bytestream Example

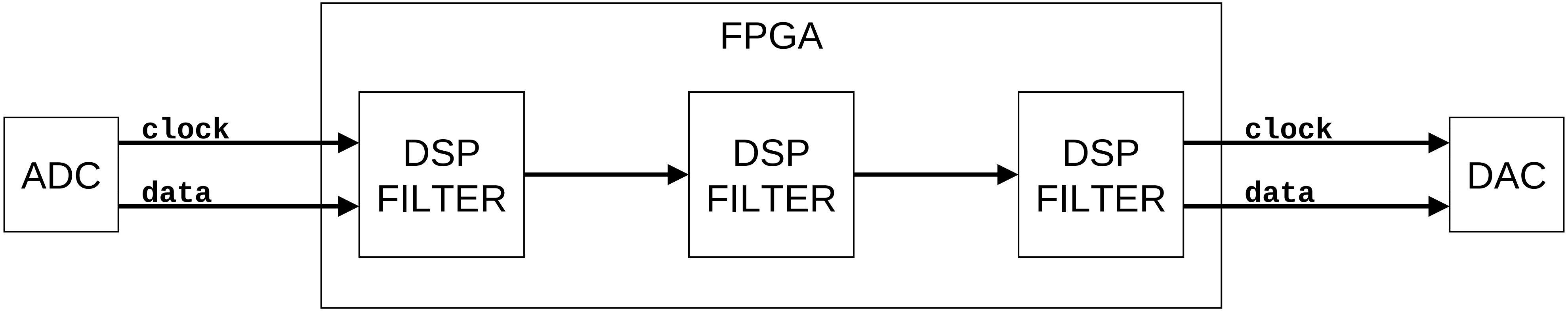
Using System Console to test and analyze streaming peripherals.

Objective

- These slides attempt to illustrate how quickly and easily the System Console environment can be assembled into a test platform for analysis and testing of streaming components.
- The test platform for all of these examples is the developed on the NEEK development board, however, they could easily be ported to any other board.

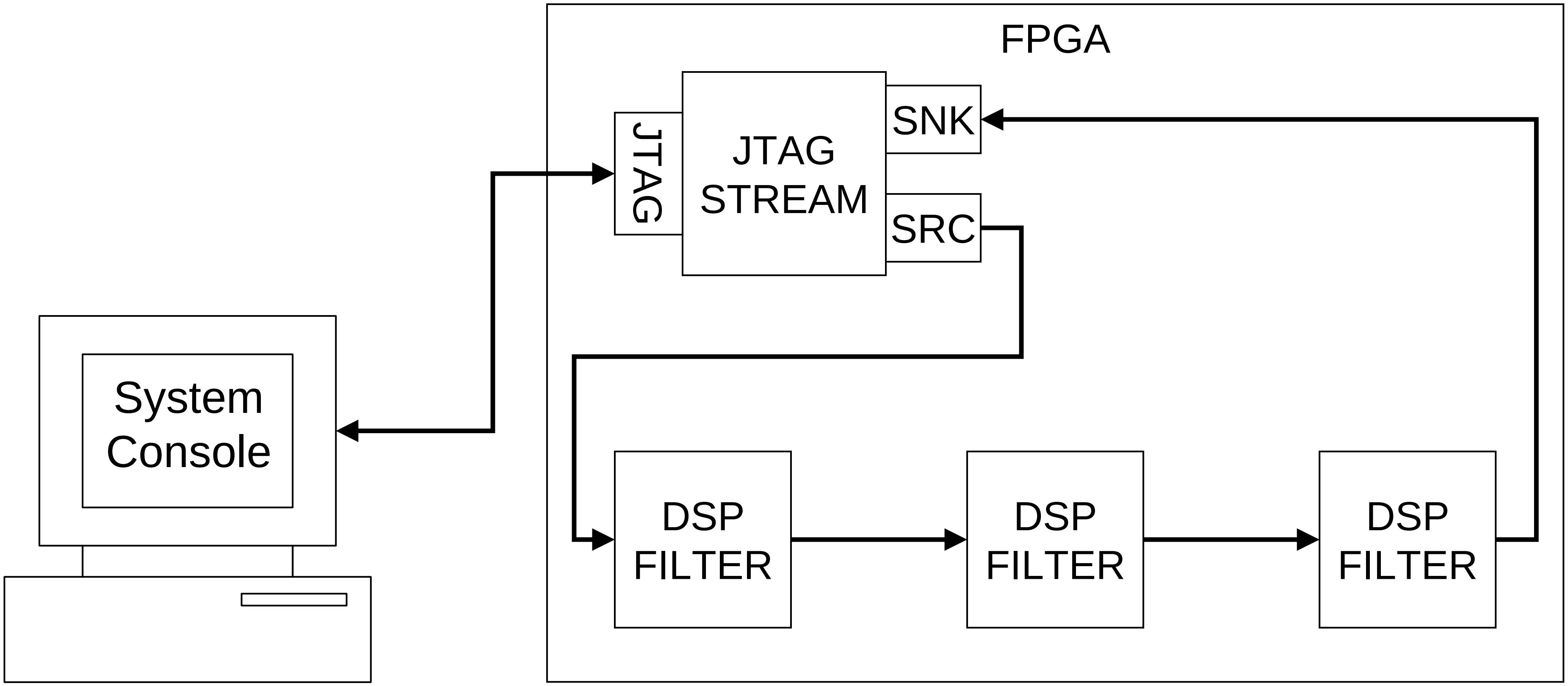
A Problem

- How do you test and validate the functionality of DSP filter algorithms in a typical design?
  - At speed…
  - Inside the FPGA…



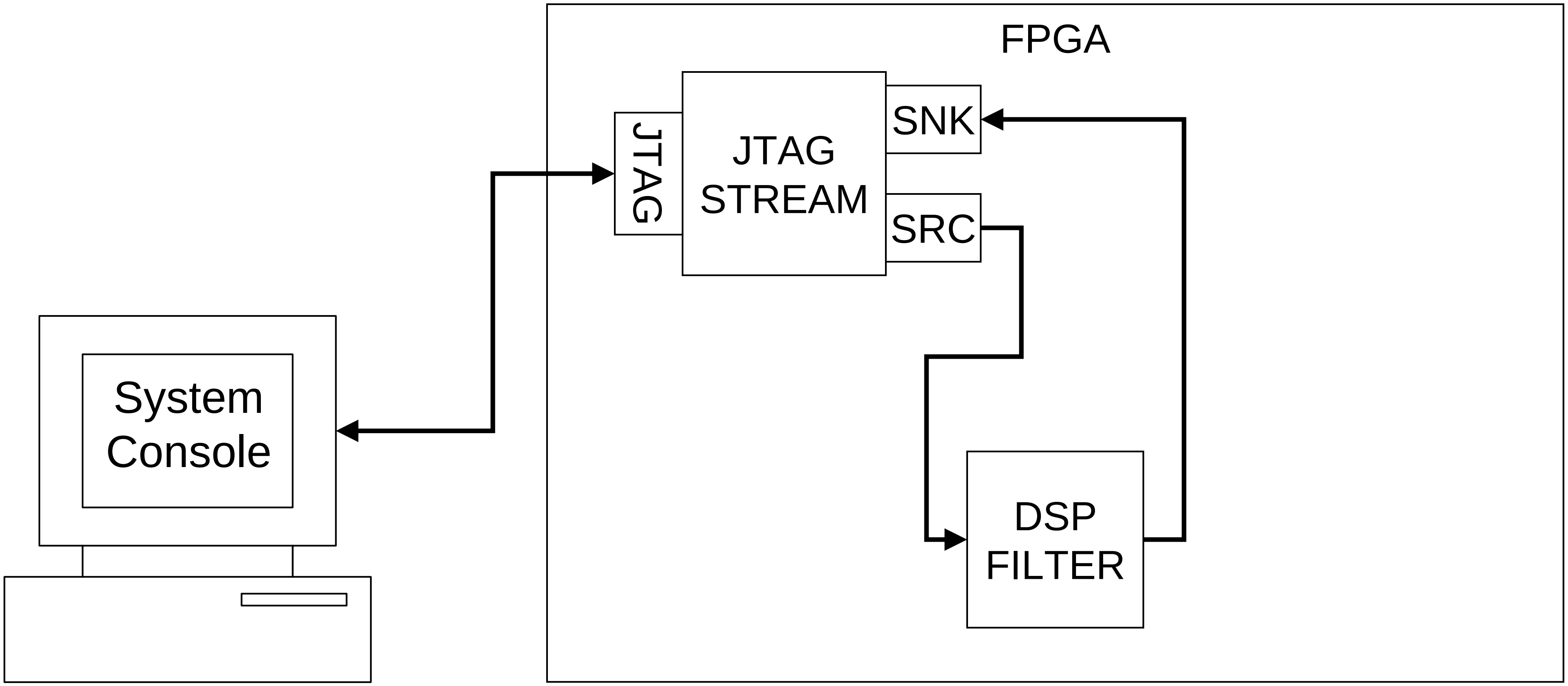
A Solution

- Use System Console to source data into your design and sink results from it.
- Please find documentation on the JTAG Stream interface shown below in the Quartus II Version 8.0 Handbook, Volume 5: Embedded Peripherals, Chapter 10.
- Please find documentation on System Console in the System Console User Guide.



A Solution

- This model could even be used to perform unit testing of individual blocks.

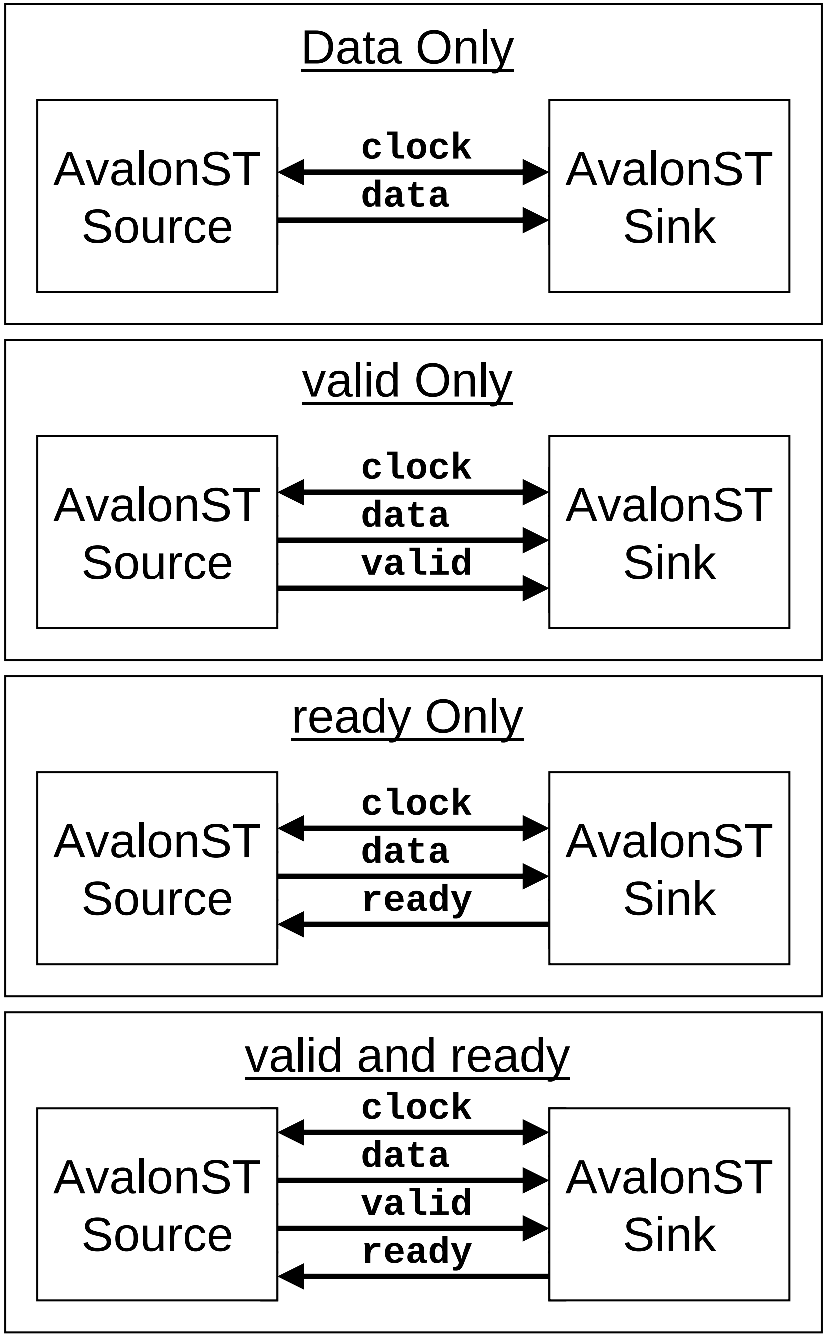


The Avalon ST Interface

A data streaming interface protocol supported within SOPC Builder.

Avalon ST Basics

- The simplest Avalon ST interface consists of a data path transferring data from the source interface to the sink interface, synchronous to the clock.
- You can add an enable strobe to the interface called "valid". The valid strobe is driven by the source to the sink to indicate that the source is driving the next data word onto the interface. If a source and sink only share a valid strobe in common, then the valid strobe acts as an enable, and data is clocked from the source to the sink on clocks that have valid asserted. The valid strobe operates like a forwarding enable strobe into the data path.
- You could alternatively add and an enable strobe to the interface called "ready". The ready strobe is driven by the sink to the source to indicate the sink is accepting the next data word from the interface. If a source and sink only share a ready strobe in common, then the ready strobe acts as an enable and data is clocked from the source to the sink on clocks that have ready asserted. The ready strobe operates like a back pressuring enable strobe back thru the data path.
- You can add both a "valid" and a "ready" strobe to the interface as well. The ready strobe can have an attribute called ready latency which can delay the data transfer clock by a specified latency, and the coordination of the valid and ready strobes must then account for this ready latency, but this goes beyond the scope of this discussion.

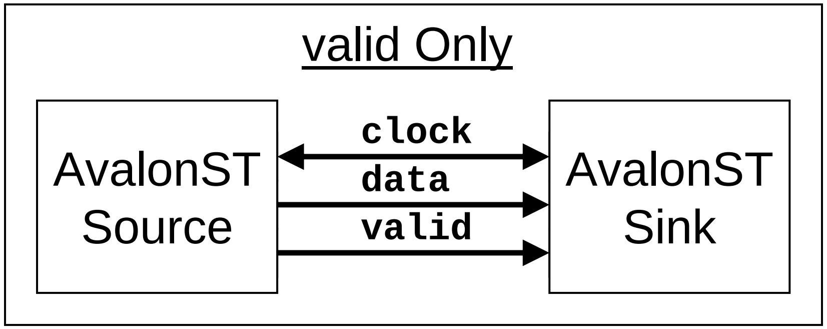


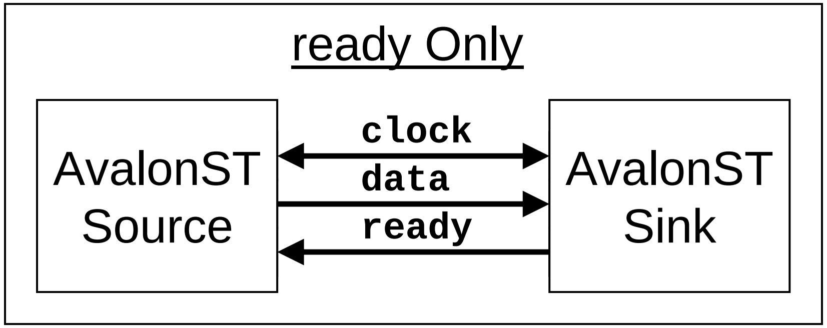
Avalon ST Basics

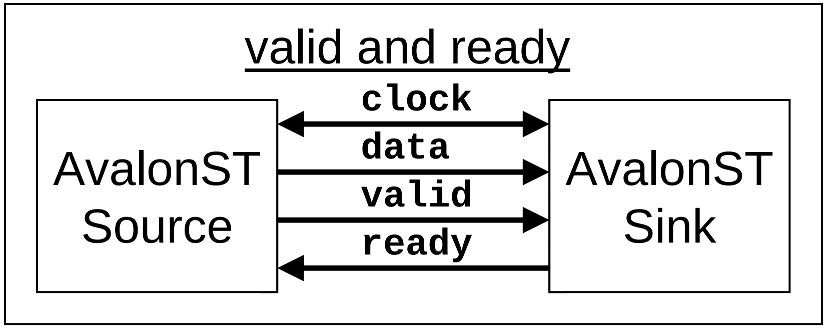
- Avalon ST interfaces can also add more complexity, like channel information, error information, etc. The complete scope of the Avalon ST interface is documented in the Avalon Interface Specification.

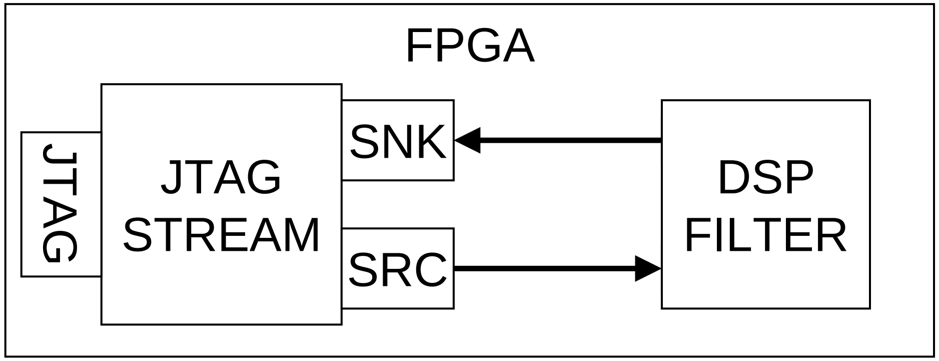
Avalon ST and System Console

- If your DSP filter block has some form of Avalon ST enable strobe on it as shown on the right, then you could connect it directly to the JTAG Stream interface that connects to the system console environment, as shown below.
- Since the JTAG Stream component has valid and ready strobes on it to pace the data thru the interface, streaming data from the relatively slow JTAG interface thru the DSP filter block and back can by totally synchronized by the ready and valid strobes.



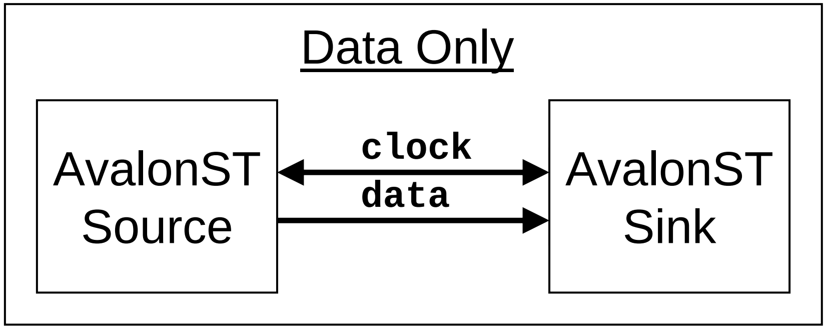






Avalon ST and this example.

- For purposes of this example, we will focus on the simplest of these Avalon ST interfaces, data only, so there will be no ready or valid strobes to provide enable queues to the interfaces.
- This is a perfectly valid way to use the Avalon ST interface. The role of ready and valid in this case are described in the Avalon Interface Specification as follows:
  - Ready – Sources without a ready input cannot be backpressured, and sinks without a ready output never need to backpressure.
  - Valid – Sources without a valid output implicitly provide valid data on every cycle that they're not being backpressured, and sinks without a valid input expect valid data on every cycle that they are not backpressuring.
- Though this is a very simple streaming interface consisting of data, synchronized to a clock, it also presents a challenge to interface with over the relatively slow JTAG interface, since there are no enable signals available to pace the data thru the components.

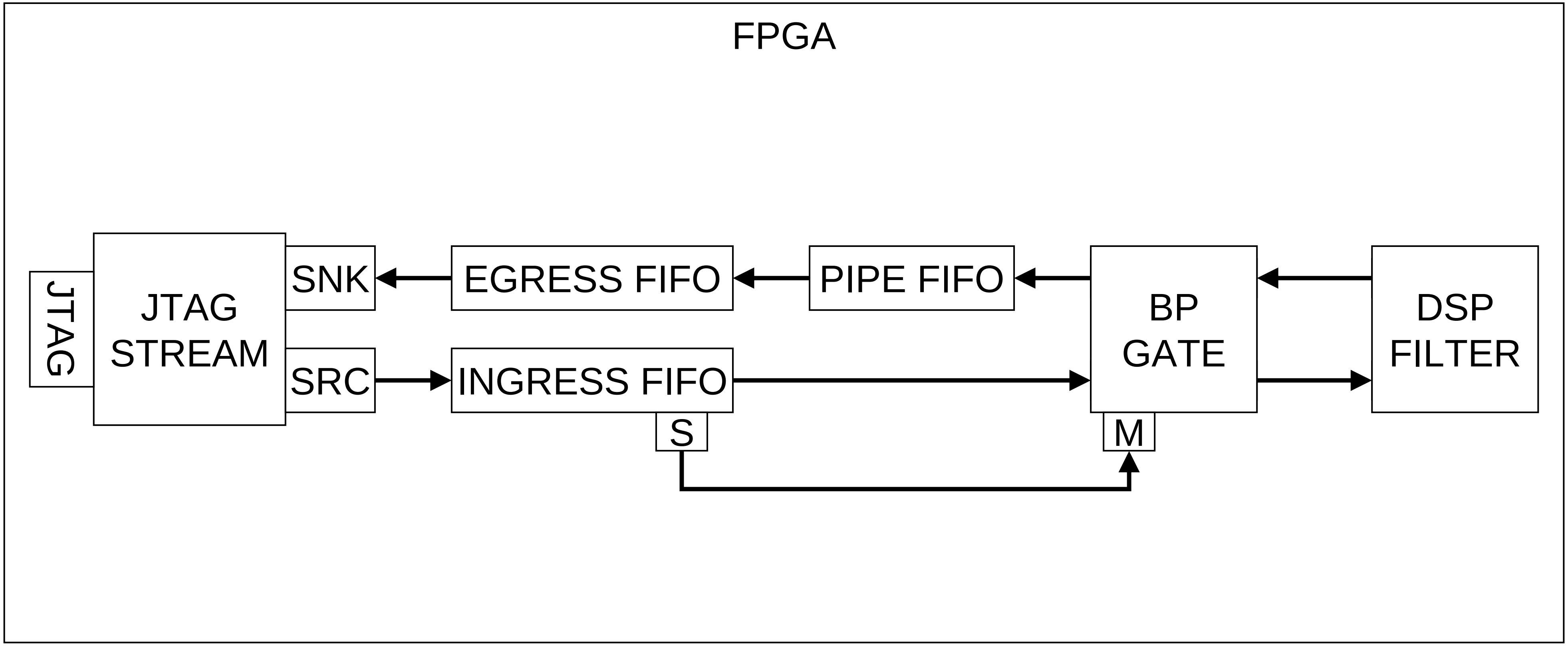


Synchronizing the Stream

For components with no synchronization strobes.

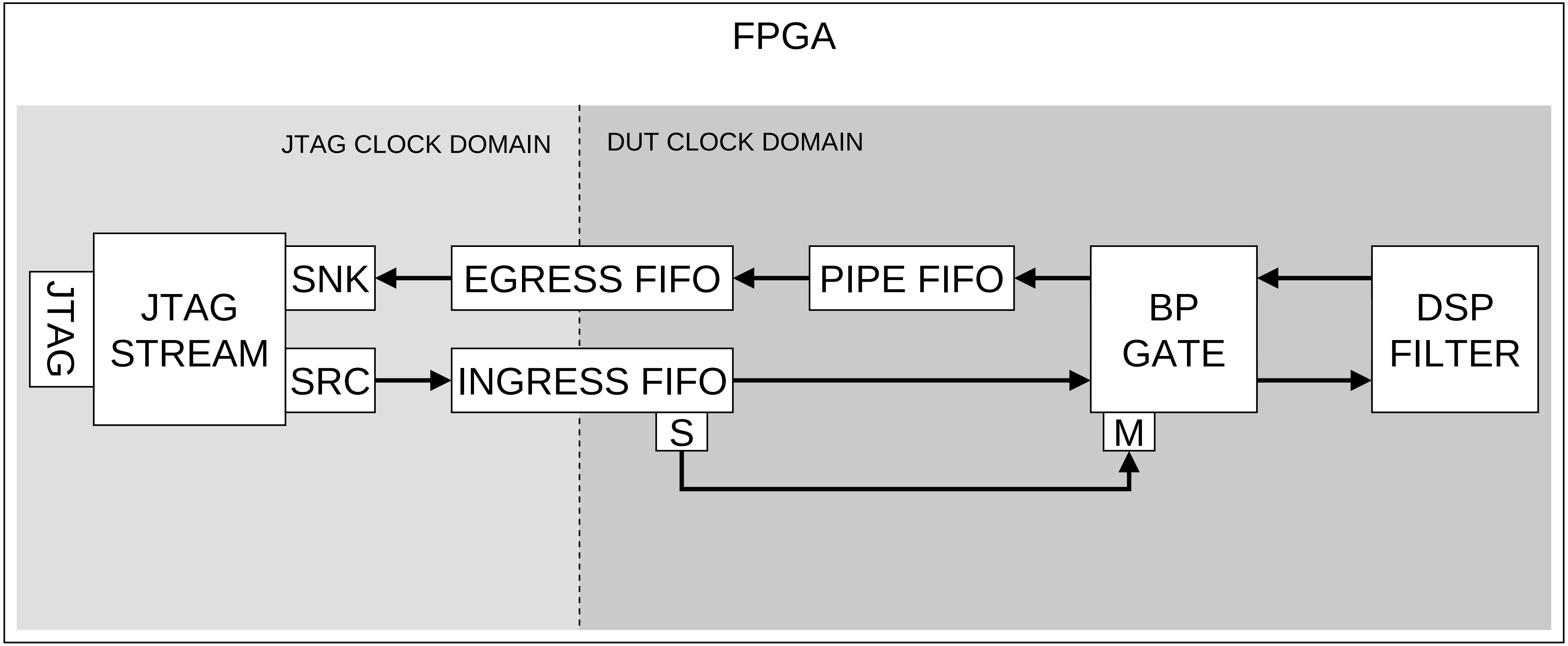
Creating a Synchronization GATE

- In the simplest case where the DSP filters only uses clock and data, there’s a bit of infrastructure needed to facilitate proper operation at speed within the FPGA.
- A back pressure “gating” component is inserted as shown below to allow an ingress FIFO to accumulate a specified amount of data. An Avalon Master polls the Avalon Slave on the ingress FIFO to detect the fill level.
- Once the fill level is achieved, the GATE opens and dumps the data thru the DUT at the clock rate of the DUT.
- The egress FIFO is a bit bigger than the ingress data such that the pipeline depth of the DUT is accounted for, and the GATE component also ensures that the egress FIFO is only filled with data after the ingress GATE has opened.
- System Console may now be used to fill the ingress FIFO with input and extract the results from the egress FIFO.



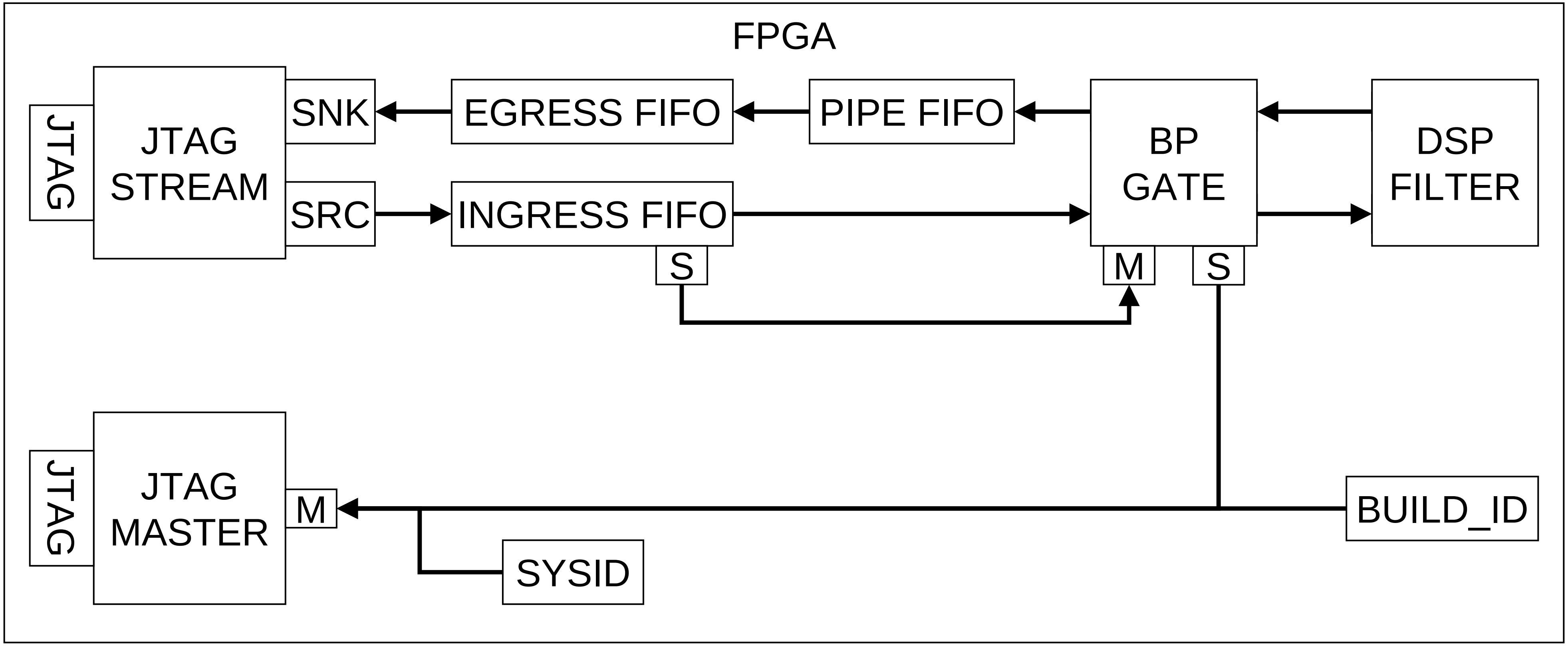
Running at speed.

- In this example the clock domain of the JTAG system console infrastructure is placed in a slower clock domain, and the ingress and egress FIFOs are created from DCFIFOs which accomplish the clock domain crossing into the DUT clock domain.



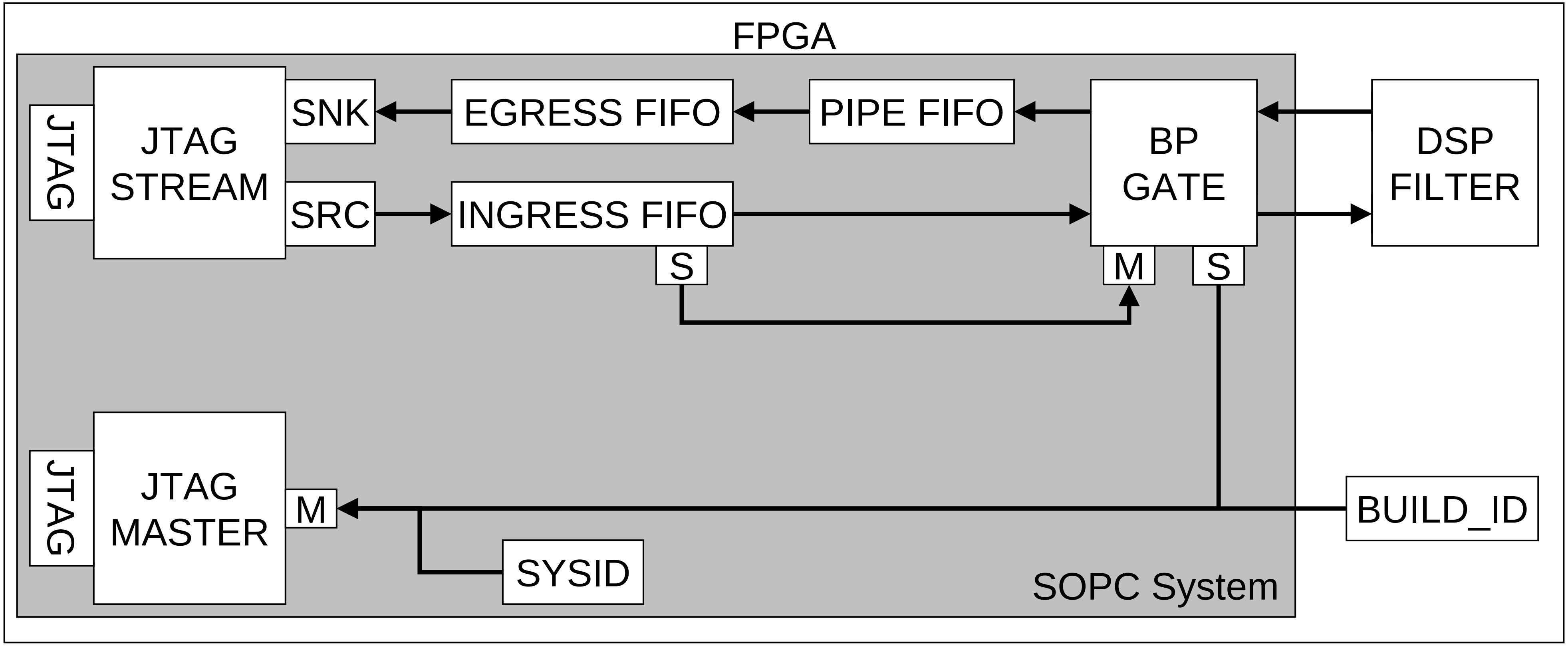
A more complete solution

- We can create a small SOPC system like this to provide the streaming channel as well as system identification and build identification thru a JTAG Master interface.
- The JTAG Master is also given access to a slave on the GATE component which allows system console to adjust the fill trigger level for the GATE component.
- Please find documentation on the JTAG Master interface shown below in the Quartus II Version 8.0 Handbook, Volume 5: Embedded Peripherals, Chapter 12.
- At this point the user can simply stitch their build ID and DUT blocks into the SOPC system in a top level module. Optionally, the user could build these blocks right into the SOPC system itself.



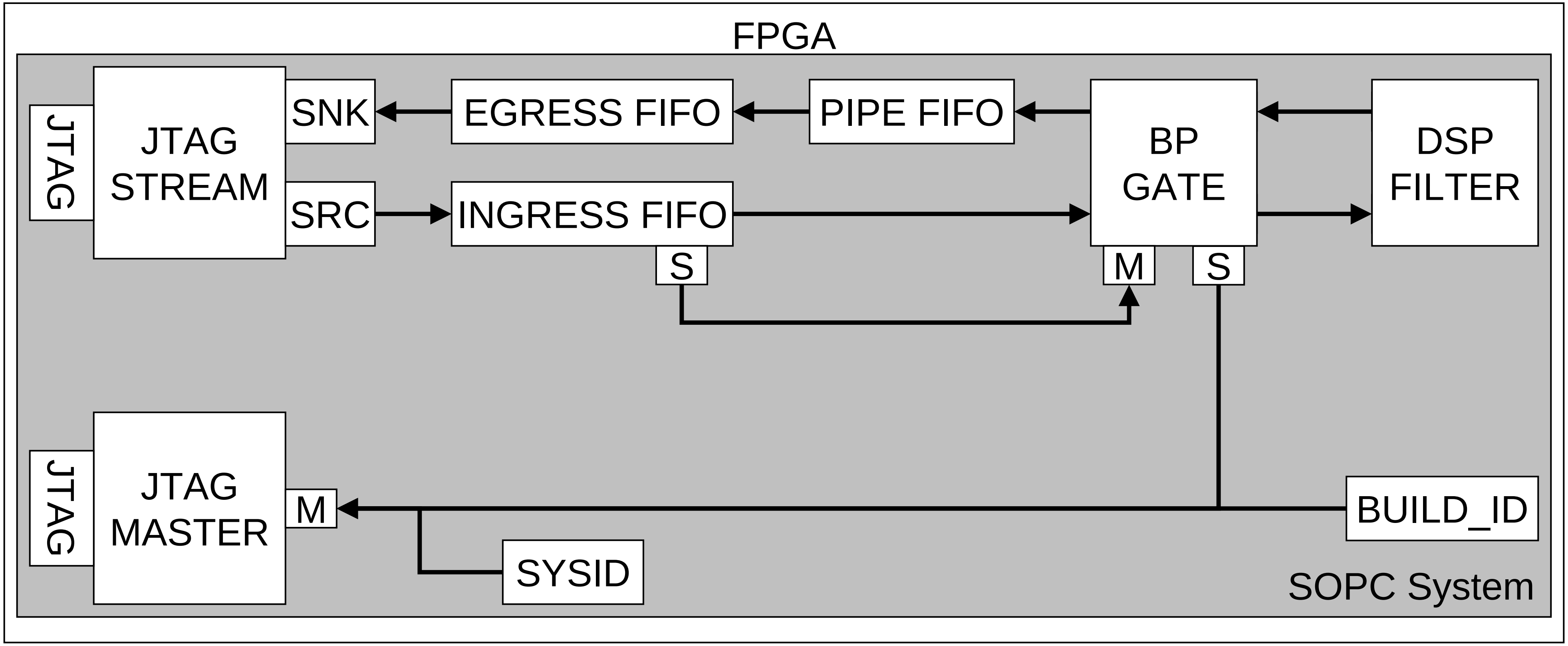
Separated System Approach

- This picture illustrates the approach of keeping the users modules external to the SOPC system.
- In this case the user would create the SOPC system shown below and then manually stitch their components into the top level HDL module that SOPC Builder generates.



Integrated System Approach

- This picture illustrates the approach of placing the users modules inside the SOPC system.
- In this case the user would create the SOPC system shown below and import their HDL component into SOPC Builder and place them directly into the system.
- There would be no manual manipulation of the top level modules in this case.



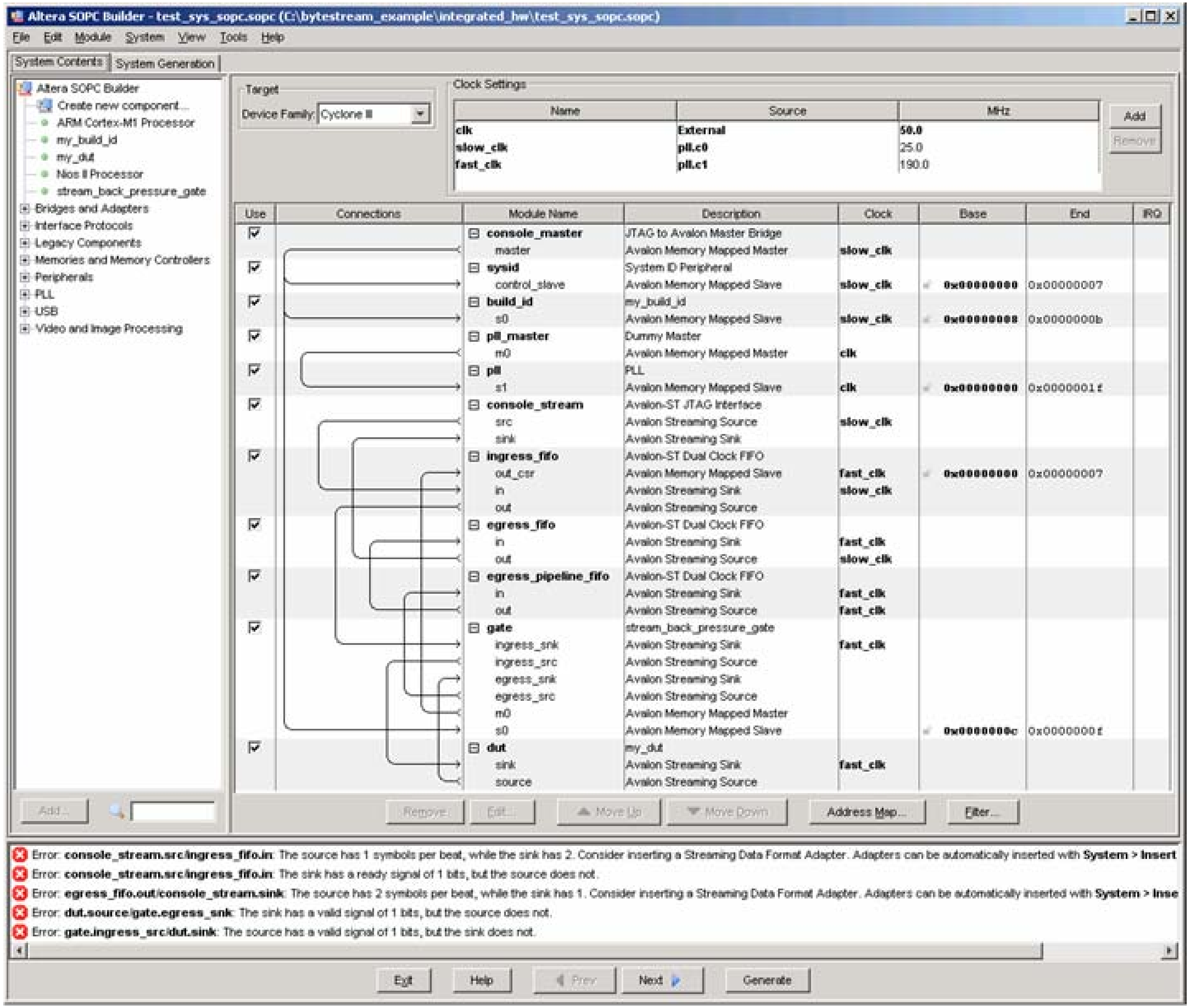
Integrated vs Separated SOPC Approach

- There is no advantage to either design flow from an SOPC Builder or Altera tools perspective, although the integrated approach requires potentially less manual intervention by humans.
- Each of the examples should produce exactly the same result in hardware, assuming that the manual stitching is done properly in the separated approach.
- In each flow, during a component's development cycle, you could edit the source code and recompile the design without requiring SOPC Builder generation as long as you are not changing the SOPC interface ports, masters, slaves, sources and sinks.
- In either case, if you adjust any of the SOPC interface ports, you will be required to regenerate the SOPC system to account for the alterations in the new revision.
- Generally speaking, personal preference or user design flow requirement will generally determine which approach is best for any given user environment.

Creating the System in SOPC Builder

The integrated approach.

This is the system we want to create.



Device Family: Cyclone II

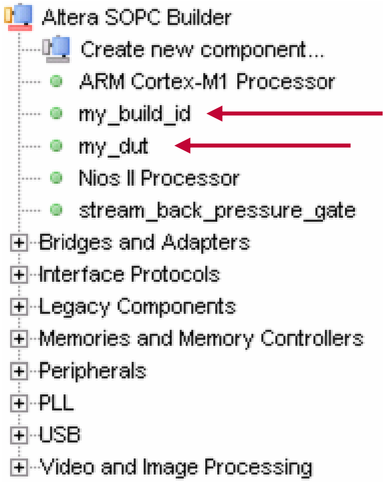
Name	Source	MHz
clk	**External**	**50.0**
slow_clk	**pll.c0**	25.0
fast_clk	**pll.c1**	190.0

Use	Module Name	Description	Clock	Base	End	IRQ
☑	⊟ **console_master**	JTAG to Avalon Master Bridge				
	master	Avalon Memory Mapped Master	**slow_clk**			
☑	⊟ **sysid**	System ID Peripheral				
	control_slave	Avalon Memory Mapped Slave	**slow_clk**	**0x00000000**	0x00000007	
☑	⊟ **build_id**	my_build_id				
	s0	Avalon Memory Mapped Slave	**slow_clk**	**0x00000008**	0x0000000b	
☑	⊟ **pll_master**	Dummy Master				
	m0	Avalon Memory Mapped Master	**clk**			
☑	⊟ **pll**	PLL				
	s1	Avalon Memory Mapped Slave	**clk**	**0x00000000**	0x0000001f	
☑	⊟ **console_stream**	Avalon-ST JTAG Interface				
	src	Avalon Streaming Source	**slow_clk**			
	sink	Avalon Streaming Sink				
☑	⊟ **ingress_fifo**	Avalon-ST Dual Clock FIFO				
	out_csr	Avalon Memory Mapped Slave	**fast_clk**	**0x00000000**	0x00000007	
	in	Avalon Streaming Sink	**slow_clk**			
	out	Avalon Streaming Source				
☑	⊟ **egress_fifo**	Avalon-ST Dual Clock FIFO				
	in	Avalon Streaming Sink	**fast_clk**			
	out	Avalon Streaming Source	**slow_clk**			
☑	⊟ **egress_pipeline_fifo**	Avalon-ST Dual Clock FIFO				
	in	Avalon Streaming Sink	**fast_clk**			
	out	Avalon Streaming Source	**fast_clk**			
☑	⊟ **gate**	stream_back_pressure_gate				
	ingress_snk	Avalon Streaming Sink	**fast_clk**			
	ingress_src	Avalon Streaming Source				
	egress_snk	Avalon Streaming Sink				
	egress_src	Avalon Streaming Source				
	m0	Avalon Memory Mapped Master				
	s0	Avalon Memory Mapped Slave		**0x0000000c**	0x0000000f	
☑	⊟ **dut**	my_dut				
	sink	Avalon Streaming Sink	**fast_clk**			
	source	Avalon Streaming Source				

```
Error: console_stream.src/ingress_fifo.in: The source has 1 symbols per beat, while the sink has 2. Consider inserting a Streaming Data Format Adapter. Adapters can be automatically inserted with System > Insert
Error: console_stream.src/ingress_fifo.in: The sink has a ready signal of 1 bits, but the source does not.
Error: egress_fifo.out/console_stream.sink: The source has 2 symbols per beat, while the sink has 1. Consider inserting a Streaming Data Format Adapter. Adapters can be automatically inserted with System > Inse
Error: dut.source/gate.egress_snk: The sink has a valid signal of 1 bits, but the source does not.
Error: gate.ingress_src/dut.sink: The source has a valid signal of 1 bits, but the sink does not.
```

Our Custom Components

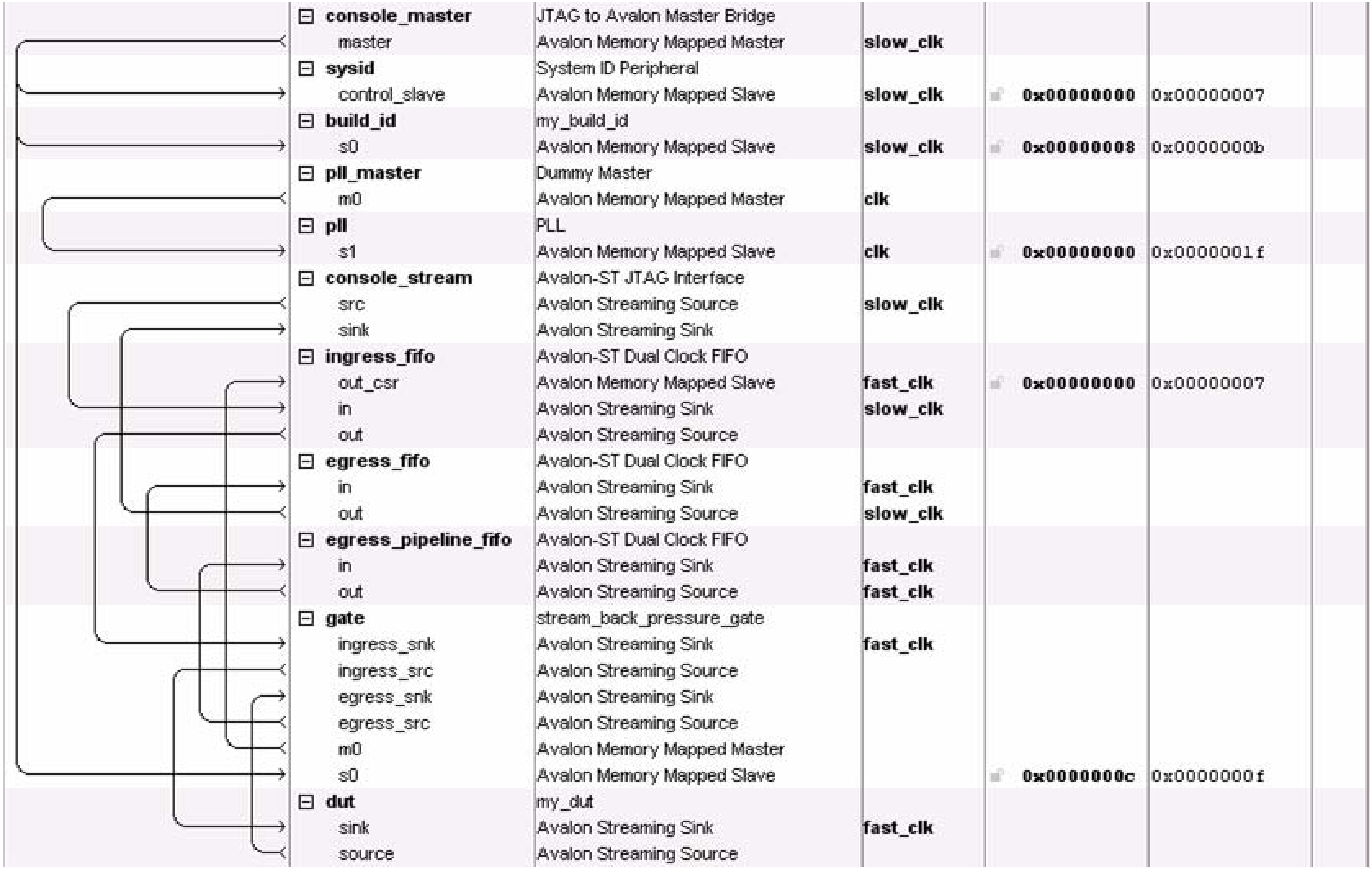
- In this example our DUT component will simply invert the data that it receives on the input stream and pass it to the output stream.
- The streaming component that we want to test, my_dut, as well as the my_build_id peripheral are custom HDL components that we should import with the SOPC Builder Component Editor to get started.
- This process is described in the Quartus II Version 8.0 Handbook, Volume 4: SOPC Builder.
- The stream_back_pressure_gate peripheral is already created for us to use as the GATE peripheral shown in the previous block diagrams.



Altera SOPC Builder
Create new component...
ARM Cortex-M1 Processor
my_build_id
my_dut
Nios II Processor
stream_back_pressure_gate
Bridges and Adapters
Interface Protocols
Legacy Components
Memories and Memory Controllers
Peripherals
PLL
USB
Video and Image Processing

The integrated system.

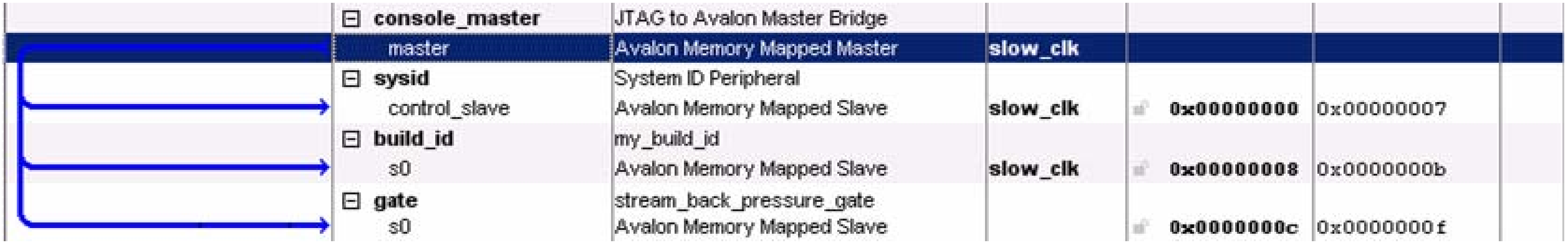
- The following slides will walk thru how this system is put together and why it is put together this way.



⊟ **console_master**	JTAG to Avalon Master Bridge			
master	Avalon Memory Mapped Master	**slow_clk**		
⊟ **sysid**	System ID Peripheral			
control_slave	Avalon Memory Mapped Slave	**slow_clk**	**0x00000000**	0x00000007
⊟ **build_id**	my_build_id			
s0	Avalon Memory Mapped Slave	**slow_clk**	**0x00000008**	0x0000000b
⊟ **pll_master**	Dummy Master			
m0	Avalon Memory Mapped Master	**clk**		
⊟ **pll**	PLL			
s1	Avalon Memory Mapped Slave	**clk**	**0x00000000**	0x0000001f
⊟ **console_stream**	Avalon-ST JTAG Interface			
src	Avalon Streaming Source	**slow_clk**		
sink	Avalon Streaming Sink			
⊟ **ingress_fifo**	Avalon-ST Dual Clock FIFO			
out_csr	Avalon Memory Mapped Slave	**fast_clk**	**0x00000000**	0x00000007
in	Avalon Streaming Sink	**slow_clk**		
out	Avalon Streaming Source			
⊟ **egress_fifo**	Avalon-ST Dual Clock FIFO			
in	Avalon Streaming Sink	**fast_clk**		
out	Avalon Streaming Source	**slow_clk**		
⊟ **egress_pipeline_fifo**	Avalon-ST Dual Clock FIFO			
in	Avalon Streaming Sink	**fast_clk**		
out	Avalon Streaming Source	**fast_clk**		
⊟ **gate**	stream_back_pressure_gate			
ingress_snk	Avalon Streaming Sink	**fast_clk**		
ingress_src	Avalon Streaming Source			
egress_snk	Avalon Streaming Sink			
egress_src	Avalon Streaming Source			
m0	Avalon Memory Mapped Master			
s0	Avalon Memory Mapped Slave		**0x0000000c**	0x0000000f
⊟ **dut**	my_dut			
sink	Avalon Streaming Sink	**fast_clk**		
source	Avalon Streaming Source			

The memory mapped components.

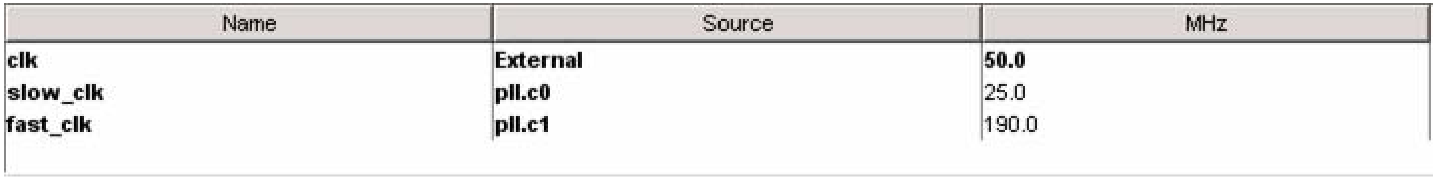
- In order to facilitate system identification as well as run time manipulation of the gate component a small memory mapped section is created in this design to allow the System Console to peek and poke thru the JTAG Master into the memory mapped slaves.
- The sysid peripheral provides a 64 bit system identification value that is updated each time the SOPC system is generated. This can be validated at run time from the System Console environment and provides a useful debug assistance by knowing what revision of a hardware design is programmed into the FPGA.
- The build_id peripheral is a user supplied custom component that has the same motivation of the sysid peripheral but since it is under user control, the user may manipulate the value of this ID at will, regardless of the state of an actual SOPC system regeneration. For instance if the user wishes to make source code changes to their DUT peripheral, but does not require SOPC system regeneration, they could modify the value of this build ID to account for the updated revision in their hardware.
- The gate peripheral provides a slave interface that allows us to query and change the FIFO fill trigger level that the gate component uses to trigger the gate opening.



⊟ **console_master**	JTAG to Avalon Master Bridge			
master	Avalon Memory Mapped Master	**slow_clk**		
⊟ **sysid**	System ID Peripheral			
control_slave	Avalon Memory Mapped Slave	**slow_clk**	**0x00000000**	0x00000007
⊟ **build_id**	my_build_id			
s0	Avalon Memory Mapped Slave	**slow_clk**	**0x00000008**	0x0000000b
⊟ **gate**	stream_back_pressure_gate			
s0	Avalon Memory Mapped Slave		**0x0000000c**	0x0000000f

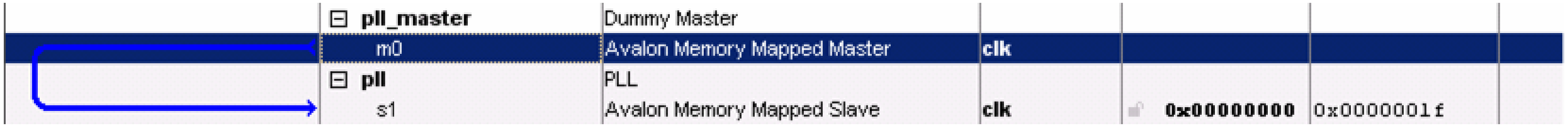
System Clocks

- We place a PLL into the system to create the clocks that we desire.
- The PLL takes the 50MHz clock in from the onboard oscillator, “clk”, and generates a “slow_clk” of 25MHz for our System Console infrastructure as well as a “fast_clk” of 190MHz for our DUT infrastructure.
- In the 3C25C8 device on the NEEK board, the DCFIFOs seem to be the limiting factor for Fmax of around 195MHz, so that drove the 190MHz selection for the DUT clock domain.



Name	Source	MHz
clk	**External**	**50.0**
slow_clk	**pll.c0**	25.0
fast_clk	**pll.c1**	190.0

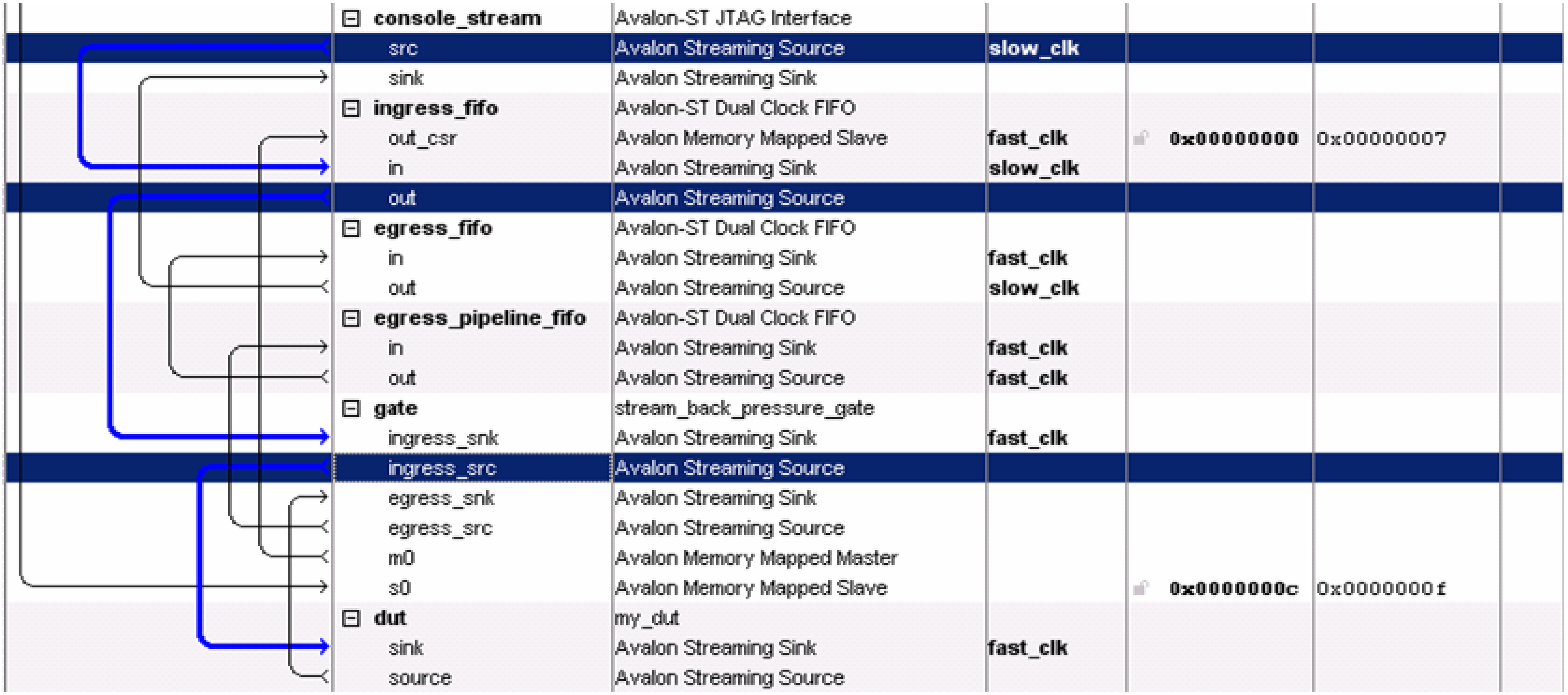
- We place a “Dummy Master” in the design and connect it to the PLL slave interface for two reasons.
- First, there is nothing practical that can be done with this slave interface, so we simply need a master to terminate it with.
- Second, with a dummy master we can place it in the same clock domain as the PLL slave and thus save an auto generated clock adapter from being created for us.



⊟ **pll_master**	Dummy Master			
m0	Avalon Memory Mapped Master	**clk**		
⊟ **pll**	PLL			
s1	Avalon Memory Mapped Slave	**clk**	**0x00000000**	0x0000001f

The ingress stream path.

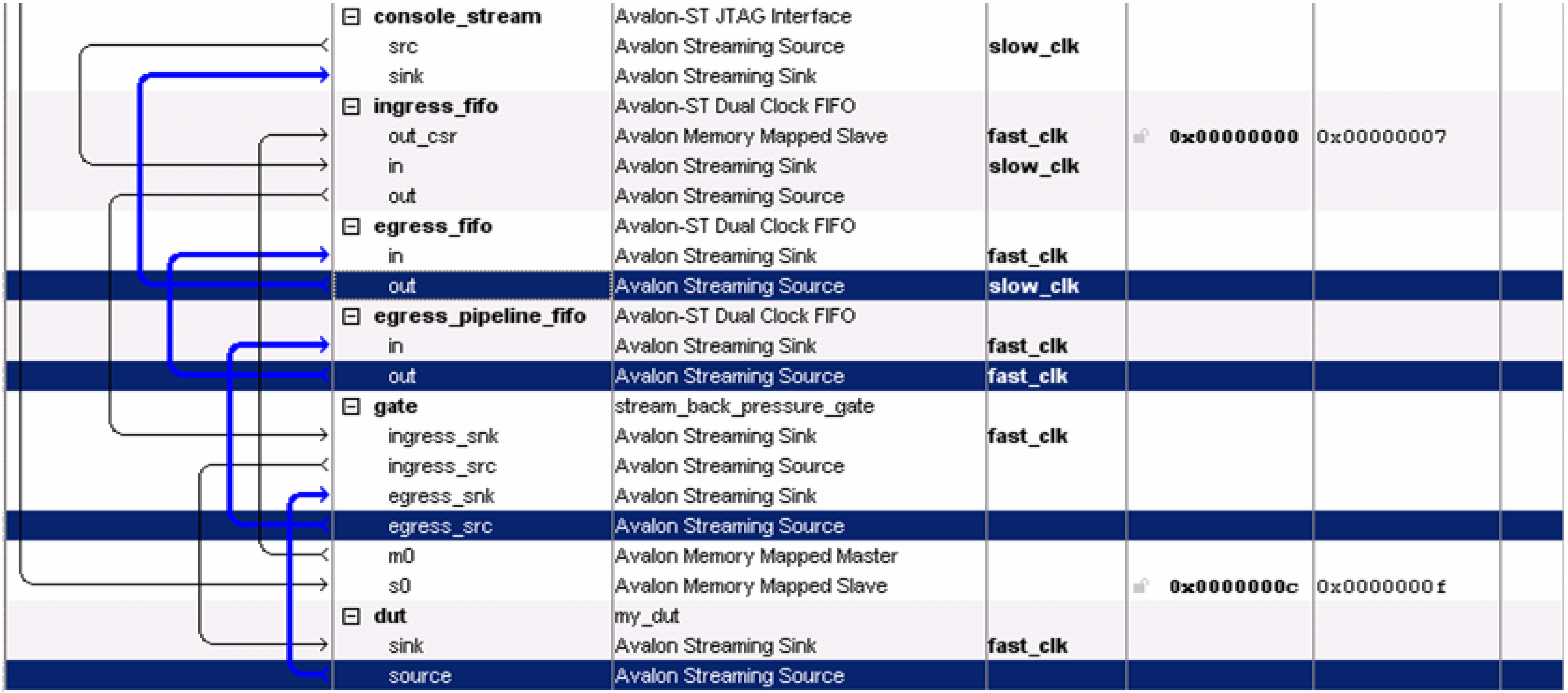
- You can see the ingress stream path in the picture below. All of the source interfaces are highlighted which in turn highlights a blue path to their respective sink counterparts.
- The stream begins at the console_stream peripheral as a 1 symbol per beat, 8 bit per symbol source interface with a valid strobe.
- The stream then enters the ingress_fifo peripheral sink, which is a 2 symbol per beat, 8 bit per symbol interface with ready and valid strobes.
- The ingress_fifo source then feeds the sink of the gate peripheral.
- The gate source then feeds the DUT sink interface which has no valid or ready strobes, but accepts 2 symbols per beat, 8 bit per symbol.



⊟ **console_stream**	Avalon-ST JTAG Interface			
src	Avalon Streaming Source	**slow_clk**		
sink	Avalon Streaming Sink			
⊟ **ingress_fifo**	Avalon-ST Dual Clock FIFO			
out_csr	Avalon Memory Mapped Slave	**fast_clk**	**0x00000000**	0x00000007
in	Avalon Streaming Sink	**slow_clk**		
out	Avalon Streaming Source			
⊟ **egress_fifo**	Avalon-ST Dual Clock FIFO			
in	Avalon Streaming Sink	**fast_clk**		
out	Avalon Streaming Source	**slow_clk**		
⊟ **egress_pipeline_fifo**	Avalon-ST Dual Clock FIFO			
in	Avalon Streaming Sink	**fast_clk**		
out	Avalon Streaming Source	**fast_clk**		
⊟ **gate**	stream_back_pressure_gate			
ingress_snk	Avalon Streaming Sink	**fast_clk**		
ingress_src	Avalon Streaming Source			
egress_snk	Avalon Streaming Sink			
egress_src	Avalon Streaming Source			
m0	Avalon Memory Mapped Master			
s0	Avalon Memory Mapped Slave		**0x0000000c**	0x0000000f
⊟ **dut**	my_dut			
sink	Avalon Streaming Sink	**fast_clk**		
source	Avalon Streaming Source			

The egress stream path.

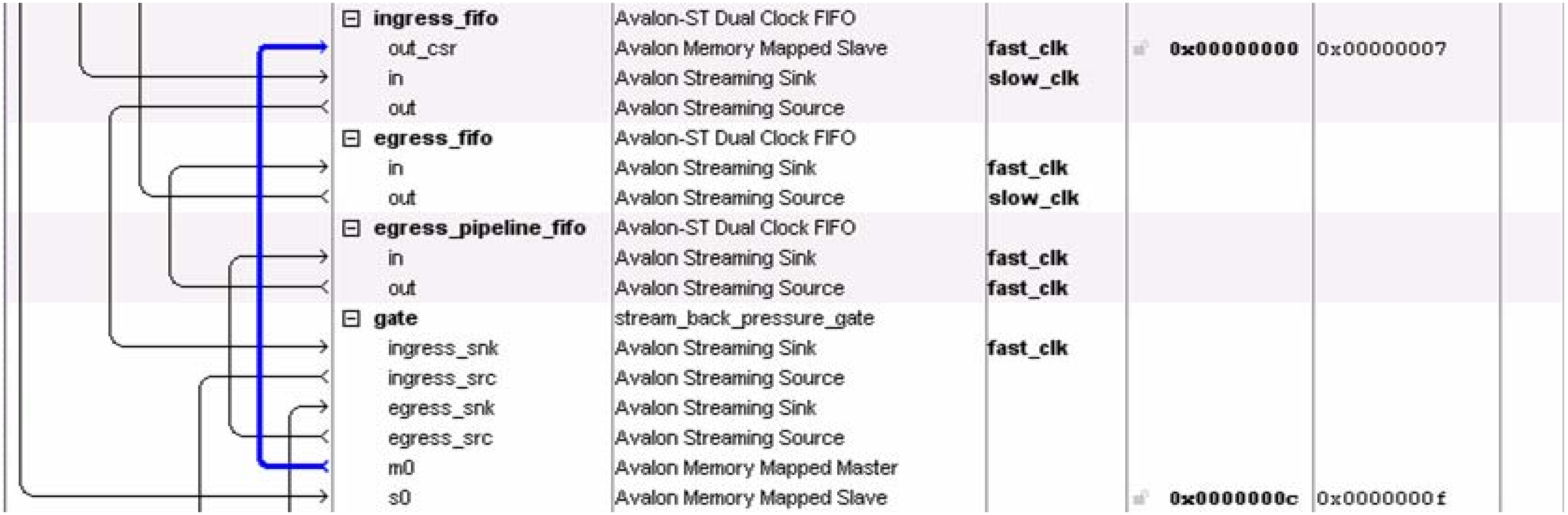
- You can see the egress stream path in the picture below. All of the source interfaces are highlighted which in turn highlights a blue path to their respective sink counterparts.
- The stream begins at the dut peripheral as a 2 symbol per beat, 8 bit per symbol source interface with no ready and no valid strobes. The stream then enters the gate sink interface which has ready and valid strobes.
- The gate source then feeds the egress_pipeline_fifo sink, this FIFO is placed in the system to absorb any pipeline delay created by the DUT peripheral. When the gate opens, the first few results out of the DUT will be the stale pipeline flush.
- The egress_pipeline_fifo source then feeds the egress_fifo sink. The egress FIFO is sized to the same depth as the ingress FIFO.
- The egress_fifo source then feeds the console_stream sink. The egress_fifo source is a 2 symbol per beat, 8 bit per symbol interface with ready and valid strobes, and the console_stream sink is a 1 symbol per beat, 8 bit per symbol interface with ready and valid strobes.



⊟ **console_stream**	Avalon-ST JTAG Interface			
src	Avalon Streaming Source	**slow_clk**		
sink	Avalon Streaming Sink			
⊟ **ingress_fifo**	Avalon-ST Dual Clock FIFO			
out_csr	Avalon Memory Mapped Slave	**fast_clk**	**0x00000000**	0x00000007
in	Avalon Streaming Sink	**slow_clk**		
out	Avalon Streaming Source			
⊟ **egress_fifo**	Avalon-ST Dual Clock FIFO			
in	Avalon Streaming Sink	**fast_clk**		
out	Avalon Streaming Source	**slow_clk**		
⊟ **egress_pipeline_fifo**	Avalon-ST Dual Clock FIFO			
in	Avalon Streaming Sink	**fast_clk**		
out	Avalon Streaming Source	**fast_clk**		
⊟ **gate**	stream_back_pressure_gate			
ingress_snk	Avalon Streaming Sink	**fast_clk**		
ingress_src	Avalon Streaming Source			
egress_snk	Avalon Streaming Sink			
egress_src	Avalon Streaming Source			
m0	Avalon Memory Mapped Master			
s0	Avalon Memory Mapped Slave		**0x0000000c**	0x0000000f
⊟ **dut**	my_dut			
sink	Avalon Streaming Sink	**fast_clk**		
source	Avalon Streaming Source			

Detecting the ingress fill level.

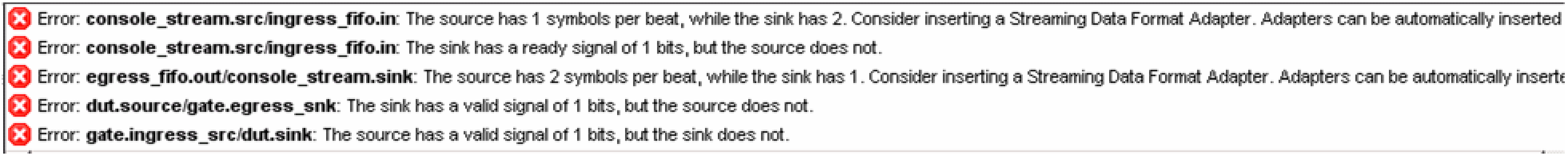
- The Avalon MM master on the gate component allows the component to read the out_csr slave that is provided on the ingress_fifo.
- The ingress_fifo provides a 24 bit fill level value that is read at offset 0 in the out_csr slave.



⊟ **ingress_fifo**	Avalon-ST Dual Clock FIFO			
out_csr	Avalon Memory Mapped Slave	**fast_clk**	**0x00000000**	0x00000007
in	Avalon Streaming Sink	**slow_clk**		
out	Avalon Streaming Source			
⊟ **egress_fifo**	Avalon-ST Dual Clock FIFO			
in	Avalon Streaming Sink	**fast_clk**		
out	Avalon Streaming Source	**slow_clk**		
⊟ **egress_pipeline_fifo**	Avalon-ST Dual Clock FIFO			
in	Avalon Streaming Sink	**fast_clk**		
out	Avalon Streaming Source	**fast_clk**		
⊟ **gate**	stream_back_pressure_gate			
ingress_snk	Avalon Streaming Sink	**fast_clk**		
ingress_src	Avalon Streaming Source			
egress_snk	Avalon Streaming Sink			
egress_src	Avalon Streaming Source			
m0	Avalon Memory Mapped Master			
s0	Avalon Memory Mapped Slave		**0x0000000c**	0x0000000f

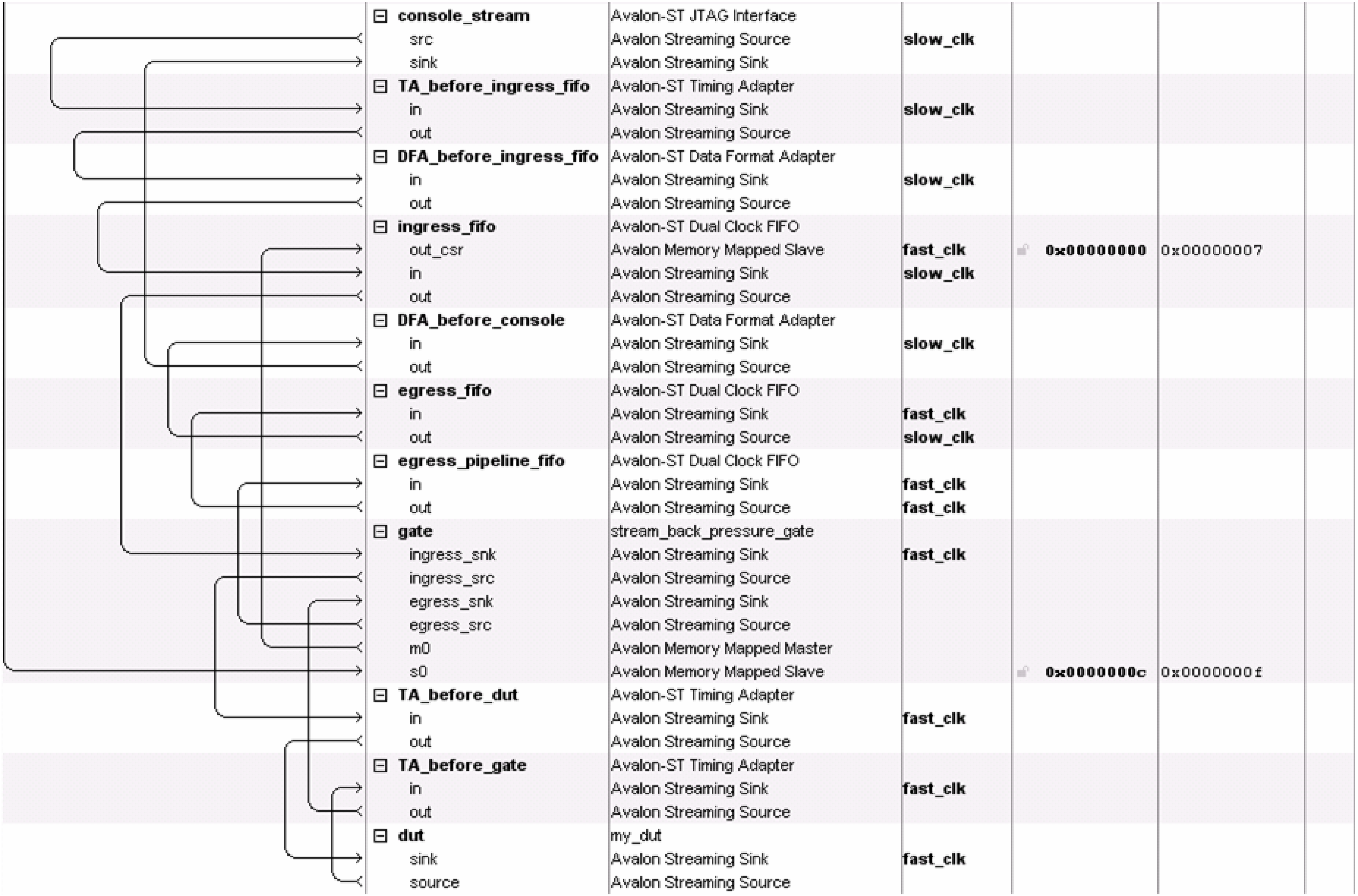
Dealing with the Avalon ST interface mismatches .

- If you paid attention to the Avalon ST interface descriptions on the previous slides, you noticed that we were connecting things together that did not match up exactly. Some interfaces had different data widths, others had different enable strobes, etc.
- SOPC Builder displays these errors in it’s GUI, and if you notice, it suggests that we allow it to automatically insert adapters into our system to compensate for these differences among the components.
- The next step is to do just that, have SOPC Builder automatically insert the necessary Avalon ST adapters for us.



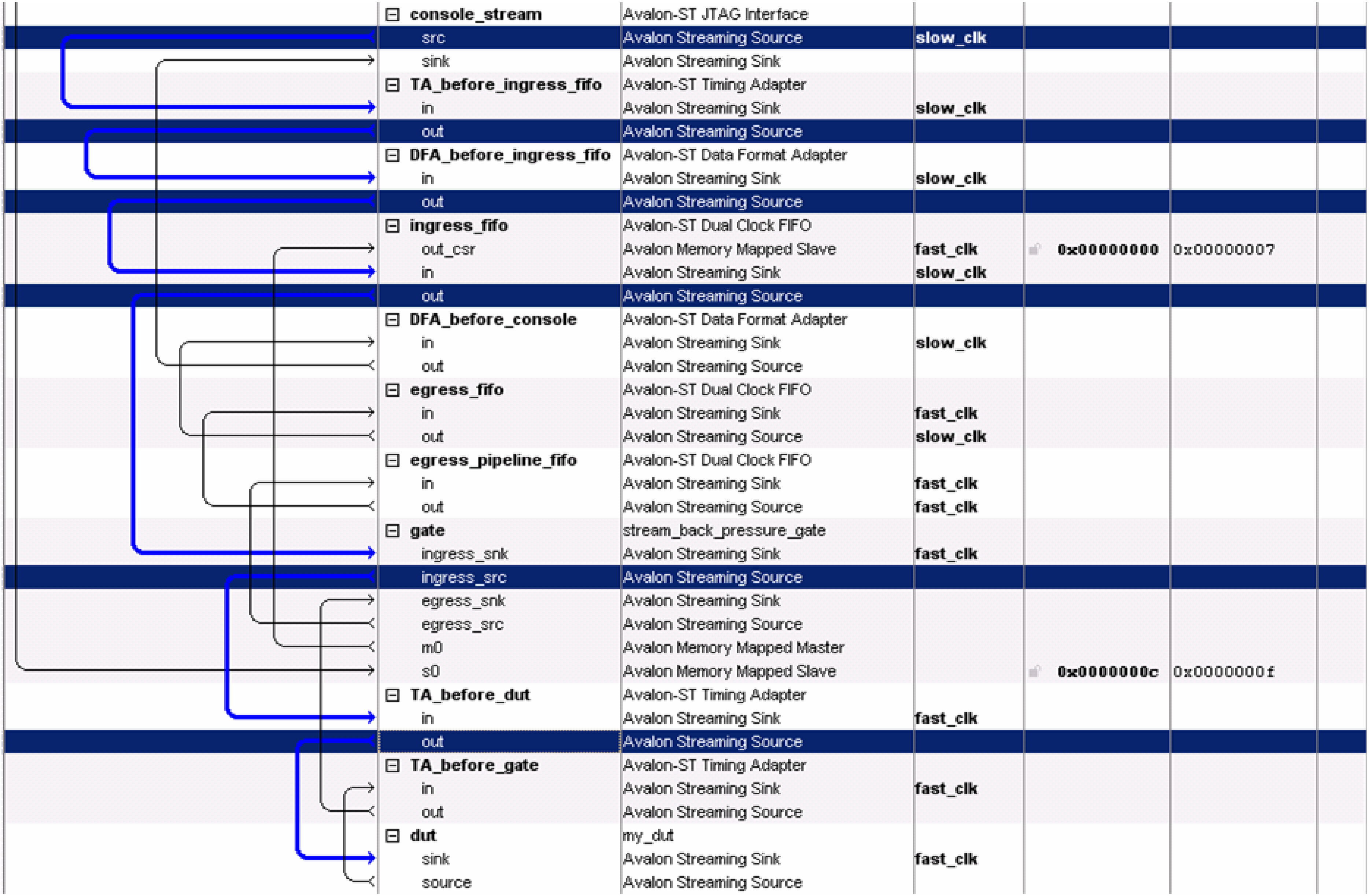
```
Error: console_stream.src/ingress_fifo.in: The source has 1 symbols per beat, while the sink has 2. Consider inserting a Streaming Data Format Adapter. Adapters can be automatically inserted
Error: console_stream.src/ingress_fifo.in: The sink has a ready signal of 1 bits, but the source does not.
Error: egress_fifo.out/console_stream.sink: The source has 2 symbols per beat, while the sink has 1. Consider inserting a Streaming Data Format Adapter. Adapters can be automatically inserte
Error: dut.source/gate.egress_snk: The sink has a valid signal of 1 bits, but the source does not.
Error: gate.ingress_src/dut.sink: The source has a valid signal of 1 bits, but the sink does not.
```

The auto adapted system.



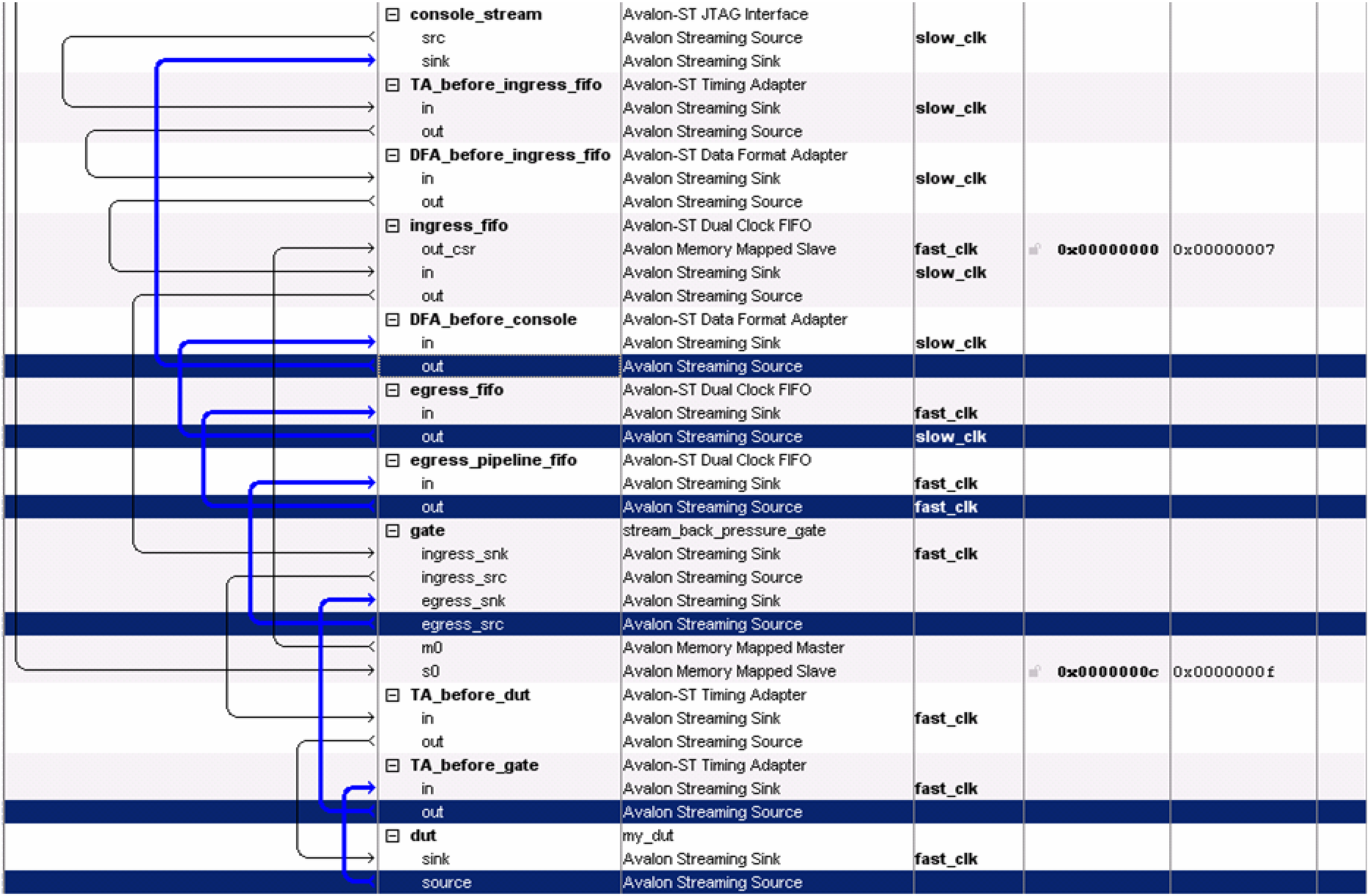
⊟ **console_stream**	Avalon-ST JTAG Interface			
src	Avalon Streaming Source	**slow_clk**		
sink	Avalon Streaming Sink			
⊟ **TA_before_ingress_fifo**	Avalon-ST Timing Adapter			
in	Avalon Streaming Sink	**slow_clk**		
out	Avalon Streaming Source			
⊟ **DFA_before_ingress_fifo**	Avalon-ST Data Format Adapter			
in	Avalon Streaming Sink	**slow_clk**		
out	Avalon Streaming Source			
⊟ **ingress_fifo**	Avalon-ST Dual Clock FIFO			
out_csr	Avalon Memory Mapped Slave	**fast_clk**	**0x00000000**	0x00000007
in	Avalon Streaming Sink	**slow_clk**		
out	Avalon Streaming Source			
⊟ **DFA_before_console**	Avalon-ST Data Format Adapter			
in	Avalon Streaming Sink	**slow_clk**		
out	Avalon Streaming Source			
⊟ **egress_fifo**	Avalon-ST Dual Clock FIFO			
in	Avalon Streaming Sink	**fast_clk**		
out	Avalon Streaming Source	**slow_clk**		
⊟ **egress_pipeline_fifo**	Avalon-ST Dual Clock FIFO			
in	Avalon Streaming Sink	**fast_clk**		
out	Avalon Streaming Source	**fast_clk**		
⊟ **gate**	stream_back_pressure_gate			
ingress_snk	Avalon Streaming Sink	**fast_clk**		
ingress_src	Avalon Streaming Source			
egress_snk	Avalon Streaming Sink			
egress_src	Avalon Streaming Source			
m0	Avalon Memory Mapped Master			
s0	Avalon Memory Mapped Slave		**0x0000000c**	0x0000000f
⊟ **TA_before_dut**	Avalon-ST Timing Adapter			
in	Avalon Streaming Sink	**fast_clk**		
out	Avalon Streaming Source			
⊟ **TA_before_gate**	Avalon-ST Timing Adapter			
in	Avalon Streaming Sink	**fast_clk**		
out	Avalon Streaming Source			
⊟ **dut**	my_dut			
sink	Avalon Streaming Sink	**fast_clk**		
source	Avalon Streaming Source			

The ingress flow thru the auto adapters.



⊟ **console_stream**	Avalon-ST JTAG Interface			
src	Avalon Streaming Source	**slow_clk**		
sink	Avalon Streaming Sink			
⊟ **TA_before_ingress_fifo**	Avalon-ST Timing Adapter			
in	Avalon Streaming Sink	**slow_clk**		
out	Avalon Streaming Source			
⊟ **DFA_before_ingress_fifo**	Avalon-ST Data Format Adapter			
in	Avalon Streaming Sink	**slow_clk**		
out	Avalon Streaming Source			
⊟ **ingress_fifo**	Avalon-ST Dual Clock FIFO			
out_csr	Avalon Memory Mapped Slave	**fast_clk**	**0x00000000**	0x00000007
in	Avalon Streaming Sink	**slow_clk**		
out	Avalon Streaming Source			
⊟ **DFA_before_console**	Avalon-ST Data Format Adapter			
in	Avalon Streaming Sink	**slow_clk**		
out	Avalon Streaming Source			
⊟ **egress_fifo**	Avalon-ST Dual Clock FIFO			
in	Avalon Streaming Sink	**fast_clk**		
out	Avalon Streaming Source	**slow_clk**		
⊟ **egress_pipeline_fifo**	Avalon-ST Dual Clock FIFO			
in	Avalon Streaming Sink	**fast_clk**		
out	Avalon Streaming Source	**fast_clk**		
⊟ **gate**	stream_back_pressure_gate			
ingress_snk	Avalon Streaming Sink	**fast_clk**		
ingress_src	Avalon Streaming Source			
egress_snk	Avalon Streaming Sink			
egress_src	Avalon Streaming Source			
m0	Avalon Memory Mapped Master			
s0	Avalon Memory Mapped Slave		**0x0000000c**	0x0000000f
⊟ **TA_before_dut**	Avalon-ST Timing Adapter			
in	Avalon Streaming Sink	**fast_clk**		
out	Avalon Streaming Source			
⊟ **TA_before_gate**	Avalon-ST Timing Adapter			
in	Avalon Streaming Sink	**fast_clk**		
out	Avalon Streaming Source			
⊟ **dut**	my_dut			
sink	Avalon Streaming Sink	**fast_clk**		
source	Avalon Streaming Source			

The egress flow thru the auto adapters.



Name	Description	Clock	Base	End
⊟ **console_stream**	Avalon-ST JTAG Interface			
src	Avalon Streaming Source	**slow_clk**		
sink	Avalon Streaming Sink			
⊟ **TA_before_ingress_fifo**	Avalon-ST Timing Adapter			
in	Avalon Streaming Sink	**slow_clk**		
out	Avalon Streaming Source			
⊟ **DFA_before_ingress_fifo**	Avalon-ST Data Format Adapter			
in	Avalon Streaming Sink	**slow_clk**		
out	Avalon Streaming Source			
⊟ **ingress_fifo**	Avalon-ST Dual Clock FIFO			
out_csr	Avalon Memory Mapped Slave	**fast_clk**	**0x00000000**	0x00000007
in	Avalon Streaming Sink	**slow_clk**		
out	Avalon Streaming Source			
⊟ **DFA_before_console**	Avalon-ST Data Format Adapter			
in	Avalon Streaming Sink	**slow_clk**		
out	Avalon Streaming Source			
⊟ **egress_fifo**	Avalon-ST Dual Clock FIFO			
in	Avalon Streaming Sink	**fast_clk**		
out	Avalon Streaming Source	**slow_clk**		
⊟ **egress_pipeline_fifo**	Avalon-ST Dual Clock FIFO			
in	Avalon Streaming Sink	**fast_clk**		
out	Avalon Streaming Source	**fast_clk**		
⊟ **gate**	stream_back_pressure_gate			
ingress_snk	Avalon Streaming Sink	**fast_clk**		
ingress_src	Avalon Streaming Source			
egress_snk	Avalon Streaming Sink			
egress_src	Avalon Streaming Source			
m0	Avalon Memory Mapped Master			
s0	Avalon Memory Mapped Slave		**0x0000000c**	0x0000000f
⊟ **TA_before_dut**	Avalon-ST Timing Adapter			
in	Avalon Streaming Sink	**fast_clk**		
out	Avalon Streaming Source			
⊟ **TA_before_gate**	Avalon-ST Timing Adapter			
in	Avalon Streaming Sink	**fast_clk**		
out	Avalon Streaming Source			
⊟ **dut**	my_dut			
sink	Avalon Streaming Sink	**fast_clk**		
source	Avalon Streaming Source			

Putting it all together.

- Once we have the viable SOPC system, we allow SOPC Builder to generate the system for us.
- Now we need to tie the SOPC system module into the FPGA top level module, which can be done quite easily in a simple top level wrapper as shown below.
- In this integrated system flow, there are really only two signals required by this system, the clock and the reset from the I/O pins of the FPGA. All the other module interconnection is done within the SOPC system instance.

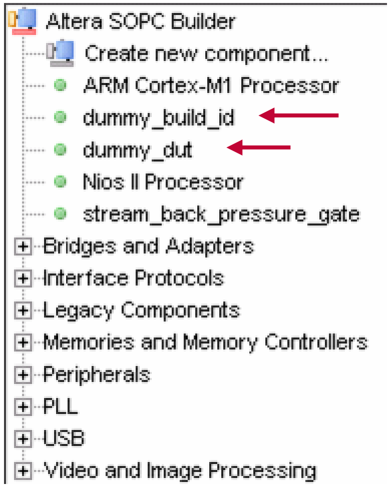
```
module test_project_top (

    input clk,
    input reset_n
);

test_sys_sopc test_sys_sopc_inst (
    // 1) global signals:
    .clk         (clk),
    .fast_clk    (),
    .reset_n     (reset_n),
    .slow_clk    (),

    // the_console_stream
    .resetrequest_from_the_console_stream   ()
);

endmodule
```

Creating the System in SOPC Builder

The separated approach.

The separated approach.

- The previous section describe the SOPC system construction from an integrated approach, where the first step was to import our custom components as full fledge SOPC Builder components and then add them into the system.
- In the separated approach, what we will do first is create a couple of component “shells” which only have the SOPC interfaces defined, but don’t have any actual HDL standing behind them. These SOPC interfaces will be promoted to the top level port map of the SOPC generated HDL, and then we will manually stitch our custom hardware components into that top level.
- These component shells are shown to the right as “dummy_build_id” and “dummy_dut”.

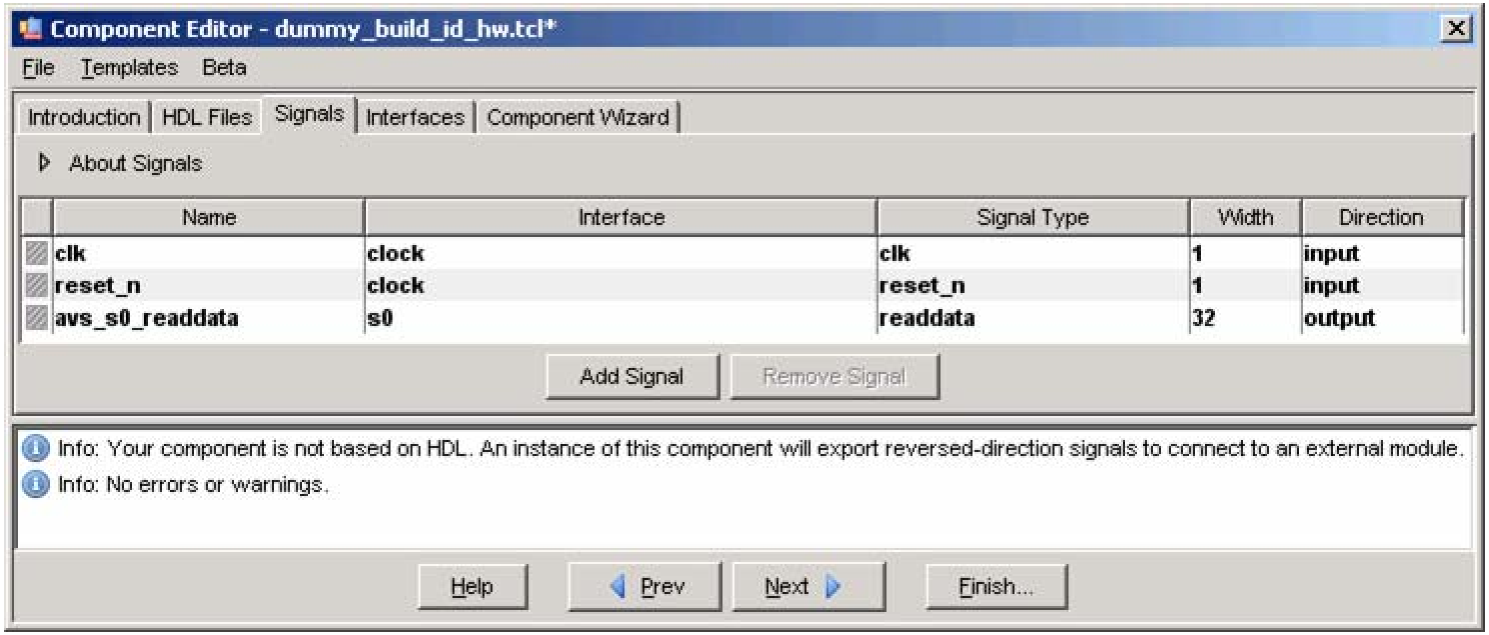


Altera SOPC Builder
- Create new component...
- ARM Cortex-M1 Processor
- dummy_build_id
- dummy_dut
- Nios II Processor
- stream_back_pressure_gate

Bridges and Adapters
Interface Protocols
Legacy Components
Memories and Memory Controllers
Peripherals
PLL
USB
Video and Image Processing

Creating dummy_build_id

- Creating the dummy_build_id shell is quite easy, you launch component editor like you would to import a component, however, you skip the HDL Files import and go straight to the Signals tab of the GUI.
- Select the typical Avalon Slave interface from the templates menu and you'll get a list of typical Avalon slave signals populated in the dialog.
- Remove all the unwanted signals until you have something like shown below.
- Finish up the naming and such for this component a click the finish button to save off the hw.tcl file that defines this component.



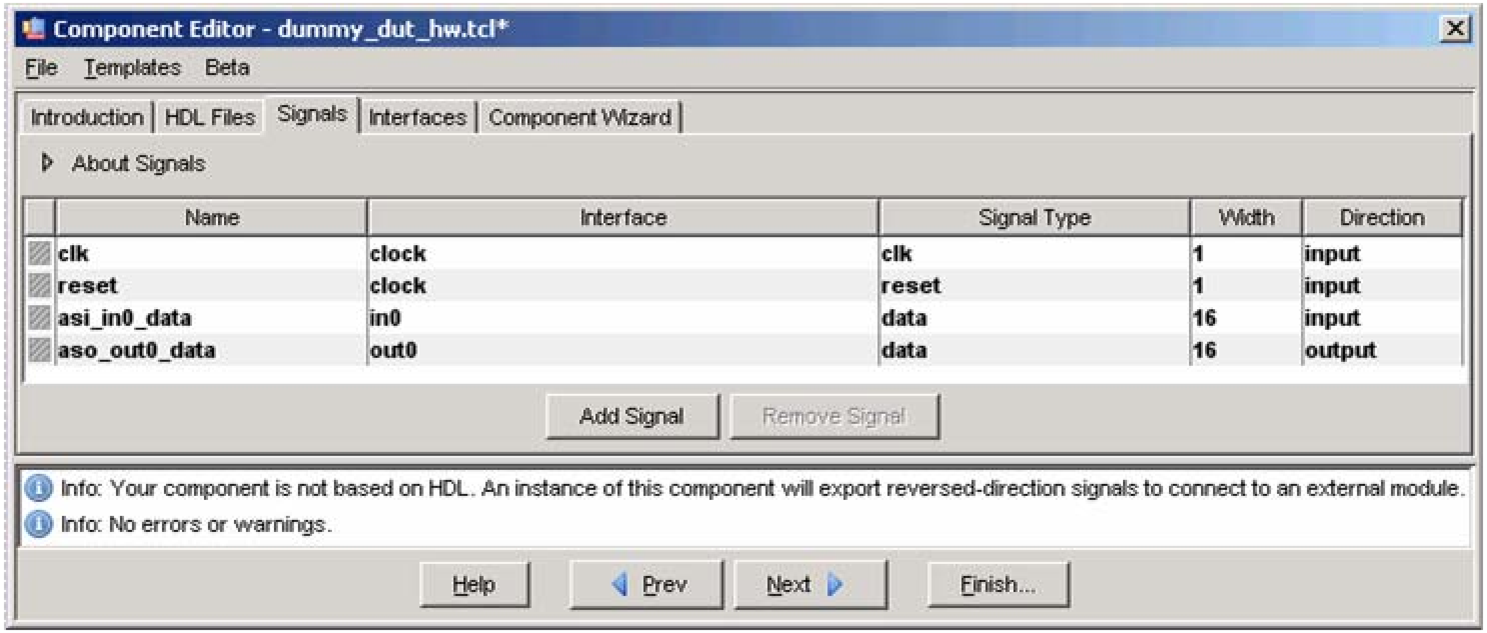
Name	Interface	Signal Type	Width	Direction
clk	clock	clk	1	input
reset_n	clock	reset_n	1	input
avs_s0_readdata	s0	readdata	32	output

Info: Your component is not based on HDL. An instance of this component will export reversed-direction signals to connect to an external module.

Info: No errors or warnings.

Creating dummy_dut

- Creating the dummy_dut shell is quite easy, you launch component editor like you would to import a component, however, you skip the HDL Files import and go straight to the Signals tab of the GUI.
- Select the typical Avalon sink interface from the templates menu and you'll get a list of typical Avalon sink signals populated in the dialog. Then select the typical Avalon source interface from the templates menu to create those signals as well
- Remove all the unwanted signals until you have something like shown below.
- Finish up the naming and such for this component a click the finish button to save off the hw.tcl file that defines this component.



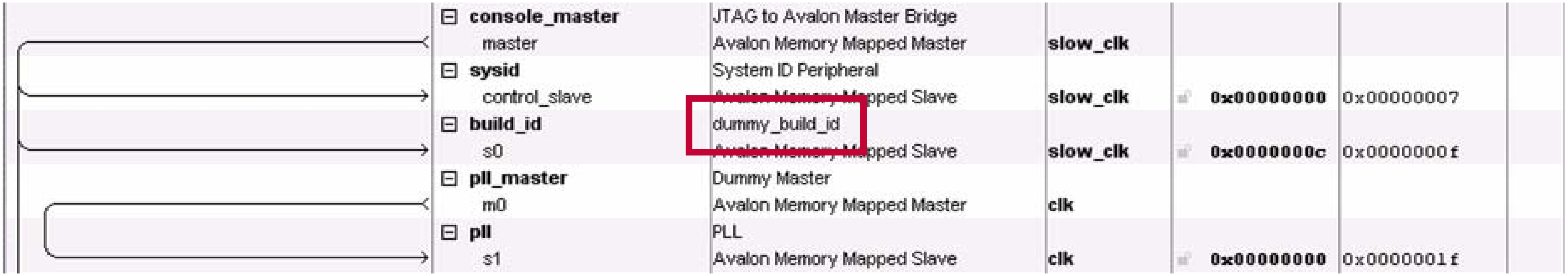
	Name	Interface	Signal Type	Width	Direction
▨	clk	clock	clk	1	input
▨	reset	clock	reset	1	input
▨	asi_in0_data	in0	data	16	input
▨	aso_out0_data	out0	data	16	output

Info: Your component is not based on HDL. An instance of this component will export reversed-direction signals to connect to an external module.

Info: No errors or warnings.

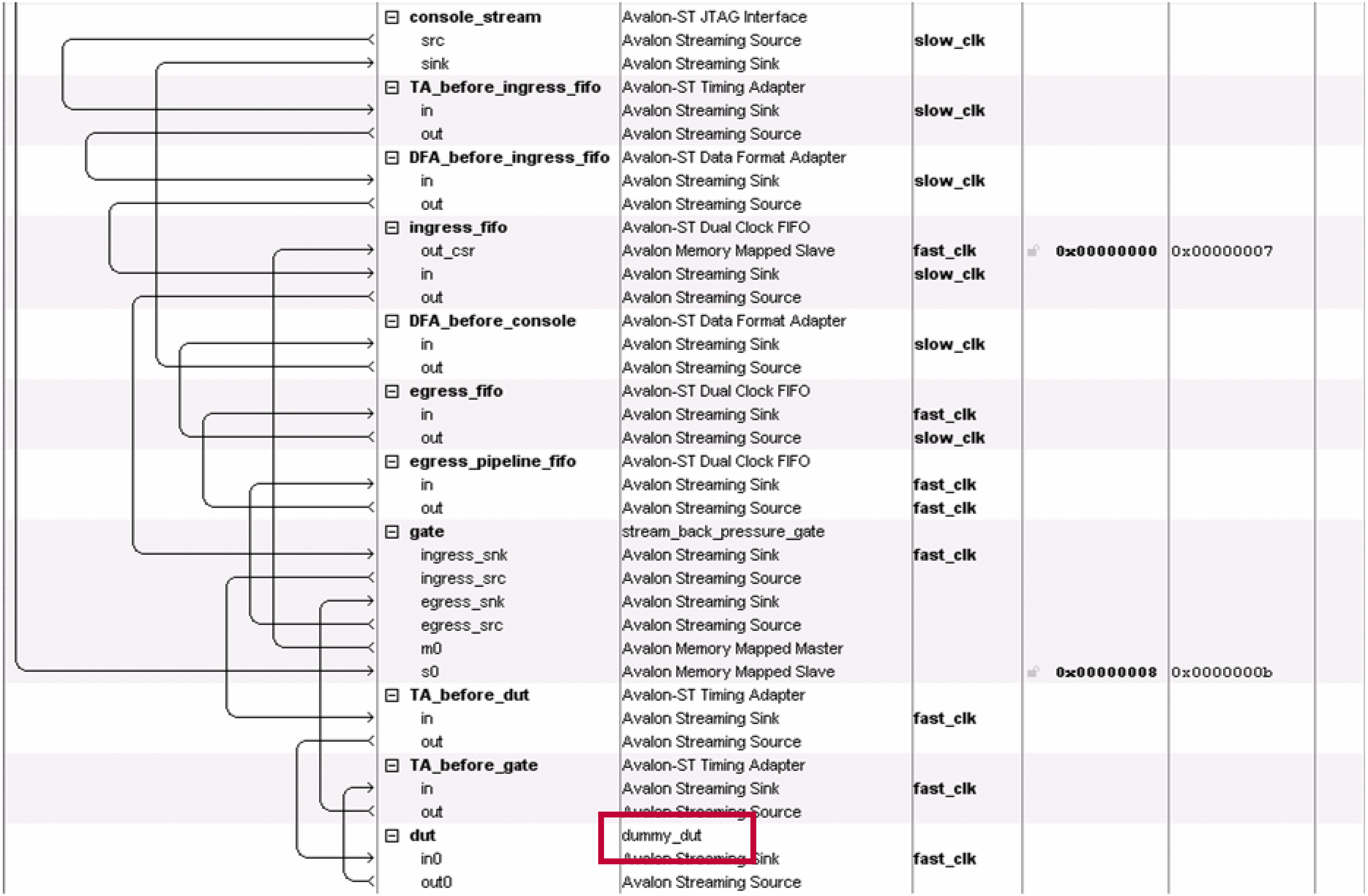
The separated system

- The separated system should look nearly identical to the previously constructed integrated system.
- There are only two subtle differences shown on this slide and the following slide. The buid_id component is actually the dummy_build_id and the dut is actually the dummy_dut.



⊟ **console_master**	JTAG to Avalon Master Bridge			
master	Avalon Memory Mapped Master	**slow_clk**		
⊟ **sysid**	System ID Peripheral			
control_slave	Avalon Memory Mapped Slave	**slow_clk**	**0x00000000**	0x00000007
⊟ **build_id**	dummy_build_id			
s0	Avalon Memory Mapped Slave	**slow_clk**	**0x0000000c**	0x0000000f
⊟ **pll_master**	Dummy Master			
m0	Avalon Memory Mapped Master	**clk**		
⊟ **pll**	PLL			
s1	Avalon Memory Mapped Slave	**clk**	**0x00000000**	0x0000001f

The separated system



⊟ **console_stream**	Avalon-ST JTAG Interface			
src	Avalon Streaming Source	**slow_clk**		
sink	Avalon Streaming Sink			
⊟ **TA_before_ingress_fifo**	Avalon-ST Timing Adapter			
in	Avalon Streaming Sink	**slow_clk**		
out	Avalon Streaming Source			
⊟ **DFA_before_ingress_fifo**	Avalon-ST Data Format Adapter			
in	Avalon Streaming Sink	**slow_clk**		
out	Avalon Streaming Source			
⊟ **ingress_fifo**	Avalon-ST Dual Clock FIFO			
out_csr	Avalon Memory Mapped Slave	**fast_clk**	**0x00000000**	0x00000007
in	Avalon Streaming Sink	**slow_clk**		
out	Avalon Streaming Source			
⊟ **DFA_before_console**	Avalon-ST Data Format Adapter			
in	Avalon Streaming Sink	**slow_clk**		
out	Avalon Streaming Source			
⊟ **egress_fifo**	Avalon-ST Dual Clock FIFO			
in	Avalon Streaming Sink	**fast_clk**		
out	Avalon Streaming Source	**slow_clk**		
⊟ **egress_pipeline_fifo**	Avalon-ST Dual Clock FIFO			
in	Avalon Streaming Sink	**fast_clk**		
out	Avalon Streaming Source	**fast_clk**		
⊟ **gate**	stream_back_pressure_gate			
ingress_snk	Avalon Streaming Sink	**fast_clk**		
ingress_src	Avalon Streaming Source			
egress_snk	Avalon Streaming Sink			
egress_src	Avalon Streaming Source			
m0	Avalon Memory Mapped Master			
s0	Avalon Memory Mapped Slave		**0x00000008**	0x0000000b
⊟ **TA_before_dut**	Avalon-ST Timing Adapter			
in	Avalon Streaming Sink	**fast_clk**		
out	Avalon Streaming Source			
⊟ **TA_before_gate**	Avalon-ST Timing Adapter			
in	Avalon Streaming Sink	**fast_clk**		
out	Avalon Streaming Source			
⊟ **dut**	dummy_dut			
in0	Avalon Streaming Sink	**fast_clk**		
out0	Avalon Streaming Source			

Putting it all together.

- Once we have the viable SOPC system, we allow SOPC Builder to generate the system for us.
- Now we need to tie the SOPC system module into the FPGA top level module, which can be done quite easily in a simple top level wrapper as shown to the right.
- In this separated system flow, in addition to wiring up the clock and the reset from the I/O pins of the FPGA, all the other module interconnection is done manually.
- You can see the extra interface ports that have sprung out of the top level port map of the SOPC system.

```
module test_project_top (

    input clk,
    input reset_n
);

wire fast_clk;
wire slow_clk;
wire [31:0] avs_s0_readdata_from_the_build_id;
wire reset_n_to_the_build_id;
wire [15:0] asi_in0_data_to_the_dut_in0;
wire reset_to_the_dut_in0;
wire [15:0] aso_out0_data_from_the_dut_out0;

test_sys_sopc test_sys_sopc_inst (
    // 1) global signals:
    .clk                                    (clk),
    .fast_clk                               (fast_clk),
    .reset_n                                (reset_n),
    .slow_clk                               (slow_clk),

    // the_build_id_s0
    .avs_s0_readdata_from_the_build_id      (avs_s0_readdata_from_the_build_id),
    .reset_n_to_the_build_id                (reset_n_to_the_build_id),

    // the_console_stream
    .resetrequest_from_the_console_stream   (),

    // the_dut_in0
    .asi_in0_data_to_the_dut_in0            (asi_in0_data_to_the_dut_in0),
    .reset_to_the_dut_in0                   (reset_to_the_dut_in0),

    // the_dut_out0
    .aso_out0_data_from_the_dut_out0        (aso_out0_data_from_the_dut_out0)
);
```

Putting it all together.

- So we create some wires and manually stitch the SOPC port map into the top level port maps of our other components.

```
my_build_id my_build_id_inst (
    // Clock Interface
    .csi_clock_clk                          (slow_clk),
    .csi_clock_reset                        (reset_n_to_the_build_id),

    // MM Slave Interface
    .avs_s0_readdata                        (avs_s0_readdata_from_the_build_id)
);

my_dut my_dut_inst (
    // Clock Interface
    .csi_clock_clk                          (fast_clk),
    .csi_clock_reset                        (reset_to_the_dut_in0),

    // ST Sink Interface
    .asi_sink_data                          (asi_in0_data_to_the_dut_in0),

    // ST Source Interface
    .aso_source_data                        (aso_out0_data_from_the_dut_out0)
);

endmodule
```

Accessing the hardware system from System Console.

Running this example on a NEEK board.

- This example has been provided with two project archives, one that illustrates the integrated system flow and the other illustrates the separated system flow.
- You can use either one of these systems as an example that runs on the NEEK development board.
- System Console TCL scripts are provided to allow very high level access and control over this hardware model. These scripts allow you to do things like:
  - Validate the system ID peripheral.
  - Display the build id value.
  - Query or change the FIFO fill trigger level of the gate component.
  - Create a binary test data file.
  - Stream a binary data file thru the hardware design and capture the results into an output file.

Getting into the System Console.

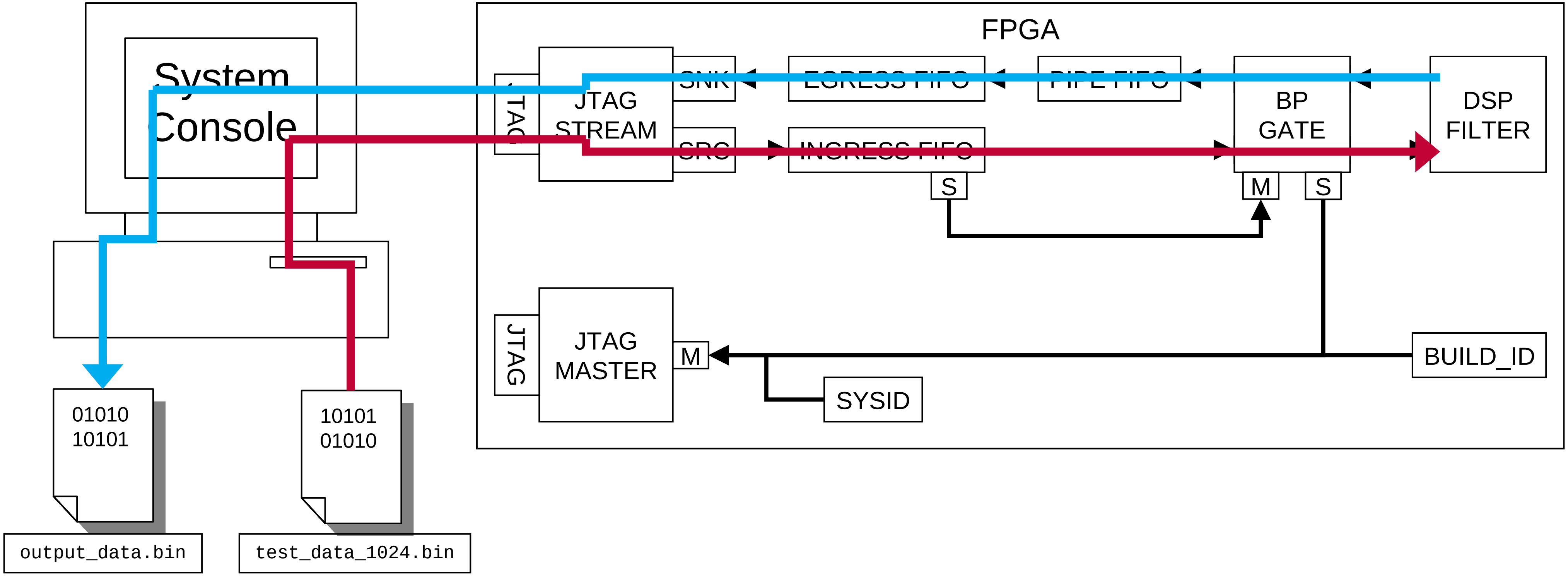
- To begin, select a example archive and expand it on your hard drive in a path that has no spaces in it. Either example should be fine as they should both work identically.
- Make sure that you have the 8.0SP1 Quartus II and Nios II development tools installed on your workstation. You don’t need the Nios II tools to run System Console, but the following directions assume that you have them and makes use of various utilities provided in them. This example does not discuss alternate ways of accomplishing the same results.
- Connect your NEEK board to your workstation and power on the NEEK board.
- Open a Nios II Command Shell
  - Start -> Programs -> Altera -> Nios II 8.0 EDS -> Nios II 8.0 Command Shell
- In the Nios II command shell change directory to the sc_tcl directory contained in the example design directory.
  - cd "C:\bytestream_example\integrated_hw\sc_tcl“
- Next we need to program the FPGA with the precompiled SOF file.
  - nios2-configure-sof ../test_project_top.sof
- Then we need to launch the System Console command shell.
  - system-console --project_dir=..

Operating within the system console shell.

- Once the system console shell comes up, the first thing we need to do is initialize the shell with all of the scripts provided with this example. The source code to these TCL scripts is available in the directory that we should be running out of, sc_tcl. Source the initialization script like this:
  - source sc_init.tcl
- If that script runs successfully, then we should be able to validate the system ID peripheral in the FPGA with this command:
  - sc_validate_sysid $sc_env
- If we have valid hardware in the FPGA then we should be able to display the build id value with this command:
  - sc_build_id_display $sc_env
- Now if we want to test the hardware streaming, we can create a binary data file for testing with this command which creates an incrementing 16 bit value for the specified count into the specified file:
  - sc_create_incrementing_test_file 1024 "test_data_1024.bin“
- Now that we have some test data, we can pass it thru the hardware with this command:
  - sc_test_stream $sc_env "test_data_1024.bin" "output_data.bin" 2048 3072
- At this point the file “output_data.bin” should have the results that were collected from that data stream. You should be able to spot the inverted values of that 16 bit incrementing pattern. The pipeline depth of our hardware example is just 1, so the first 16 bit value should be whatever the last value was presented into the sink interface of our component.
- Now we can change the gate trigger level and try a smaller pass of data, we change the trigger level with this command:
  - sc_write_gate_trigger_level $sc_env 512
- And now we modify the test command to account for less input data:
  - sc_test_stream $sc_env "test_data_1024.bin" "output_data.bin" 1024 3072
- In general while you are in system console you should be able to type “sc_<tab>” to have a list of all the commands beginning with “sc_” appear. Then you can up or down arrow to select one.
- You should also be able to type “help <command>” to get help for any given command. All of the commands provided with this example should report help back, and hopefully it’s helpful.

What’s the test doing?

- The picture below illustrates what the streaming test is doing that we execute on the previous slide.
- First system console opens the test data input file and reads in the binary data.
- Then system console streams the data into the hardware system.
- The “DSP Filter” in our example simply inverts the data.
- Then system console streams the results out of the hardware and stores them in the output data file.



Summary

- This presentation has illustrated how System Console and the JTAG components in SOPC Builder can quickly and easily be combined to create a test and analysis environment for streaming peripherals.
- All of the source code for the TCL scripts as well as the HDL code for the hardware peripherals is provided in the example archives delivered with this example. The TCL sources are contained in the “sc_tcl” directory, and the custom hardware sources are contained in the “ip” directory.
- For more information on the System Console and SOPC Builder, you should refer to their respective manuals, mentioned earlier in this presentation.