target device family and clocking configuration (synchronous or asynchronous). The Simple Dual-port RAM is like a true dual-port RAM where only the Write interface of the A port and the Read interface of B port are connected. The operating modes define the Write-to-Read relationship of the A or B ports, and only impact the relationship between A and B ports during an address collision.

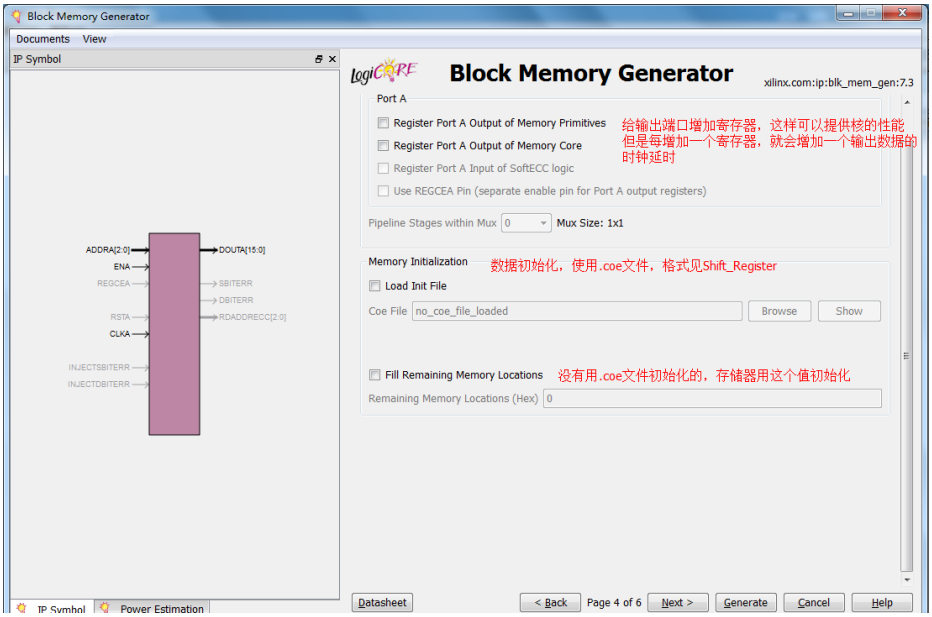
IP 配置

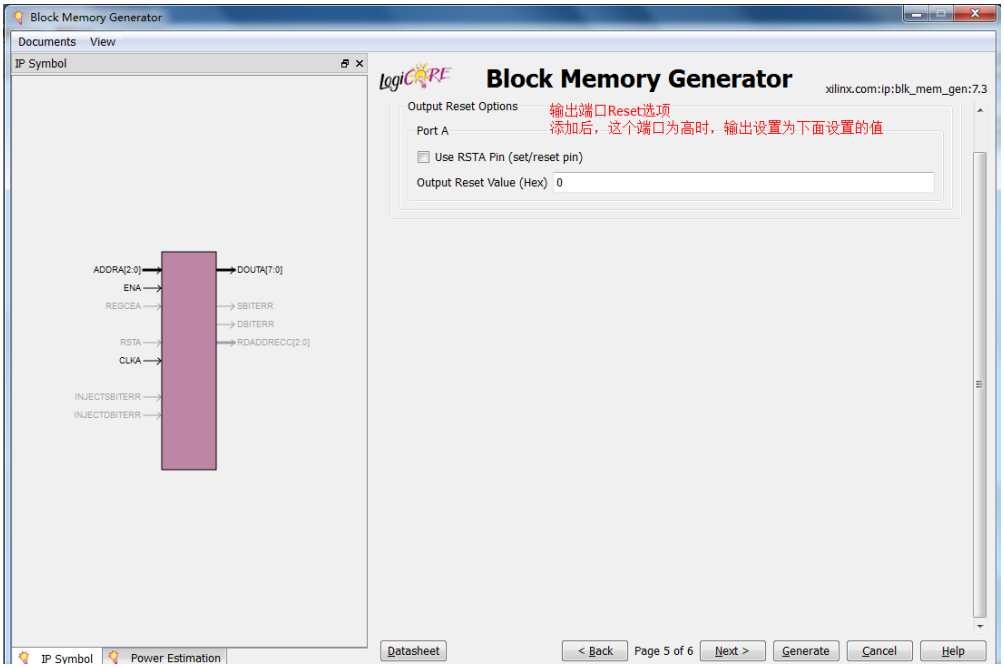


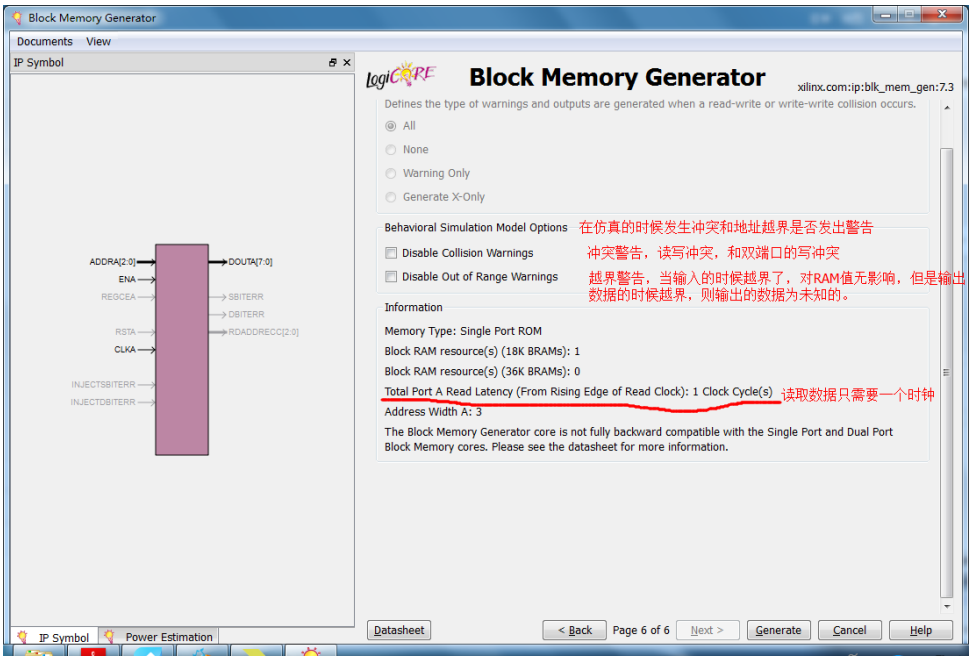

Block Memory Generator
Documents View
IP Symbol
ADDRA[2:0]
DINA[15:0]
ENA
REGCEA
WEA[0:0]
RSTA
CLKA
INJECTSBITERR
INJECTDBITERR
ADDRB[2:0]
DINB[15:0]
ENB
REGCEB
WEB[0:0]
RSTB
CLKB
DOUTA[15:0]
SBITERR
DBITERR
RDADDRECC[2:0]
DOUTB[15:0]
IP Symbol
Power Estimation
LogiCORE
Block Memory Generator
xilinx.com:ip:blk_mem_gen:7.3
选择存储器类型
Memory Type
True Dual Port RAM
Single Port RAM 单输入输出端口的RAM
Simple Dual Port RAM 简单的双输入输出端口RAM
True Dual Port RAM 正式的双输入输出端口RAM
Single Port ROM 单输入ROM
Dual Port ROM 双输入ROM
Clocking Opti
Common
Addressing Options
Enable 32-bit Address
ECC Options
这个选项只有选择了双输入端口的RAM才会设置
ECC Type
No ECC
ECC表示错误校正能力
Write Enable
Use Byte Write Enable
Byte Size 9 bits
这个要选择了RAM才能选择表示每次输入的是一个byte
WEA和WEB用来选择DINA和DINB的那几个byte输入到
RAM中，byte的宽度有两种8byte和9byte，9byte包含了
一个奇偶校验位。
Algorithm
Defines the algorithm used to concatenate the block RAM primitives. See the datasheet for more information.
Minimum Area 使用最少的RAM基本块数（最少的数据选择器）和RAM空间
Low Power 低功耗，每次读写，都只操作一个RAM基本块
Fixed Primitives 自己定义使用的固定基本块，选择这个要求用户对直接要创建的RAM和基本块如何组成RAM比较了解，否则创
Primitive (Write Port A) : 8kx2
Actual Primitive(s) Used : 16kx2, 8kx2
建的RAM会非常浪费RAM空间
Datasheet
< Back
Page 2 of 6
Next >
Generate
Cancel
Help


Block Memory Generator
Documents View
IP Symbol
ADDRA[2:0]
DINA[15:0]
ENA
REGCEA
WEA[0:0]
RSTA
CLKA
INJECTSBITERR
INJECTDBITERR
ADDRB[2:0]
DINB[15:0]
ENB
REGCEB
WEB[0:0]
RSTB
CLKB
DOUTA[15:0]
SBITERR
DBITERR
RDADDRECC[2:0]
DOUTB[15:0]
IP Symbol
Power Estimation
LogiCORE
Block Memory Generator
xilinx.com:ip:blk_mem_gen:7.3
选择存储器类型
Memory Type
True Dual Port RAM
Single Port RAM 单输入输出端口的RAM
Simple Dual Port RAM 简单的双输入输出端口RAM
True Dual Port RAM 正式的双输入输出端口RAM
Single Port ROM 单输入ROM
Dual Port ROM 双输入ROM
两个端口写入的是同一个区域
两个端口写入的是不同的区域相当于两个独立的RAM
Clocking Opti
Common
Addressing Options
Enable 32-bit Address
ECC Options
这个选项只有选择了双输入端口的RAM才会设置
ECC Type
No ECC
ECC表示错误校正能力
Write Enable
Use Byte Write Enable
Byte Size 9 bits
这个要选择了RAM才能选择表示每次输入的是一个byte
WEA和WEB用来选择DINA和DINB的那几个byte输入到
RAM中，byte的宽度有两种8byte和9byte，9byte包含了
一个奇偶校验位。
Algorithm
Defines the algorithm used to concatenate the block RAM primitives. See the datasheet for more information.
Minimum Area 使用最少的RAM基本块数（最少的数据选择器）和RAM空间
Low Power 低功耗，每次读写，都只操作一个RAM基本块
Fixed Primitives 自己定义使用的固定基本块，选择这个要求用户对直接要创建的RAM和基本块如何组成RAM比较了解，否则创
Primitive (Write Port A) : 8kx2
Actual Primitive(s) Used : 16kx2, 8kx2
建的RAM会非常浪费RAM空间
Datasheet
< Back
Page 2 of 6
Next >
Generate
Cancel
Help


Block Memory Generator
Documents View
IP Symbol
Block Memory Generator
xilinx.com:ip:blk_mem_gen:7.3
Port A Options
Memory Size
如果为byte write方式时输入的位宽为8或9的整数倍
Read Width 16 Range: 1..4608
设置RAM中单元（输入输出数据）位宽
Read Depth 8 Range: 2..9011200
设置RAM的深度
Operating Mode
只有RAM事才有这个选项
用来设置读写顺序的，防止读写冲突
Write First
先写后读，这样读出的数据刚刚输入的数据，（也可能为无效数据）
Read First
先读后写，这样读出了的数据之前RAM中的数据
No Change
写数据的时候不允许读数据，此时不能读取数据，没有数据传递给输出
Enable
芯片使能信号选择端口
Always Enabled
Use ENA Pin
ADDRA[2:0]
ENA
REGCEA
RSTA
CLKA
INJECTSBITERR
INJECTDBITERR
DOUTA[15:0]
SBITERR
DBITERR
RDADDRECC[2:0]
IP Symbol
Power Estimation
Datasheet
< Back
Page 3 of 6
Next >
Generate
Cancel
Help


Block Memory Generator
Documents View
IP Symbol
Block Memory Generator
xilinx.com:ip:blk_mem_gen:7.3
Port A
Register Port A Output of Memory Primitives
Register Port A Output of Memory Core
给输出端口增加寄存器，这样可以提供核的性能
但是每增加一个寄存器，就会增加一个输出数据的时钟延时
Register Port A Input of SoftECC logic
Use REGCEA Pin (separate enable pin for Port A output registers)
Pipeline Stages within Mux 0
Mux Size: 1x1
Memory Initialization
数据初始化，使用.coe文件，格式见Shift_Register
Load Init File
Coe File no_coe_file_loaded
Browse
Show
Fill Remaining Memory Locations
没有用.coe文件初始化的，存储器用这个值初始化
Remaining Memory Locations (Hex) 0
ADDRA[2:0]
ENA
REGCEA
RSTA
CLKA
INJECTSBITERR
INJECTDBITERR
DOUTA[15:0]
SBITERR
DBITERR
RDADDRECC[2:0]
IP Symbol
Power Estimation
Datasheet
< Back
Page 4 of 6
Next >
Generate
Cancel
Help





模块说明

ROM your_instance_name (

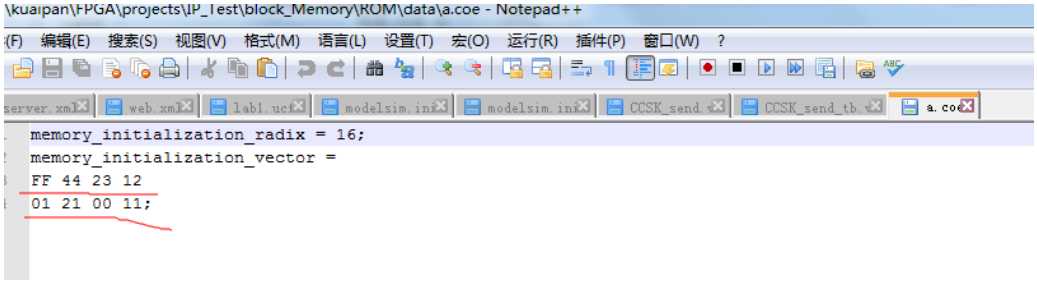
.clka(clka), // input clka ----------a 端口的时钟

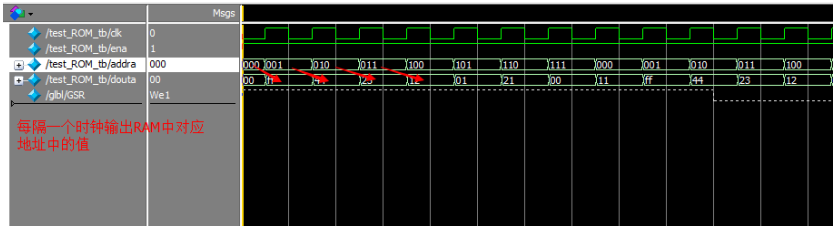
.ena(ena), // input ena ----------时钟使能

.addra(addra), // input [2 : 0] addra----- 地址端口

.douta(douta) // output [7 : 0] douta--- 数据端口

);

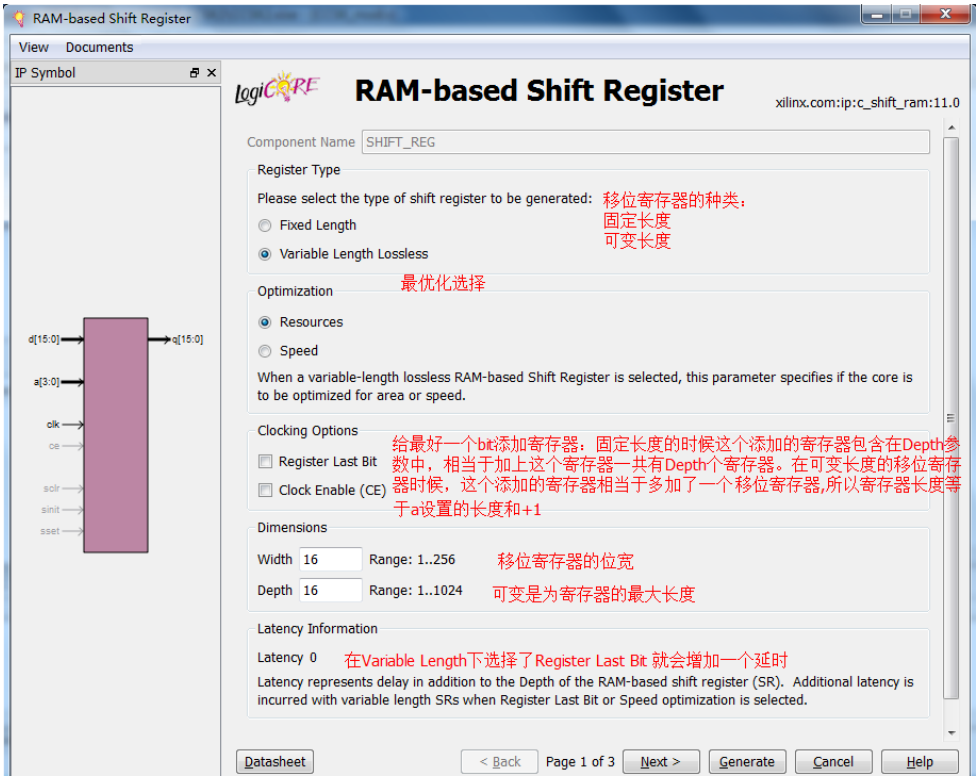


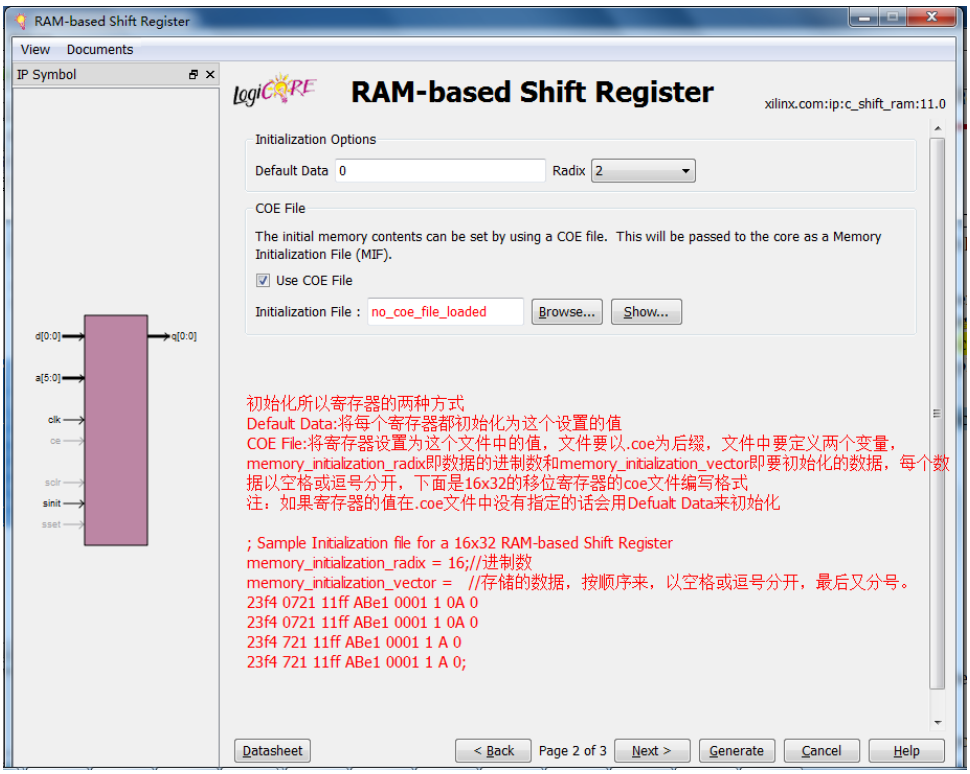


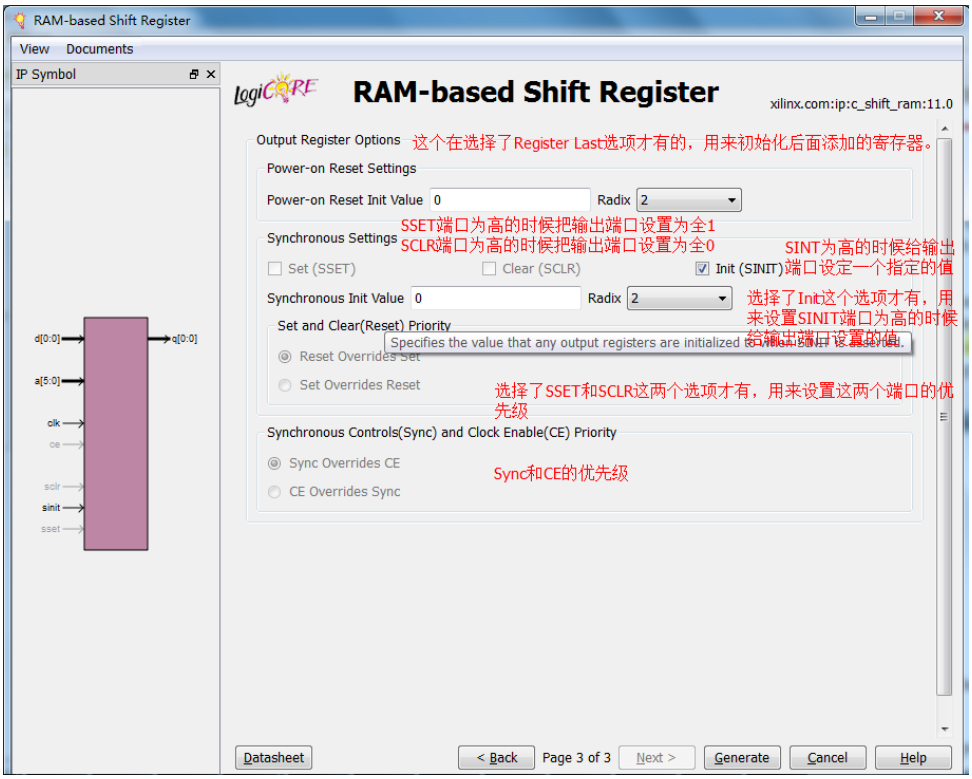
6.Shift-Register

基于 RAM 的移位寄存器提供为非常高效的多比特宽度的移位寄存器，用于像 FIFO 或做延时等的应用。可以生成固定长度的移位寄存器或者可变长度的移位寄存器。

1.配置







2.模块说明

SHIFT_REG your_instance_name (

.a(a), // input [2 : 0] a

.d(d), // input [0 : 0] d

.clk(clk), // input clk

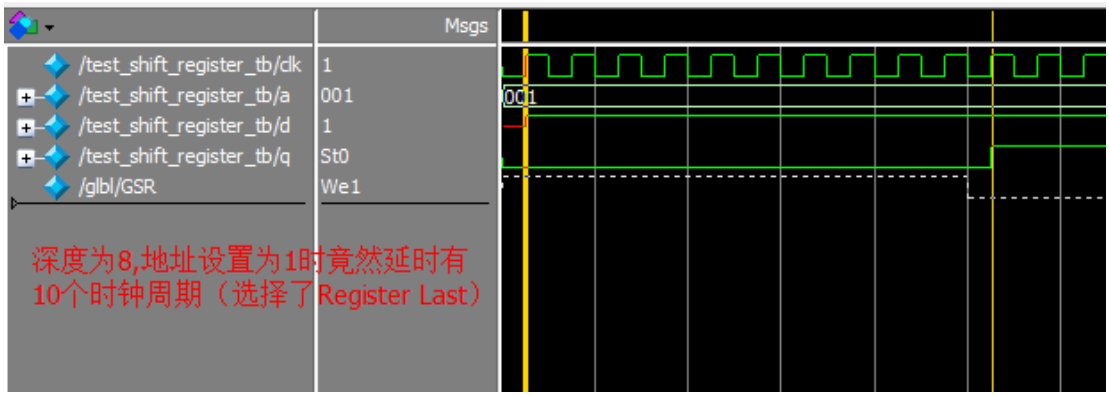
.q(q) // output [0 : 0] q

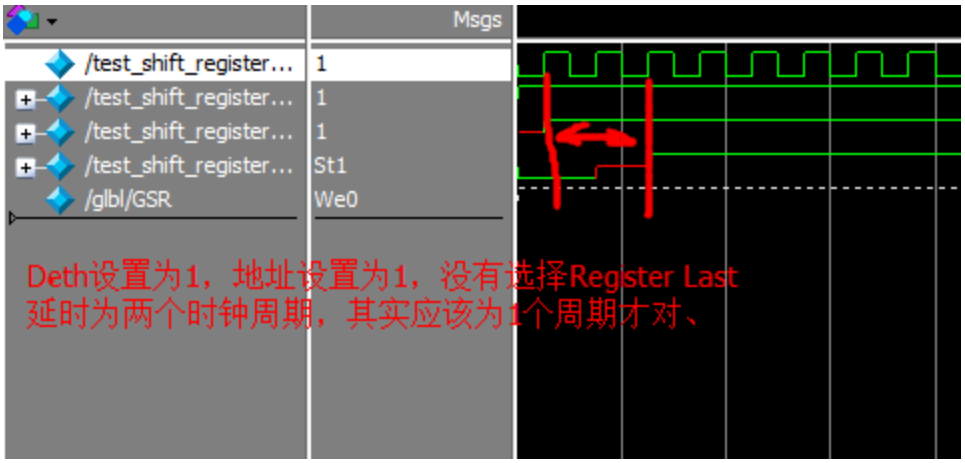
);

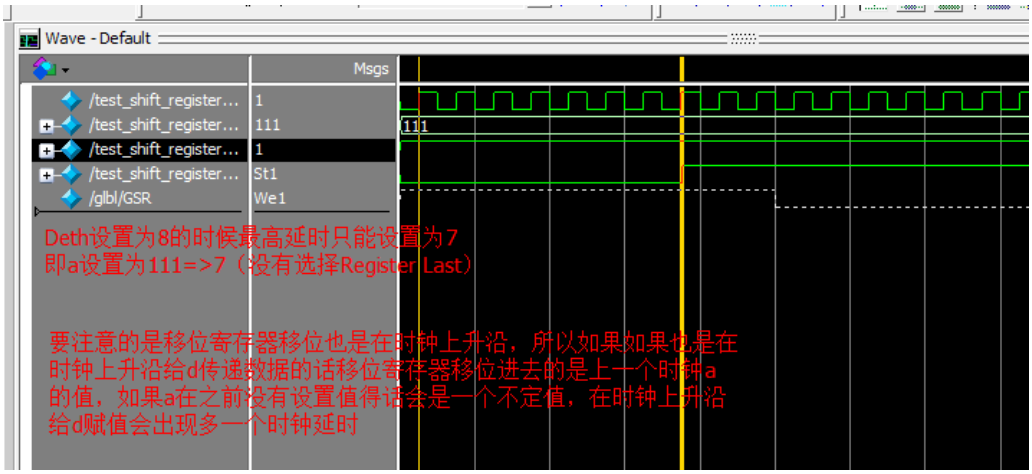
3.变化长度

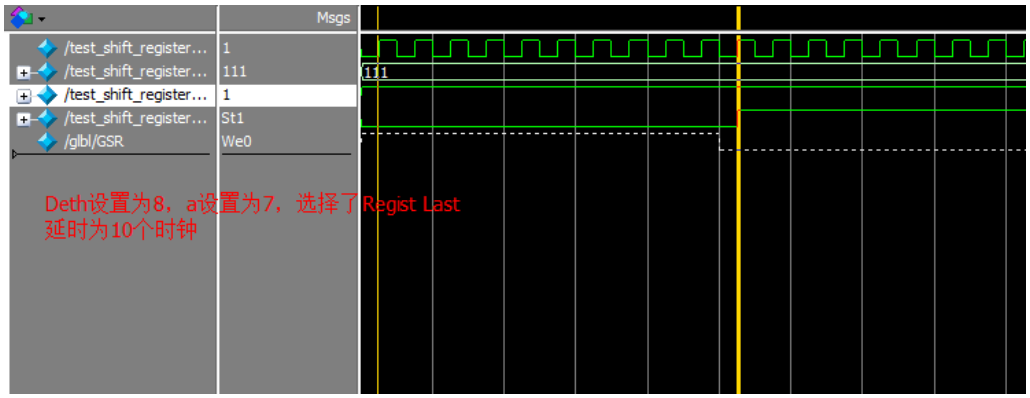
选择了 Regist Last 时不知为什么，不管怎么设置 a 的值延时都变

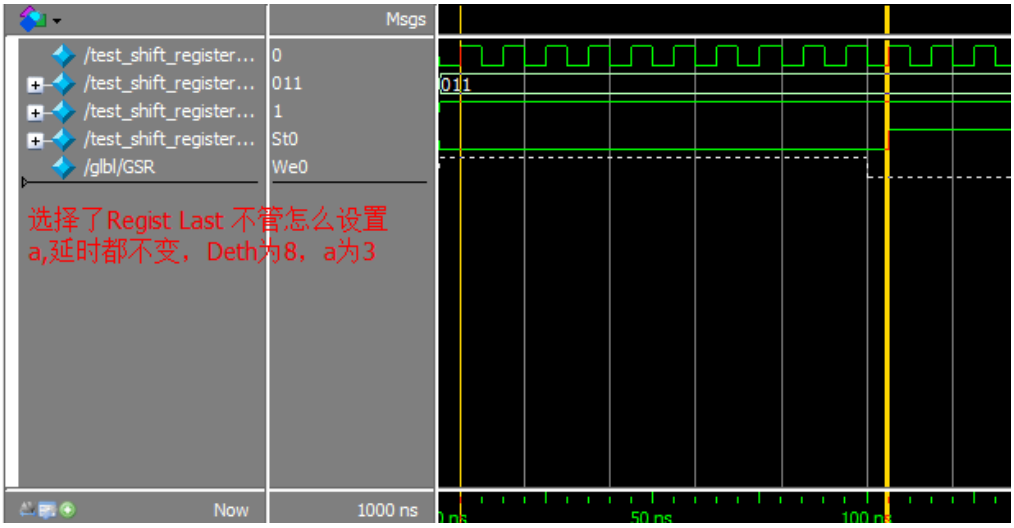
由于在移位寄存器也是在时钟上升沿移位的，如果在时钟上升沿给 d 赋值的话，这个值要在下一个时钟上升沿才能移位到寄存器，所以会多出一个时钟的延时。



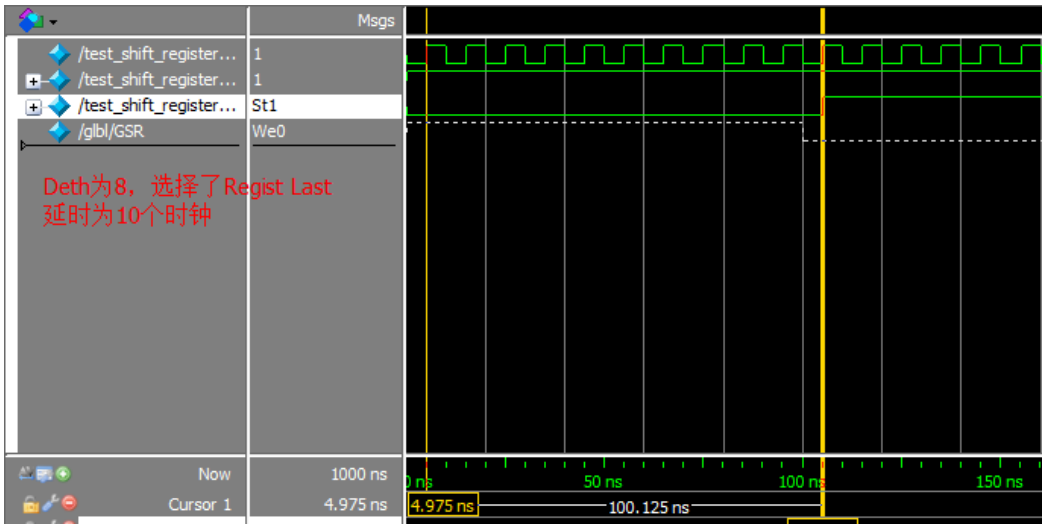


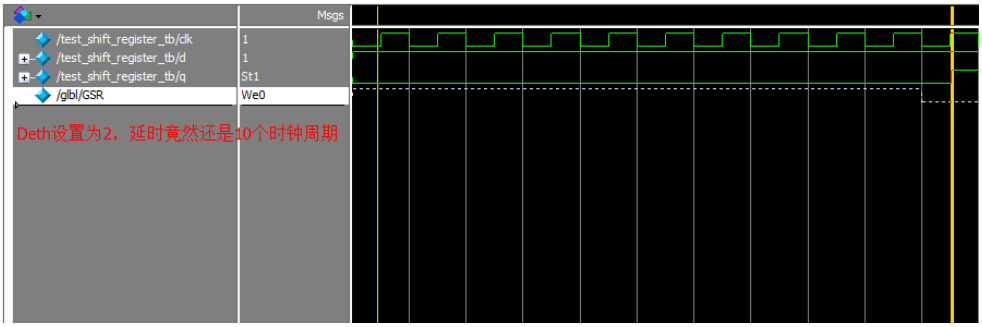

Wave - Default
Msgs
/test_shift_register...
/test_shift_register...
/test_shift_register...
/test_shift_register...
/glbl/GSR
1
111
1
St1
We1
111
Deth设置为8的时候最高延时只能设置为7
即a设置为111=>7（没有选择Register Last）
要注意的是移位寄存器移位也是在时钟上升沿，所以如果如果也是在
时钟上升沿给d传递数据的话移位寄存器移位进去的是上一个时钟a
的值，如果a在之前没有设置值得话会是一个不定值，在时钟上升沿
给d赋值会出现多一个时钟延时


Msgs
/test_shift_register...
/test_shift_register...
/test_shift_register...
/test_shift_register...
/glbl/GSR
1
111
1
St1
We0
111
Deth设置为8，a设置为7，选择了Regist Last
延时为10个时钟


Msgs
/test_shift_register...
/test_shift_register...
/test_shift_register...
/test_shift_register...
/glbl/GSR
0
011
1
St0
We0
011
选择了Regist Last 不管怎么设置
a,延时都不变，Deth为8，a为3
Now
1000 ns
50 ns
100 ns

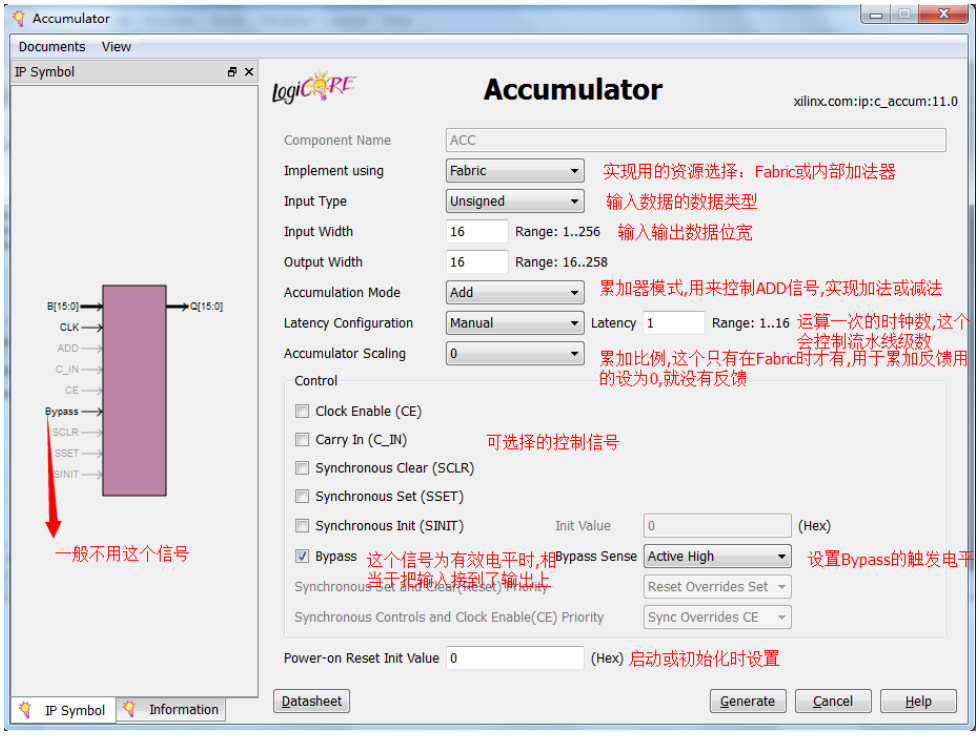
4.固定长度





7.ACC 累加器

1.GUI 设置



上面说一般不用 Bypass 这个信号 ,是不对的 当累加器要重复使用的时候 输出端要么使用 SCLR 清零要么用 Bypass. 以不至于把上一次的结果也累加上，Bypass 最重要的是可以给累加器置初值

2.模块使用说明

ACC your_instance_name (

.b(b), // input [0 : 0] b ----------数据输入端

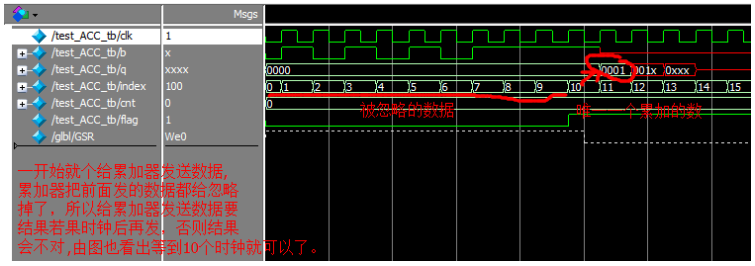
.clk(clk), // input clk ----------时钟

.q(q) // output [3 : 0] q ----------累加结果

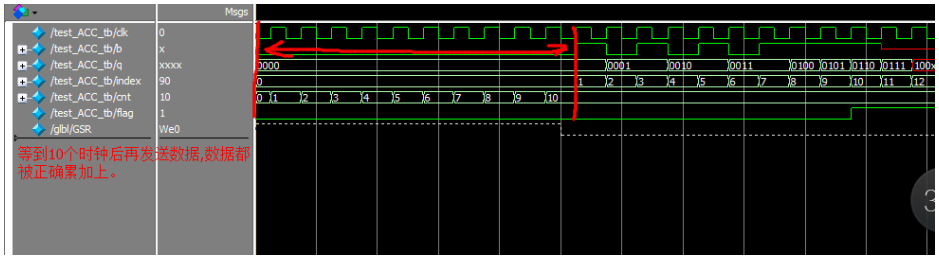
);

3.使用注意事项

没有添加等到时钟，就发送数据



添加等到时钟后发送数据



测试代码

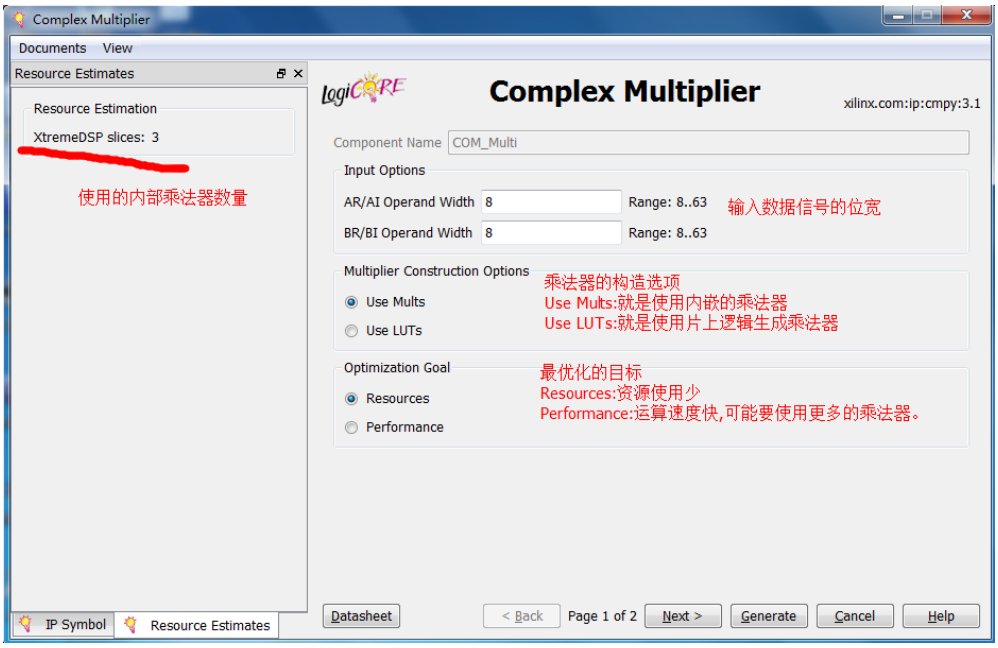
```
47        a[4] = 1;
48        a[5] = 0;
49        a[6] = 1;          //初始化发送数据,寄存器
50        a[7] = 1;
51        a[8] = 1;
52        a[9] = 1;
53     end
54     integer index,cnt;
55     reg flag;             //是否发送完10个数据到累加器的标志，发送完后为1；
56     initial begin
57        b = 0;             //这个要加上，否则累加开始时会得到一个不定值
58        clk = 0;
59        cnt = 0;
60        index = 0;
61        flag = 0;
62        forever #5 clk = ~clk;//产生时钟
63     end                   这里设置的运算周期是1个时钟周期,所以可以每个时钟上升沿都可以发
64     always@(posedge clk) 送数据
65        if(cnt == 10)      //延时10个时钟,因为启动或rst累加器的时候要结果若果时钟后才能送入数据
66           begin
67              b = a[index];//发送数据
68              index = index + 1;
69              if(index == 10)
70                 flag = 1;    //数据发送完成
71           end
72        else
73           if(!flag)
74              cnt = cnt + 1;
75           else
76              cnt = cnt;
```

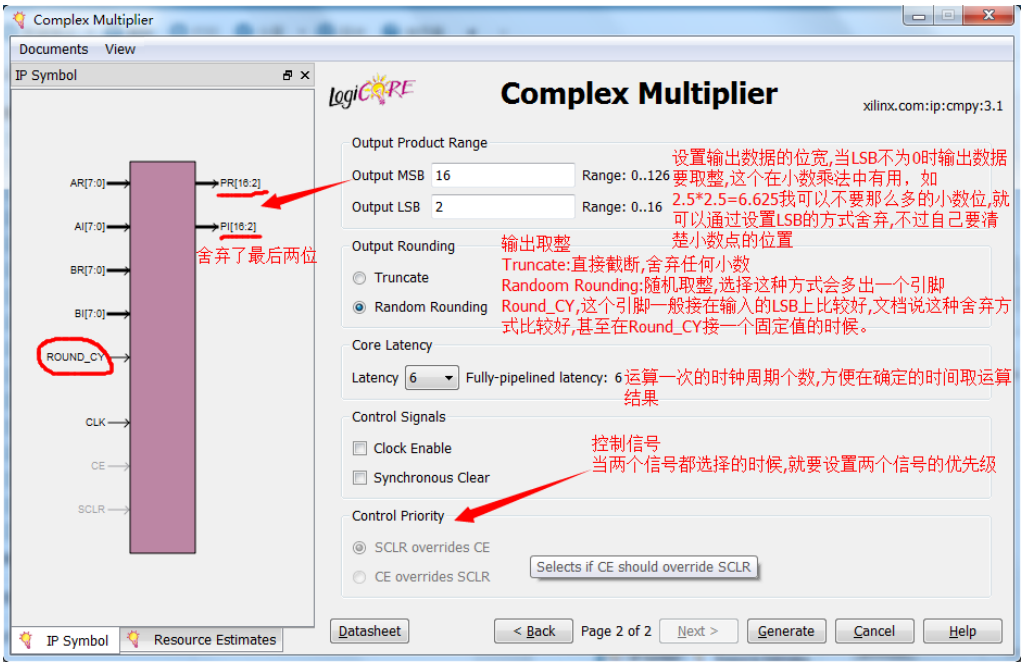
8.复数乘法器

数据用二进制补码格式------>说明运算的数必须是有符号数，如果不是有符号数，运算之前要进行符号扩展

运算的时钟周期可以配置。

1.GUI 设置





COM_Multi YourInstanceName (

.ar(ar), // input [7 : 0] ar---复数输入 A

.ai(ai), // input [7 : 0] ai

.br(br), // input [7 : 0] br--复数输入 B

.bi(bi), // input [7 : 0] bi

.clk(clk), // input clk

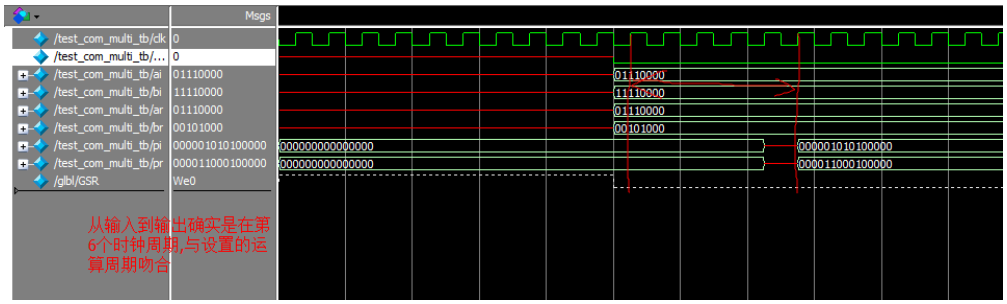
.round_cy(round_cy), // input round_cy---取整随机信号

.pr(pr), // output [14 : 0] pr ---舍弃了最低的两位，这里还有 15 位，最高位是多余的。

.pi(pi)); // output [14 : 0] pi

```
   initial begin
      clk = 0;
      forever #5 clk = ~clk;
   end

   //(01.110000+j01.110000)*(00.101000-j00.0100000)=>(1.75+j1.75)*(0.625-j0.25)=>1.53125+j0.65625
   initial begin

      #100 ai = 8'b01110000;      //又忘了加8'b了!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
      bi = 8'b11110000;           //-0.25,原码10.0100000反码11.101111补码11.110000
      ar = 8'b01110000;
      br = 8'b00101000;        //小数位数为6+6=12位,舍弃两位则为10位,所以后10位为小数[9:0],前面的是整数[14:10],有5位为整数
      round_cy = ai[0];        //两接在最低位上
   end

endmodule
```

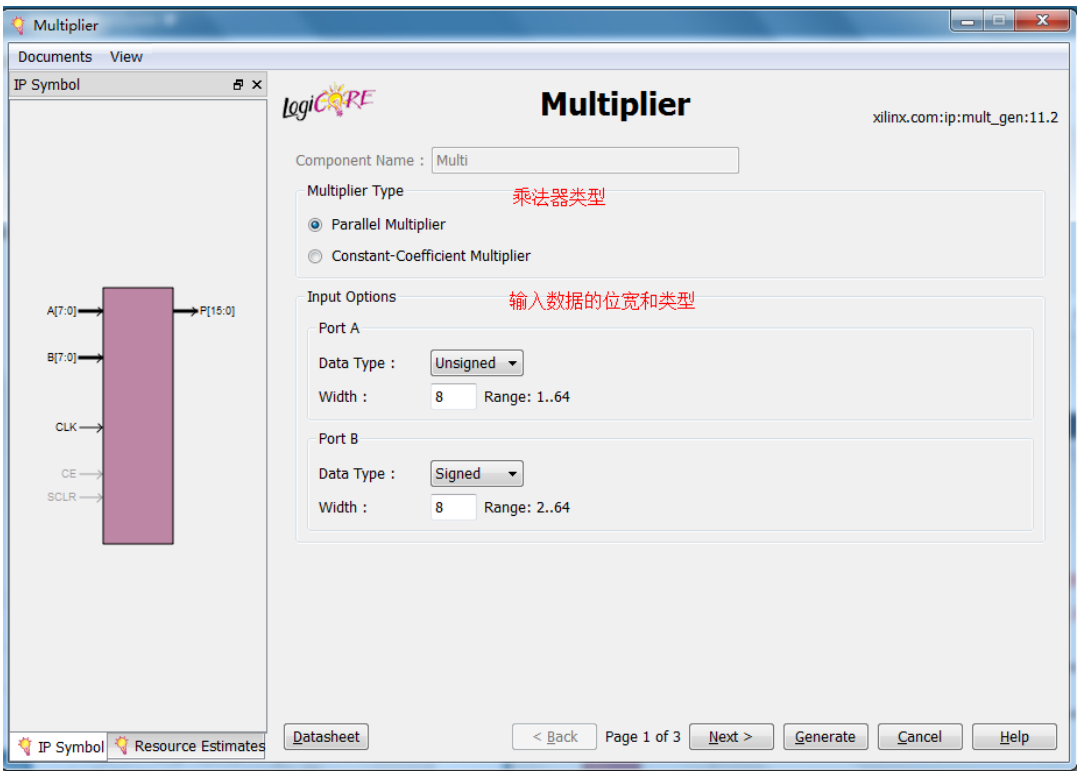
测试用的代码

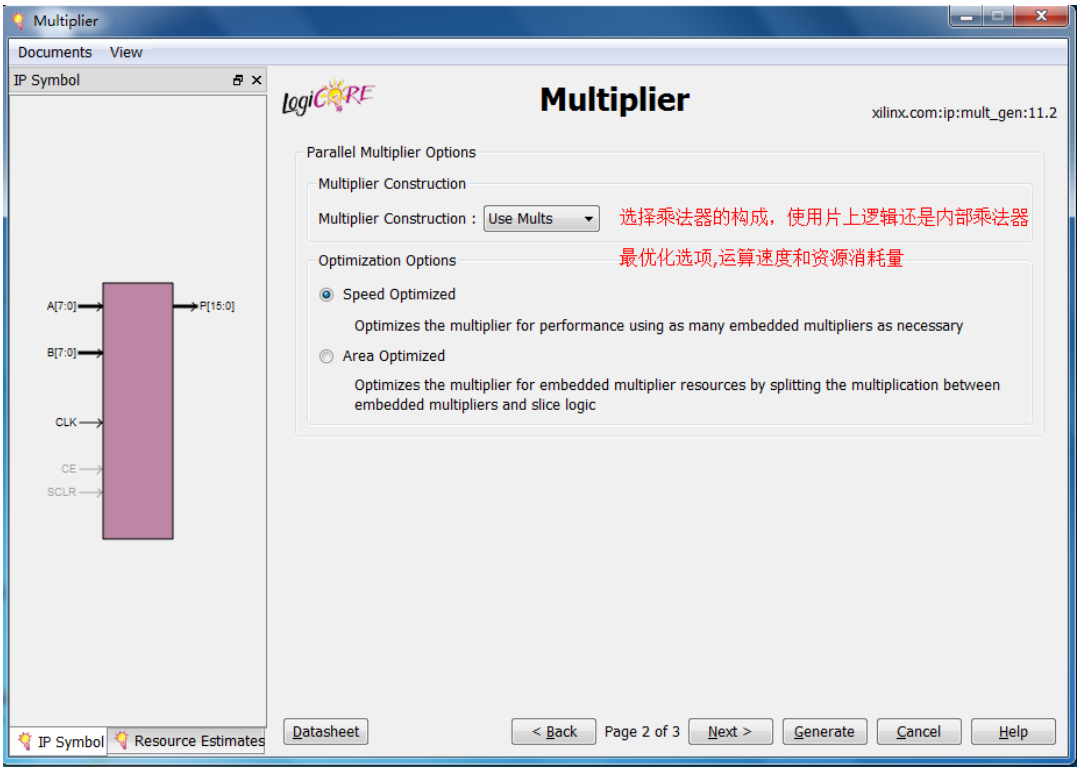


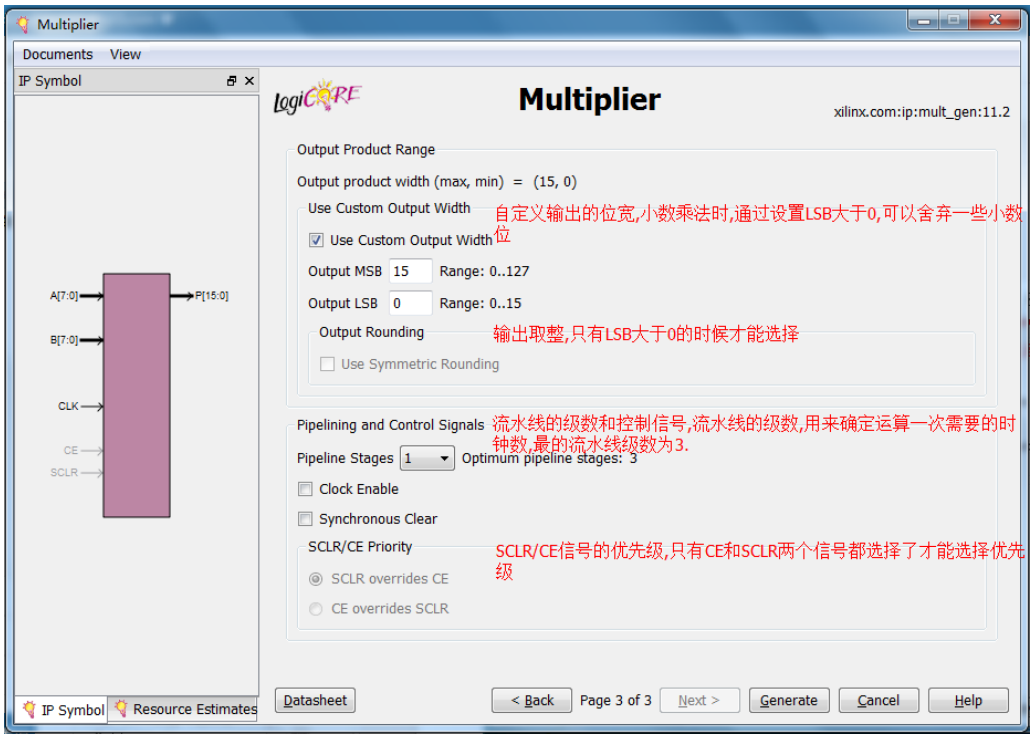
测试得到的波形

9.乘法器

1.GUI 设置


Multiplier
Documents View
IP Symbol
LogiCORE
Multiplier
xilinx.com:ip:mult_gen:11.2
Component Name : Multi
Multiplier Type
乘法器类型
Parallel Multiplier
Constant-Coefficient Multiplier
Input Options
输入数据的位宽和类型
Port A
Data Type : Unsigned
Width : 8 Range: 1..64
Port B
Data Type : Signed
Width : 8 Range: 2..64
A[7:0]
B[7:0]
CLK
CE
SCLR
P[15:0]
IP Symbol
Resource Estimates
Datasheet
< Back
Page 1 of 3
Next >
Generate
Cancel
Help


Multiplier
Documents View
IP Symbol
LogiCORE
Multiplier
xilinx.com:ip:mult_gen:11.2
Parallel Multiplier Options
Multiplier Construction
Multiplier Construction : Use Mults
选择乘法器的构成，使用片上逻辑还是内部乘法器
Optimization Options
最优化选项,运算速度和资源消耗量
Speed Optimized
Optimizes the multiplier for performance using as many embedded multipliers as necessary
Area Optimized
Optimizes the multiplier for embedded multiplier resources by splitting the multiplication between embedded multipliers and slice logic
A[7:0]
B[7:0]
CLK
CE
SCLR
P[15:0]
IP Symbol
Resource Estimates
Datasheet
< Back
Page 2 of 3
Next >
Generate
Cancel
Help



上面的流水线为一级，也可以选择不要流水线，既 0 级这样运算保证会在一个时钟内完成

2.模块使用说明

Multi your_instance_name (

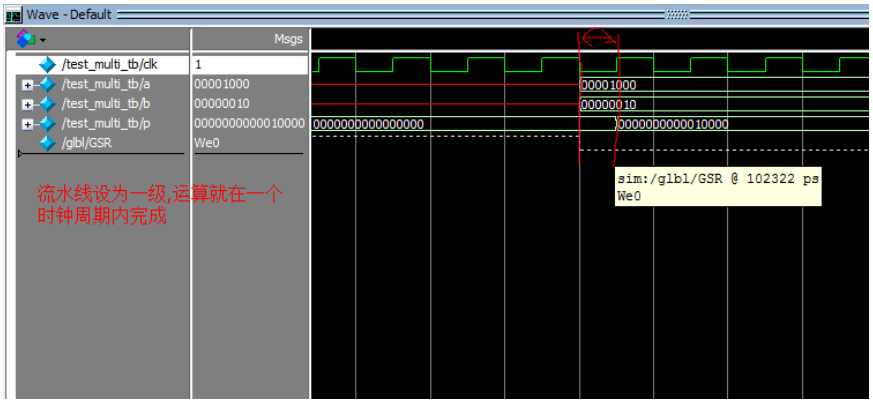
.clk(clk), // input clk ----------时钟

.a(a), // input [7 : 0] a ----------乘数 a

.b(b), // input [7 : 0] b ----------乘数 b

.p(p) // output [15 : 0] p --------结果

);



10.FFT

1.时序

(1).设置 FFT 类型

fwd_inv = 0; //IFFT ，为 1 的时候为 FFT

fwd_inv_we = 1; //使能 fwd_inv

(2).启动 FFT

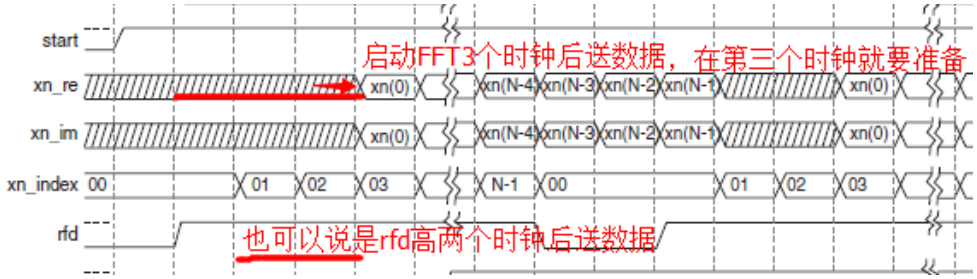
start = 1;

注意：这些变量在声明的时候就要初始化，否则 IP 核运行的时候检测到时未知值的时候就会出错，启动不了

(3).给 FFT 内核送数据，这个最要注意。

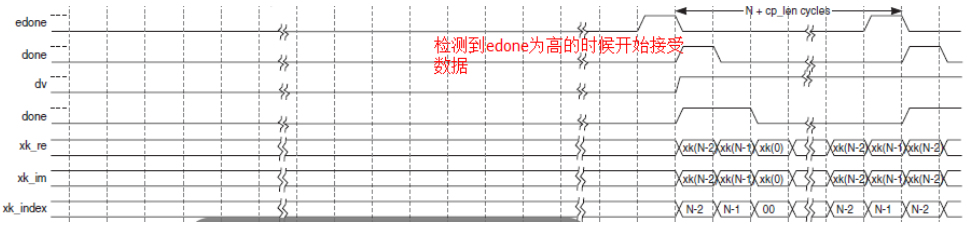
1.启动 FFT 后要延时 3 个时钟才能给内核送数据，这是从时序上要求的---加一个移位寄存器

2.送的数据的个数要和 FFT 内核设置的一样。




```
always@(posedge clk)
if(start ==1) {
if(delayCout==2) //延时 3 个时钟，考虑到阻塞赋值，所以为 2
    writeData;
else
    delayCout ++;
}
always@(posedge clk)
if(start ==1) {
if(delayCout==1) //延时 2 个时钟
    writeData;
else
    delayCout ++;
}
```

（4）数据接收



接收数据开始的时候有一个 cp_len 的长度是不需要的。

可以这样读，在 done 为低，dv 为高的时候读取数据

```
always@(posedge clk)
if(done == 0&&dv==1)
    readData
```

输出 xk_re,xk_im 位宽的说明

(1).选择的数据为 scaled 结构，和浮点结构的　输出位宽=输入位宽

(1).如果选择为 unscaled 结构则　输出位宽=输入位宽 + log2(转换点数) + 1.其中整数位宽+1， 小数位宽+10

(2)对于单精度的数据格式，输出为 32 位的。

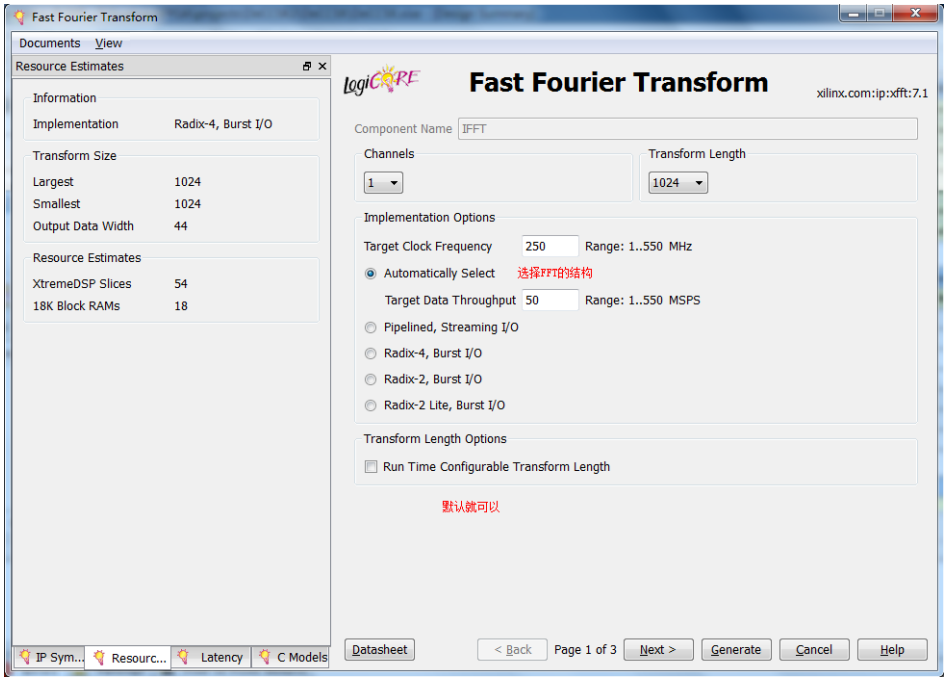
缩小规则

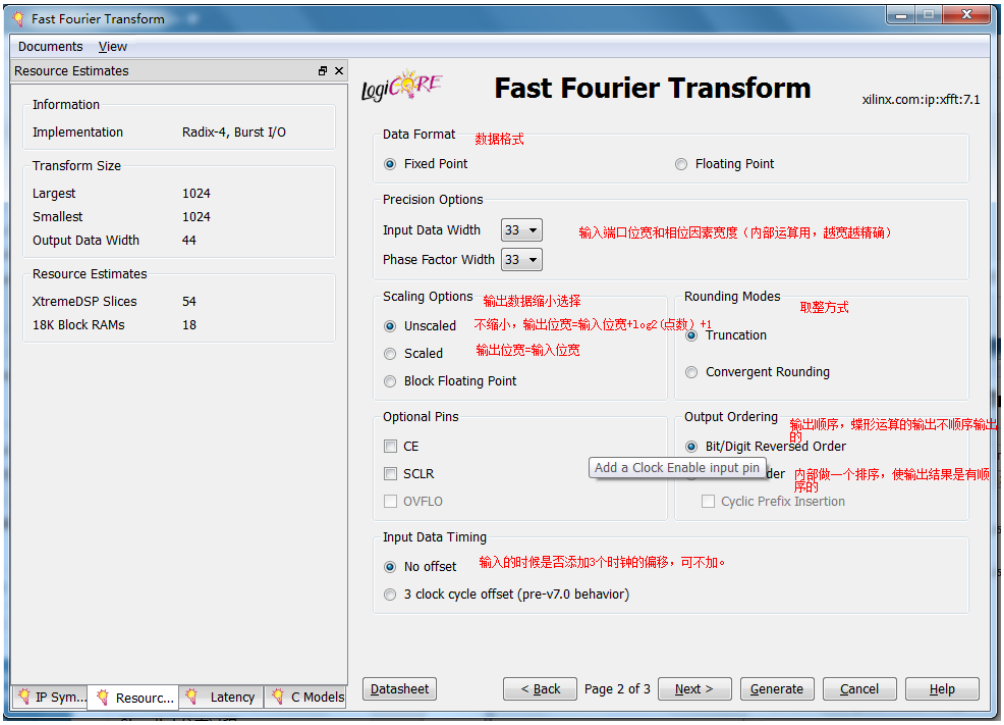
选择了 unscaled 结构，这要配置 SCALE_SCH 使各级蝶形运算的结果进行相应的移位。

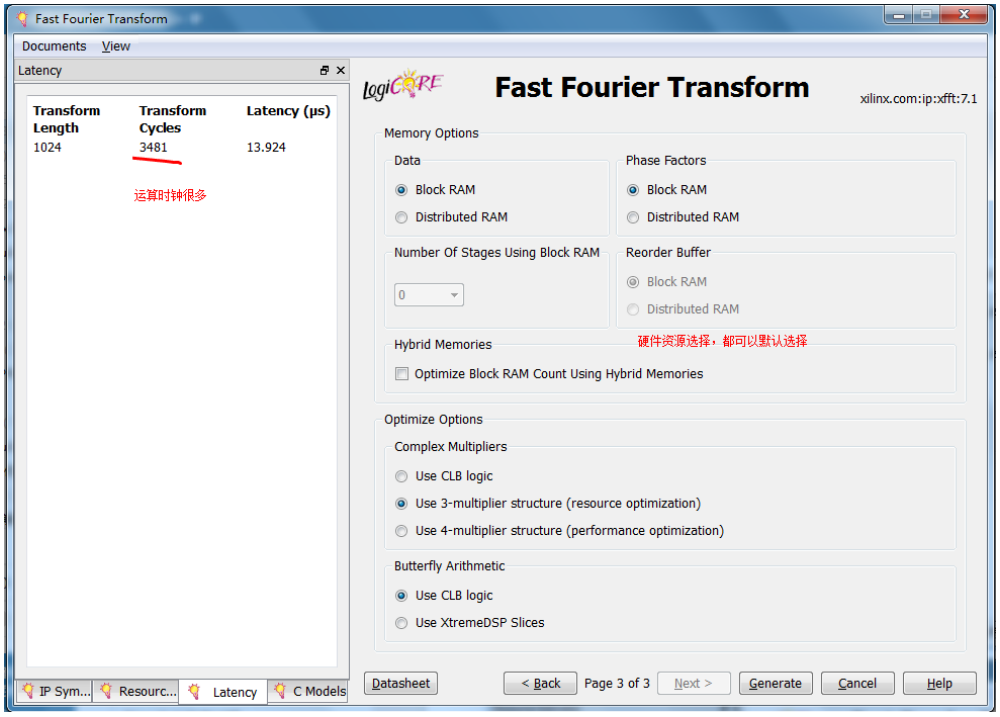
对于基 4，总共的蝶形级数为 log4(转换点数），

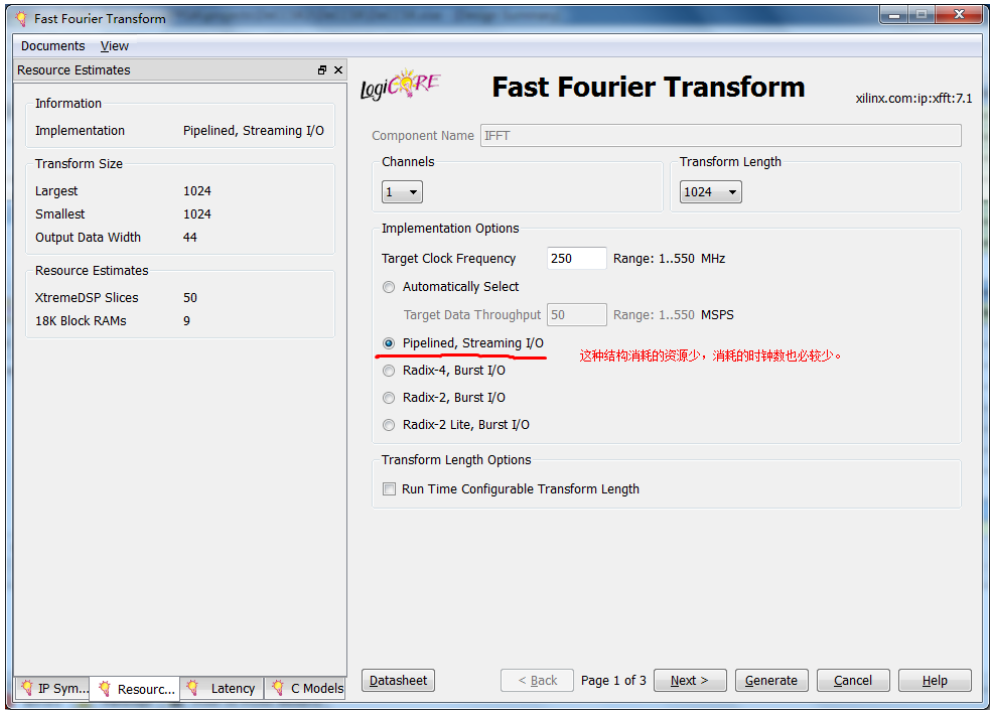
对应基 2，总共的蝶形级数为 log2(转换点数）

for Radix-4, when N = 1024, [01 10 00 11 10] translates to a right shift by 2 for stage 0, shift by 3 for stage 1, no shift for stage 3, a shift of 2 in stage 3, and a shift of 1 for stage 4 (there are log4(1024) = 5 Radix-4 stages).









11.FIFO

1.fifo 分为两种，一中是输入输出时钟相同的 fifo 另一种是输入输出时钟不相同的。

2.fifo 是没有地址的，是通过 full 和 empty 信号来判断是否可写和可读。

3.用 full 和 empty 来判断 fifo 中存储的状态有些太极端了，可以通过 read_count 和 write_count 来判断存储的状态

write_count:为 fifo 中存储的余量。