和循环一样，在C程序中，数组也是一种直观且有用的软件结构，然而在HDL中，数组的访问经常成为性能上的瓶颈，当然像HLS中对循环有许多优化策略一样，在不改变HLS的软件结构下，HLS对数组也有一系列的优化方案。

首先我们得先了解C语言里面的数组经过HLS综合后，在HDL中的表现形式。通常情况下，数组一般被综合为存储器结构，比如RAM、ROM或者FIFO等，而设计内部的中间数组变量会根据设置被综合成block RAM，LUTRAM或者register（寄存器），不同的存储方式对“时间和面积”都有不同的影响。

为什么说数组的综合会成系统的性能的瓶颈呢？下面就举个例子说明下：

#include array\_mem\_bottleneck.h

dout\_t array\_mem\_bottleneck(din\_t mem[N]) {

dout\_t sum=0;

int i;

SUM\_LOOP:for(i=2;i<N;++i)

sum += mem[i] + mem[i-1] + mem[i-2];

return sum;

}

SUM\_LOOP循环是为了在一个循环周期内加上数组mem[N]相邻的三个元素，看上去这个加法可以在一个时钟周期内完成，但实际上如果我们约束这个循环一个时钟周期完成的时候系统会提示无法进行，因为通常情况下数组会被综合成RAM，假如设置为单端口RAM，则在同一个时钟周期，只能对数组进行读或者写，所以在此men[N]三个相邻的元素至少得需要三个时钟周期读完才能进行加和；即使可以设置成双端口RAM，也只是同时可以读数据和写数据，也完不成一个时钟周期内的加和运算。

当然，当我们分析上面例子中的C程序时，就发现其实在每个循环都会对men[N]中的两个元素进行重复读取，例如当N=2时，每个循环需要读取men[0],men[1]，men[2],而在下一个循环N=3时，需要读取men[1],men[2],men[3],这样会造成men[1],men[2]重复读取而影响综合效率，如果程序改写为下列形式，则一个时钟周期内仅需读取一个数组值。

#include array\_mem\_perform.h

dout\_t array\_mem\_perform(din\_t mem[N]) {

din\_t tmp0, tmp1, tmp2;

dout\_t sum=0;

int i;

tmp0 = mem[0];

tmp1 = mem[1];

SUM\_LOOP:for (i = 2; i < N; i++) {

tmp2 = mem[i];

sum += tmp2 + tmp1 + tmp0;

tmp0 = tmp1;

tmp1 = tmp2;

}

return sum;

}

当然并不是所有的程序都像上述例子那样通过修改C代码实现综合的优化，更多的情况是我们希望不改变C语言结构就能得到优化，在这里HLS也提供的方案是允许通过设置对数组进行“分割”（Partition）和“重组”(Reshape)。顾名思义，数组分割就是将一个多维数多元素的数组“分割”成多个相对独立的模块，当设置将数组分割成多个模块(BLOCK)时，HLS综合成RTL RAM，当将数组分割成单独的元素，则每个元素都以寄存器形式存储，这样就可以缩短数组的读取时间了。

下面将说明下数组分割

请看下面程序，

**#include** "array\_RAM.h"

**void** **array\_RAM** (dout\_t d\_o[4], din\_t d\_i[4], didx\_t idx[4]) {

**int** i;

For\_Loop: **for** (i=0;i<4;i++) {

d\_o[i] = d\_i[idx[i]];

}

}

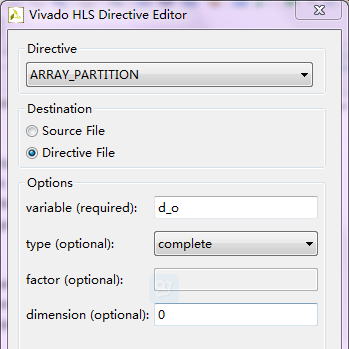
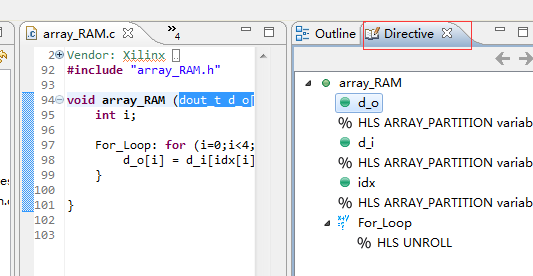
我们对数组d\_o[4],d\_i[4],idx[4]进行分割，分割方式如下

在directive栏双击数组名称，会出现命令设置窗口，在directive下拉栏中选择arrry\_partition，针对options窗口中进行解释

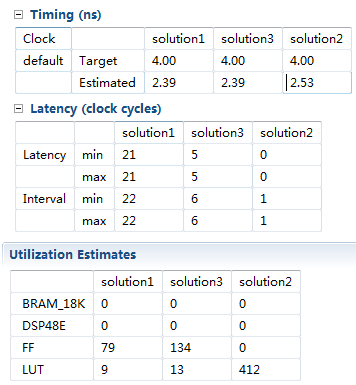
首先是type，其指的是对数组处理的不同结构，其类型有block，cyclic，complete三种，其分别代表的意思如下图



而数组可以分割成任何的维数，也就是设置选项里面的dimension，如果此处没有指定维数的话，则将其假定为0，其表示的是所有的维数即所有元素。假设有个三维数组需要被分割，my\_array[10][6][4]，如果分割维数为3，则将原数组分割成my\_array\_0[10][6]/ my\_array\_1[10][6], my\_array\_2[10][6], my\_array\_3[10][6]四个数组，如果分割维数为1，则将原数组分割成my\_array\_0[6][4], my\_array\_1[6][4]…my\_array\_9[6][4],等10个数组，如果分割维数为0，则原数组就会分为240个单独的元素。



在本例子里对数组的处理是type为complete，维数为0，也就是全部分解为寄存器存储，最终综合的结果如下图所示：



其中solution1为对原程序不做任何约束优化处理，solution3为对程序中的循环进行展开处理，solution2则是在循环展开的同时对数组进行上述的处理，由综合后的时间和面积报告可得，数组的处理后消耗的主要是LUT资源，即存储资源，存储为寄存器方式能大大降低程序的延时，而当数组未展开的情况下，则需要大量的触发器资源，这在设计中资源利用上是需要注意的。

上面说了数组不仅可以综合成RAM还能综合成ROM，那到底什么情况下会综合成ROM，综合成ROM有什么好处呢？

前面说了，数组综合成RAM可以读数据也可以写数据，但有些时候我们初始化的数组是固定的，不再改变，也就是我们只需要读这个数组而不需要写这个数组，这个时候我们就可以选择将该数组综合成ROM，这时候就涉及到数组的初始化了，对同一个数组的初始化典型的有以下两种方式。

int coeff[8] = {-2, 8, -4, 10, 14, 10, -4, 8, -2};

static int coeff[8] = {-2, 8, -4, 10, 14, 10, -4, 8, -2};

在有些情况下，这两种方式数组初始化的最终得到程序功能是一样的，但是其在HLS综合后的面积和时间占用会大大不同，第一次情况下的数组初始化在数组每次调用时，都会重新赋值给RAM，对于综合成单端口RAM情况下，都需要8个时钟周期才能将数组值写入RAM，在这八个时钟周期之间，针对这个数组的其他运算都无法进行，这样就大大增加程序的时延，但是在第二种情况下，添加static修饰后，数组会在首次被调用的写入RAM，而在之后的调用过程中，数组都会记住上一次的值，直接用上次存储数组的RAM，而不需要重新消耗一个RAM往里面写值，这样就大大降低了程序的时延。

特别需要注意的是有的时候看起来一个函数是必须可写的，如下面的例子

**#include** "array\_ROM.h"

dout\_t **array\_ROM**(din1\_t inval, din2\_t idx)

{

din1\_t lookup\_table[256];

dint\_t i;

**for** (i = 0; i < 256; i++) {

lookup\_table[i] = 256 \* (i - 128);

}

**return** (dout\_t)inval \* (dout\_t)lookup\_table[idx];

}

数组lookup\_table[256]首先是由式子得到的，所以必须是可写的，但是经过HLS综合后，lookup\_table[256]却被综合成ROM，当你强制令lookup\_table[256]综合成RAM，却会出现以下信息

@W [RTMG-274] Memory 'array\_ROM\_lookup\_table' is read-only, switch it to a ROM.

这就是Vivado HLS自身优化的地方，在一些情况下能判断数组是否后期写入数据还是只是固定用数组里面的数据，像那面那个例子HLS就能提前离线计算出lookup\_table[256]的值，而不需要浪费FPGA自身的乘法器进行在线计算数组的值。还有种情况就是如果在算法中出现类似sin等三角函数赋值的时候，HLS也是先离线求出确定的值，然后再给数组赋值，这样就能大大降低数组赋值时间，当然这些都是vivado HLS自身的功能，我们不需要额外的设置优化。

其实，总体来说，对数组上的优化，HLS工具自身已经具有强大的功能，自身能确定最优的硬件方案，而可供我们优化的地方就在于数组的分割的选择了。

最后，针对上次的循环展开里面数组读取的问题，在这里就可以解释一下了。在数组的乘法中