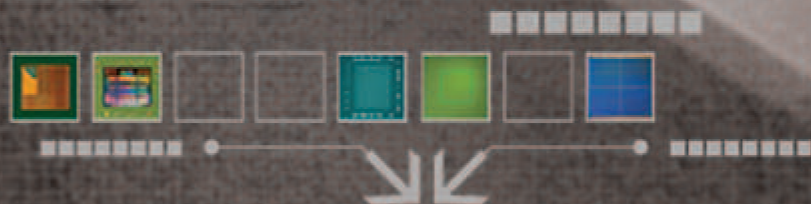
XILINX
ALL PROGRAMMABLE™

XILINX专家大讲堂

—Vivado 设计套件篇



赛灵思市场部制作

2013年秋季

Vivado 设计套件

行业首款 SoC 增强型设计套件

Vivado™ Design Suite 提供全新构建的 SoC 增强型、以 IP 和系统为中心的下一代开发环境，以解决系统级集成和实现的生产力瓶颈。

主要文档

- Vivado Design Suite Backgrounder
- Vivado IP Integrator Backgrounder

快速链接

- Vivado Design Suite 评估和 WebPACK
- IP 中心
- 支持和技术文档
- 支持的目标参考设计

- Vivado 课程
- 高层次综合课程

Vivado Design Suite 与 7 系列 All Programmable FPGA 完美结合在一起，可提供高达 3 速度等级的性能优势，同时将功耗平均降低约 35%。此外，您还可以更轻松地对更多的设计进行布线，相对同类竞争解决方案而言，其 LUT 利用率提高了 20% 以上。

目　录

- 十招加速 Vivado IPI 设计
- Vivado HLS 中指针作为 top 函数参数的处理
- Vivado HLS 中的浮点设计编码风格与技巧
- 编写高效 Vivado HLS 工程 testbench 的三个要素

Vivado 设计套件

行业首款 SoC 增强型设计套件

Vivado™ Design Suite 提供全新构建的 SoC 增强型、以 IP 和系统为中心的下一代开发环境，以解决系统级集成和实现的生产力瓶颈。

主要文档

- Vivado Design Suite Backgrounder
- Vivado IP Integrator Backgrounder

快速链接

- Vivado Design Suite 评估和 WebPACK
- IP 中心
- 支持和技术文档
- 支持的目标参考设计

- Vivado 课程
- 高层次综合课程

Vivado Design Suite 与 7 系列 All Programmable FPGA 完美结合在一起，可提供高达 3 速度等级的性能优势，同时将功耗平均降低约 35%。此外，您还可以更轻松地对更多的设计进行布线，相对同类竞争解决方案而言，其 LUT 利用率提高了 20% 以上。

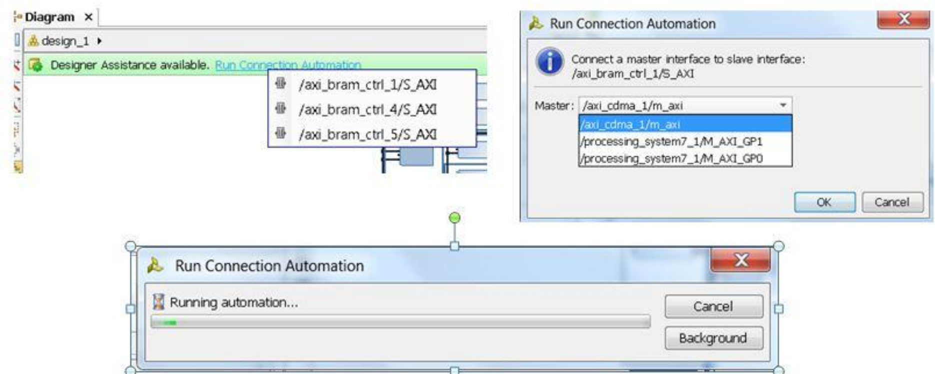
十招加速 Vivado IP Integrator 设计

作者：Hank Fu，Xilinx 处理器专家

Xilinx 在 Vivado 2013.2 中，推出了全新的图形化设计工具 IP Integrator，让嵌入式设计更加直观。新工具上手，也会有些不适应的地方。下面十招，可以让你更快适应 IP Integrator，开始定制自己的嵌入式系统芯片。凭借 IP Integrator，可以像 Freescale、TI 一样设计 SOC，自豪吗？

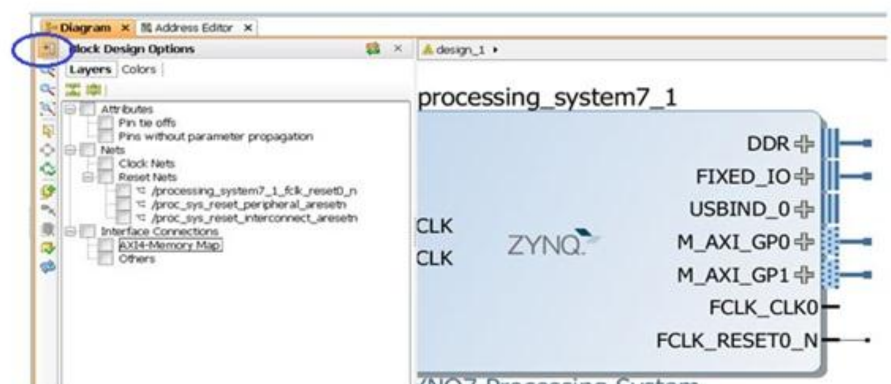
第一招： 使用“Run Connection Automation”自动连接 IP。

Connect AXI interface automatically



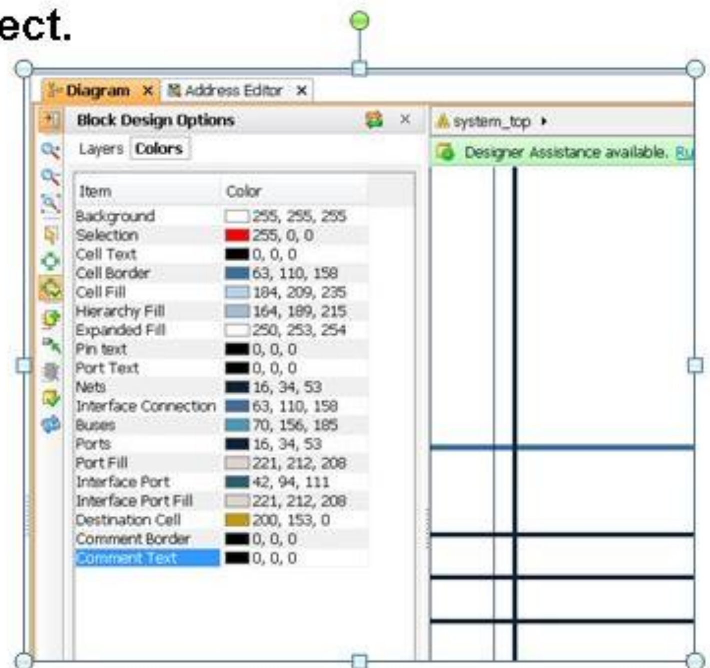
第二招： 隐藏暂时不关心的信号类型，简化设计图。点击左上角的推门图标，马上就有一个新窗口“Block Design Options”显示,再选中其中的“Layers”。这个窗口按类型显示了所有的信号。一类信号，就是一个 layer。只有选中的 layer，才会在设计图中显示。

Hide connection in order to make the diagram simplified.



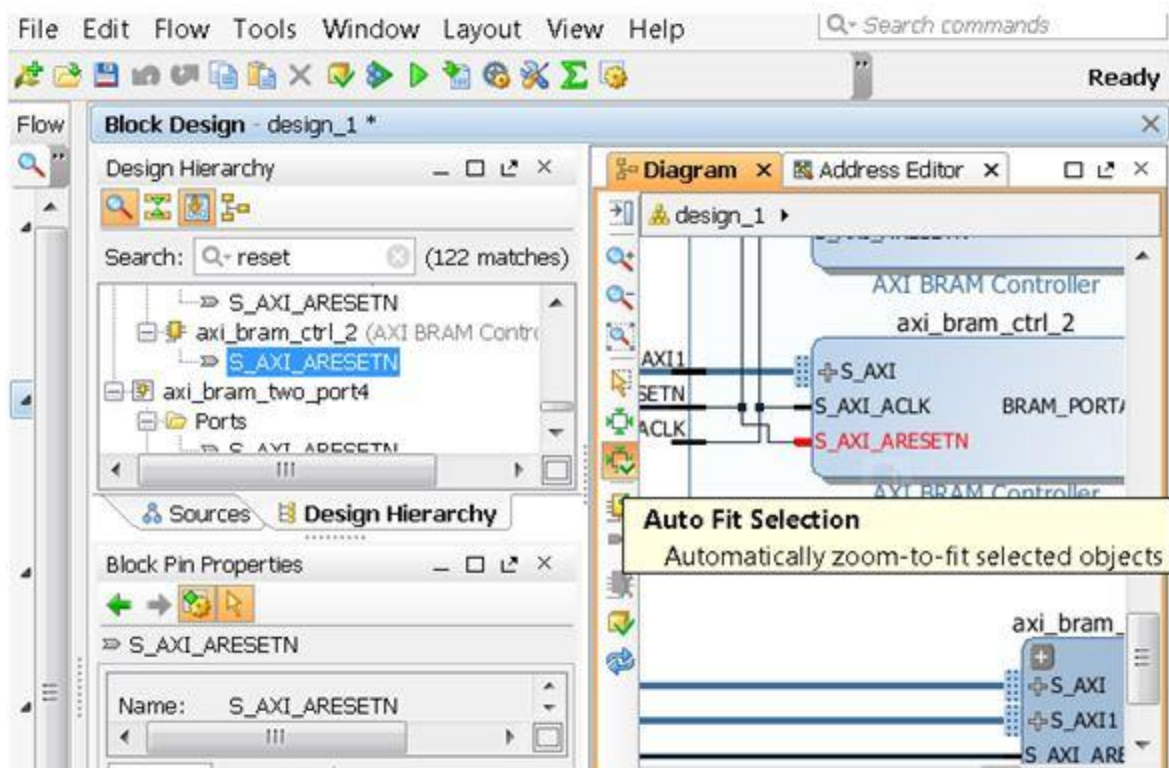
第三招：指定信号颜色，让信号更显眼。比如可以给选中的信号设置红色等更明亮的颜色。同样是“Block Design Options”窗口,再选中其中的“Colors”，则会有颜色配置窗口显示，可以按对象类型指定颜色。常用的是给选中的信号“Selection”指定一个显眼的颜色。

Set color for specified object.

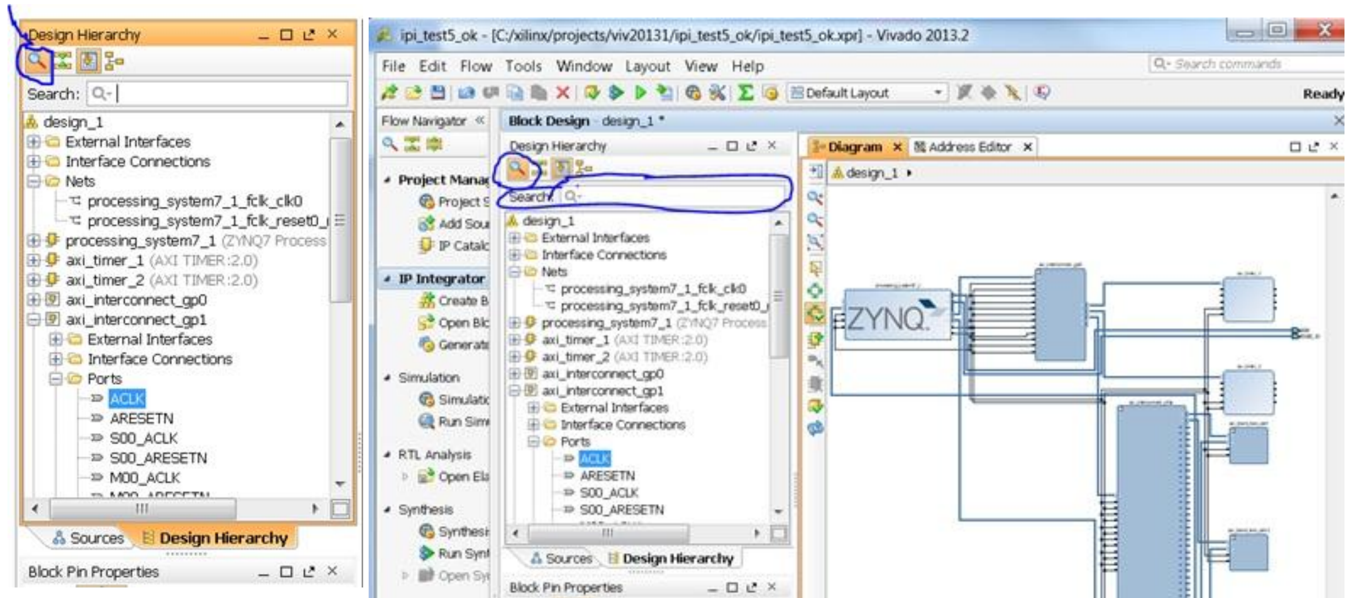


第四招：选中“Auto Fit Selection”模式，让设计图适配你的屏幕大小。图太小了，可以点 40aa.png 放大。图太大了，可以点 40a0.png 变小。点击 40a3.png，设计图可以适配你的窗口大小。如果图太大了，选中一个信号，还不一定找得到它在哪儿。那就请 40a4.png 帮忙，它能让你选中的信号在图的正中，而且高亮显示。

Select, highlight, and centralize the object.

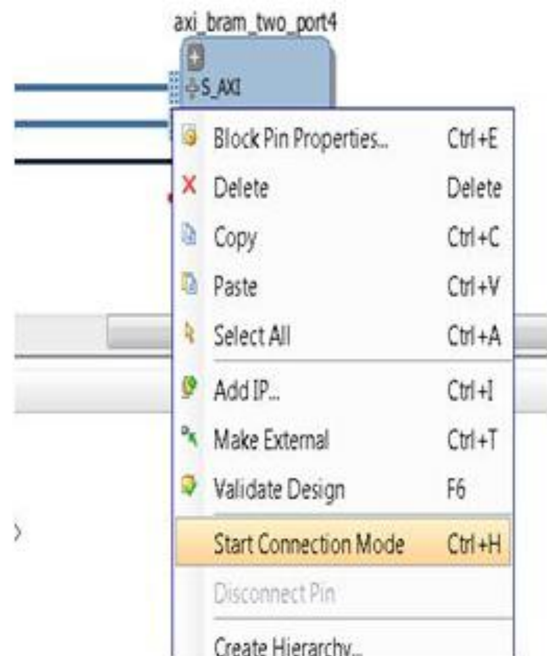


第五招：在“Design Hierarchy”中搜索信号，让信号自己跑出来。点击 5a00a.jpg，能显示搜索框，输入信号名，就可以找到相关信号。如果“Auto Fit Selection”模式被选中，再点击信号名，信号就会在设计图窗口的正中央高亮显示。



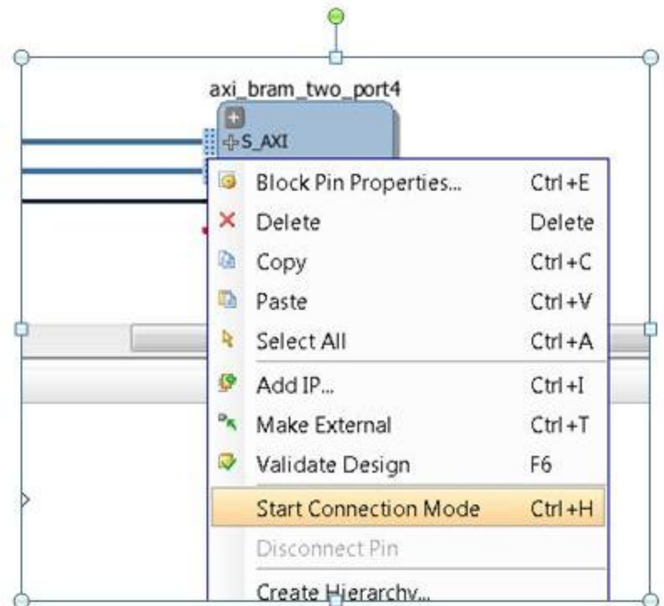
第六招： 保持连接模式，轻松寻找目标信号，实现信号链接。有时图太大，拖信号会比较麻烦。图小了，可以显示全部，找信号麻烦。图大了，显示不全。这时可以选择源信号，在右键菜单中选择“”，然后再去从容不迫找到目标信号，比如去搜索框中搜索，最后点击目标信号，就可以实现连接。

Keep Connection Mode



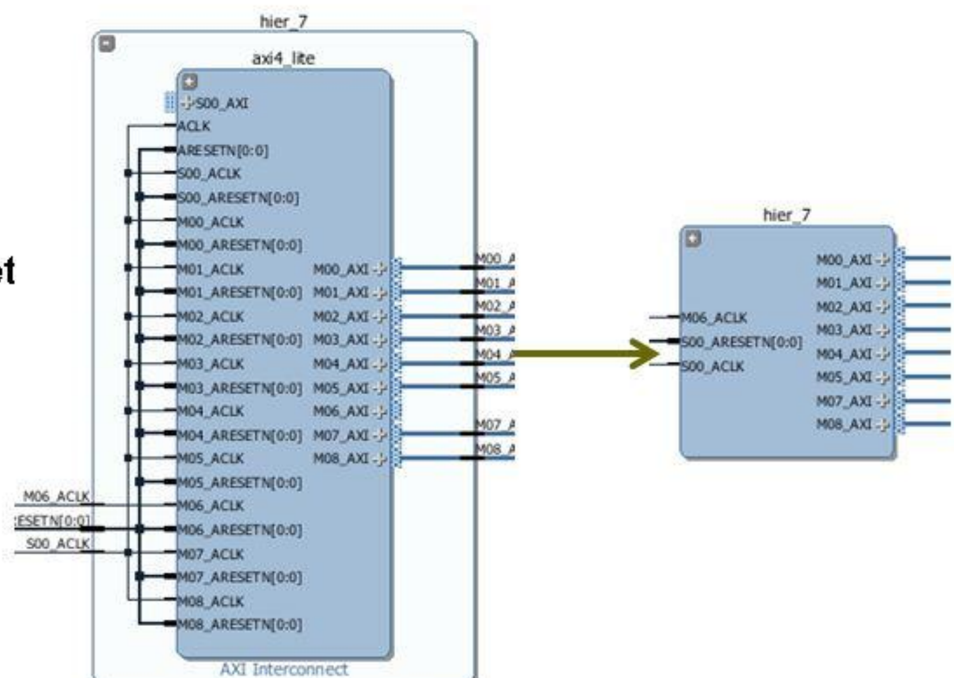
第七招：则把前面几招串起来，在一个大的设计中可以快速连线。它包括使用“Auto Fit Selection”高亮显示选中的信号，使用查找信号找到源信号，使用“Start Connection Mode”保持连接状态，再使用查找信号找到目标信号，最后点击目标信号就实现信号连接。

- Select “Auto Fit Selection”
- Select source signal by searching
- Select “Start Connection Mode”
- Find target signal by searching
- Click target signal



第八招：创建层次化模块“Hierachy”隐藏信号和模块，简化设计图。比如下图中间的 interconnect，有都使用相同的时钟和复位信号，但是有很多这种信号，看起来很复杂。创建“Hierachy”后，只有一个时钟和一个复位信号，简单多了。

- Create hierachy to hide signals with same source or desination.
- Very useful for reset and clock signals.



第九招，使用 tcl 脚本。TCL 脚本也可以实现信号连接等功能。

- get_bd_nets
  - Get a list of ports.
  - get_bd_nets /axi*
- connect_bd_net
  - Connect port and pin object list
  - connect_bd_net [get_bd_pins /processing_system7_1/IRQ_F2P] [get_bd_pins /xlconcat_3/dout]
- connect_bd_intf_net
  - connect_bd_intf_net [get_bd_intf_pins /processing_system7_1/M_AXI_GP0] [get_bd_intf_pins /axi4_lite/S00_AXI]

第十招：输出 IP Intergrator 设计做为 IP，可以再次 重用。具体请参考 UG994 中的 Packaging the Block Diagram。

Trick: Export IP

- Export the whole IP Integrator design as one IP catalog
  - UG994 Packaging the Block Diagram

©Copyright 2012 Xilinx, Inc. XILINX, the Xilinx logo, Virtex, Spartan, ISE and other designated brands included herein are trademarks of Xilinx in the United States and other countries. All other trademarks are the property of their respective owners.

Vivado HLS 中
指针作为 top 函数参数的处理

作者：Harvest Guo, Xilinx DSP 专家

Vivado 设计套件

行业首款 SoC 增强型设计套件

Vivado™ Design Suite 提供全新构建的 SoC 增强型、以 IP 和系统为中心的下一代开发环境，以解决系统级集成和实现的生产力瓶颈。

主要文档

- Vivado Design Suite Backgrounder
- Vivado IP Integrator Backgrounder

快速链接

- Vivado Design Suite 评估和 WebPACK
- IP 中心
- 支持和技术文档
- 支持的目标参考设计

- Vivado 课程
- 高层次综合课程

Vivado Design Suite 与 7 系列 All Programmable FPGA 完美结合在一起，可提供高达 3 速度等级的性能优势，同时将功耗平均降低约 35%。此外，您还可以更轻松地对更多的设计进行布线，相对同类竞争解决方案而言，其 LUT 利用率提高了 20% 以上。

指针作为 C 语言精华，对于软件设计者比较好理解，但是在 xilinx vivado HLS 高级语言综合的设计中，由于其综合后对应的硬件元素难以用软件的概念解释，常常令程序设计者和 VHLS 工具使用者头痛。本文采用浅显易懂的描述方式，结合具体的 c 代码例子，详细描述了常用三种指针的设计类型，以及其作为顶层函数参数时，采用不同的编码风格和 HLS 约束策略，满足设计者对指针作为 RTL 接口的需求。

基本指针类型

基本指针类型指的是指针没有运算或者没有多次的存取（读写）。指针作为 top 函数的参数时，指针综合为 wire 型或者握手协议类型接口。如下例子 1-1：

```
void pointer_basic (dio_t *d) {
static dio_t acc = 0;
acc += *d;
*d = acc;
}
```

例子 1-1 基本类型指针作为顶层函数参数

在这个例子中，只是简单的读写指针指向的变量值，并没有对指针做偏移或者指针（地址）运算，其接口综合为线型的 RTL 接口。

指针运算类型。

指针作为 top 层函数参数，并且函数中有对指针运算时，我们称之为指针运算类型。指针运算常常限制指针可能综合的接口类型。如下例中，指针做了偏移运算用于累加数据，从第二个值开始读出累加，并将每次累加结果写入上一个地址中。

```
void pointer_arith (dio_t *d) {
static int acc = 0;
int i;
for (i=0;i<4;i++) {
acc += *(d+i+1);
*(d+i) = acc;
}
}
```

例子 1-2 指针运算类型作为顶层函数参数

下面代码例子 1-3 是这个指针运算类型仿真的 testbench。因为函数 pointer_arith 内部的 for 循环进行数据累加，testbench 通过数组 d[5]分配了地址空间并对数组赋值。

```
int main () {
dio_t d[5], ref[5];
int i, retval=0;
FILE *fp;
// Create input data
for (i=0;i<5;i++) {
d[i] = i;
ref[i] = i;
}
// Call the function to operate on the data
pointer_arith(d);
// Save the results to a file
fp=fopen("result.dat","w");
printf(" Din Dout\n", i, d);
for (i=0;i<4;i++) {
fprintf(fp, "%d \n", d[i]);
printf(" %d %d\n", ref[i], d[i]);
}
fclose(fp);
// Compare the results file with the golden results
retval = system("diff --brief -w result.dat result.golden.dat");
if (retval != 0) {
printf("Test failed!!!\n");
retval=1;
} else {
printf("Test passed!\n");
}
// Return 0 if the test
return retval;
}
```

例子 1-3 指针运算类型作为顶层函数参数的 testbench

在 C 编译环境下仿真上面例子 1-3 的代码，结果如下：

```
Din :   Dout
0       1
1       3
2       6
3       10
Test passed!
```

指针运算带来的问题是，通常情况下，指针偏移是不规则的，不能按顺序存取指针数据。而 Wire，握手类型或者 Fifo 接口类型没有办法乱序存取数据。

对于 wire 类型接口来说，当设计本身准备好接收数据时可以读入数据，或者当数据准备好 ready 时，可以写出数据。对握手和 Fifo 类型接口，当控制信号允许操作进行时，读入或写出数据。

在上面 wire,握手或者 FIFO 类型接口的情况下，数据从 0 元素开始，必须按顺序到达（写入）。在指针运算的例子 1-2 中，第一个数据从索引 1 开始读入（i 从 0 开始，0+1=1），对应于 testbench 中数据 d[5]的第二个元素。

当这种情况在硬件应用时，需要某种格式的数据索引，这种情况对于 wire 类型，或者握手类型还是 Fifo 类型来说，都不支持。像上例 1-2 指针运算的代码，只能综合成 ap_bus 接口，因为这种接口带有地址，当数据存取（读写）时，用于对应的数据索引指示。

还有一种方法，代码必须修改成如下例子 1-4 的风格，用数据 array 作为接口替代指针。这种方法应用了 array 作为 top 层参数时综合成 RAM 接口（ap_memory）的原理，memory 接口可以用地址作为数据的索引并且可以乱序执行，不必顺序存取操作。

```
void array_arith (dio_t d[5]) {
static int acc = 0;
int i;
for (i=0;i<4;i++) {
acc += d[i+1];
d[i] = acc;
}
}
```

例子 1-4 指针运算类型作为顶层函数参数修改为 array

Wire 类型、握手类型或 Fifo 类型接口仅仅可用在数据流方式，因此不能用在与指针运算相关的地方（除非数据从索引 0 开始并顺序处理）。同时注意，如果想综合为 FIFO 接口，Fifo 接口类型必须是只读或者只写，不能有读又有写操作。

多次读写（存取）指针类型

多次读写指针类型一般用作描述一个数据流方式的接口。

当 top 层函数参数使用指针，函数体对指针进行多次存取操作时，必须仔细考虑。在同一函数中对一个指针多次的读或者写，就会有多次指针存取发生，从而引起下列问题：

1. 对任何函数指针参数的多次存取要使用 volatile 限定符。
2. 对于 Top 层函数，如果要做 RTL 代码的混合仿真（co-sim），任何这种指针参数必须有这个接口存取次数的详细说明。
3. 确保在综合前验证 C 功能，确定符合功能要求，保证 C 模型正确。

如果设计模型要求函数参数指针多次存取，推荐使用数据流模式模型化设计，使用数据流模型可以避免我们将会在下面讨论到的，使用多次读写指针带来的一些问题。

这个章节使用设计例子 1-5 糟糕的数据流类型指针（pointer_stream_bad）解释，当多次存取指针时，为什么要使用 volatile 限定符。同时使用设计例子 1-8 好的数据指针类型（pointer_stream_better）来说明，为什么当 top 层函数参数包含有这种指针接口的设计时，应该用 C testbench 仿真验证确保设计的行为级模型正确。

在下面的例子 1-5 中，指针 d_i 读了 4 次并且 d_o 写了 2 次，设计的本意是存取操作通过 fifo 接口，综合后的 RTL 以数据流的方式读入或者写出数据。

```
void pointer_stream_bad ( dout_t *d_o, din_t *d_i) {
din_t acc = 0;
acc += *d_i;
acc += *d_i;
*d_o = acc;
acc += *d_i;
acc += *d_i;
*d_o = acc;
}
```

例子 1-5 糟糕的数据流指针类型

用于验证的 C testbench 如下：

```
int main () {
din_t d_i;
dout_t d_o;
int retval=0;
FILE *fp;
// Open a file for the output results
fp=fopen("result.dat","w");
// Call the function to operate on the data
for (d_i=0;d_i<4;d_i++) {
pointer_stream_bad(&d_o,&d_i);
fprintf(fp, "%d %d\n", d_i, d_o);
}
fclose(fp);
// Compare the results file with the golden results
retval = system("diff --brief -w result.dat result.golden.dat");
if (retval != 0) {
printf("Test failed !!!\n");
retval=1;
} else {
printf("Test passed !\n");
}// Return 0 if the test
return retval;
}
```

例子 1-6 数据流类型指针 testbench

下面让我们详细理解 volatile 数据标示符。在上面代码例子 1-6 中，我们的目的是输入指针 d_i 和输出指针 d_o 作为 RTL 接口的 FIFO 或者握手信号使用，这样可以保证如下功能：

- 当 RTL 的 d_i 端口每次读入时，上行数据输出模块输出一个新数据。
- 当 RTL 的 d_o 端口每次写出时，下行模块接收一个新数据。

然而，标准的 C 编译器编译时，对每一个指针的多次读写编译为单次读写：就 C 编译器来说，在函数执行过程中没有指示出 d_i 数据的变化，而且只于最终 d_o 写出相关（其它的多次写出在函数完成时被覆盖）。

Vivado HLS 与 gcc 编译器的行为一致，优化这些多次的读和写，最终只执行一次读和写操作。当 RTL 执行时，在读和写接口上，也仅执行一次读和写操作。

这种设计存在的一个基本问题，是仿真激励 testbench 和设计不能准确的对设计者希望的 RTL 接口行为建模：

- 设计者希望 RTL 接口在处理时，多次读入和写出数据，能够数据流模式的输入和输出。
- Testbench 只提供了单次的数据输入和返回输出数据。

C 仿真显示了如下的结果，展示了每次输入将计算 4 次，但是一旦写入，4 次都是相同的值用于累加操作，而不是 4 次不同的读入操作。

Din : Dout

```
0    0
1    4
2    8
3    12
```

这个设计可以通过设置 volatile 限定符，实现多次 RTL 接口的读入和写出。如下面代码例子 1-7 所示：

```
#include "pointer_stream_better.h"
void pointer_stream_better ( volatile dout_t *d_o, volatile din_t *d_i) {
din_t acc = 0;
acc += *d_i;
acc += *d_i;
*d_o = acc;
acc += *d_i;
acc += *d_i;
*d_o = acc;
}
```

例子 1-7 好的数据流指针类型参数

Volatile 限定符限定 C 编译器和 vivado HLS，对指针存取不做任何假设，数据是变化无常的，并且指针的存取不能够优化，对同一指针的多次存取可能读写不同的值，而不是相同不变的值。

上例 1-7 可以使用上面的仿真 testbench，但是 volatile 限定符阻止了对指针存取的优化，综合后的 RTL 设计执行 4 次输入 d_i 接口的读入和 2 次输出 d_o 接口的写出操作。

虽然我们使用了 volatile 关键字，但 testbench 和函数的代码风格（同一指针的多次读写）仍然存在问题，testbench 不能完全模型化多次不同的读入和写出操作。

在这种情况下，执行了 4 次读入，但 4 次读入的是相同的数据。同时，执行两次独立的写出，每次写出正确的数据，但是 testbench 仅仅抓到最好一次读出的数据。中间的读写过程可以通过使能 cosim_design 的 create a trace file 选项，通过保存 RTL 仿真过程中 dump 波形文件查看。

上面的例子也能应用 wire 型的约束作为接口，如果约束为 FIFO 接口，Vivado HLS 将会产生一个 RTL 的 testbench 用于数据流方式，每次读入一个新数据。但是由于从 testbench 中不能读入可用的新数据，RTL 仿真验证将会失败。这是由于 testbench 不能正确模型化多次的读写导致的。

一种比较好的方法是使用 HLS::stream<>模型化的数据流接口。

与软件不同，硬件系统本身的并行允许利用数据流方式，当数据连续的输入设计中处理同时连续的输出数据，RTL 设计在完成处理现存的数据之前，能够接收新的数据。

上面的例子说明，当用软件模型化一个存在的硬件应用时（模型化实际硬件的并行和数据流方式），模型化数据流对软件来说是很至关重要的事情。

这里有几个方法可以考虑：

- 简单的增加 volatile 限定符，如同上例中一样。Testbench 如果不能模型化读入和写出想要的数据，RTL 仿真时使用原 C 代码的 testbench 可以导致仿真失败，但是通过查看 trace file 波形文件，可以看到正确的读写操作执行。
- 修改代码显式的表示模型化读入和写出。见下面的代码。
- 修改代码，使用数据流接口类型。数据流方式的数据类型允许精确模型化使用数据流的硬件。

下面修改的代码例子 1-8 确保 4 次从 testbench 读入希望的值，并写 2 次输出。由于指针存取是串行执行并从位置 0 开始，在综合时，会产生数据流方式接口。

```
void pointer_stream_good ( volatile dout_t *d_o, volatile din_t *d_i) {
din_t acc = 0;
acc += *d_i;
acc += *(d_i+1);
*d_o = acc;
acc += *(d_i+2);
acc += *(d_i+3);
*(d_o+1) = acc;
}
```

例子 1-8 多次存取指针参数函数优化

修改 Testbench 模型化函数的实际运行，读入 4 次不同的值。新的 testbench 例子 1-9 仅模型化了单次的处理，模型化多次处理，需要增加输入数据集，并且多次调用这个函数。

```
int main () {
din_t d_i[4];
dout_t d_o[4];
int i, retval=0;
FILE *fp;
// Create input data
for (i=0;i<4;i++) {
d_i[i] = i;
}
// Call the function to operate on the data
pointer_stream_good(d_o,d_i);
// Save the results to a file
fp=fopen("result.dat","w");
for (i=0;i<4;i++) {
if (i<2)
fprintf(fp, "%d %d\n", d_i[i], d_o[i]);
else
fprintf(fp, "%d \n", d_i[i]);
}
fclose(fp);
// Compare the results file with the golden results
retval = system("diff --brief -w result.dat result.golden.dat");
if (retval != 0) {
printf("Test failed !!!\n");
retval=1;
} else {
printf("Test passed !\n");
}
// Return 0 if the test
return retval;
}
```

例子 1-9 多次存取指针参数函数的 testbench

用例子 1-9 的 testbench 验证算法，输出下面的结果，显示单次处理有二个输出，第一个输出是前 2 次的读入数据进行累加的值；第二个输出是，前 2 个输入的累加值，再加上后 2 次读入数据的累加结果。

```
Din : Dout
0     1
1     6
2
3
```

从最终的仿真结果可以看出，当函数接口使用多次指针存取时，与 RTL 仿真结果一致。

VHLS 中指针作为 top 函数参数的总结

对于基本指针类型，即函数体只进行一次的指针读写操作，指针综合为 wire 类型或者握手信号类型接口。

指针运算类型，是指函数体对指针进行偏移操作，通常情况下，由于指针偏移不是顺序执行的，综合为 ap_bus 类型接口。但当指针偏移是完全的顺序执行时，可以综合约束为 FIFO 接口类型。如果设计接口需要 ap_memory 类型，可以将带有指针运算的指针类型改写为 array 型，array 名等同于数组首地址，经过这样的修改，带有指针运算的指针类型综合为 memory 接口。

多次存取同一指针变量时，称此指针变量为多次存取指针类型。当多次存取指针作为顶层函数参数时，对其处理有三种方法。首先，如果设计本意的操作是数据流方式操作，即每次对指针的读入或者写出可能是不同的值，对这个指针变量增加 volatile 限定符即可，但需要 testbench 确保正确模型化实际数据输入输出操作。其次，修改代码风格，显式的表示多次指针存取的地址。最后，对于数据流方式操作的指针类型，直接使用 VHLS 提供的数据类型 streaming 接口的 c++模板类，hls::stream<>类型。

©Copyright 2012 Xilinx, Inc. XILINX, the Xilinx logo, Virtex, Spartan, ISE and other designated brands included herein are trademarks of Xilinx in the United States and other countries. All other trademarks are the property of their respective owners.

Xilinx Vivado HLS 中 Floating-Point（浮点）设计编码风格与技巧

作者：George Wang，Xilinx DSP 专家

Vivado 设计套件

行业首款 SoC 增强型设计套件

Vivado™ Design Suite 提供全新构建的 SoC 增强型、以 IP 和系统为中心的下一代开发环境，以解决系统级集成和实现的生产力瓶颈。

主要文档

- Vivado Design Suite Backgrounder
- Vivado IP Integrator Backgrounder

快速链接

- Vivado Design Suite 评估和 WebPACK
- IP 中心
- 支持和技术文档
- 支持的目标参考设计
- Vivado 课程
- 高层次综合课程

Vivado Design Suite 与 7 系列 All Programmable FPGA 完美结合在一起，可提供高达 3 速度等级的性能优势，同时将功耗平均降低约 35%。此外，您还可以更轻松地对更多的设计进行布线，相对同类竞争解决方案而言，其 LUT 利用率提高了 20% 以上。

尽管通常 Fixed-Point（定点）比 Floating-Point（浮点）算法的 FPGA 实现要更快，且面积更高效，但往往有时也需要 Floating-Point 来实现。这是因为 Fixed-Point 有限的数据动态范围，需要深入的分析来决定整个设计中间数据位宽变化的 pattern，为了达到优化的 QoR，并且要引入很多不同类型的 Fixed-Point 中间变量。而 Floating-Point 具有更大的数据动态范围，从而在很多算法中只需要一种数据类型的优势。

Xilinx Vivado HLS 工具支持 C/C++ IEEE-54 标准单精度及双精度浮点数据类型，可以比较容易，快速地将 C/C++ Floating-Point 算法转成 RTL 代码。与此同时，为了达到用户期望的 FPGA 资源与性能，当使用 Vivado HLS directives 时需要注意 C/C++编码风格与技巧相结合。

编码风格

单双精度浮点数学函数

```
#include <math.h>
float example(float var)
{
return log(var); //  双精度自然对数
}
```

在 C 设计中，这个例子，Vviado HLS 生成的 RTL 实现将输入转换成双精度浮点,并基于双精度浮点计算自然对数,然后将双精度浮点输出转换成单精度浮点.

```
#include <math.h>
float example(float var)
{
return logf(var); //  单精度自然对数
}
```

在 C 设计中，logf 才是单精度自然对数，这个例子 Vviado HLS 生成的 RTL 实现将基于单精度浮点计算自然对数，而且没有输入输出单双精度的互转。

浮点运算优化

我们先来看一个例子，三个从代数上看起来差不多的写法，但其在 Vivado HLS 中综合出来的是三个完全不一样的结果。

```
void example(float *m0, float *m1, float *m2, float var)
{
*m0 = 0.2 * var; // 双精度浮点乘法，单双精度类型转换
*m1 = 0.2f * var; // 单精度浮点乘法
*m2 = var / 20.0f; // 单精度浮点除法
}
```

Vivado HLS 将日 m0, m1, m2 综合成不同的 RTL 实现。

因为 0.2 是一个不能精确表征的双精度数字， 所以 m0 运算会被 Vivado HLS 综合成一个双精度浮点乘法， 并且将 var 转换成双精度， 然后将双精度乘法输出 m0 转换成单精度。

特别注意，如果希望 Vivado HLS 综合出单精度常熟，需要在常数后面加 f, 如 0.2f。这样 m1 综合成一个单精度乘法的输出。同理，m2 将被 Vivado HLS 综合成单精度除法的输出。

我们来看另外一个例子。

```
void example(float *m0, float *m1, float var)
{
*m0 = 0.2f * 5.0f * var; // *m0 = var;常数乘法被优化掉
*m1 = 0.2f * var * 5.0f; // 两个双精度浮点乘法
}
```

再来看另一个例子。

```
void example(float *m0, float *m1, float var)
{
*m0 = 0.5 * var; //
*m1 = var/2; //
}
```

m0 运算会被 Vivado HLS 综合成一个双精度浮点乘法， 并且将 var 转换成双精度， 然后将双精度乘法输出 m0 转换成单精度。

m1 运算会被 Vivado HLS 综合成简单的右移运算。所以如果用户希望实现对 var 除以 2， 就写成 m1 这种表达式，而不是 m0 的表达式。

并行度与资源复用

由于浮点运算相比整型，定点运算耗用更可观的资源。Vivado HLS 会尽量用更有效的资源来实现浮点运算，当数据的相关性及约束许可的情况下，在 Vivado HLS 中，会尽量复用一些浮点运算单元。为了说明这个，我们看一个简单的三个浮点加法例子， Vivado HLS 复用一个浮点加法器来串行实现三个浮点加法，代码及 HLS 综合结果如下。

```
void example(float *r, float a, float b,
float c, float d)
{
*r = a + b + c + d;
}
```

如果希望并行三个浮点加法来实现,可以加上 pipeline directive，由于浮点运算的精度与运算顺序有很大的关系，HLS 工具不会改变用户代码的计算顺序，这样只是个级联结构。代码及 HLS 综合结果如下，latency 是 35：

```
void example(float *r, float a, float b,
float c, float d)
{
#pragma HLS PIPELINE
*r = a + b + c + d;
}
```

如果希望并行三个浮点加法来实现,同时降低 latency，可以加上 pipeline directive 的同时将代码简单修改成加法树结构，这样的代码及 HLS 综合结果如下， latency 从 35 降低到 23：

```
void example (float *r, float a, float b,
float c, float d)
{
#pragma HLS PIPELINE
float e, f;
e = a + b;
f = c + d;
*r = e + f;
}
```

有时设计需要更高的 throughput 及更低的 latency。这时就需要提高设计的并行度。以下面例子来说明，在 Vivado HLS 就需要对 for 循环 loop 加 pipeline 与 unroll 的 directives。同时需要通过设置 a,b,r0 为 FIFO，并对其重排以提高 I/O 带宽两倍。这样 Vivado HLS 就会综合出两个浮点加法来并行实现，这是因为每个加法器计算是完全独立的。

```
void example(float r0[32], float a[32], float b[32])
{
#pragma HLS interface ap_fifo port=a,b,r0
#pragma HLS array_reshape cyclic factor=2 variable=a,b,r0
```

```
for (int i = 0; i < 32; i++)
{
#pragma HLS pipeline
#pragma HLS unroll factor=2
r0[i] = a[i] + b[i];
   }
}
```

然而，如果更多复杂的运算，或许会导致不独立的浮点运算，在这种情况下，Vivado HLS 不能重新排列这些运算的顺序，这样会导致更低的，不是所期望的复用。 下面举例来说明如何提高带有反馈浮点运算的性能。

这个例子的累加会导致 recurrence，并且通常浮点加法的 latency 大于一个时钟周期，加的 pipeline directive 并不能达到一个时钟周期完成一次累加的 throughput。

```
float example(float x[32])
{
#pragma HLS interface ap_fifo port=x
float acc = 0;
for (int i = 0; i < 32; i++)
{
#pragma HLS pipeline
acc += x[i];
}
return acc;
}
```

为了对上面例子并行展开，可以对代码如下做较小的改动，也就是拆成先部分累加，再最后累加，当然也需要对输入数据进行简单的重新排列，以获得相应的 I/O 带宽，从而达到期望的并行度。

```
float top(float x[32])
{
#pragma HLS interface ap_fifo port=x
float acc_part[4] = {0.0f, 0.0f, 0.0f, 0.0f};
for (int i = 0; i < 32; i += 4) { //  手动 unroll by 4
for (int j = 0; j < 4; j++) { //  部分累加
#pragma HLS pipeline
acc_part[j] += x[i + j];
}
for (int i = 1; i < 4; i++) { //最后累加
#pragma HLS unroll
acc_part[0] += acc_part[i];
}
return acc_part[0];
}
```

©Copyright 2012 Xilinx, Inc. XILINX, the Xilinx logo, Virtex, Spartan, ISE and other designated brands included herein are trademarks of Xilinx in the United States and other countries. All other trademarks are the property of their respective owners.

编写高效 Vivado HLS 工程 testbench 的三个要素

作者：Harvest Guo，Xilinx DSP 专家

在 C 程序的设计中，任何一个 C 程序的顶层都是 main()函数。而在 vivado HLS 的设计中，只要函数的层次在 main()函数以下，都可以被综合。但是每个 vivado HLS 工程只能指定一个 top 层函数作为输出 RTL 模块的顶层，其它和这个函数层次平行，不需要被综合的函数都可以作为 testbench 来使用。这样就带来一个问题，如何编写 vivado HLS 工程的 testbench 更高效，或者说能更好的让 HLS 工具自动重用 C testbench 验证产生的 RTL 代码就变得非常重要。

通常，在 Vivado HLS 中，好的 C testbench 设计原则是 testbench 设计和需要实现的算法函数分别保存在不同的文件中，并且充分利用头文件。Testbench 常常包含了一些 HLS 综合不支持的操作，比如通过文件的读写取得仿真数据并保存结果，或者打印一些测试结果进行分析。在头文件中，完成对 testbench 中所有的数据类型和函数的定义，以及包含共享的设计文件和函数库。

Vivado HLS 中，只能指定一个 top 层函数用于综合，top 层函数可以包含多个子函数。当需要综合多个并行层次的函数时，可以编写一个 wrapper 函数作为 top 层函数，将需要综合的多个并行函数封装起来。

C testbench 的目不仅是要验证需要综合的 top 函数功能正确（C 编译器验证环境），同时重用 C testbench 作为综合产生 RTL 代码的仿真激励，HLS 工具自动调用 C testbench 来验证 RTL 功能的一致性（C 编译器和 RTL 仿真器的协同仿真环境）。这样，编写一个好的风格 testbench 可以很好的提高设计的验证效率，如果在 HLS 综合前和综合过程中，需要修改综合函数的代码，可以用 testbench 验证，确保需要综合的 C 算法功能正确。

Vivado HLS 中推荐高效的 testbench 具有如下三个特征：

1. Testbench 代码和需要综合的 C 算法代码保存在不同的文件中（例子 1-1）。输入多个不同的数据，对需要综合的 Top 层函数执行多次的处理和验证。还可以进行 top 函数多样性的测试。Testbench 和 C 设计分开不同的文件使得 HLS 工程非常清晰，它们分别作为 test bench 文件和 source 文件加入 HLS 的 project 中。(Testbench 和要综合实现的设计文件分别保存不是 HLS 强制的，也可以保存在同一个文件中，如果保存为同一个文件，在 HLS 工程，需要指定这个文件既是 testbench 文件也是 source 文件)。

Vivado 设计套件

行业首款 SoC 增强型设计套件

Vivado™ Design Suite 提供全新构建的 SoC 增强型、以 IP 和系统为中心的下一代开发环境，以解决系统级集成和实现的生产力瓶颈。

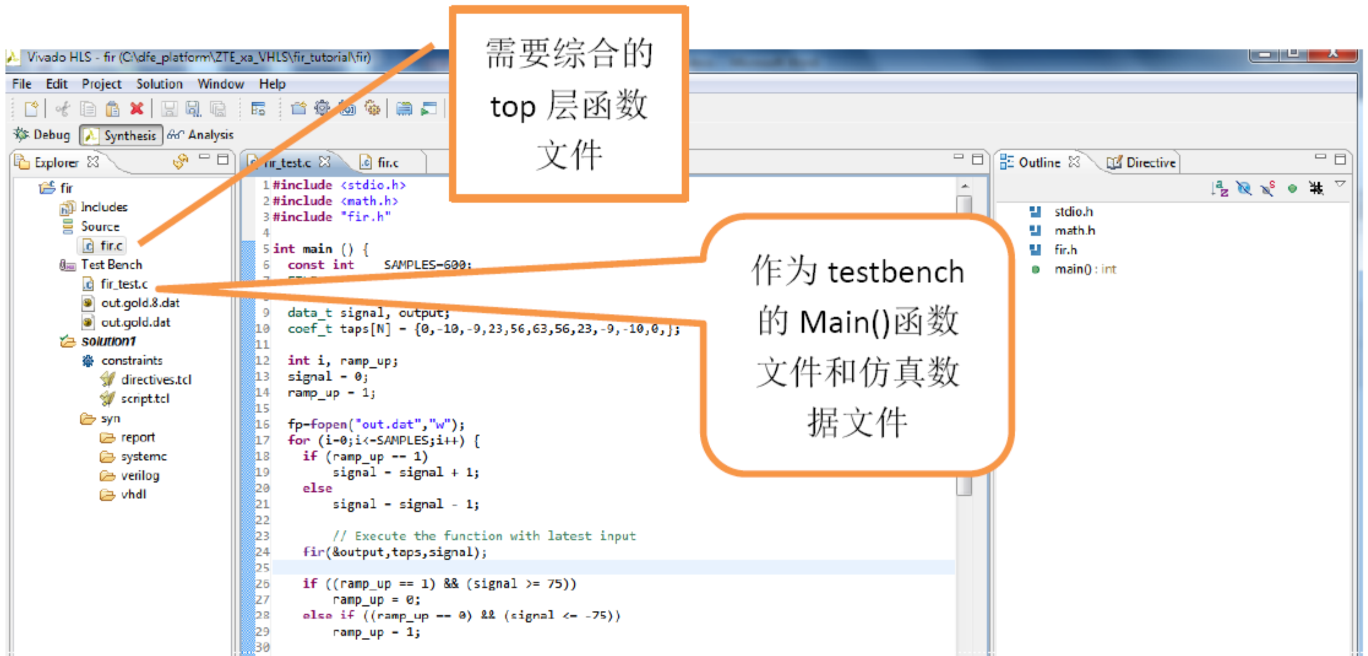
主要文档

- Vivado Design Suite Backgrounder
- Vivado IP Integrator Backgrounder

快速链接

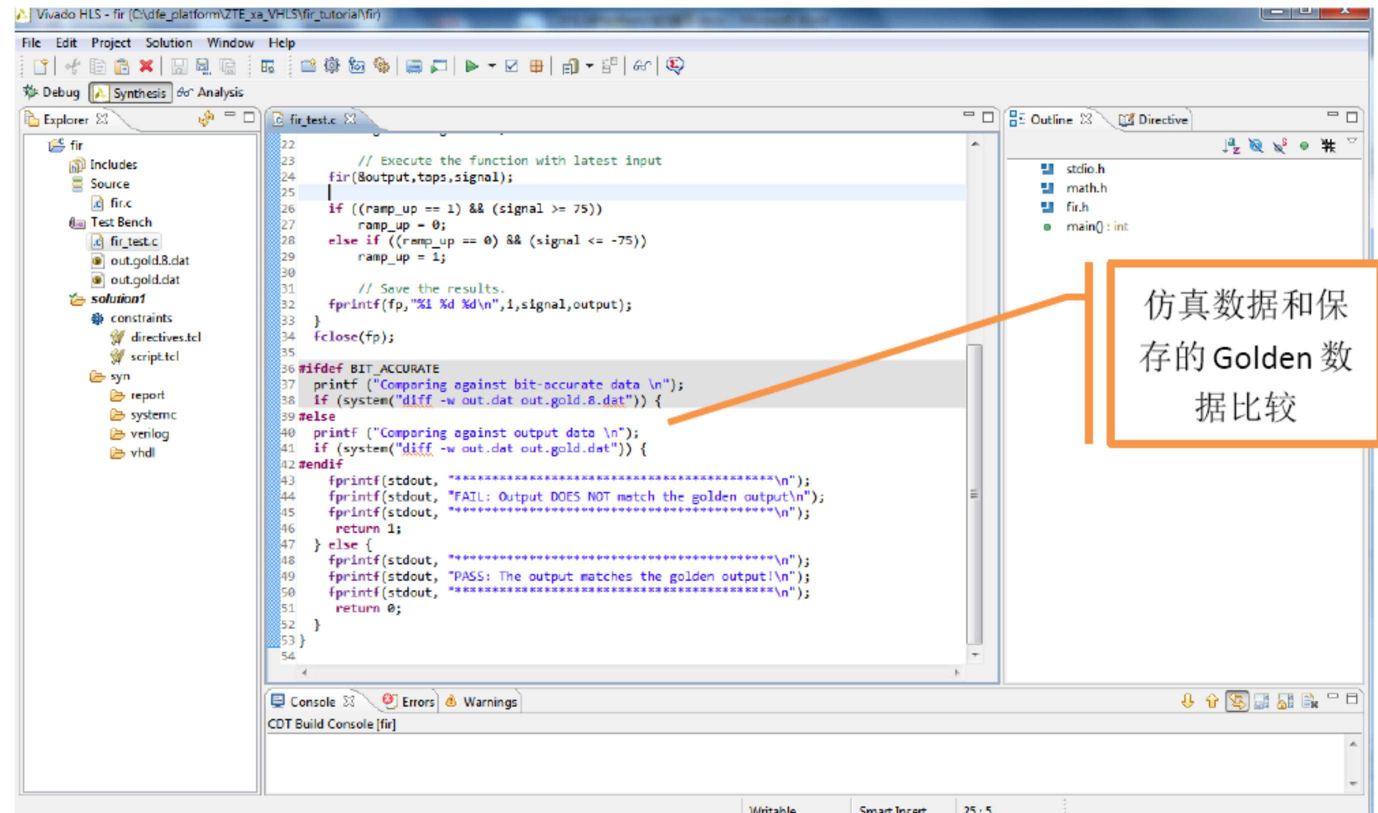
- Vivado Design Suite 评估和 WebPACK
- IP 中心
- 支持和技术文档
- 支持的目标参考设计
- Vivado 课程
- 高层次综合课程

Vivado Design Suite 与 7 系列 All Programmable FPGA 完美结合在一起，可提供高达 3 速度等级的性能优势，同时将功耗平均降低约 35%。此外，您还可以更轻松地对更多的设计进行布线，相对同类竞争解决方案而言，其 LUT 利用率提高了 20% 以上。

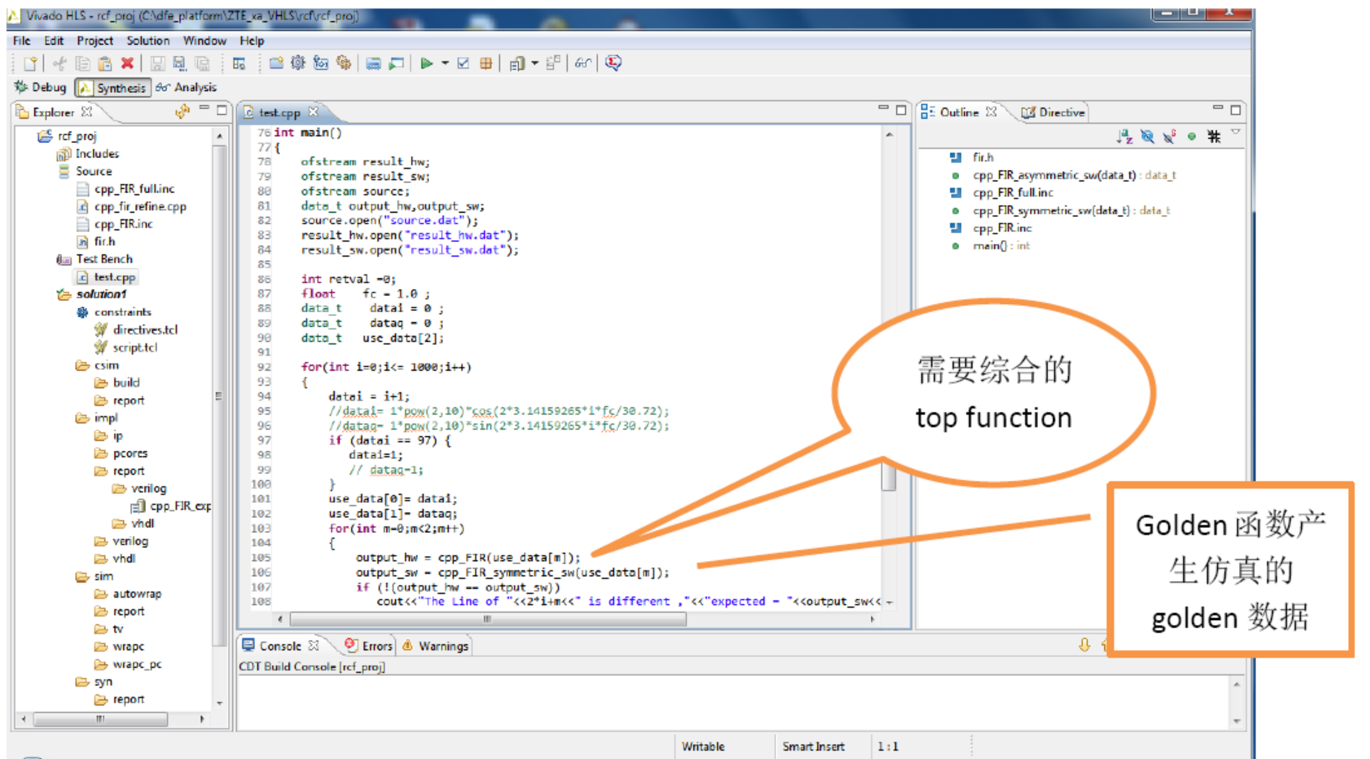


例子 1-1 testbench 函数和要综合的 top 函数保存在不同文件中

2. Testbench 具有自测试功能，testbench 调用需综合的 top 函数，仿真输出结果与已知正确的数值进行对比。已知正确的数值可以通过文件读入（例子 2-1），也可以由 testbench 的其它部分的仿真代码产生（例子 2-2）。

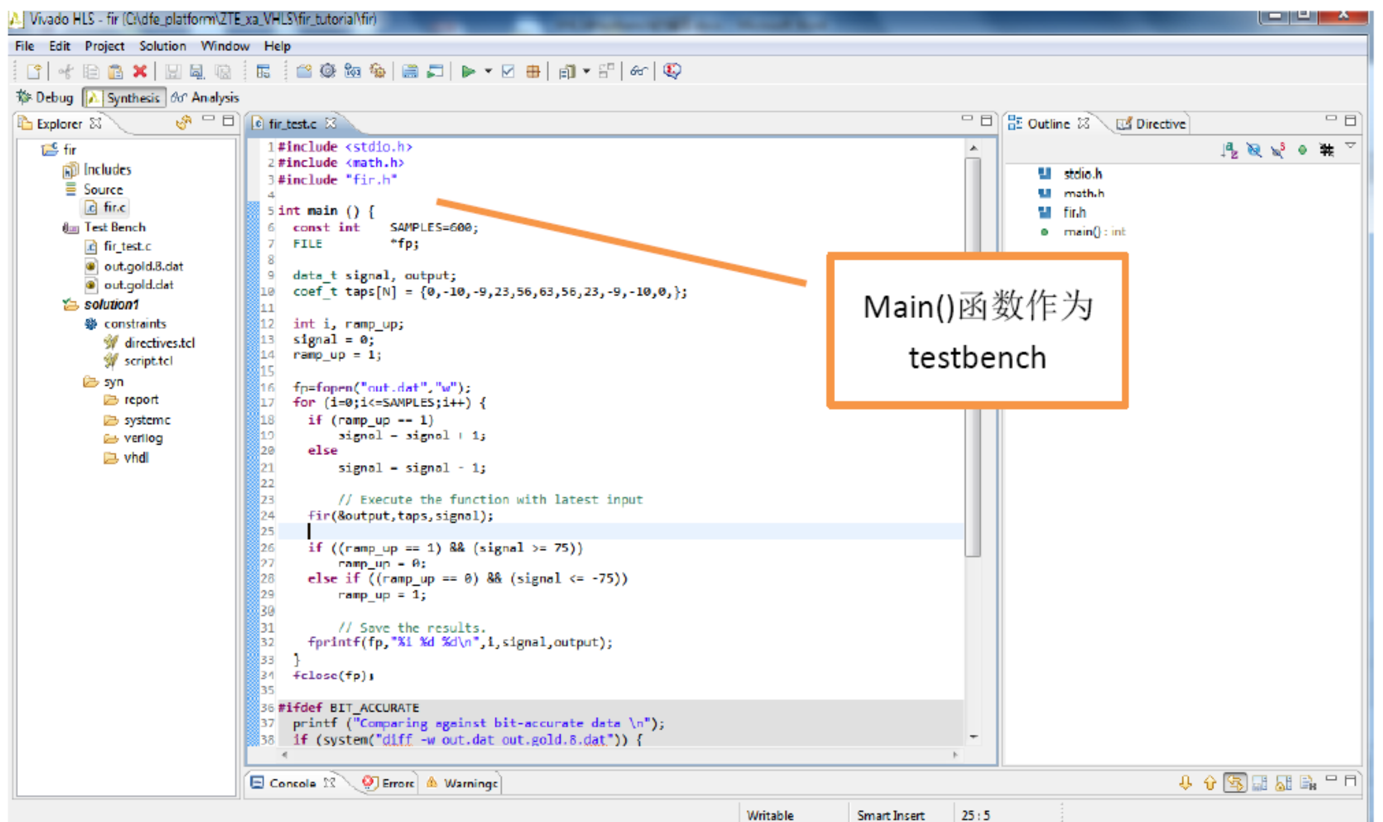


例子 2-1： testbench 仿真数据与保存的 Golden 数据比较



例子 2-2： testbench 仿真数据与 Goden 函数输出的数据比较

3. Main()函数作为 testbench 函数（例子 3-1），如果仿真 top 函数正确，main()函数返回 0 值；如果仿真不通过，返回任意非 0 的值即可（例子 3-2）。（之所以要求仿真正确返回 0 值，是因为 HLS 工具自动进行 RTL 验证时，如果 testbench 返回 0 值，HLS 认为仿真正确，而返回其它值时，HLS 报告仿真失败）。



例子 3-1： Main 函数作为 testbench

例子 3-2： Testbench 函数的返回值

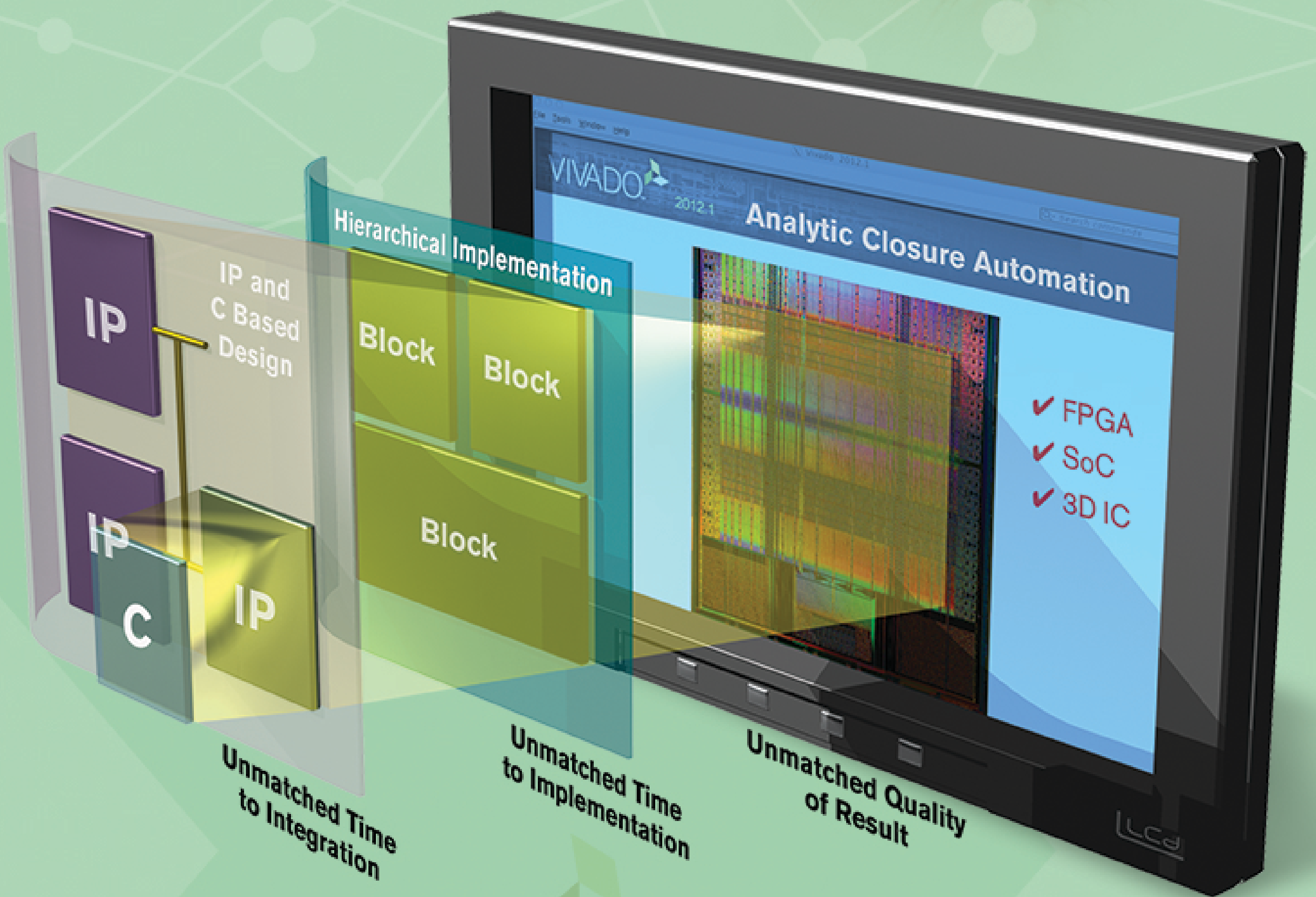
综上所述，掌握三个基本的 VivadoHLS 工程 testbench 编写原则，可以轻松书写 VHLS 高效的 testbench 仿真激励。首先，testbench 和要综合的顶层函数分别保存在不同的文件中，使得 vivadoHLS 工程简洁清楚；其次，testbench 具有自测试功能，通过调用 Golden 函数或者已保存好的 golden 数据，与仿真的结果进行比较，使得 HLS 自动重用 C testbench 来验证 RTL 仿真变得简单；最好，数据比较一致返回 “0” 值，错误不一致时返回任意非 “0” 值即可。

XILINX ALL PROGRAMMABLE™

©Copyright 2012 Xilinx, Inc. XILINX, the Xilinx logo, Virtex, Spartan, ISE and other designated brands included herein are trademarks of Xilinx in the United States and other countries. All other trademarks are the property of their respective owners.

A Generation Ahead

The First
SoC Strength Design Suite



VIVADO™

XILINX
ALL PROGRAMMABLE™