C66x KeyStone Training
HyperLink

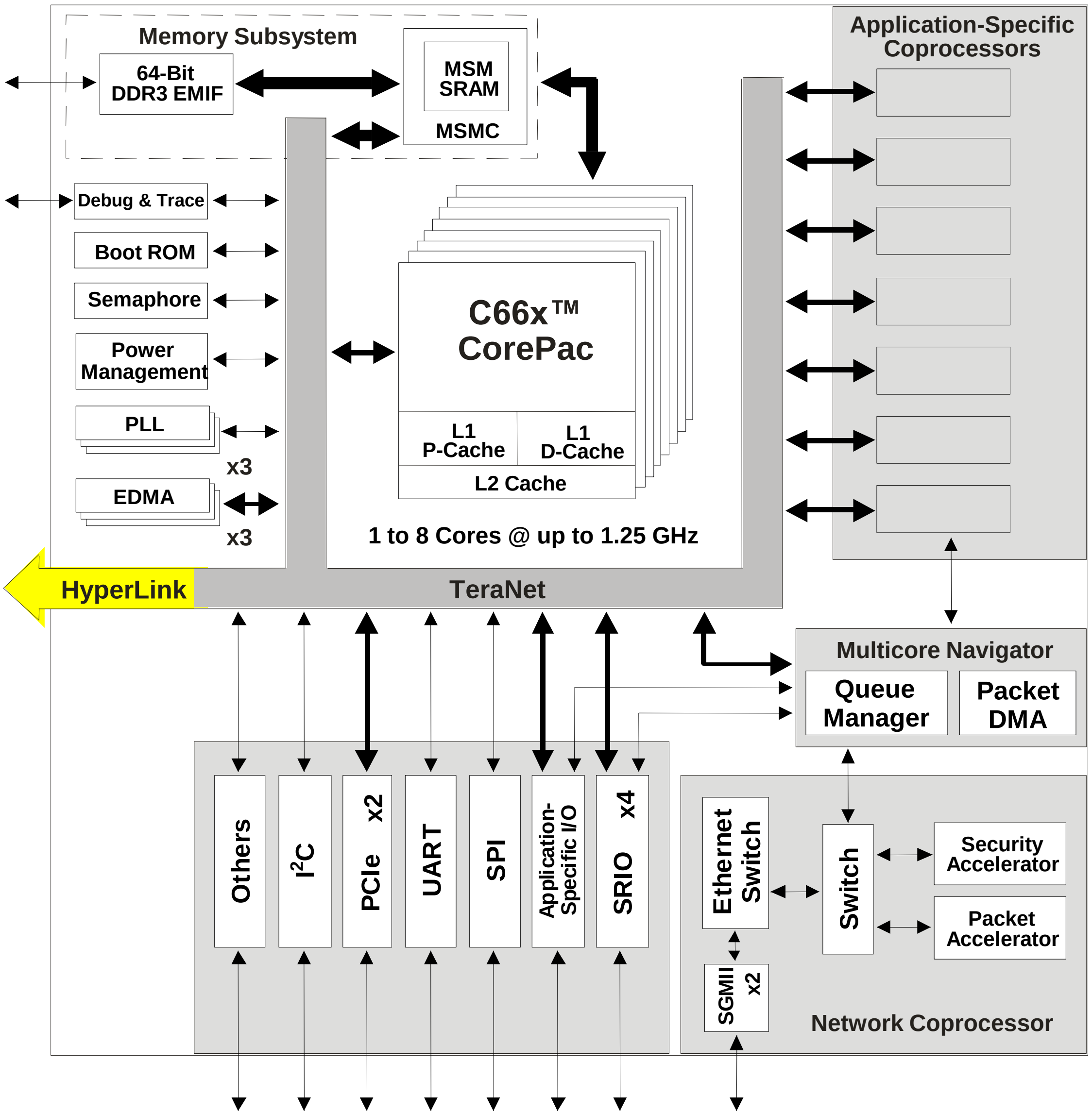
Agenda

1. HyperLink Overview
2. Address Translation
3. Configuration
4. Example and Demo

Agenda

1. **HyperLink Overview**
2. Address Translation
3. Configuration
4. Example and Demo

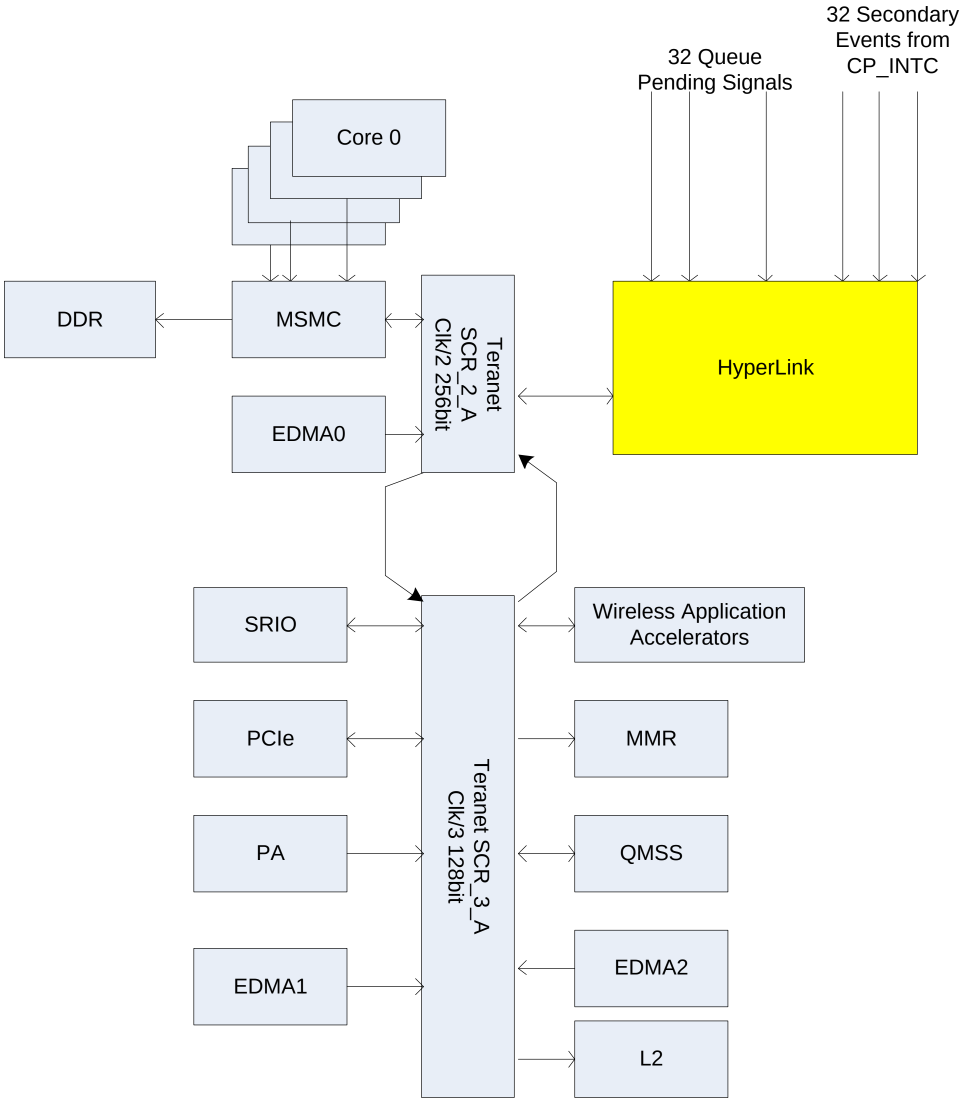
HyperLink Bus



CorePac & Memory Subsystem
Memory Expansion
Multicore Navigator
Network Coprocessor
External Interfaces
TeraNet Switch Fabric
Diagnostic Enhancements
HyperLink Bus

- Provides the capability to expand the C66x to include hardware acceleration or other auxiliary processors
- Four lanes with up to 12.5 Gbaud per lane

HyperLink in KeyStone



HyperLink Advantages

- Expands internal bus across chip boundaries
  - Fast (50 Gbaud)
  - Low power (50+% saving compared to other serial interfaces)
  - Low latency
  - Industry standard SerDes
  - Future support for FPGA

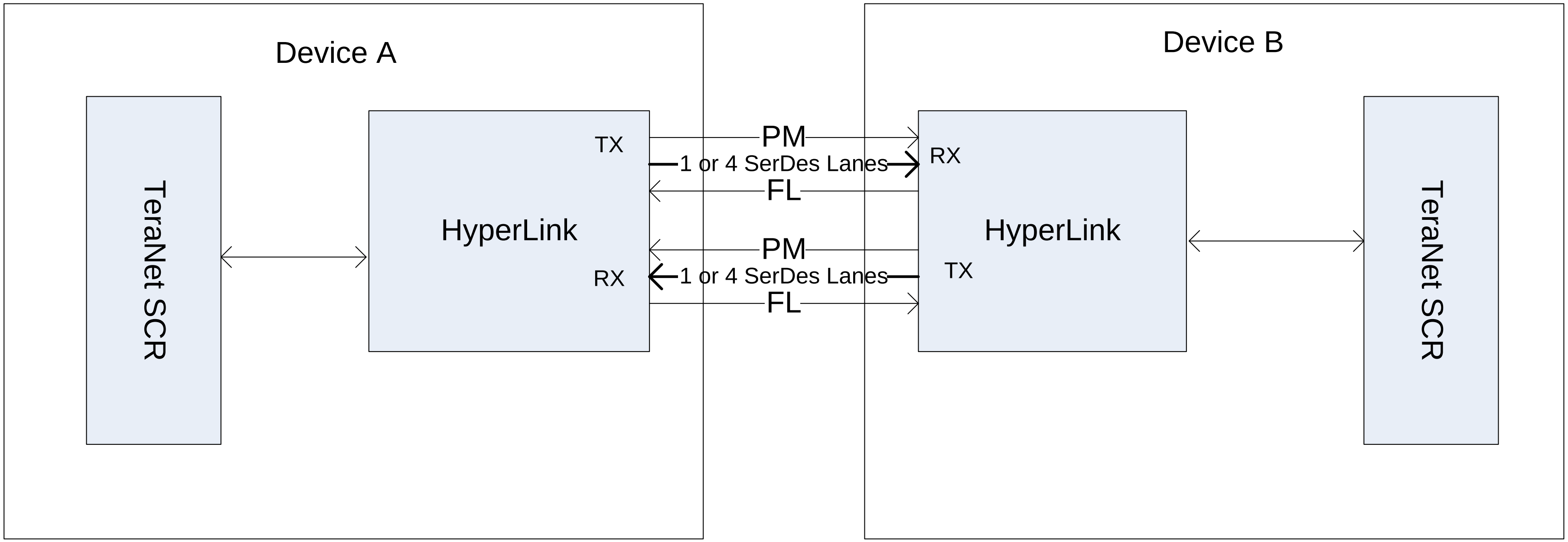
Lower Power
Scalable Processing Expansion

- Many use cases
  - Remote access of accelerators
  - Expand processing capability by adding 4 or 8 cores
  - Reduce system power by disabling I/O, accelerators on remote device

1 Cortex A8
4 DSP cores

4 – 8 DSP cores

HyperLink External Interfaces



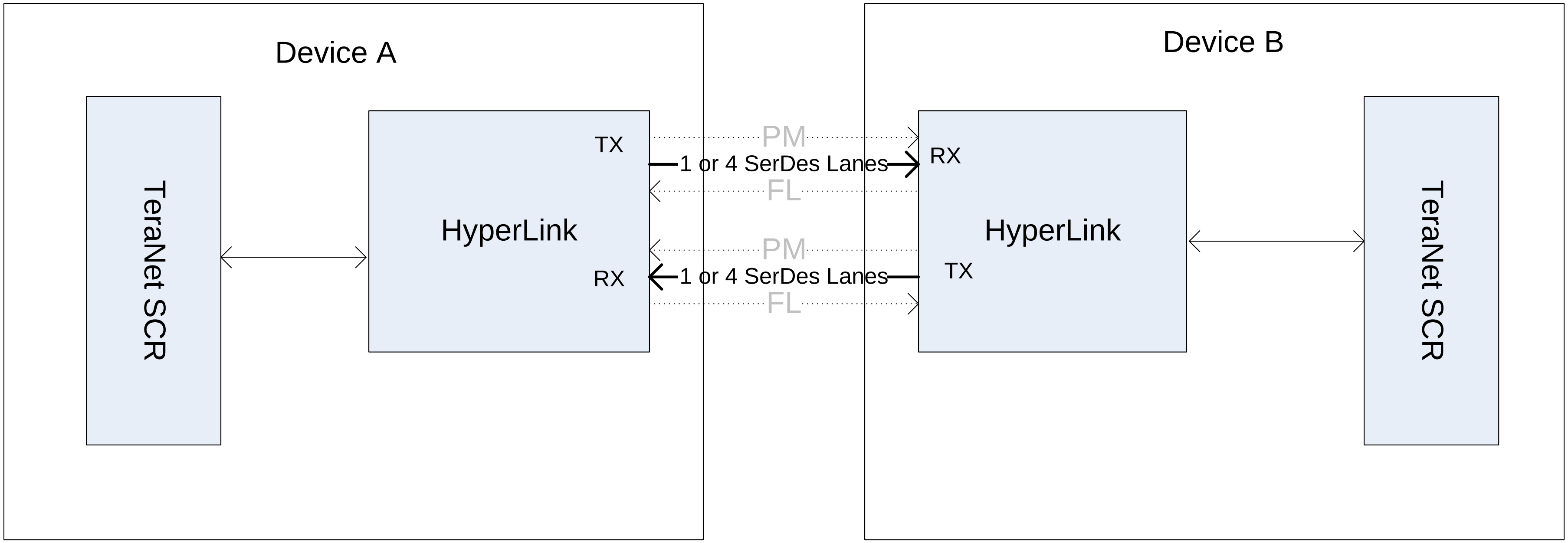
Sideband:

- LVCMOS signal for control
- Dedicated control for each direction
- Flow Control (FL) and Power Management (PM)

Data:

- SerDes: 1 or 4 lanes
- Supports up to 12.5 GBaud per lane

HyperLink External Interfaces



Sideband:

- LVCMOS signal for control
- Dedicated control for each direction
- Flow Control (FL) and Power Management (PM)

Data:

- SerDes: 1 or 4 lanes
- Supports up to 12.5 GBaud per lane

NOTE: The PM and FL are transparent to the user after setting the registers.

Packet-Based Transfer Protocol

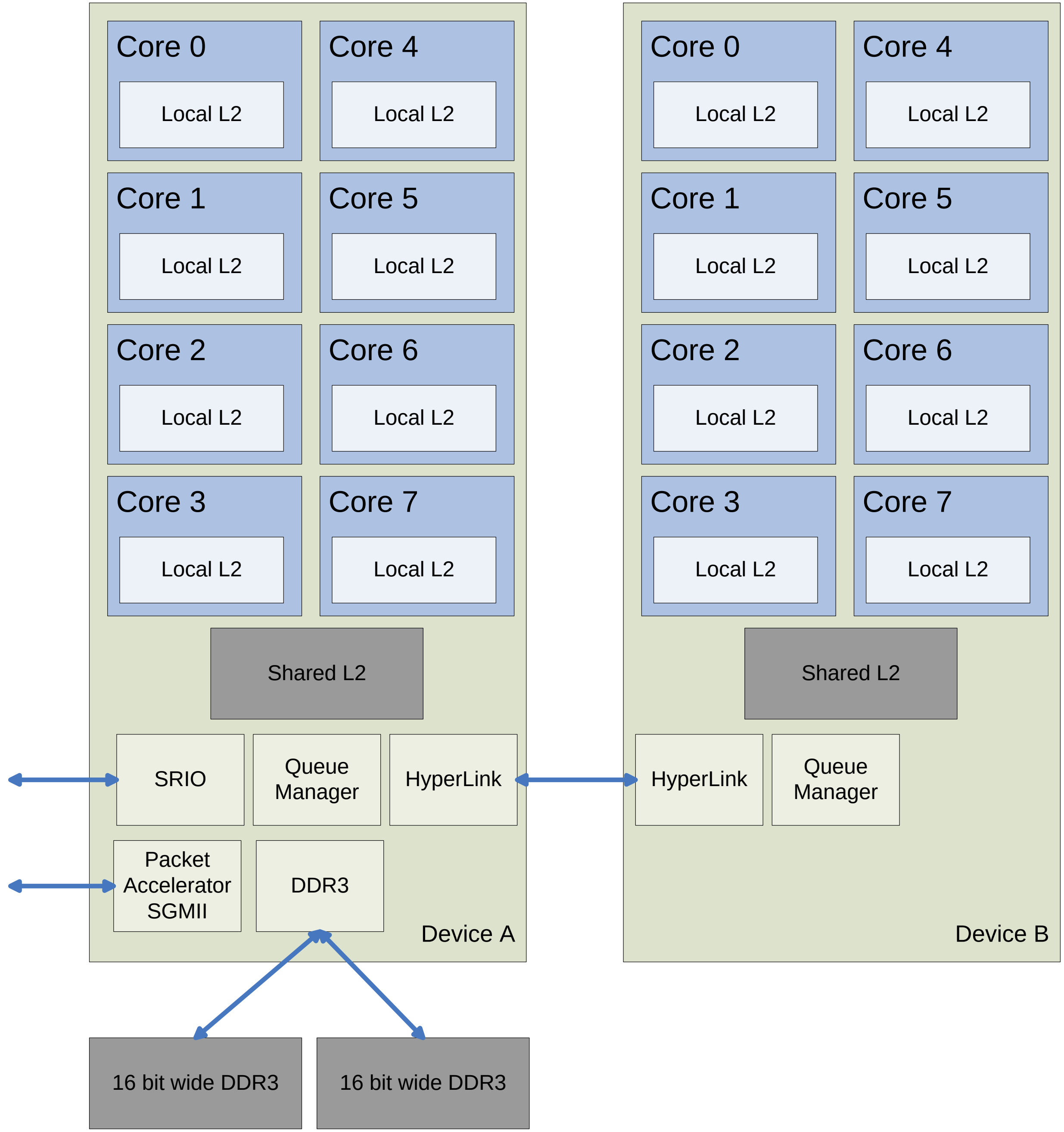
- Four read/write transactions
  - Write Request / Data Packet
  - Optional Write Response Packet
  - Read Request Packet
  - Read Response Data Packet
- Interrupt Request Packet passes event to remote side
- Multiple outstanding transactions
- 8 byte packet header (currently up to 64 bytes)
- 8b/9b error correction

HyperLink Functionality

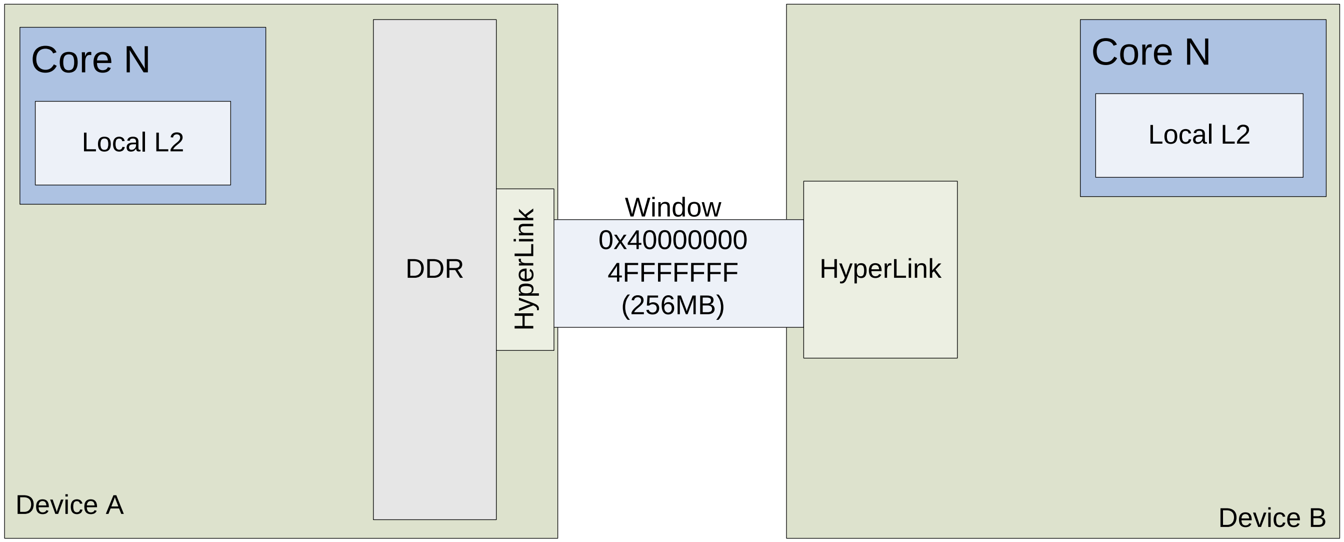
From the user point of view:

- Access remote device memory
  - Ability to write to remote device memory
  - Ability to read from remote device memory
- Ability to generate event / interrupt in the remote device

Example of HyperLink Use Case



HyperLink Model



Device B (local/Tx) can see up to 256MB of Device A (remote/Rx) memory:

- Available memory can be divided up to 64 distinct segments.
- All segments are the same size (local perspective).
- Segment size range is 256B to 256MB.
- All segments are aligned to 128KB boundaries.

HyperLink Interrupts Features

- 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

Agenda

1. HyperLink Overview
2. **Address Translation**
3. Configuration
4. Example and Demo

Segmentation

- The total visible window is 256MB.
- The application can define up to 64 segments.
- The segment size on the remote side is 256B to 256MB.
- All segments are aligned on 17 bits alignment.
- On the local side, the HyperLink memory is between 0x4000_0000 to 0x4FFF_FFFF.
- On the remote side, the HyperLink memory address range is device dependent, but is typically 0x0000_0000 to 0xFFFF_FFFF.

How is the local address (0x4XXX XXXX) translated to the remote address?

Offset Into a Segment

Largest Segment Size in Bytes (Power of 2)	Number of Bits For Offset	Maximum Number of Segments	Number of Bits Needed to Choose Segment
256M	28	1	0
128M	27	2	1
8M	23	32	5
4M	22	64	6
2M	21	64	6
16K	14	64	6

What Does Translation Involve?

Translation process inputs on the local/transmit side:

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

Process information sent from the local to the remote/receive side:

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

Translation process outputs on the remote/receive side:

1. Complete remote address
2. Privilege ID

Segment Lookup Table (Rx)

- The Segment Lookup Table is internal to the HyperLink and is not memory mapped.
- Each segment has a row:
  - 64 rows
  - information in each line
    - 16 bits are the MSB Segment Base Address
    - 5 bits are the Remote Segment Size (defines what mask is used when calculating the offset into the segment)
- The application loads the table row-by-row (segment-by-segment):
  - First, the Segment Base Address and segment size is written to register Rx Address Segment Value (base address + 0x3c) hyplnkRXSegIdxReg_s
  - Then the Segment Number is written to register Rx Address Segment Index (base + 0x38) hyplnkRXSegValReg_s

- During the translation process, the Segment Index is extracted from the upper bits of the local HyperLink address.

Low Level Driver Data Structures

Here are the data structures with brief descriptions:

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 Interupt 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

Example LLD: Write Multiple Registers

```
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

Building the Segment Lookup Table

"Build on the receive side for the transmit side specifications"

Here is one simple procedure for building the Segment Lookup Table:

1. Determine the maximum segment size that can be used (Power of 2), where N = The number of bits needed to address into the segment.
2. Calculate the number of bits needed for the segments (but no more than 6).
3. For each segment, load the base address and the remote segment size into the appropriate row of the table.
    - The base address is chosen so that N LSB are all zeros.
    - Only the upper 16 bits are written into the table.
4. If the number of segments is not Power of 2, add rows to complete to Power of 2 with empty segments (Size 0).

Segment Lookup Table: Example 1

Show the remote DDR addresses between 0x8000_0000 and 0x8FFF_FFFF (addressed in one consecutive 256MB segment):

- 28-bit offset
- 0 bits for choosing the segment (only one segment)
- One row in Segment Lookup Table

0x8000 Size 27 (size 0x01000_0000 = 256MB and the offset mask will be 0xfff_ffff)

Table 3-14 Rx Address Segment Value Register Field Descriptions

If rxlen_val = 16, the segment size is 0x00002_0000
If rxlen_val = 17, the segment size is 0x00004_0000
If rxlen_val = 18, the segment size is 0x00008_0000
If rxlen_val = 19, the segment size is 0x00010_0000
If rxlen_val = 20, the segment size is 0x00020_0000
If rxlen_val = 21, the segment size is 0x00040_0000
If rxlen_val = 22, the segment size is 0x00080_0000
If rxlen_val = 23, the segment size is 0x00100_0000
If rxlen_val = 24, the segment size is 0x00200_0000
If rxlen_val = 25, the segment size is 0x00400_0000
If rxlen_val = 26, the segment size is 0x00800_0000
If rxlen_val = 27, the segment size is 0x01000_0000

Segment Lookup Table: Example 2

- 8 segments
- Each segment of size 0x0100_0000 (16MB)
- Addresses start at 0x8000_0000, 0x8200_0000, 0x8400_0000, and continue up to 0x8E00_0000
- The maximum size is 16MB. That is, 24 bits.
- 3 bits to choose the segment (8 segments).

Row 0	0x8000_0000	Size 23
Row 1	0x8200_0000	Size 23
Row 2	0x8400_0000	Size 23
Row 3	0x8600_0000	Size 23
Row 4	0x8800_0000	Size 23
Row 5	0x8A00_0000	Size 23
Row 6	0x8C00_0000	Size 23
Row 7	0x8E00_0000	Size 23

Size 23 = 0x0010 0000 = 16MB and the mask that will be used is 0x000f ffff

Segment Lookup Table: Example 3

- 8 segments
- 7 each of size 0x0100_0000 (16MB)
- Addresses start at 0x8000_0000, 0x8100_0000, 0x8200_0000, and continue up to 0x8600_0000.
- The last segment is 32MB starting at address 0x8700_0000.
- The maximum size is 32MB. That is, 25 bits.
- 3 bits to choose the segment (8 segments)

Row 0	0x8000_0000	Size 23
Row 1	0x8100_0000	Size 23
Row 2	0x8200_0000	Size 23
Row 3	0x8300_0000	Size 23
Row 4	0x8400_0000	Size 23
Row 5	0x8500_0000	Size 23
Row 6	0x8600_0000	Size 23
Row 7	0x8700 _0000	Size 24

Segment Lookup Table: Example 4

- 9 segments
  - The first segment is the MSMC (4MB = 22 bits).
  - The next 8 segments are L2 memory of each core (512KB = 19 bits).
- The maximum size is 4MB. That is, 22 bits.
- 6 bits to choose the segment (64 segments)

Row 0	0x0C00_0000	Size 21 (4MB)
Row 1	0x1080_0000	Size 18 (512KB)
Row 2	0x1180_0000	Size 18
Row 3	0x1280_0000	Size 18
Row 4	0x1380_0000	Size 18
Row 5	0x1480_0000	Size 18
Row 6	0x1580_0000	Size 18
Row 7	0x1680_0000	Size 18
Row 8	0x1780_0000	Size 18
Row 9	0x0000_0000	Size 0 (0)
Row 10	0x0000_0000	Size 0

… and so on to Row 15.

On the Local/Transmit Side

- Information included in the Address Word:
  - Offset into Segment
  - Segment Index
  - Privilege ID Value

- Tx Address Overlay Control Register (base + 0x1c) controls the overlay of the Privilege ID.

- The Privilege ID lookup Table has 16 rows and is loaded using two registers:
  - Rx address PrivID Index –> base + 0x30 hyplnkRXPrivIDIdxReg_s
  - Rx Address PrivID Value -> base + 0x34 hyplnkRXPrivIDValReg_s

Local/Tx Side: Example

- The Tx Address Overlay Control Register (base + 0x1c) controls the overlay of the Privilege ID index. hyplnkTXAddrOvlyReg_s
- Agreed values for the HyperLink Privilege Index:
  - 13 (0xD) if the request comes from a core
  - 14 (0xE) if the request initiated from another master
  - Tx Address information bits 0 to 27, depends on the value of txigmask
  - Privilege Index bits 28-31
  - txigmask = Determine what address bits will be sent to the remote side:

    11 → mask 0x0FFF_FFFF ,10 → 0x07FF_FFFF

    8 → 0x01FF_FFFF, 0 → 0x0001_FFFF
  - Tx Address Overlay Control Register is shown below.

31 20	19 16	15 12	11 8	7 4	3 0
Reserved	txsecovl	Reserved	Txpriviovl	Reserved	txigmask
R	R/W	R	R/W	R	R/W

For other possible configurations, refer to the HyperLink User Guide.

On the Remote/Receive Side

Five registers control the behavior of the remote/receive side:

1. Rx Address Selector Control (base + 0x2c) controls how the address word is decoded; hyplnkRXAddrSelReg_s
2. Rx Address PrivID Index (base + 0x30) is used to build the Privilege Lookup Table; hyplnkRXPrivIDIdxReg_s
3. Rx Address PrivID Value (base + 0x34) is used to build the Privilege Lookup Table; hyplnkRXPrivIDValReg_s
4. Rx Address Segment Index (base + 0x38) is used to build the Segment Lookup Table; hyplnkRXSegIdxReg_s
5. Rx Address Segment Value (base + 0x3c) is used to build the Segment Lookup Table; hyplnkRXSegValReg_s

Remote/Rx Side: Example

- Rx Address Selector Control (base + 0x2c; hyplnkRXAddrSelReg_s) controls how the receiver decodes:
  - Location in the address word and value of security bit (not used)
  - Location in the address word of Privilege Index.
  - Location in the address word of the index into segment lookup table
  - The mask that is used for extracting the offset
    - rxprividsel = 12 (bits 28-31)
    - rxsegsel depends on the maximum segment size:

    12 → mask 0x0FFF_FFFF, 10 → 0x03FF_FFFF,

    8 → 0x00FF_FFFF ,   1 → 0x0001_FFFF
- The Rx Address Selector Control register is shown below.

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

For other possible configurations, refer to the HyperLink User Guide.

Remote (Rx) Address Examples

Building upon each of the prior examples, this section demonstrates how to calculate the address value that is sent to the remote/receive side.

- The local address is 0x4567_89a0
- Assume Privilege ID 0xD (request from a core) was loaded to Index 5 (0101) in the PrivID table.
- The address sent to the other side is 0x5567_89a0.

What address is accessed in the remote side?

Segment Lookup Table: Example 1

Show the remote DDR addresses between 0x8000_0000 and 0x8FFF_FFFF (addressed in one consecutive 256MB segment):

- 28-bit offset
- 0 bits for choosing the segment (only one segment)
- One row in Segment Lookup Table

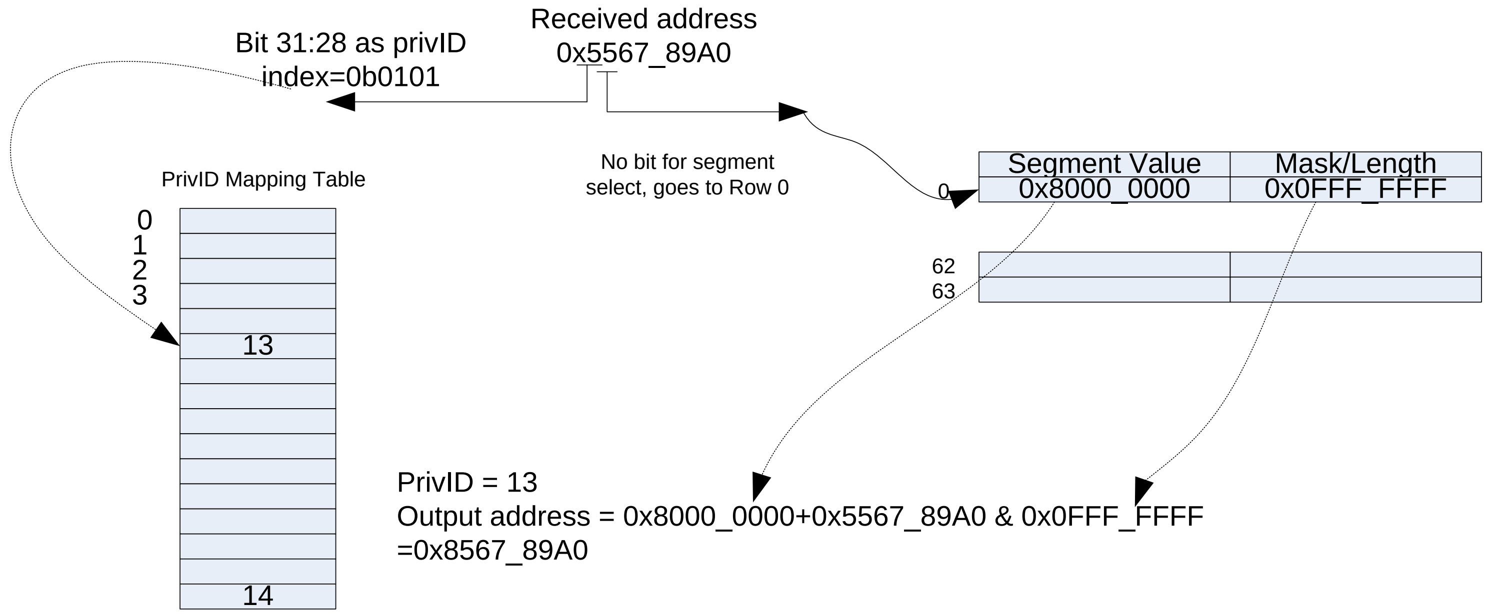
0x8000   Size 27 (size 0x1000_0000 = 256MB)

Remote (Rx) Address: Example 1

Show the remote DDR addresses between 0x8000_0000 and 0x8FFF_FFFF (addressed in one consecutive 256MB Segment):

- 28 bit offset - 0x0567 89a0
- Bits 28-31   0x0101 = 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
(Segment size = 28 -> offset mask = 0x0fff ffff



Segment Lookup Table: Example 2

- 8 segments
- Each segment of size 0x0100_0000 (16MB)
- Addresses start at 0x8000_0000, 0x8200_0000, 0x8400_0000, and continue up to 0x8E00_0000
- The maximum size is 16MB. That is, 24 bits.
- 3 bits to choose the segment (8 segments).

Row 0	0x8000_0000	Size 23
Row 1	0x8200_0000	Size 23
Row 2	0x8400_0000	Size 23
Row 3	0x8600_0000	Size 23
Row 4	0x8800_0000	Size 23
Row 5	0x8A00_0000	Size 23
Row 6	0x8C00_0000	Size 23
Row 7	0x8E00_0000	Size 23

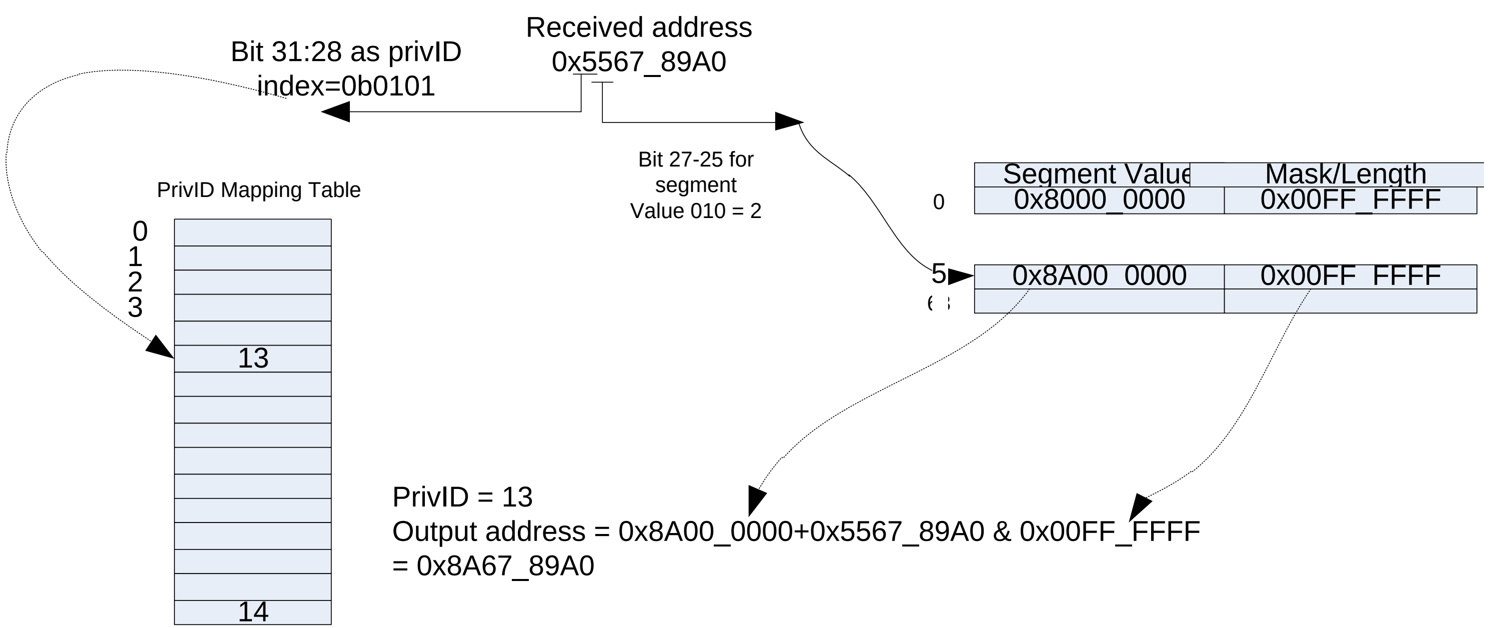
Size 23 = 0x0010 0000 = 16MB

Remote (Rx) Address: Example 2

- 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



Segment Lookup Table: Example 3

- 8 segments
- 7 each of size 0x0100_0000 (16MB)
- Addresses start at 0x8000_0000, 0x8100_0000, 0x8200_0000, and continue up to 0x8600_0000.
- The last segment is 32MB starting at address 0x8700_0000.
- The maximum size is 32MB. That is, 25 bits.
- 3 bits to choose the segment (8 segments)

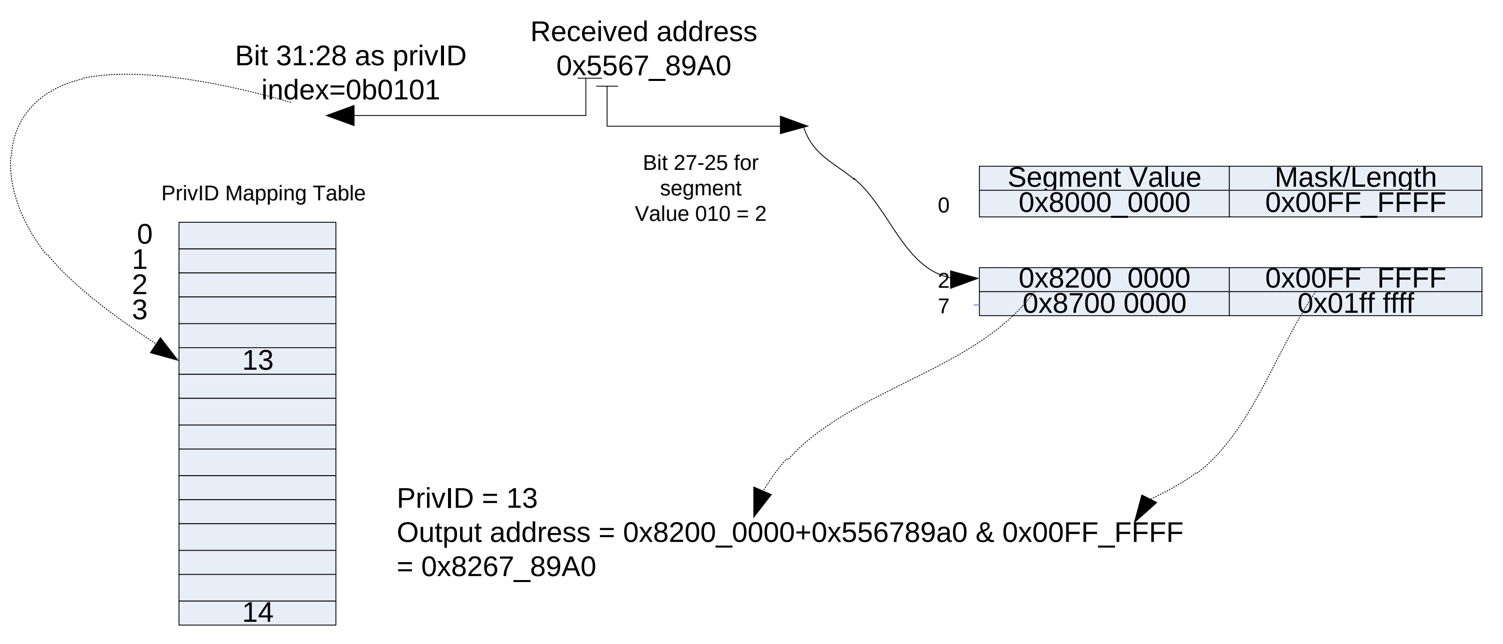
Row 0	0x8000_0000	Size 23
Row 1	0x8100_0000	Size 23
Row 2	0x8200_0000	Size 23
Row 3	0x8300_0000	Size 23
Row 4	0x8400_0000	Size 23
Row 5	0x8500_0000	Size 23
Row 6	0x8600_0000	Size 23
Row 7	0x8700 _0000	Size 24

Remote (Rx) Address: Example 3

- 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



Segment Lookup Table: Example 4

- 9 segments
  - The first segment is the MSMC (4MB = 22 bits).
  - The next 8 segments are L2 memory of each core (512KB = 19 bits).
- The maximum size is 4MB. That is, 22 bits.
- 6 bits to choose the segment (64 segments)

Row 0	0x0C00_0000	Size 21 (4MB)
Row 1	0x1080_0000	Size 18 (512KB)
Row 2	0x1180_0000	Size 18
Row 3	0x1280_0000	Size 18
Row 4	0x1380_0000	Size 18
Row 5	0x1480_0000	Size 18
Row 6	0x1580_0000	Size 18
Row 7	0x1680_0000	Size 18
Row 8	0x1780_0000	Size 18
Row 9	0x0000_0000	Size 0 (0)
Row 10	0x0000_0000	Size 0

… and so on to Row 15.

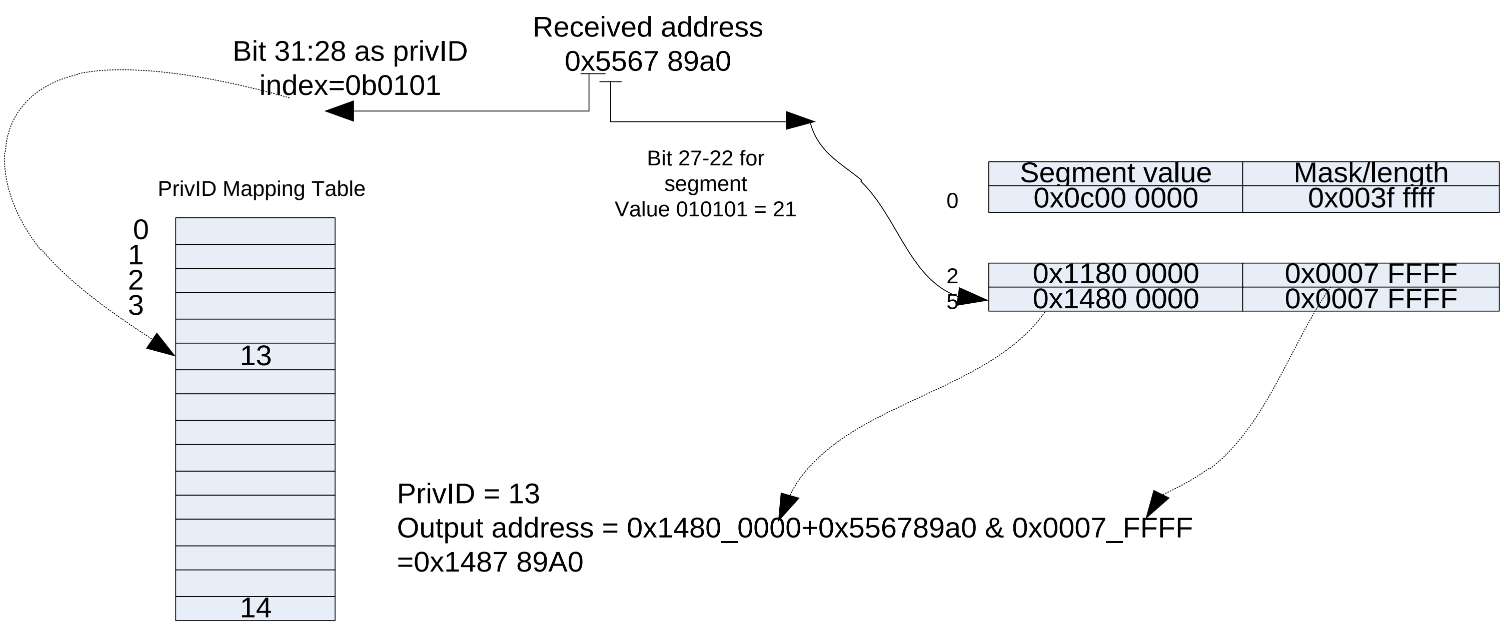
Remote (Rx) Address: Example 4

- 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, the address is 0x1480 0000 + 0x0007 89a0 = 0x1487 89a0

(L2 memory of Core 4)



Agenda

1. HyperLink Overview
2. Address Translation
3. **Configuration**
4. Examples and Demo

Chip Level Configuration

1. Enable power domain for peripherals using CSL routines.

   Enabling power to peripherals involves the following four functions:

   CSL_PSC_enablePowerDomain()

   CSL_PSC_setMosuleNextState()

   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()

Platform Level Configuration

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

Basic HyperLink LLD Functions

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 Functions

- Configuration functions are part of the HyperLink example in the PDK release 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)

Example: Configuration Function

```
/*****************************************************************
******************
 * 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 */
```

Agenda

1. HyperLink Overview
2. Address Translation
3. Configuration
4. **Example and Demo**

Example and Demo

- Included in the PDK (Platform Development Kit) release are a set of examples for each of the peripherals.
- For HyperLink, there is one example that can be configured either as a single-EVM loopback or between two C66x EVM boards.
- Location of the example:

`pdk_C6678_1_0_0_19\packages\ti\drv\exampleProjects\hyplnk_exampleProject`

- The loopback flag is in the file **`hyplnkLLDCfg.h`**

For More Information

- For more information, refer to the KeyStone Architecture HyperLink User Guide http://www.ti.com/lit/SPRUGW8
- For questions regarding topics covered in this training, visit the support forums at the TI E2E Community website.