SYNOPSYS®
Predictable Success

定点算法设计流程

July 2010

Author

Holger Keding, Ph.D.
Corporate Application Engineer,

Synopsys GmbH,
Herzogenrath,
Germany

引言

“为什么这些琐碎的事会拖延项目进程？”这是某些项目经理和工程师经常问的问题，那是因为他们从来没有尝试过将浮点算法或系统转为定点的痛苦。后者是软硬件实现的参考和基础。一般来说，系统不会采用浮点实现，因为这样功耗更高，面积更大，费用更贵。

多数算法会首先开发一个浮点模型，然后在实现阶段做量化。算法量化的过程非常繁琐，所以这一阶段往往费时费力。

- 对浮点变量进行量化，一种办法是用整型数据来表示浮点数，常用整型数据包括主机内置的数据类型（int，long，等），或者一些基于内置类型的自定义数据类型。量化过程需要采用移位或者比特取位操作，写代码的时候容易出错，也很难调试。一旦出错就会耽误项目进度。
- 另一种方法是使用专门的库，这些库提供一些常用的定点数据类型用于建模和仿真。这些定点数据类型支持用户自定义数据字长以及整数部分的比特长度，提供简便的函数与操作符用于量化处理。用户在做定点转化的时候，不需要强制对浮点数据限幅，浮点和定点数据可以在算法中共存。这些特点可以使量化过程尽可能简单，也易于调试。这种方法的缺点是会大大降低仿真速度。采用这些库的定点系统相比原来的浮点系统，仿真速度会慢10-50倍。因此并不适合那些需要即时得到结果的应用。

另外，人们往往会忽略的是，量化过程其实是根据应用和实现的需要对通用的浮点模型做相应处理。我们把浮点系统看作是一个通用的IP，可以在不同的平台中使用。创建一个浮点IP是整个开发环节中的必要步骤。而通过浮点IP转化得到的定点IP并不通用。因为不同的应用场景和实现方式对模型的精确度、面积、成本等因素的考虑都不相同。因此为某一个平台量化一个的定点IP属于项目中的额外开销。

本文介绍了工程师如何使用工具和方法来应对定点化的挑战，将量化过程所带来的开销降到最低。优化的仿真技术可以可以帮助工程师解决面临的困难。

技术介绍

将系统从浮点转化为定点表示有许多方法，但本质上都是一样的。

1. 定义和简化算法结构：系统设计的开始阶段一般都是算法研究和选择，不会考虑太多算法的实现。因此系统中会包含许多理想化的算法。比如开发一个低通滤波器模型，可以先将时域信号转换到频域，去掉频谱中高于截止频率的部分，然后再换回到时域。如果采用很长的FFT/IFFT序列做时频转换，当然可以得到理想的低通滤波器算法。但是当最后实现的时候，我们多会选择低复杂度、容易量化的FIR或者IIR滤波器模型。因此在量化开始之前，我们必须首先定义并简化算法结构，这样才不至于在实现阶段一切推倒重来。要知道，量化一个频域滤波器是非常困难的。

另一个简化算法结构的方法：我们在浮点算法模型中一般会定义许多参数，这样可以使算法模型尽可能通用，保证算法IP的灵活性，满足不同应用场景和系统的要求。例如在FFT算法中可以定义的fft_length 参数—根据这个参数来分配内存、计算比例因子等。然而在实现阶段，保持这种灵活性会提高量化过程的复杂度，增加实现成本。因此假设我们已经知道了fft_length 的值，就应该去掉算法中的冗余部分，代之以常量或者查找表，降低算法量化和实现的开销。

在浮点算法中常常会采用一些函数调用，例如用sin(x) 或者 cos(x) 函数来计算变量x的正弦或余弦值。这些函数是标准C库中的一部分，不在浮点算法中。

要量化这些调用了浮点函数的算法，有两种方法：

一种是保留浮点函数调用，只是对函数参数和运行结果做量化。例如对正弦函数的计算结果采用8比特量化，其中整数部分用2比特。量化模式采用rounding：

```
y = sc_fix(sin(x),8,2,SC_RND,SC_TRN);
```

这种方法简单明了，我们可以得到一个比特精确的运行结果。但是该方法存在一个明显的缺点，就是调用函数的定点实现方式和原来的浮点形式可能会不一样。以正余弦为例，定点实现的时候会采用cordic算法、泰勒级数或者查找表来表示sin() 或 cos() 函数。因此即时采用了相同的定点格式，定点函数的运算结果与原来的数值也会有差别。这点差别可能不会对算法的性能产生影响，但是导致的结果是浮点模型无法作为比特精确的参考模型。

还有一种方法就是按照最终实现的结构来创建浮点函数。比如直接用cordic算法、泰勒级数或者查找表来表示正弦浮点函数。这样量化的函数和最终的实现完全一致，量化模型就可以作为比特精确的参考模型。

2. 确定需要量化的函数中的关键变量：如何定义这些关键变量的定点格式关系到量化的误差，或者说是相比浮点函数的性能损失。常见的关键变量包括状态变量或数组、滤波器的系数、累加值、以及输入和输出接口的值。
3. 收集关键变量的统计信息：确定关键变量的统计信息非常重要，例如值的范围、分布等等。通过浮点变量的范围可以确定所需的整数部分字长(iwl)，也就是定点格式中小数点左边的比特数。为了得到合理的统计值，需要运行足够多的仿真，激励信号也应该尽量合理选择。仿真运行的过程中，需要记录和收集所需的数据。仿真完成后，可以根据这些数据计算变量相应的iwl。该步骤通常需要在程序中手动添加相应的代码，设置统计数据的存储位置，以及将统计数据和各关键变量相关联等。
4. 确定关键变量的精确表示：当获得所有关键变量的iwls以后，就需要确定其中每一个关键变量的定点表示，或者说是字长(wl)。字长的确定只能通过采用某些指标评估算法的整体性能来实现，例如采用误比特率（BER）等。算法评估的参考标准当然是浮点模型。而对性能的约束，比如对BER的要求，则需要事先设定。要确定某个变量的wl，首先要将其转化成定点格式，或者说是用定点格式来表示。定点格式的iwl是确定值，而wl作为参数是可以调整的。用户选择不同的wl做多次仿真，将仿真结果和相应的wl做成表格，作为确定关键变量字长的依据。在多数设计流程中，这一步骤最为繁琐。对于定点数据类型，用户通常面临两难的选择。如果采用内置的数据类型，仿真速度会很快，但是代码编写和调试都非常困难。如果采用专门的定点数据格式，固然易于使用，但是仿真太慢。对关键变量的量化顺序推荐从全局到局部，也就是说从顶层的信号和接口开始。各个模块之间的接口确定以后，对模块本身的定点化可以并行进行。
5. 确定其余变量的定点格式：当关键变量的量化完成以后，还会有一些其他变量，例如局部或者临时变量，依然是浮点格式。这些变量的定点格式由关键变量决定，多数情况可以通过计算获得。如果计算不能实现，则还需要额外的仿真，重复步骤3和步骤4，来确定所需的iwl和wl。

根据最终实现的要求，上述步骤也会有调整。比如通过软件来实现算法的时候，可以依据处理器的寄存器或者内存宽度来确定关键变量的wl。但是这只能简化步骤4，其他步骤还是必须的。整个量化的过程中，步骤3和步骤4通常需要手工完成，易出错，也最耗时，是最繁琐的部分。

System Studio的解决方案

System Studio针对量化过程中面临的问题提出了一整套解决方案，包括提供独特的仿真优化技术和高效的设计工具。许多工具都是专门用于量化过程的，还有一部分则是针对设计流程的。

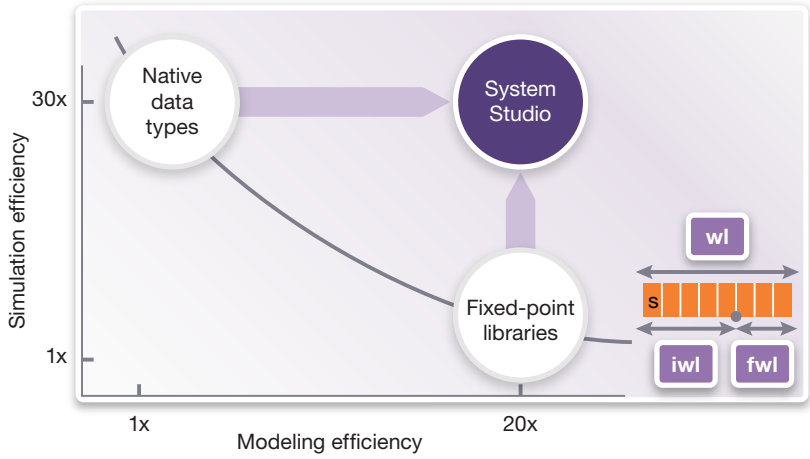

图1：System Studio支持高效建模和快速仿真

支持的数据类型

System Studio支持IEEE 1666标准中定义的SystemC定点数据类型。

数据类型sc_fix<> 和 sc_fixed<>中包含的参数有字长(wl)，整数部分字长(iwl)，量化模式（rounding，truncation等），以及溢出模式（wraparound，truncation等）。相比内置的数据类型，SystemC数据类型支持以下特点：

- 字长可以根据需要任意设定
- 字长wl和整数部分字长iwl可以不同，因此变量从浮点到定点转换时不需要调整幅度，浮点数据和定点数据联合仿真时，也不需要添加或者重写代码。算法中的所有变量都保留了原来的取值范围，这不仅减少了代码的开销，也有利于算法分析/调试。例如用12比特来表示π，我们在调试的时候看到的值依然是3.14左右，而不是一个12868的整型数。
- 数据强制类型转换时，如果遇到溢出（wrapping， clipping，等）和量化（truncation， rounding，等）处理，不需要通过比特取位、移位或者其他操作来实现。SystemC提供了大量溢出和量化模式供用户选择。
- 另外，所有的定点数据参数，包括字长、整数部分字长、类型转换模式等，都可以在仿真时修改。用户可以评估不同的字长（包括整数部分字长）和模式对算法性能的影响。
- IEEE 1666定点数据类型提供许多预定义的操作符，建模效率很高。例如做定点加法的时候，不需要手动调整操作数的小数点位置，这些会由操作符自动完成。
- System Studio针对SystemC数据类型提供了专门的代码调试支持。用户在调试的时候可以直接看到定点类型数据的数值，而不是一个SystemC类的所有细节。System Studio的这一特性为用户提供了便利。

快速定点仿真

利于SystemC的定点数据类型库做仿真会极大的降低仿真速度。因为一些诸如小数点对齐，或者存放运算的中间结果等操作都需要在运行时完成。

而System Studio对上述问题做了专门的优化。System Studio会对算法做全面的分析，将SystemC的数据类型还原成主机内置的数据类型，提前确定小数点的位置，优化所需的移位和类型转换等操作。

这大大加快了仿真的速度。此时算法中的定点数据就如同采用了整型数据加上移位和比特取位来表示一样。这意味着System Studio克服了前面提到的定点仿真的两难处境。在利用SystemC定点数据类型提高建模效率的同时，System Studio还能以类似采用内置数据类型的速度来运行仿真。正如图1所示，相比浮点算法，System Studio的定点仿真速度只会降低1-3倍（会根据数据类型、量化/溢出操作的次数以及数据大小等而不同），而不是通常情况下的30倍。

代码工具和统计数据收集

在技术介绍的第3部分，我们提到量化过程的第一步是收集数据的统计信息，包括变量范围，分布等。在System Studio中，用户不需要为此手动添加和修改代码—只要在系统中选择需要监测的端口、变量或者数组，相关的数据就会自动收集和计算。例如System Studio可以自动计算变量的iwl、均值、方差和分布。

数据类型转换

许多有经验的项目组都会尽量避免创建功能重复的IP，降低维护的成本。如果能将算法的浮点表示和定点表示都放在同一个模型中，就可以实现降低维护成本的目的。System Studio利用条件类型参数来实现这一功能。例如，在顶层模块可以将布尔参数FxSim 设为true，将整个系统用作定点仿真，否则用作浮点仿真—相关的例子参见表格3。这样，利用相同的代码可以同时表示浮点和定点模型。

有用的特性

System Studio还有许多其他的特性可以帮助用户提高量化过程的效率，例如：

- 利用System Studio基于Tcl的脚本接口可以很方便的控制仿真运行，例如分析字长参数对定点仿真性能的影响时，可以使用Tcl脚本逐个选择字长，记录结果。利用脚本也可以实现对仿真结果的后处理等。
- System Studio支持并行仿真，也就是说可以对同一个仿真采用不同的参数集，同时分配到各个主机上运行。运行结果由发起仿真的主机自动收集和同步。这也能帮助用户快速得到结果。

教程实例

在本节，我们以载波同步/QPSK接收机为例，介绍基于System Studio的定点设计流程。该实例同时包含在System Studio的培训库里，与软件一起发布。从2009.12-SP1开始，本教程中演示的每一个步骤都会包含在培训库的lab_advanced_fixed_point 库中。用户可以阅读本教程，同时在培训库查找更详细的资料，或者直接操作。lab_advanced_fixed_point 库中不仅包含了实例的原始浮点模型，还包含了量化过程中从步骤1到步骤5的中间环节。其中的README文件给出了详细的说明和仿真结果。

算法

QPSK前端接收机中的核心部分是根余弦滤波器。在教程的浮点模型中，RRCosine 模型来自System Studio的/filter 库。RRCosine 中包含了一个FFT运算、频域滤波、以及从频域到时域的IFFT运算。RRC滤波后是相位恢复，包括相位误差检测、环路滤波（NCO）以及相位补偿。最后，是QPSK解调模块和信道解码模块（差分格雷）。信道解码模块由于本身已经采用了整型数据，不需要量化，因此不需要考虑。

定义和简化算法结构

本教程中对算法结构的定义和简化分为两个步骤：

- 替换：就是用易于实现的结构替换理想的算法
- 优化：就是去掉算法中冗余的部分

替换：用FIR滤波器取代根余弦滤波器

在该步骤中，根余弦滤波器模型的算法结构会以FIR滤波器的形式来实现，以取代原来的理想频域滤波结构。仿真结果表明，在设定的标准内（BER<5*10e-6），该结构可以大大降低系统的复杂度，同时不影响性能。用户可以利用System Studio的滤波器设计工具QED来设计滤波器系数—RRCosine 模型的参数可以直接作为滤波器设计工具的输入。

- FIR filter, Filter Type: RootRaisedCosine
- Frequency Mode: Hertz
- Symbol Rate: 1
- Rolloff Factor: 0.25
- Sampling Rate: 4
- Passband Ripple: 0.1dB
- Stopband Ripple: 30dB
- Type/Windowing: Rectangular
- Number of Coefficients: 17

QED会给出浮点形式的滤波器系数，存在文件step1/float_coeffs.dat中。filter 库中的FIR 滤波器模型可以用作实现滤波器的原型。利用System Studio的profiler（或者其他）可以看出，接收机滤波器（rx_filter ）的计算复杂度明显降低（大概是RRCosine 模型的1/4），而总的BER保持不变。上述修改可以参考lab_advanced_fixed_point/step1。

优化：简化根余弦FIR滤波器模型

接下来，可以将filter 库中的FIR 滤波器模型拷贝一份到本地库中做更进一步的优化。lab_advanced_fixed_point/ 库中步骤2的主要任务是：

- 去掉存储滤波器系数的环节
- 将滤波器系数的初始化从读文件改为设置本地数组变量
- 引入额外的数据类型参数，用于为某些关键变量设置专门的定点数据格式。比如用Taccu 设置累加值的数据类型，使其能用更多的比特位来表示。

替换：用查找表替换浮点三角函数

另外，我们需要将一些浮点的函数调用，例如sin(x), cos(x), arctan2(x,y)等用查找表来替换，这种实现方式成本低，而且可以比特精确的表示。

原始的相位补偿算法利用了channel 库中的Rotation 模块，实现方式如下：

```
OutData = InData*polar(1.0,-InPhase);
```

这可以重写成：

```
OutData = InData * complex(cos(-InPhase),sin(-InPhase));
```

cos() 和 sin()函数可以利用basemath 库中的CosTable 和 SinTable 模型来替换。这些模型的参数TableSize 可以指示查找表的大小。用户可以为不同的迭代选择不同的查找表大小，记录对BER的影响。

> 提示
> 在做定点设计时需要特别注意查找表的大小。因为查找表本身已经是输出的量化结果了。如果输出的量化步骤不同，也就是低比特位和查找表的低比特位不相符，那么查找表就会引起错误。因此开始阶段应该尽可能选择大的查找表，逐步尝试

对于函数arctan2(x,y) (或者 arg(c)，c是复数)也可以使用查找表，但是其实现的方式并不如 cos()那么直接。函数arctan2(x,y)有两个参数，可以利用参数的符号确定角度的象限，同时利用arctan(x/y) 的结果计算角度的值。计算r=x/y的值时并不需要针对r和相应的角度采用非线性查找表。basemath 库中的ArcTan2Table 模型可以实现这一功能。就像CosTable 模型一样，它也有 TableSize 参数可以定义查找表的大小。

经过上述替换以后，接收机算法中所有的外部变量都以比特精确的形式存在。换句话说，对算法的量化就可以开始了。

确定算法中的关键变量

量化过程中的关键变量很容易确定：

- 接口变量，就是输入/输出端口及其浮点参数
- 其他关键变量—包括PRIM模型中的数组：关键变量是算法中最重要的部分。关键变量的精确度，或者说是其定点格式的选择，对算法的性能起了决定作用。它们的定点格式不像某些临时变量或者中间结果，无法由其他变量推导而来。常见的关键变量包括累加器的值，匹配滤波器中FIR模型的系数等。

> 提示：
> 为了简化量化过程，关键变量在仿真过程中要保证可见，也就是说应该在System Studio的PRIM模型头文件中声明。如果关键变量只是在main_action()内部声明为局部变量，例如滤波器的累加值，就需要修改代码。

收集关键变量的统计信息

System Studio可以根据收集到的统计信息计算每一个关键变量的整数部分字长（iwl）。选择包含关键变量的实例，在context menu中选中Highlight Instance in Hierarchy Tree，就可以打开设计的Hierarchy Tree 视图，其中选中的实例会高亮显示，见图2。

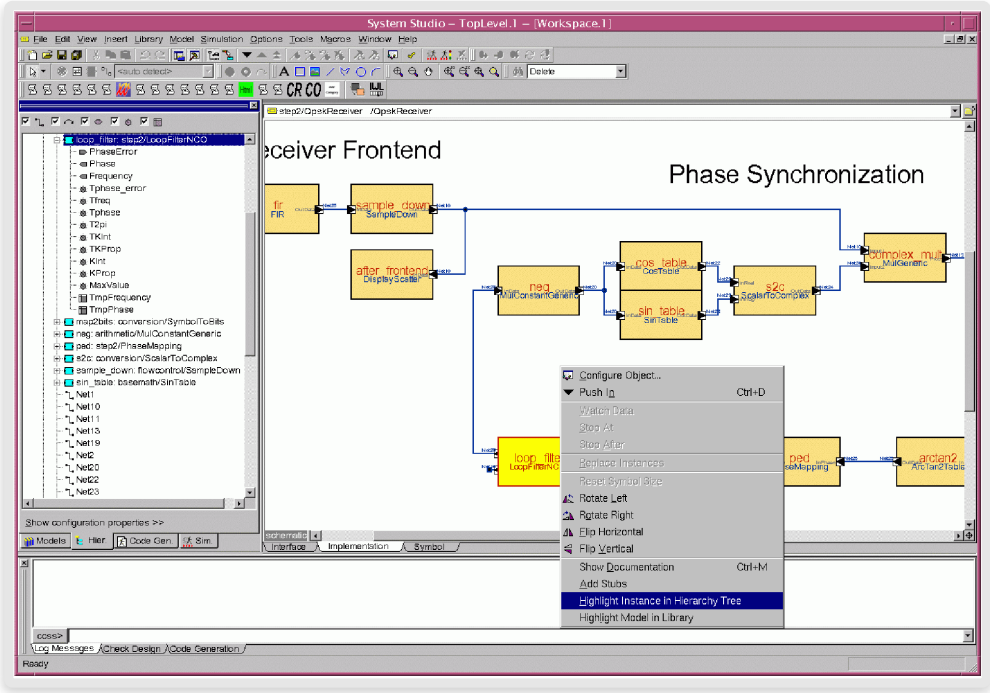

图2：在System Studio的分层设计视图中选择一个实例

在Hierarchy Tree 视图中选择Show Configuration properties，打开模型的属性页。将相应关键变量的collect_stat属性值设为true。仿真完成后System Studio会在仿真目录下生成一个数据库文件（.ocd），数据库文件可以在hierarchy tree视图中显示：

- 在Hierarchy Tree 视图中选择Configuration Properties>Merge… from the context menu（鼠标右键），在activation/iteration 目录下找到 design_properties.ocd，选择OK。
- Configuration Properties对话框中会显示收集到的数据，如图2。对量化来说最重要的数值是变量的最大值和最小时，以及定点格式的推荐值。一般来说，推荐的定点格式是个保守值，就是说推荐的定点格式会包含浮点取值的完整范围。例如，对loop_filter模型的PhaseError输入端口，System Studio给出的推荐值是sc_fixed<26,1>，整数部分的字长为1（iwl=1），只保留了正负信息。而总字长（wl=26）一般来说都可以减少，并不会引入太大的量化误差。所以接下来的仿真需要确定实现成本和算法性能的折衷。System Studio还会为每个变量给出数值分布的直方图，用户可以利于数据视图工具DAVIS查看。
- 如果程序有多个测试向量集，必须对每个测试向量收集统计信息并生成数据库文件，再把所有的结果合并在一起。只有这样才能保证统计数据能真实的反映系统要求。

提示：
在引入定点数据类型之前，最好先按照定点实现的要求修改算法结构。在浮点模型中，端口/信号和变量基本都是float或者double型，但是在定点模型中，一般会定义多种定点格式。比如FIR滤波器中，累加器的类型为Taccu ，输入/输出端口为Tsample，系数为Tcoeff 。这样可以简化量化和将算法映射到软硬件平台的流程。

有些类型参数只会影响模型内部的计算结果，有些则会影响全局，比如输入输出端口的数据类型。模型的端口必须定义相同的数据类型，才能实现信号的传输。DFG模型中最好不要为每个连接都定义类型参数，这样容易出错。可以利于分层的类型参数，就像step2/CarrierRecoveryTop 中的类型参数Tsample 和Tphase 那样，参看图3的截屏。

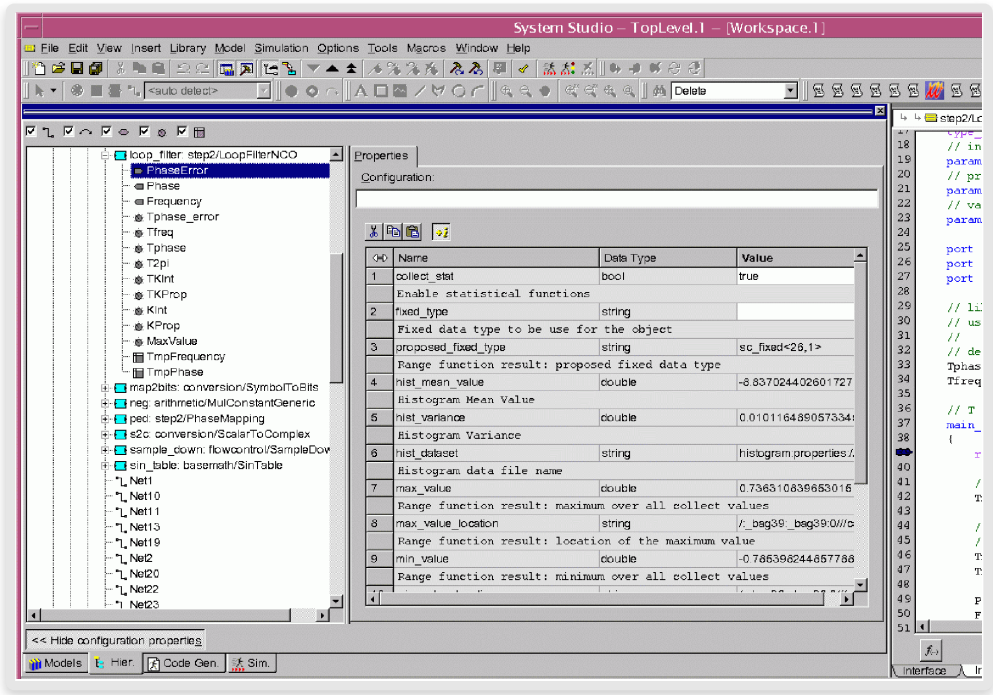

图3：仿真结果合并后的属性配置对话框

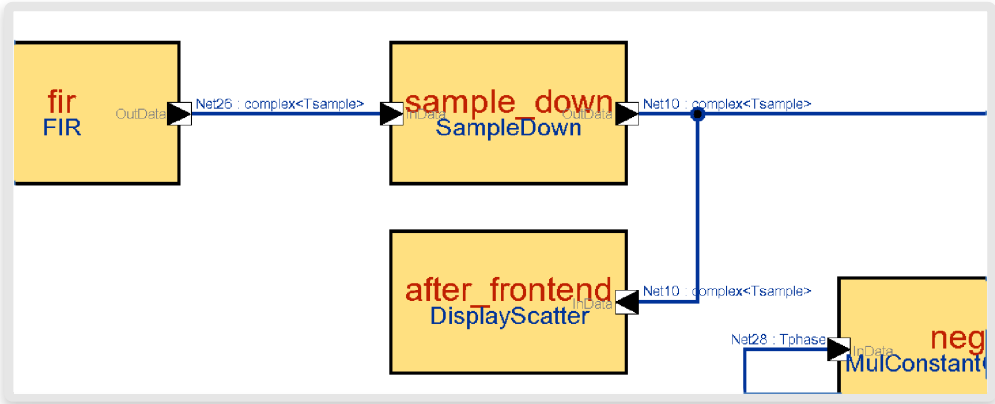

图4：使用类型参数，保证DFG模型中的类型一致性

> 提示：
> SystemC中的定点类型有两种： sc_fix[ed] 和 sc_fixe[ed]_fast。前者可以定义任意的字长，后者则是用一个double型来存储定点值，固定为52比特精度。如果该精度满足要求，用户应该优先使用sc_fix[ed]_fast类型，因为这比采用sc_fix[ed]的仿真快2x左右。

确定关键变量的精度

下一步是将关键变量从浮点类型换成定点类型，定点数据的字长作为参数，由仿真来确定。我们可以定义一个系统性能指标，比如BER，来评估不同字长对系统的影响。见图4。

用户可以利用System Studio的脚本功能自动评估字长和BER之间的关系。比如在QPSK接收机模型中设置顶层参数wl，利用脚本来遍历wl的取值并自动启动仿真。

Parameter Name	Data Type	Value
Tsample	type_param	sc_fix_fast(wl,3,SC_RND, SC_WRAP)

表1：利用字长参数wl评估不同字长对BER的影响

仿真控制文件就是一个脚本，用户可以通过该文件对仿真设置不同的参数组合。接下来的例子中，仿真控制文件设置字长的范围从4到16，而仿真点数相对较少。

```
set_value NumSamples        1e6
for {set wl 4} {$wl <= 16} {incr wl 1} {
  set_value wl $wl
  run_iteration
}
```

要使得期望的BER值保持在10-6的范围，这么少的仿真点数并不可靠。这样做的目的只是为了很快获得一个大致的范围，告诉用户应该从哪个字长值开始做长仿真。从仿真结果可以看出，字长小于6时BER相对较高。所以可以设定字长的取值范围为6到10。

```
netbatch: job 6 has been submitted to grd system with id 4638550
netbatch: job 7 has been submitted to grd system with id 4638551
netbatch: job 8 has been submitted to grd system with id 4638552
netbatch: job 9 has been submitted to grd system with id 4638553
netbatch: job 10 has been submitted to grd system with id 4638554
wl = 6 : BER: 4.4e-07
wl = 7 : BER: 0.0
```

当使用10e8个数据点做长仿真时，建议使用System Studio的Netbatch工具，将不同的仿真同时提交到服务器组中并行运行。支持的任务提交系统包括Sun Microsystems的SunGrid Engine (也叫 Gridware)，还有Platform Computing的LSF。以Gridware为例，要并行提交任务，需要在Start Options对话框中选择Gridware，然后在SCF文件的第一个循环中设置字长参数，提交仿真，在第二个循环中等待并读取仿真结果。使用Gridware的SCF文件示例具体可以参照lab_advanced_fixed_point 库中step3 和 step4 包含的CarrierRevoveryTopNB.scf。下面给出了主要部分的内容：

```
set_value NumSamples        1e8
set simname $env(SIM_NAME)
…

set wl_min 6
set wl_max 10

for {set wl $wl_min} {$wl <= $wl_max} {incr wl 1} {
  set_value wl $wl
  nb_submit_iteration $wl
}

for {set wl $wl_min} {$wl <= $wl_max} {incr wl 1} {
  puts -nonewline "wl = $wl : "
  nb_catch_iteration $wl
  set biterrors($wl) [show_value /$simname/ber/biterrors]
  set counter($wl) [show_value /$simname/ber/counter]
  puts "BER: [expr $biterrors($wl) / double( $counter($wl) )]"
}
```

在shell中可以看出所有的任务都会立即提交，任务完成后，结果会打印：

```
netbatch: job 6 has been submitted to grd system with id 4638550
netbatch: job 7 has been submitted to grd system with id 4638551
netbatch: job 8 has been submitted to grd system with id 4638552
netbatch: job 9 has been submitted to grd system with id 4638553
netbatch: job 10 has been submitted to grd system with id 4638554
wl = 6 : BER: 4.4e-07
wl = 7 : BER: 0.0
```

从仿真结果可以看出，当类型Tsample 的字长大于7时并不会显著改善系统的BER。此时其他的数据类型还都是浮点，所以选择最小的字长一也就是7一并不可取。实际上还有许多变量需要量化，每一次量化都会引入额外的误差并累积。这将导致先量化的变量字长太短，后量化的变量需要很大的字长才能弥补累积的误差损失。因此建议在选择字长时比仿真结果多出一两个比特，这会使得整个系统的字长较小。当字长确定以后，参数wl就可以用定值取代：

Parameter Name	Data Type	Value
Tsample	type_param	sc_fix_fast(8,3,SC_RND, SC_WRAP)

表2：将字长参数wl用常数8替代

重复上述步骤，对其他端口和变量进行量化：首先做一个短仿真选择字长范围，然后使用长仿真确定结果。如果仿真中的定点变量较多，推荐使用System Studio的定点快速优化技术来加快仿真速度。用户只需按照图5所示，选择Code Generation 页中的Fast fixed-point 选项。该选项将SystemC的定点数据类型映射成主机的内置数据类型，减少了运行时的任务和存储开销。仿真中的定点类型用的越多，优化的效果就越明显。

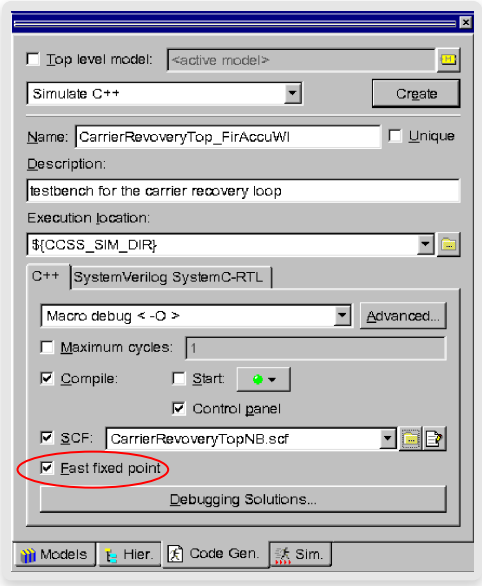

图5：利用DAVIS显示字长和BER的关系

提示：
对变量进行量化时，可以选择不同的量化模式，也就是说对最低比特（LSB）有不同的处理方式。常用的模式有rounding （SC_RND） 和truncation （SC_TRN）。SC_TRN 是SystemC定点数据类型的默认处理模式。truncation的缺点是总是引入负的量化误差。无论是无符号数还是二进制补码表示的数，最低位的截取都会使值变小。

提示：

对复杂系统的量化是一个递归的过程，量化会引入许多非线性的影响，令人困惑。而下面这些特性对用户来说用处很大：

1. 能够保存设计结果，记录获得的变量字长和整数部分字长。
2. 用户有机会撤销修改，或者选择曾经正确的状态
3. 可以比较系统浮点模型与相应定点（或者部分定点）模型之间的性能差异

为了满足上述要求，以下的一些方法会有帮助：

1. 当运行长仿真评估某一变量的字长时，首先在仿真的Code Generation 面板中设定一个易识别的工程名，比如在确定接收机FIR滤波器的系数字长时，取工程名为CarrierRecovery_RecFirCoeffWl—这也是生成的仿真目录名。仿真结束后，不要删除这个目录，因为里面包含了很多有用的信息，比如仿真结果、用来设置参数的仿真控制文件等。万一以后有需要，可以检查这些结果及设置。
2. 利用版本控制软件，如CVS 或者 ClearCase 等来保存相关的中间结果。也就是说，对所有的接收机信号和端口量化完成以后，使用可识别的标签或者标题做check-in，并提供相关的文档。这样万一以后设计出错，可以利用以前的正确版本做参考，或者直接到恢复以前的版本。当然也可以对每个正确的中间步骤手动拷贝存档，但是这样很不方便。
3. 保持模型在System Studio中的兼容性，也就是说浮点模型和定点模型可以自由转换。为了保持不同模型的一致性，要求浮点和定点模型共享同一份源代码。这可以通过在顶层模型中定义布尔参数（如 FxSim），并在实例的Configure Object 对话框中设置参数值来实现。

Parameter Name	Data Type	Value
Tsample	type_param	FxSim?sc_fix_fast(8,3,SC_RND, SC_WRAP):float

表3：利用条件类型参数实现浮点仿真和定点仿真的转换

对中间结果做量化

当所有的关键变量量化完成以后，系统基本上已经是比特精确了。还剩余一些需要量化的中间结果，包括除法操作等。另外有一些中间变量可能用了过多的字长，这会增加实现的成本。

对大多数操作来说—除了除法—中间结果的字长（wl）、整数部分字长（iwl）和小数部分字长（fwl，wl＝iwl+fwl）可以从操作数推导而来（a和b是有符号数）：

- a ± b = c: iwl(c) = max(iwl(a),iwl(b)) +1; fwl(c) = max(fwl(a),fwl(b)
- a * b = c: iwl(c) = iwl(a)+iwl(b); fwl(c) = fwl(a) + fwl(b)

对除法a / b = c来说，如果b很小的话，整数部分字长就会很大，所以即时a和b的范围已知，也很难确定定点格式。最安全也是最常用的方法是将表达式拆分。以LoopFilter中的表达式为例：

```
TmpPhase = TmpPhase - MaxValue * floor(TmpPhase/MaxValue + 0.5);
```

可以拆分成两个表达式，定义变量（比如phase_by_2pi）作为一个中间结果，利用System Studio来仿真和确定它的范围和整数部分字长（iwl）：

```
phase _ by _ 2pi = TmpPhase/MaxValue;
TmpPhase = TmpPhase - MaxValue * floor(phase _ by _ 2pi + 0.5);
```

当iwl确定以后，利用强制类型转换将除法的结果以字长wl为参数来表示，然后再利用仿真确定wl的值：

```
phase _ by _ 2pi = sc _ fix(TmpPhase/MaxValue ,wl,2);
TmpPhase = TmpPhase - MaxValue * floor(phase _ by _ 2pi + 0.5);
```

多数情况并不需要那么复杂。现在的例子中MaxValue是个常数（2π），除法可以改为乘以MaxValue-1。表达式的值与类型为Tphase 的变量TmpPhase 的字长及整数部分字长都应该接近。因此除法的结果采用相同的数据类型比较合理。

```
Tphase phase _ by _ 2pi = TmpPhase/MaxValue;
TmpPhase = TmpPhase - MaxValue * floor(phase _ by _ 2pi + 0.5);
```

该假设需要通过仿真来验证。本例中的仿真结果表明，中间变量采取与最后结果相同的定点数据格式并没有使BER有改变。

类似对中间结果的优化方法也可以用于加法、减法和乘法操作。SystemC定点操作的结果都会存在一个类型为sc_fxval的数据中，该数据可以保持操作结果的所有精度。中间结果实际使用了多少比特可以通过计算得出，看下面的表达式

```
TmpPhase = TmpPhase - MaxValue * floor(phase _ by _ 2pi + 0.5);
```

定义Tphase = sc_fix(8,4) 和 T2pi = sc_ufix(6,3)，表达式可以改写成：

```
TmpPhase -= sc_fix(MaxValue*sc_fix(4,0,phase_by_2pi,SC_RND,SC_WRAP),9,3);
```

可以用上述比特精确的表达式作为最终实现的参考模型，或者尝试进一步减少所需的字长。加减法操作中，操作数一般都采用相同的定点格式，因此这里可以试着减少第二个操作数的字长。

```
Tphase tmp_prod = MaxValue * sc_fix(4,0,phase_by_2pi,SC_RND,SC_WRAP);
TmpPhase = TmpPhase - tmp_prod;
```

同样的，必须验证减少的字长会不会对性能指标，比如BER，产生很大影响。

基准结果

如果fast fixed-point 选项在做定点仿真时没有选中，则采用IEEE 1666定点格式的SystemC仿真与浮点仿真相比，速度会减慢很多。具体的结果可以参看图6。

System Studio的fast fixed-point 选项可以改善定点仿真的性能。为了更清楚的显示这一点，我们采用了108个数据点来仿真 完整的载波恢复模型，包括用发射端、信道和接收机。其中发射端与信道采用浮点模型，接收机则既有浮点模型又有定点模型。

仿真的主机采用了64位的Intel Xeon处理器，操作系统为RedHat Enterprise Linux 5.3，主频为2666MHz。

	Floating-Point [sec (min:sec)]	Fast Fixed-Point [sec (min:sec)]	Fixed-point [sec (h:min:sec)]	
64-bit mode	88 (1:28)	107 (1:47)	using sc_fix_fast types	3328 (0:55:28)
			using sc_fix types	5901 (1:38:21)
32-bit mode	107 (1:47)	184 (3:04)	using sc_fix_fast types	12664 (3:31:04)
			using sc_fix types	24691 (6:51:31)

表4：浮点仿真代码、System Studio fast fixed-point仿真代码与没有优化的定点代码的性能比较结果

仿真结果表明，在64位机器上，采用fast fixed-point选项的System Studio仿真比没有经过优化的SystemC定点仿真快31倍。而与浮点仿真相比，只慢了22%左右，这里的速度损失主要是源自定点代码中额外的一些类型转换、移位或者舍入操作。在32位机器上，差别更明显。fast fixed-point代码比没有优化的代码快了69倍，比浮点代码慢了72%。

为了使测试结果更全面，我们也比较了支持无限精度的sc_fix数据类型仿真。本教程中很大的字长并不实用，用有限精度的定点类型sc_fix_fast已经足够了。仿真结果清楚的显示了两种类型的性能差别。如果定点的字长和整数部分字长在代码生成时就已知，那么System Studio的fast fixed-point优化会将所有的定点类型都映射为整型数据。因此不论对于有限精度还是无限精度的定点类型来讲，本例中产生的fast fixed-point仿真代码其实是一样的。

实验结果对比清楚的说明了仿真优化的价值。对于某些复杂系统仿真，如果选中了System Studio的fast fixed-point优化选项，一般一两个小时就能得到结果。没有的话往往要第二天才行。

总结

本文描述了用户在算法定点化流程中经常会面临的困境，比如是利用主机内置的整型数据来表示定点数，还是选择专门的定点数据类型以及相应的操作和类型转换等。我们讨论了不同方法的优缺点，然后展示了System Studio的fast fixed-point优化技术如何将两种方法的优点结合起来，为用户提供简易的建模调试方法，同时又保证出色的仿真性能。

而随后的教程主要介绍了算法从浮点表示转化为定点过程中的关键步骤。教程中包含了一些针对定点设计的有用提示。所有的例子都可以在System Studio的培训资料中找到。

最后我们给出了教程实例的运行结果，对System Studio的速度优势有了一个直观的了解。System Studio的fast fixed-point定点仿真与采用IEEE 1666 SystemC库的定点仿真相比，速度提高了31-69倍。而与浮点仿真相比，速度降低了22-72%。由此可见，System Studio可以大大加快设计流程，避免耽误项目进度。