The TLM 2.0 Mixed Endianness Example

James Aldis

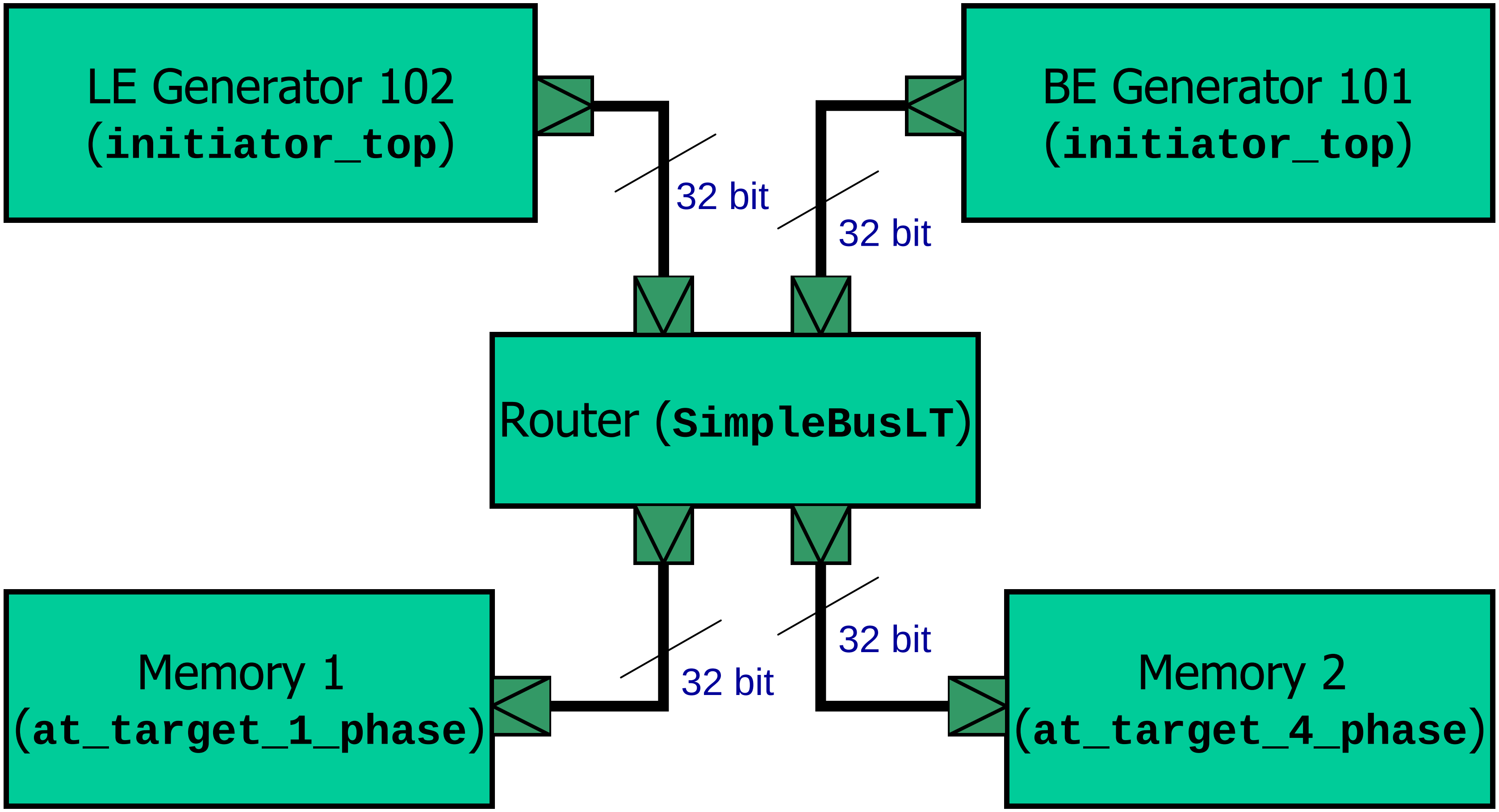
Texas Instruments France

January 2008

Modification of “lt” Example

- Demonstrates the use of TLM endianness-conversion functions
- Illustrates data and address modification effects of BIG- and LITTLE-endian initiators sharing a memory
- Allows interactive experimentation
- Interactively execute “instructions” alternately on two initiators
  - `one big- and one little-endian`
  - `store data from BE and view it from LE or vice-versa`
  - `store 32-bit data and view as 8-bit or 16-bit, etc`

Platform Structure



LE Generator 102
(**initiator_top**)

BE Generator 101
(**initiator_top**)

Router (**SimpleBusLT**)

Memory 1
(**at_target_1_phase**)

Memory 2
(**at_target_4_phase**)

tlm_initiator_socket **tlm_target_socket**

How to run this example (Linux)

- Set `SYSTEMC_HOME`
- `cd examples/tlm/lt_mixed_endian/build-unix`
- `make`
- `make run (uses default input)`
- `./lt.exe (interactive input)`

How to run this example (MSVC)

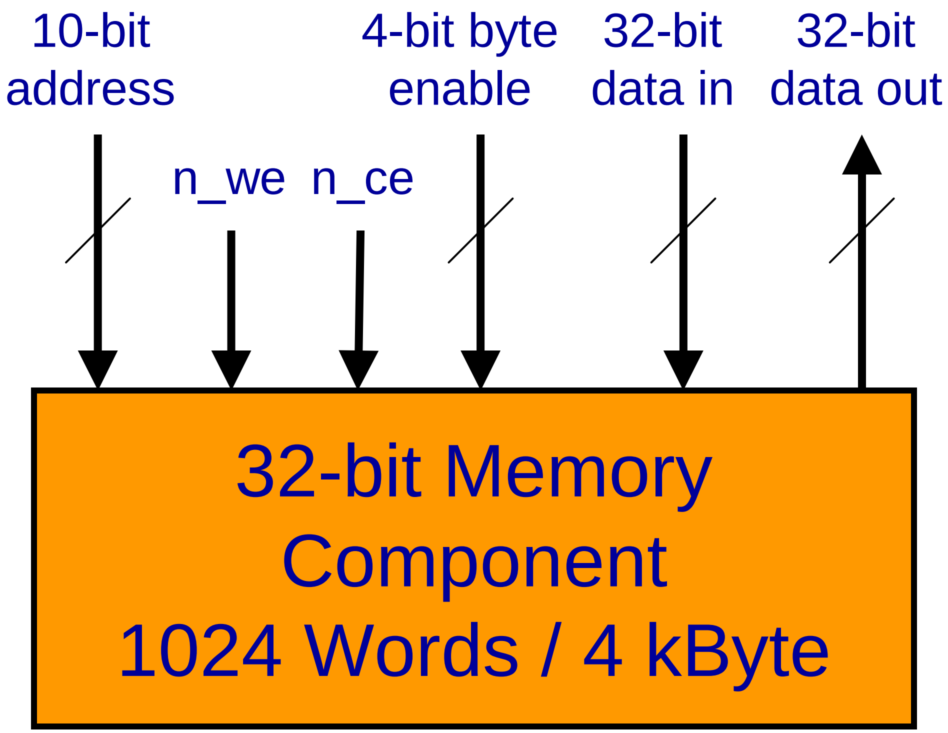
- Open a explorer window on **`examples/tlm/lt_mixed_endian/build-windows`**
- Launch **`lt.sln`**
- Select **‘Property Manager’** from the **‘View’** menu
- Under **‘lt_extension_mandatory > Debug | Win32’** select **‘systemc’**
- Select **‘Properties’** from the **‘View’** menu
- Select **‘User Macros’** under **‘Common Properties’**
- Update the **‘`SYSTEMC`’** entry and apply
- Select **‘Debugging’** under **‘Configuration Properties’**
- Click in the **‘`Command Arguments`’** box and type in

Component details

- Generator 101
  - 32-bit Big-endian initiator
  - stdin provides instructions to execute (loads and stores)
  - stdin provides data for stores
  - data from loads written to stdout
- Generator 102
  - Identical but Little-endian
- Instruction Set for both Generators
  - l8, l16, l32 (load byte, halfword or word)
  - s8, s16, s32 (store byte, halfword or word)
  - w (switch control to other generator)
  - q (quit)
  - see `examples/tlm/lt_mixed_endian/results/input.txt` for example instructions
- Bus and Memory
  - Exactly as in lt example
  - Some kind of 32-bit routing and memory system

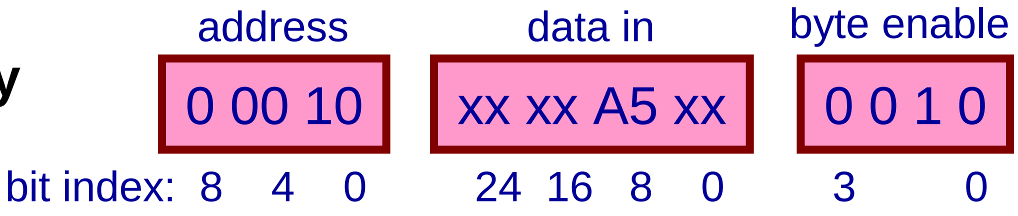
The System Being Modelled

- Both memories are simple arrays of 4-byte words
- Neither the memories nor the interconnect is aware of the endianness of the transactions
- Neither the memories nor the interconnect does any data modification:
  - what the initiators put on the bus goes unchanged to the memory
- Memories are endianness-neutral
  - provide a consistent memory image to BE and to LE initiators
  - “What I write, I read back the same”
  - there are no attributes on the hardware bus except those shown opposite



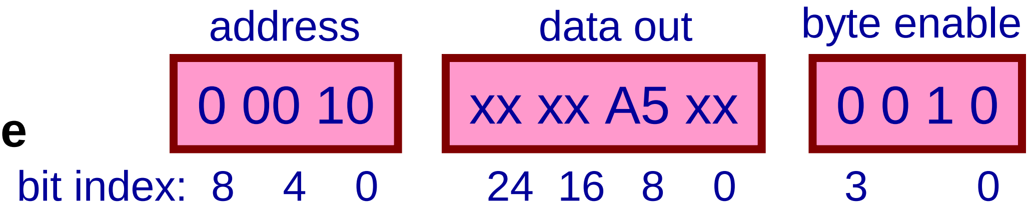
The System Being Modelled

- **"What the other writes, I may read back at a different address"**
- **Example 1: single byte access**
  - **BE initiator writes byte A5 to address 66 = 64 + 2**
    - **see opposite for hardware bus signals**
  - **LE initiator will find it at address 65 = 64 + 1**
    - **see opposite for hardware bus signals**



	address	data in	byte enable
	0 00 10	xx xx A5 xx	0 0 1 0
bit index:	8 4 0	24 16 8 0	3 0

Address bus takes the index of the correct 32-bit word (66/4 = 16). Fractional part of the address (2) used inside initiator to select a byte lane according to **big-endian** convention (more significant bits are at lower addresses)



	address	data out	byte enable
	0 00 10	xx xx A5 xx	0 0 1 0
bit index:	8 4 0	24 16 8 0	3 0

Little-endian convention is that more significant bits are at *higher* addresses. To get the same data, fractional part of the address is 1

The System Being Modelled

- **“What the other writes, I may read back at a different address”**
- **Same effect for aligned 16-bit or 32-bit data**
  - **But no address change for 32-bit**
- **Example 2: 32-bit integer access**
  - **BE initiator writes 32-bit integer 04030201 to address 256**
    - **see opposite for hardware bus signals**
  - **LE initiator will find the same data at the same address**
    - **see opposite for hardware bus signals**

	address	data in	byte enable
	0 00 40	04 03 02 01	1 1 1 1
bit index:	8 4 0	24 16 8 0	3 0

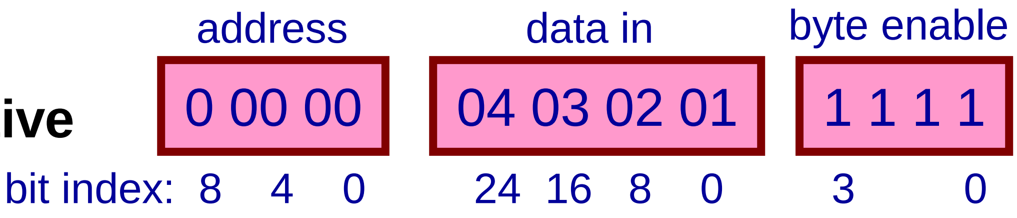
Address bus takes the index of the correct 32-bit word (256/4 = 64). **Big-endian** convention is that more significant bits of the integer are at lower addresses, which are at more significant bits of the bus data word.

	address	data out	byte enable
	0 00 40	04 03 02 01	1 1 1 1
bit index:	8 4 0	24 16 8 0	3 0

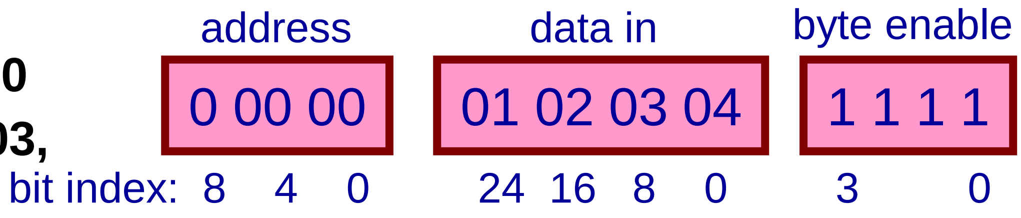
Little-endian convention is that more significant bits of the integer are at *higher* addresses, which are at more significant bits of the bus data word. Therefore it is the same as big-endian!

The System Being Modelled

- **“What the other writes, I may read back distorted”**
- **Example 3: write 4 consecutive bytes and read back as an integer**
  - **BE initiator writes bytes 04, 03, 02, 01 to addresses 0, 1, 2, 3**
    - **see opposite for hardware bus signals**
  - **Both initiators will read the integer 04030201 at address 0**
  - **LE initiator writes bytes 04, 03, 02, 01 to addresses 0, 1, 2, 3**
    - **see opposite for hardware bus signals**
  - **Both initiators will read the integer 01020304 at address 0**



Big-endian convention is that more significant bits of the data bus word are lower addresses.



Little-endian convention is that more significant bits of the data bus word are *higher* addresses.

The TLM Model of the System

- The TLM model correctly models the above data and address distortions and all other possible ones
  - In particular it gets hairy for non-address-aligned transactions
- The address, data array and byte enable array in the transaction payload object are
  - identical to the internal opcode of the initiator
    - *if the initiator endianness matches the host CPU endianness*
  - modified
    - *if the initiator endianness is different from the host CPU's*
- The internal data storage of the memory models
  - is not visible (we are not using DMI in this example)
  - But could be a simple memcpy() between the data array in the transaction payload object and an array of unsigned char in the memory model
- Therefore we can say that
  - *the TLM 2.0 interfaces are always "host-endian"*
- Model is functionally identical on BE and LE host CPUs
  - *but internal structure (length, address, byte enables) of transaction payload objects will differ*