C66x KeyStone Training
HyperLink

Agenda

- Overview
- Address Translation
- Configuration
- Performance
- Example

Overview

- **Overview**
- Address Translation
- Configuration
- Performance
- Example

Overview: What is HyperLink?

High-speed chip-to-chip interface that connects…

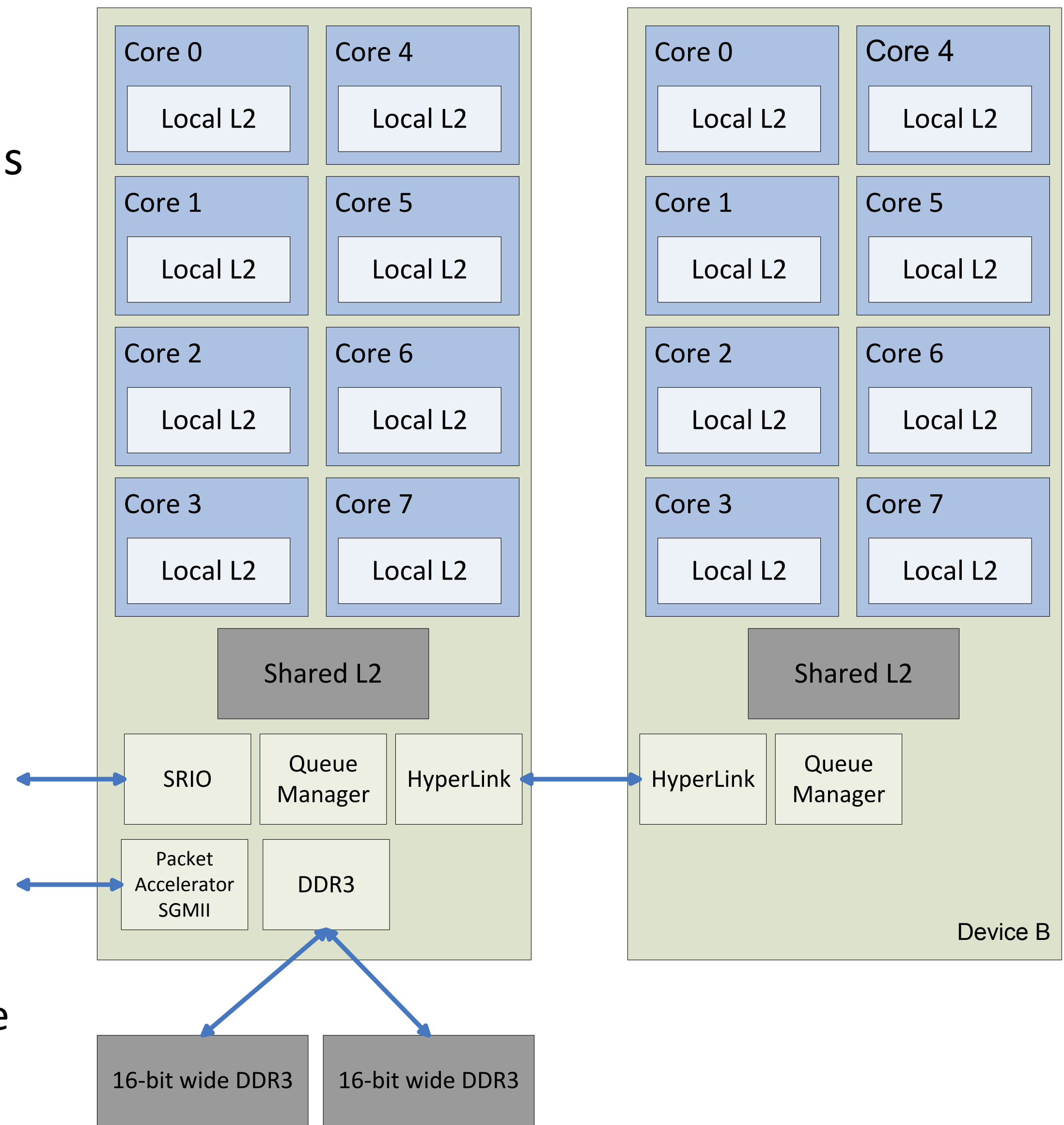
- Keystone devices to each other
  or
- Keystone device to an FPGA

Key Features and Advantages

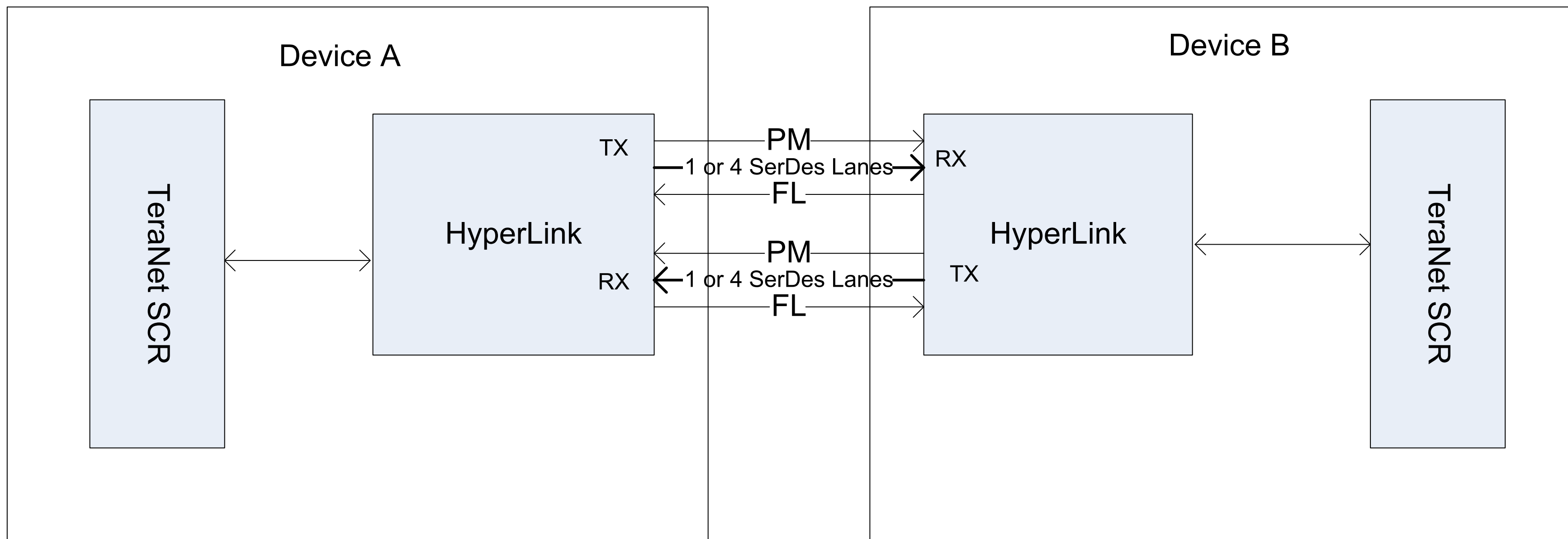
- High-speed -- 4 lanes at 12.5 Gbps/lane
- Low power -- 50% less than similar serial interfaces
- Low latency, low protocol overhead and low pin count
- Industry-standard SerDes

Overview: Example Use Case with 6678

- Device A sends packet frame to Device B for processing and receives result; Both transactions via HyperLink.
- Enables scalable solutions with access to remote CorePacs to expand processing capability. Device B acts as codec accelerator in this case.
- Reduce system power consumption by allowing users to disable I/O and peripherals on remote device.
  - Device A: all peripherals active
  - Device B: only HyperLink active



Overview: HyperLink External Interfaces



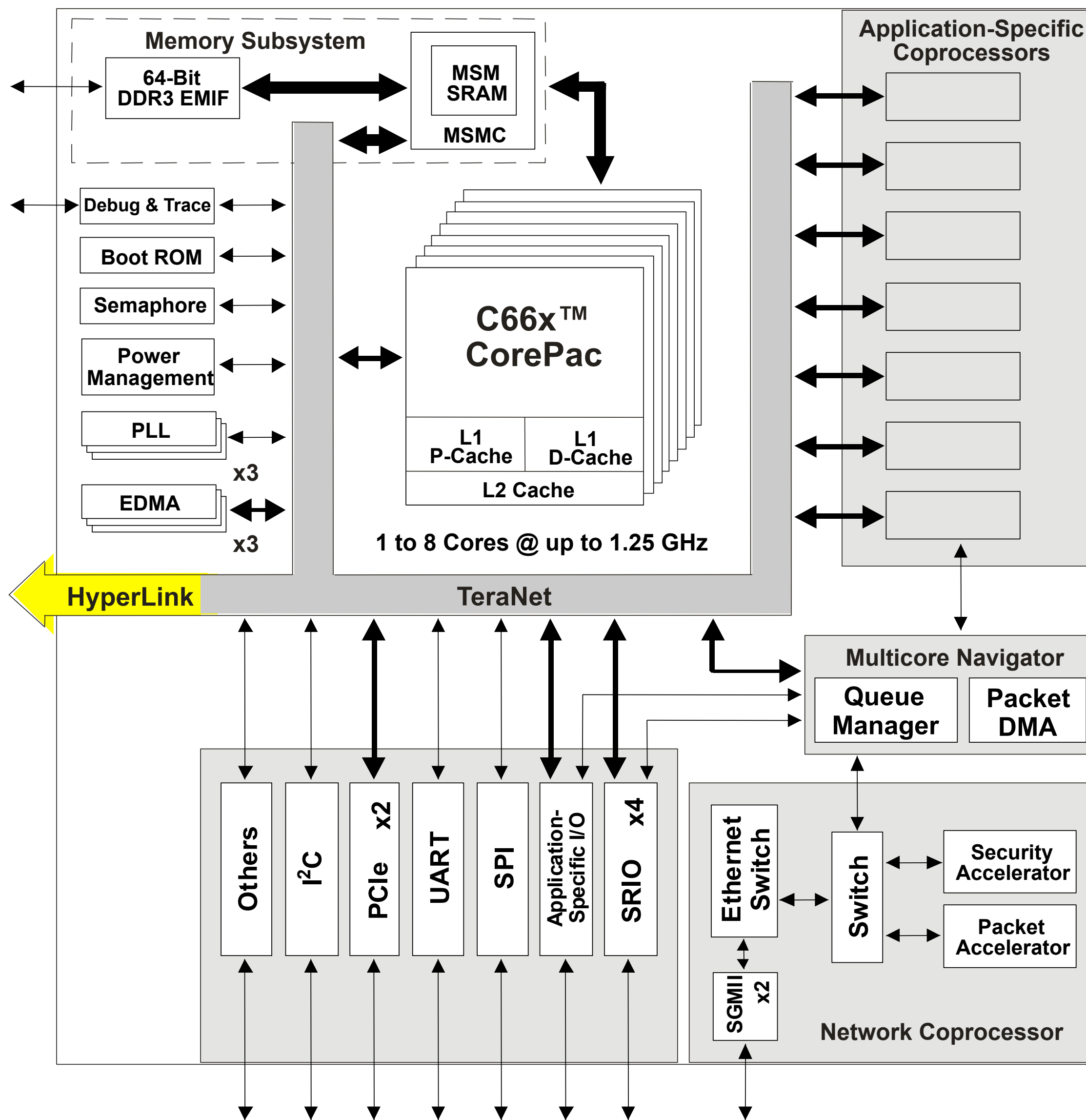
Data Signals SerDes-based

- 1-lane or 4-lane mode, with 12.5 Gbps data rate per lane

Control Signals LVCMOS-based

- Flow control (FL) and Power Management (PM)
- Auto managed by HyperLink after initial, one-time configuration by user
- FL managed on per-direction basis; RX sends throttle to TX
- PM dynamically managed per-lane, per-direction based on traffic

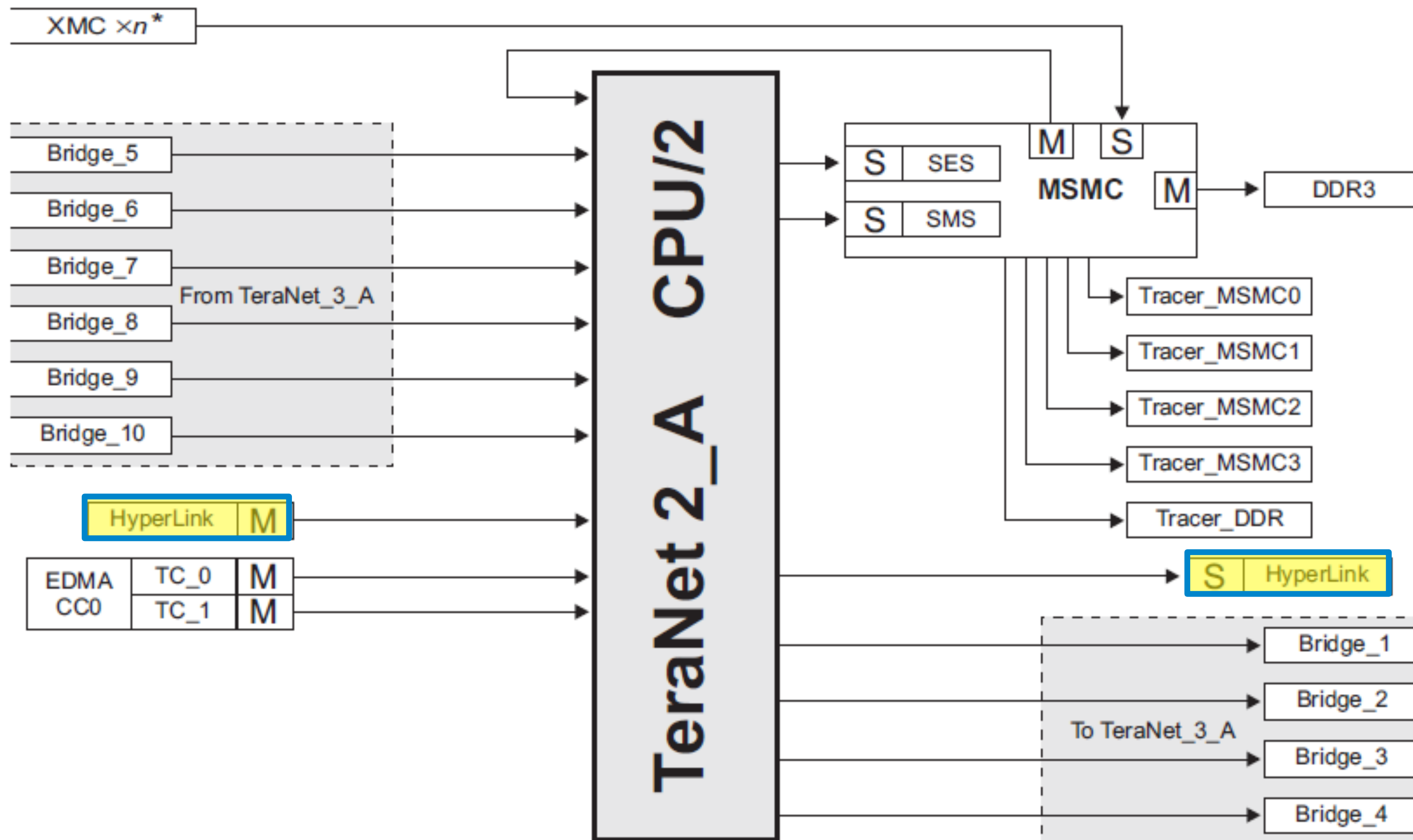
Overview: HyperLink and TeraNet



- C66x CorePacs, EDMA & peripherals are interconnected via **TeraNet** switch fabric
- **HyperLink** seamlessly extends TeraNet from one device to another
- Enables read/write transactions, as well as relaying & generation of interrupts between devices

Overview: TeraNet Connections

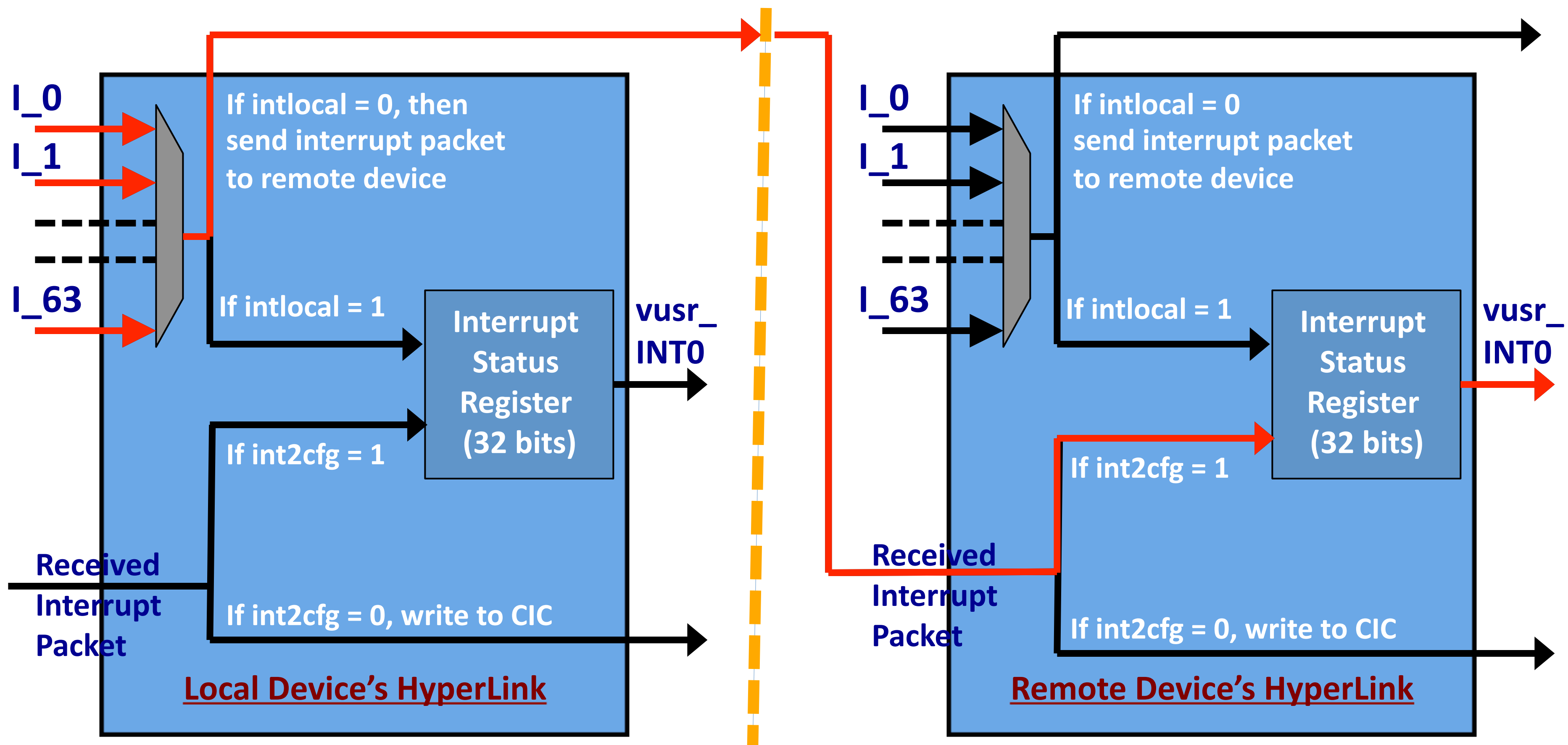
- C66x CorePacs, EDMA & peripherals classified as master or slave
- Master initiates read/write transfers. Slave relies on master
- HyperLink master and slave ports connected via TeraNet 2A



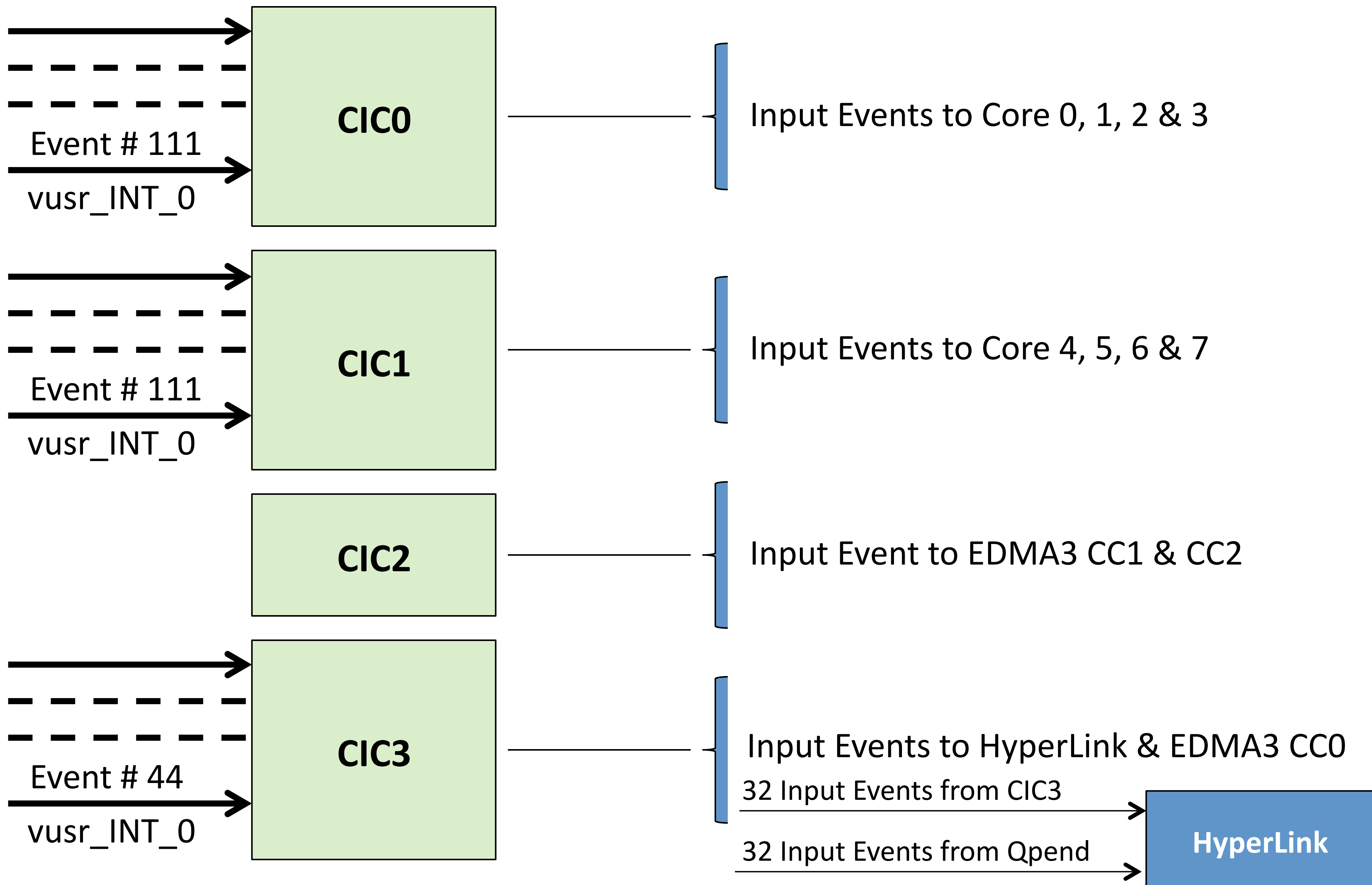
Overview: HyperLink Interrupts

64 interrupt inputs to HyperLink module:

- 0-31 from Chip Interrupt Controller (CIC) # 3
  - CIC3 events include GPIO, Trace, & Software-Triggered
- 32-63 from Queue manager (QMSS) pend event



Overview: HyperLink Interrupts



Overview: Packet-based Protocol

- HyperLink offers a packet-based transfer protocol that supports multiple outstanding read, write and interrupt transactions

- Users can use HyperLink to:
  - Write to remote device memory
  - Read from remote device memory
  - Generate events / interrupt in the remote device

- Read/Write transactions with 4 packet types
  - Write Request / Data Packet
  - Write Response Packet (optional)
  - Read Request Packet
  - Read Response Data Packet

- Interrupt Packet passes event to remote side

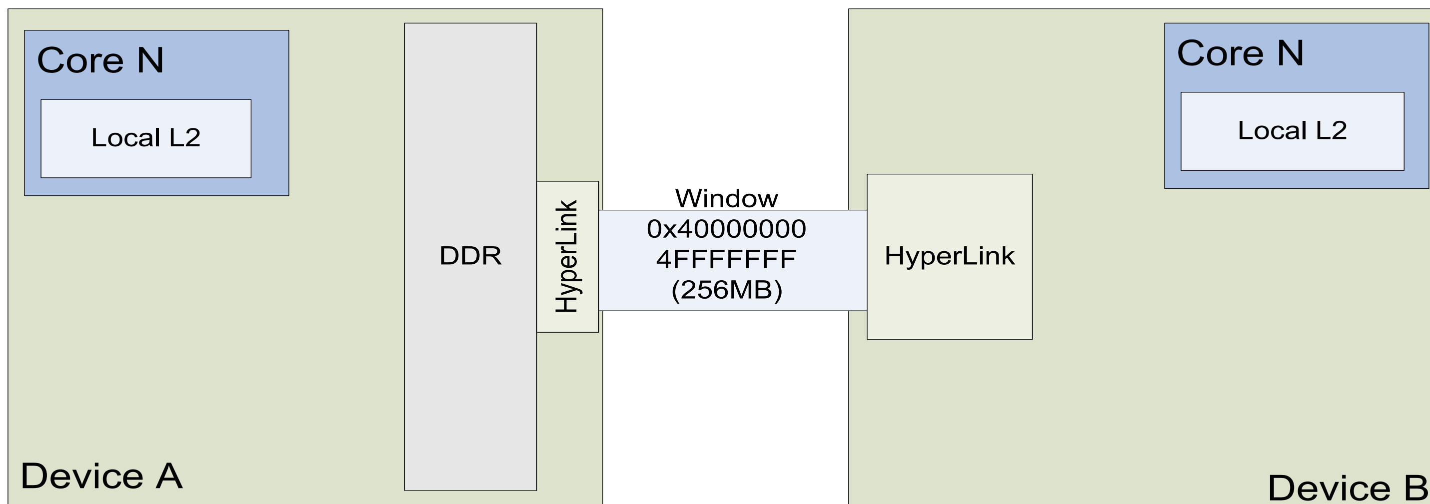
- 16-byte packet header for 64-byte payload, and 8b/9b encoding

Address Translation

- Overview
- **Address Translation**
- Configuration
- Performance
- Example

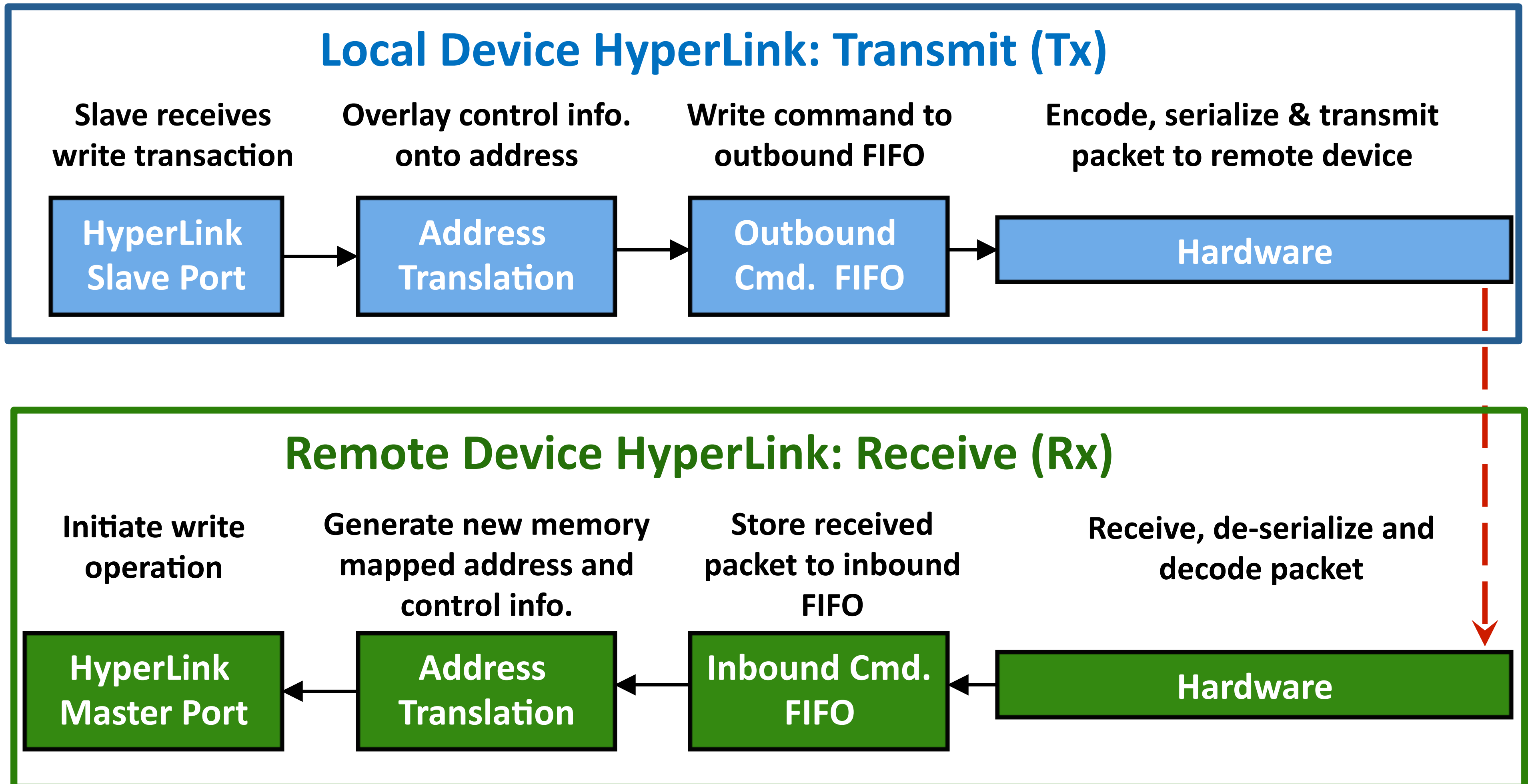
Address Translation: Motivation

- Device A (Tx) can view max. **256MB** of Device B (Rx) memory**.
- Tx side: HyperLink memory space is **0x4000_0000 to 0x4FFF_FFFF**
- Rx side: HyperLink memory space is device dependent, but typically somewhere in the 0x0000_0000 to 0xFFFF_FFFF address range
  For example: DDR 0x8000_0000 to 0x8FFF_FFFF
- Requires mechanism to convert local (Tx) address to remote (Rx) address
- The local side (Tx side) manipulates the address, the remote side (Rx) does address translation



** For each core

Address Translation: Write Example



Address Translation on Remote Side

- HyperLink supports up to 64 different memory segments at Rx.
- Segment size – Minimum 512 bytes, Maximum 256 MB
- Segments have to be aligned on 64 KB (0x0001_0000) boundary, which implies that the least-significant 16 bits of segment base address is always 0.

Address Translation: Segmentation

Number of bits used to represent address offset and number of bits used to choose segment depend on size of largest segment.

Largest Segment Size in Bytes (Power of 2)	Number of Bits for Address Offset	Maximum Number of Segments**	Number of Bits to Choose Segment
256 MB 0x0FFF_FFFF	28	1 = 2^0	0
128 MB 0x07FF_FFFF	27	2 = 2^1	1
8 MB 0x007F_FFFF	23	32 = 2^5	5
4 MB 0x003F_FFFF	22	64 = 2^6	6
2 MB 0x001F_FFFF	21	64 = 2^6	6
16 KB 0x0000_3FFF	14	64 = 2^6	6

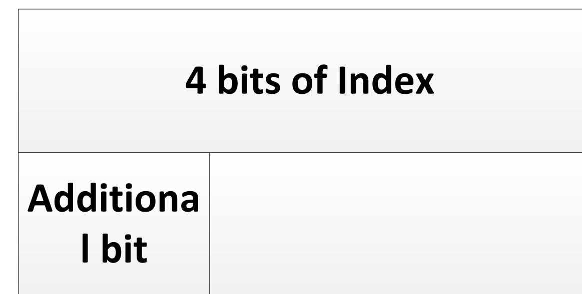
** single core point of view

Address Translation: Considerations

- TX side does not have to know the internal architecture of the RX side.
- The system was designed to be "generic" to enable support for future device architectures (for example, larger window).
- Result – Address translation is more generic and thus a little complex. This presentation will try to simplify it.

Address Translation: Overload

- Overload means using the same bit for more than one purpose.
- Result – Look up tables might require duplication.
- Example – if index to lookup table shares a bit with other value (security bit), the table must be duplicated.

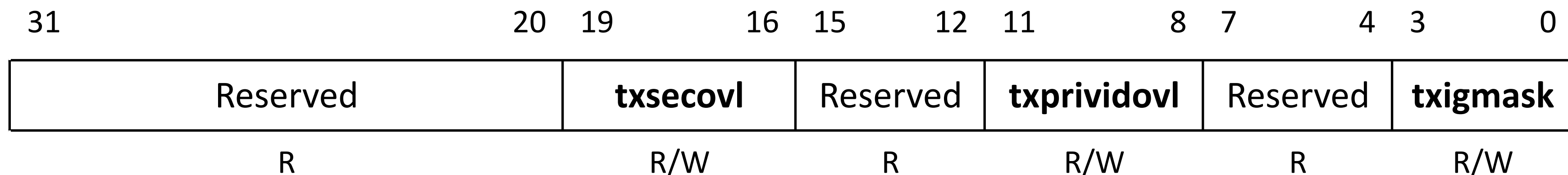


Value in the table in index 0xxx must be the same as the value in 1xxx

Address Manipulation: Tx Side Registers

Tx Address Overlay Control Register

- User configures PrivID / Security bit overload in this register
- Register is at address *HyperLinkCfgBase* + 0x1c. For 6678 that is 0x2140_001c
- If using HyperLink LLD, hyplnkTXAddrOvlyReg_s represents this register



Address Translation: Tx Side Registers

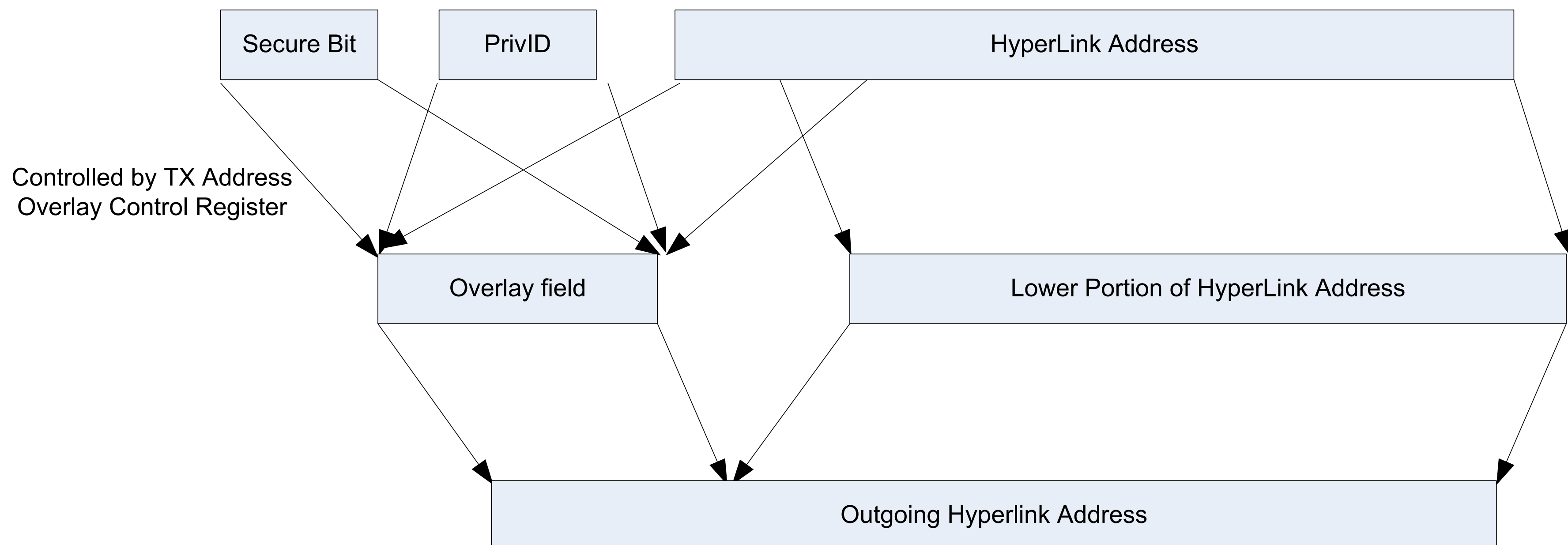
Register Field	Purpose	Bits	Range
txigmask	Selects mask that is logically ANDed to incoming address. Determines what address bits will be sent to remote side. Examples: 0 → mask = 0x0001_FFFF, 10 → mask = 0x07FF_FFFF	4	Mask varies from 0x 01ffff (value 0) to 0xffffffff (value 15)
txprividovl	Selects where PrivID will be placed in outgoing address Example: 12 → TxAddress [31-28] = PrivID [3-0]	4	4 bits (from 17-20 to 28-31) 3 bits (29-31) 2 its (30-31) 1 bit (31) 0 – no privID
txsecovl	Selects where Security Bit is placed in outgoing address	4	No security bit 1 bit (from bit 17 to 31)

Remember the Overloads!!!

Address Manipulation: Tx Side

Objective: Overlay control information onto address field. Control information consists of PrivID index and Security bit:

- PrivID index indicates which master is making the request.
  - PrivID index is 4 bits.
  - PrivID (on RX side) value is usually 0xD if request from core; 0xE if from other master
- Security bit indicates whether the transaction is secure or not.



Address Translation: Rx Side Registers

Rx Address Selector Control Register

- Register is at address *HyperLinkCfgBase* + 0x2c. For 6678, that is 0x2140_002c
- If using HyperLink LLD, hyplnkRXAddrSelReg_s represents this register

Rx Address Selector Control Register (more details in HyperLink User's Guide)

31–26	25	24	23–20	19–16	15–12	11–8	7–4	3–0
Reserved	**rxsechi**	**rxseclo**	Reserved	**rxsecsel**	Reserved	**rxprividsel**	Reserved	**rxsegsel**
R	R/W	R/W	R	R/W	R	R/W	R	R/W

Address Translation: Rx Side Registers

Register Field	Purpose	Bits	Range
rxsechi	Deals with secure signal	1	0-1
rxseclo	Deals with secure signal	1	0-1
rxsecsel	The overlay location of the secure signal bit	4	16-31
rxsegsel	Selects which bits of the incoming RxAddress to use as an index to lookup segment length and size from the Segment LUT. Depends on max. segment size. Example: rxsegsel=6 → use RxAddress [27-22] as index to LUT and the offset mask is 3fffff (22 bits offset address)	4	6 bits (17-22 to 26-31) 5 bits (27-31) 4 bits (28-31) 3 bits (29-31) 2 bits (30-31) 1 bits (31) 0 bits
rxprividsel	Selects which bits of the incoming RxAddress to use as PrivID index PrivID index is used as the row # to lookup PrivID value from LUT Example: rxprividsel=12 → RxAddress [31-28] as index to LUT	4	4 bits (17-20 to 28-31) 3 bits (29-31) 2 bits (30-31) 1 bit (31) 0 bits

Remember the Overloads!!!

HyperLink User's Guide – rxsegsel

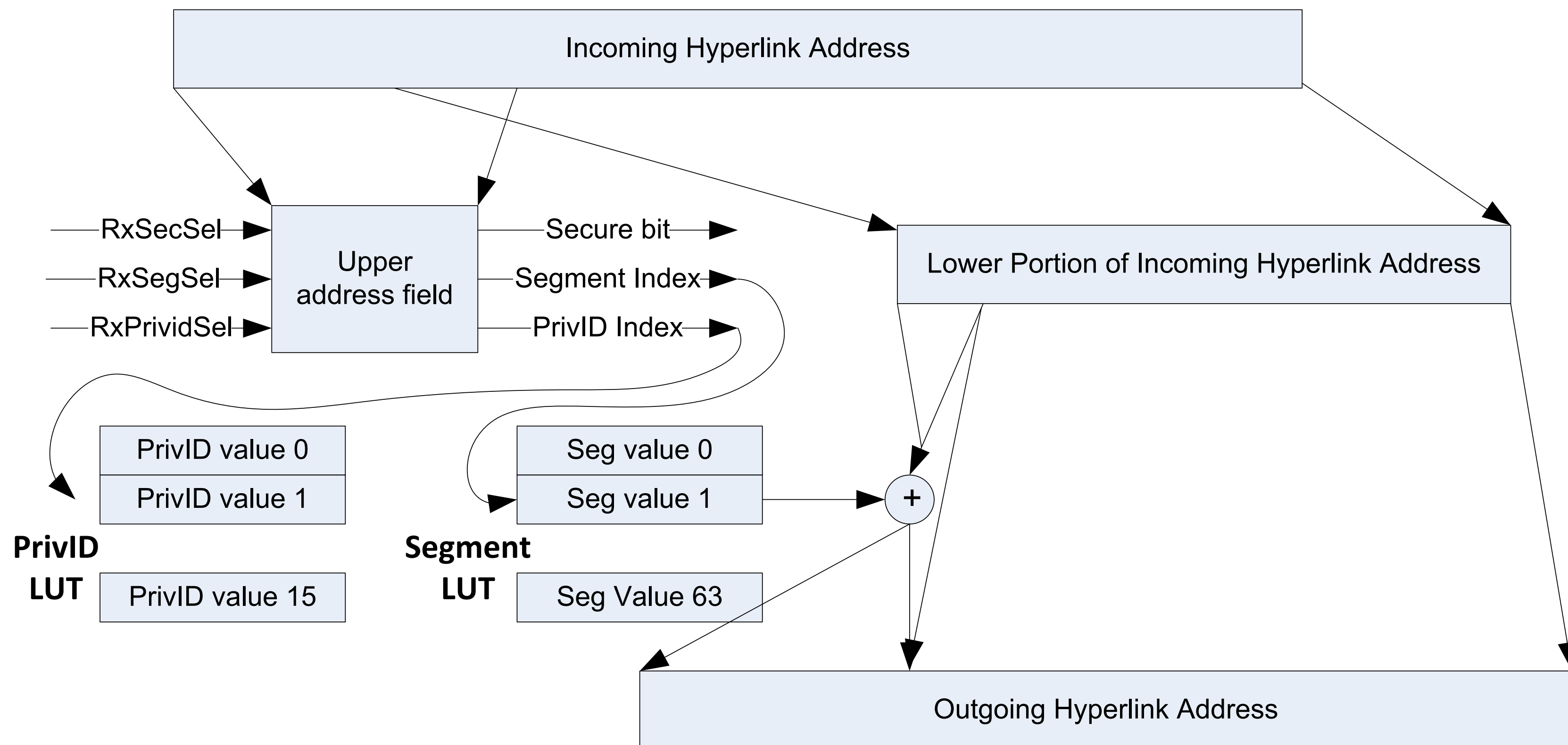
http://www.ti.com/lit/sprugw8

Table 3-10 gives the rxsegsel values. A typical line looks like the following:

if rxsegsel = 6 use RxAddress 27-22 as index to lookup segment/length table, use 0x003fffff as offset mask

Address Translation: Rx Side

Objective: Regenerate address mapped to remote memory space, along with Security bit and PrivID from incoming address, based on values in *Rx Address Selector Control Register* and LUTs.



Address Translation: Rx Side LUTs

SEGMENT LUT

hyplnkRXSegTbl_t [numSegments], with numSegments<=64 & power of 2

Each entry in the LUT consists of:

- 16-bit **rxSegVal**, the upper 16-bits of each segment's base address
- 5-bit **rxLenVal**, which represents the segment size as per table on the right and a mask

rxLenVal	Size
0 – 7	0
8	512B
. . .	. . .
21	4MB
. . .	. . .
27	256MB

Example Scenario

4 segments, 4 MB each, with base addresses:

- 0x8000_0000
- 0x8200_0000
- 0x8400_0000
- 0x8600_0000

Then Segment LUT will be:

Segment #	rxSegVal	rxLenVal
0	0x8000	21
1	0x8200	21
2	0x8400	21
3	0x8600	21

Address Translation: Rx Side LUTs

Privilege ID LUT

hyplnkRXPrivTbl_t [numPriv], with numPriv <=16 & power of 2

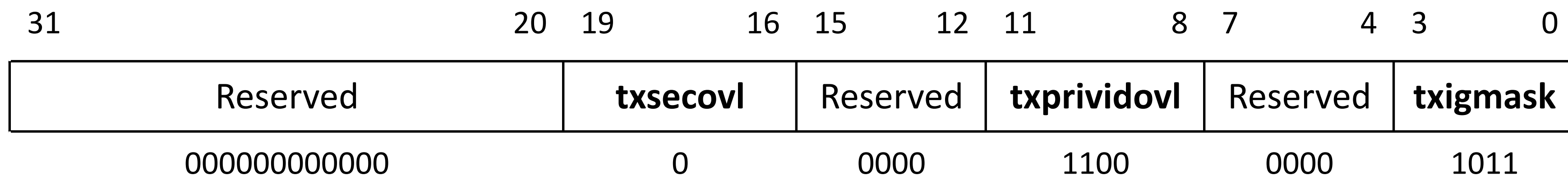
Each entry in the LUT consists of:

- A value between 0-15 that represent the privilege ID of the master
- Common use, value D if comes from any core, E if from any other master

Examples

We will now present several examples that can be used on KeyStone devices with the following limitations:

- No security bit
- The privilege ID index is in the 4 MSB of the address; bits 28-31
- We will cover the RX overlay registers, and the different LUTs
- On the TX side, always send the upper 28 bits, so that:
  - txsecovl = 0
  - txprividovl = 12 (bits 28-31)
  - txigmask = 11 (0x0fffffff)



31–20	19–16	15–12	11–8	7–4	3–0
Reserved	**txsecovl**	Reserved	**txprividovl**	Reserved	**txigmask**
000000000000	0	0000	1100	0000	1011

RX Side, Privilege LUT

The look-up table shown is for a privID with the following characteristics:

- All remote cores will have PrivID of D
- All other masters have ID of E
- 4 bits are used to express the PrivID index

Questions:

- What happens if there is a security bit in bit location 28?
- What if the security bit is in bit location 31?

NOTE: KeyStone II uses a fixed PrivID for remote HyperLink access. We strongly suggest the user fill all tables with the value 0xE (KeyStone II fixed value).

Index	Value
0	D = 1101
1	D = 1101
2	D = 1101
3	D = 1101
4	D = 1101
5	D = 1101
6	D = 1101
7	D = 1101
8	E=1110
9	E=1110
10	E=1110
11	E=1110
12	E=1110
13	E=1110
14	E=1110
15	E=1110

Address Translation: Example 1 (1/2)

Problem Statement: Build the Segment LUT for the following:

- Remote DDR 0x8000_0000 - 0x8FFF_FFFF
- One 256MB segment
- Accessible by all 16 masters on the local side

Solution:

1. Because the segment size is 256M, the offset mask must be 0x0fff ffff and thus, rxsegsel = 12. The index to lookup table is bits 28-31, and 0x0fffffff is the mask
2. It looks like the table should have only one, segment 0, rxSegVal = 0x8000, and rxLenVal = 27
3. No security bit
4. Privilege index can be any number from 0 to 15. In this example, (and all examples in the presentation), we use rxprividsel = 12; That is, bits 28-31.
5. **Notice the overlay of the master privilD on the index. This means that the segment index can be any number between 0 and 15. So the first line must be repeated 16 times.**

rxLenVal	Size
0 – 7	0
8	512 B
. . .	. . .
21	4M B
. . .	. . .
27	256 MB

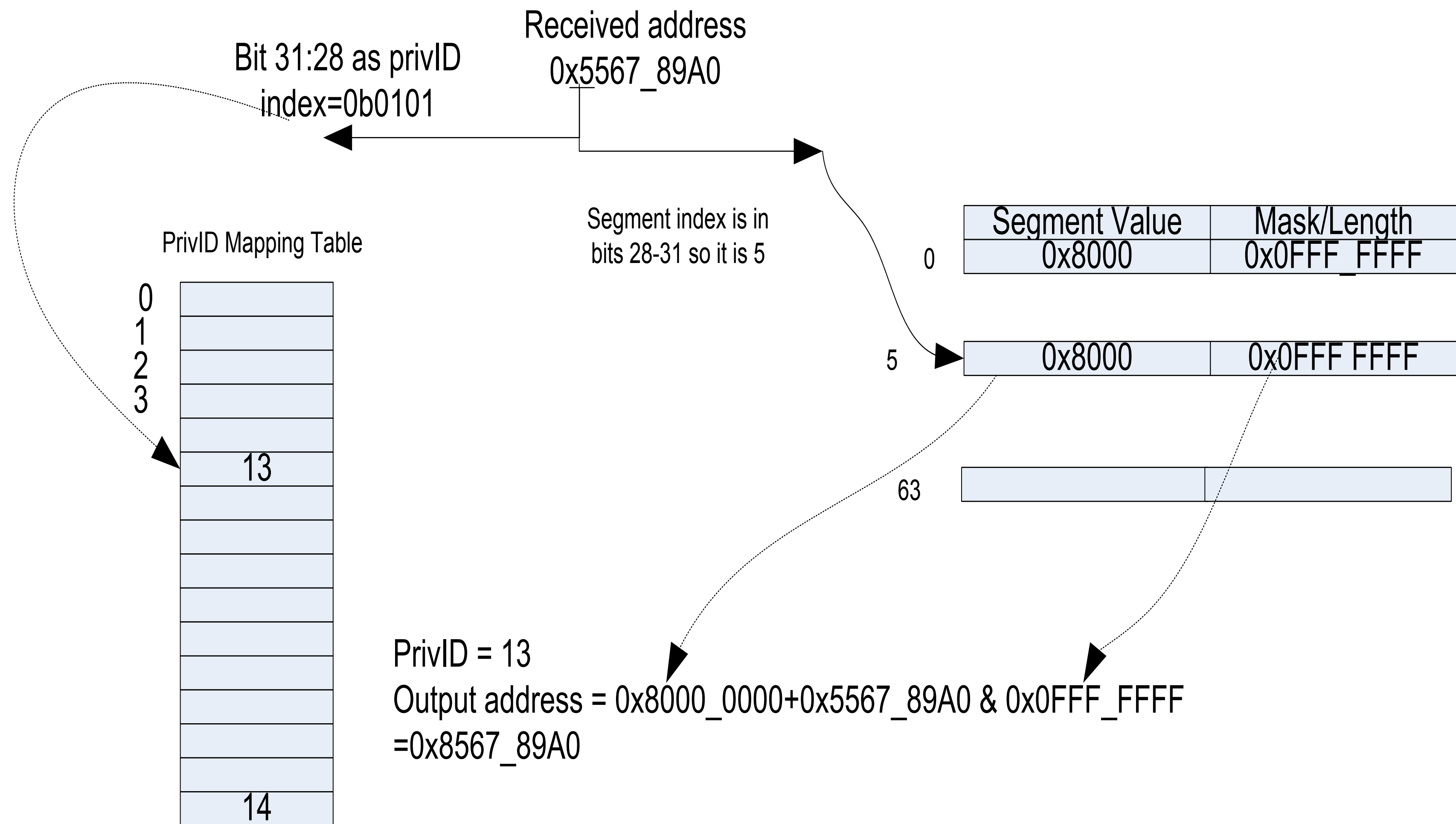
Address Translation: Example 1 (2/2)

Segment #	rxSegVal	rxLenVal
0	0x8000	237
1	0x8000	27
2	0x8000	27
3	0x8000	27
4	0x8000	27
5	0x8000	27
6	0x8000	27
7	0x8000	27

Segment #	rxSegVal	rxLenVal
8	0x8000	237
9	0x8000	27
10	0x8000	27
11	0x8000	27
12	0x8000	27
13	0x8000	27
14	0x8000	27
15	0x8000	27

Address Translation: Rx Side Example 1

- Choose a read or write address from Core 5 and address 4567 89a0:
- HyperLink Tx side builds the following address: 5567 89a0
- Following the previous example, what address will be read?



Address Translation: LUT Example 2

Problem Statement: Build the Segment LUT for the following scenario:

- 8 segments
- Each segment of size 0x0100_0000 (16MB) at 0x8000_0000, 0x8200_0000, … 0x8E00_0000

Solution

1. Because the segment size is 16M, the offset mask must be 0x00ff ffff and thus, rxsegsel = 8. The index to lookup table is bits 24-29, and 0x00ffffff is the mask.
2. The table should have 8 rows, each starting on a different address (0x8000_0000, 0x8200_0000, etc.), and a len of 23.
3. No security bit
4. Privilege index can be any number from 0 to 15. In this example, (and all examples in the presentation) we use rxprividsel = 12; That is, bits 28-31.

Address Translation: LUT Example 2

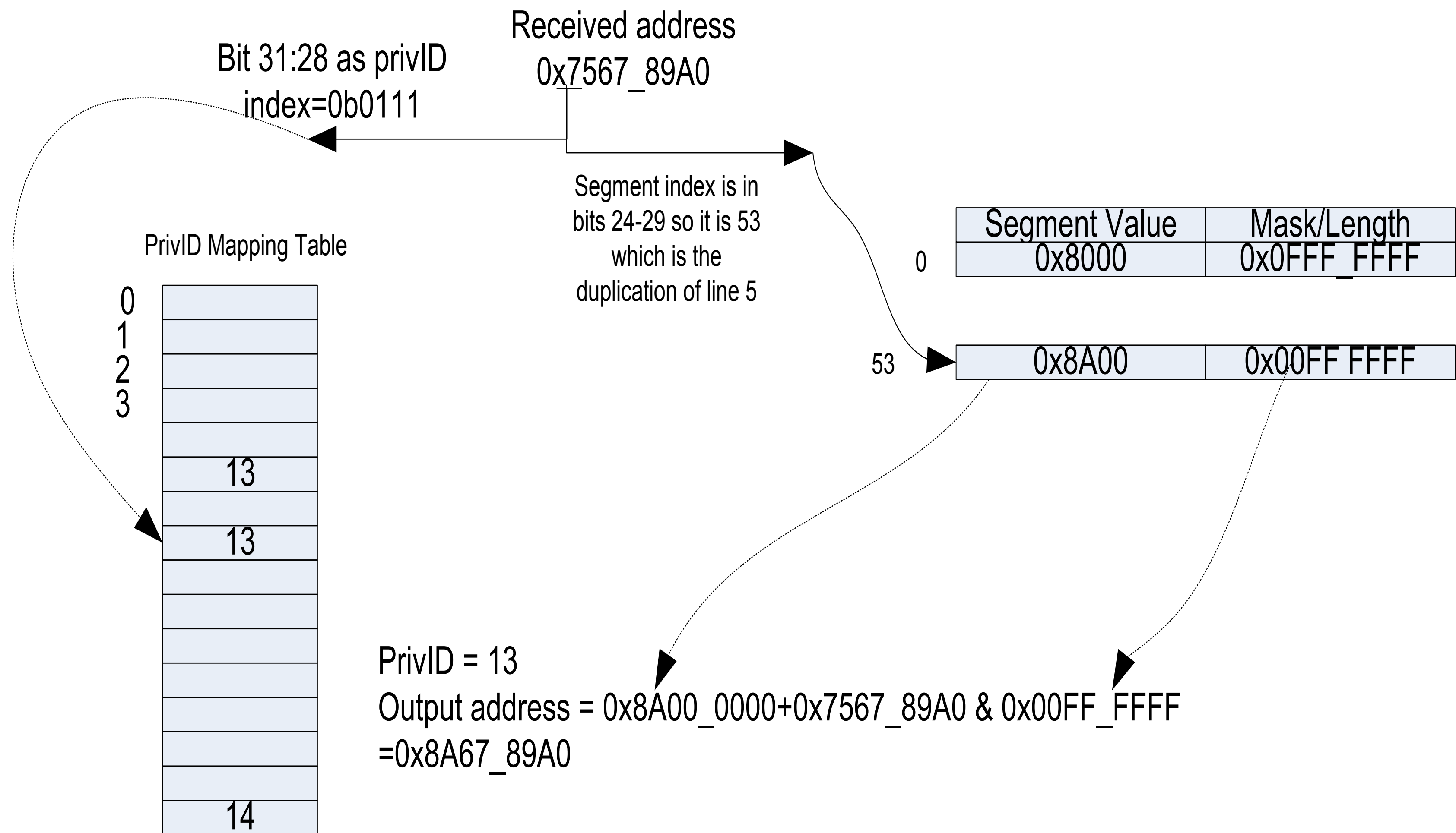
5. Notice the overlay of the master PrivID on the index. The last 2 bits of the index (bit 28-29) can be any value. So repeat the 8 rows 4 times at indexes XXYAAA, where A is the index into the table, A is supposed to be zero, and XX may be any number.
6. To prevent reading a wrong address, load the table rows in the lines that have Y=1 with zero memory.

Segment #	rxSegVal	rxLenVal	Segment #	rxSegVal	rxLenVal
0	0x8000	23	8	0x0000	0
1	0x8200	23	9	0x0000	0
2	0x8400	23	10	0x0000	0
3	0x8600	23	11	0x0000	0
4	0x8800	23	12	0x0000	0
5	0x8A00	23	13	0x0000	0
6	0x8C00	23	14	0x0000	0
7	0x8E00	23	15	0x0000	0

The table to the left will be repeated four times: 16-31, 32-47, 48-63

Address Translation: Rx Side Example 2

- Choose a read or write address from Core 7 and address 4567 89a0
- HyperLink Tx side builds the following address: 7567 89a0
- Following the previous example, what address will be read?



Address Translation: LUT Example 3

Problem Statement: Build the Segment LUT for the following scenario:

- 8 segments
- 7 of size 16MB at 0x8000_0000, 0x8100_0000
- 1 of size 32MB at 0x8700_0000

Solution:

1. Because the maximum segment size is 32M, the offset mask must be 0x01ff ffff and thus, rxsegsel = 9. The index to lookup table is bits 25-30 and 0x001fffff is the mask for the 32M. However, for the smaller size, the mask is different. For 16M, the mask is 0x000f ffff.
2. The table should have 8 rows, each starting on a different address (0x8000_0000, 0x8100_0000, etc.), and len of 23 where the last one will have len of 24.
3. No security bit
4. Privilege index can be any number from 0 to 15. In this example, (and all examples in the presentation) we use rxprividsel = 12; That is, bits 28-31.

Address Translation: LUT Example 3(2)

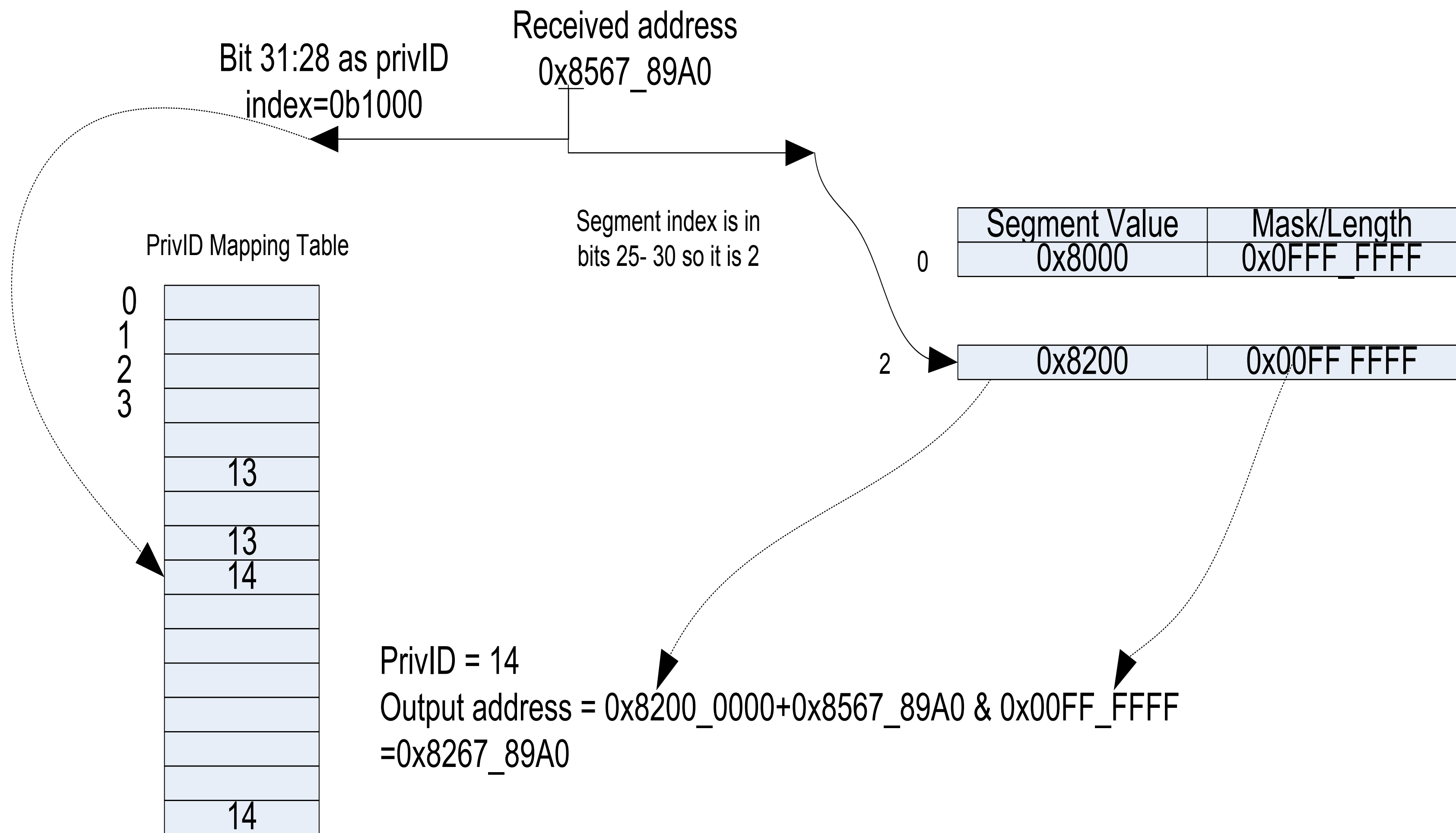
5. **Notice the overlay of the master PrivID on the index. The last 3 bits of the index (bit 28-30) can be any value. So we must repeat the 8 rows 8 times.**

Segment #	rxSegVal	rxLenVal	Segment #	rxSegVal	rxLenVal
0	0x8000	23	8	0x8000	23
1	0x8100	23	9	0x8100	23
2	0x8200	23	10	0x8200	23
3	0x8300	23	11	0x8300	23
4	0x8400	23	12	0x8400	23
5	0x8500	23	13	0x8500	23
6	0x8600	23	14	0x8600	23
7	0x8700	24	15	0x8700	24

The table to the left will be repeated 8 times 8-15, 16-23. 24-31, 32-39, 40-47, 48-55, 56-63

Address Translation: Rx Side Example 3

- Choose a read address from master with privilege 8 and address 4567 89a0.
- HyperLink Tx side builds the following address: 8567 89a0
- Following the previous example, what address will be read?



Address Translation: LUT Example 4

Problem Statement: Build the Segment LUT for C6678 device with the following scenario:

- 9 segments
- 1st segment of 4MB in MSMC
- 2nd to 9th segments of 512KB in L2 memory of each core

Solution:

1. Because the maximum segment size is 4M, the offset mask must be 0x003f ffff and thus, rxsegsel = 6. The index to the lookup table is bits 22-26 and 0x03f ffff is the mask for the 4M. However, for the smaller size, the mask is different. For 512K, the mask is 0x07 ffff.
2. The table should have 16 rows. The first one starts at 0x0c00 0000 with len of 21 (4M), 8 rows each starting at 0x1N80_0000 (N = 0 to 7) with len of 18, and 7 dummy rows of len=0.
3. No security bit
4. Privilege index can be any number from 0 to 15. In this example, (and all examples in the presentation), we use rxprividsel = 12; That is, bits 28-31.

Address Translation: LUT Example 4(2)

Segment #	rxSegVal	rxLenVal	Segment #	rxSegVal	rxLenVal
0	0x0C00	21	8	0x1780	18
1	0x1080	18	9	0x0000	0
2	0x1180	18	10	0x0000	0
3	0x1280	18	11	0x0000	0
4	0x1380	18	12	0x0000	0
5	0x1480	18	13	0x0000	0
6	0x1580	18	14	0x0000	0
7	0x1680	18	15	0x0000	0

No overlay … but to prevent errors, you must either:

- Fill the table with zero rows

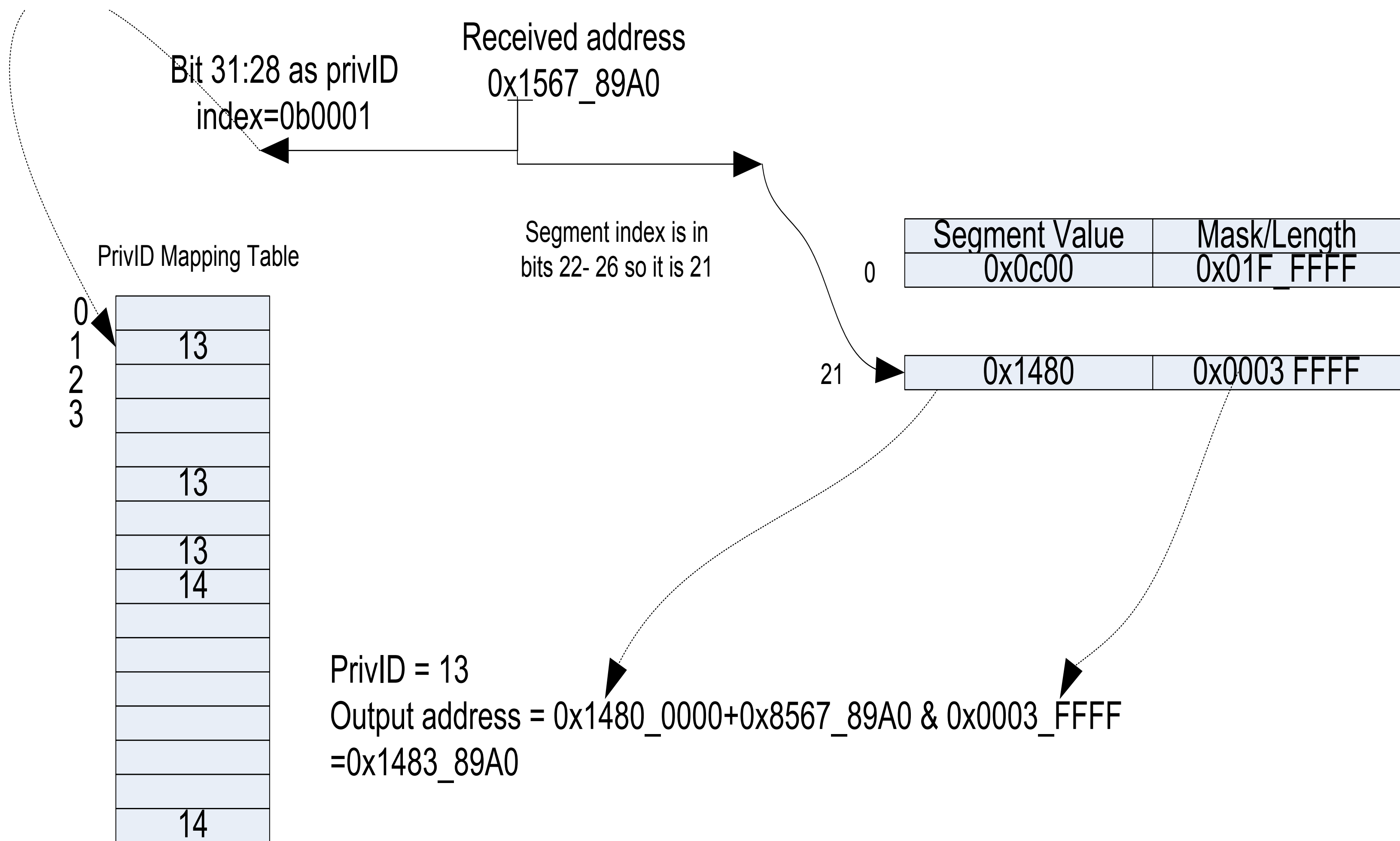
  or

- Duplicate the 16 rows 4 times.

In this example, we duplicate the 16 rows 4 times

Address Translation: Rx Side Example 4

- Choose a read address from Core 1 and address 4567 89a0.
- HyperLink Tx side builds the following address: 1567 89a0
- Following the previous example, what address will be read?



Address Translation: Rx Side Registers

Five registers control the behavior of the Rx side:

1. Rx Address Selector Control (base + 0x2c)
   Controls how the address word is decoded; hyplnkRXAddrSelReg_s

2. Rx Address PrivID Index (base + 0x30)
   Used to build/read Privilege Lookup Table; hyplnkRXPrivIDIdxReg_s

3. Rx Address PrivID Value (base + 0x34)
   Used to build Privilege Lookup Table; hyplnkRXPrivIDValReg_s

4. Rx Address Segment Index (base + 0x38)
   Used to build/read Segment Lookup Table; hyplnkRXSegIdxReg_s

5. Rx Address Segment Value (base + 0x3c)
   Used to build Segment Lookup Table; hyplnkRXSegValReg_s

Address Translation: Rx Side Registers

To program the LUT:

- *Write* to Rx Address PrivID/Segment Index Register.
- *Write* to Rx Address PrivID/Segment Value Register, which will populate the corresponding index in the LUT with this value.

To check LUT content:

- *Write* to Rx Address PrivID/Segment Index Register.
- *Read* Rx Address PrivID/Segment Value Register, which will return value from LUT for index specified in Index Register.

Address Translation: Summary

Translation process inputs on the local/transmit side:

1. 28 bits of remote address (the upper 4 bits are 0x4)
2. Privilege ID and Secure Bit

Process information sent from local to remote/receive side:

1. Lower portion of remote address – offset into segment
2. Segment Index
3. Privilege ID
4. Secure Bit

Translation process outputs on the remote/receive side:

1. Complete remote address
2. Privilege ID

Configuration

- Overview
- Address Translation
- **Configuration**
- Performance
- Example

Configuration: Typical Flow

Application typically follows this flow to enable & configure HyperLink:

1. PLL, Power, and SerDes:
   a) Setup PLL.
   b) Enable power domain for HyperLink.
   c) Configure SerDes.
   d) Confirm that power is enabled.
2. Register Configurations:
   a) Enable HyperLink via HyperLink Control Register (base + 0x4).
   b) Once the link is up, both devices can see each other's registers.
      Here there are three choices:
      i. Device configures own registers
      ii. One master programs registers for both devices
      iii. Direction-based
   c) Register configuration involves specifying **address translation** scheme on Tx and Rx side, and any event/interrupt configuration.

Configuration: APIs

Chip Support Library (CSL) and HyperLink Low-Level Drivers (LLD) make available APIs that can be used to configure HyperLink.

General recommendations:

- Wherever LLD functions are available to do something, use LLD.
- If LLD API does not exist for what you want to achieve, use CSL.
- Leverage functions from the HyperLink LLD example project.

Configuration: Typical Flow, Step 1

1. Enable power domain for peripherals using CSL routines.

   Enabling power to peripherals involves the following four functions:

   CSL_PSC_enablePowerDomain()

   CSL_PSC_setModuleNextState()

   CSL_PSC_startStateTransition()

   CSL_PSC_isStateTransitionDone()

2. Reset the HyperLink and load the boot code for the PLL.

   Write 1 to the reset field of control register (address base + 0x04)

   CSL_BootCfgUnlockKicker();

   CSL_BootCfgSetVUSRConfigPLL ()

3. Configure the SERDES.

   CSL_BootCfgVUSRRxConfig()

   CSL_BootCfgVUSRTxConfig()

Configuration: Typical Flow, Step 2

1. HyperLink Control registers
2. Interrupt registers
3. Lane Power Management registers
4. Error Detection registers
5. SerDes Operation registers
6. Address Translation registers

Configuration: HyperLink LLD APIs

hyplnkRet_e Hyplnk_open (int portNum, Hyplnk_Handle *pHandle)

Hyplnk_open creates/opens a HyperLink instance.

hyplnkRet_e Hyplnk_close (Hyplnk_Handle *pHandle)

Hyplnk_close Closes (frees) the driver handle.

hyplnkRet_e Hyplnk_readRegs (Hyplnk_Handle handle, hyplnkLocation_e location, hyplnkRegisters_t *readRegs)

Performs a configuration read.

hyplnkRet_e Hyplnk_writeRegs (Hyplnk_Handle handle, hyplnkLocation_e location, hyplnkRegisters_t *writeRegs)

Performs a configuration write.

hyplnkRet_e Hyplnk_getWindow (Hyplnk_Handle handle, void **base, uint32_t *size)

Hyplnk_getWindow returns the address and size of the local memory window.

uint32_t Hyplnk_getVersion (void)   Hyplnk_getVersion

returns the HYPLNK LLD version information.

const char * Hyplnk_getVersionStr (void)   Hyplnk_getVersionStr

returns the HYPLNK LLD version string.

Configuration: HyperLink LLD Example API

```
hyplnkRet_e Hyplnk_writeRegs ( Hyplnk_Handle          handle,
                               hyplnkLocation_e       location,
                               hyplnkRegisters_t *    writeRegs
                             )
```

Performs a configuration write.

Writes one or more of the device registers

It is the users responsibility to ensure that no other tasks or cores will modify the registers while they are read, or betwen the time the registers are read and they are later written back.

The user will typically use **Hyplnk_readRegs** to read the current values in the registers, modify them in the local copies, then write back using **Hyplnk_writeRegs**.

It is guaranteed that all registers can be written together. The actual ordering will, for example, write index registers before the associated value registers

On exit, the actual written values are returned in each register's reg->raw.

Since the peripheral is shared across the device, and even between peripherals, it is not expected to be dynamically reprogramed (such as between thread or task switches). It should only be reprogrammed at startup or when changing applications. Therefore, there is a single-entry API instead of a set of inlines since it is not time-critical code.

Return values:

hyplnkRet_e status

Parameters:

handle	[in] The HYPLNK LLD instance identifier
location	[in] Local or remote peripheral
writeRegs	[in] List of registers to write

Configuration: HyperLink LLD Data Structures

hyplnkChipVerReg_s	Specification of the Chip Version Register
hyplnkControlReg_s	Specification of the HyperLink Control Register
hyplnkECCErrorsReg_s	Specification of the ECC Error Counters Register
hyplnkGenSoftIntReg_s	Specification of the HyperLink Generate Soft Interrupt Value Register
hyplnkIntCtrlIdxReg_s	Specification of the Interrupt Control Index Register
hyplnkIntCtrlValReg_s	Specification of the Interrupt Control Value Register
hyplnkIntPendSetReg_s	Specification of the HyperLink Interrupt Pending/Set Register
hyplnkIntPriVecReg_s	Specification of the HyperLink Interrupt Priority Vector Status/Clear Register
hyplnkIntPtrIdxReg_s	Specification of the Interupt Control Index Register
hyplnkIntPtrValReg_s	Specification of the Interrupt Control Value Register
hyplnkIntStatusClrReg_s	Specification of the HyperLink Interrupt Status/Clear Register
hyplnkLanePwrMgmtReg_s	Specification of the Lane Power Management Control Register
hyplnkLinkStatusReg_s	Specification of the Link Status Register
hyplnkRegisters_s	Specification all registers
hyplnkRevReg_s	Specification of the HyperLink Revision Register
hyplnkRXAddrSelReg_s	Specification of the Rx Address Selector Control Register
hyplnkRXPrivIDIdxReg_s	Specification of the Rx Address PrivID Index Register
hyplnkRXPrivIDValReg_s	Specification of the Rx Address PrivID Value Register
hyplnkRXSegIdxReg_s	Specification of the Rx Address Segment Index Register
hyplnkRXSegValReg_s	Specification of the Rx Address Segment Value Register
hyplnkSERDESControl1Reg_s	Specification of the SerDes Control And Status 1 Register
hyplnkSERDESControl2Reg_s	Specification of the SerDes Control And Status 2 Register
hyplnkSERDESControl3Reg_s	Specification of the SerDes Control And Status 3 Register
hyplnkSERDESControl4Reg_s	Specification of the SerDes Control And Status 4 Register
hyplnkStatusReg_s	Specification of the HyperLink Status Register
hyplnkTXAddrOvlyReg_s	Specification of the Tx Address Overlay Control Register

Performance

- Overview
- Address Translation
- Configuration
- **Performance**
- Example

HyperLink Performance

Silicon Results with C6678

Theoretical bound is 35.56 Gbps
Results are in 31.39 – 34.53 Gbps range

Payload (bytes)	Payload (bits)	No. of Lanes	SRC/DST	AET for Wr	Actual Throughput (Wr) Gbps
4096	32768	4	L2/DDR3	954	34.35
8192	65536	4	L2/DDR3	2088	31.39
16384	131072	4	L2/DDR3	3975	32.97
32768	262144	4	L2/DDR3	7592	34.53

Example

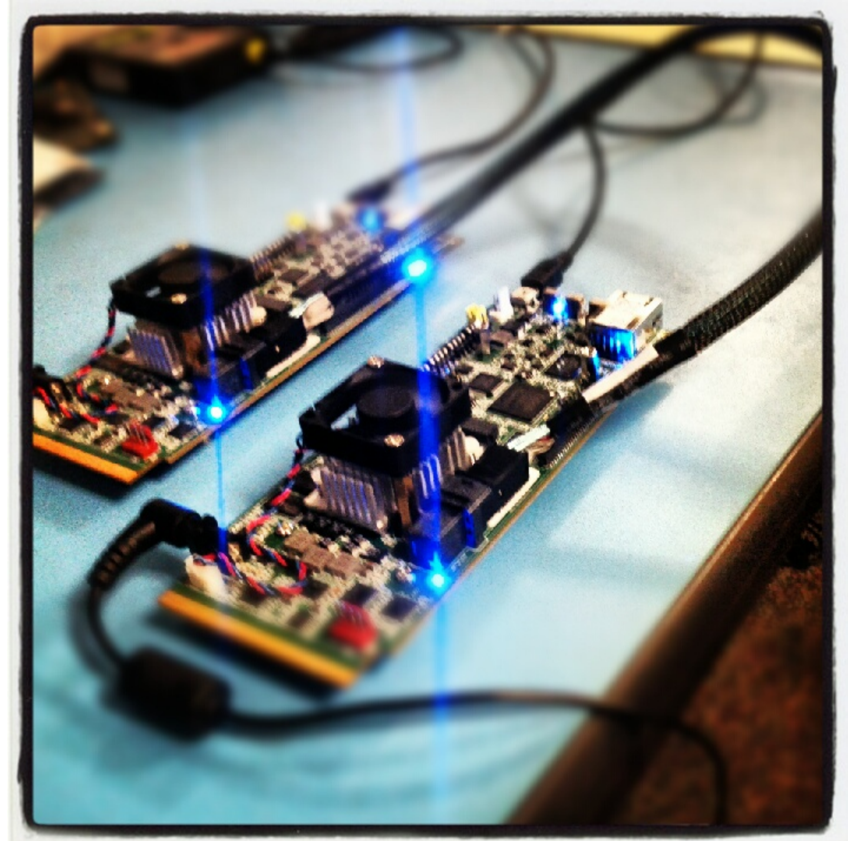
- Overview
- Address Translation
- Configuration
- Performance
- **Example**

HyperLink Example: Demo

- When you install TI's Multicore Software Development Kit (MCSDK), one of the packages it installs is the Platform Development Kit (PDK).
- Path to example: **`pdk_C6678_x_x_x_xx\packages\ti\drv\exampleProjects\hyplnk_exampleProject`**
- Example can be run in loopback mode on one 6678, or in 6678-to-6678 mode
- The mode is defined using a loopback flag in header file *hyplnkLLDCfg.h,* as:

```
#define hyplnk_EXAMPLE_LOOPBACK
```

- We will now switch to CCS to run the example in a board-to-board mode. The two 6678 EVMs are connected with a HyperLink external cable, as shown in the picture.



HyperLink Example: Leverage Functions

- Useful configuration functions are part of the HyperLink example and can be used "as is" or be modified by users.

  PDK_INSTALL_PATH\ti\drv\hyplnk\example\common\hyplnkLLDIFace.c

- Some of the configuration functions are:
  - **hyplnkRet_e hyplnkExampleAssertReset (int val)**
  - **Void hyplnkExampleSerdesCfg (uint32_t rx, uint32_t tx)**
  - **hyplnkRet_e hyplnkExampleSysSetup (void)**
  - **Void hyplnkExampleEQLaneAnalysis (uint32_t lane, uint32_t status)**
  - **hyplnkRet_e hyplnkExamplePeriphSetup (void)**

For More Information

- Refer to the Keystone HyperLink User's Guide
- Connect HyperLink C66x to FPGA using the Integretek IP-HyperLink core.
- Device-specific Data Manuals for the KeyStone SoCs can be found at TI.com/multicore.
- Multicore articles, tools, and software are available at Embedded Processors Wiki for the KeyStone Device Architecture.
- View the complete C66x Multicore SOC Online Training for KeyStone Devices, including details on the individual modules.
- For questions regarding topics covered in this training, visit the support forums at the TI E2E Community website.

BACKUP SLIDES

HyperLink Performance: Theoretical bound

Theoretical bound calculation on write throughput for HyperLink:

6678 does 8b/9b encoding, therefore

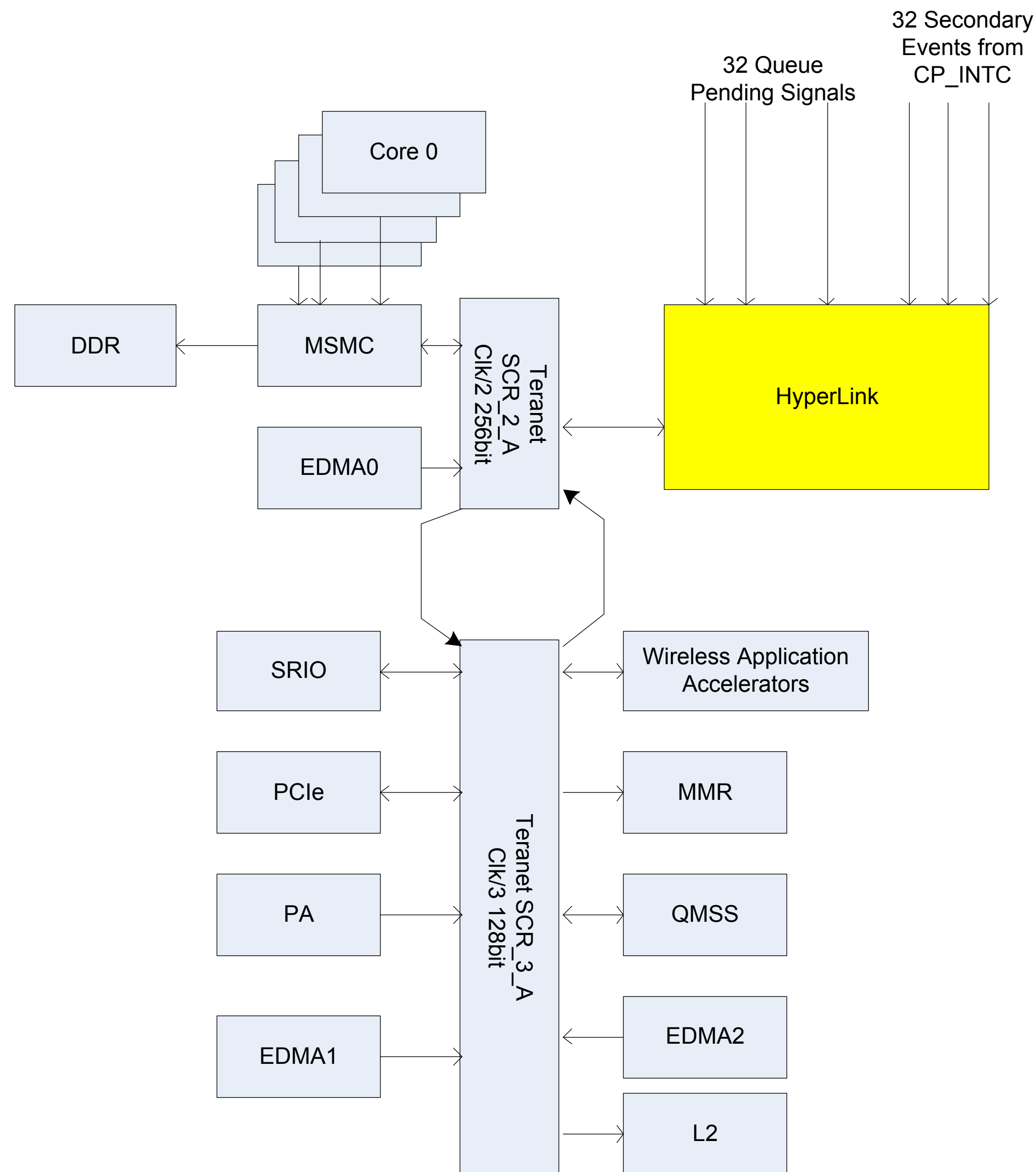
Useful data bandwidth = 50 x 8 / 9 = 44.44 Gbps

16bytes header for every 64bytes of data (max. write burst)

Effective max. data write throughput = 44.44 * 64/(64+16)

= **35.56 Gbps**

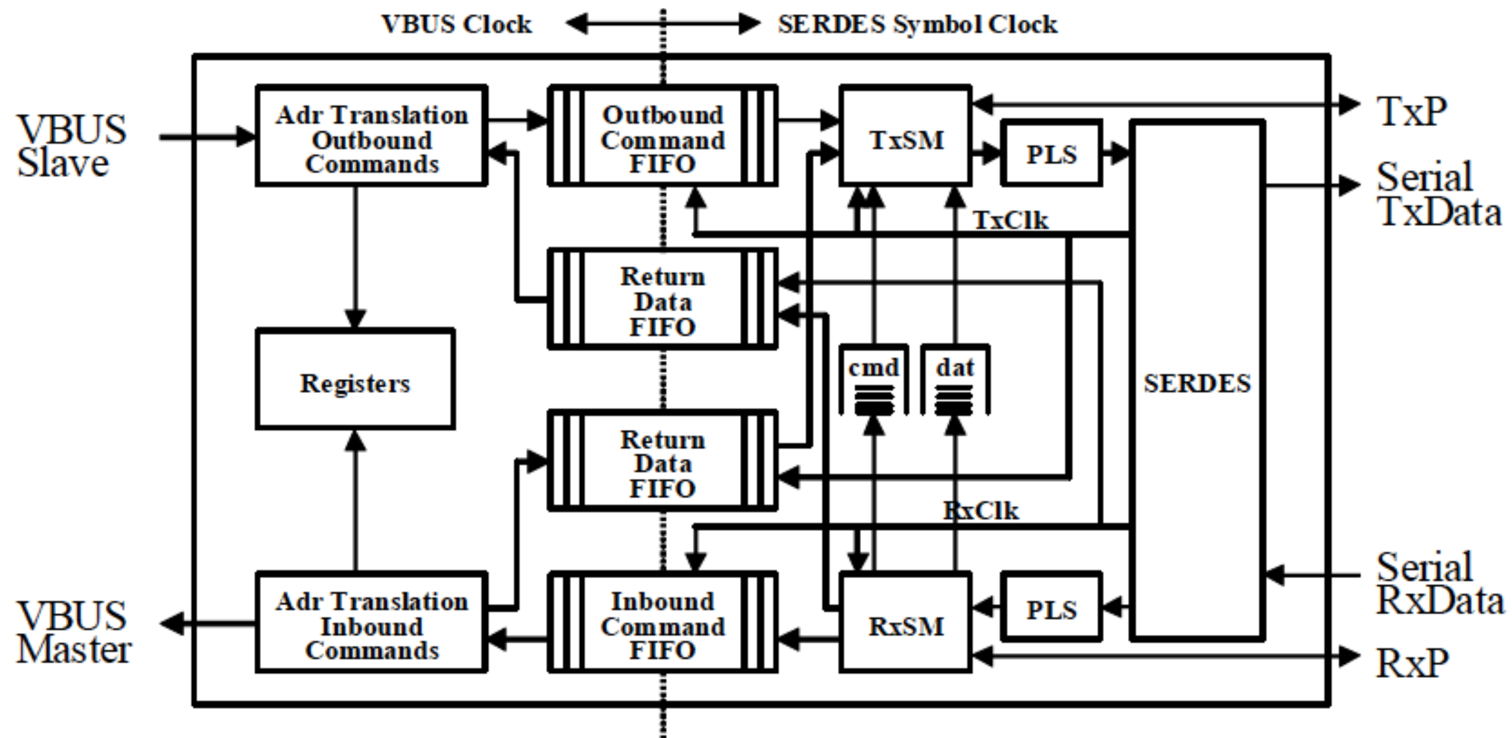
Overview: TeraNet Connections & Interrupts



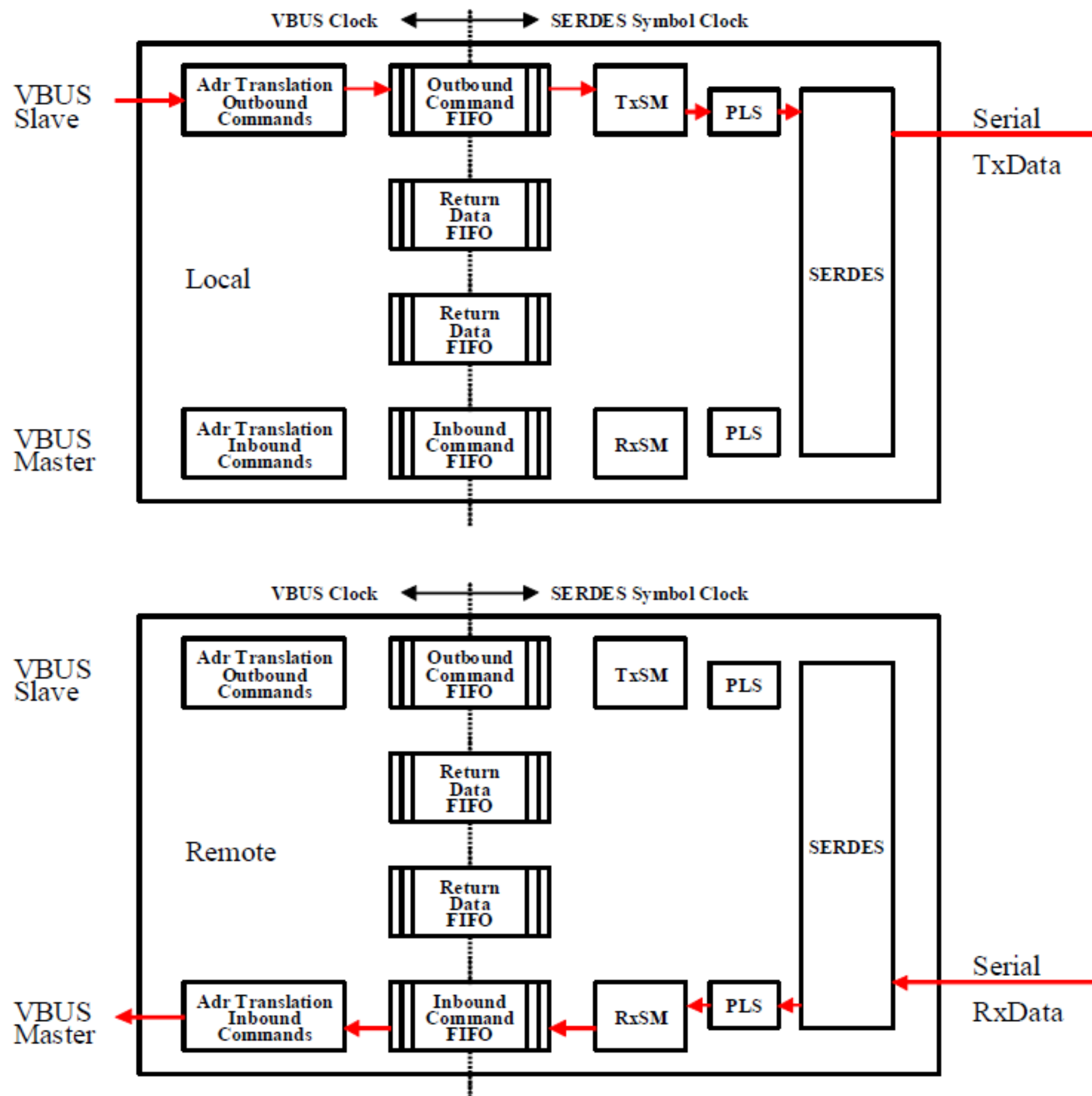
Overview: HyperLink Interrupts

- Detection - detected an interrupt to the HyperLink local device that was generated either as software interrupt (writing to interrupt register) or as hardware
- Forward – generate an interrupt packet and send it to the remote unit
- Mapping – receive an interrupt packet from the remote and forward it to the configure location in the local device
- Generating – generate an interrupt in the local device

Address Translation: Block Diagram



Protocol: Write Operation



Address Translation: Rx Side Example 1

Solution Explained

- 256MB segment → 28-bit offset → mask = 0x0FFF_FFFF
- 0x0567 89a0 address
- Bits 28-31 → 0b0101 = 5
- txigmask = 11 mask 0x0FFF_FFFF
- Address sent to the receive/remote side = 0x5567_89a0

On the receive side, the address is
0x8000_0000 + 0x0567_89a0 = 0x8567_89a0

Address Translation: Rx Side Example 2

Solution Explained

- 8 segments, each segment of size 0x0100_0000 (16M)
- Addresses start at 0x8000_0000, 0x8200_0000, 0x8400_0000, to 0x8E00_0000
- 24 bits offset – 0x067_89a0
- Segment number 0101 = 5

Row 5 0x8A00_0000 Size 23 (mask = 0x00ff ffff)

On the receive side,
the address is 0x8A00_0000 + 0x0067_89A0 = 0x8A67_89A0

Address Translation: Rx Side Example 3

Solution Explained

- 8 segments, 7 each of size 0x0100_0000 (16M)
- Addresses start at 0x8000_0000, 0x8100_0000, 0x8200_0000, to 0x8600_0000.
- For 8 segments, the maximum size is 32M. That is, 25 bits.
- 25 bits offset, 3 bits segment number 010 = 2

Row 2 0x8200_0000 Size 23 (mask = 0x00ff ffff)

On the receive side,
the address is 0x8200_0000 + 0x0067_89A0 = 0x8267_89A0

Address Translation: Rx Side Example 4

Solution Explained

- 9 segments
  - The first 8 segments are L2 memory of each core (512K = 19 bits).
  - The 9th segment is the MSMC (4M = 22 bits).
- The maximum size is 4M. That is, 22 bits.
- 6 bits to choose the segment (64 segments)
- 22 bits offset Segment number 010101 = 21 ????

Row 5   0x1480 0000   Size 18

On the receive side,
address is 0x1480 0000 + 0x0007 89a0 = 0x1487 89a0 (L2, Core 4)

HyperLink Example: SerDes Configuration

```
/**********************************************************************
 * Sets the SERDES configuration registers
 *********************************************************************/
void hyplnkExampleSerdesCfg (uint32_t rx, uint32_t tx)
{
  CSL_BootCfgUnlockKicker();

  CSL_BootCfgSetVUSRRxConfig (0, rx);
  CSL_BootCfgSetVUSRRxConfig (1, rx);
  CSL_BootCfgSetVUSRRxConfig (2, rx);
  CSL_BootCfgSetVUSRRxConfig (3, rx);

  CSL_BootCfgSetVUSRTxConfig (0, tx);
  CSL_BootCfgSetVUSRTxConfig (1, tx);
  CSL_BootCfgSetVUSRTxConfig (2, tx);
  CSL_BootCfgSetVUSRTxConfig (3, tx);


} /* hyplnkExampleSerdesCfg */
```