上一篇文章中介绍了HLS中关于C语言一些关键属性，然后重点介绍了数据类型在HLS中的应用，这一篇文章讲下C中的循环在HLS处理的过程及其优化。

众所周知，在C语言中，循环是算法最直观和简明的表达方式，当然在HLS中，循环也能很好的被综合成硬件语言。当然C中的循环能被HLS做出各种处理，最终优化的结果也是不一样的。

循环的优化主要包裹循环展开，循环平坦和循环合并。当然这些优化的方法只能应用在一些场合，下面来举例具体说明下。

首先是循环展开，在默认情况下，HLS不对循环进行展开的，也就是说：

1. 在一个状态内，实现对每个C循环的迭代操作；
2. 在相同的资源内，实现对每个C循环的迭代操作。

显然，这种情况下，程序综合出来的硬件代码的延迟和吞吐量都是最低的。比如说对于下面的代码。

Void loop(…)

{

Add: for(i=3;i>=0;i++)

{

b=a[i]+b;

}

}

如果不针对此循环进行展开，则运行完这个加法至少需要4个周期，如果进行循环展开，则最快只需要一个周期，当然这样会浪费更多的硬件资源。这就需要在设计中自我选择了。

当然并不是所有的循环都能进行循环展开，对于变化的函数边界来说，由于对工具来说不知道有多个循环的本体，故无法做出循环展开操作。例如：

**#include** "ap\_cint.h"

**#define** N 32

**typedef** int8 din\_t;

**typedef** int13 dout\_t;

**typedef** uint5 dsel\_t;

dout\_t **code028**(din\_t A[N], dsel\_t width) {

dout\_t out\_accum=0;

dsel\_t x;

LOOP\_X:**for** (x=0;x<width; x++) {

out\_accum += A[x];

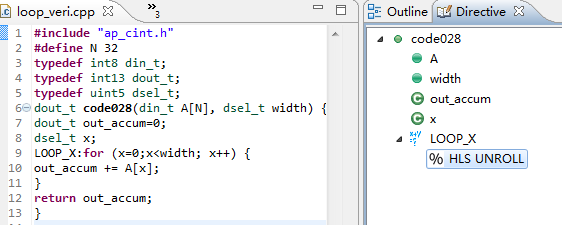
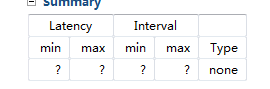
}

**return** out\_accum;

}

由于loop\_x的边界width是函数的参数，如果你对循环进行强制展开，综合报告会显示

@W [XFORM-503] Cannot unroll loop 'LOOP\_X' (loop\_veri.cpp:9) in function 'code028': cannot completely unroll a loop with a variable trip count.此函数的综合后的综合报告会显示 ，其延迟和间隔均为？也是因为这里的原因。

前面介绍了循环展开是在牺牲面积的前提下提升速度，而在循环的优化中还有一种就是设置循环流水线，其在某些场合能在面积和速度上作出最好的折衷。如下面的例子

**#include** "loop\_pipeline.h"

dout\_t **loop\_pipeline**(din\_t A[N]) {

**int** i,j;

**static** dout\_t acc;

LOOP\_I:**for**(i=0; i < 20; i++){

LOOP\_J: **for**(j=0; j < 20; j++){

acc += A[j] \* i;

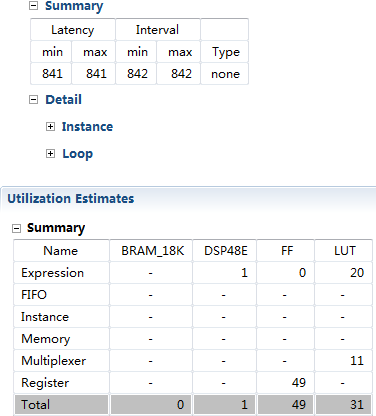
}

}

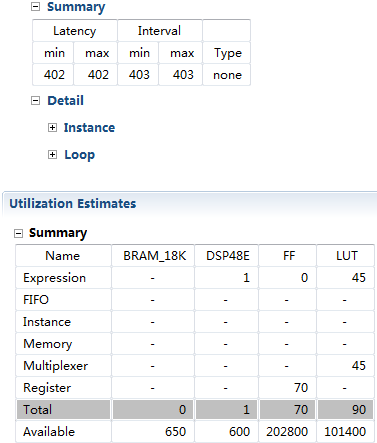
**return** acc;

}

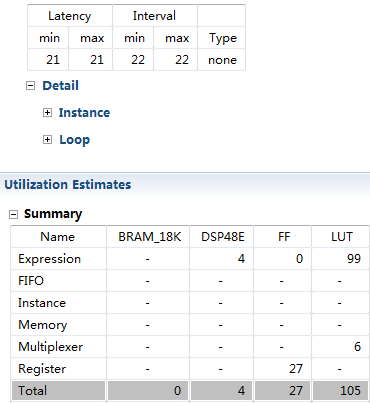
如果循环展开和循环流水线都不设置的话，其综合报告为

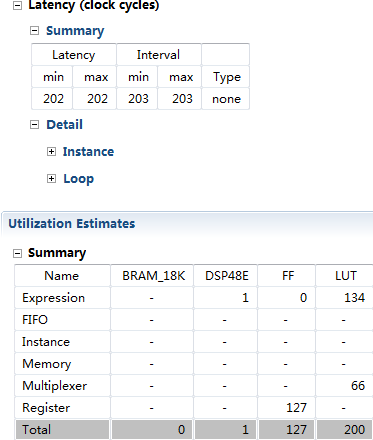


这里有两重循环，在这里可以选择设置循环流水线

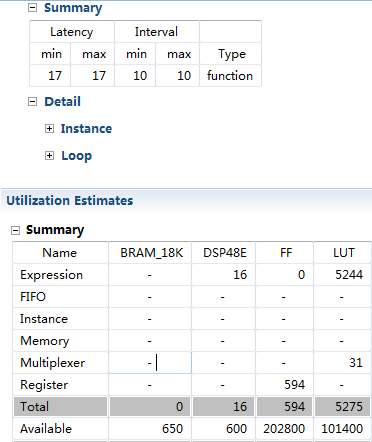
如果将里层的LOOP\_J进行循环流水线，也就是加法和乘法同时进行，这将延迟会缩小一半，报告如图，

如果将外层的LOOP\_I进行循环流水线，这时HLS就会将里层的循环进行自动循环展开，按道理说这里的延迟应该差不多是20左右，但是综合出来的结果却是相对于里层展开延迟的一半？主要是因为数组A[]综合出来的存储方式，在默认情况下A[]被综合成双端口RAM，所以A[20]至少需要10个时钟周期才能完成，所以这样算下来延迟在200多，如果更改A[]的存储方式为寄存器方式，则综合报告为

具体数组的存储方式将在下篇说明，



还有的一种就是对整个函数进行循环流水线，也就是将LOOP\_J和LOOP\_I同时都展开，然后进行流水线运算，此时对延时影响大的是程序中数组A[]的读取，这点将在后期C中数组在HLS综合情况一点说明。



所以从上面的例子可以看出循环的不同流水线处理对循环的优化影响是很大的，当然在优化的过程中还需要多关注资源的占用情况，特别是乘法多的地方会占用大量的乘法器（DSP48E），而一般FPGA资源中乘法器模块数量相对都少，所以选择不同优化方法对乘法器占用情况是不一样的。

循环平坦化

在C语言中，有时候我们的设计都是循环一层套一层，这对于C来说当然是看起来有条理和易于理解，但在Vivado HLS综合时，其逻辑会显得比较复杂，综合出来的代码延时也会比较长，当然对于一些“完美循环”和“半完美循环”，HLS可以自动地将嵌套的循环“平坦化”处理，这就和代码的书写有关系了。其中

完美循环：（1）只有最里面的循环有循环体；（2）在循环描述中没有指定的逻辑；（3）循环的边界是常数。

半完美循环：（1）只有最里面的循环有循环体；（2）在循环描述中没有指定的逻辑；（3）循环的边界是变量。

对于完美循环平坦化处理如图所示：

for(i=3;i>0;i--)

for(j=5;j>0;j++)

{

[Loop body]

}

转为 for(k=15;k>0;k--){

[Loop body]

}

下面就一个例子说明下循环平坦化的作用。

**#include** "loop\_imperfect.h"

**void** **loop\_imperfect**(din\_t A[N], dout\_t B[N]) {

**int** i,j;

dint\_t acc;

LOOP\_I:**for**(i=0; i < 20; i++){

acc = 0;

LOOP\_J: **for**(j=0; j < 20; j++){

acc += A[i] \* j;

}

B[i] = acc / 20;

}

}

对于这个实例，官方的技术文档显示的是由于存在，无法进行平坦化，里层的LOOP\_J需要两个周期输出结果，如果进行流水线处理，然后在之后的时间只需要1个周期输出结果，这样20个循环需要21个时钟周期完成20个循环的运算，而且加上外层循环整个函数需要480个时钟周期完成运算。

官方然后将此程序进行了改写，改成完美的嵌套循环。

**#include** "loop\_imperfect.h"

**void** **loop\_imperfect**(din\_t A[N], dout\_t B[N]) {

**int** i,j;

dint\_t acc;

LOOP\_I:**for**(i=0; i < 20; i++){

//acc = 0;

LOOP\_J: **for**(j=0; j < 20; j++){

**if**(j==0) acc = 0;

acc += A[i] \* j;

**if**(j==19) B[i] = acc / 20;

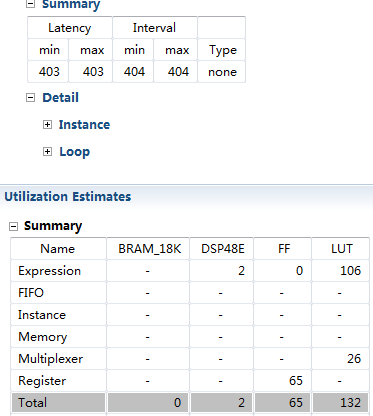
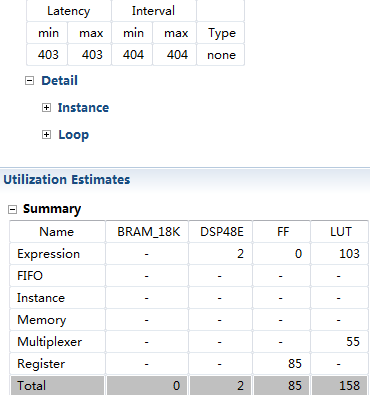
}

//B[i] = acc / 20;

}

}

然后将LOOP\_J进行流水线处理，得到的结果是延迟为401个时钟周期，然后在实际实验中，按照这样的方式进行试验会发现试验结果



除了资源上的占用减少了，延迟并未减少，并且在综合报告上，HLS对于前者不完美的循环也能平坦化处理，所以这里是有疑惑的地方？

循环合并

当遇到多个循环时，在一定情况下，Vivado hls可以自动地合并循环。与手工修改代码相比，这是一种比较快的方法，通过就循环合并，允许探索更加高效的实现结构。当然循环合并的规则如下：

1. 如果循环边界都是变量，则边界的值应该相同；
2. 如果循环边界上常数，最大边界值用作合并循环的边界；
3. 不能合并带有可变边界和常数边界的循环；
4. 对于所合并的循环之间的代码，不能破坏代码的正确执行；
5. 总是顺序执行对FIFO或者FIFO接口的读取操作。

但是对于不能合并的循环，比如下例

**#include** "loop\_sequential.h"

**void** **loop\_sequential**(din\_t A[N], din\_t B[N], dout\_t X[N], dout\_t Y[N], dsel\_t xlimit, dsel\_t ylimit) {

dout\_t X\_accum=0;

dout\_t Y\_accum=0;

**int** i,j;

SUM\_X:**for** (i=0;i<xlimit; i++) {

X\_accum += A[i];

X[i] = X\_accum;

}

SUM\_Y:**for** (i=0;i<ylimit; i++) {

Y\_accum += B[i];

Y[i] = Y\_accum;

}

}

式中，因为SUM\_X和SUM\_Y有不同的边界，所以他们并不能合并，但是，这两个循环之间并没有联系，也就是说SUM\_Y和SUM\_X可以同时执行，而不需要按照C语言顺序SUM\_Y在SUM\_X之后执行，但是如果将这个程序进行综合，HLS并不能解析到这层关系，造成延迟比较大。所以这个时候如果我们对程序进行修改。

**#include** "loop\_functions.h"

**void** **sub\_func**(din\_t I[N], dout\_t O[N], dsel\_t limit) {

**int** i;

dout\_t accum=0;

SUM:**for** (i=0;i<limit; i++) {

accum += I[i];

O[i] = accum;

}

}

**void** **loop\_functions**(din\_t A[N], din\_t B[N], dout\_t X[N], dout\_t Y[N], dsel\_t xlimit, dsel\_t ylimit) {

dout\_t X\_accum=0;

dout\_t Y\_accum=0;

**int** i,j;

sub\_func(A,X,xlimit);

sub\_func(B,Y,ylimit);

}

在这个程序中，把循环当做一个函数，然后在主函数内分别调用，这个时候HLS就能自动将分别调用的两个循环并行执行，这样的话在综合后其延迟是第一个的一半。