HLS #1

HLS（High Level Synthesis），高层次综合工具，可以使用户通过添加directives（制导语句）和constrains（约束），将C/C++/System C代码直接转为FPGA RTL（Verilog，VHDL，SystemC）代码

Xilinx最新的HLS设计工具已经从Vivado HLS改为Vitis HLS

Vitis HLS支持将C、C++和OpenCL函数硬连线到器件逻辑互连结构和RAM/DSP块上

制导语句和约束

制导语句（directives、指导语句），依赖于源程序，开发中会在源程序中加入一些类似于注释的语句，称为制导语句。特别的在C语言中，可以使用#pragma来标记制导语句。常见的标准为OpenMP，较新的还有OpenACC、OpenHMPP。

在适合并行的地方加入制导语句，可以提高计算设备的效能比。



约束（constrains），根据目标设备的资源限制，设置合适的资源约束，以限制资源使用量并优化性能。

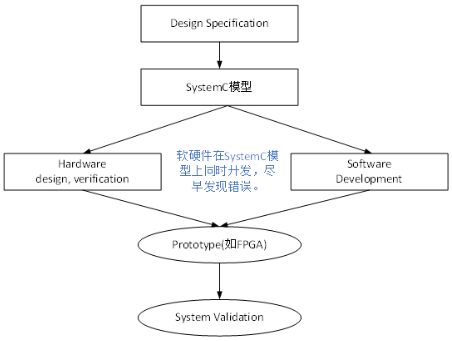
SystemC

系统建模语言，基于C++开发的开源library，理论上和常见的boost库没有任何差别。只要环境支持C++，就可以进行SystemC建模和仿真。可以让软硬件并行开发，加快产品面世

* 硬件架构探索，建立算法、性能模型；
* 验证工程师作为参考模型（通过DPI接口调用）；
* 设计工程师将其作为design spec，设计RTL；
* 软件工程师作为软件开发的硬件模型；
* 使多种提前测试成为可能；

在软硬件设计之前先开发System C模型（如TLM模型），硬件部门将模型转化为RTL（寄存器传输级），软件部门在此模型上开发软件，软硬件的任何错误都能被尽早发现。

*TLM模型在各组件之间建立一个专用通道，让信息只在这个通道里流动，因此避免了全局变量，public成员变量和config机制通讯时带来的不便*



<https://blog.csdn.net/gsjthxy/article/details/126099348>

SystemC框架图，底层是C++及其标准库，也可包括其他的C++库，如boost库。

SystemC library包含多种功能，最核心的有如下几项：

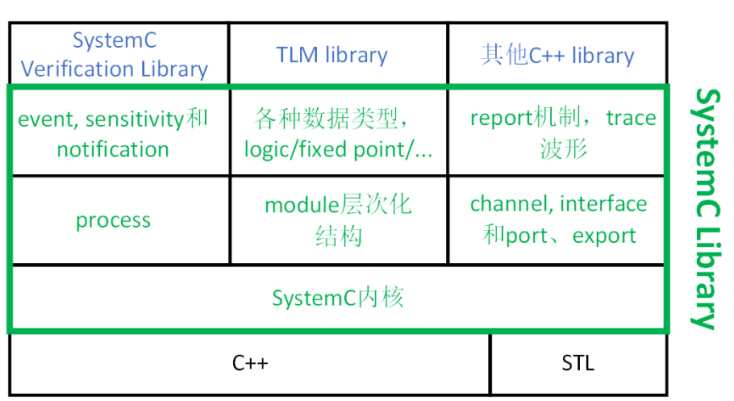
SystemC内核：用于调度SystemC process，功能和Verilog、VHDL仿真内核非常相似；可以认为类似于Linux内核的极度简化版本。

SystemC process：由用户定义，模拟硬件和软件的行为。其并行运行，由SystemC内核调度执行。

SystemC event：在特定时间点发生的事件，SystemC内核和Verilog仿真内核一样是事件驱动型内核。

SystemC module：用于将建模对象模块化、层次化；

SystemC channel/interface/port/export：用于实现模块间通信



<https://blog.csdn.net/gsjthxy/article/details/126099348>

HDL

硬件描述语言Hardware Description Language，以文本形式描述数字系统硬件的结构和行为的语言，可以表示逻辑电路图、逻辑表达式，还可以表示数字逻辑系统所完成的逻辑功能。

Verilog HDL和VHDL是最流行的两种HDL

[https://blog.csdn.net/landyjzlai/article/details/128647128]

Verilog HDL和VHDL

Verilog HDL和VHDL（Very-High-Speed Integrated Circuit Hardware Description Language，超高速集成电路硬件描述语言）是世界上最流行的两种硬件描述语言，都是在20世纪80年代中期开发出来的。前者由Gateway Design Automation公司（该公司于1989年被Cadence公司收购）开发，后者为美国军方组织开发。两种HDL均为IEEE标准。

——中国、日本、美国大部分采用Verilog，而VHDL在欧洲有较广泛的应用。

Verilog可以从五个层次对电路（系统）进行描述，包括:系统级、算法级、寄存器传输级（即RTL级）、门级、开关级。我们平时用的最多的为RTL级，故Verilog代码也经常被称为RTL代码。

HLS #2

图片所在文档：https://docs.amd.com/v/u/en-US/ug474\_7Series\_CLB

用户通过Vitis HLS将C/C++翻译成HDL，最终映射成FPGA内部的LUT、DSP及RAM资源等

SLICE：Xilinx7系中有两种SLICE：SLICEL、SLICEM，SLICEM比SLICEL多了写存储数据功能

https://blog.csdn.net/suiyaopu8894/article/details/106458705

FPGA每个SLICE基本都由LUT、进位链、MUX、寄存器组成

LUT（look up table）：查找表

MUX（[Multiplexer](https://blog.csdn.net/ZBigluck/article/details/129120143?ops_request_misc=%7B%22request%5Fid%22%3A%22E2AAF123-E12F-4A02-A39C-2D2F1FCCC153%22%2C%22scm%22%3A%2220140713.130102334..%22%7D&request_id=E2AAF123-E12F-4A02-A39C-2D2F1FCCC153&biz_id=0&utm_medium=distribute.pc_search_result.none-task-blog-2~all~top_click~default-2-129120143-null-null.142^v100^control&utm_term=MUX" \t "https://so.csdn.net/so/_blank)）：多路选择器

LUT：

FPGA以查找表代替门电路，如6位输入1位输出的任意门电路，都可以用LUT6来表示，LUT6本质是64\*1（深度为64，位宽为1）的ROM（Read Only Memory），将6位输入当做6位地址线，则输出就是当前地址存储位。

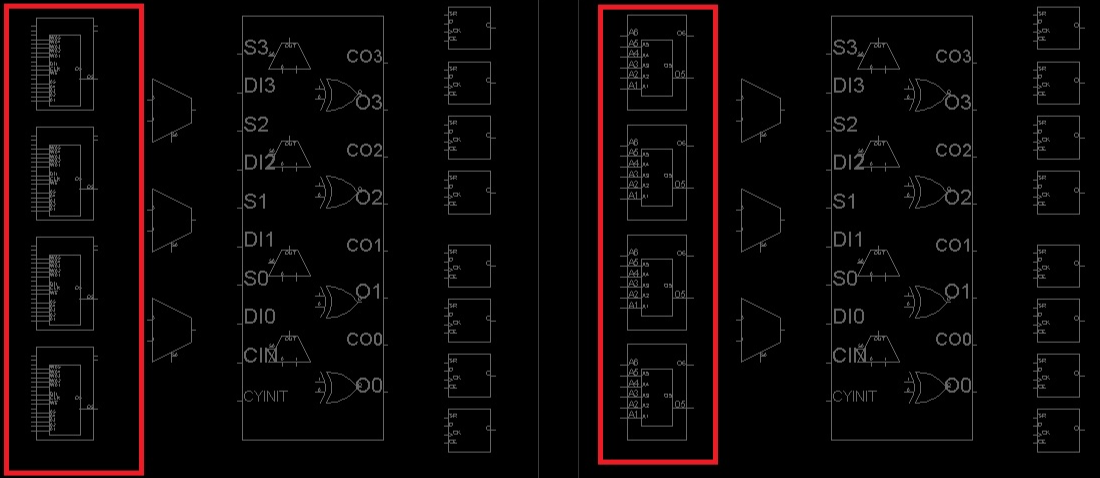
LUT6可以用作5位输出2位输出，因为LUT6实际由2个LUT5（5位输入1位输出）组成，只需要分别对I0-I4指定真值表就可以实现2位输出，将2位输出经过由I5控制的MUX输出为O6，即可实现6位输入1位输出，即LUT6，此时O5仍有上面LUT5的输出，但一般不用；使用LUT6的5位输入2位输出时，I5将强制置1，使O6的输出为下面LUT5的输出。

输出是1位，输入比6位少时，仅需无视高位填LUT6真值表，再写入LUT6即可

输入大于1位，或输入比6位多时，只需要用多个LUT6并行处理，这就会使多个输入经过多条路径才能得到输出，而路径之间延迟难以一致，实际应用中会使用寄存器来进行同步。

DRAM（Distributed RAM，分布式随机存储器）：

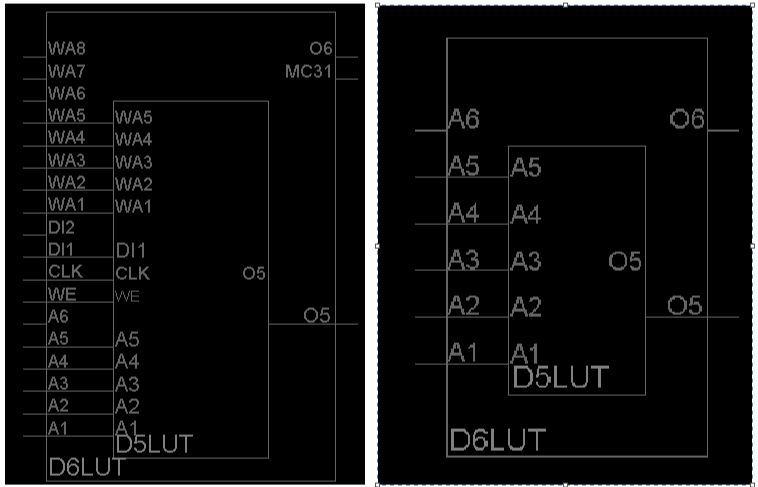
左侧为SLICEM，具有写存储数据功能，LUT6可以做RAM，右侧为SLICEL，使用只读的LUT6，二者仅红框里的LUT6部分不同



两者的LUT6：

相同点：都具有地址输入线（A1-A6），2个输入口（O5-O6）

不同点：SLICEM的LUT6具有写地址输入线（WA1-WA8），写数据端（DI1-DI2），写使能端（WE）



1个SLICEM中有4个LUT6，可以组合为多种DRAM：

·32x1或64x1的单端口DRAM，1个LUT6

·32x1或64x1的双端口DRAM，2个LUT6

·32x6或64x3的简单双端口DRAM，4个LUT6

·32x2或64x1的四端口DRAM，4个LUT6

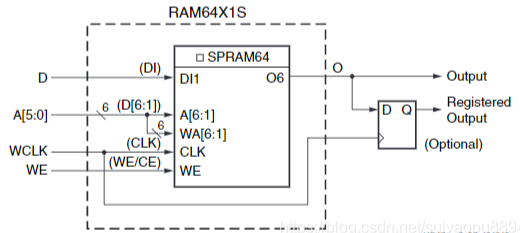
配合MUX使用可以组成更大深度的DRAM：

·128x1的单端口DRAM，2个LUT6+1个MUX

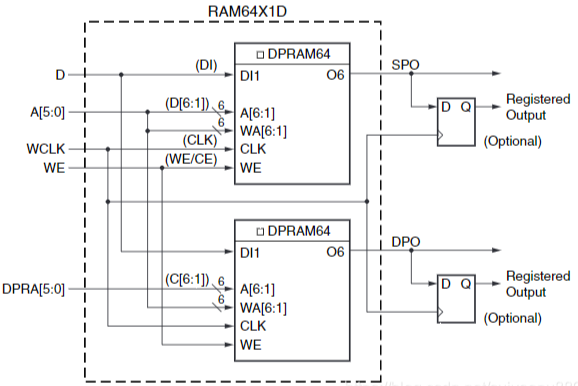
·128x1的双端口DRAM，4个LUT6+2个MUX

·256x1的单端口DRAM，4个LUT6+3个MUX

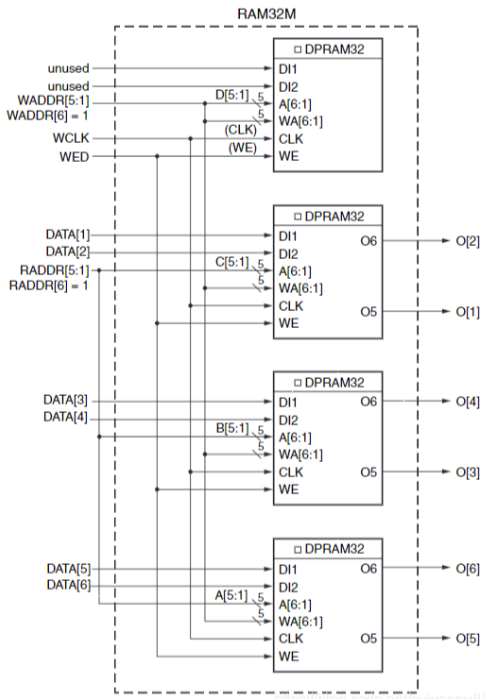
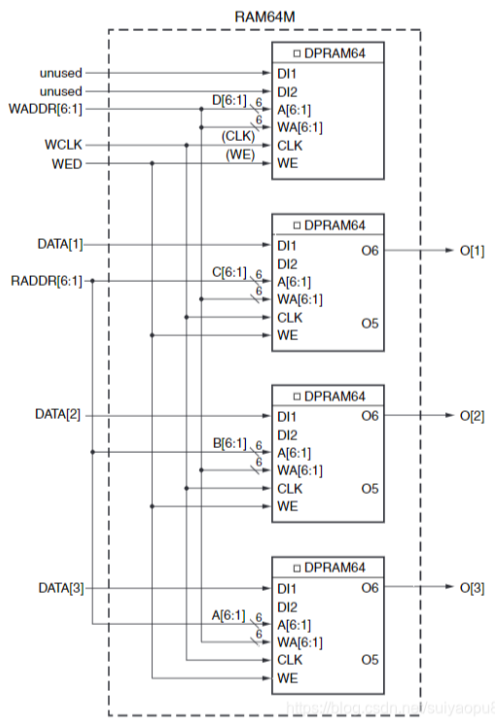
单端口DRAM：同步读，同步写，DI1数据输入，A[5:0]指定地址，WE置高写数据，WE置低读数据



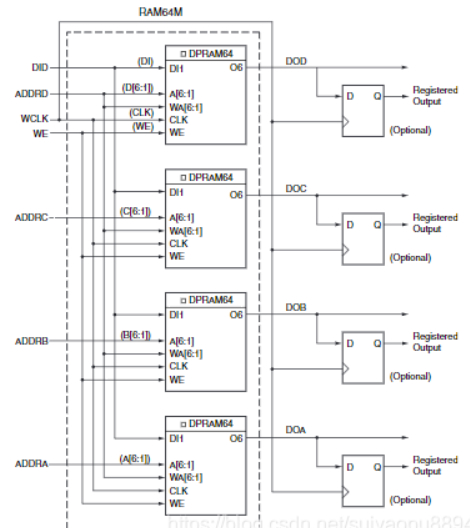
双端口DRAM：一个端口可同步写（通过A[5:0]指定两个LUT6的地址同时写入）、异步读，另一个端口只能异步读。上方LUT6通过A[5:0]指定读取地址，下方LUT6通过DPRA[5:0]指定读取地址。



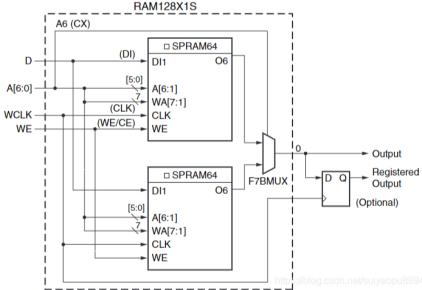
简单双端口DRAM：一个端口WADDR[6:1]只可同步写，另一端口RADDR[6:1]只可异步读。64x3简单双端口DRAM（左图）可以存储64个3位数据，DATA[3:1]并行输入，三个O6（O[3:1]）并行输出；32\*6可以存储32个6位数据，DATA[6:1]输入，O[6:1]输出。（第一个LUT6也可以使用吗？）



四端口DRAM：一个端口（ADDRD）可同步读、异步写，其他三个端口（ADDRA、ADDRB、ADDRC）仅可异步读，与双端口类似，四个LUT存相同数据，但每个端口都能各自读不同地址内容（提高并发读取性能）。



128x1单端口DRAM：由2个64x1单端口DRAM+1个MUX组成。写入时A[6:0]用于指定地址WA[7:1]；读取时A[5:0]用于指定地址，A[6]用于MUX选择控制。



32位移位寄存器：SRLC32E，使用SLICEM的LUT配置为32位移位寄存器。（太复杂了，之后详细分析）

移位寄存器中的数据在脉冲信号作用下依次逐位移动，因此每个LUT可以将串行数据延迟1到32个时钟周期。移位寄存器使用电平转移，即输入是多少V输出就是多少V，可能出现亚稳态电平。

<https://blog.csdn.net/qq_43433724/article/details/137976382>

<https://blog.csdn.net/lsh11111/article/details/136158623>

<https://blog.csdn.net/apple_53311083/article/details/132259262>

移位寄存器的应用包括：

•延迟或延迟补偿

•同步FIFO和内容可寻址存储器（CAM）

移位寄存器功能包括：

•写入操作

•动态读取访问

•通过5位地址总线A[4:0]执行

•通过改变地址，可以异步读出32位中的任何一位（在O6 LUT输出端，在基元上称为Q）

•此功能在创建较小的移位寄存器（小于32位）时非常有用。

-例如，在构建13位移位寄存器时，将地址设置为第13位。

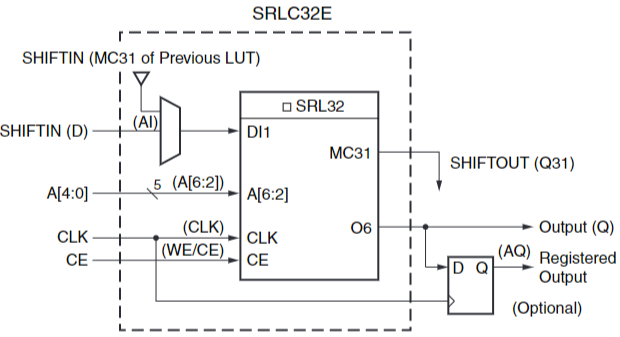
•存储元件或触发器可用于实现同步读取。

-触发器的时钟输出决定了整体延迟并提高了性能。

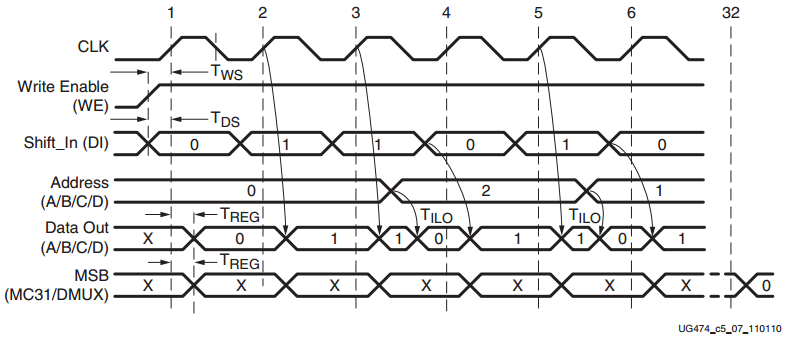
-但是增加了一个额外的时钟延迟周期。

•不支持设置或重置移位寄存器。

图2-15是32位移位寄存器的逻辑框图。

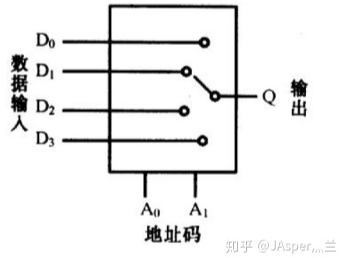


移位寄存器时序图

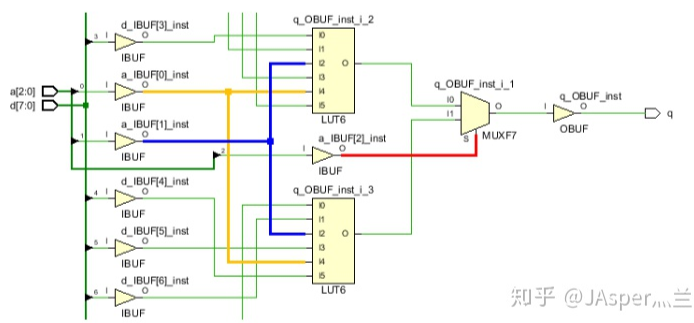


MUX：多路数据选择器

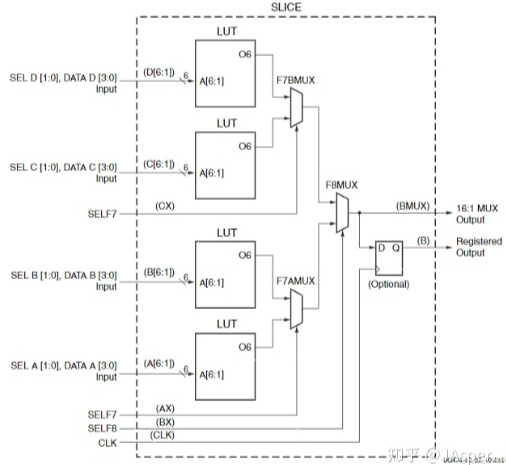
下图为四选一数据选择器（MUX4\_1），由2位地址码A0-A1选择将数据输入D0-D3的其中一位传送到输出。因为也是6位输入1位输出，所以MUX4\_1本质上也是LUT6，只不过它有着特定的连线方式



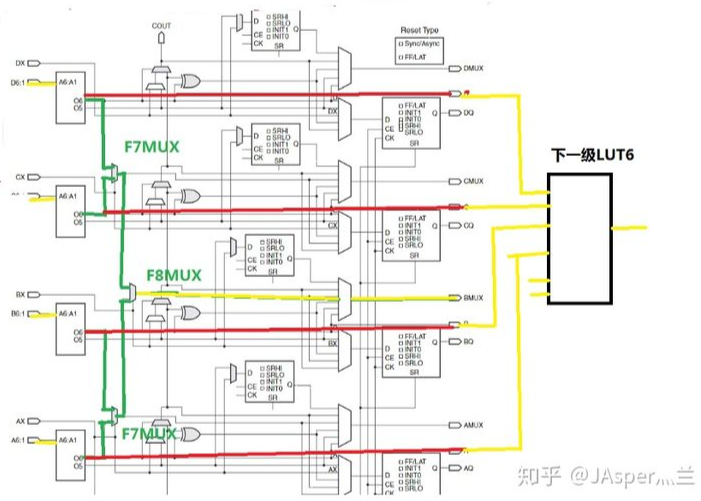
使用2个LUT6和一个MUXF7可以组成一个MUX8\_1，其原理图如下，低2位地址线a[1:0]（黄线和蓝线）分别连接到2两个LUT6的同一位置，高1位（红线）用于连接MUXF7进行数据选择；8位数据输入d[7:0]分别经过2个LUT6进行4选1，最终由MUXF7进行2选1。



MUX16\_1：与MUX8\_1类似，先通过4个MUX4\_1从16个数据选出4个（低2位地址），再通过2个MUXF7选出2个（次高位地址），最后通过MUXF8进行2选1（最高位地址）。



16选1中的4选2选1不单独使用LUT6进行4选1的原因：下图是一个SLICE，红线是使用5个LUT6进行16选1，绿线是通过4个LUT6+MUX进行16选1。可以看出红线路径无法保证延迟一致性，走线越长延迟偏差就越大，容易产生毛刺，造成逻辑错误。这也是MUXF7和MUXF8这样设计（靠近LUT6，且结构对称）的原因，其本身就是为多路选择器设计的



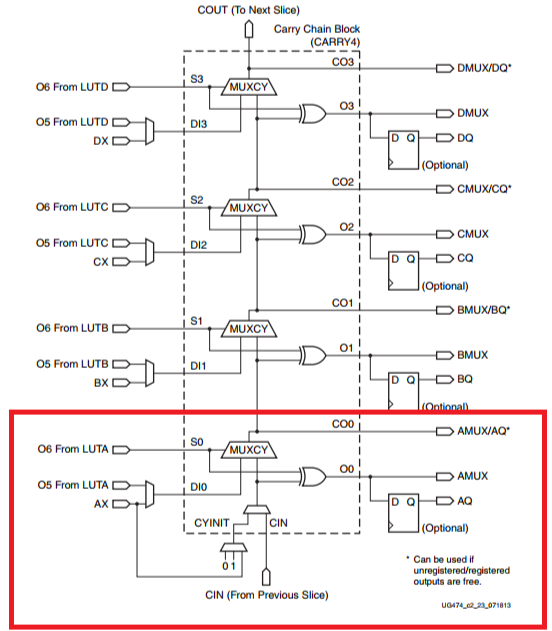
进位链：

基础：全加器

全加器最低位CIN置1为减法，如0101-0010=0011，0101补码不变，(-0010)补码为反码+1，将0101和0010的反码传入，CIN置1即为反码+1操作，等价于0101+1110=0011。

进位链结构：

红框部分为它的一个基本单元（本质为全加器）



·CIN=进位输入（当它在最低位时，置0做加法，置1做减法）

·S0=A0⊕B0，它来自LUTA（作为LUT2使用）的O6

·O0=S0⊕CIN=A0⊕B0⊕CIN（相当于是全加器的S），它来自LUTA的O5或外面外部输入AX

·DI0 = A0或者B0（两个加数中的一个，配合S0使用）

MUXCY（下一级进位）原理：

产生进位的条件是在A、B以及CIN中有两个或两个以上的1。

·S0=0时，可能两个加数都为0（此时进位为0）或两个加数都为1（此时进位为1），因此MUXCY选择DI0（两个加数的任意一个）作为进位输出（CO0）。此时不需要管上一级进位，因为0+1=1，不会产生进位。

·S0=1时，即A0与B0中有1个为1，此时如果上一级进位为1则进位输出为1，否则不进位，即上一级进位CIN和进位输出CO相同，因此MUXCY选择上一级进位（CIN0）作为进位输出（CO0）。

以一个8位加法为例子：

module carrychain(

input [7:0] a,

input [7:0] b,

output[7:0] o

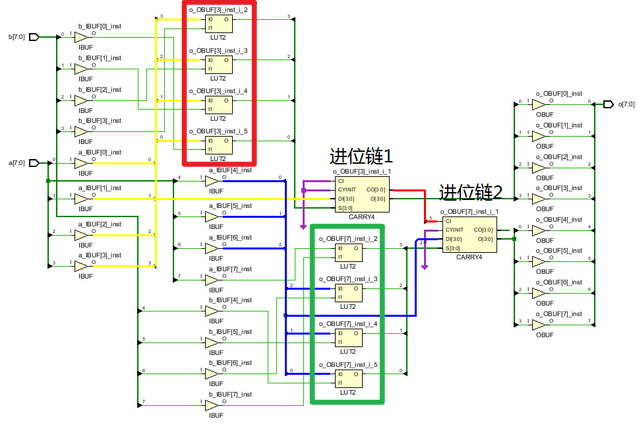
);

assign o=a+b;

endmodule

因为1个LUT6只能作为4位全加器使用，所以会使用到2个LUT6，其内部连线如下：

a和b的低4位传到了红框中，结果传入进位链1，第4位进位传到进位链2，并输出低4位运算结果o[3:0]；高4位传到绿框中，结果传入进位链2，最终输出高4位运算结果o[7:4]



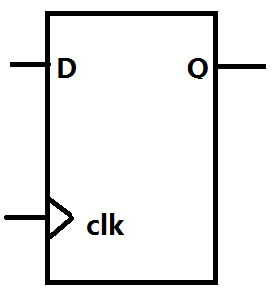
进位链级联数有限制，其取决于当前列SLICE的个数，且不能跨逻辑区域，逻辑区域里进位链最大级数=逻辑区域内一列的SLICE个数，全加器位宽为个数x4。

存储单元：

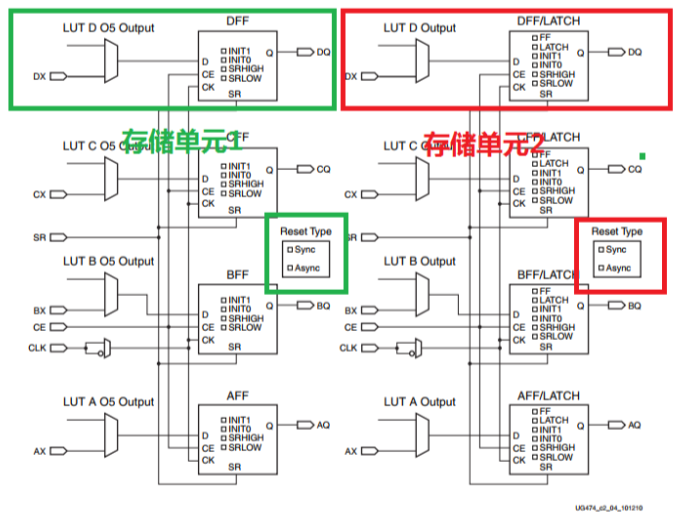
基础：存储单元。触发器，通过时钟沿到来改变存储的输出状态；锁存器，通过电平变换改变存储的输出状态。

FPGA中触发器使用较多。

触发器特性：在下一次时钟沿到来之前，输出Q不变，时钟沿触发时，输出D的电平转移到Q。以3.3VMOS为例，输入电平高于高电平阈值2.0V时，Q输出高电平3.3V，输入低于低电平阈值0.7V时，Q输出低电平0V。相比LUT移位寄存器的电平转移，触发器可以降低亚稳态的影响（如2.1V、0.7V）。



Xilinx7系的存储单元结构主要分两种，如下图存储单元1、2。



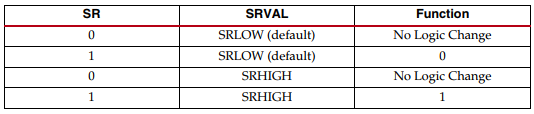
相同点：

有数据输入端D、时钟使能端CE、时钟输入端CK、复位端SR

能通过配置INIT1、INIT0，将上电/全局复位后的初始电平置为高或低

能通过配置SRHIGH、SRLOW来控制用户复位后的电平（推荐高电平复位，因为无需额外资源消耗，低电平还需要加个LUT做反相）

同一SLICE中，同一类型存储单元的用户复位可以配置为同步或异步复位。



例1：INIT和SR复位电平的配置

module and\_8\_8(

input clk,

input rst,

input [7:0]i\_data\_1,

input [7:0]i\_data\_2,

output[7:0]o\_data

);

reg [7:0]r\_data\_tmp=8'b0; //这里将会8个存储单元配置为INIT0 ,如果改成8'b1111\_1111则配置成INIT1

always@(posedge clk)

begin

if(rst)

r\_data\_tmp<=8'b0;//这里将会8个存储单元配置为SRLOW ,如果改成8'b1111\_1111则配置成SRHIGH

else

r\_data\_tmp<=i\_data\_1&i\_data\_2;

end

assign o\_data=r\_data\_tmp;

Endmodule

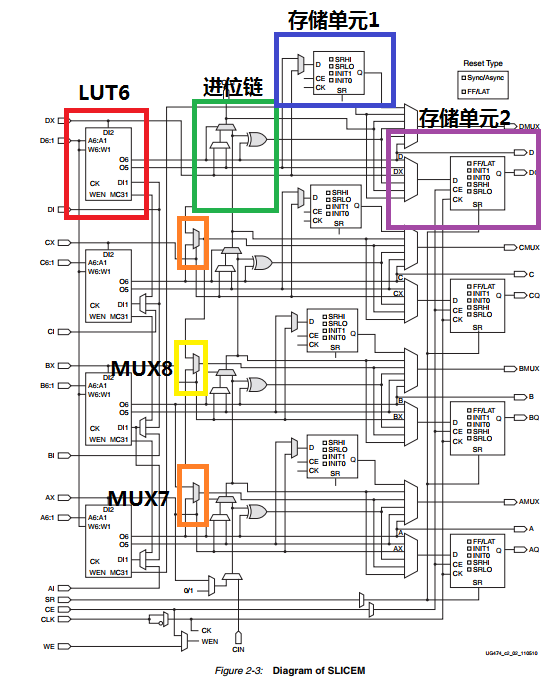
不同点：

存储单元1只能做触发器，存储单元2既能做触发器也能做锁存器。

存储单元1只能由外部旁路输入或所在行LUT6的O5输出作为输入。存储单元2在1的基础上，还能由所在行LUT6的O6输出、MUX7/8的输出、进位链的输出作为输入。

CLB：

SLICEM的架构图如下，可以分为相同的4行，SLICEL与之类似，差别仅为LUT6不具备存储功能。



蓝紫框为存储单元1、2，verilog中每1位reg几乎都会用到1个存储单元。输入取决于单元前的MUX，存储单元1有2种输入方式，存储单元2有6种输入方式。

红框为LUT6，内部由2个LUT5和1个MUX组成，可以实现6入1出、5入2出、4选1功能。SLICEM中还可以做DRAM、移位寄存器以实现数据读写、移位输出功能。

橙黄框MUXF7、MUXF8，将LUT拼接为更大深度的LUT、DRAM或移位寄存器（MUX不会增加数据宽度）

绿框进位链，实现多位加法器功能，位宽取决于逻辑区域列高。

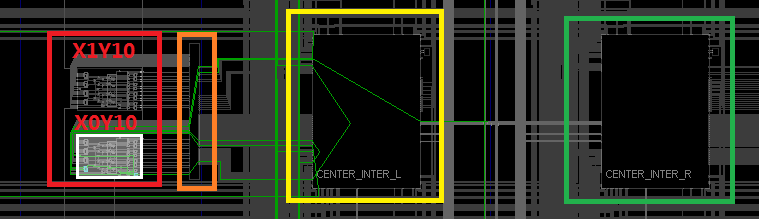
接近输出的MUX（A/B/C/DMUX输出），与存储单元2前的MUX特性类似，接收6个输入，有5个输入连在一起，只不过第一个是直接输出，第二个打一拍再输出；第一个MUX不接受外部旁路输入，接收存储单元1的输出作为输入。（两个MUX都没有数据选择端，运行时的输出是确定的）

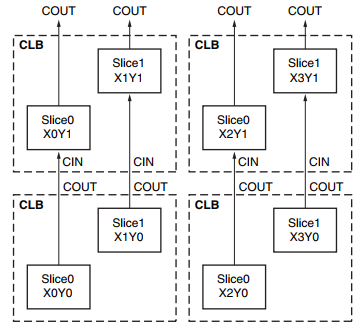
SLICE在CLB中的排列：

一个CLB有2个SLICE，上一级SLICE进位链输出连接到下一级SLICE进位链输入。

其他数据的互联：

同一CLB内的SLICE通过CLB内的快速互联单元（橙框）直连，黄框和绿框用于将线连到更远处。





适用于 Vitis 内核流程的接口

Vitis 内核流程为已编译的内核对象 (.xo) 提供支持，以便从主机应用和赛灵思的 Xilinx Run Time (XRT) 来执行软件控制。如 Vitis 统一软件平台文档中的[内核属性](https://docs.xilinx.com/access/sources/dita/topic?resourceid=fiv1568160307462.html&Doc_Version=2022.2 English&url=ug1393-vitis-application-acceleration" \t "https://docs.amd.com/r/2022.2-%E7%AE%80%E4%BD%93%E4%B8%AD%E6%96%87/ug1399-vitis-hls/_blank) 中所述，此流程具有非常具体的接口要求，Vitis HLS 必须满足这些要求。

Vitis HLS 支持多种存储器、串流和寄存器接口范例，其中每个范例都遵循某个接口协议并使用适配器来与外部世界进行通信。

·存储器范例 (m\_axi)：内核通过存储器（如 DDR、HBM、PLRAM/BRAM/URAM）来访问数据

·串流范例 (axis)：数据从其它串流源（例如，视频处理器或其它内核）串流至内核中，也可从该内核流出。

·寄存器范例 (s\_axilite)：内核通过寄存器接口来访问数据，软件则通过寄存器读/写来访问数据。

Vitis HLS

第一个工程：xc7z020clg400-1

先执行C仿真，得到初步结果

再执行C综合，可以看到C代码被综合成RTL代码后，使用的硬件资源评估、性能评估等

然后跑RTL和C的联合仿真CoSimulation（dump trace选择port可以查看波形）

“Block-level IO Handshake”端口为握手端口信号

“SW-to-HW Mapping”为输入输出数组或变量端口，端口包含地址总线、使能信号、写使能信号、数据总线等

New Solution：

<https://blog.csdn.net/weixin_44007264/article/details/122949673>

右键myloop，插入PIPELINE

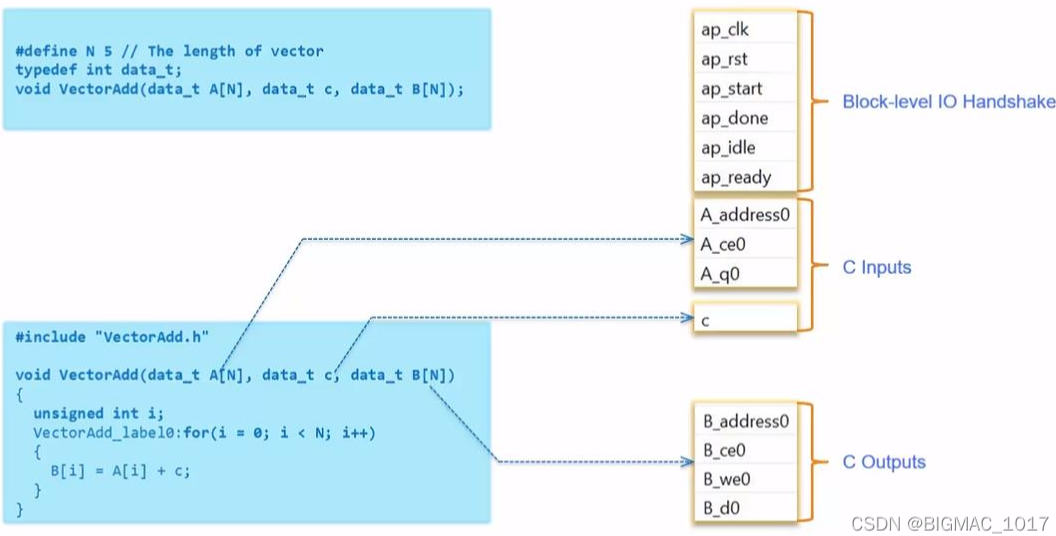
针对solution2进行综合

Project栏-Compare Reports对比两个solution的报告，Vitis HLS较新的版本中，即便没有设置directives，也会默认添加HLS PIPELINE指令，因此两个Report是相同的

对于每个C代码的For循环创建一个标签

Main正常执行返回0，错误返回1，便于判断仿真出错

C例程代码使用HLS综合后，生成的接口结构：



<https://blog.csdn.net/kzz6991/article/details/107405868>