HyperLink 编程和性能考量

冯华亮/Brighton Feng
Communication Infrastructure

摘要

HyperLink 为两个 KeyStone 架构 DSP 之间提供了一种高速，低延迟，引脚数量少的通信接口。HyperLink 的用户手册已经详细的对其进行了描述。本文主要是为 HyperLink 的编程提供了一些额外的补充信息。

同时本文还讨论了 HyperLink 的性能，提供了在各种操作条件下的性能测试数据。对影响 HyperLink 性能的一些参数进行了讨论。

文章的最后附上对应本文的应用代码。

目录

1 HyperLink 介绍

HyperLink 为两片 DSP 之间提供一种高速、低延迟，引脚数少的通信连接接口。

HyperLink 的设计速度最高速率支持 12.5Gbps，目前在大部分的 KeyStone DSPs 上, 由于受限于 SerDes 和板级布线，速度接近为 10Gbps 。HyperLink 是 TI 专有的外设接口。相对于用于高速 Serdes 接口的传统的 8b10b 编码方式，HyperLink 减少了编码冗余，编码方式等效于 8b9b。单片 DSP 为 HyperLink 提供 4 个 SerDes 通道，所以 10Gbps 的 HyperLink 理论吞吐率为 10*4*(8/9)= 35.5Gbps= 4.44GB/s.

HyperLink 使用了 PCIE 类似的内存映射机制，但它为多核 DSP 提供了一些更灵活的特性。本文将会使用几个范例来详细解释这一点。

本文还讨论了 HyperLink 的性能，提供了在各种操作条件下的性能测试数据。对影响 HyperLink 性能的一些因素进行了讨论。

2 HyperLink 配置

本节提供了一些配置 HyperLink 模块的补充信息。

2.1 Serdes 配置

Serdes 必须配置成期望的链接速度。 图 1 表示了输入参考时钟和输出时钟之间的关系。

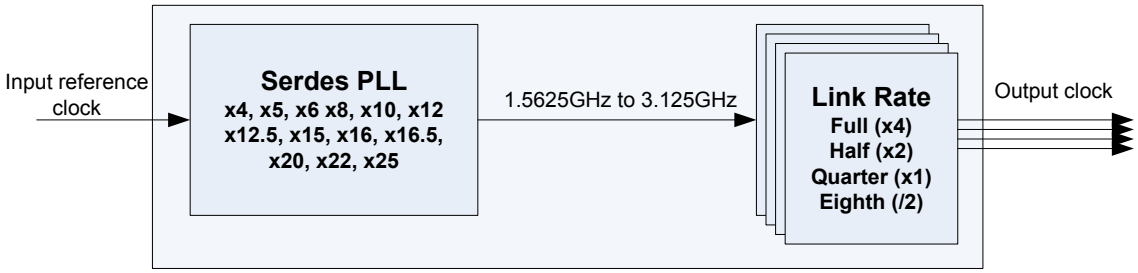

图 1 HyperLink Serdes 配置

输入参考时钟建议限制在 156.25MHz ~312.5MHz 范围内。Serdes PLL 的倍频系数必须合理配置生成的内部时钟（internal clock） 限制在 1.5625GHz ~ 3.125GHz 范围内。

最后的链接速度由内部时钟(internal clock)驱动，通过 link rate 配置来得到。

2.2 HyperLink 存储映射配置

HyperLink 的存储映射非常的灵活。 HyperLink 的用户手册对此作了详细的描述。本节将用两个例子来详细的解释它。图 2 是第一个例子。

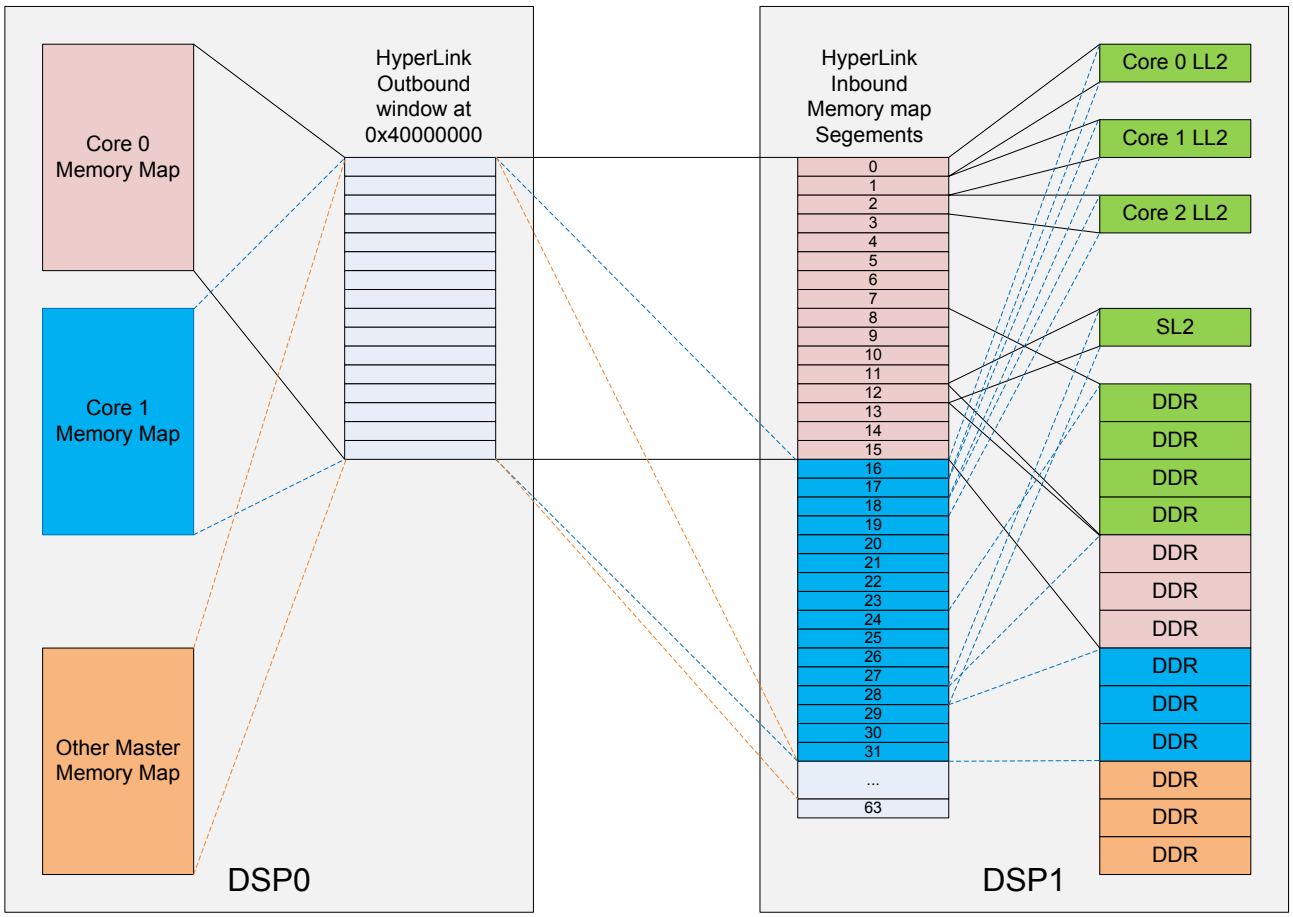

图 2 通过 HyperLink 窗口 映射到远端不同类型的存储空间

在这个例子里面， DSP1 的存储空间映射到了 DSP0 的存储空间窗口 0x40000000~0x50000000

DSP0 可以访问 DSP1 的所有内存空间，包括 LL2，SL2，DDR，就像访问自己的本地的存储空间一样。在 DSP0 上，所有的 Master 都可以通过以 0x40000000 起始的 Outbound 窗口地址来访问 DSP1 的存储空间，但是不同 master 事实上可能访问到 DSP1 上不同的存储空间。原因是 HyperLink 发送侧传输数据时，会将 PrivID 一起传输。接受侧通过 PrivID 值，可以建立不同的地址映射表，

对 DSP0 与 DSP1 的内存映射关系总结在下表(表 1)。

表 1 本地不同内核（core0 与 core1）与远端存储空间的映射范例

本地地址 (容量)	远端 DSP 的地址	
	本地内核 0 映射到的远端地址	本地内核 1 映射到远端地址

0x40000000 (16MB)	0x10000000 (LL2)	0x10000000 (LL2)
0x41000000 (16MB)	0x11000000 (LL2)	0x11000000 (LL2)
0x42000000 (16MB)	0x12000000 (LL2)	0x12000000 (LL2)
……	……	……
0x48000000 (16MB)	0x88000000 (DDR)	0x88000000 (DDR)
0x49000000 (16MB)	0x89000000 (DDR)	0x89000000 (DDR)
……	……	……
0x4C000000 (16MB)	0x0C000000 (SL2)	0x0C000000 (SL2)
0x4D000000 (16MB)	0x8C000000 (DDR)	0x8F000000 (DDR)
0x4E000000 (16MB)	0x8D000000 (DDR)	0x90000000 (DDR)
0x4F000000 (16MB)	0x8E000000 (DDR)	0x91000000 (DDR)

通过上表的配置，可知

当 DSP0 的 core 0/1 访问 0x40800000，它事实上 访问了 DSP1 上的 LL2 地址空间。

当 DSP0 的 core0 访问 0x4D000000，它事实上访问了 DSP1 上 DDR 的地址空间 0x8C000000

当 DSP0 的 core1 访问 0x4D000000 ，它事实上访问了 DSP1 上 DDR 的地址空间 0x8F000000

与本文档对应的范例工程将 HyperLink 配置成上述的内存映射关系。下面是关键部分的配置代码。

```
/*----------------Initialize HyperLink address map----------------------*/
/*use 28 bits address for TX (256 MB) */
HyperLink_cfg.address_map.tx_addr_mask = TX_ADDR_MASK_0x0FFFFFFF;

/*overlay PrivID to higher 4 bits of address for TX*/
HyperLink_cfg.address_map.tx_priv_id_ovl = TX_PRIVID_OVL_ADDR_31_28;

/*Select higher 4 bits of address as PrivID for RX*/
HyperLink_cfg.address_map.rx_priv_id_sel = RX_PRIVID_SEL_ADDR_31_28;

/*use bit 24~29 (4 bits (24~27) MSB address, 2 bits (28~29)
remote PriviID) as index to lookup segment/length table*/
HyperLink_cfg.address_map.rx_seg_sel = RX_SEG_SEL_ADDR_29_24;

/*map local memory into the same segments for all PrivID (remote masters)*/
for(iSeg= 0; iSeg<8; iSeg++)
for(iPrvId=0; iPrvId<4; iPrvId++)
{
  HyperLink_cfg.address_map.rx_addr_segs[(iPrvId<<4)|iSeg].Seg_Base_Addr=
    0x10000000+iSeg*0x01000000;
  HyperLink_cfg.address_map.rx_addr_segs[(iPrvId<<4)|iSeg].Seg_Length=
    RX_SEG_LEN_0x0_0100_0000;
```

```
}

/*map a part of DDR3 into the same segments for all PrvID (remote masters)*/
for(iSeg= 8; iSeg<0xC; iSeg++)
for(iPrvId=0; iPrvId<4; iPrvId++)
{
  HyperLink_cfg.address_map.rx_addr_segs[(iPrvId<<4)|iSeg].Seg_Base_Addr=
    DDR_SPACE_ACCESSED_BY_HYPERLINK+(iSeg-8)*0x01000000;
  HyperLink_cfg.address_map.rx_addr_segs[(iPrvId<<4)|iSeg].Seg_Length=
    RX_SEG_LEN_0x0_0100_0000;
}

/*map SL2 into same segement for all PrvID (remote masters)*/
for(iPrvId=0; iPrvId<4; iPrvId++)
{
  HyperLink_cfg.address_map.rx_addr_segs[(iPrvId<<4)|0xC].Seg_Base_Addr=
    0x0C000000;
  HyperLink_cfg.address_map.rx_addr_segs[(iPrvId<<4)|0xC].Seg_Length=
    RX_SEG_LEN_0x0_0100_0000;
}

/*map different DDR3 sections into the segements
of different PrvID (remote masters)*/
for(iPrvId=0; iPrvId<4; iPrvId++)
for(iSeg= 0xD; iSeg<=0xF; iSeg++)
{
  HyperLink_cfg.address_map.rx_addr_segs[(iPrvId<<4)|iSeg].Seg_Base_Addr=
    DDR_SPACE_ACCESSED_BY_HYPERLINK+0x04000000+(iPrvId*3+iSeg-0xD)*0x01000000;
  HyperLink_cfg.address_map.rx_addr_segs[(iPrvId<<4)|iSeg].Seg_Length=
    RX_SEG_LEN_0x0_0100_0000;
}
```

对于一些简单的应用，可能只是想访问远程 DSP 的 DDR 空间,那么下面的例子用于这种情况。存储映射关系如下图所示。

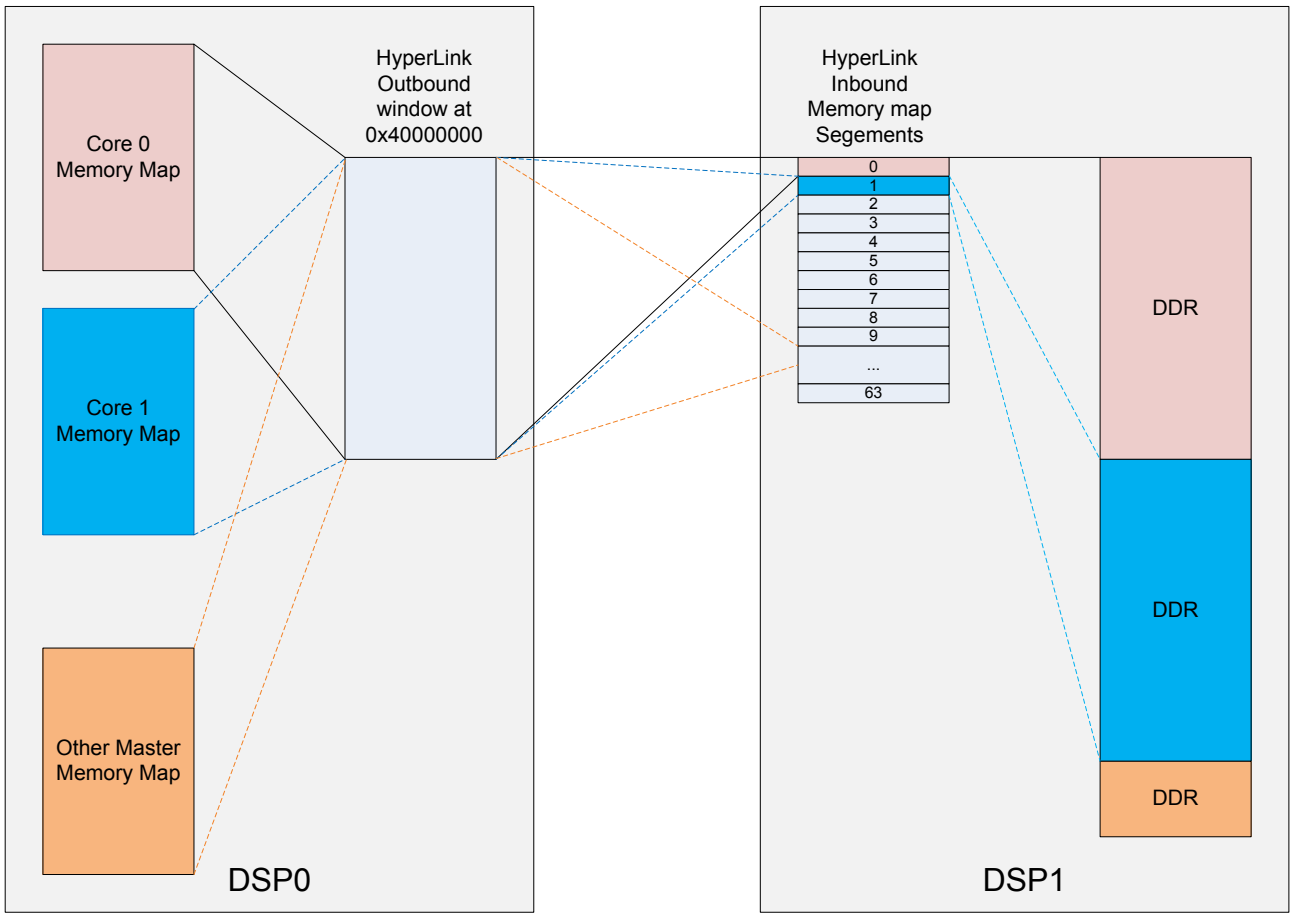

图 3 通过 HyperLink 窗口只映射到远端的 DDR 空间

这是最简单的例子,但是却可以访问远端 DSP 的大块存储空间。DSP0 上的每个 master(core 或者其他外设) 都可以可以访问 DSP1 上 256MB DDR 空间。下表描述了 core0 和 core1 的对 remote DSP DDR 存储映射。

表 2 本地 cores 映射到远端 DSP 的 DDR 的简单例子

本地地址 (size)	远端 DSP 地址空间	
	core 0 映射到的存储空间	Core 1 映射到的存储空间
0x40000000 (256MB)	0x90000000 (DDR)	0xA0000000 (DDR)

3 HyperLink 性能考虑

本节将让设计者对 HyperLink 访问远程存储空间的性能评估有基本的认识。同时提供了在不同的操作条件下获得的性能测试数据。大部分测试是在最理想的测试条件进行，以评估可以获得的最大吞吐量。

本文所描述的绝大部分性能数据是在 C6670EVM 上获得。C6670 EVM 上 DDR 配置成 64bit 位宽 1333M， HyperLink 速率配置成 10Gbit。

一些影响 HyperLink 访问性能的因素在本节中将会被讨论到。

3.1 通过 HyperLink 实现存储拷贝的性能

下表（表 3）描述了使用 HyperLink 在 LL2 与远程大块线性存储空间进行数据传送测试获得的传输带宽。传输块的大小为 64KB。带宽的计算是通过计算传输总的字节数除以传输所用的时间获得。

表 3 使用内核 core 通过 HyperLink 实现数据拷贝达到的带宽 (MB/s)

	Bandwidth (MB/s)	Remote memory		
Local configuration	Source->Destination	DDR	SL2	LL2
none Cacheable, none prefetchable	LL2->HyperLink	908	908	908
	HyperLink->LL2	50	63	52
	HyperLink->HyperLink	43	55	49
32KB L1 cache, prefetchable	LL2->HyperLink	1164	1164	1164
	HyperLink->LL2	323	383	294
	HyperLink->HyperLink	214	258	219
32KB L1 cache, 和 256K L2 cache, prefetchable	LL2->HyperLink	514	583	461
	HyperLink->LL2	538	620	474
	HyperLink->HyperLink	460	628	504

上述数据展示了 cache 能够极大的改善 DSP 内核通过 HyperLink 读取数据的性能。

但是 L2 cache 却遏制了通过 HyperLink 写数据的性能，这是因为 L2 是 write-allocate cache。对于使能 L2cache 后的写操作，它总是会先从将要写入的存储区读取 128 字节的数据到 L2 cache ，然后在 L2 cache 中修改数据，最后在 cache 冲突的时候回写回到原先的存储区，或者人为的回写回原存储区。

表 4 使用 EDMA 方式通过 HyperLink 实现数据拷贝达到的带宽（MB/s）

Bandwidth (MB/s)	Remote memory		
Source -> Destination	DDR	SL2	LL2
LL2 -> HyperLink	3584	3584	3583
SL2->HyperLink	3597	3596	3596
DDR->HyperLink	3571	3583	3584
HyperLink->LL2	1818	1849	1522
HyperLink->SL2	1828	1848	1525
HyperLink->DDR	1811	1848	1525

HyperLink->HyperLink	1032	1849	1525

上述 EDMA 吞吐率数据是通过 TC0 (传输控制器 0)和 CC0(通道控制器 0)上测试得到，其他 TCs 的数据会比 TC0 稍低。整个传输的瓶颈是在 HyperLink，不是在 EDMA 传输控制器上。

上述测试结果表明通过 HyperLink 进行写操作的性能会比通过 HyperLink 进行读操作的性能要好。

远程 DSP 存储空间类型不会对带宽造成明显的影响。访问远程 DSP 的 SL2 会比 LL2 快一些。

目前，通过 HyperLink 来访问远程 DSP 存储空间（相对其他接口）是具有最高的带宽性能的，但是访问远程存储空间比访问本地存储空间还是要慢。下表对比了访问本地 LL2 和 DDR 与远程 DDR 的吞吐性能。

表 5 访问远程 DDR 与本地 DDR 带宽对比

目标地址 / 源地址	DSP core with L1 and L2 cache (MB/s)		EDMA (MB/s)	
	Local DDR	Remote DDR	Local DDR	Remote DDR
Local LL2	2109	514	5252	3584

大体来说，对本地存储空间的写入吞吐率是对远程空间进行写入操作的吞吐率的 3 倍。对远程空间的读性能会更差些。我们应该尽量避免远程读取数据。

3.2 DSP core 通过 HyperLink 进行远程访问的延迟

DSP 核通过 HyperLink 访问远程空间的性能高度依赖于 cache。当 DSP 内核通过 HyperLink 来访问远程存储空间的时候，一个 TR(传输请求)可能会被生成并传送给 XMC（这取决于数据是否可以进入 cache 和被预取）。TR 将会是下面中的一种。

- 一个单一的元素- 如果存储空间不能被 cache 和预存取。
- 一个 L1 cache line – 如果存储空间可以进入 cache ，但是 L2 cache 没有被使能。
- 一个 L2 cache line -如果存储空间可以进入 cache，同时 L2 cache 被使能.
- 如果存储空间可以被预存取, 预存取将会被使能为一个预存取的 buffer slot.

如果 L1/L2cache 或者预存取命中， Hyperlink 端口不会有数据传输

远程空间数据可以被本地 L1 cache/L2 cache 缓存，或者都没有被 cache。如果对应存储空间的 MAR(Memory Attribute Register) 寄存器上的 PC(Permit copy)位没有被置位，那么对应存储区的数据将不会进入 cache。

如果 MAR 寄存器上 PC 位被置位，同时 L2 的 cache 空间是 0（L2 被全部配置成 SRAM），那么外部存储空间的数据可以进入 L1cache。

如果 MAR 寄存器上 PC 位被置位，L2 的 Cache 空间大于 0．那么外部存储空间的数据就可以进入 L1cache 和 L2cache。

读取远程存储空间数据也可以使用 XMC 中的 prefetch buffer。该特性可以在 MAR 寄存器 PFX(PreFetchable eXternally)被置位后使能。

地址步进长度也会影响 Cache 和 Prefetch buffer 的使用效果。连续空间的访问可以最充分的利用 cache 和 prefetch buffer，从而达到更好的性能。

以 64bytes 距离或者更大间隔进行步进访问将会导致每次 L1 cache 命中失败（miss），这是因为 L1 cache line 的大小是 64byte。

以 128bytes 距离或者更大间隔进行步进访问将会导致每次 L2 cache 命中失败（miss）。

如果 cache miss 发生，那么 DSP 核就会被 stall（等待数据）。Stall 的时间长度等于传输延迟、传输间隔，数据返回时间，cache 请求延迟的总和。

下面的章节描述 DSP 内核通过 HyperLink 访问存储区的延迟。测试伪代码如下列所示。

```
flushCache();
preCycle= getTimeStampCount();
for(i=0; i< accessTimes; i++)
{
        Access a Memory unit at address;
        address+= stride;
}
cycles = getTimeStampCount()-preCycle;
cycles/Access= cycles/accessTimes;
```

下图（图 4）为 1GHz C6670EVM 上配置 DDR 64bit 1333M 测试获得的结果。通过 HyperLink 实现 512 次 LDDW(load double word) 或者 STDW(store double word)操作的性能测试。图 4 绘制了各种测试条件下的性能 。 LDB/STB 和 LDW/STW 和 LDDW/STDW 的指令周期数相同。虽然 cache 和 prefetch buffer 可以被独立配置，但是测试的时候使用的配置是：如果 cache 被使能，那么 prefetch 也被使能，如果 cache 没有被使能，那么 prefetch 也没有被使能。

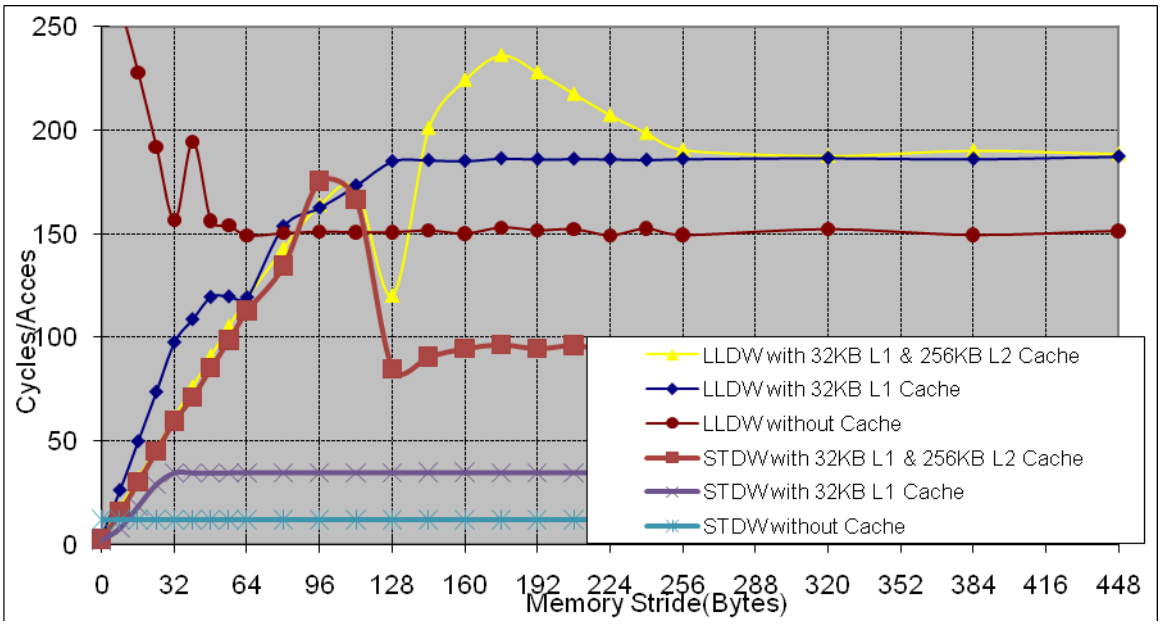

图 4 DSP Core 通过 HyperLink 对 remote DDR 进行 Load/Store 操作

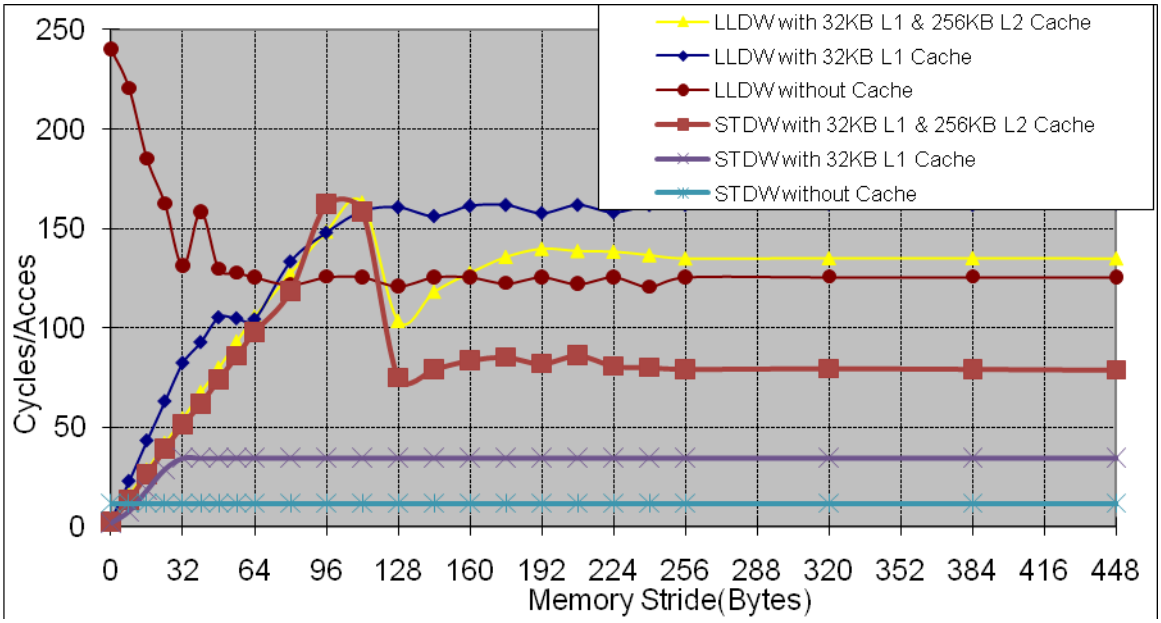

图 5 DSP Core 通过 HyperLink 对远程的 remote SL2 进行 Load/Store 操作

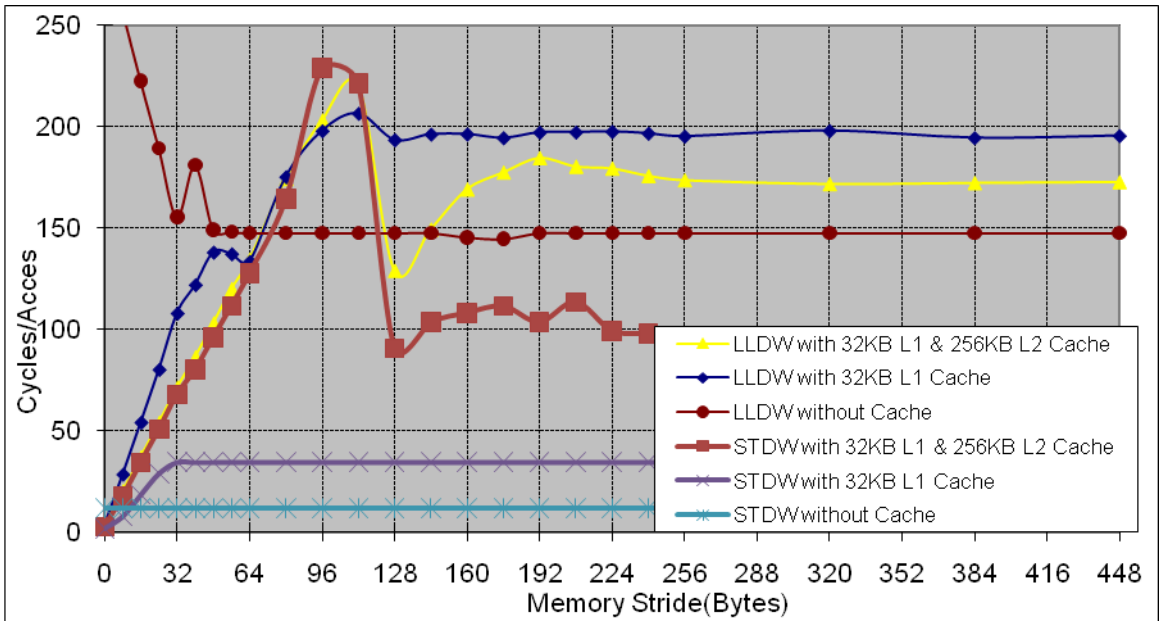

图 6 DSP Core 通过 HyperLink 对 remote LL2 进行 Load/Store 操作

Non-cachable 写是 post 操作。所以它只会 stall DSP core 很短的一段时间。

但是 read 是 non-post 的，所以 DSP 内核会等待数据的到来，所以它会 stall DSP 内核相对长一点时间。

当 cache 被使能后，DSP core 访问 remote 空间的吞吐性能高度依赖于 cache。

地址的步进间隔也会影响到 cache 的使用。连续的地址访问可以充分的利用 cache。但是地址的步进间隔超过 case line 的大小（L1 case line =64Byte,L2 Case line =128Byte）将会导致每次 cache 都无法命中，从而制约了性能。所以，对连续地址空间的数据访问（像大块数据拷贝），cache 需要被使能，在其他情况下 cache 应当不要使能。

通过上面的图可以发现通过 HyperLink 访问 DDR，SL2，LL2 在性能上并没有明显的差异。所以，正常情况下，通过 HyperLink 来共享 DDR 是一个很好的选择，因为 DDR 容量大，而且成本低。

3.3 HyperLink 传输使用 DMA 方式的开销（**overhead**）

初始延迟被定义为 EMDA 事件触发到真实数据之间的传输开始之间的延迟。因为初始延迟很难被测量。所以我们就测试传输的开销，它被定义为传输最小单元数据的延迟 。延迟的大小取决于源和目标端的类型。下表描述了使用 EDMA 在 1GHz TCI6618EVM 不同端口间传输一个字（word）时，从 EDMA 触发(写 ESR)到 EDMA 传输结束(读 IPR=1)的平均指令数目。

表 6 EDMA Transfer 平均 Overhead

destination / source	Local memory	Remote memory
Local Memory	325	322
Remote memory	853	700

表 6 中，读 Hyperlink 的延迟是 853 个指令周期， 写 Hyperlink 的延迟是 322 指令周期，因为写是 post 操作，而读是 non-post 操作。所以从 HyperLink 端口读取数据的延迟要高于写入数据到 HyperLink。

对于小批量数据传送，传输开销(overhead)是很大的顾虑，尤其是系统中队列 DMA 阻塞的时候。单一元素的传送性能较差，延迟会占用大部分时间。所以，对于小批量数据传送，必须对使用 EMDA 方式还是 DSP 核方式来访问数据进行权衡。使用内核来访问单个随机数据的延迟会比 DMA 方式延迟小很多。本文 3.2 节已经做了详细的描述。

3.4 HyperLink 中断延迟

一个 DSP 可以通过 HyperLink 来触发另外一个 DSP 的中断。通过 HyperLink 传递中断的延迟通过下列的伪代码获得测量。

```
    ……
    preTSC= TimeStampCount;
    // manually trigger the hardware event, which will generate interrupt packet to remote side
    HyperLinkRegs->SW_INT= HW_EVENT_FOR_INT_TEST;
    asm(“ IDLE”);    //wait for the queue pending interrupt
    delay= intTSC - preTSC;
    ……
interrupt void HyperLinkISR()          //HyperLink Interrupt Service Routine
{
    intTSC= TimeStampCount;       //save the Time Stamp Count when the interrupt happens
    ……
}
```

测试是在 Loopback 模式下测试。

1GHz C6670 的测试结果是 大概 710 个 DSP core cycles。

4 范例工程

本文的范例代码在 C6670EVM 上通过测试。EVM 板子上有两个 C6670 DSP，他们通过 HyperLink 互联。.

在这个例子中，DSP1 存储空间通过 HyperLink 被映射到了 DSP0 上。DSP0 通过 HyperLink 窗口访问 DSP1 的存储空间就像访问自己的本地空间一样。这个工程范例也支持 loopback 模式。在 loopback 模式下，DSP0 事实上是通过自己的 HyperLink 窗口访问了自己的本地地址空间。

本例也演示了通过 HyperLink 来实现中断传递。

工程代码的目录结构如下图所示。

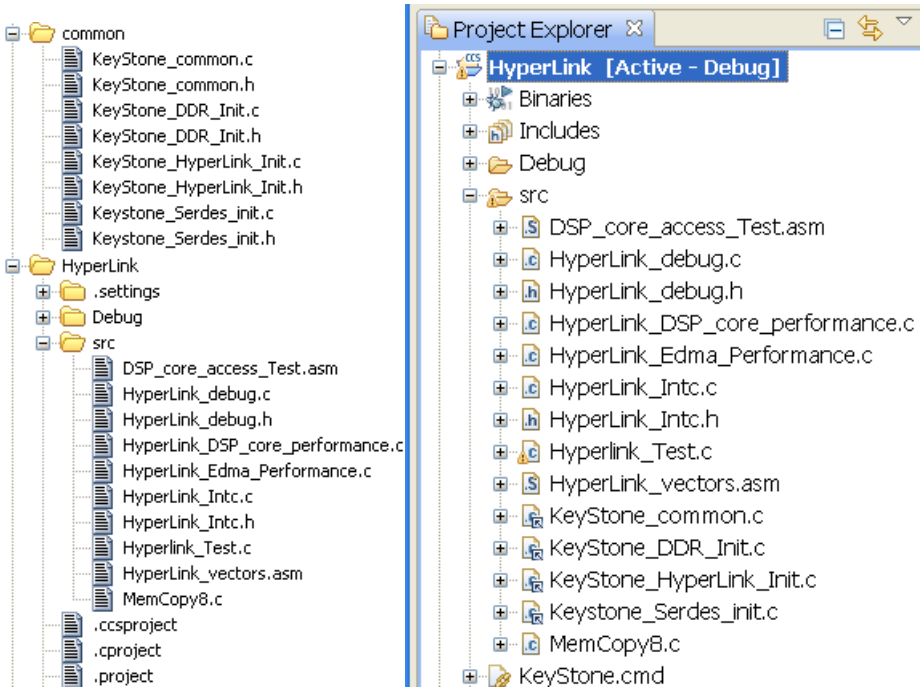

图 7 工程代码目录树

该示例代码同样可以在其他 KeyStone DSP EVM 板上运行。只是如果板子上只有一个 DSP 芯片，那么只能运行 loopback 模式。

下表列出了 KeyStone 工程的关键代码列表。

表 7 工程范例关键代码

Files	Function	Descriptions
KeyStone_HyperLink_Init.c		Low level 初始化代码
HyperLink_Test.c	HyperLink_config()	Application level 初始化代码
	HyperLink_Mem_Test()	这个函数将数据写到远程 DSP 存储空间，然后再回读回来来验证数据的一致性，确保通过 HyperLink 传输数据的结果是正确的。

	HyperLink_Interrupt_Test()	中断测试在 Loopback 模式下进行。人工触发一个中断事件。一个中断包生成，同时 loopback 到这个 DSP 并触发中断到 DSP core。函数将会测量从事件触发到进入中断服务函数 ISR 之间的时间延迟。
HyperLink_DSP_core_performance.c	MemCopyTest()	测量使用 DSP core 通过 HyperLink 进行存储拷贝的吞吐率
	LoadStoreCycleTest()	测量 DSP core 通过 HyperLink 来访问数据的的延迟。
HyperLink_Edma_Performance.c		测量使用 EDMA 方式通过 HyperLink 进行数据拷贝的吞吐率。

运行这个范例工程的步骤：

1. 连通 CCS 与 DSP EVM.
2. 下载代码到 core 0 of DSP0.
3. 下载代码到 core 0 of DSP1.
4. 先运行 DSP1 , 然后运行 DSP0. (如果 EVM 上只有一个 DSP,且运行在 loopback 模式下，那么直接下载到 DSP0，再运行)
5. 检测每个 DSP 的 stdout 窗口，验证测试结果。

典型的输出信息如下:

```
Initialize main PLL = x236/29
Initialize DDR PLL = x20/1
configure DDR at 666 MHz
HyperLink test at 10.00 GHz
HyperLink memory test passed at address 0x41800000, 0x10000 bytes
HyperLink memory test passed at address 0x48000000, 0x10000 bytes
HyperLink memory test passed at address 0x4c100000, 0x10000 bytes
    noncacheable, nonprefetchable memory copy
 908 MB/s, copy 65536 bytes from 0x820000 to 0x41800000 consumes 72137 cycles
  52 MB/s, copy 65536 bytes from 0x41800000 to 0x820000 consumes 1239592 cycles
  49 MB/s, copy 65536 bytes from 0x41800000 to 0x41810000 consumes 1336908 cycles
……
……
Throughput test with EDMA1 TC1
```

```
transfer   4 * 16384 Bytes with index=16384 from 0x11820000 to 0x42840000, consumes  18274 cycles, achieve bandwidth  3586 MB/s
transfer   4 * 16384 Bytes with index=16384 from 0x11820000 to 0x4c140000, consumes  18274 cycles, achieve bandwidth  3586 MB/s
transfer   4 * 16384 Bytes with index=16384 from 0x11820000 to 0x48300000, consumes  18274 cycles, achieve bandwidth  3586 MB/s
transfer   4 * 16384 Bytes with index=16384 from 0x42820000 to 0x11840000, consumes  53209 cycles, achieve bandwidth  1231 MB/s
transfer   4 * 16384 Bytes with index=16384 from 0x4c100000 to 0x11840000, consumes  44947 cycles, achieve bandwidth  1458 MB/s
transfer   4 * 16384 Bytes with index=16384 from 0x48200000 to 0x11840000, consumes  51679 cycles, achieve bandwidth  1268 MB/s
transfer   4 * 16384 Bytes with index=16384 from 0x c080000 to 0x42840000, consumes  18274 cycles, achieve bandwidth  3586 MB/s
transfer   4 * 16384 Bytes with index=16384 from 0x c080000 to 0x4c140000, consumes  18274 cycles, achieve bandwidth  3586 MB/s
transfer   4 * 16384 Bytes with index=16384 from 0x c080000 to 0x48300000, consumes  18274 cycles, achieve bandwidth  3586 MB/s
transfer   4 * 16384 Bytes with index=16384 from 0x42820000 to 0x c0c0000, consumes  53260 cycles, achieve bandwidth  1230 MB/s
transfer   4 * 16384 Bytes with index=16384 from 0x4c100000 to 0x c0c0000, consumes  45508 cycles, achieve bandwidth  1440 MB/s
transfer   4 * 16384 Bytes with index=16384 from 0x48200000 to 0x c0c0000, consumes  51985 cycles, achieve bandwidth  1260 MB/s
transfer   4 * 16384 Bytes with index=16384 from 0x88000000 to 0x42840000, consumes  18886 cycles, achieve bandwidth  3470 MB/s
transfer   4 * 16384 Bytes with index=16384 from 0x88000000 to 0x4c140000, consumes  19192 cycles, achieve bandwidth  3414 MB/s
transfer   4 * 16384 Bytes with index=16384 from 0x88000000 to 0x48300000, consumes  18835 cycles, achieve bandwidth  3479 MB/s
transfer   4 * 16384 Bytes with index=16384 from 0x42820000 to 0x88100000, consumes  53311 cycles, achieve bandwidth  1229 MB/s
transfer   4 * 16384 Bytes with index=16384 from 0x4c100000 to 0x88100000, consumes  45559 cycles, achieve bandwidth  1438 MB/s
transfer   4 * 16384 Bytes with index=16384 from 0x48200000 to 0x88100000, consumes  52087 cycles, achieve bandwidth  1258 MB/s
transfer   4 * 16384 Bytes with index=16384 from 0x42820000 to 0x42840000, consumes  57340 cycles, achieve bandwidth  1142 MB/s
transfer   4 * 16384 Bytes with index=16384 from 0x4c100000 to 0x4c140000, consumes  48313 cycles, achieve bandwidth  1356 MB/s
transfer   4 * 16384 Bytes with index=16384 from 0x48200000 to 0x48300000, consumes 103444 cycles, achieve bandwidth   633 MB/s
EDMA test completeInitialize DDR PLL = x20/1
```

用户可以在 HyperLink_Test.c 中的 HyperLink_config()函数中修改初始化值，然后重新编译来验证在不同配置下的 HyperLink 性能。

这个例子是在 CCS5.1 下编译，使用 pdk_c6618_1_0_0_5。如果在你的电脑上进行重新编译新的配置，你可能需要修改 csl 包含路径。

参考文献

1. KeyStone Architecture HyperLink User Guide (SPRUGW8)
2. TMS320C6670 datasheet (SPRS689)

重要声明

德州仪器(TI) 及其下属子公司有权根据 JESD46 最新标准, 对所提供的产品和服务进行更正、修改、增强、改进或其它更改， 并有权根据 JESD48 最新标准中止提供任何产品和服务。客户在下订单前应获取最新的相关信息, 并验证这些信息是否完整且是最新的。所有产品的销售都遵循在订单确认时所提供的TI 销售条款与条件。

TI 保证其所销售的组件的性能符合产品销售时 TI 半导体产品销售条件与条款的适用规范。仅在 TI 保证的范围内，且 TI 认为 有必要时才会使用测试或其它质量控制技术。除非适用法律做出了硬性规定，否则没有必要对每种组件的所有参数进行测试。

TI 对应用帮助或客户产品设计不承担任何义务。客户应对其使用 TI 组件的产品和应用自行负责。为尽量减小与客户产品和应 用相关的风险，客户应提供充分的设计与操作安全措施。

TI 不对任何 TI 专利权、版权、屏蔽作品权或其它与使用了 TI 组件或服务的组合设备、机器或流程相关的 TI 知识产权中授予 的直接或隐含权限作出任何保证或解释。TI 所发布的与第三方产品或服务有关的信息，不能构成从 TI 获得使用这些产品或服 务的许可、授权、或认可。使用此类信息可能需要获得第三方的专利权或其它知识产权方面的许可，或是 TI 的专利权或其它 知识产权方面的许可。

对于 TI 的产品手册或数据表中 TI 信息的重要部分，仅在没有对内容进行任何篡改且带有相关授权、条件、限制和声明的情况 下才允许进行复制。TI 对此类篡改过的文件不承担任何责任或义务。复制第三方的信息可能需要服从额外的限制条件。

在转售 TI 组件或服务时，如果对该组件或服务参数的陈述与 TI 标明的参数相比存在差异或虚假成分，则会失去相关 TI 组件 或服务的所有明示或暗示授权，且这是不正当的、欺诈性商业行为。TI 对任何此类虚假陈述均不承担任何责任或义务。

客户认可并同意，尽管任何应用相关信息或支持仍可能由 TI 提供，但他们将独力负责满足与其产品及在其应用中使用 TI 产品 相关的所有法律、法规和安全相关要求。客户声明并同意，他们具备制定与实施安全措施所需的全部专业技术和知识，可预见 故障的危险后果、监测故障及其后果、降低有可能造成人身伤害的故障的发生机率并采取适当的补救措施。客户将全额赔偿因 在此类安全关键应用中使用任何 TI 组件而对 TI 及其代理造成的任何损失。

在某些场合中，为了推进安全相关应用有可能对 TI 组件进行特别的促销。TI 的目标是利用此类组件帮助客户设计和创立其特 有的可满足适用的功能安全性标准和要求的终端产品解决方案。尽管如此，此类组件仍然服从这些条款。

TI 组件未获得用于 FDA Class III（或类似的生命攸关医疗设备）的授权许可，除非各方授权官员已经达成了专门管控此类使 用的特别协议。

只有那些 TI 特别注明属于军用等级或“增强型塑料”的 TI 组件才是设计或专门用于军事/航空应用或环境的。购买者认可并同 意，对并非指定面向军事或航空航天用途的 TI 组件进行军事或航空航天方面的应用，其风险由客户单独承担，并且由客户独 力负责满足与此类使用相关的所有法律和法规要求。

TI 已明确指定符合 ISO/TS16949 要求的产品，这些产品主要用于汽车。在任何情况下，因使用非指定产品而无法达到 ISO/TS16949 要求，TI不承担任何责任。

	产品		应用
数字音频	www.ti.com.cn/audio	通信与电信	www.ti.com.cn/telecom
放大器和线性器件	www.ti.com.cn/amplifiers	计算机及周边	www.ti.com.cn/computer
数据转换器	www.ti.com.cn/dataconverters	消费电子	www.ti.com/consumer-apps
DLP® 产品	www.dlp.com	能源	www.ti.com/energy
DSP - 数字信号处理器	www.ti.com.cn/dsp	工业应用	www.ti.com.cn/industrial
时钟和计时器	www.ti.com.cn/clockandtimers	医疗电子	www.ti.com.cn/medical
接口	www.ti.com.cn/interface	安防应用	www.ti.com.cn/security
逻辑	www.ti.com.cn/logic	汽车电子	www.ti.com.cn/automotive
电源管理	www.ti.com.cn/power	视频和影像	www.ti.com.cn/video
微控制器 (MCU)	www.ti.com.cn/microcontrollers		
RFID 系统	www.ti.com.cn/rfidsys		
OMAP应用处理器	www.ti.com/omap		
无线连通性	www.ti.com.cn/wirelessconnectivity	德州仪器在线技术支持社区	www.deyisupport.com

邮寄地址： 上海市浦东新区世纪大道 1568 号，中建大厦 32 楼 邮政编码： 200122