

Objective: Build and simulate a simple design following the FlexNoC design flow

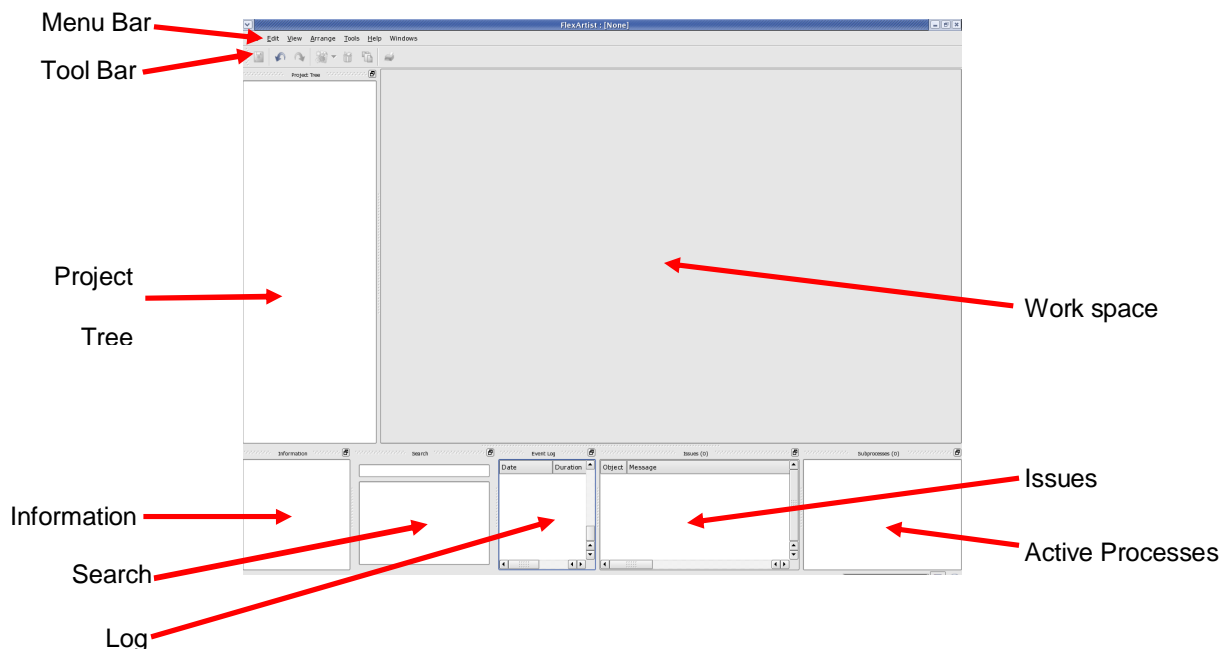
The purpose of this lab is to build a simple design and take it through a successful simulation. The example design has five initiators, four are AXI and one is AHB; and two targets, an AHB and an AXI. The interfaces are of varying widths and clock frequencies. Although a typical NoC will be more complex than this example, choosing a small design allows us to focus on the design flow. We will use FlexNoC to create everything from the start. The lab is divided into 3 parts, mirroring the 3 stages in the FlexNoC design flow:

1. Specification: Define the interface protocols for the initiator and target IPs. Add system information such as clocks, connectivity and address map.
2. Architecture: Define an Architecture and add performance parameters based on design requirements. Modify paths based on latency and bandwidth requirements.
3. Structure: Generate a NoC Structure. Use the Automatic Test Environment (ATE) to generate a test bench and test cases. Run a simple simulation scenario from the command line.

Procedure:

Open the FlexNoC GUI by entering `flexNoC &` at the command prompt (use the `&` to keep your command prompt available for later).

1. Identify the major functional areas in the GUI as shown in the screenshot below:



Many operations can be performed from the menu bar, tool bar, context sensitive pop-up menu, and/or keyboard shortcuts. This lab mostly uses right mouse clicks for the context sensitive pop up menu in the working area. On the bottom of the GUI are status and information windows which can be made visible or not and optionally stacked (with selection tabs on the bottom). It is important to monitor the Issues window for errors or missing parameters. The Information window provides more details about whatever object is selected, including issue messages. For even more detail, use the documentation that is available in the Help menu. The remaining windows will not be used in this lab.

Part 1: Specification

1. Create a new specification:
Right click in the Project Tree area and select **New → Specification** to generate a Specification. Use the default name of Specification and click **OK**.
2. Double click on the Specification object in the Project Tree.

We now see tabs along the left edge of the editor and along the top edge as well. The tabs along the left edge select the major areas of the specification so we will refer to them as major tabs. The tabs along the top edge select from within the major tabs and will be called minor tabs.

FlexNoC has generated a simple starting point. However, there are issue messages indicating that there are some undefined parameters. Don't worry, all the issues will go away soon.

FlexNoC editors include many parameters. They can be set by left-clicking the parameter cell.

First, setup the clocks.

In the Domain major tab → Clock minor tab set the clock frequency by clicking on Missing and changing the frequency to 200 (the default unit is MHz).

Rename the clock regime to rclk200 by double clicking the name.

Duplicate the clock regime and make it a 166 MHz clock named rclk166.

- a. Select the clock regime, right click, and choose Duplicate.
- b. Rename the new regime.
- c. Change the frequency to 166 MHz.

Note: It is possible to select more than one parameter and change them all at once. If you change a parameter (such as clock frequency) when the whole line is selected then all parameters on the line (including voltage) are set to the new value. This is a common mistake. You can undo a mistaken change by clicking this button



or typing ctrl-z. To avoid this mistake, click away from the line with the parameter to change, then left-click just the parameter to change.

The clock tab should look like this.

		Clock					
		Power					
		Voltage					
Column							
Row							
Modifications							
Defer							
		voltage	frequency	power	type	synthesisInfo	comment
Domain	rclk166	None	166 MHz				
	Cm			None			
	root				Root		
Interface	rclk200	None	200 MHz				
	Cm			None			
	root				Root		
Internal							
Memory Map							

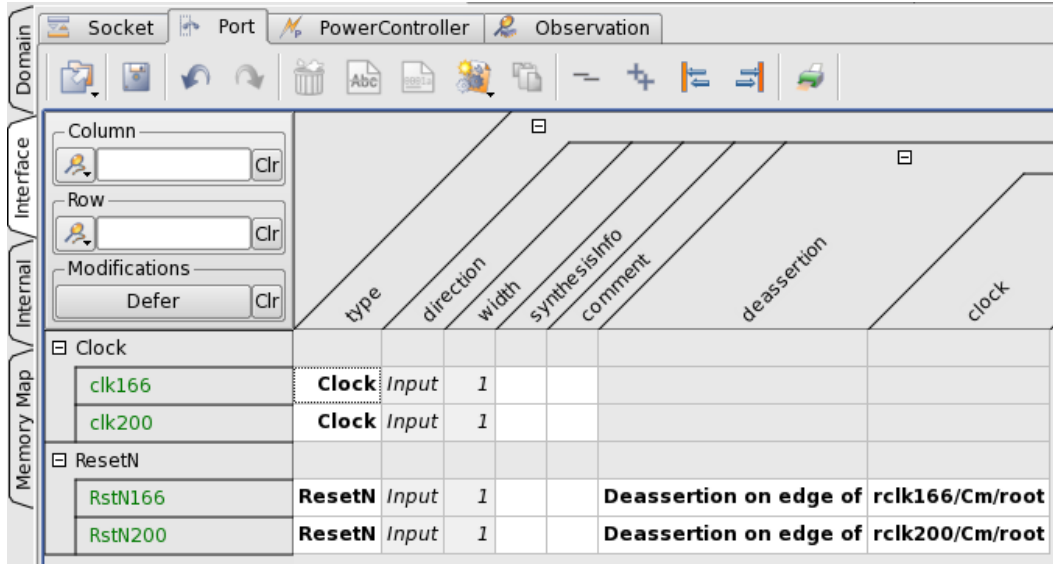
Go to the Interface Major tab → Port minor tab to define the clock port inputs to the NoC. Note that one clock and reset are already defined. Change its name from Clk to clk200

Duplicate this clock and rename the new copy clk166.

Rename the reset from RstN to RstN200, duplicate it, and rename the new copy RstN166.

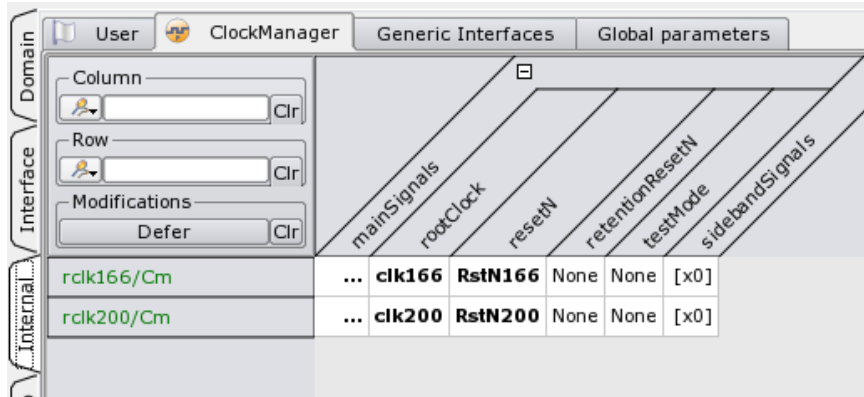
The clock sub parameter indicates which clock domain with which the reset is synchronous. Select rclk200 for RstN200 and rclk166 for RstN166.

The Ports tab should look like this.



Column	Row	Modifications	Defer	type	direction	width	synthesisInfo	comment	deassertion	clock
Clock										
clk166				Clock	Input	1				
clk200				Clock	Input	1				
ResetN										
RstN166				ResetN	Input	1		Deassertion on edge of	rclk166/Cm/root	
RstN200				ResetN	Input	1		Deassertion on edge of	rclk200/Cm/root	

Finally, in the Internal major tab → ClockManager minor tab, select clk200 and RstN200 for the main signals of the clock manager named rclk200/Cm. Select clk166 and RstN166 for the main signals of the clock manager named rclk166/Cm.



Column	Row	Modifications	Defer	mainSignals	rootClock	resetN	retentionResetN	testMode	sidebandSignals
rclk166/Cm				... clk166	RstN166	None	None	[x0]	
rclk200/Cm				... clk200	RstN200	None	None	[x0]	

3. Now that the clocks are set, let's configure the initiator and target socket interfaces.

Click on the Interface major tab → Socket minor tab. This is where the IP-specific interfaces are configured for each socket's NIU.

- Select the line for initiator0, right-click, and choose Replicate to create four more initiator sockets.
- Use duplicate to create one more target socket.
- Rename the initiator sockets: CPU, DMA, DSP, ENET, and USB.
- Rename the target sockets: DRAM and ROM.
- Click the protocol parameter text to select the whole column then left-click any of the selected cells to change all sockets to have the AXI protocol.

- vi. Click away, then click the protocol parameter for ROM. Ctrl-click the protocol parameter for ENET. Left-click the ENET protocol parameter to Change both ENET and ROM to AHB.
- vii. Click the [+] symbol above the protocol parameter to expand an array of sub-parameters. Use multi-select to change all wData parameters to a 64 bit data bus and useBigEndian parameters to False.
- viii. Set DMA to have a 128 bit data bus and set ENET and ROM to have a 32 bit data bus.
- ix. Click [+] to expect the conversion sub parameters and set busyIgnoreWaits and combHReady for AHB initiators to False.
- x. Now configure the specification for ENET, USB, and ROM to be in the 166 MHz clock domain by changing the clock parameter for those sockets to the rc1k166/Cm/root domain.

[illegible]

4. Finally, set up the connectivity and address map.

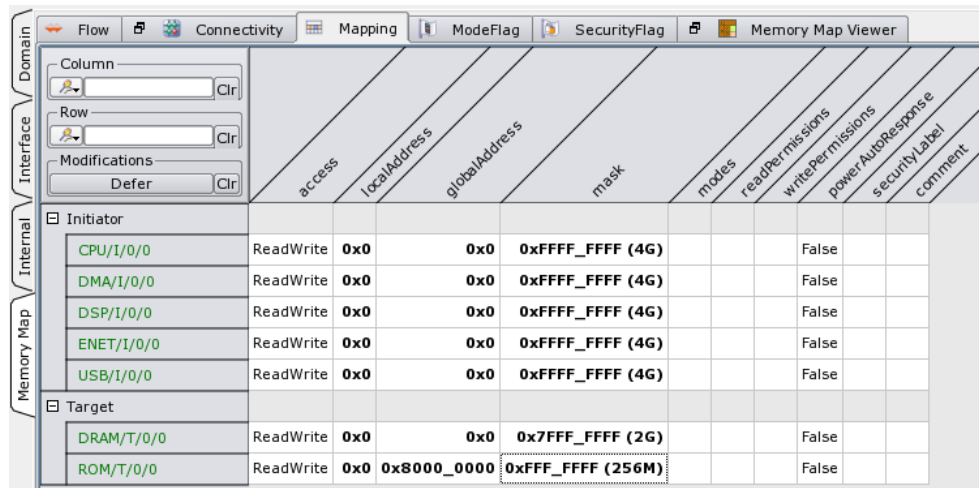
To automate connecting the optional signals of the AHB and AXI protocols, select the Specification in the Project Tree window, right-click, and choose Enable use of additional defaults.

- i. Select the Memory Map → Connectivity tab. Remove the connectivity from DMA to ROM and USB to ROM.

The screenshot shows the TI-RTOS Configuration tool with the Memory Map tab selected. The tool is organized into four vertical sections: Domain, Interface, Internal, and Memory Map. The Memory Map section displays a table of components and their connections to DRAM and ROM.

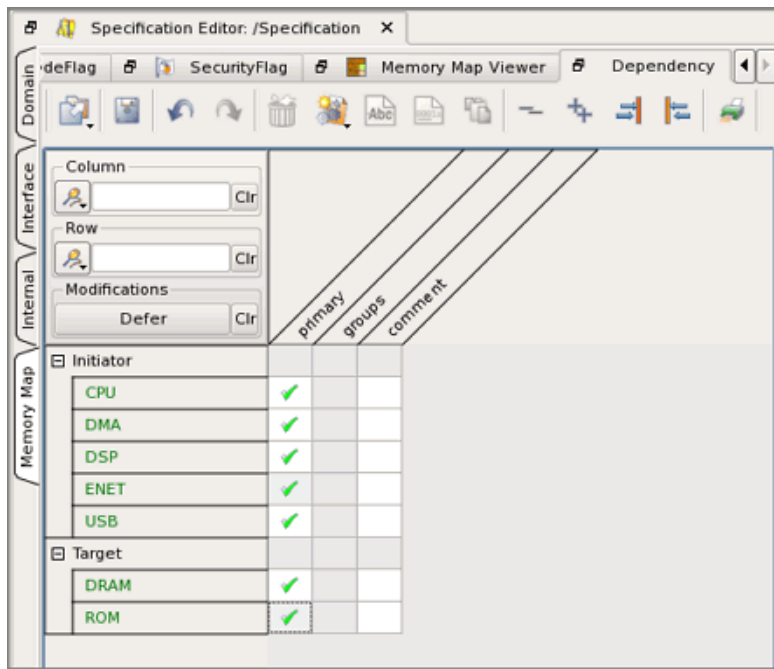
Component	DRAM	ROM
CPU/I/O	✓	✓
DMA/I/O	✓	
DSP/I/O	✓	✓
ENET/I/O	✓	✓
USB/I/O	✓	

- ii. View the Memory Map → Mapping tab. Targets are memory mapped. They are placed within an address space that is accessed by each initiator. For targets, the globalAddress is the base address for software access. The mask is the size, in bytes, of the address range allocated to the socket. For initiators, the globalAddress is the base address that the initiator can access. In most systems all initiator base addresses are set to 0x0 and the mask is set to the full 32-bit address range (mask of 0xFFFF_FFFF). Set the initiator parameters that way.
- iii. Set the target mappings for the DRAM at global address 0x0 with a 2 GB range and ROM at global address 0x8000_0000 with a 256 MB range.



Domain	Interface	Internal	access	localAddress	globalAddress	mask	modes	readPermissions	writePermissions	powerAutoResponse	securityLabel	comment
Memory Map	Initiator	CPU/I/0/0	ReadWrite	0x0	0x0	0xFFFF_FFFF (4G)				False		
		DMA/I/0/0	ReadWrite	0x0	0x0	0xFFFF_FFFF (4G)				False		
		DSP/I/0/0	ReadWrite	0x0	0x0	0xFFFF_FFFF (4G)				False		
		ENET/I/0/0	ReadWrite	0x0	0x0	0xFFFF_FFFF (4G)				False		
		USB/I/0/0	ReadWrite	0x0	0x0	0xFFFF_FFFF (4G)				False		
Memory Map	Target	DRAM/T/0/0	ReadWrite	0x0	0x0	0x7FFF_FFFF (2G)				False		
		ROM/T/0/0	ReadWrite	0x0	0x8000_0000	0xFFFF_FFFF (256M)				False		

In the Memory Map tab → Dependency minor tab, click all red boxes as the follows:



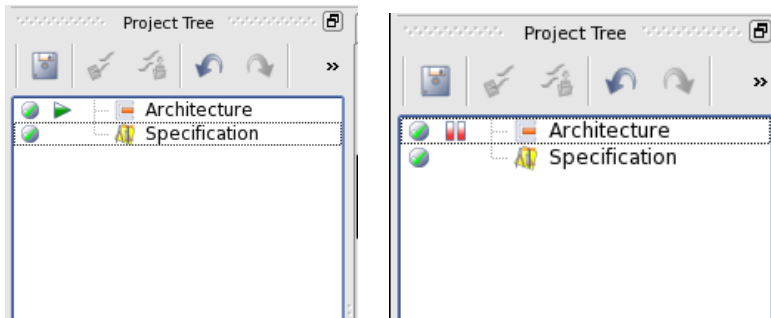
Domain	Interface	Internal	primary	groups	comment
Memory Map	Initiator	CPU	✓		
		DMA	✓		
		DSP	✓		
		ENET	✓		
		USB	✓		
Memory Map	Target	DRAM	✓		
		ROM	✓		

The system specification is complete, and no issues should be reported. It is now time to generate an Architecture for the design.

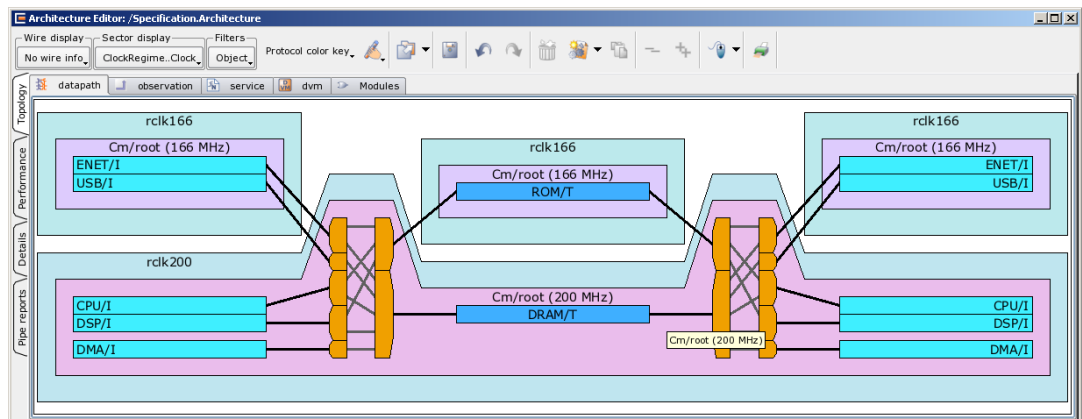
Part 2: Architecture

Select the Specification in the Project Tree, then right click and select New → Architecture. Leave the default name of Architecture.

Note the green triangle to the left of the Architecture in the Project Tree. This indicates that further changes to the Specification will propagate to the Architecture automatically. To disable automatic updating, select the Architecture object, right click, and select Disable Automatic Update. This will prevent specification updates from propagating to the architecture, except when the architecture is selected or open in an editor window.



1. Double click on the new Architecture object to open the Architecture Editor. The first view is the Topology → datapath tab.



This is a block diagram view of the NoC request and response paths. Requests go from the initiator on the left to the target in the middle and responses go from the target to the initiator on the right.

The colored sectors represent clock domains within clock regimes. The sockets are in the domains that we chose in the Specification editor.

With the cursor arrow over the block diagram view of the architecture turn the scroll wheel of the mouse. This zooms in and out of the topology view. Move the mouse while holding the middle button. This moves the architecture diagram.

2. Like the Specification Editor, the Architecture Editor has major and minor tabs. We will first work in the Performance major tab.

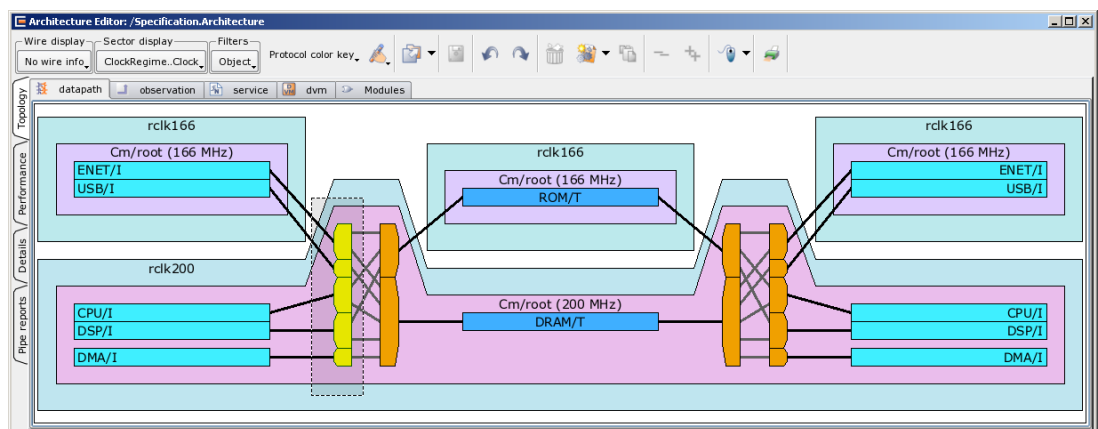
In the Performance → Generic NIU major tab configure the number of pending transactions for each initiator NIU. Set nPendingTrans to 2 for the CPU, DMA, and DSP initiator NIUs and set nPendingTrans to 4 for the DRAM target.

		Specific NIU	Generic NIU	Scheduler	Serialization	Buffering	Arbiter	Splitting	Shaping
Topology	Column								
	Row								
Performance	Modifications								
	Defer								
Details			seqAlloc	nndWrAligner	useRspUrgency	occupiedConteXOutput	useSingleWordRsp	roundSize	roundMaxLen
			nPendingOrderId	nPendingTrans	nDataBuffer	qosGenerator	rearDerBuffer	comment	
Pipe reports	Initiator Generic NIU								
	CPU/I					8	Not Applicable	1	2
	DMA/I					16	Not Applicable	1	2
	DSP/I					8	Not Applicable	1	2
	ENET/I					4	Not Applicable	1	1
	USB/I					8	Not Applicable	1	1
Pipe reports	Target Generic NIU								
	DRAM/T		HASHSUM	1	False	False	False		4
	ROM/T		NONE	1	False	False	False		1

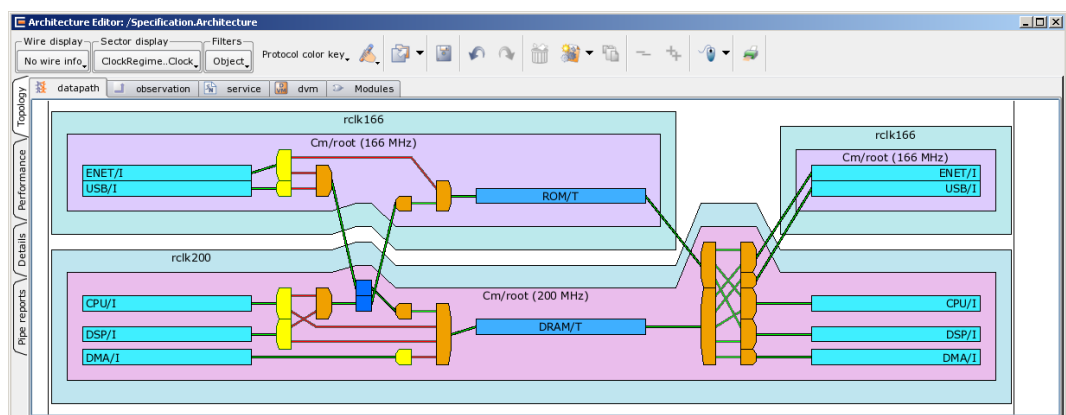
- Return to the Topology → datapath tab. Notice that packets sent between ENET and ROM cross into the rclk200 clock domain and then back to the rclk166 clock domain. This adds unnecessary latency and power consumption. Also notice that ENET and USB each have a separate clock domain crossing segment. This adds unnecessary die area.

As a first optimization we will isolate the pseudo switches (groups of demuxes and muxes connected together within the same domain). This will split them into smaller pseudo switches placed within each domain and connected with new links (blue boxes). This reduces the number of domain crossing conversions.

Select the leftmost group of demuxes (request pseudo switch) by holding SHIFT and dragging the cursor to enclose them in a selecting rectangle with left-mouse click.

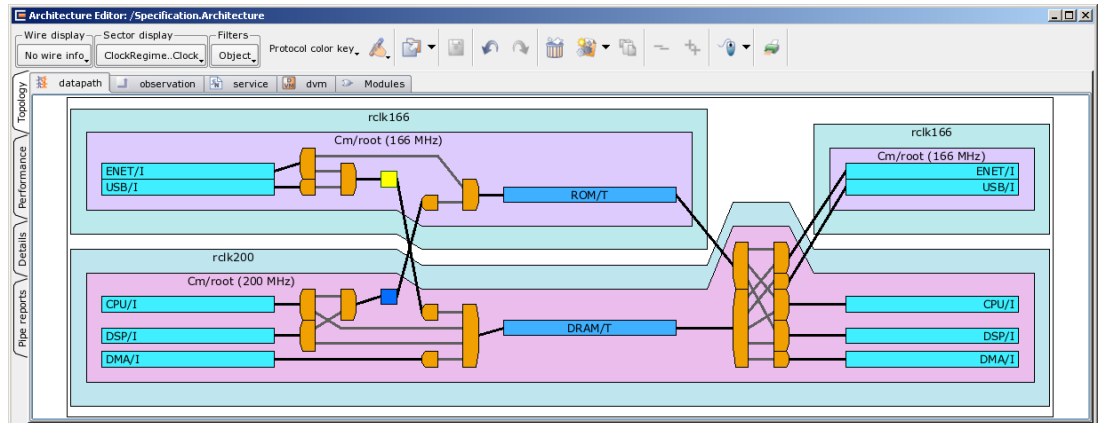



Right click and choose Isolate.

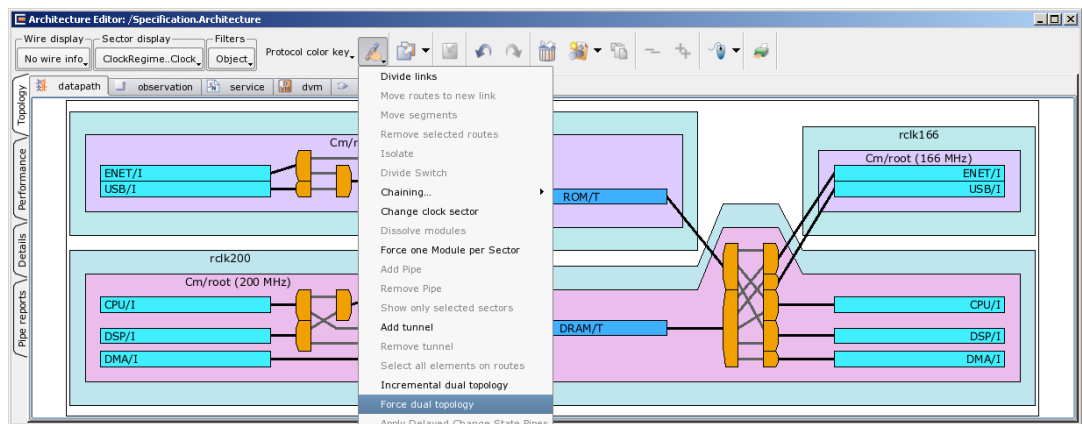


FlexNoC Lab-1

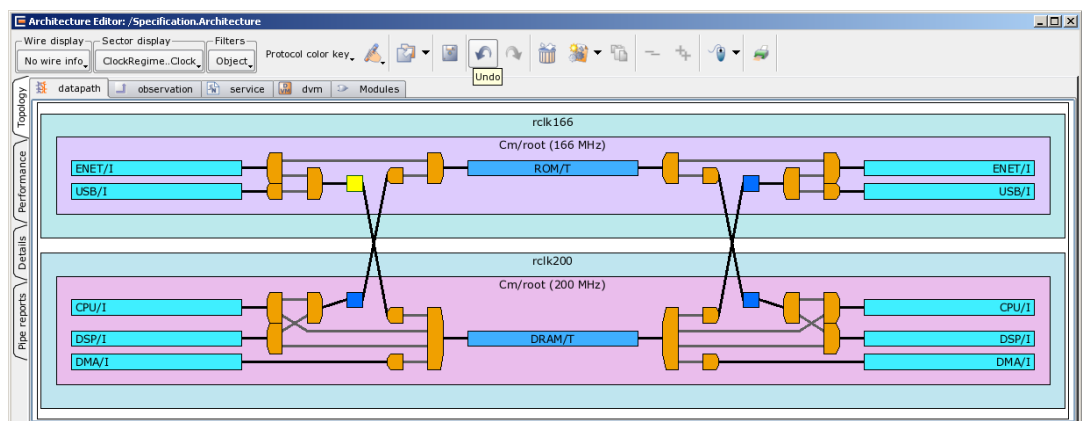
4. Drag the upper link (blue box) to the rclk166/Cm/root clock domain then right-click and choose Optimize View.



5. To make the identical changes to the response path, select the commands menu  → Force dual topology



To see a result like this.



The Architecture for this lab is complete. There should be no issue messages.

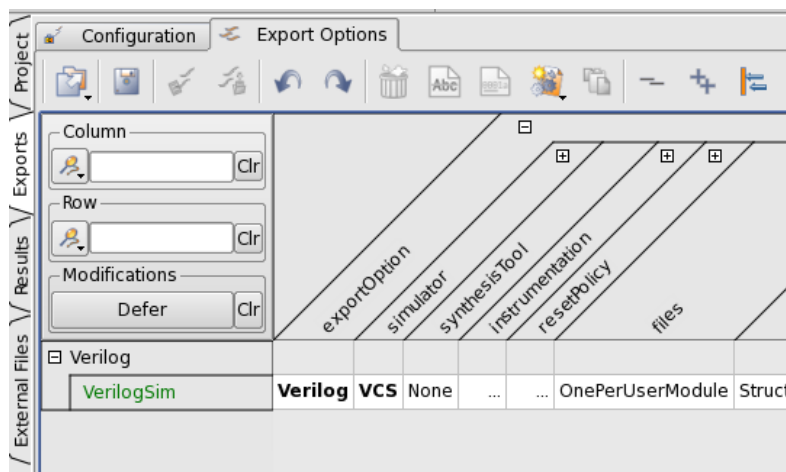
The structure provides a graphical representation of the hardware IP units generated from the architecture. While the architecture topology view looks like a block diagram, the Structure looks like a schematic diagram. Each block may have several hardware IP units. These can be identified by clicking the block and looking in the Information window.

-

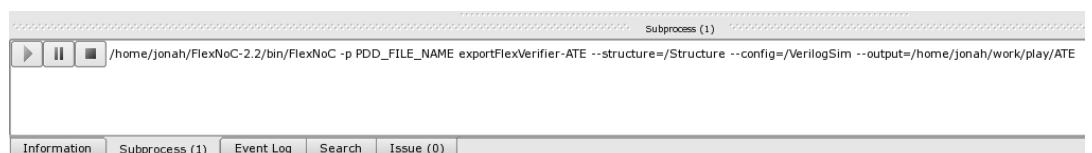
- 
- The screenshot shows the 'Netlist' window with the following content:
- | Name | Value |
|---|----------------------|
|  clockGatingAlgorithm | NoClockGating |

Now there should be no issues.

4. Now we can prepare export options for the NoC. We must define an export option for Verilog simulation, which will be used by the Flex Verifier Automatic Test Environment (ATE).
 - i. In the Project Tree right click and select New → ExportOption from the menu. Name it VerilogSim.
 - ii. Double click it to open its parameter editor.
 - iii. Change the exportOption parameter to Verilog
 - iv. Click [+] to expand the view of sub parameters.
 - v. Set the simulator parameter to your choice of simulator (NcSim, VCS or ModelSim).



5. Export area estimation report
 - i. Select Tools → Area → Export Area Estimation
6. Next we will use the Automated Test Environment (ATE) to export the design, test bench, scenarios, and other files necessary for an automatic self test simulation.
 - i. Select the Structure and VerilogSim export option in the Project Tree and choose Tools → Flex Verifier → Export ATE from the top menu bar.
 - ii. Create a new directory named ATE for the simulation files and click OK to the export the NoC and testbench Verilog RTL and test sequences.
 - iii. Note that the process is now shown in the subprocesses pane at the bottom of the screen.



- iv. When the export has finished, go back to your terminal window and cd to the ATE directory.
- v. Use `öls -lö` to see the test environment you just generated.
- vi. To run a simple connectivity test in a Verilog simulation, enter `ö./run.sh Connectivityö` on the command line.
- vii. When the simulation has completed, look at the file `öConnectivity.reportö` in the ATE directory. All paths should be OK.

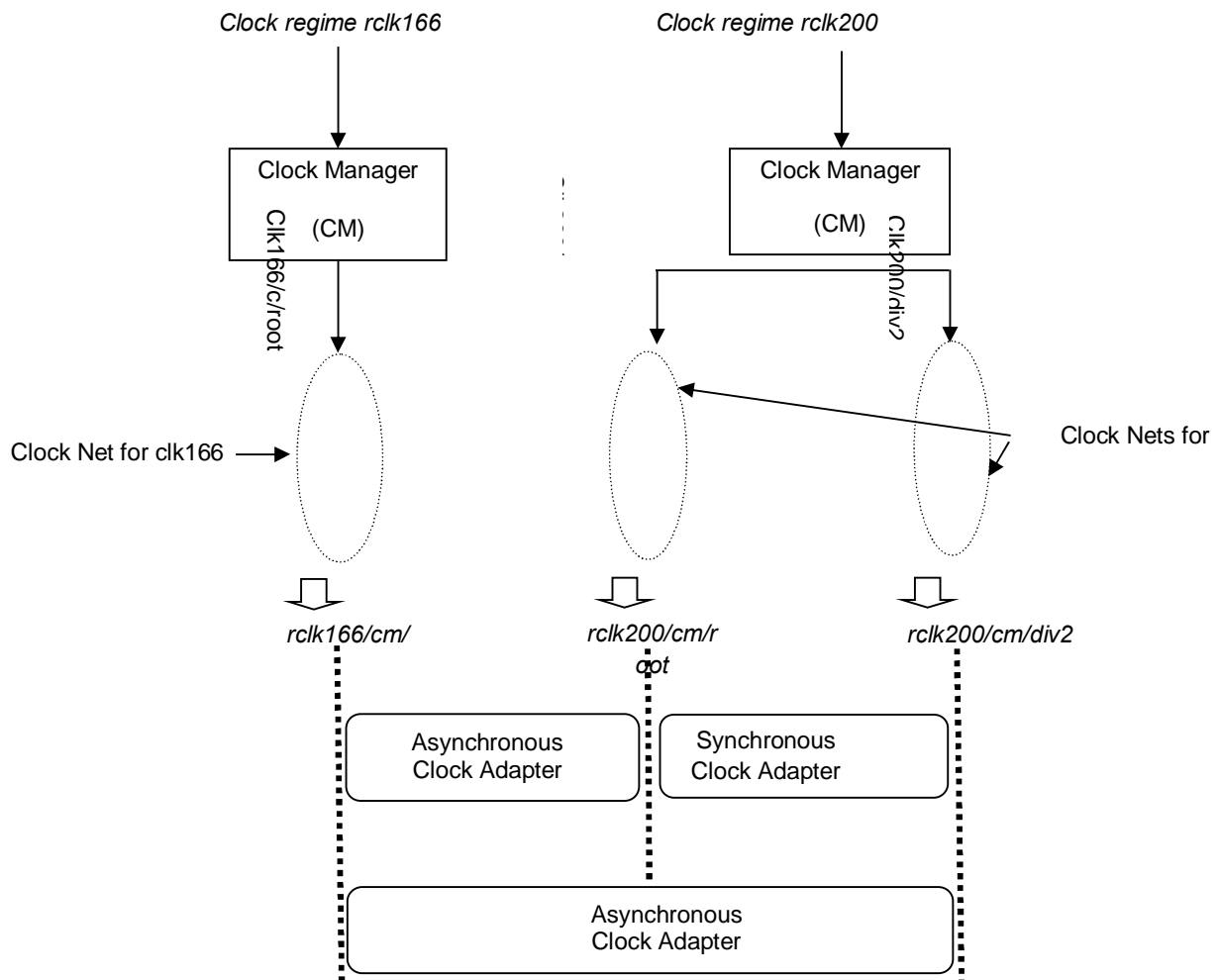
```
# Diagnostic 'Connectivity' on NoC 'Specification,Architecture,Structure' in mode 'Default' ; OK
```

```
-- Connectivity :
//InitSocket [InitId] InitAddrRange -> TargSocket [TargId] - AccessType AccessLength
CPU 0 [0x80000000:0x0] -> DRAM 0 - WR 8 bytes : OK
CPU 0 [0x80000000:0x0] -> DRAM 0 - RD 8 bytes : OK
CPU 0 [0x90000000:0x80000000] -> ROM - WR 8 bytes : OK
CPU 0 [0x90000000:0x80000000] -> ROM - RD 8 bytes : OK
CPU 0 [0x100000000:0x90000000] -> Expected Error - WR 8 bytes : OK
CPU 0 [0x100000000:0x90000000] -> Expected Error - RD 8 bytes : OK
DMA 0 [0x80000000:0x0] -> DRAM 0 - WR 16 bytes : OK
DMA 0 [0x80000000:0x0] -> DRAM 0 - RD 16 bytes : OK
DMA 0 [0x100000000:0x80000000] -> Expected Error - WR 16 bytes : OK
DMA 0 [0x100000000:0x80000000] -> Expected Error - RD 16 bytes : OK
DSP 0 [0x80000000:0x0] -> DRAM 0 - WR 8 bytes : OK
DSP 0 [0x80000000:0x0] -> DRAM 0 - RD 8 bytes : OK
DSP 0 [0x90000000:0x80000000] -> ROM - WR 8 bytes : OK
DSP 0 [0x90000000:0x80000000] -> ROM - RD 8 bytes : OK
DSP 0 [0x100000000:0x90000000] -> Expected Error - WR 8 bytes : OK
DSP 0 [0x100000000:0x90000000] -> Expected Error - RD 8 bytes : OK
ENET 0 [0x80000000:0x0] -> DRAM 0 - WR 4 bytes : OK
ENET 0 [0x80000000:0x0] -> DRAM 0 - RD 4 bytes : OK
ENET 0 [0x90000000:0x80000000] -> ROM - WR 4 bytes : OK
ENET 0 [0x90000000:0x80000000] -> ROM - RD 4 bytes : OK
ENET 0 [0x100000000:0x90000000] -> Expected Error - WR 4 bytes : OK
ENET 0 [0x100000000:0x90000000] -> Expected Error - RD 4 bytes : OK
USB 0 [0x80000000:0x0] -> DRAM 0 - WR 8 bytes : OK
USB 0 [0x80000000:0x0] -> DRAM 0 - RD 8 bytes : OK
USB 0 [0x100000000:0x80000000] -> Expected Error - WR 8 bytes : OK
USB 0 [0x100000000:0x80000000] -> Expected Error - RD 8 bytes : OK
```

7. Save the design using the name FlexNoC_Lab1.pdd.

Objective: Create new clocks and clock domains and investigate and manipulate the boundaries between the domains.

The purpose of the first part of this lab is to add a divided clock to the design and partition the design into 3 clock domains as shown in the diagram below. In the second part the objective is to observe the default serialization, take a closer look at the boundaries between different serializations, and modify the topology to minimize the discontinuities.



Procedure:

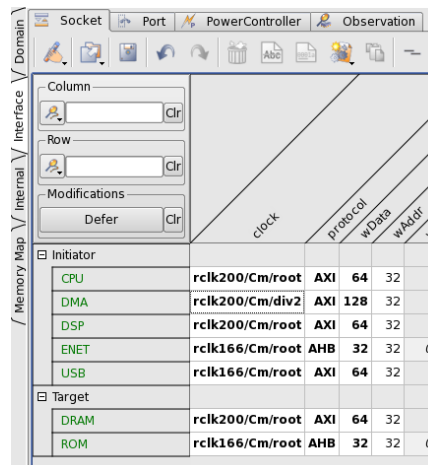
Divided Clock

1. Open the final Lab1 file in the FlexNoC GUI by entering `FlexNoC -p FlexNoc_Lab1.pdd &` at the command prompt.
2. Open the Specification Editor. In the Domain major tab and Clock minor tab, there are two clock regimes `rclk166` and `rclk200`. Define two clock nets that are derived from the same clock port `clk200`.
 - iv. In the Clock tab, select the clock `/rclk200/cm/root`, then right click and select **Duplicate**. Rename the duplicated divided clock `div2`

- v. Set the **type** to *Static* so that the divide ratio is constant.
- vi. Set the **ratio** of the divided clock to 2 as shown below:

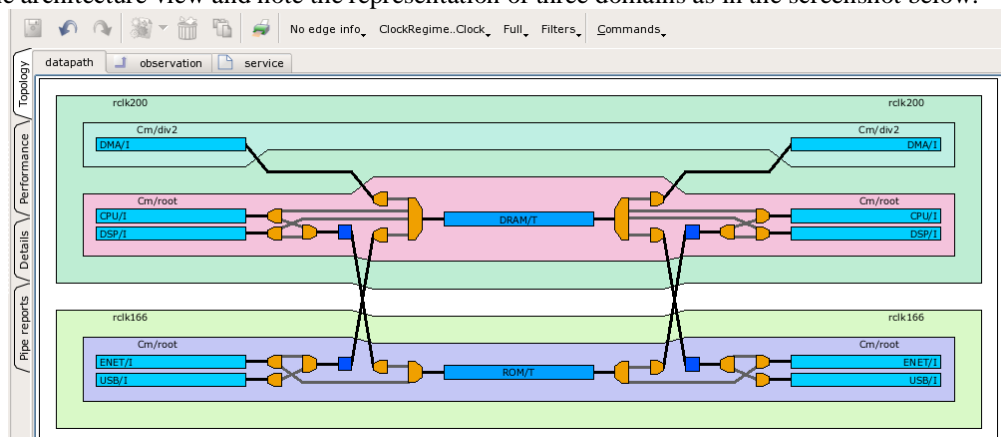
		Clock		Power		Voltage					
Domain	Interface	Internal	Memory Map		voltage	frequency	power	type	ratio	synthesistato	comment
				<input type="checkbox"/>							
				<input type="checkbox"/>	rclk166	166 MHz	None				
				<input type="checkbox"/>	Cm		None				
					root			Root			
				<input type="checkbox"/>	rclk200	200 MHz					
				<input type="checkbox"/>	Cm		None				
					div2			Static	2		
					root			Root			

3. In the Interface, Port tab select *rclk200/cm/div2*, the slowest clock, for the reset of the rclk200 domain.
4. In the Internal, ClockManager tab select **Generated** for the *div2* clock so that the divide by 2 clock is generated by the Clock Manager. The enable used internally for clock crossings defaults to Generated for a static divided clock.
5. In the Interface, Socket tab, associate a socket with the divided clock domain.
 - a. For CPU, DSP, and DRAM, leave the **clock** as *rclk200/cm/root*
 - b. For DMA change its **clock** to *rclk200/cm/div2*



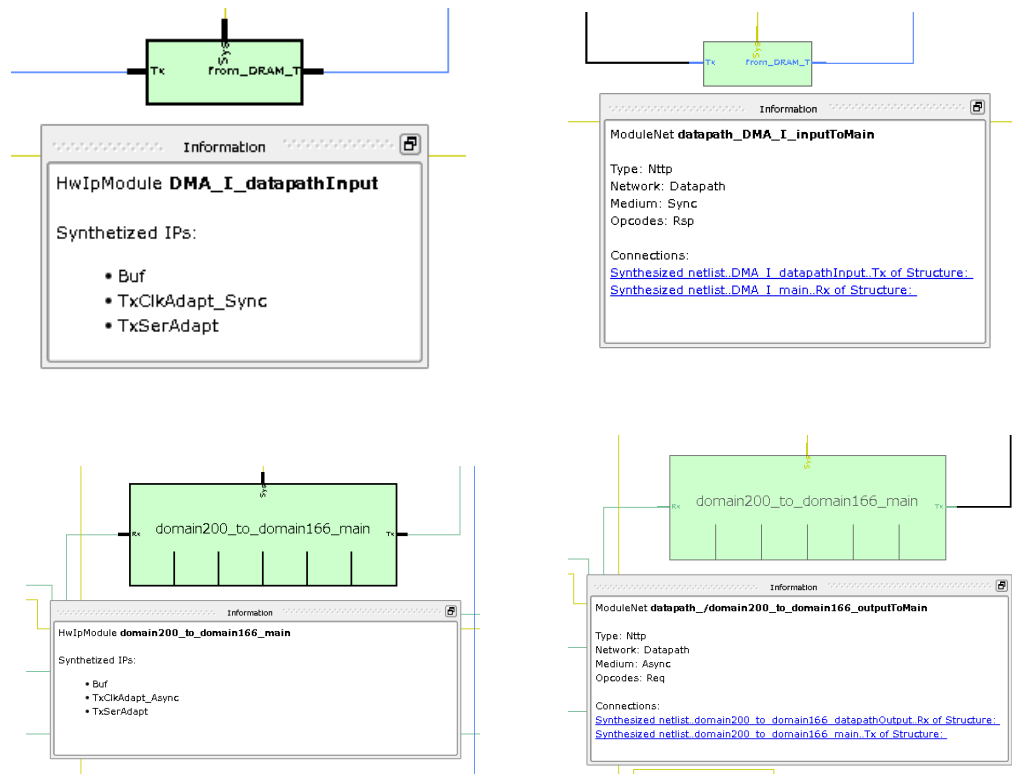
Initiator	clock	protocol	wData	wAddr	wData
CPU	rclk200/Cm/root	AXI	64	32	
DMA	rclk200/Cm/div2	AXI	128	32	
DSP	rclk200/Cm/root	AXI	64	32	
ENET	rclk166/Cm/root	AHB	32	32	0
USB	rclk166/Cm/root	AXI	64	32	
Target					
DRAM	rclk200/Cm/root	AXI	64	32	
ROM	rclk166/Cm/root	AHB	32	32	0

- c. For ENET, ROM and USB leave the **clock** as *rclk166/cm/root*
6. Open the architecture view and note the representation of three domains as in the screenshot below:



7. Open the Structure editor. As expected, a synchronous clock adapter is inserted between the DMA initiator and the DRAM target (DMA and DRAM are clocked by two clock nets belonging to the same clock regime öclk200ö). However, there is an asynchronous clock adapter between the DSP initiator and the ROM target (DSP and ROM are clocked by two clock nets belonging to different clock regimes: öclk200ö and öclk166ö). Select the blocks in the Structure editor and look at the **Information** window for details of the contents of the block.

Note that the selected blocks show Async or Sync ClkAdapt.



Save the project Lab_2.pdd.

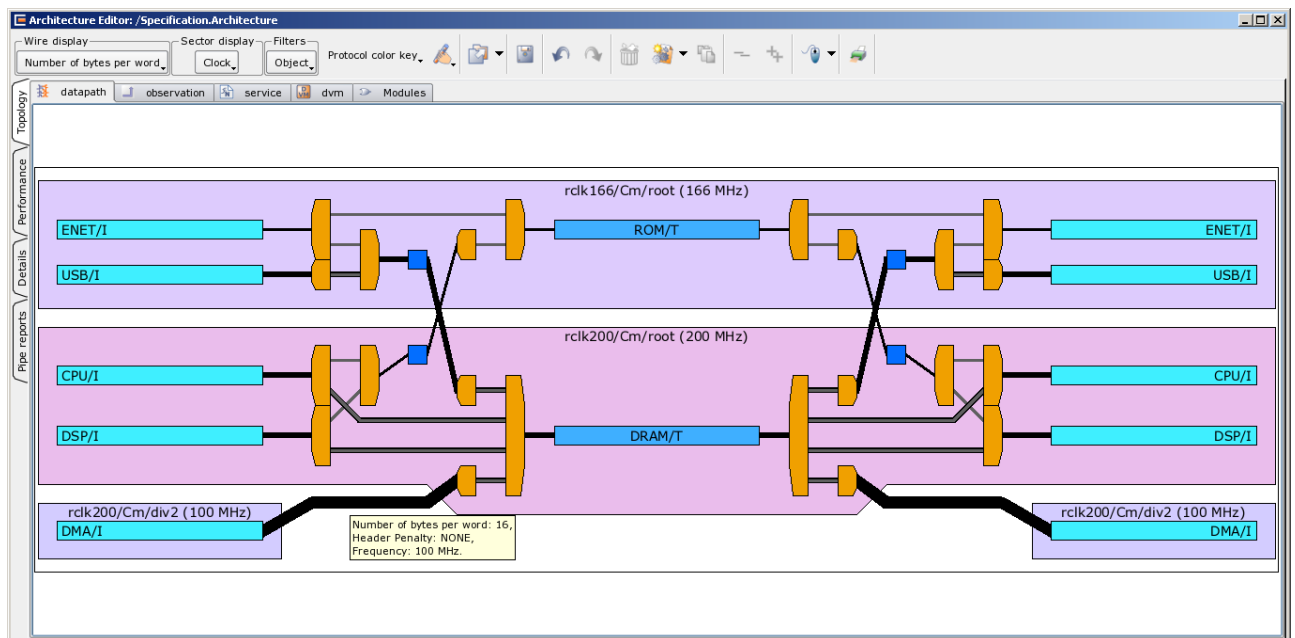
Objective: Investigate and manipulate links and the boundaries between the domains.

The purpose of this lab is to observe the default serialization, take a closer look at the boundaries between different serializations, and modify the topology to minimize the discontinuities.

Procedure:

Serialization and Boundaries

1. Open Lab_2.pdd.
2. Open the Architecture Editor Topology, datapath view and display the header penalty for each NIU. Mouseover to see the more values for different connections.
 - i. Change **No edge info** to **Header Penalty** in the middle of the tool bar. Mouseover to see the bytes per word, header penalty, and frequency values for the connections.



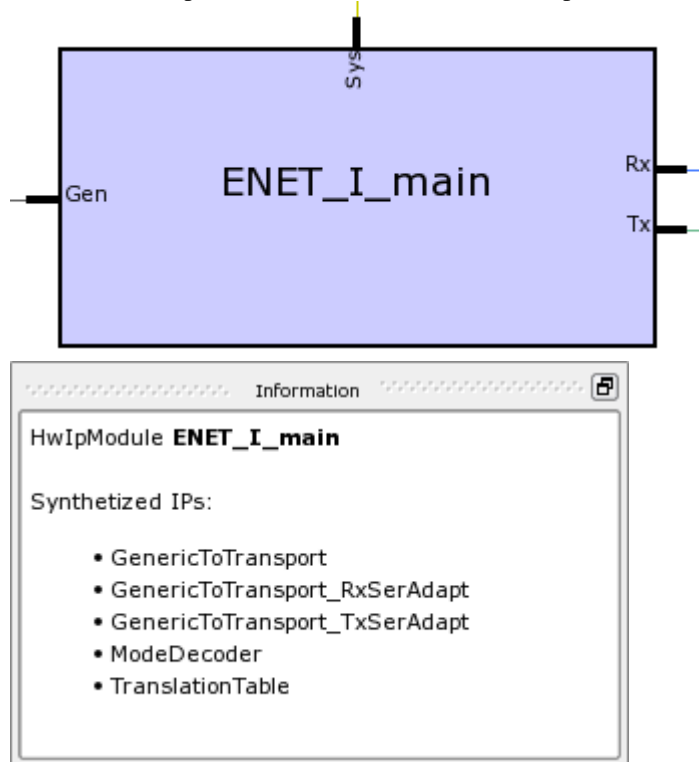
To conserve the number of wires, hdrPenalty is set by default according to the nBytePerWord settings.

nBytePerWord	Default hdrPenalty	Assumption
2 or 1	AUTO	The narrowest possible link width is desired.
4	TWO	The header size is approximately twice the width of data signals and if the header size is slightly larger it is better to have a few more wires than 3 cycles of header delay.
8	ONE	The header size is approximately the same as the width of data signals and if the header size is slightly larger it is better to have a few more wires than 2 cycles of header delay.
16 or more	ZERO	The maximum bandwidth is desired and the number of wires is unimportant for the link.

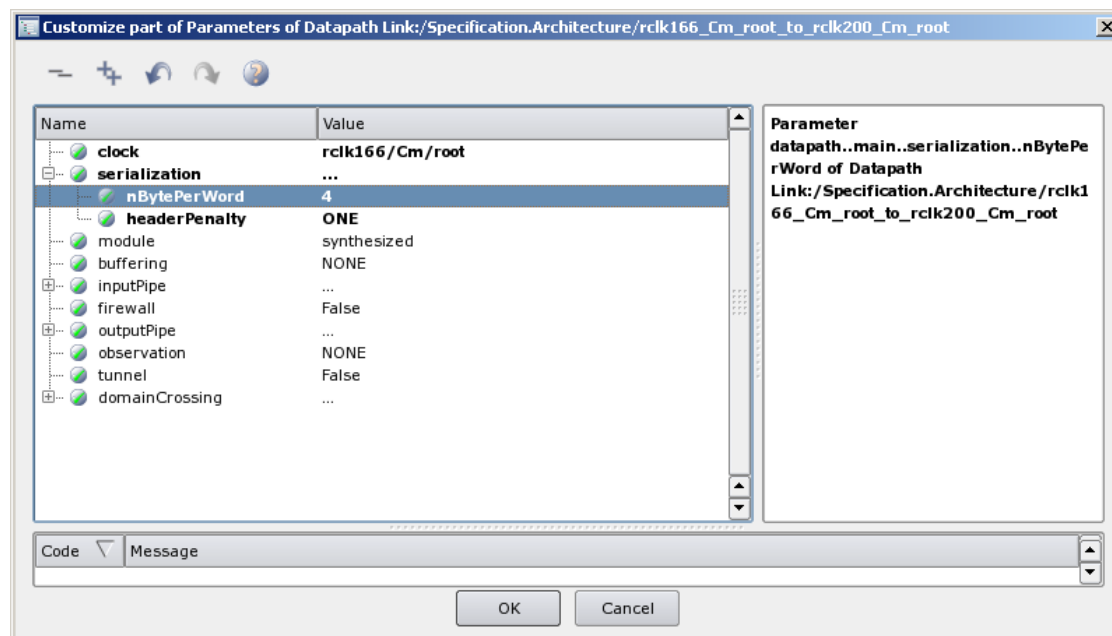
ENET and ROM have nBytePerWord = 4, therefore hdrPenalty is set to TWO. USB, CPU, DSP and DRAM have nBytePerWord = 8, therefore hdrPenalty is set to ONE. DMA has nBytePerWord = 16, therefore hdrPenalty is set to NONE.

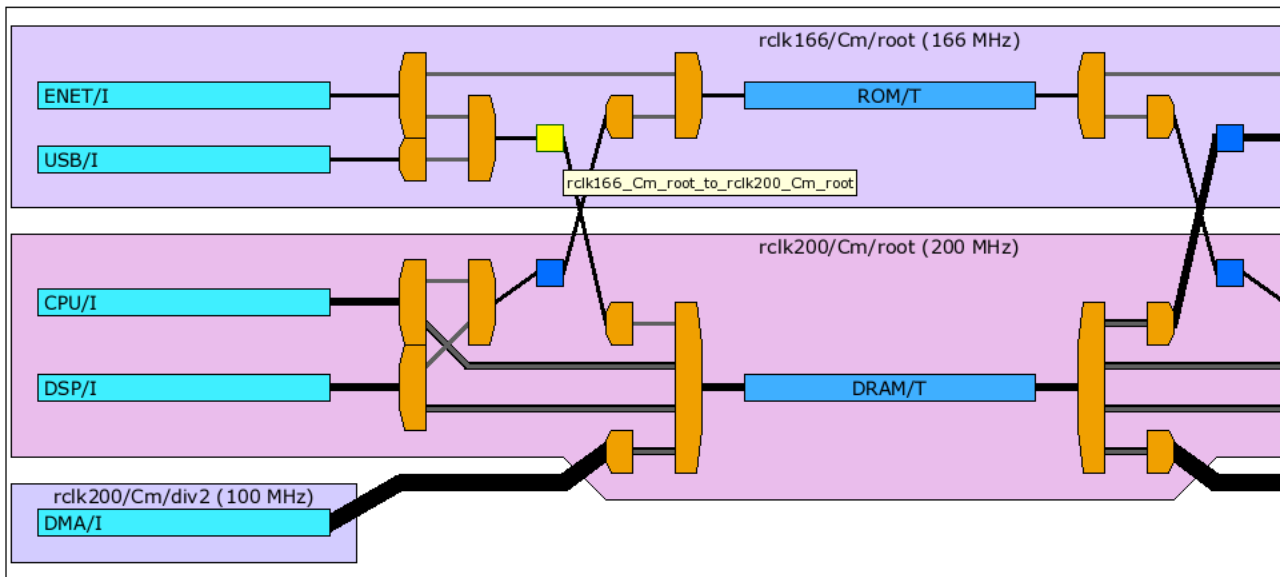
The NoC synthesis engine will recognize that for maintaining peak throughput, the route going from ENET to DRAM through the pseudo-switch should be 8 bytes wide since the DRAM is 8 bytes wide. Hence serialization

adapters will be inserted at the ENET request demux input, and at the ROM target request mux input. The serialization adapter can be seen in the Information pane of the Structure Editor window.



To reduce the size of the async adapter buffer, change the request network link between USB and DRAM from 8 bytes to 4 bytes.





Review the serialization change in the Architecture → Performance → Serialization tab.

			Specific NIU	Generic NIU	Scheduler	Serialization	Buffering	Arbiter	Splitting	Shaping
Topology	Column									
Performance	Row									
Details	Modifications									
Pipe reports										
Datapath Link										
rclk166_Cm_root_to_rclk200_Cm_root			4	ONE						
rclk166_Cm_root_to_rclk200_Cm_rootResp			8	ONE						
rclk200_Cm_root_to_rclk166_Cm_root			4	ONE						
rclk200_Cm_root_to_rclk166_Cm_rootResp			4	ONE						
Initiator Generic NIU										
CPU/I			8	ONE						
DMA/I			16	NONE						
DSP/I			8	ONE						
ENET/I			4	TWO						
USB/I			8	ONE						
Target Generic NIU										
DRAM/T			8	ONE						
ROM/T			4	TWO						

This decreased the bandwidth available for requests from ENET and USB to DRAM. This is acceptable if those initiators do more reads than writes. We left the corresponding link in the response path with the maximum necessary bandwidth.

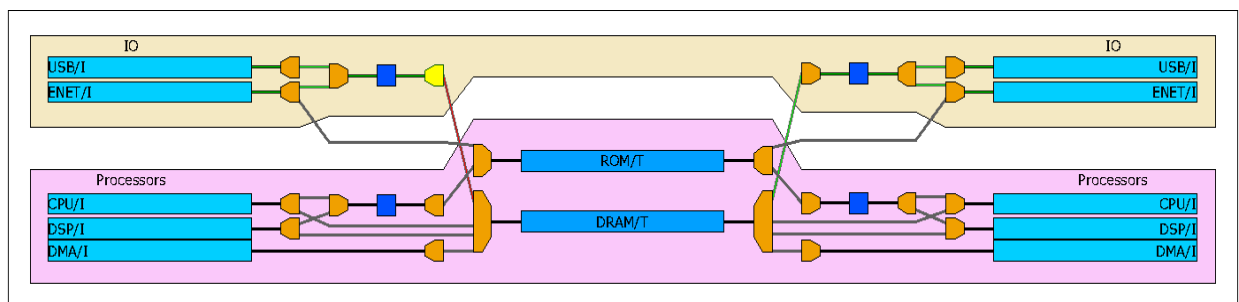
Objective: Create and manipulate hierarchy using modules and manipulate topology using 3 simple commands

The purpose of the first part of this lab is to add hierarchy to the design and move blocks into the new hierarchies. In the second part the objective is to manipulate the topology based on design and implementation requirements.

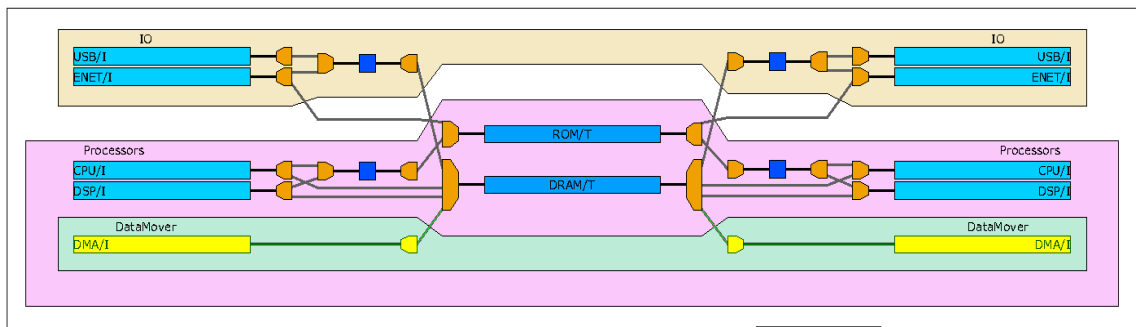
Procedure:

PART 1: Hierarchy Manipulation

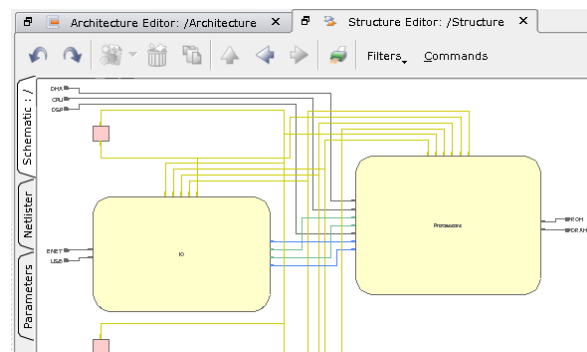
1. Open the final Lab1 file in the FlexNoC GUI by entering `FlexNoC -p FlexNoC_Lab1.pdd &` at the command prompt.
2. Open the topology view of the Architecture Editor and click the pull down `Sector display` tab displaying **Clock Regime..Clock** in the Toolbar of the Architecture Editor and select **Module**. Now modules may be seen in the Architecture Editor. Note that there are no Modules defined yet.
 - i. Right click in the Architecture Topology area and select **New -> Module**. Name it *Processors* and click **OK** or press ENTER.
 - d. Click and drag the CPU, DSP and DMA initiators; and the ROM and DRAM targets into the new module
 - ii. Right click in the Architecture Diagram area and select **New -> Module**. Name it *IO* and note that when you create this module you have a choice of where to put it - either in the top level (*Architecture*) or inside *Processor*. Click on the arrow to the right of *Architecture*, choose *Architecture* and click **OK** or press ENTER.
 - iii. Click and drag the USB and ENET initiators into the IO module
 - iii. Finish separating the design into two hierarchies by dragging the pseudo-switches into their corresponding Modules.
 - vii. The resulting design should look like the screenshot below:



- iv. Add another level of hierarchy inside the Processors module. Click to select the *Processors* module and then add a new module called *DataMover*. Note that the default location is inside *Processors* because the *Processors* module was selected.
 - a. Drag the DMA and its nearest mux and demux into the *DataMover* module. The resulting design should look like the screenshot below:



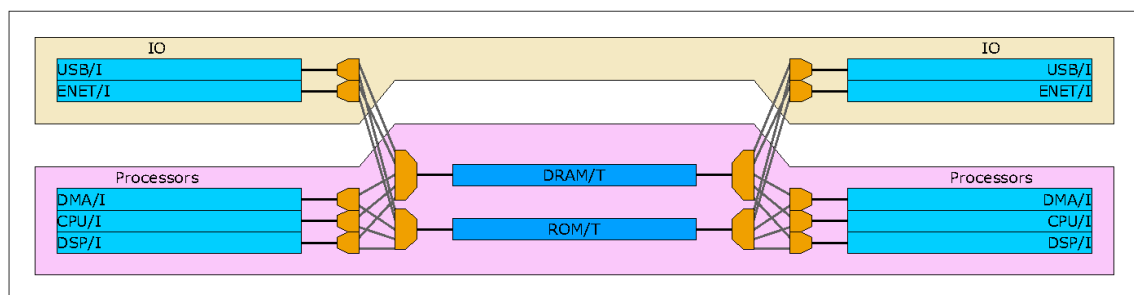
3. Open the Structure Editor, Schematic view. Note that the modules are represented as yellow boxes.



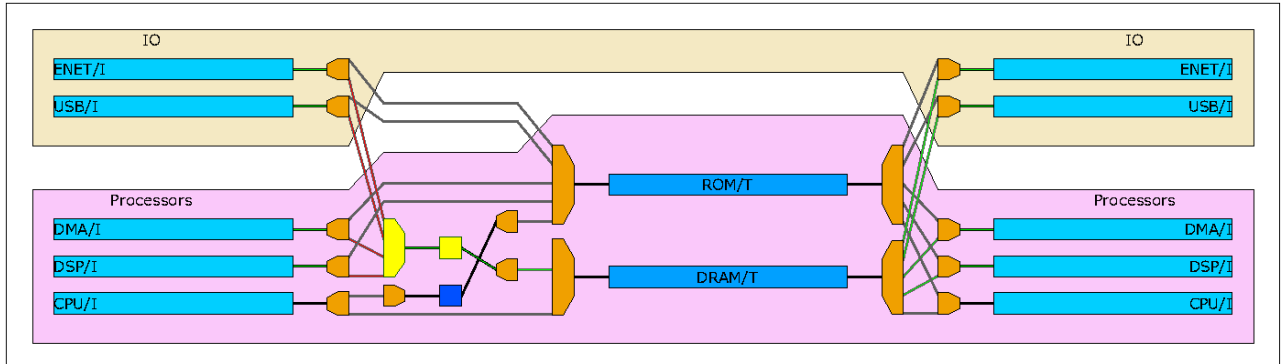
- i. Double click on the *IO* module to see the structure inside.
 - ii. Use the up arrow to go up to the top level of hierarchy or use the left (back) arrow to go to the previous level of hierarchy.
 - iii. Select the *Processors* block and use the down arrow to enter that module. There is another black box inside which can be investigated.
4. Go back to the Architecture Editor, select the *DataMover* module and right click and select **Dissolve modules** to remove that level of hierarchy.


PART 2: Topology Manipulation

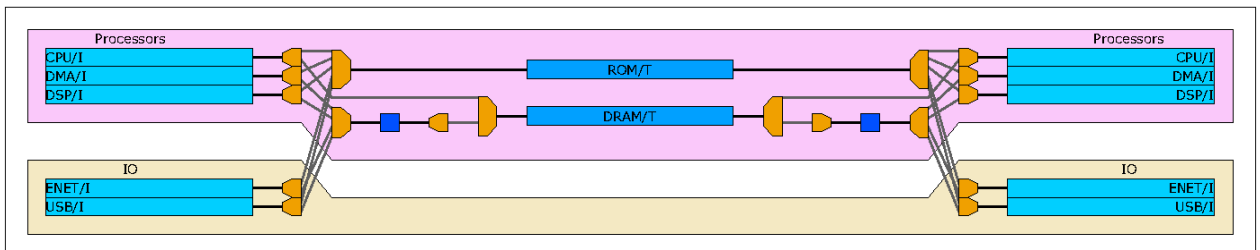
5. Remove the topology and connectivity changes from Lab 1 to make it easier to see the effects of the upcoming topology changes.
 - i. In the Architecture Editor, remove the topology changes from Lab 1 by selecting all the links (blue boxes) right click, and choose **delete**. Hold the CTRL key down while left clicking to make multiple selections.
 - ii. Go to the Specification Editor and add the two missing connections to make the design fully connected. The topology should look like the screenshot below:



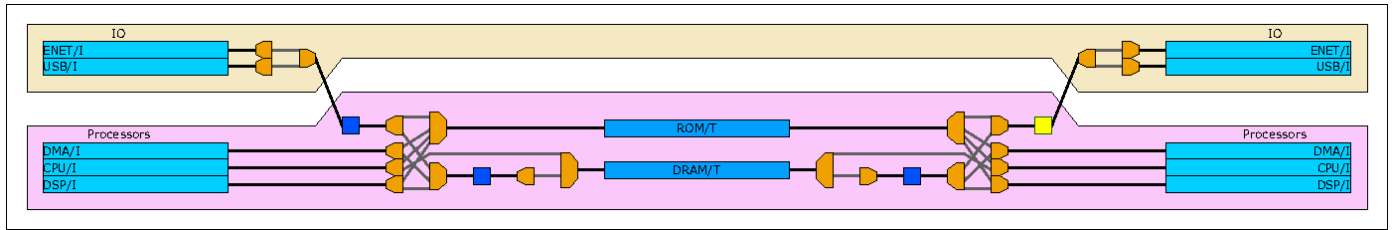
6. What if you need to minimize the latency and the amount of contention the CPU encounters when accessing the DRAM? To make a more direct path from a high priority, low latency core like the CPU to a shared resource like the DRAM, use **Divide Switch**.
 - i. Select the demux closest to the CPU and the mux closest to the DRAM (both on the request side) to define the path from the CPU to the DRAM
 - ii. Right click and select **Divide Switch**.
 - iii. Select everything not in placed in either hierarchy and put in the Processors module. Note that there are now 2 inputs to the mux feeding the DRAM. One for the CPU, and one for everything else. The CPU has a shorter path to the DRAM and only has to arbitrate with the winner of the arbitration of the other 4 initiators. The CPU now has a more direct and efficient path to the DRAM. The resulting topology should look like the screenshot below:



- iv. If the response path needs to be improved as well, use the commands menu  → **Force dual topology**.
7. To remove unnecessary links and cones (1 input, 1 output muxes or demuxes), use **Delete link**.
 - i. Begin with the results from Step 6 above. Select the link closest to the CPU.
 - ii. Right click and select **Delete**.
 - iii. Note that this has removed the link and the cones on either side of the link in the path from the CPU to the ROM, simplifying the less critical path. Again, **Incremental dual topology** may be used to make the same optimizations to the response path. Delete the unconnected link and cones.



8. What if the 2 hierarchies are actually 2 different FPGAs? It would be a waste of precious FPGA IO resources to have 8 connections between the FPGAs, so how can we minimize the connectivity between the modules? Use **Move segments** to group multiple connections together.
 - i. Begin with the results from Step 8 above. Select the 2 demuxes closest to the ENET and USB and the 2 muxes closest to the ENET and USB in the Processors module to define the 4 request paths from the *IO* hierarchy to the *Processors* hierarchy.
 - ii. Right click and select **Move segments**
 - iii. Select the mux to the left of the link and move it into *IO*, and select the demux to the right of the new link and the new link and move them into *Processors*. We now have a single connection between the FPGAs on the request side.
 - iv. In this case, we want to use **Force dual topology** to make the same improvements to the response side. The resulting topology should look like the screenshot below:



9. Save the design using the name FlexNoC_Lab4.pdd

Objective: View, add and edit pipeline stages

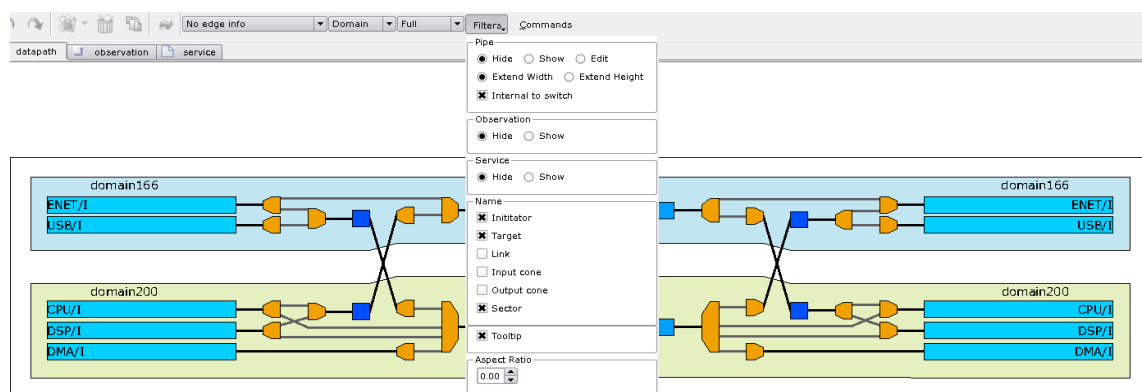
The purpose of this lab is to view and edit pipeline stages in order to meet timing constraints.

Once a topology is chosen based on the appropriate area/performance/congestion trade-offs, an important step for the back-end convergence of the design will be to add pipeline stages as necessary to achieve the target frequencies. The packet-based NoC architecture allows any amount of pipelining, so that pipeline stages can be inserted purely based on timing constraints.

Procedure:

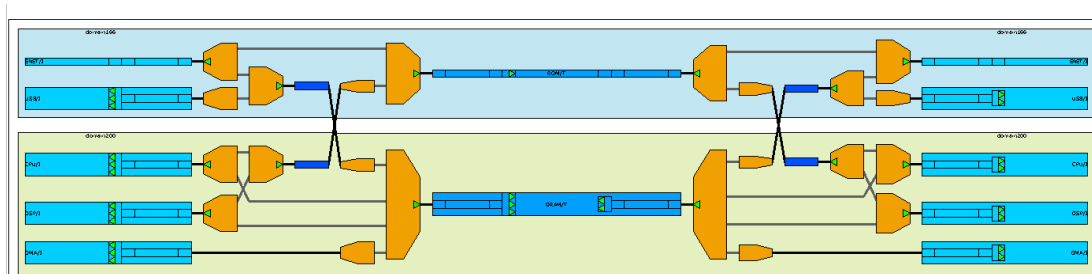
Begin by opening FlexNoC_Lab1.pdd

1. Open the Architecture Editor, topology View
2. Note that the **Filters** pulldown menu can alternate between modes that hide, display, or allow editing of the optional pipeline stages

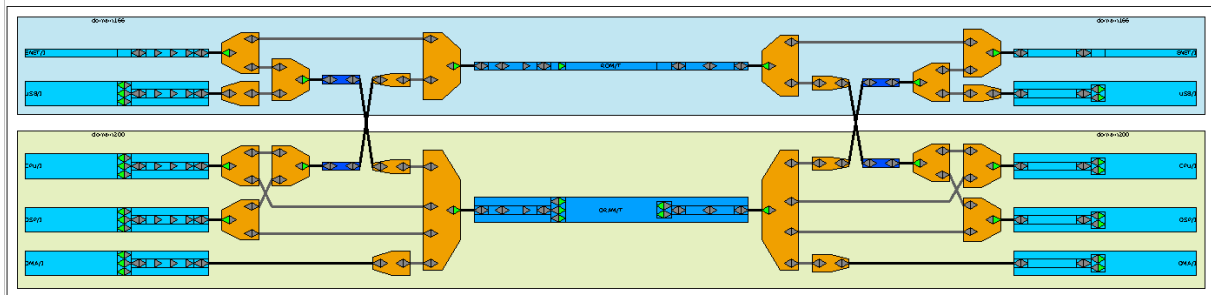


Pipeline stages can be inserted in the packet transport at the inputs and outputs of NoC architecture objects (muxes, demuxes, configurable links, and more) and also within initiator and target NIUs in several predefined places.

- i. Pipeline display mode of the Architecture editor shows where pipeline registers have been automatically inserted:



- ii. Pipeline editing mode also shows where pipeline stages can be inserted:



The GUI shows potential pipeline registers represented by gray triangles, and existing pipeline registers by light green triangles.

Pipeline stages may be enabled or disabled by selecting the triangle, right clicking and selecting **Add Pipe** or **Remove Pipe**. You can also toggle between enabled and disabled pipeline stages by double clicking on a triangle symbol.

In pipeline editing mode you can see that each individual input and output of muxes, demuxes, initiator NIU or target NIU can be set with one or both of two pipelining stages: forward pipeline or backward pipeline.

As their name implies, forward pipelining inserts pipeline stages on the signals travelling forward, from left to right, that is, the packet itself. Such pipeline stages break timing paths on the forward-moving signals, at the expense of area and a clock of latency. Conversely, backward pipelining breaks timing on the signals moving backward, that is, flow control. A backward pipe adds area but does not cost any latency.

3. Click on the **Pipe reports** tab and review the **datapath forward Pipes** minor tab. Identify the corresponding enabled pipeline stages in the CPU to ROM request and response paths.

Architecture Editor: /Architecture

Topology		datapath forward Pipes		datapath backward Pipes
req+Res		DRAM/T/I/O		ROM/T/I/O
CPU/I/O	2+2	3+2		
DMA/I/O	2+1			
DSP/I/O	2+2	3+2		
ENET/I/O	3+1	2+1		
USB/I/O	3+1			

4. Do the same for the **datapath backward Pipes**.

Topology		datapath forward Pipes		datapath backward Pipes
req+Res		DRAM/T/I/O		ROM/T/I/O
CPU/I/O	2+2	2+2		
DMA/I/O	1+2			
DSP/I/O	2+2	2+2		
ENET/I/O	1+3	1+1		
USB/I/O	1+3			

Objective: To learn how to integrate a service network for NoC error logging

The purpose of this lab is to learn how to implement observers and service register sockets in order to integrate a service network into your design and to allow register configuration and error-detection/logging.

The example design is based on the Lab 1 file.

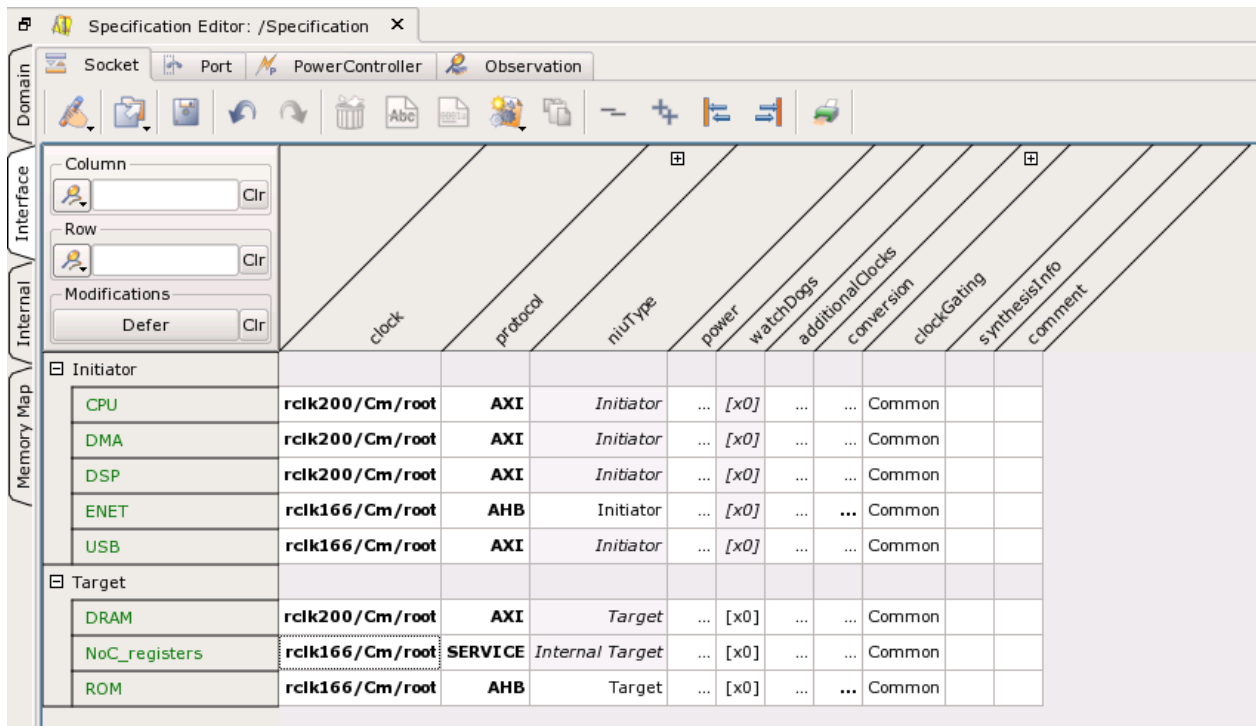
Procedure:

Open the Lab1 file in the FlexNoC GUI by entering `FlexNoC -p FlexNoc_Lab1.pdd&` at the command prompt; in order not to overwrite existing files, please save it as a new file such as **FlexNoC_Lab6.pdd**

Part 1: Specification

First introduce a new target socket of type SERVICE, it will be the access-point from the NoC to the service network

1. In the Project Tree area double click on the Specification, go to the Interface pane, select the Socket tab and create a new target socket with **right-click -> Socket (w/ Target)**
 - i. Rename it as NoC_registers
 - ii. Select SERVICE in the protocol column
 - iii. Set /domain166 in the domain column

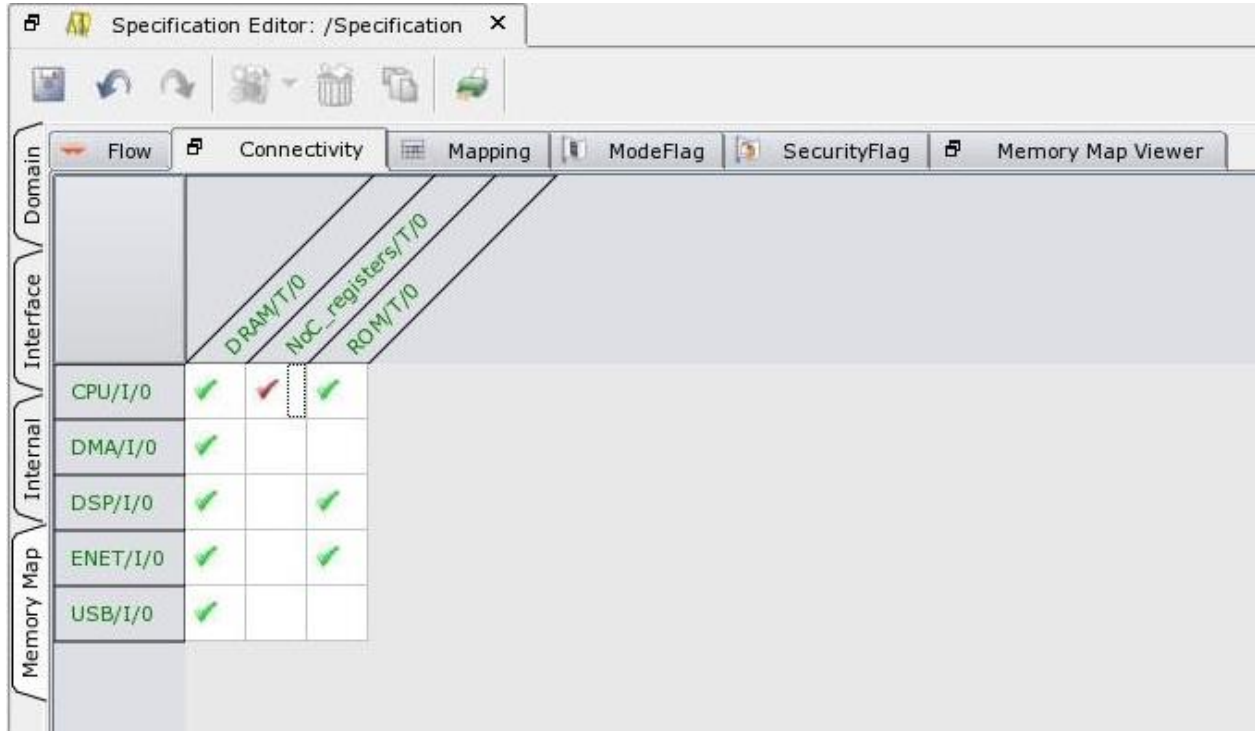


The screenshot shows the Specification Editor GUI with the Socket tab selected. The table below represents the data shown in the GUI:

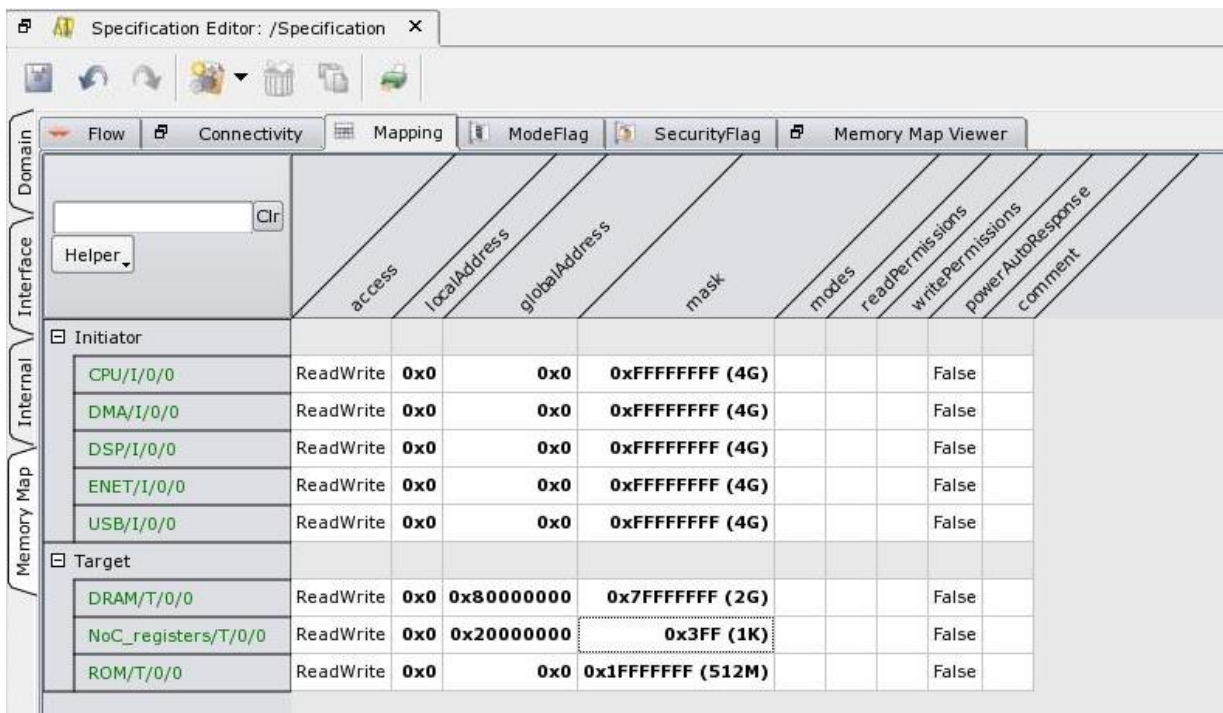
	clock	protocol	niuType	power	watchDogs	additionalClocks	conversion	clockGating	synthesisInfo	comment
Initiator										
CPU	rclk200/Cm/root	AXI	Initiator	...	[x0]	Common		
DMA	rclk200/Cm/root	AXI	Initiator	...	[x0]	Common		
DSP	rclk200/Cm/root	AXI	Initiator	...	[x0]	Common		
ENET	rclk166/Cm/root	AHB	Initiator	...	[x0]	Common		
USB	rclk166/Cm/root	AXI	Initiator	...	[x0]	Common		
Target										
DRAM	rclk200/Cm/root	AXI	Target	...	[x0]	Common		
NoC_registers	rclk166/Cm/root	SERVICE	Internal Target	...	[x0]	Common		
ROM	rclk166/Cm/root	AHB	Target	...	[x0]	Common		

FlexNoC Lab-6

2. Select the Memory Map pane and go to the Connectivity tab
 - i. Enable access from the CPU to the NoC_registers target socket enabling the corresponding connection



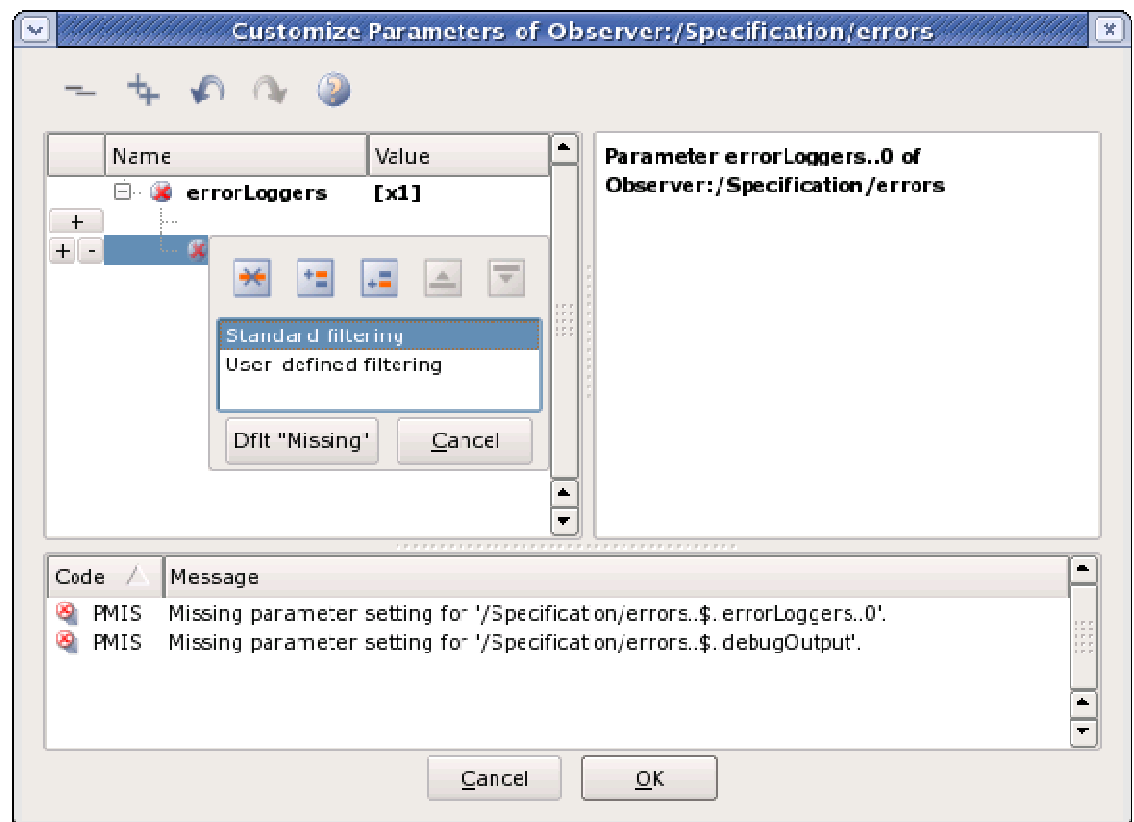
- ii. Map this target socket into the memory map. Go to the Mapping tab and re-arrange the memory mapping as shown below:



FlexNoC Lab-6

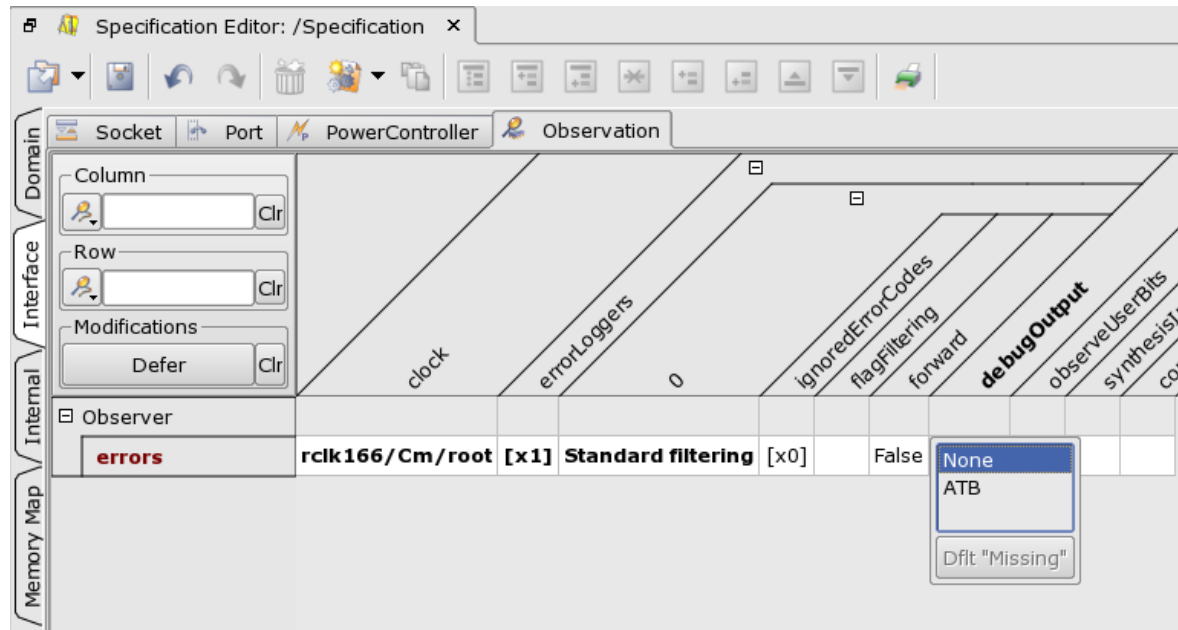
Looking to the *Issues* panel you can see some configuration is still missing. We have added a way for the CPU to access the Service Network, but there is nothing to access yet. Let's introduce an observer, which is a hardware unit that will gather and log error information in its internal registers, accessible via the service network. An observer can collect errors detected by target NIUs or by probed links; in this example we will enable the first class of error sources.

3. Introduce an observer
 - i. Open the Specification editor and under Interface, select the Observation tab
 - ii. Create a new Observer unit with **right-click** -> **New** -> **Observer** and rename it as *errors*
 - iii. In the for the clock parameter select rclk166/Cm/root.
 - iv. Click the errorLoggers parameter to customize the number and type of error loggers.
 - v. Click [+] to create one error logger and set it to use Standard Filtering.



FlexNoC Lab-6

- vi. For this lab choose None for debugOutput.



We have completed the specification part and we may now proceed with the architecture refinement.

Part 2: Architecture

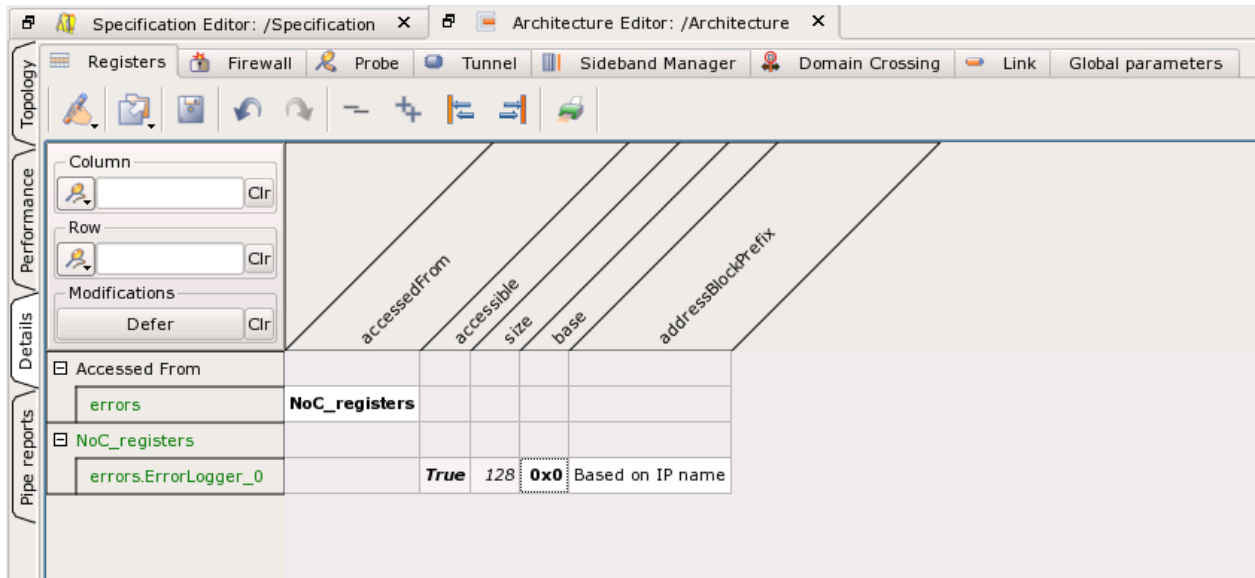
Now we need to refine the architecture.

4. Complete NoC_registers NIU parameterization. In the Project tree double click on the Architecture, or select the Architecture and open it with its editor (**right-click -> Open with -> Architecture Editor**)

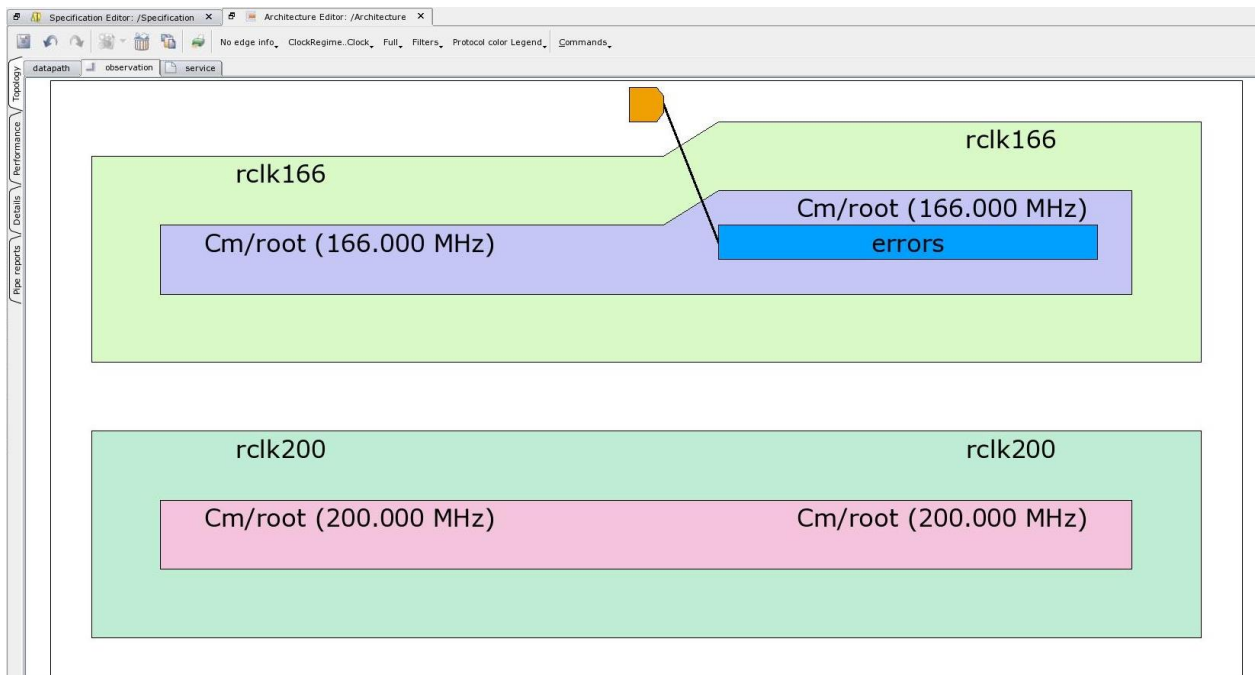
To solve the remaining issues, we need to reconfigure the *errors* observer we created in the Specification. In the Architecture Editor, select the Details tab

5. Describe the configured register and define its base
 - i. Make the errors register accessed from NOC_registers by pulling down the selection tab and choose the NoC_registers service network that we created in the specification.
 - ii. Set the base of this observer to 0x0. This will map the observer's registers at the first memory location of the service network memory segment

FlexNoC Lab-6



6. Map the Observer into the service network accessed through NoC_registers
 - i. Select the Topology major tab and select the observation minor tab: as you can see only errors is present in the observation pane. The Service Network will be enabled for both DRAM and ROM in the next step.



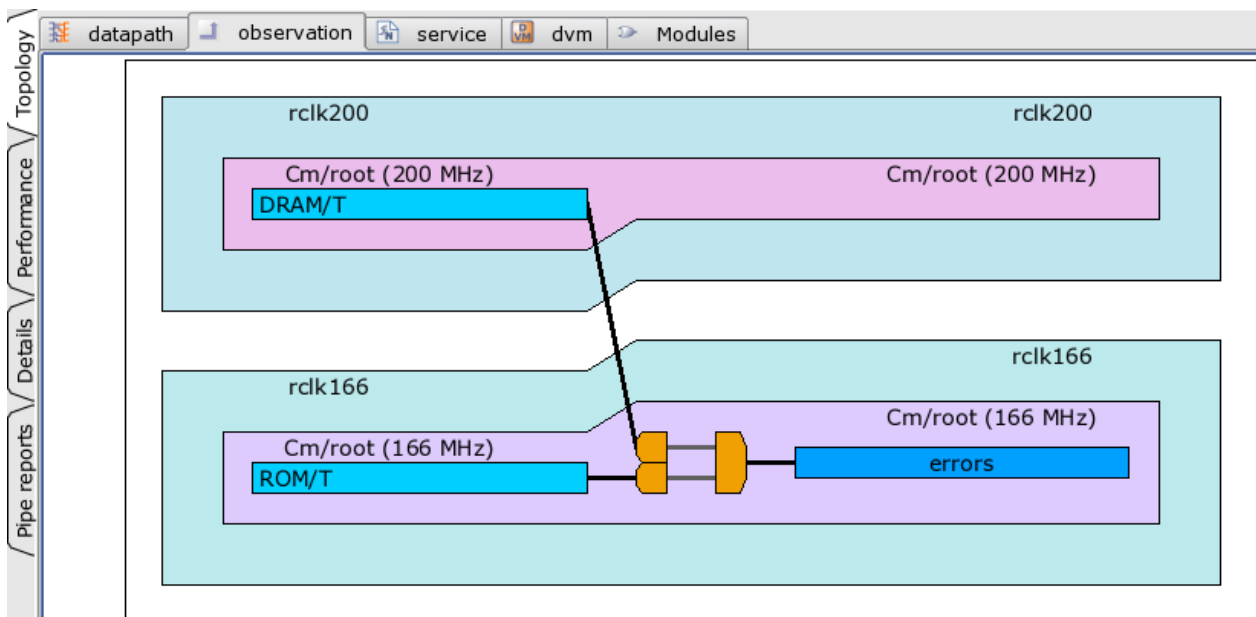
- ii. Let's enable the service network for both DRAM and ROM. For that go back to the datapath tab and double click on ROM/T. This will open a customizer window for the target NIU.
- iii. Change observation from NONE to ERRORS and select errors in the observer

FlexNoC Lab-6

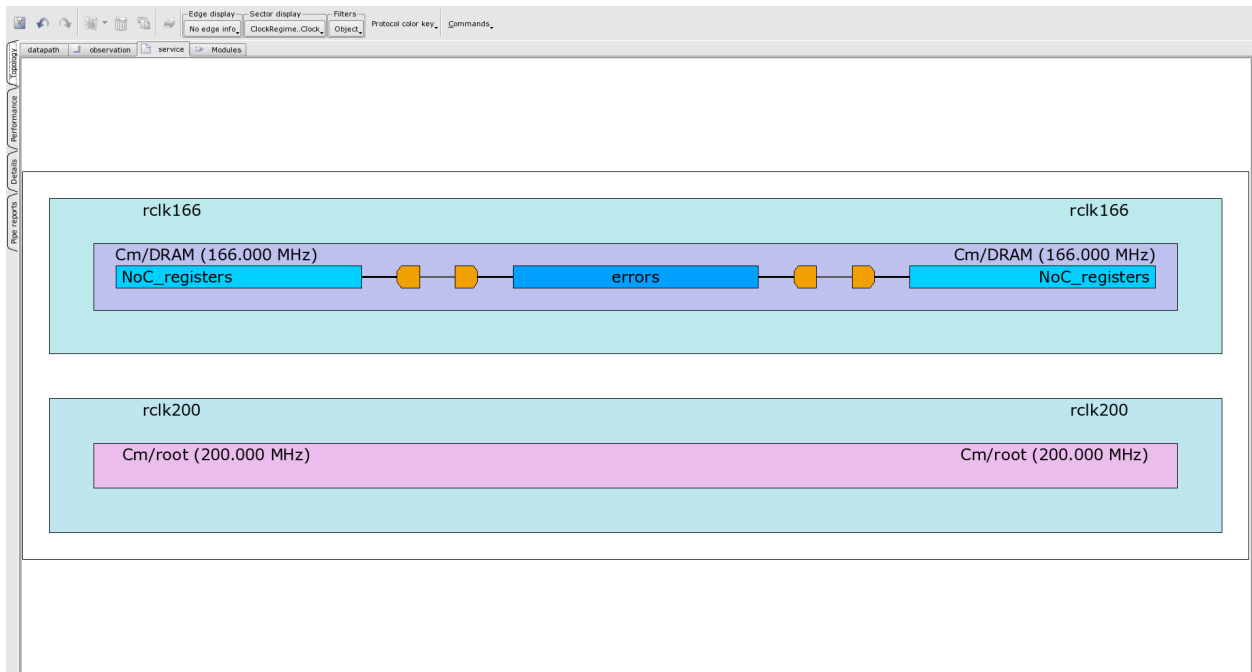
Customize part of Parameters of Target Generic NIU:/Architecture/ROM/T

Name	Value
module	None
serialization	...
performance	...
observation	ERRORS
withHeaderRegister	True
observer	errors
domainCrossing	...
inputPipe	...
outputPipe	...

- iv. Repeat step ii and iii for the DRAM
- v. Review the observation network minor tab



- vi. Now check the service network minor tab



Here you can see the created service network: requests come from the NoC_registers access point (on the left) and return to the same access-point as responses (right side).

In summary, to create a service network for accessing error logging registers:

- Introduce a SERVICE socket to provide an access-point to your service network from the NoC
- Introduce an observer to gather error information
- Enable error detection in the target NIUs you are interested in and associate them to the observer
- Map the observer into the service network

Objective: To learn how to enable boot-mode based memory mapping using FlexNoC specification

The purpose of this lab is to learn how to use specification mode-flags in order to change memory mappings. For this lab, suppose that we want to have a boot memory segment that can be selectively associated to either the ROM, the FLASH_CTRL or the DRAM

Procedure:

1. Open the Lab1 file with the FlexNoC GUI by entering `öFlexNoC -p FlexNoc_Lab1.pdd&ö` at the command prompt; in order not to overwrite existing files, please save it as a new one like **FlexNoC_Lab7.pdd**

Part 1: Specification

First let's modify the memory mapping in order to create a memory overlap between ROM and DRAM

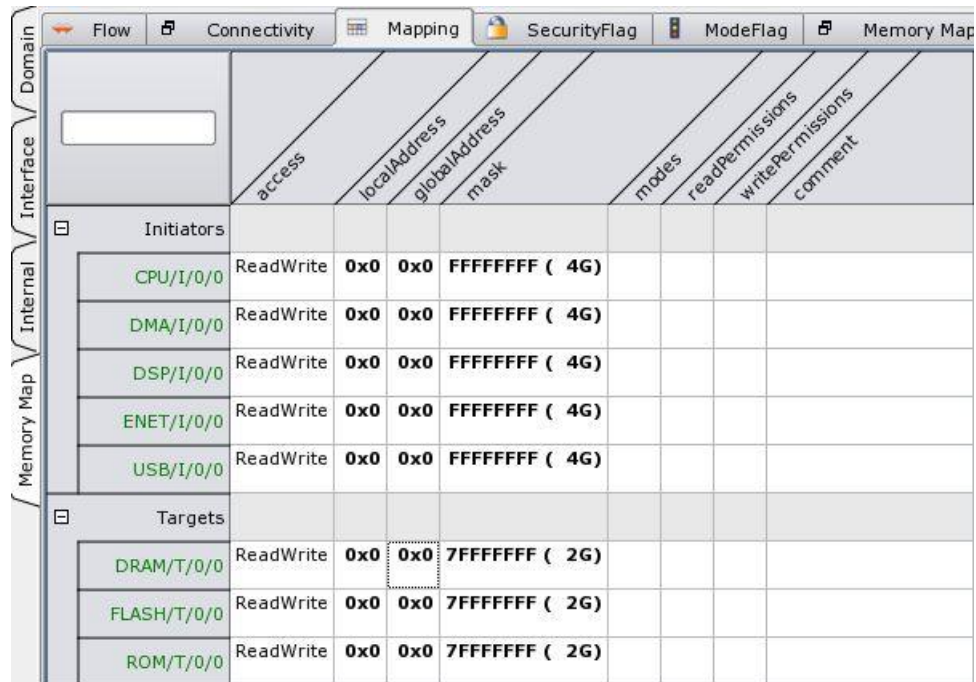
2. In the Project Tree area double click on the Specification, go to the Memory Map major panel and select the Mapping minor panel
 - i. Change ROM **globalAddress** from 0x80000000 to 0x0 and set its mask to 0x7FFF_FFFF (2G). This will intentionally create an address mapping conflict between the address ranges of ROM and DRAM.

We need to introduce a new target socket for the flash controller

3. Introduce a new socket for the flash controller IP
 - i. Go to the Interface pane and select the Socket tab
 - ii. Select DRAM socket and duplicate it using right-click and selecting **Duplicate**
 - iii. Select the new socket DRAM1 and rename it as FLASH.

Initiator									
CPU	rclk200/Cm/root	AXI	Initiator	...	[x0]		
DMA	rclk200/Cm/root	AXI	Initiator	...	[x0]		
DSP	rclk200/Cm/root	AXI	Initiator	...	[x0]		
ENET	rclk166/Cm/root	AHB	Initiator	...	[x0]		
USB	rclk166/Cm/root	AXI	Initiator	...	[x0]		
Target									
DRAM	rclk200/Cm/root	AXI	Target	...	[x0]		
FLASH	rclk200/Cm/root	AXI	Target	...	[x0]		
ROM	rclk166/Cm/root	AHB	Target	...	[x0]		

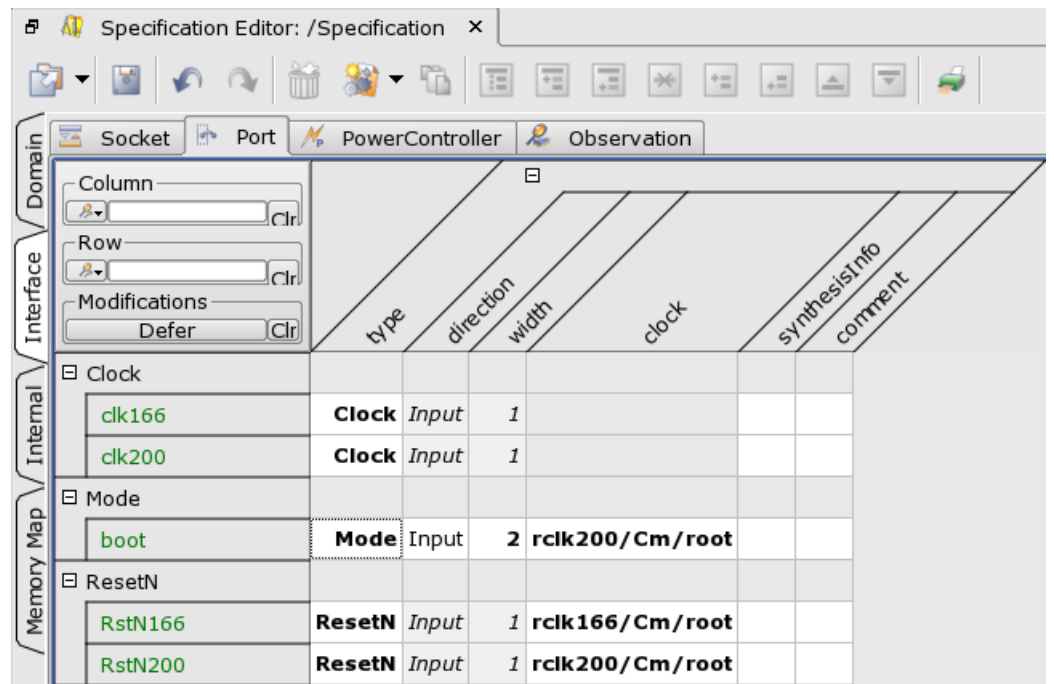
Select the Memory Map pane and go to the Mapping tab: you can see now that ROM, FLASH and DRAM share the same memory segment of size 2Gbytes starting at the address 0x0.



Domain	Interface	Internal	Memory Map	access	localAddress	globalAddress	mask	modes	readPermissions	writePermissions	comment
Initiators											
			CPU/I/0/0	ReadWrite	0x0	0x0	FFFFFFFF (4G)				
			DMA/I/0/0	ReadWrite	0x0	0x0	FFFFFFFF (4G)				
			DSP/I/0/0	ReadWrite	0x0	0x0	FFFFFFFF (4G)				
			ENET/I/0/0	ReadWrite	0x0	0x0	FFFFFFFF (4G)				
			USB/I/0/0	ReadWrite	0x0	0x0	FFFFFFFF (4G)				
Targets											
			DRAM/T/0/0	ReadWrite	0x0	0x0	7FFFFFFF (2G)				
			FLASH/T/0/0	ReadWrite	0x0	0x0	7FFFFFFF (2G)				
			ROM/T/0/0	ReadWrite	0x0	0x0	7FFFFFFF (2G)				

Now let's introduce a mode-port. For this project it will be a boot mapping pin input to our NoC that will be used as a select pin to choose from which target IP the system will boot.

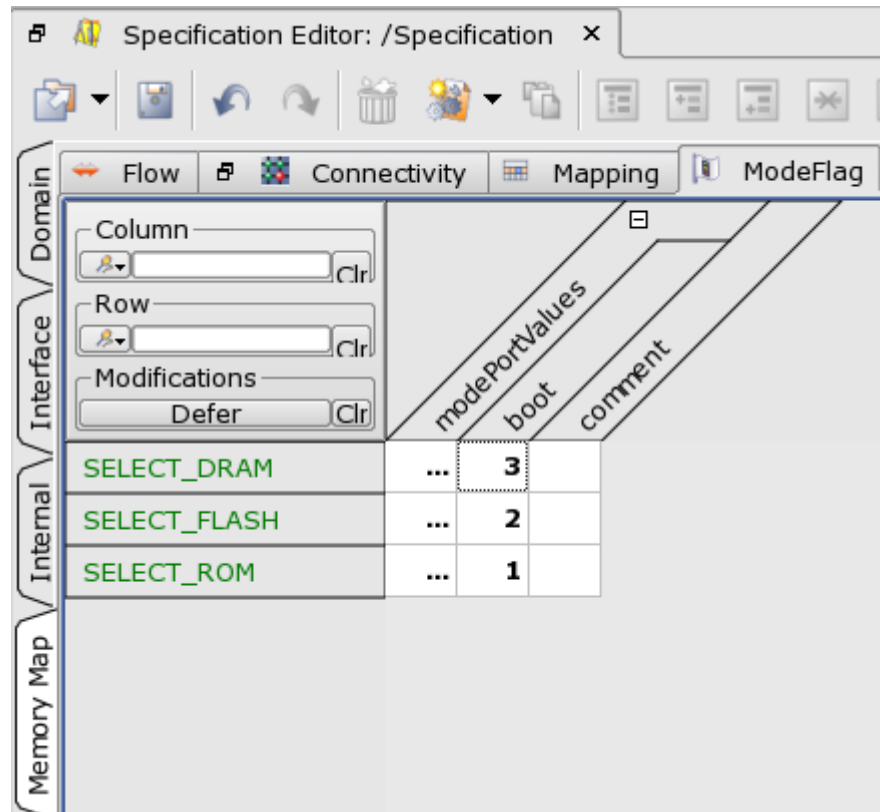
4. Introduce mode-port
 - i. Select the Interface pane and go to the Port tab
 - ii. Create a new port with **right-click New->Port**
 - iii. Give it the name *boot* and configure it as below (type = mode, width = 2, domain = rclk200/Cm/root)



Domain	Interface	Internal	Memory Map	type	direction	width	clock	synthesisInfo	comment
Clock									
			clk166	Clock	Input	1			
			clk200	Clock	Input	1			
Mode									
			boot	Mode	Input	2	rclk200/Cm/root		
ResetN									
			RstN166	ResetN	Input	1	rclk166/Cm/root		
			RstN200	ResetN	Input	1	rclk200/Cm/root		

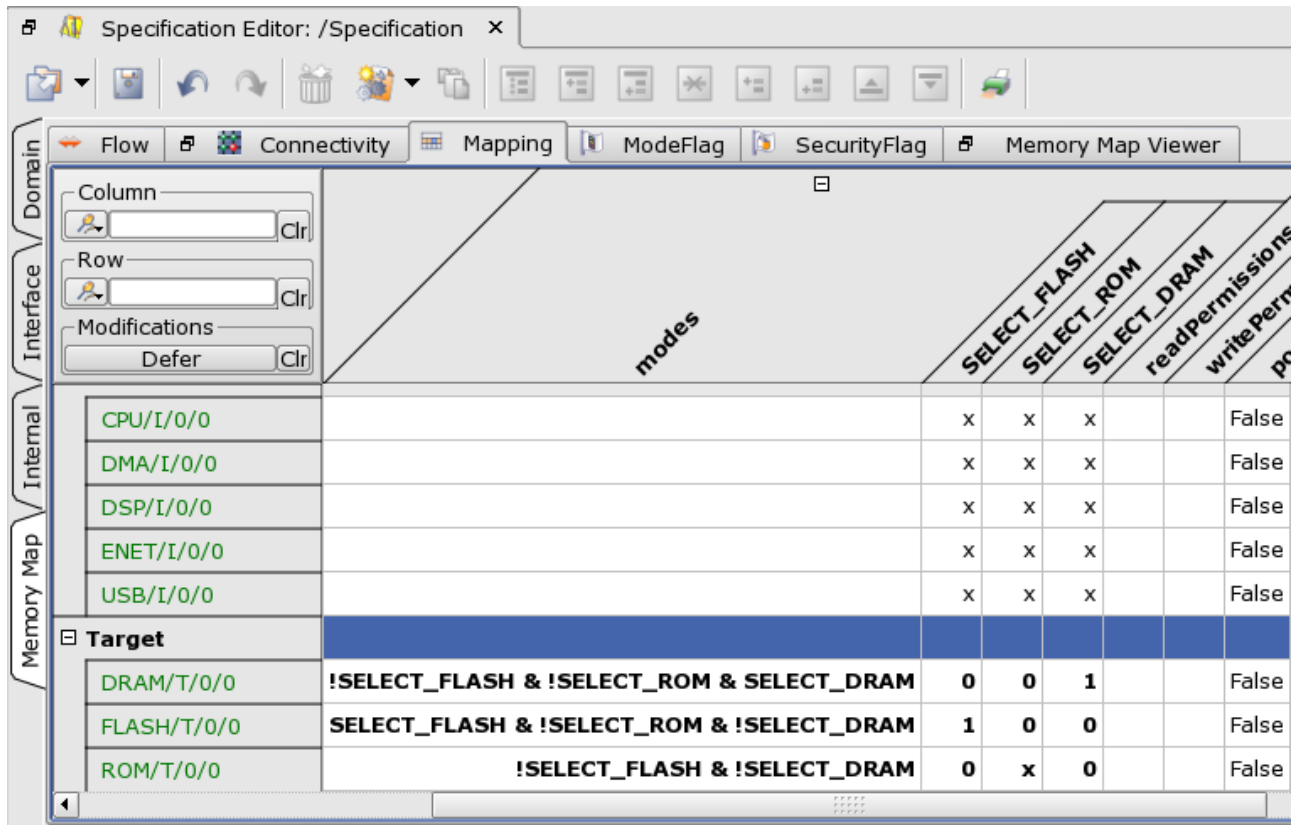
2 bits will be enough to cover the 3 boot modes we will define. Now we must introduce three mode-flags to alternatively select either the ROM or the FLASH or the DRAM

5. Introduce mode-flags
 - i. In the Memory Map major tab select the ModeFlag minor tab
 - ii. Right click and select NEW
 - a. Give it the name SELECT_ROM
 - b. Expand the modePortValues parameter group, click on the boot parameter and set it equal to 1
 - iii. Repeat step ii twice in order to create two more mode-flags, rename them respectively SELECT_FLASH and SELECT_DRAM, configure them as show in the picture below;



Now we have three different mode-flags that can be used to affect the memory mapping

6. Introduce mode-flags into the memory mapping
 - i. In the Mapping tab, look at the modes parameter sub-group
 - ii. In the row corresponding to the DRAM, set /SELECT_DRAM flag to 1, /SELECT_FLASH and /SELECT_ROM to 0
 - a. You can see the corresponding mode shows the boolean formula capturing this mode-flags combination
 - iii. Similarly, in the row corresponding to the FLASH, set /SELECT_FLASH flag to 1, /SELECT_DRAM and /SELECT_ROM to 0
 - iv. Finally, in the row corresponding to the ROM, set /SELECT_ROM flag to x, /SELECT_DRAM and /SELECT_FLASH to 0
 - a. This last setting will cause ROM enabling whenever SELECT_DRAM and SELECT_FLASH flags are both zero, in other words whenever boot pin is either 0 or 1 (look at ModeFlag tab in case of doubt)



Everything is complete and ready to simulate. As usual, the architecture and structure have been automatically updated and you do not need to perform any further operation; before proceeding with the simulation let's summarize what we've done.

To enable boot-modes you have to:

- Introduce mode-ports, to provide boot selectors to your NoC interface
- Introduce mode-flags, as combinations of different states of boot-mode input states
- Introduce mode-flags into the memory mapping, in order to associate memory ranges to combinations of mode-flags

To simulate your system please repeat the steps you did previously in previous labs; we will use ATE to export the design, test bench, scenarios, and other files necessary for automatic self test

7. With the Structure selected, choose **Tools -> Flex Verifier -> Export FlexVerifier-ATE** from the top menu bar.
 - i. Again, click **OK** to the export option we just defined, but put the output in a new directory. Click on the Create New Folder icon and name the new folder `ate`. Select the `ate` directory and click **Choose** to begin the export.
 - ii. When the export has finished, go back to your terminal window and `cd` to the `ate` directory. Use `ls -l` to see the test environment you just generated. To run a simple connectivity test, enter `./run.sh Connectivity` on the command line.
 - iii. When the simulation has completed, look at the file `Connectivity.report` in the `ate` directory. Note that all the connections specified in the connectivity table are tested for each initiator to each connected target.

```

Connectivity report - /home/carlo/base/customersView/base/customers/training/labs_v2/Lab3_v2/NSG/
File Edit Search Preferences Shell Macro Windows

# Diagnostic 'Connectivity' on NoC 'Structure' in mode 'Default' : OK

-- Connectivity :
//InitSocket [InitId] InitAddrRange -> TargSocket [TargId] - AccessType AccessLength
Changing mode to '!SELECT_DRAM & !SELECT_FLASH & !SELECT_ROM' :
CPU 0 [0x80000000:0x0] -> ROM - WR 8 bytes : OK
CPU 0 [0x80000000:0x0] -> ROM - RD 8 bytes : OK
CPU 0 [0x100000000:0x80000000] -> Expected Error - RD 8 bytes : OK
CPU 0 [0x100000000:0x80000000] -> Expected Error - WR 8 bytes : OK
Changing mode to '!SELECT_DRAM & SELECT_FLASH & !SELECT_ROM' :
CPU 0 [0x80000000:0x0] -> FLASH 0 - WR 8 bytes : OK
CPU 0 [0x80000000:0x0] -> FLASH 0 - RD 8 bytes : OK
CPU 0 [0x100000000:0x80000000] -> Expected Error - RD 8 bytes : OK
CPU 0 [0x100000000:0x80000000] -> Expected Error - WR 8 bytes : OK
Changing mode to 'SELECT_DRAM & !SELECT_FLASH & !SELECT_ROM' :
CPU 0 [0x80000000:0x0] -> DRAM 0 - WR 8 bytes : OK
CPU 0 [0x80000000:0x0] -> DRAM 0 - RD 8 bytes : OK
CPU 0 [0x100000000:0x80000000] -> Expected Error - RD 8 bytes : OK
CPU 0 [0x100000000:0x80000000] -> Expected Error - WR 8 bytes : OK
Changing mode to '!SELECT_DRAM & SELECT_FLASH & !SELECT_ROM' :
DMA 0 [0x80000000:0x0] -> FLASH 0 - WR 16 bytes : OK
DMA 0 [0x80000000:0x0] -> FLASH 0 - RD 16 bytes : OK
DMA 0 [0x100000000:0x80000000] -> Expected Error - RD 16 bytes : OK
DMA 0 [0x100000000:0x80000000] -> Expected Error - WR 16 bytes : OK
Changing mode to 'SELECT_DRAM & !SELECT_FLASH & !SELECT_ROM' :
DMA 0 [0x80000000:0x0] -> DRAM 0 - WR 16 bytes : OK
DMA 0 [0x80000000:0x0] -> DRAM 0 - RD 16 bytes : OK
DMA 0 [0x100000000:0x80000000] -> Expected Error - RD 16 bytes : OK
DMA 0 [0x100000000:0x80000000] -> Expected Error - WR 16 bytes : OK
Changing mode to '!SELECT_DRAM & !SELECT_FLASH & !SELECT_ROM' :
DSP 0 [0x80000000:0x0] -> ROM - WR 8 bytes : OK
DSP 0 [0x80000000:0x0] -> ROM - RD 8 bytes : OK
DSP 0 [0x100000000:0x80000000] -> Expected Error - RD 8 bytes : OK
DSP 0 [0x100000000:0x80000000] -> Expected Error - WR 8 bytes : OK
Changing mode to '!SELECT_DRAM & SELECT_FLASH & !SELECT_ROM' :
DSP 0 [0x80000000:0x0] -> FLASH 0 - WR 8 bytes : OK
DSP 0 [0x80000000:0x0] -> FLASH 0 - RD 8 bytes : OK
DSP 0 [0x100000000:0x80000000] -> Expected Error - RD 8 bytes : OK
DSP 0 [0x100000000:0x80000000] -> Expected Error - WR 8 bytes : OK
Changing mode to 'SELECT_DRAM & !SELECT_FLASH & !SELECT_ROM' :
DSP 0 [0x80000000:0x0] -> DRAM 0 - WR 8 bytes : OK
DSP 0 [0x80000000:0x0] -> DRAM 0 - RD 8 bytes : OK
DSP 0 [0x100000000:0x80000000] -> Expected Error - RD 8 bytes : OK
DSP 0 [0x100000000:0x80000000] -> Expected Error - WR 8 bytes : OK
Changing mode to '!SELECT_DRAM & !SELECT_FLASH & !SELECT_ROM' :
ENET 0 [0x80000000:0x0] -> ROM - WR 4 bytes : OK
ENET 0 [0x80000000:0x0] -> ROM - RD 4 bytes : OK
ENET 0 [0x100000000:0x80000000] -> Expected Error - RD 4 bytes : OK
ENET 0 [0x100000000:0x80000000] -> Expected Error - WR 4 bytes : OK
Changing mode to '!SELECT_DRAM & SELECT_FLASH & !SELECT_ROM' :
ENET 0 [0x80000000:0x0] -> FLASH 0 - WR 4 bytes : OK
ENET 0 [0x80000000:0x0] -> FLASH 0 - RD 4 bytes : OK
ENET 0 [0x100000000:0x80000000] -> Expected Error - RD 4 bytes : OK
ENET 0 [0x100000000:0x80000000] -> Expected Error - WR 4 bytes : OK
Changing mode to 'SELECT_DRAM & !SELECT_FLASH & !SELECT_ROM' :
ENET 0 [0x80000000:0x0] -> DRAM 0 - WR 4 bytes : OK
ENET 0 [0x80000000:0x0] -> DRAM 0 - RD 4 bytes : OK
ENET 0 [0x100000000:0x80000000] -> Expected Error - RD 4 bytes : OK
ENET 0 [0x100000000:0x80000000] -> Expected Error - WR 4 bytes : OK
Changing mode to 'SELECT_DRAM & SELECT_FLASH & !SELECT_ROM' :
USB 0 [0x80000000:0x0] -> FLASH 0 - WR 8 bytes : OK
USB 0 [0x80000000:0x0] -> FLASH 0 - RD 8 bytes : OK
USB 0 [0x100000000:0x80000000] -> Expected Error - RD 8 bytes : OK
USB 0 [0x100000000:0x80000000] -> Expected Error - WR 8 bytes : OK
Changing mode to 'SELECT_DRAM & !SELECT_FLASH & !SELECT_ROM' :
USB 0 [0x80000000:0x0] -> DRAM 0 - WR 8 bytes : OK
USB 0 [0x80000000:0x0] -> DRAM 0 - RD 8 bytes : OK
USB 0 [0x100000000:0x80000000] -> Expected Error - RD 8 bytes : OK
USB 0 [0x100000000:0x80000000] -> Expected Error - WR 8 bytes : OK

```