Application Report
Lit. Number – July, 2012

HyperLink Programming and Performance consideration

Brighton Feng Communication Infrastructure

ABSTRACT

HyperLink provides a highest-speed, low-latency, and low-pin-count communication interface between two Keystone DSPs. The HyperLink user's guide describes a lot about these. This application note provides some complementary information about the programming of the HyperLink.

This application report also discusses the performance of HyperLink, provides measured performance data achieved under various operating conditions. Some factors affecting HyperLink performance are discussed.

Example code is provided along with this application note.

Preliminary

Contents

1 Introduction 3
2 HyperLink configuration 3
2.1 Serdes configuration 3
2.2 HyperLink memory map configuration 4
3 HyperLink performance consideration 8
3.1 Performance for memory copy over HyperLink 8
3.2 Latency for DSP core access remote memory through HyperLink 9
3.3 Overhead of DMA Transfer over HyperLink 12
3.4 Delay for HyperLink interrupt 13
4 Example Projects 13
References 17

Figures

Figure 1. HyperLink Serdes configuration 3
Figure 2. Example of HyperLink mapping different memories 4
Figure 3. Example of HyperLink mapping DDR only 7
Figure 4. DSP Core Load/Store on remote DDR through HyperLink 11
Figure 5. DSP Core Load/Store on remote SL2 through HyperLink 11
Figure 6. DSP Core Load/Store on remote LL2 through HyperLink 12
Figure 7. Directory structure of example codes 14

Tables

Table 1. Example of memory map at different cores 4
Table 2. Simple example of memory map at different cores 7
Table 3. Bandwidth (MB/s) for memory copy over HyperLink with DSP core 8
Table 4. Bandwidth (MB/s) for memory copy over HyperLink with EDMA 8
Table 5. memory copy Comparison between local and remote memory 9
Table 6. Average EDMA Transfer Overhead 12
Table 7. Key codes of the example 14

Preliminary

1 Introduction

HyperLink provides a highest-speed, low-latency, and low-pin-count communication interface between two Keystone DSPs.

The speed of HyperLink was designed for 12.5Gbps, on most Keystone DSPs, the speed is limited to 10Gbps because of Serdes or Layout limitation. The HyperLink is a TI-specific peripheral, compared to the traditional 8b10b encoding scheme for high speed Serdes interfaces, HyperLink reduces the encoding overhead; the efficient encoding scheme in HyperLink is equivalent to 8b9b. There are 4 Serdes lanes for HyperLink, so the theoretic throughput of 10Gbps HyperLink is 10*4*(8/9)= 35.5Gbps= 4.44GB/s.

The HyperLink uses similar memory map scheme like PCIE, but it provides more flexible features designed for multicore DSP. This application note illustrates it with some usage cases.

This application report also discusses the performance of HyperLink, provides measured performance data achieved under various operating conditions. Some factors affecting HyperLink performance are discussed.

Preliminary

2 HyperLink configuration

This section provides some complementary information for the configuration of some HyperLink modules.

2.1 Serdes configuration

Serdes need be configured to achieve expected link speed. Following figure shows the relationship between input reference clock and clocks derived from it.

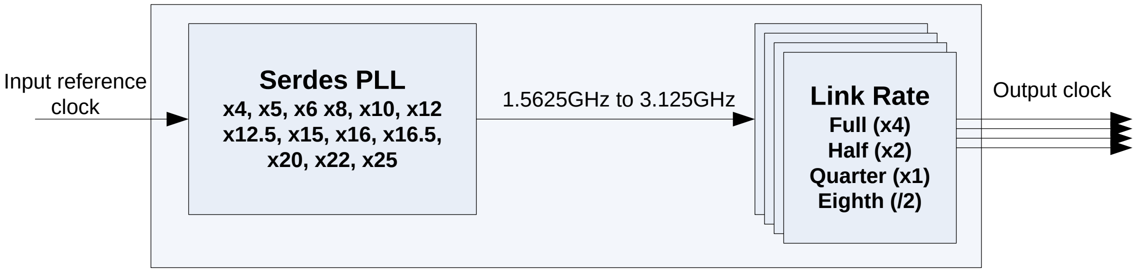


Figure 1. HyperLink Serdes configuration

Input clock are recommended to be between 156.25MHz and 312.5MHz, the Serdes PLL multiply factor should be setup to generate internal clock between 1.5625GHz and 3.125GHz.

The final link speed are derived from the internal clock according to the link rate configuration.

2.2 HyperLink memory map configuration

The HyperLink memory map is very flexible. The HyperLink user's guide describes a lot about these. This section illustrates it with two examples. Following figure shows the first example.

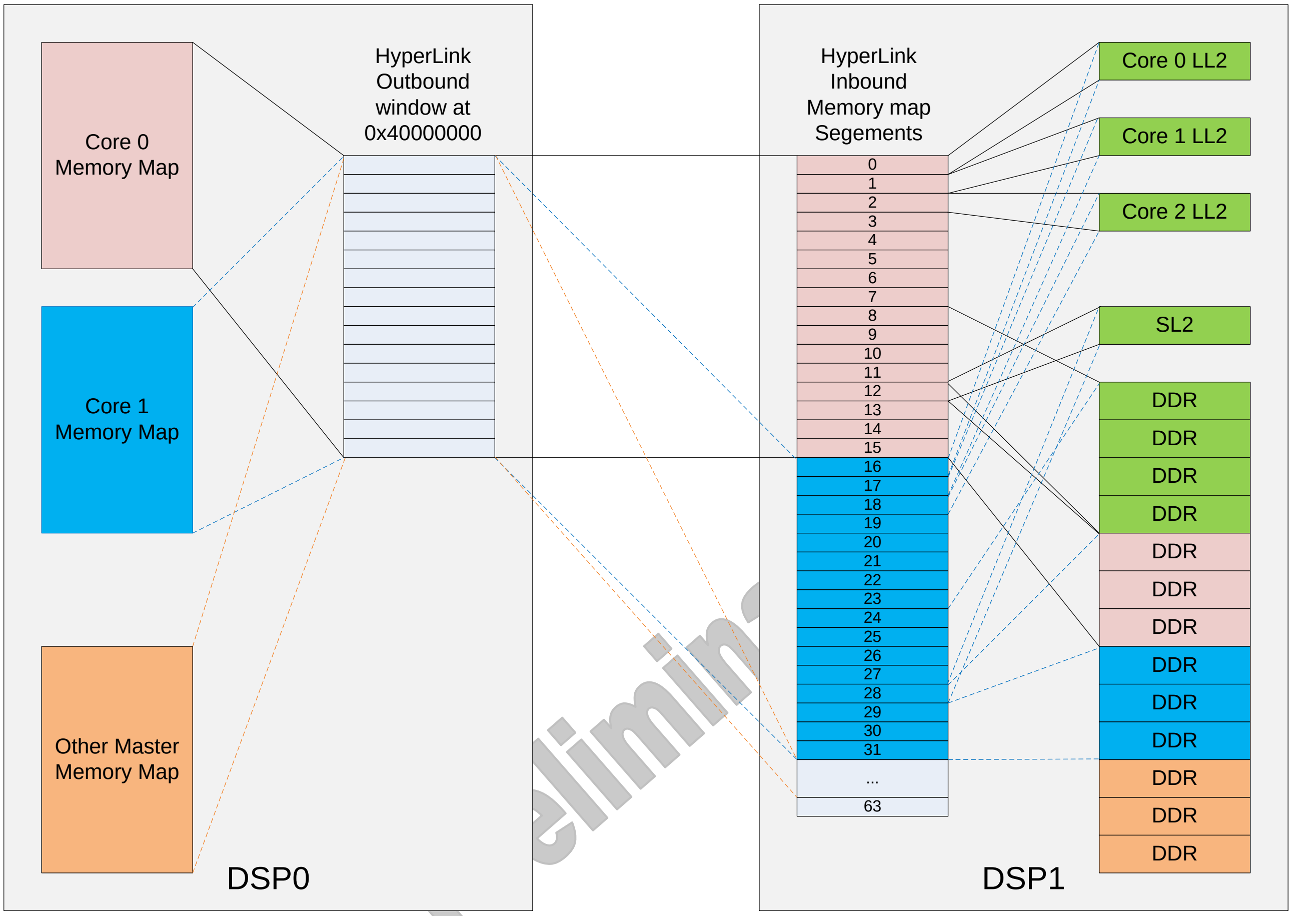


Figure 2. Example of HyperLink mapping different memories

In this example, DSP1's memory are mapped to DSP0's memory window from 0x40000000 to 0x50000000, the DSP0 can access all memory types in DSP1 including LL2 (Local L2), SL2 (Shared L2) and DDR just like access DSP0's local memory. In DSP0, all masters access same outbound window at 0x40000000, but they may actually access different memory space in DSP1, because the PrivID of the masters in DSP0 is carried with address and the inbound memory segment configuration of DSP1 is set up separately for different PrivID.

The memory map for DSP core 0 and core1 in this example are summarized in following table.

Table 1. Example of memory map at different cores

Local Address (Size)	Address at remote DSP	
	For core 0	For core 1
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)

With this configuration, when DSP0 accesses address 0x40800000, it actually accesses the LL2 memory of DSP1. When core 0 of DSP0 access address 0x4D000000, it actually accesses the 0x8C000000 in DDR of DSP1; while core 1 of DSP0 accesses address 0x4D000000 will actually access the 0x8F000000 in DDR of DSP1.

The example project along with this application note configures HyperLink in this way. Below is the key part of the configuration code.

liminary

```
/*----------------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++)
```

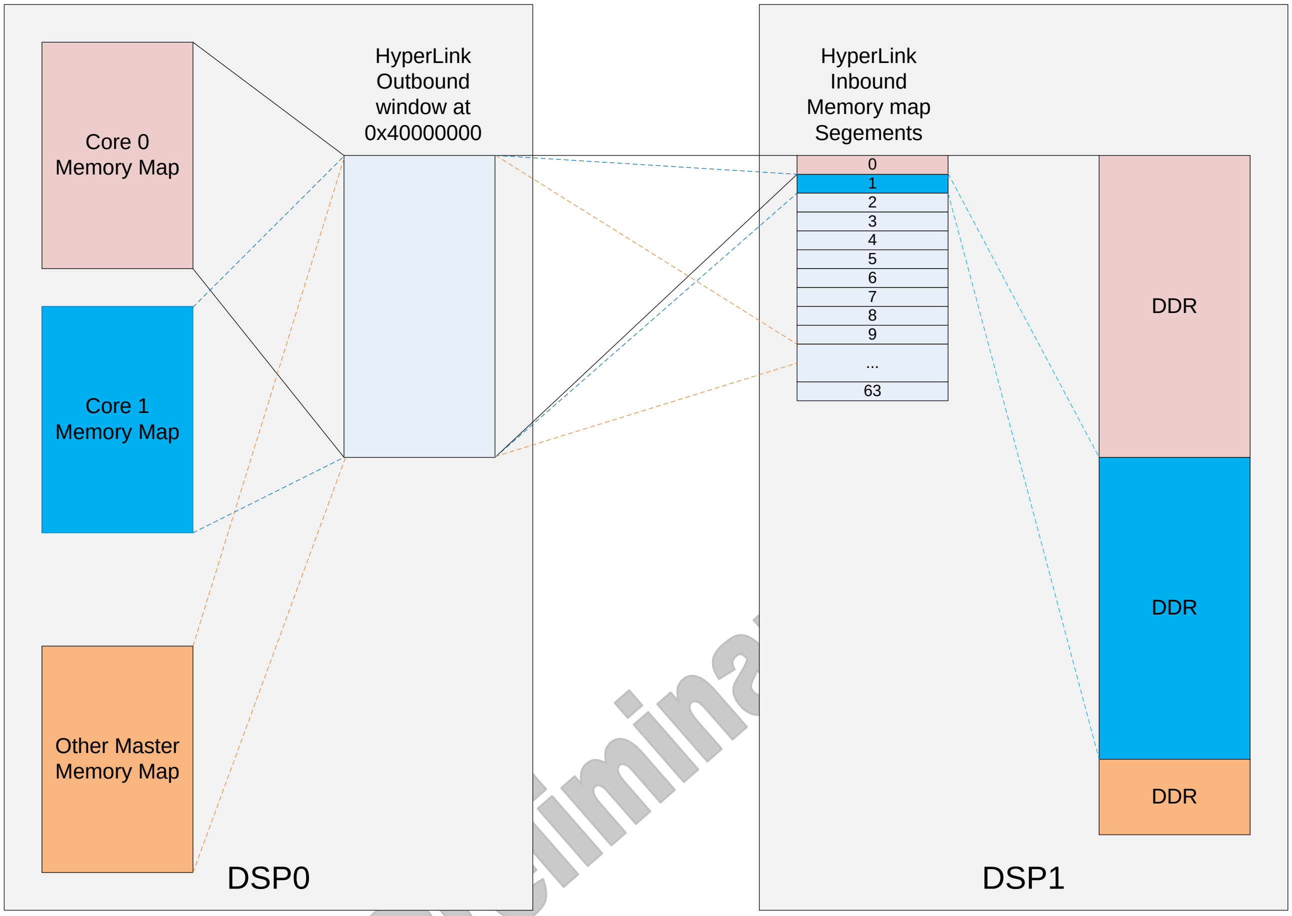
```
{
  hyperLink_cfg.address_map.rx_addr_segs[(iPrvId<<4)|iSeg].Seg_Base_Addr=
    DDR_SPACE_ACCESSED_BY_HYPERLINK+(iSeg-8)*0x01000000;
  hyperLink_cfg.address_map.rx_addr_segs[(iPrvId<<4)|iSeg].Seg_Length=
    RX_SEG_LEN_0x0_0100_0000;
}

/*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;
}
```

For some simple applications just need access large DDR space in remote DSP, below is a simple example for it.

Prelimina



Preliminary

Figure 3. Example of HyperLink mapping DDR only

This is the simplest example, while the largest memory space in remote DSP can be accessed. Each master in DSP0 can access 256MB dedicated DDR space in DSP1.

The memory map for DSP core 0 and core 1 in this example are summarized in following table.

Table 2. Simple example of memory map at different cores

Local Address (Size)	Address at remote DSP	
	For core 0	For core 1
0x40000000 (256MB)	0x90000000 (DDR)	0xA0000000 (DDR)

3 HyperLink performance consideration

This Section gives designers a basis for estimating HyperLink performance for access remote memory, provides measured performance data achieved under various operating conditions. Most of the tests operate under best-case situations to estimate maximum throughput that can be obtained. The transfers described in this document serve as a sample of interesting or typical performance conditions.

Most of the performance data in this document is examined on the C6670 EVM (EValuation Module) with 64-bit 1333M DDR memory, HyperLink link rate is set as 10Gbps.

Some factors affecting HyperLink access performance are discussed in this section.

3.1 Performance for memory copy over HyperLink

Following table shows the transfer bandwidth measured for large linear memory block copy between local LL2 and remote memory through HyperLink. In these tests, the memory block size is 64KB. The bandwidth is measured by taking total bytes copied and dividing it by the time it used.

Table 3. Bandwidth (MB/s) for memory copy over HyperLink with DSP core

	Bandwidth (MB/s)	Remote memory		
Local configuration	Source->Destination	DDR	SL2	LL2
NoneCacheable, nonprefetchable	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, and 256K L2 cache, prefetchable	LL2->HyperLink	514	583	461
	HyperLink->LL2	538	620	474
	HyperLink->HyperLink	460	628	504

Preliminary

Above data show cache improves the performance of DSP core reading through HyperLink dramatically.

But L2 cache throttles the performance of writing through HyperLink dramatically because L2 cache is write-allocate cache. For write operation with L2 cache, it always read 128 bytes including the accessed data into a cache line firstly, and then modify the data in the L2 cache. This data will be written back to memory through HyperLink if cache conflict happens or by manual writeback.

Table 4. Bandwidth (MB/s) for memory copy over HyperLink with EDMA

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

Above EDMA throughput data are measured on TC0 (Transfer Controller 0) of EDMA CC0 (Channel Controller 0). The throughput with other TCs are slightly less than TC0. The bottleneck of the throughput is on HyperLink, not on EDMA transfer controller.

Above data show the performance of writing over HyperLink is much better than read over HyperLink.

The remote memory type does not affect the bandwidth obviously. Accessing DDR and SL2 on remote DSP is slightly faster than LL2 on remote DSP.

Though HyperLink is the highest throughput interface on current DSPs, access remote memory through HyperLink is still much slower than access local memory. Following table compares the throughput for copying from local LL2 to local DDR and remote DDR.

Table 5. memory copy Comparison between local and remote memory

destination source	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

Preliminary

Generally speaking, the throughput for writing local memory is about 3 times as the throughput for writing remote memory through HyperLink. Read from remote memory through HyperLink is much worse, so we should try to avoid it.

3.2 Latency for DSP core access remote memory through HyperLink

DSP core access remote memory through HyperLink highly depends on the cache. When the DSP core accesses remote memory through HyperLink, a TR (transfer request) may be generated (depending on whether the data are cached and prefetchable) to the XMC. The TR will be for one of the following:

- a single element - if the memory space is non-cacheable, nonprefetchable
- a L1 cache line - if the memory space is cacheable and the L2 cache is disabled,
- a L2 cache line - if the memory space is cacheable and L2 cache is enabled.
- If the space is prefetchable, prefetch may be generated for a prefetch buffer slot.

No transfer request is generated in the case of an L1/L2 cache or prefetch hit.

Remote memory can be cached by L1 cache, L2 cache, or neither. If the PC (Permit Copy) bit of appropriate MAR (Memory Attribute Register) for a HyperLink mapped memory space is not set, it is not cacheable. If the PC bit of MAR is set and L2 cache size is zero (all L2 is defined as SRAM), the external memory space is cached by L1. If the MAR bit is set and L2 cache size is greater than 0, the external memory space is cached by L2 and L1.

Read to remote memory can also utilize the prefetch buffer in XMC, which is programmable by software through PFX (PreFetchable eXternally) bit of MAR (Memory Attribute Register).

The address increment (or memory stride) affects cache and prefetch buffer utilization. Contiguous accesses utilize cache and prefetch memory to the fullest. A memory stride of 64 bytes or more causes every access to miss in the L1 cache because the L1 line size is 64 bytes. A memory stride of 128 bytes causes every access to miss in L2 because the L2 line size is 128 bytes.

If cache miss happens, the DSP Core will stall, waiting for the return data. The length of the stall is equal to the sum of the transfer latency, transfer duration, data return time, and some small cache request overhead.

The following sections examine the latency for DSP Core accesses memory through HyperLink. The pseudo codes for this test are like following:

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

Following figures show data collected from 1GHz C6670 EVM with 64-bit 1333M DDR. The time required for 512 LDDW (LoaD Double Word) or STDW (STore Double Word) operations on remote memory through HyperLink was measured, and the average time for each operation is reported. The cycles for LDB/STB, and LDW/STW are same as LDDW/STDW. Though cache and prefetch buffer can be configured independently, for these test cases with cache enabled, the prefetch buffer is enabled as well; the prefetch buffer is also disabled for the cases with cache disabled.

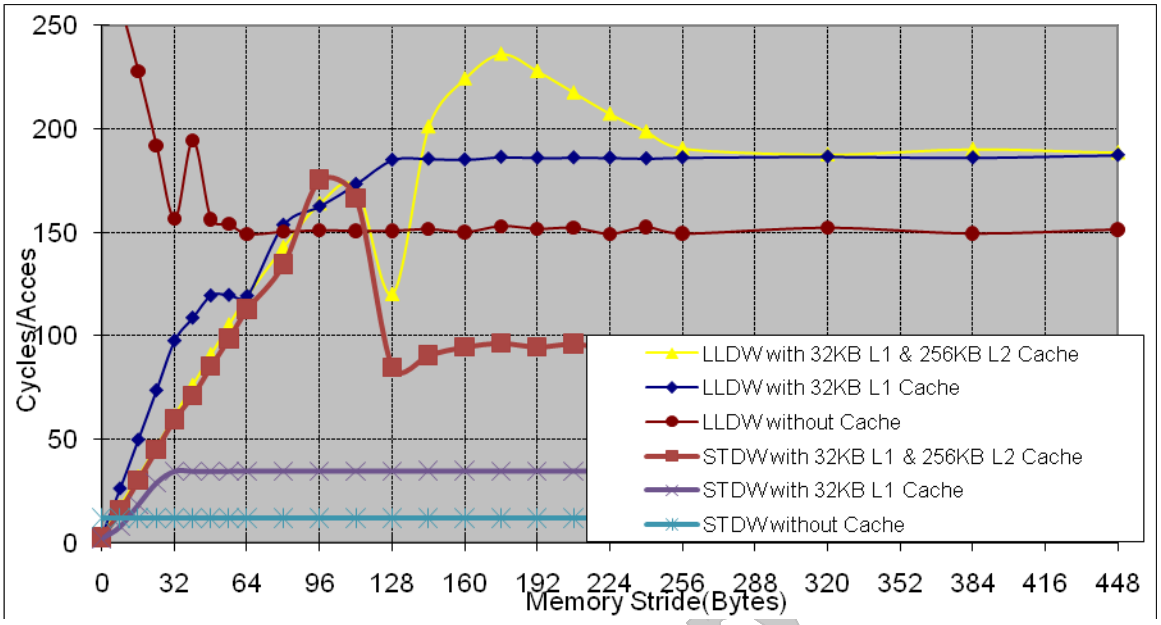


Figure 4. DSP Core Load/Store on remote DDR through HyperLink

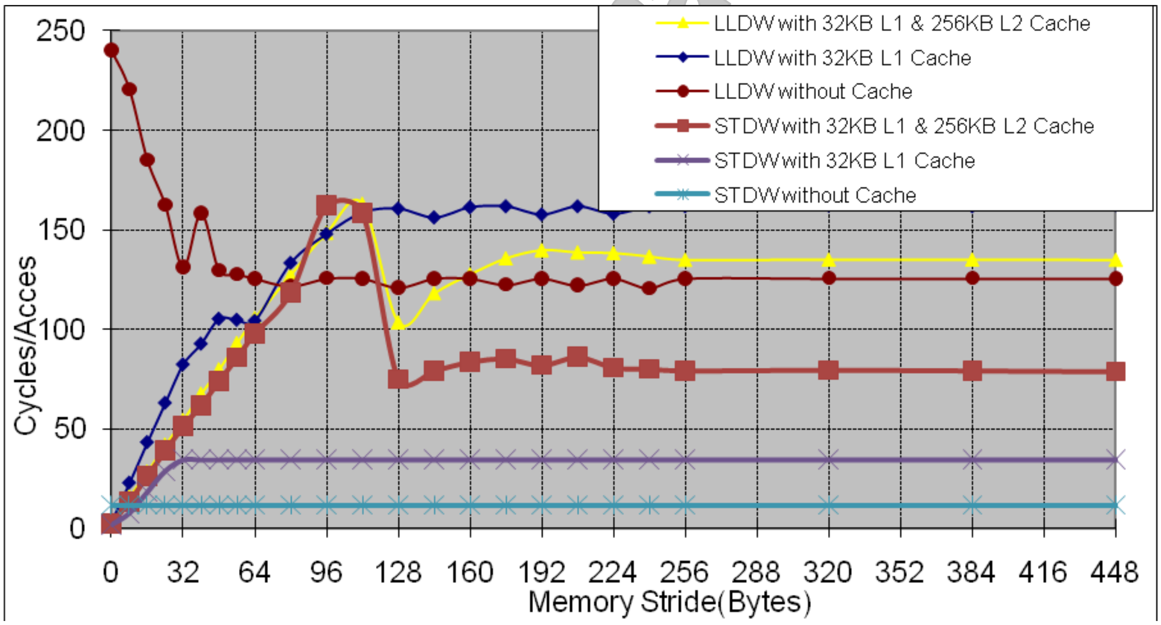


Figure 5. DSP Core Load/Store on remote SL2 through HyperLink

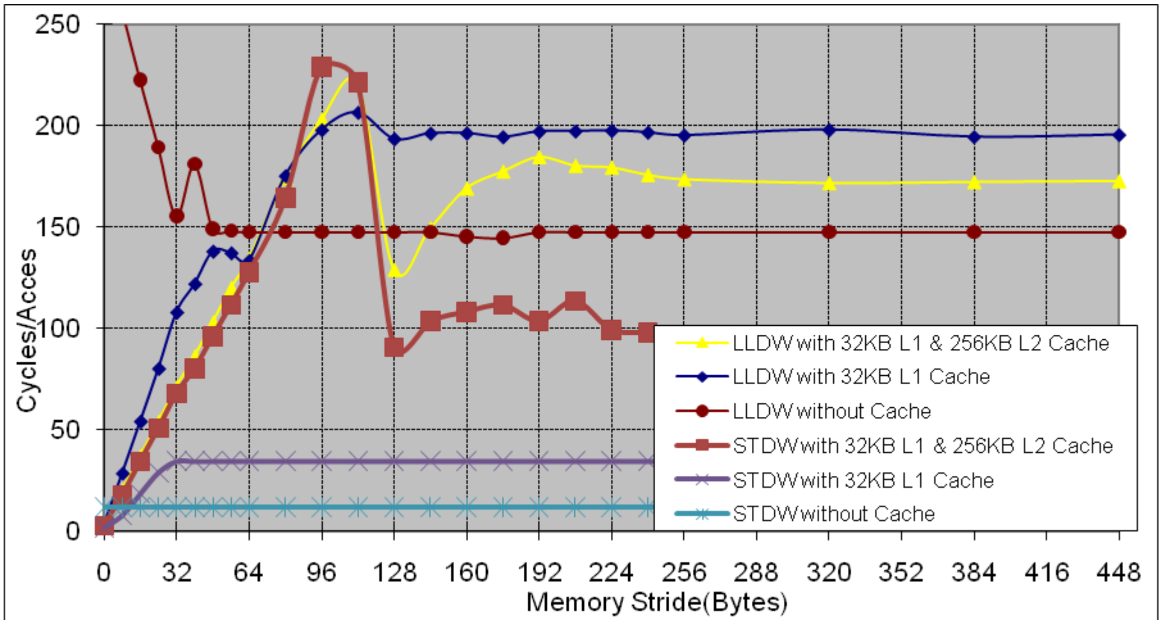


Figure 6. DSP Core Load/Store on remote LL2 through HyperLink

Non-cachable write is post operation, so it stalls DSP core very shortly; while read is non-post, DSP core will wait for the return data, so, it stalls DSP core relative long time.

When cache is enabled, DSP core access remote memory highly depends on the cache. The address increment (or memory stride) affects cache utilization. Contiguous accesses utilize cache to the fullest. A memory stride larger than cache line size (64 Bytes for L1 cache, 128 Bytes for L2 cache) causes every access to miss, results in a full cache line fetch for only one access, thus throttles the performance. So, for contiguous access (like memory copy discussed in above section), cache should be enabled; otherwise, cache should be disabled.

Above figures do not show obvious difference between accessing DDR, SL2 and LL2 through HyperLink. So, normally, sharing DDR through HyperLink is a good idea because its large size and low cost.

Preliminary

3.3 Overhead of DMA Transfer over HyperLink

Initial latency is defined as the time between EDMA event happen to real data transfer begin. Since initial latency is hard to measure. We measured transfer overhead instead; it is defined as the sum of the Latency, and the time to transfer smallest element. The values vary based on the type of source/destination endpoints. Following tables show the average cycles measured between EDMA trigger (write ESR) and EDMA completion (read IPR=1) for smallest transfer (1 word) between different ports on 1GHz C6670 EVM.

Table 6. Average EDMA Transfer Overhead

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

Because write is post operation, while read is non-post operation, so, latency for read from Hyperlink is much higher than the write to HyperLink.

Transfer overhead is a big concern for short transfers and need to be included when scheduling DMA traffic in a system. Single-element transfer performance will be latency-dominated. So, for small transfer, you should make the trade off between DMA and DSP core. The latency for single random DSP core access is much less than DMA, refer to above section 3.2 for more details.

3.4 Delay for HyperLink interrupt

One DSP can trigger the interrupt on another DSP through HyperLink. The delay for HyerLink interrupt is measured with following pseudo code:

```
    ……
    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
    ……
}
```

This test is done with internal loopback mode.

This delay measured on 1GHz C6670 is about **710** DSP core cycles.

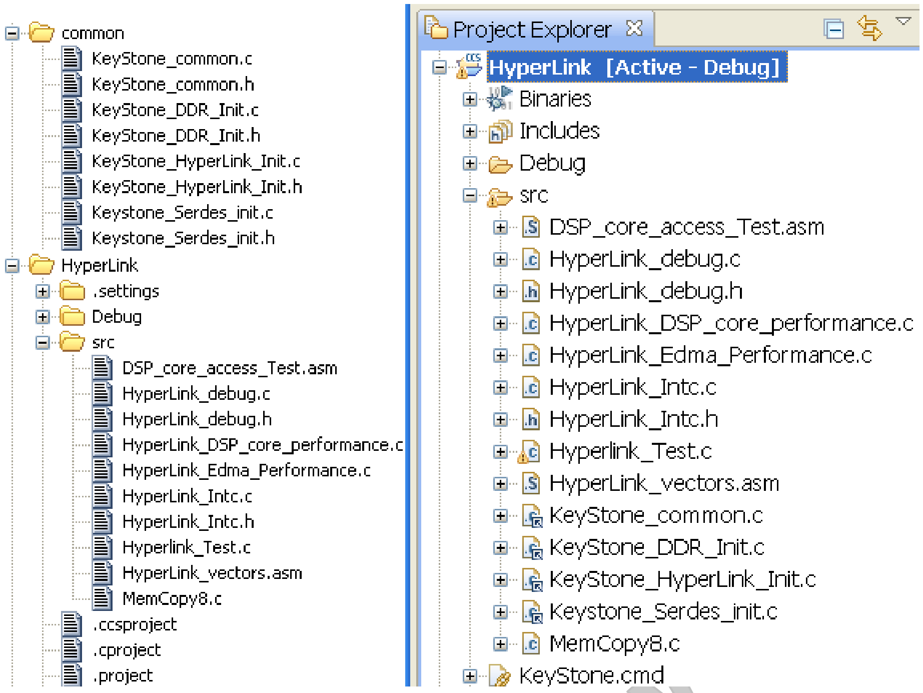
4 Example Projects

The example codes along with this application note are based on the dual C6670 EVM. The EVM board has two C6670 DSPs, they are connected through HyperLink.

In this example, DSP1’s memory are mapped to DSP0 through HyperLink (see above section 2.2 for details), so, DSP0 accesses DSP 1 through the HyperLink memory window just like access other local memory space. Internal loopback is also supported in this example, for this case, DSP0 actually access its own local memory through HyperLink memory window.

Interrupt generation through HyperLink is also demonstrated in this example.

The directory structure of the projects is as below:



```
common
    KeyStone_common.c
    KeyStone_common.h
    KeyStone_DDR_Init.c
    KeyStone_DDR_Init.h
    KeyStone_HyperLink_Init.c
    KeyStone_HyperLink_Init.h
    Keystone_Serdes_init.c
    Keystone_Serdes_init.h
HyperLink
    .settings
    Debug
    src
        DSP_core_access_Test.asm
        HyperLink_debug.c
        HyperLink_debug.h
        HyperLink_DSP_core_performance.c
        HyperLink_Edma_Performance.c
        HyperLink_Intc.c
        HyperLink_Intc.h
        Hyperlink_Test.c
        HyperLink_vectors.asm
        MemCopy8.c
    .ccsproject
    .cproject
    .project
```

```
Project Explorer
HyperLink  [Active - Debug]
    Binaries
    Includes
    Debug
    src
        DSP_core_access_Test.asm
        HyperLink_debug.c
        HyperLink_debug.h
        HyperLink_DSP_core_performance.c
        HyperLink_Edma_Performance.c
        HyperLink_Intc.c
        HyperLink_Intc.h
        Hyperlink_Test.c
        HyperLink_vectors.asm
        KeyStone_common.c
        KeyStone_DDR_Init.c
        KeyStone_HyperLink_Init.c
        Keystone_Serdes_init.c
        MemCopy8.c
    KeyStone.cmd
```

Figure 7. Directory structure of example codes

Preliminary

The example code can also be run on other Keystone DSP EVMs, but only internal loopback test can be run if there is only one DSP on the EVM.

The key codes are listed in following table.

Table 7. Key codes of the example

Files	**Function**	**Descriptions**
KeyStone_HyperLink_Init.c		Low level initialization code.
Hyperlink_Test.c	Hyperlink_config()	Application level initialization code.
	HyperLink_Mem_Test()	this function write data to remote memory unit and then readback from remote memory for verification. This make sure memory access through HyperLink is correct.
	HyperLink_Interrupt_Test()	this interrupt test is done in loopback mode, a hardware event is trigger manually, a interrupt packet is generated and loopback to this DSP and trigger interrupt to the DSP core. The latency between trigger and the entry of the ISR are measured.
HyperLink_DSP_core_performance.c	MemCopyTest()	Measure the throughput of memory copy with DSP core over HyperLink

	LoadStoreCycleTest()	Measure the latency of DSP core access memory throughput HyperLink
HyperLink_Edma_Performance.c		Measure the throughput of memory copy with EDMA over HyperLink

The steps to run the examples are:

1. Conncet CCS to the DSP.
2. Load the program to core 0 of DSP0.
3. Load the program to core 0 of DSP1.
4. Run DSP1 firstly, then run DSP0. (To run internal loopback only on one DSP, just load the program to the first DSP and run it.)
5. Check the stdout window of each DSP for test result.

The typical output information is as below:

```
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
```

User can change the initialization values in Hyperlink_config() function in “Hyperlink_Test.c” and rebuild the project to verify different configuration of HyperLink.

This example project are built with CCS5.1, pdk_c6618_1_0_0_5. To rebuild the project with your new configurations on your PC, you may need to change the CSL include path.

Preliminary

References

1. KeyStone Architecture HyperLink User Guide (SPRUGW8)
2. TMS320C6670 datasheet (SPRS688)

Preliminary