

HOWTO Supplemental Reset Components for Qsys

Document Revision 1.0

Built on December 27, 2021

Contents

Introduction	1
Power On Reset Component	2
Reset Debouncer Component	2
Reset Event Counter Component	3
Trivial Default Avalon Slave Component	4
PLL Reset Monitor Component	7
Reset Assertion Delay Component	11
Event Timer Component	13
EMIF System Example Explanation	16
Reset Until Ack Component	25
Component HDL Comment Blocks	26
Power On Reset Component	26
Reset Debouncer Component	26
Reset Event Counter Component	27
Trivial Default Avalon Slave Component	28
PLL Reset Monitor Component	28
Reset Assertion Delay Component	29
Event Timer Component	30
Reset Until Ack Component	30

Introduction

This document explains how to implement the Supplemental Reset Components for Qsys in practical real world applications. Each of the components is described in it's own section below to illustrate how it could be deployed in the Qsys environment. The final section of this document contains the comment block from the top of each HDL file that describes every component, since that represents the technical details for the component implementation.

Power On Reset Component

The POR (Power On Reset) component is a very trivial component that is intended to assert a reset signal from the FPGA entry into user mode and release the reset signal after a user specified duration. For more details on the component functionality please refer to the comment block at the top of the HDL file that describes the component, or see code Listing 1 below.

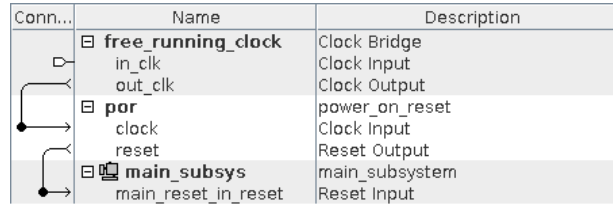
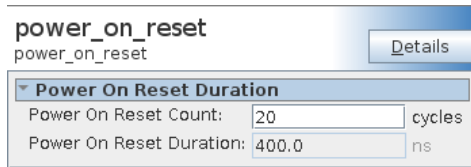
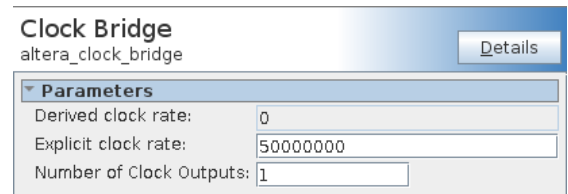


Figure 1: POR Qsys System

Please refer to the *por* instance of the *power_on_reset* component shown in Figure 1. The POR component requires a free running and stable *clock* input, typically provided by an external clock, not an internal PLL. When the POR component enters user mode the *reset* output will be asserted and will remain asserted until the user specified clock count occurs. The reset output from the POR component can be applied to any Qsys reset input interface. In the example shown above in Figure 1, you can see that we simply have this reset drive an entire subsystem reset input interface.



(a) POR Parameters



(b) Free Running Clock Parameters

Figure 2: Parameters

The POR component requires one parameter to be set which indicates how long the shift chain is that implements the reset delay. In the example above you can see that we have set the Power On Reset Count to 20 clock cycles, and the component knows that the frequency that our free running clock has declared is 50MHz so the POR component displays the calculated duration in nanoseconds.

Reset Debouncer Component

The Reset Debouncer component is designed to debounce noisy reset signals like you may find in a push button switch that generates a reset input to your device. When an assertion edge is detected at the *reset_input* interface, the Reset Debouncer will assert the *reset_output* interface and it will remain asserted until the *reset_input* signal is deasserted for a user specified amount of time. Then the Reset Debouncer will release the *reset_output* interface. There is an Avalon slave interface on the Reset Debouncer such that a master in the system can read the internal status register to examine the assertion and deassertion counts experienced by the component. The *power_on_reset* input ensures that the internal counters are cleared only at power on. For more details on the component functionality please refer to the comment block at the top of the HDL file that describes the component, or see code Listing 2 below.

Connections	Name	Description
□	free_running_clock	Clock Bridge
	in_clk	Clock Input
□	external_reset_pin	Reset Bridge
	in_reset	Reset Input
□	por	power_on_reset
	clock	Clock Input
□	rd	reset_debouncer
	debounce_clock	Clock Input
□	power_on_reset	Reset Input
	reset_input	Reset Input
□	reset_output	Reset Output
	s0_clk	Clock Input
□	s0_reset	Reset Input
	s0	Avalon Memory Mapped Slave
□	async_subsystem	async_subsystem
	cr_clk	Clock Input
□	cr_reset	Reset Input
	master_m0	Avalon Memory Mapped Master

Figure 3: Reset Debouncer Qsys System

Please refer to the *rd* instance of the *reset_debouncer* component shown in Figure 3. The *debounce_clock* should be driven by a free running and stable clock, typically provided by an external clock, not an internal PLL. The *power_on_reset* input should be driven by a signal as we described in the Power On Reset Component section above. The *reset_input* interface should be driven by the potentially noisy reset input signal. The *reset_output* interface can drive any appropriate Qsys reset input interface. The *s0_clk* and *s0_reset* should be driven by appropriate clock and reset domains to support the *s0* Avalon slave interface.

reset_debouncer
reset_debouncer

Details

Debounce Counter
Debounce Counter Width: 22 bits
Debounce Counter Duration: 8.388608E7 ns

Figure 4: Reset Debouncer Parameters

The only parameter that this component requires is the width of the debounce counter which determines the terminal count for the debounce delay. In this example we have set this to 22 bits which produces a terminal count of about 84 milliseconds.

Reset Event Counter Component

The Reset Event Counter component is designed to count reset events. When an assertion or deassertion edge is detected at the *reset_event* interface, the Reset Event Counter will increment the assertion or deassertion counters respectively. There is an Avalon slave interface on the Reset Event Counter such that a master in the system can read the internal status register to examine the assertion and deassertion events experienced by the component. The *power_on_reset* input ensures that the internal counters are cleared only at power on. For more details on the component functionality please refer to the comment block at the top of the HDL file that describes the component, or see code Listing 3 below.

In Figure 8 we have highlighted the main system reset sources in the Qsys system. There is a POR reset and an external reset pin in the example that are not highlighted, but the *reset_output* of the *rd* component and the *access_event_reset* of the *tdas* component are connected to each of the main system reset inputs, which are primarily comprised of Avalon MM interfaces in this system. You should notice that the *reset_event* input on the *rec* component is only connected to the *access_event_reset* reset output. This means that the Reset Event Counter will only count resets that are generated by the Trivial Default Avalon Slave component.

Figure 9: Trivial Default Avalon Slave Parameters

The Trivial Default Avalon Slave component has a number of parameters that you can implement.

- **Number of Data Bytes** - this parameter allows you to specify how many byte lanes you want the component to expose on the slave interface. There is no requirement for this to be anything in particular, but you can optimize the interconnect fabric that is generated by Qsys by aligning the size of this slave with the masters that connect to it so that no unnecessary width adapters are generated for instance.
- **readdata port pattern** - this parameter allows you to specify what data pattern will be returned by the default slave if it responds to a read access. You may only specify 8-bits here, and those 8-bits are repeated in every byte lane of the slave. A useful value to program into this is 0x00 when you connect this default slave to a Nios II processor's instruction master. That way if the Nios II jumps to an undecoded address in the system it will essentially fetch a "call 0x00" instruction which will force it to jump to address 0x00. As long as there is no decoded slave peripheral at location 0x00, the default slave will be accessed again and again provide the same instruction to the Nios II. When developing in the lab this can be a desirable failure mode that allows you to connect to the Nios II processor with a debugger.
- **Enable slave response port** - this parameter enables the Avalon *response* port which allows you to signal read responses back from the slave.
- **response port pattern** - this parameter allows you to specify what the read or write response pattern will be if the *response* or *writeresponse* ports are enabled. There are four options that can be signaled, *OKAY*, *RESERVED*, *SLAVE ERROR* and *DECODE ERROR*.
- **Enable slave writeresponse port** - this parameter enables the Avalon *writeresponse* port that allows you to signal write responses back from the slave.
- **waitrequest Never Responds** - this parameter prevents the default slave from ever responding to any read or write accesses. Any master that reads or writes this slave when this mode is enabled will stall forever. This

state can only be cleared by asserting the *reset* to the component, or asserting the *clear_event* signal if that interface has been enabled.

- **Enable clear_event input** - this parameter enables the *clear_event* conduit interface which allows you to clear any access events that have been captured by this component.
- **Role of clear_event conduit** - this parameter sets the role of the *clear_event* conduit signal. This allows you to connect this conduit directly to another conduit interface within Qsys.
- **Enable access_event_conduit output** - this parameter exposes the *access_event_conduit* interface which is asserted whenever a read or write transaction is received at the default slave. You can clear this event by resetting the component or asserting the *clear_event* conduit input.
- **Role of access_event_conduit** - this parameter sets the role of the *access_event_conduit* conduit signal. This allows you to connect this conduit directly to another conduit interface within Qsys.
- **Enable access_event_reset output** - this parameter exposes the *access_event_reset* interface which is asserted whenever a read or write transaction is received at the default slave. You can clear this reset by resetting the component or asserting the *clear_event* conduit input.
- **Enable access_event_interrupt output** - this parameter exposes the *access_event_interrupt* interface which is asserted whenever a read or write transaction is received at the default slave. You can clear this interrupt by resetting the component or asserting the *clear_event* conduit input.

PLL Reset Monitor Component

The PLL Reset Monitor component is designed to reset and monitor the locked state of an Altera PLL.

PLL maintenance can be a bit tedious, a PLL can stutter into the locked state, and once locked it may drop out of lock for only a brief instance which requires the PLL to be reset to recover. All PLLs should be reset as the FPGA enters user mode to ensure that the desired PLL configuration is properly achieved as the PLL attempts its initial lock. After you release reset to the PLL you must allow the PLL to potentially stutter in and out of the locked state for the minimum lock period which is generally defined as 1ms for most Altera PLLs. This means that you should ignore lock assertions during this initial lock period as you may see false lock assertions. Some later generation FPGA families are equipped with internal hysteresis that masks the locked output for about 1K cycles of the reference clock, but even then you could witness stuttering in certain situations. Once the initial lock period has expired if the locked output ever deasserts at all, even for a nanosecond, then you must assume that the PLL has lost lock and you must reset the PLL in order to ensure that it reacquires the desired PLL configuration again.

Now in reality, with a properly designed circuit board, power supplies and filtering, and no assembly defects, the behavior of the Altera PLL is generally much more stable and reliable than the previous paragraph may indicate. Most folks never witness any lock stuttering as the PLL exits reset, nor do they ever witness PLL lock failures after lock has been achieved, especially at room temperature in a well controlled lab environment. And it is possible that you can characterize the device to behave within much better tolerances in your specific environmental conditions. Nevertheless, the conditions described above must be considered as worst case possibilities, and you may witness this behavior if anything about your design or assembly or deployment environment are marginal in any way.

The PLL Reset Monitor component attempts to satisfy all of these maintenance requirements for a PLL. The component provides the user with a *pll_reset_request* input that you assert when you wish to reset the PLL. The component then asserts *pll_reset* to the PLL for a user specified duration. Most Altera PLLs require only a 10ns reset pulse to ensure the reset is captured internally. When the component releases *pll_reset* it then monitors the *pll_locked* input to determine if the PLL has gained lock and maintains lock. If the user specified lock wait time expires and the PLL is locked, then the *lock_success_reset* output is asserted, and from that point forward if the PLL ever loses lock, then the *lock_failure_reset* output is asserted. The component also provides an Avalon slave interface so that masters in the Qsys system can examine the status register inside the component. By connecting these signals appropriately in your Qsys system, you should be able to achieve most common reset responses required in environments that manage PLLs. For more details on the component functionality please refer to the comment block at the top of the HDL file that describes the component, or see code Listing 5 below.

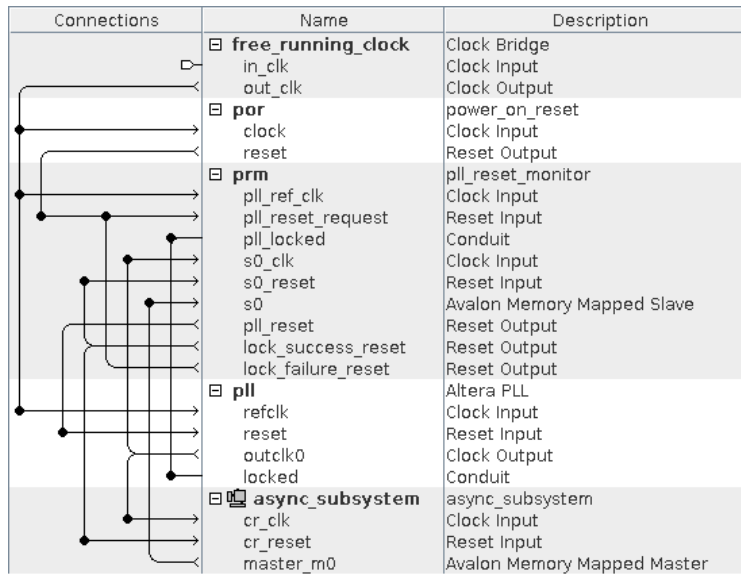


Figure 10: PLL Reset Monitor Qsys System

Please refer to the *prm* instance of the *pll_reset_monitor* component shown in Figure 10. The *pll_ref_clock* should be driven by a free running and stable clock, just like the actual PLL reference clock. This is typically provided by an external clock, not an internal PLL. The *s0_clk* and *s0_reset* should be driven by appropriate clock and reset domains to support the *s0* Avalon slave interface. The other signals on this component will be highlighted and described below.

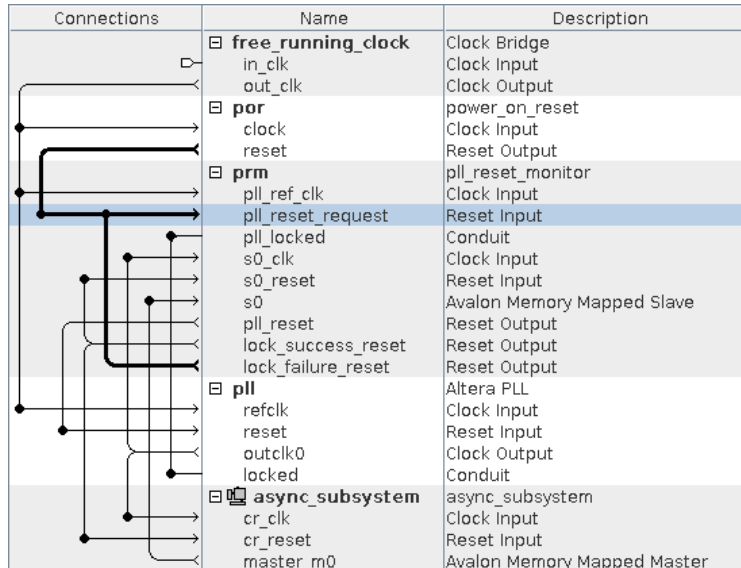


Figure 11: PRM Reset Request Highlight

In Figure 11 we have highlighted the *pll_reset_request* input. You can see that it is driven by the POR reset output and the *lock_failure_reset* output. The POR reset ensures that this component resets the PLL as the FPGA enters user mode, and the *lock_failure_reset* will reset the PLL if a lock failure is ever detected.

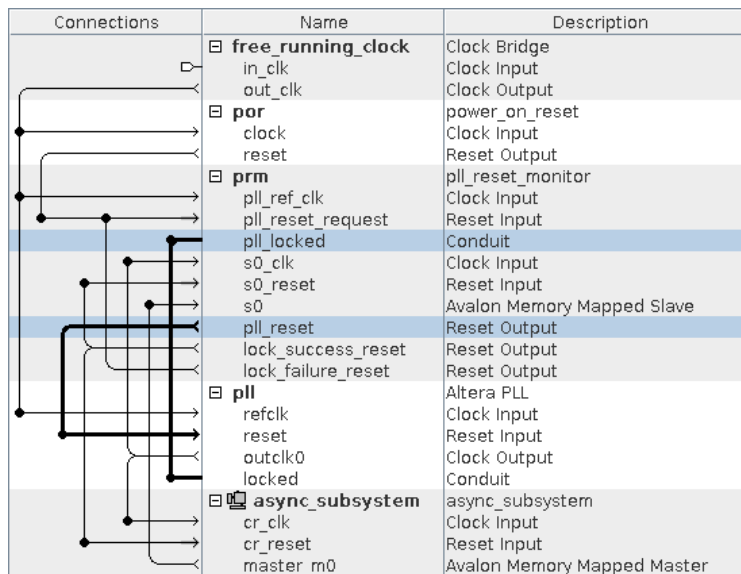


Figure 12: PRM PLL Reset Highlight

In Figure 12 we have highlighted the *pll_reset* output. You can see that it drives the PLL reset input. The *pll_locked* input is also highlighted, it receives the locked indication from the PLL so that the PLL Reset Monitor component can monitor the lock status of the PLL and respond accordingly.

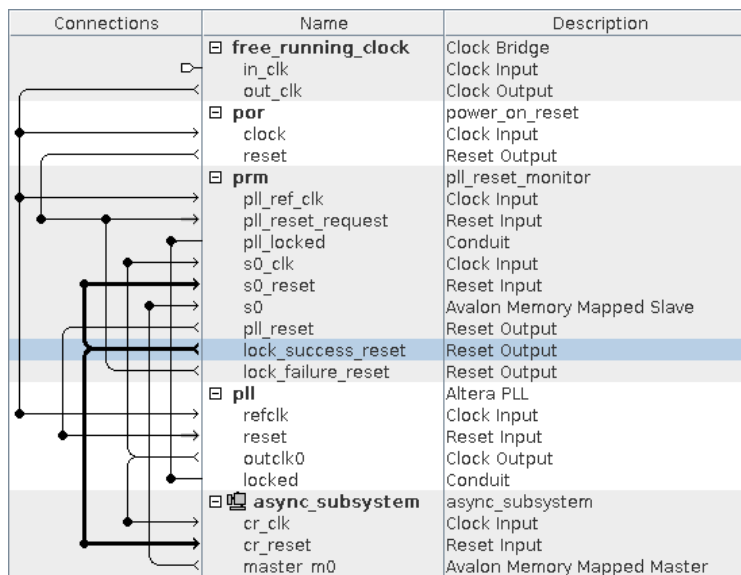


Figure 13: PRM Lock Success Reset Highlight

In Figure 13 we have highlighted the *lock_success_reset* output. You can see that it drives all of the basic Qsys system resets, primarily involving the Avalon MM interfaces. When the PLL Reset Monitor component is reset, this *lock_success_reset* output will be driven low, and when the PLL successfully achieves locked status, this output will be driven high. As you'll see in the next figure, we can instruct Qsys how to interpret the polarity of this as a reset interface, so we can easily configure this reset to hold the main Qsys system elements in reset until the PLL has achieved lock.

pll_reset_monitor
pll_reset_monitor Details

Reset Counter

Reset Counter Width: 5 bits
Reset Counter Duration: 640.0 ns

Lock Counter

Lock Counter Width: 16 bits
Lock Counter Duration: 1310720.0 ns

PLL Locked Input

Role of pll_locked conduit: export

Lock Success Output

☒ Enable lock_success reset output
lock_success reset polarity: active lo reset_n
☐ Enable lock_success conduit output
Role of lock_success conduit: lock_success

Lock Failure Output

☒ Enable lock_failure reset output
lock_failure reset polarity: active hi reset
☐ Enable lock_failure conduit output
Role of lock_failure conduit: lock_failure

Figure 14: PLL Reset Monitor Parameters

The PLL Reset Monitor component has a number of parameters that you can implement.

- **Reset Counter Width** - this parameter allows you to specify the width of the PLL reset duration counter. This will be the duration of the reset pulse that the component delivers to the PLL. If the frequency of the *pll_ref_clk* is known, then this component will also calculate the duration of the PLL reset pulse in nanoseconds for you.
- **Lock Counter Width** - this parameter allows you to specify the width of the PLL lock mask counter. This will be the duration that the component masks or ignores lock stutter after it releases the PLL reset output. If the frequency of the *pll_ref_clk* is known, then this component will also calculate the duration of the PLL lock mask duration in nanoseconds for you.
- **Role of pll_locked conduit** - this parameter sets the role of the *pll_locked* conduit signal. This should allow you to align the role of this conduit interface with the role of the locked conduit from the PLL so that you can connect the two conduits within the Qsys system.
- **Enable lock_success reset output** - this parameter exposes the lock success indication as a reset interface so that you can connect this to other reset interfaces within the Qsys system.
- **lock_success reset polarity** - this parameter sets the polarity of the lock success reset interface. This does not change the behavior of the *lock_success_reset* output, it simply changes the way Qsys interprets the signal as either an active high “reset” interface or as an active low “reset_n” interface. When this component is reset, this interface will drive low and when PLL lock is successfully achieved, this interface will drive high, regardless of how this parameter is set.
- **Enable lock_success conduit output** - this parameter exposes the lock success indication as a conduit interface so that you can connect this to other conduit interfaces within the Qsys system.
- **Role of lock_success conduit** - this parameter sets the role of the lock success conduit signal. This allows you to connect this conduit directly to another conduit interface within Qsys.
- **Enable lock_failure reset output** - this parameter exposes the lock failure indication as a reset interface so that you can connect this to other reset interfaces within the Qsys system.
- **lock_failure reset polarity** - this parameter sets the polarity of the lock failure reset interface. This does not change the behavior of the *lock_failure_reset* output, it simply changes the way Qsys interprets the signal

as either an active high “reset” interface or as an active low “reset_n” interface. When this component is reset, this interface will drive low and when a PLL lock failure is detected, this interface will drive high, regardless of how this parameter is set.

- **Enable lock_failure conduit output** - this parameter exposes the lock failure indication as a conduit interface so that you can connect this to other conduit interfaces within the Qsys system.
- **Role of lock_failure conduit** - this parameter sets the role of the lock failure conduit signal. This allows you to connect this conduit directly to another conduit interface within Qsys.

Reset Assertion Delay Component

The Reset Assertion Delay component is designed to implement a user specified delay between two reset output interfaces. The first reset output is asserted immediately as this component detects a reset input event, and the delayed reset output is asserted at some user specified delay later. This is useful in situations where you may need to reset a PLL for instance but you need to place other system logic into reset before you reset the PLL. This can arise in situations where you may have synchronous reset domains, or other situations that cannot tolerate the loss of a clock during the PLL reset event. Most Altera PLLs will disable the clock outputs while they are in the reset state. For more details on the component functionality please refer to the comment block at the top of the HDL file that describes the component, or see code Listing 6 below.

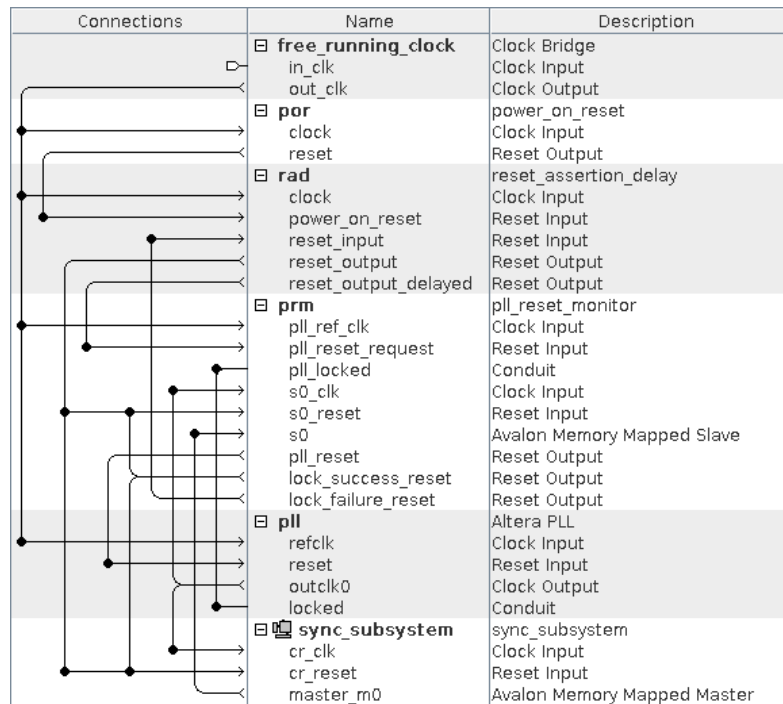


Figure 15: Reset Assertion Delay Qsys System

Please refer to the *rad* instance of the *reset_assertion_delay* component shown in Figure 15. The *clock* should be driven by a free running and stable clock, typically provided by an external clock, not an internal PLL. The other signals on this component will be highlighted and described below.

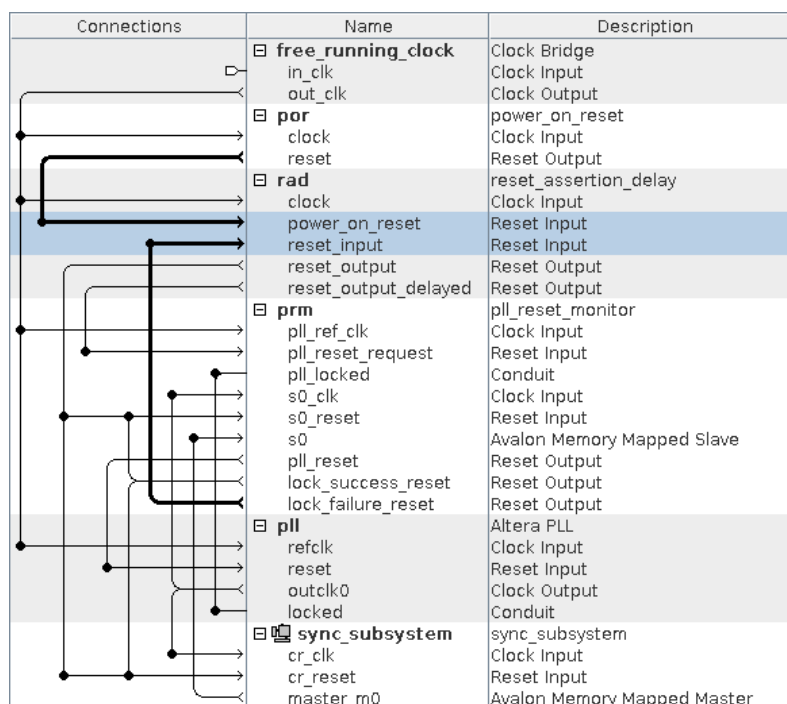


Figure 16: RAD Reset Inputs Highlight

In Figure 16 we have highlighted the *power_on_reset* input and the *reset_input* ports. You can see that these are driven by the POR reset output and the *lock_failure_reset* output from the PLL Reset Monitor that we described above. The POR reset ensures that this component asserts both reset outputs as the FPGA enters user mode, and the *lock_failure_reset* will trigger the reset assertion delay sequence if a lock failure is ever detected.

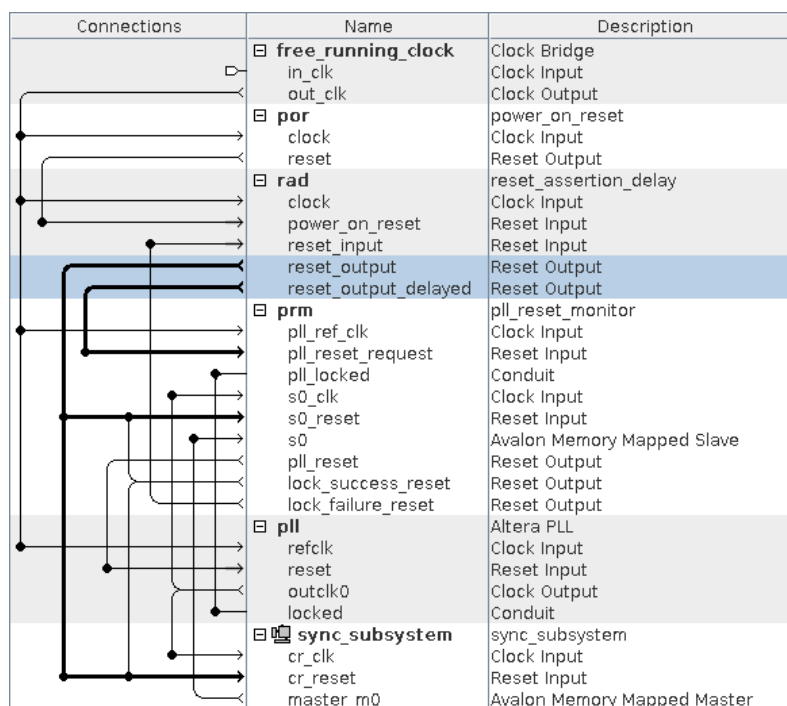
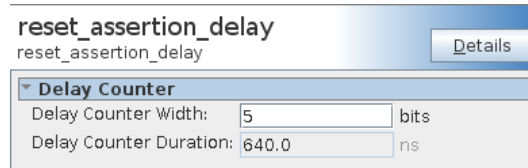


Figure 17: RAD Reset Outputs Highlight

In Figure 17 we have highlighted the *reset_output* and *reset_output_delayed* outputs. You can see that the *reset_output* signal drives the reset inputs of the main system components that we want to place into reset prior to

resetting the PLL. And the *reset_output_delayed* signal drives the PLL Reset Monitor *pll_reset_request* input to reset the PLL.

Please note that in this Reset Assertion Delay Qsys system we have changed the main subsystem name shown at the bottom of the image from *async_subsystem* as it has been named in all of the previous component example systems to *sync_subsystem*. This is intended to imply that the reset requirements inside that Qsys subsystem are synchronous rather than asynchronous. You can easily create this reset requirement in a Qsys system by simply inserting an OnChip Memory component, as those components require synchronous resets. You can identify Qsys components that require synchronous resets by the presence of the *reset_req* Avalon port on the reset interface.



reset_assertion_delay	
reset_assertion_delay	
Details	
▼ Delay Counter	
Delay Counter Width:	5 bits
Delay Counter Duration:	640.0 ns

Figure 18: Reset Assertion Delay Parameters

The PLL Reset Monitor component has one parameter that you specify, the delay counter width. This sets the duration of the delay between the assertion from the *reset_output* signal to the *reset_output_delayed* signal. If the frequency of the *clock* is known, then this component will also calculate the duration of the delay in nanoseconds for you.

Event Timer Component

The Event Timer component is designed to provide more complex sequencing of system resets that may require a delay to allow for certain logic blocks to initialize or calibrate themselves before moving along with the system reset sequencing requirements. This situation can arise in Qsys systems that implement EMIF controllers where there is an integrated PLL which must be managed and the controller logic goes through a calibration phase and an initialization phase before the controller is ready for use. In the example EMIF system that we show below, we assume that the user wants to hold off the main system reset release until the EMIF controller has achieved PLL lock, successfully calibrated and then successfully initialized itself. To accomplish this we implement two Event Timer components, one that monitors the calibration status and one that monitors the initialization status. Fundamentally what the Event Timer component does is monitor the state of an event input to detect if the event has asserted within a user specified period of time otherwise a timeout condition occurs and a timeout output indication asserts. If the event input asserts within the timeout period then an acquired output indication asserts and the event input is further monitored in case it ever deasserts. If the event input deasserts after a successful assertion, then a loss output indication asserts. For more details on the component functionality please refer to the comment block at the top of the HDL file that describes the component, or see code Listing 7 below.

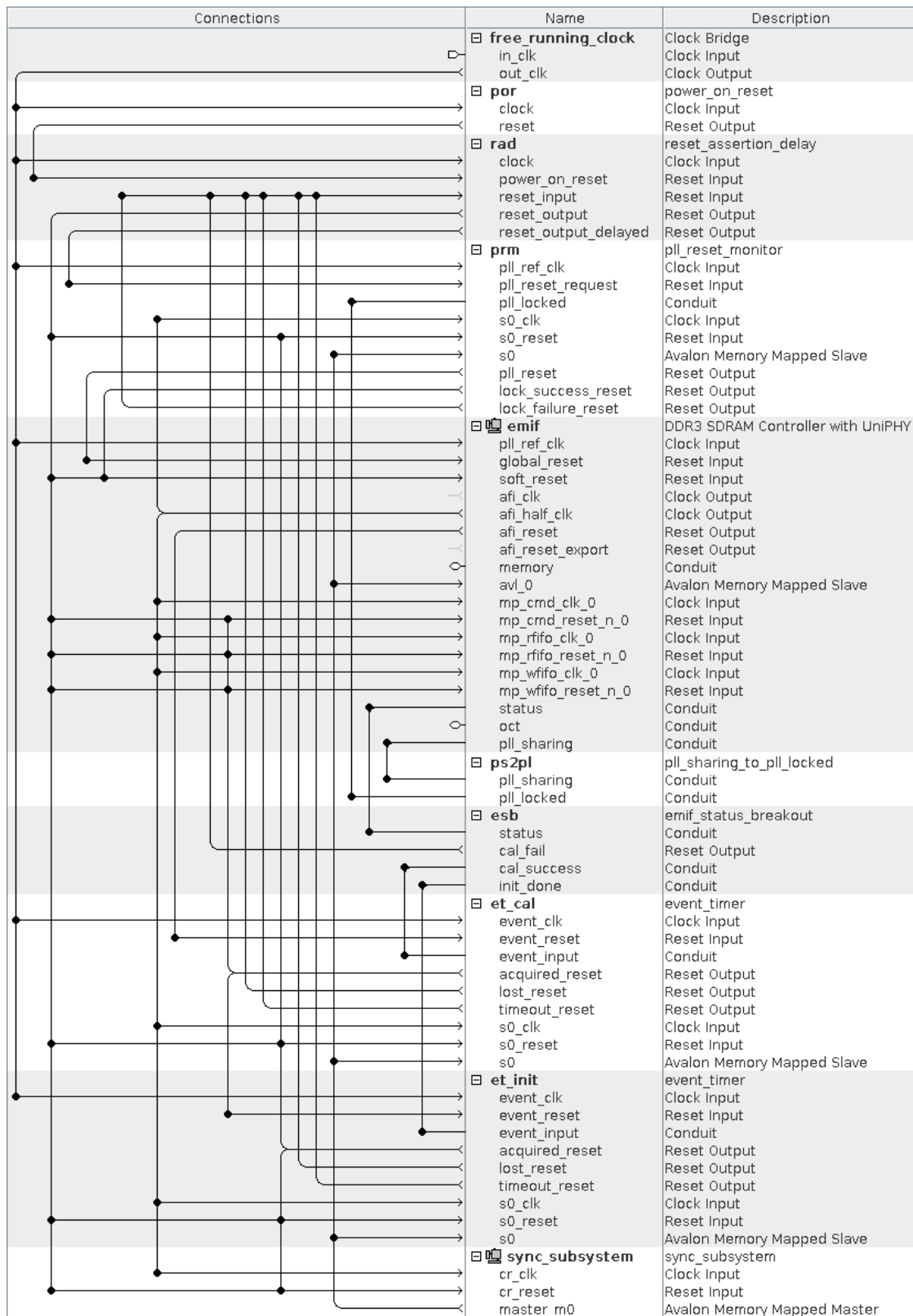


Figure 19: Event Timer Qsys System

Please refer to the *et_cal* instance of the *event_timer* component shown in Figure 19. The *event_clk* should be driven by a free running and stable clock, typically provided by an external clock, not an internal PLL. The *event_reset* should be driven by an appropriate reset output, typically driven by an appropriate stage of the overall system reset sequencing logic. The Event Timer component can only be cleared or restarted by a reset on the

event_reset port. The *acquired_reset*, *lost_reset* and *timeout_reset* are the outputs that represent the event acquisition state, event lost state and the event timeout state respectively. These outputs should be wired into the Qsys system to affect the desired sequencing. A more detailed discussion of the reset sequencing strategy in this example will be shown below. The *s0_clk* and *s0_reset* should be driven by appropriate clock and reset domains to support the *s0* Avalon slave interface. The Avalon slave interface allows masters in the system to query the status register in the component.

Figure 20: Event Timer Parameters

The Event Timer component has a number of parameters that you can implement.

- **Timeout Counter Width** - this parameter allows you to specify the width of the timeout duration counter. This will be the duration that the component waits after it exits the reset state before it declares a timeout condition. If the frequency of the *event_clk* is known, then this component will also calculate the duration of the delay in nanoseconds for you.
- **Role of event_input conduit** - this parameter sets the role of the *event_input* conduit signal. This allows you to connect this conduit directly to another conduit interface within Qsys.
- **Enable acquired reset output** - this parameter exposes the acquired indication as a reset interface so that you can connect this to other reset interfaces within the Qsys system.
- **acquired reset polarity** - this parameter sets the polarity of the acquired reset interface. This does not change the behavior of the *acquired_reset* output, it simply changes the way Qsys interprets the signal as either an active high “reset” interface or as an active low “reset_n” interface. When this component is reset, this interface will drive low and when the acquired state is achieved, this interface will drive high, regardless of how this parameter is set.
- **Enable acquired conduit output** - this parameter exposes the acquired indication as a conduit interface so that you can connect this to other conduit interfaces within the Qsys system.
- **Role of acquired conduit** - this parameter sets the role of the acquired conduit signal. This allows you to connect this conduit directly to another conduit interface within Qsys.
- **Enable lost reset output** - this parameter exposes the lost indication as a reset interface so that you can connect this to other reset interfaces within the Qsys system.

- **lost reset polarity** - this parameter sets the polarity of the lost reset interface. This does not change the behavior of the *lost_reset* output, it simply changes the way Qsys interprets the signal as either an active high “reset” interface or as an active low “reset_n” interface. When this component is reset, this interface will drive low and when the lost state is achieved, this interface will drive high, regardless of how this parameter is set.
- **Enable lost conduit output** - this parameter exposes the lost indication as a conduit interface so that you can connect this to other conduit interfaces within the Qsys system.
- **Role of lost conduit** - this parameter sets the role of the lost conduit signal. This allows you to connect this conduit directly to another conduit interface within Qsys.
- **Enable timeout reset output** - this parameter exposes the timeout indication as a reset interface so that you can connect this to other reset interfaces within the Qsys system.
- **timeout reset polarity** - this parameter sets the polarity of the timeout reset interface. This does not change the behavior of the *timeout_reset* output, it simply changes the way Qsys interprets the signal as either an active high “reset” interface or as an active low “reset_n” interface. When this component is reset, this interface will drive low and when the timeout state is achieved, this interface will drive high, regardless of how this parameter is set.
- **Enable timeout conduit output** - this parameter exposes the timeout indication as a conduit interface so that you can connect this to other conduit interfaces within the Qsys system.
- **Role of timeout conduit** - this parameter sets the role of the timeout conduit signal. This allows you to connect this conduit directly to another conduit interface within Qsys.

EMIF System Example Explanation

Now let’s take a deeper look at the architecture presented in the EMIF example from Figure 19. This system leverages most of the components that have been described above in this HOWTO document. For more information on any of these supplemental reset components please refer to the appropriate section of this document.

Connec...	Name	Description
	free_running_clock	Clock Bridge
	in_clk	Clock Input
	out_clk	Clock Output
	por	power_on_reset
	clock	Clock Input
	rad	reset_assertion_delay
	clock	Clock Input
	prm	pll_reset_monitor
	pll_ref_clk	Clock Input
	s0_clk	Clock Input
	emif	DDR3 SDRAM Controller with UniPHY
	pll_ref_clk	Clock Input
	afi_clk	Clock Output
	afi_half_clk	Clock Output
	mp_cmd_clk_0	Clock Input
	mp_rfifo_clk_0	Clock Input
	mp_wfifo_clk_0	Clock Input
	ps2pl	pll_sharing_to_pll_locked
	et_cal	event_timer
	event_clk	Clock Input
	s0_clk	Clock Input
	et_init	event_timer
	event_clk	Clock Input
	s0_clk	Clock Input
	sync_subsystem	sync_subsystem
	cr_clk	Clock Input

Figure 21: EMIF System Clocks

In Figure 21 we filtered the view to show only the two clock domains in the system. The free running input clock is driving all of the reset sequencing components and the EMIF PLL reference input. The *afi_half_clk* output from the *emif* instance is driving the rest of the clock inputs in the system.

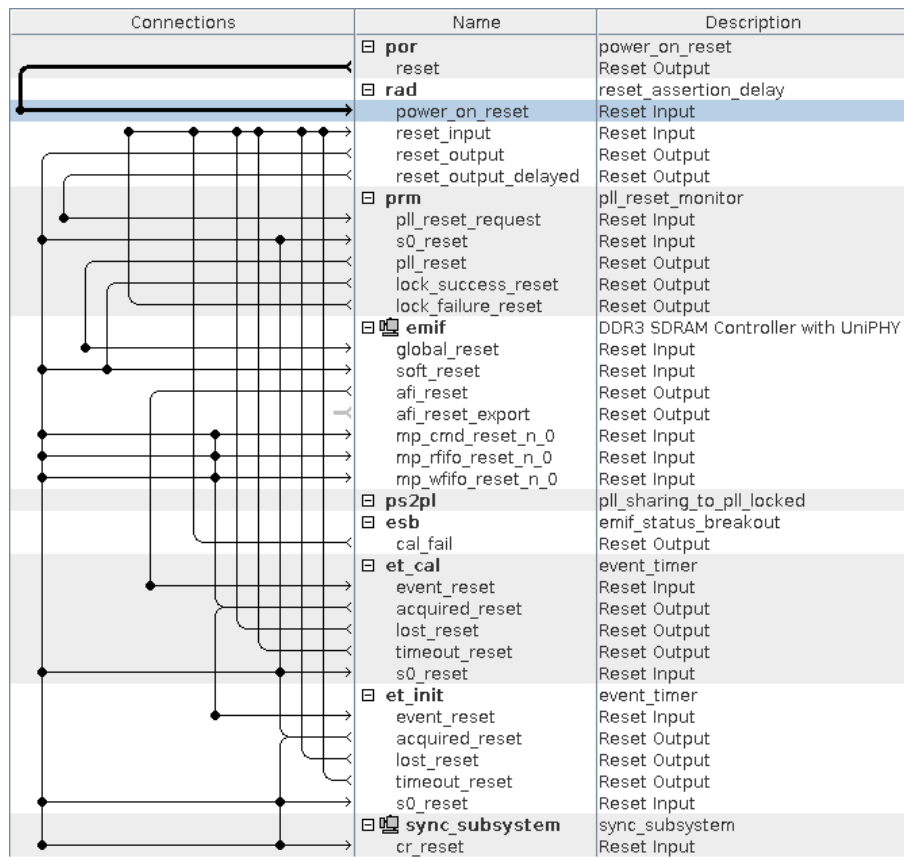


Figure 24: EMIF Reset Stage 1

In Figure 24 we highlight the POR reset that ensures that the Reset Assertion Delay component is reset as the FPGA enters user mode. The outputs of the Reset Assertion Delay component will place everything else in the system into reset.

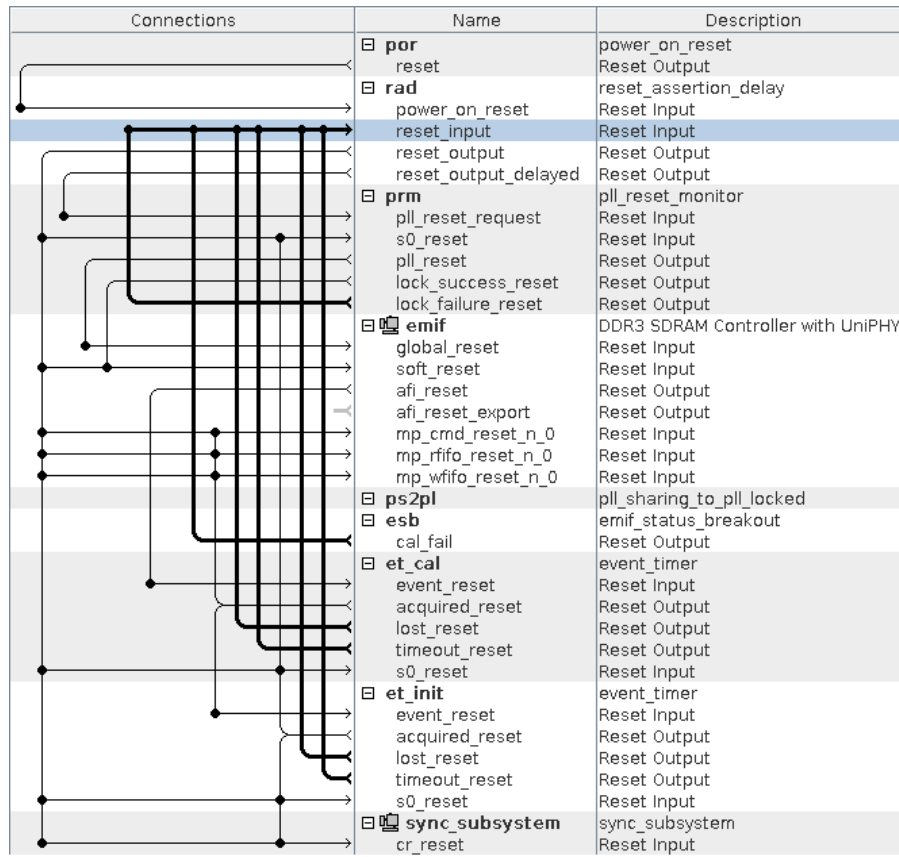


Figure 25: EMIF Reset Stage 2

In Figure 25 we highlight the *reset_input* of the Reset Assertion Delay component. This can be considered the main reset for the system, any of the six reset sources that drive this interface can force a complete system reset sequence to occur. When this reset is asserted the outputs of the Reset Assertion Delay component will place everything else in the system into reset. The six sources that can cause the complete system reset are as follow:

- *prm/lock_failure_reset* - the lock failure indicator from the PLL Reset Monitor component will trigger the complete system reset when a PLL lock failure is detected in the EMIF PLL.
- *esb/cal_fail* - the EMIF calibration failure indicator from the EMIF component will trigger the complete system reset when an EMIF calibration failure is signaled.
- *et_cal/lost_reset* - the EMIF calibration loss indicator from the EMIF calibration Event Timer will trigger the complete system reset when an EMIF calibration success indication is negated.
- *et_cal/timeout_reset* - the EMIF calibration timeout indicator from the EMIF calibration Event Timer will trigger the complete system reset when an EMIF calibration success is not achieved within the specified timeout period.
- *et_init/lost_reset* - the EMIF initialization loss indicator from the EMIF initialization Event Timer will trigger the complete system reset when an EMIF initialization success indication is negated.
- *et_init/timeout_reset* - the EMIF initialization timeout indicator from the EMIF initialization Event Timer will trigger the complete system reset when an EMIF initialization success is not achieved within the specified timeout period.

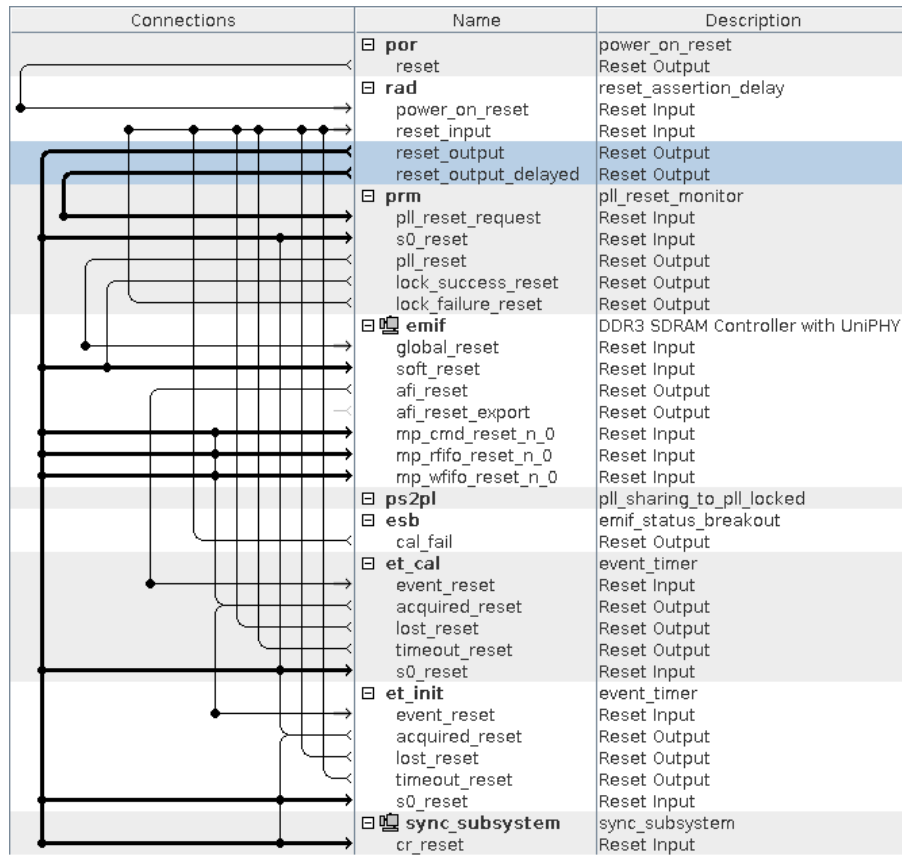


Figure 26: EMIF Reset Stage 3

In Figure 26 we highlight the *reset_output* interface of the Reset Assertion Delay component which will be asserted immediately when the *reset_input* interface is driven active and will place all of the main system level resets into reset ahead of the PLL reset. The *reset_output_delayed* interface of the Reset Assertion Delay component is also highlighted. This reset will be asserted after the user specified delay has passed after the *reset_output* assertion, and this interface will reset the PLL by driving the PLL Reset Monitor component.

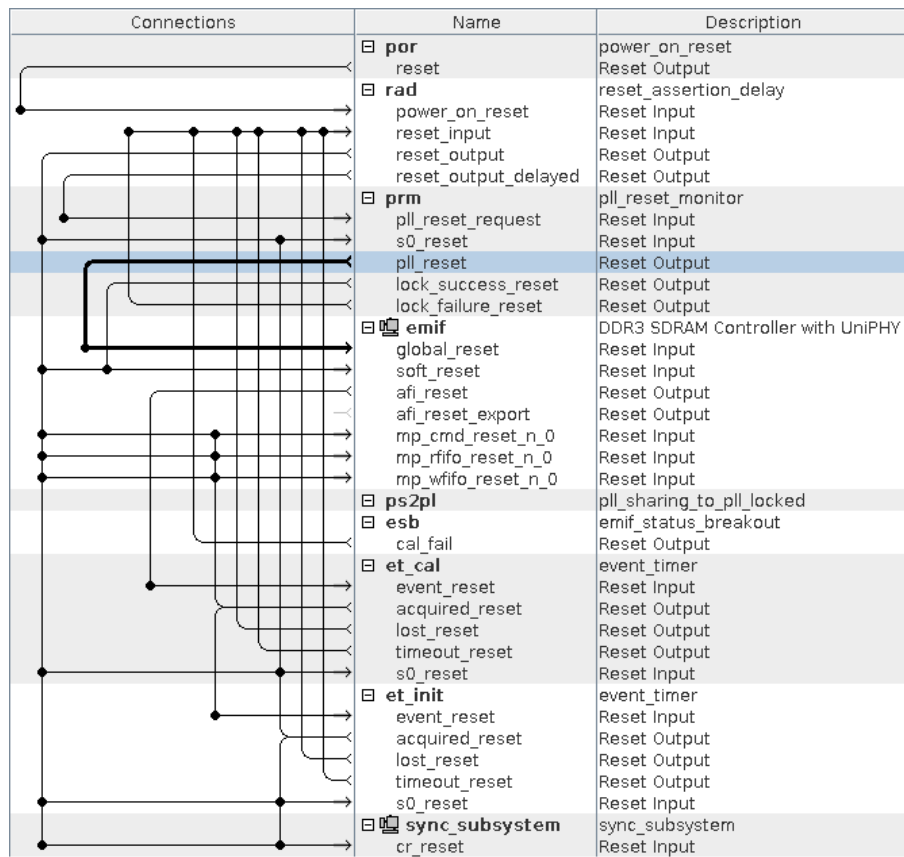


Figure 27: EMIF Reset Stage 4

In Figure 27 we highlight the *pll_reset* interface of the PLL Reset Monitor component. This resets the PLL in the EMIF component through the *global_reset* interface.

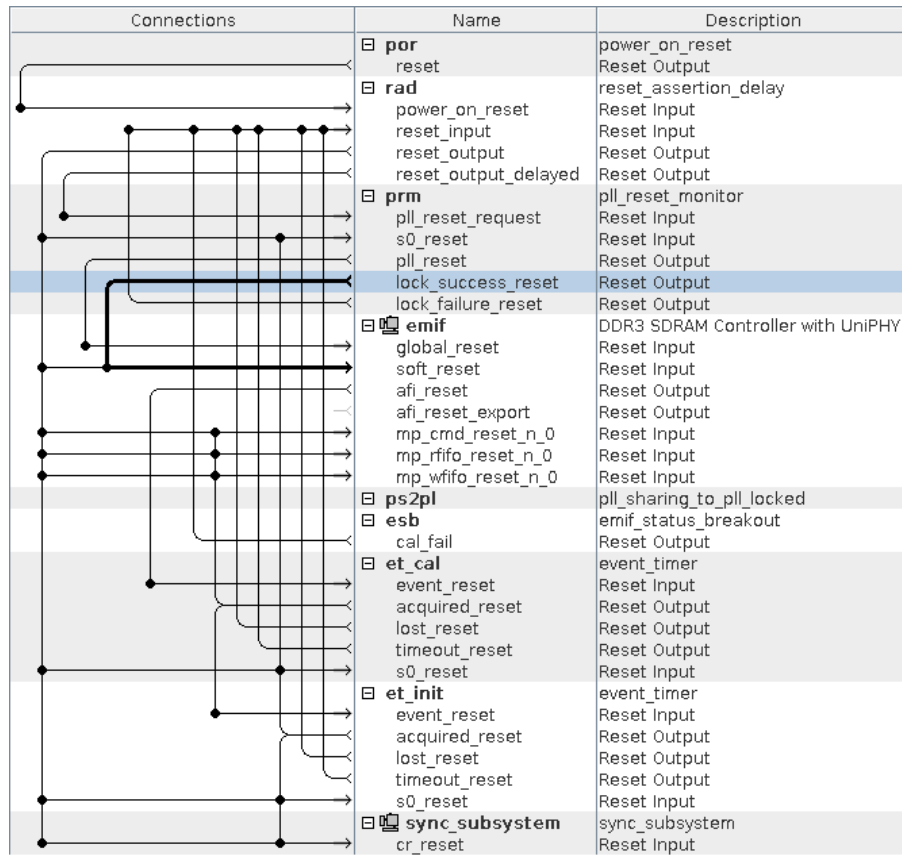


Figure 28: EMIF Reset Stage 5

In Figure 28 we highlight the *lock_success_reset* interface of the PLL Reset Monitor component. After the PLL has been reset, we would expect that it will eventually achieve lock. Once the PLL lock is achieved, then this reset output will release the *soft_reset* interface on the EMIF component which allows the EMIF controller to begin it's calibration phase.

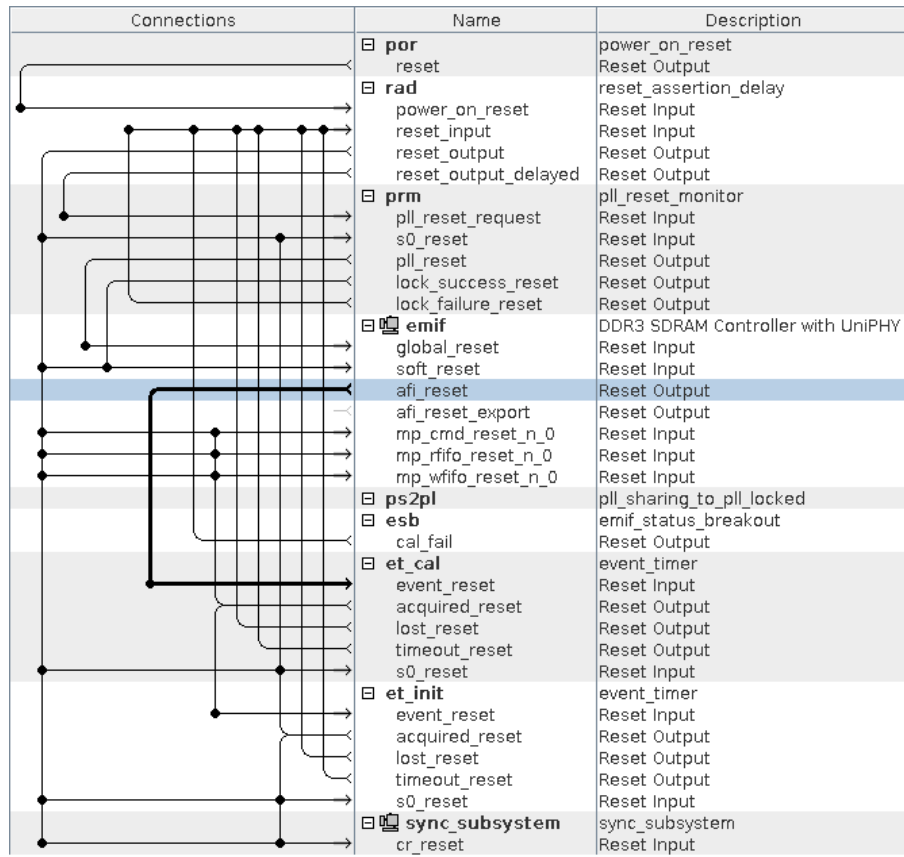


Figure 29: EMIF Reset Stage 6

In Figure 29 we highlight the *afi_reset* interface of the EMIF component. This reset will release immediately once there is no reset assertions on the *global_reset* interface or the *soft_reset* interface. When this reset releases it allows the *et_cal* Event Timer component to begin timing the calibration sequence.

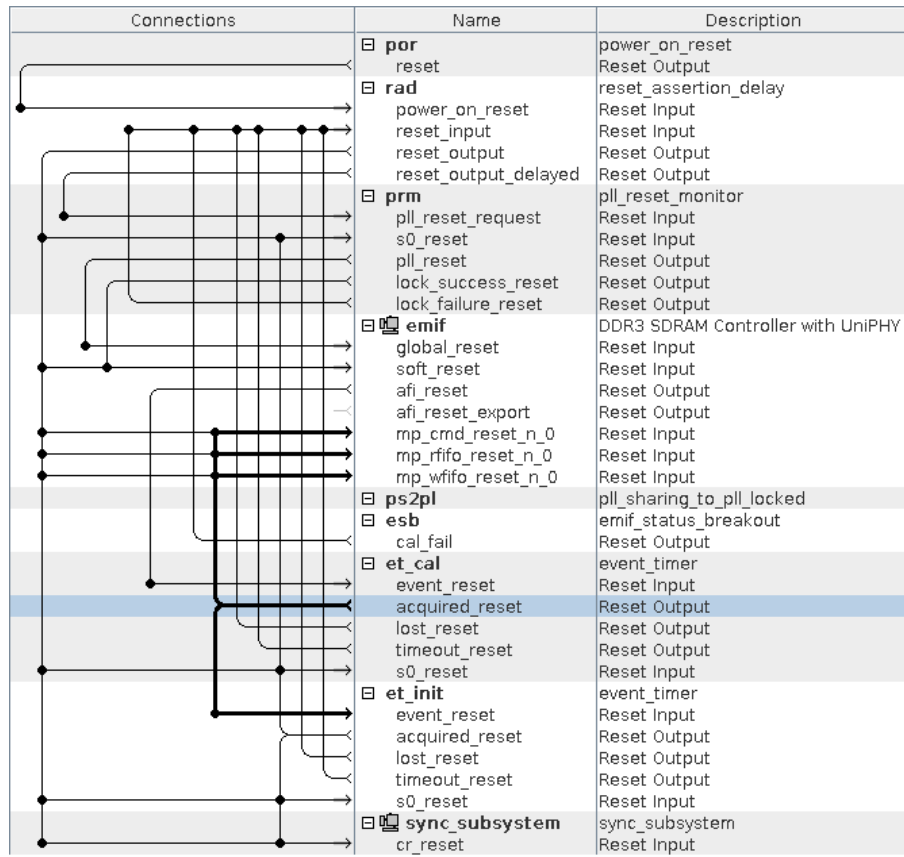


Figure 30: EMIF Reset Stage 7

In Figure 30 we highlight the *acquired_reset* interface of the EMIF calibration Event Timer component. When the EMIF calibration indication is successfully captured within the timeout period, this reset will release the *mp_*fifo_reset_n_0* interfaces on the EMIF component which allows the EMIF to enter its initialization phase. At the same time the *et_init* Event Timer component is released and can begin timing the initialization sequence.

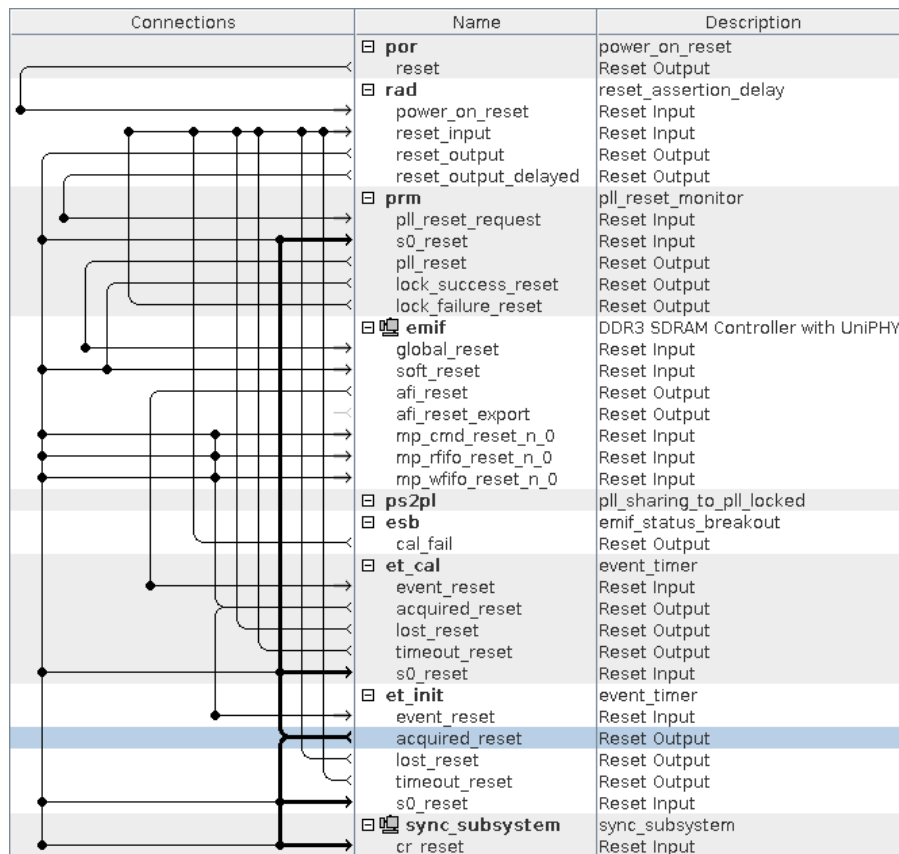


Figure 31: EMIF Reset Stage 8

In Figure 31 we highlight the *acquired_reset* interface of the EMIF initialization Event Timer component. When the EMIF initialization indication is successfully captured within the timeout period, this reset will release the system wide reset domain that covers all the Avalon MM interfaces and the *sync_subsystem*. At this point, our entire system has sequenced out of reset and it will run uninterrupted unless one of the six system reset request events occur that drive the *reset_input* of the Reset Assertion Delay component.

Reset Until Ack Component

The RUA (Reset Until Ack) component is a very trivial component that is intended to assert a reset output signal when a reset assert input signal is active, then hold the reset output asserted until the reset assert input signal is inactive and the reset release input signal is active. Fundamentally it operates like a simple latch. For more details on the component functionality please refer to the comment block at the top of the HDL file that describes the component, or see code Listing 8 below.

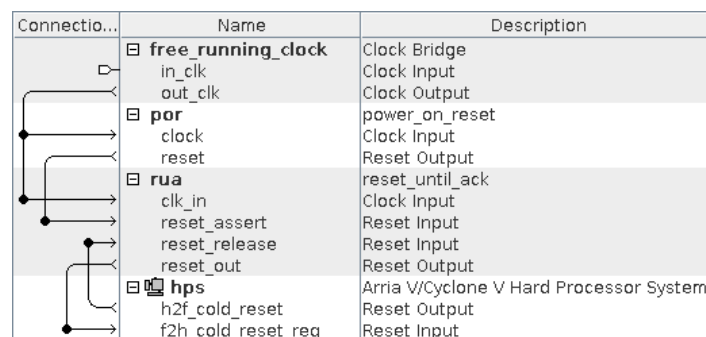


Figure 32: RUA Qsys System

Please refer to the *rua* instance of the *reset_until_ack* component shown in Figure 32. The RUA component requires a free running and stable *clk_in* input, typically provided by an external clock, not an internal PLL. The RUA component provides two reset inputs, one that asserts the reset output and another that releases the reset output. When the *reset_assert* signal is driven active, the *reset_out* signal is asserted and held until the *reset_assert* signal is driven inactive and the *reset_release* signal is driven active, at which point the *reset_out* signal is deasserted. This allows a brief reset pulse to be extended into a given reset domain until a positive acknowledgment is received that indicates the reset was taken. In the example shown above in Figure 32, you can see that we have a POR reset drive the *reset_assert* input of the RUA component and then the *reset_out* drives into the HPS subsystem to request a cold reset. The cold reset indication of the HPS subsystem is fed back into the *reset_release* to acknowledge that it has taken the cold reset which was requested.

The RUA component has no parameters that affect its configuration or operation.

Component HDL Comment Blocks

This section contains the HDL comment block listings from each of the Supplemental Reset Components for Qsys.

Power On Reset Component

Listing 1: power_on_reset.v

```
locate the output registers of the component with something like this:
[get_registers {*power_on_reset:*|output_reg}]

*/
`timescale 1 ps / 1 ps
module power_on_reset #(
    parameter POR_COUNT = 20          // MUST BE 2 or greater
) (
    input  wire  clk,
    output wire  reset
);

wire sync_dout;
altera_std_synchronizer #(
    .depth (POR_COUNT)
) power_on_reset_std_sync_inst (
    .clk      (clk),
    .reset_n  (1'b1),
    .din      (1'b1),
    .dout     (sync_dout)
);
```

Reset Debouncer Component

Listing 2: reset_debouncer.v

assertion and deassertion counts of the reset_input are counted and provided through the slave interface. Also at each deassertion of the reset_input signal the debounce_count counter is captured and provided through the slave interface. NOTE: only the each initial assertion edge is counted, but then each deassertion edge is counted as the debounce counter runs and the value captured for the debounce_count counter is the final occurrence of deassertion that was captured. Also, each counter will stop incrementing when it reaches its maximal_count, it will not wrap.

The format of the 32-bits in the slave register are as follows:

```
assertion_edge_count[31:29] - 3-bit field
deassertion_edge_count[28:24] - 5-bit field
```

```
capture_debounce_count [23:0] - 24-bit field
```

The SDC file that accompanies this component provides constraints that cut the asynchronous input paths of the altera_reset_synchronizer instance.

To constrain the outputs of this component in your own SDC constraints, you can locate the output registers of the component with something like this:

```
[get_registers {*reset_debouncer:*|reset_output_reg}]
```

```
*/
'timescale 1 ps / 1 ps
module reset_debouncer #(
    parameter DEBOUNCE_COUNTER_WIDTH = 16
) (
    input wire clk,
    input wire power_on_reset,

    input wire reset_input,
    output wire reset_output,

    input wire s0_clk,
    input wire s0_reset,
    input wire s0_read,
    output wire [31:0] s0_readdata
);
```

Reset Event Counter Component

Listing 3: reset_event_counter.v

There is an Avalon Slave interface on this component that provides access to read the values of the two counts that are captured in this component.

The format of the 32-bits in the slave register are as follows:

```
deassertion_count_sync [31:16] - 16-bit field
assertion_count_sync   [15:0]  - 16-bit field
```

The SDC file that accompanies this component provides constraints that cut the asynchronous input paths of the altera_reset_synchronizer instance.

```
*/
'timescale 1 ps / 1 ps
module reset_event_counter #(
    parameter COUNTER_WIDTH = 16
) (
    input wire clk,
    input wire power_on_reset,
    input wire reset_event,

    input wire s0_clk,
    input wire s0_reset,
    input wire s0_read,
    output wire [31:0] s0_readdata
);

// asynchronously capture and synchronize the power_on_reset
wire power_on_reset_sync;
```

Trivial Default Avalon Slave Component

Listing 4: trivial_default_avalon_slave.v

```
11: DECODEERROR
```

The response ports can be suppressed by setting the NO_RESPONSE_PORT parameter provided in the hw.tcl script.

The slave can be set to always respond, or never respond by using the parameter NEVER_RESPOND.

There is an access_event indication that is provided on three different outputs, an interrupt, a reset and a conduit interface. Each of these interfaces can be suppressed or enabled with an hw.tcl parameter. The access_event will be cleared with a reset or the assertion of the clear_event input.

The clear_event input is an optional conduit input that can be used to clear the access_event indicators and release the waitrequest port in NEVER_RESPOND mode. The rising edge of the clear_event input is the trigger for clearing the event. This input is asynchronously captured and synchronized internally so it must be asserted for a minimum of 2 clock cycles. There is an hw.tcl parameter that can be used to suppress this interface.

```
*/
`timescale 1 ps / 1 ps
module trivial_default_avalon_slave #(
    parameter DATA_BYTES = 1,
    parameter READ_DATA_PATTERN = 0,
    parameter RESPONSE_PATTERN = 0,
    parameter NEVER_RESPOND = 0
) (
    input wire clock_clk,
    input wire reset_reset,

    input wire slave_read,
    output wire [((DATA_BYTES * 8) - 1):0] slave_readdata,
    output wire slave_readdatavalid,
    output wire [1:0] slave_response,
    output wire slave_writeresponsevalid,
    input wire slave_write,
```

PLL Reset Monitor Component

Listing 5: pll_reset_monitor.v

for PLL lock acquisition, so you should select the value of LOCK_COUNTER_WIDTH to produce a period near that timeframe. At the expiration of the lock_count the state of the pll_locked input is evaluated to produce a lock_success or lock_failure output, and from that point on if the pll_locked glitches or drops for any amount of time the lock_failure output should be asserted. This circuit allows the PLL lock to stutter into the locked state and then monitor for even a small asynchronous glitch that indicates the loss of lock.

To achieve this behavior the pll_locked input is passed through the altera_reset_synchronizer core to asynchronously detect glitches and synchronize the pll_locked input into the proper clock domain.

There is an Avalon Slave interface on this component that provides access to

read the values of two counts that are captured in this component. First, each time the `pll_locked` input is asserted, the `lock_count` value is captured into a register which can be read through the slave. Second, each time the `pll_locked` input is deasserted, a `unlock_count` counter is incremented and provided thru the slave interface as well. The `lock_counter_width` field contains the value of 19 minus the `LOCK_COUNTER_WIDTH` parameter.

The format of the 32-bits in the slave register are as follows:

```
unlock_count_sync      [31:24] - 8-bit field
lock_counter_width     [23:20] - 4-bit field
capture_lock_count_sync [19:0]  - 20-bit field
```

The SDC file that accompanies this component provides constraints that cut the asynchronous input paths of the `altera_reset_synchronizer` instance.

To constrain the outputs of this component in your own SDC constraints, you can locate the output registers of the component with something like this:

```
[get_registers {*pll_reset_monitor:*|delayed_pll_reset}]
[get_registers {*pll_reset_monitor:*|lock_success_reg}]
[get_registers {*pll_reset_monitor:*|lock_failure_reg}]
```

```
*/
`timescale 1 ps / 1 ps
module pll_reset_monitor #(
    parameter RESET_COUNTER_WIDTH = 10,
    parameter LOCK_COUNTER_WIDTH  = 16
) (
    input  wire      pll_ref_clk,
    input  wire      pll_reset_request,
    input  wire      pll_locked,

    output wire      pll_reset,
    output wire      lock_success,
    output wire      lock_success_reset,
    output wire      lock_failure,
    output wire      lock_failure_reset,

    input  wire      s0_clk,
```

Reset Assertion Delay Component

Listing 6: `reset_assertion_delay.v`

To constrain the outputs of this component in your own SDC constraints, you can locate the output registers of the component with something like this:

```
[get_registers {*reset_assertion_delay:*|reset_output_reg}]
[get_registers {*reset_assertion_delay:*|delay_count_expired}]
```

```
*/
`timescale 1 ps / 1 ps
module reset_assertion_delay #(
    parameter DELAY_COUNTER_WIDTH = 5
) (
    input  wire      clk,
    input  wire      power_on_reset,

    input  wire      reset_input,
    output wire      reset_output,
```

```

        output wire          reset_output_delayed
    );

    // capture glitches and synchronize the input
    wire reset_input_capture;
    altera_reset_synchronizer #(
        .ASYNC_RESET (1),

```

Event Timer Component

Listing 7: event_timer.v

There is an Avalon Slave interface on this component that provides access to read the value of the timeout_count counter value that is captured at the assertion of the event_input signal.

The format of the 32-bits in the slave register are as follows:

```
capture_timeout_count_sync [31:0] - 32-bit field
```

To constrain the outputs of this component in your own SDC constraints, you can locate the output registers of the component with something like this:

```

[get_registers {*event_timer:*|timeout_count_expired}]
[get_registers {*event_timer:*|event_acquired}]
[get_registers {*event_timer:*|event_lost}]

```

```

*/
`timescale 1 ps / 1 ps
module event_timer #(
    parameter TIMEOUT_COUNTER_WIDTH = 16
) (
    input  wire      event_clk,
    input  wire      event_reset,
    input  wire      event_input,

    output wire      timeout,
    output wire      timeout_reset,
    output wire      acquired,
    output wire      acquired_reset,
    output wire      lost,
    output wire      lost_reset,

    input  wire      s0_clk,

```

Reset Until Ack Component

Listing 8: reset_until_ack.v

```

        output wire  reset_out_reset
    );

    reg reset_out;

    assign reset_out_reset = reset_out;

    always @ (posedge clk_in_clk or posedge reset_assert_reset) begin
        if(reset_assert_reset) begin

```