9. Optimizing Qsys System Performance

QII51024-13.0.0

This chapter provides information on optimizing system interconnect performance for designs generated by the Altera® Qsys system integration tool.

The foundation of any large system is the interconnect logic used to connect hardware blocks or components. Creating interconnect logic is prone to errors, is time consuming to write, and is difficult to modify when design requirements change. The Qsys system integration tool addresses these issues by providing an automatically generated and optimized interconnect designed to satisfy your system requirements.

Qsys supports standard Avalon®, AMBA® AXI3™ (version 1.0), AMBA AXI4™ (version 2.0), and AMBA APB™ 3 (version 1.0) interfaces. For more information about Avalon and AMBA interfaces, refer to the *Avalon Interface Specifications* and the *AMBA Protocol Specifications* on the ARM® website. AXI4-Lite is not supported.

For more discussion about determining which interface standard you want to use to create your Qsys design, refer to the *Creating a System With Qsys* chapter in volume 1 of the *Quartus II Handbook*.

Following the design practices recommended in this chapter may improve the clock frequency, throughput, logic utilization, or power consumption of your Qsys design. When you design a Qsys system, use your knowledge of your design intent and goals to further optimize system performance beyond the automated optimization available in Qsys.

The following sections describe Qsys support for optimization of interconnect logic:

- "Designing with Avalon and AXI Interfaces" on page 9–1
- "Using Hierarchy in Systems" on page 9–3
- "Using Concurrency in Memory-Mapped Systems" on page 9–5
- "Insert Pipeline Stages to Increase System Frequency" on page 9–10
- "Using Avalon Bridges" on page 9–10
- "Increasing Transfer Throughput" on page 9–21
- "Reducing Logic Utilization" on page 9–28
- "Reducing Power Consumption" on page 9–34
- "Design Examples" on page 9–39

Designing with Avalon and AXI Interfaces

Qsys Avalon and AXI interconnect for memory-mapped interfaces is flexible, partial crossbar logic that connects master and slave interfaces.

Avalon Streaming (Avalon-ST) links connect point-to-point, unidirectional interfaces, and are typically used in data stream applications. Each a pair of components is connected without any requirement to arbitrate between the data source and sink.

Because Qsys supports multiplexed memory-mapped and streaming connections, you can implement systems that use multiplexed logic for control and streaming logic for data in a single design.

For more information about designing streaming and memory-mapped components, refer to the *Creating Qsys Components* chapter in volume 1 of the *Quartus II Handbook*.

Designing Streaming Components

When you design streaming component interfaces, you must consider integration and communication for each component in the system. One common consideration is buffering data internally to accommodate latency between components. For example, if the component's Avalon-ST output or source of streaming data is back-pressured because the `ready` signal is deasserted, then the component must back-pressure its input or sink interface to avoid overflow.

You can use a FIFO to back-pressure internally on the output side of the component, so that the input can accept more data even if the output is back-pressured. Then, you use the FIFO `almost full` flag to back-pressure the sink interface or input data when the FIFO has only enough space left to satisfy the internal latency. You drive the data valid signal of the output or source interface with the `not empty` flag of the FIFO when that data is available.

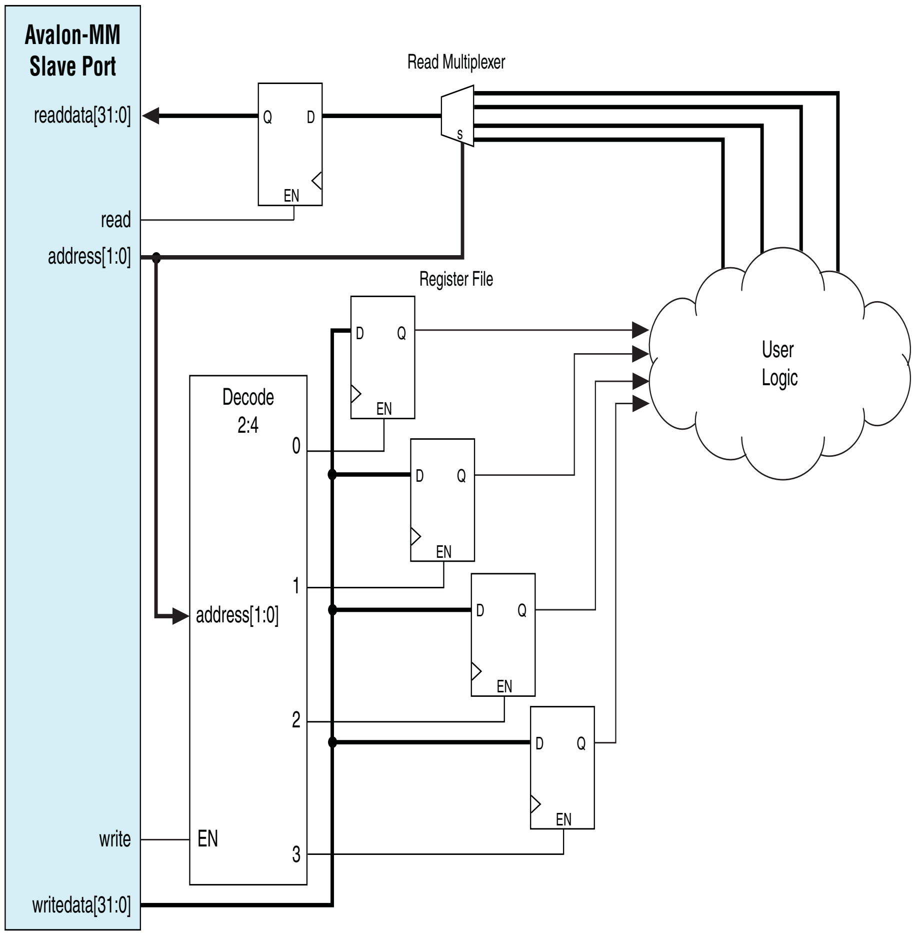
AXI streaming and bridge components are not available in the Quartus II software, version 12.1.

Designing Memory-Mapped Components

When designing with memory-mapped components, "Example of Control and Status Registers (CSR) in a Slave Component" on page 9–3 is an example that you can use to implement any component that contains multiple registers mapped to memory locations. Components that implement read and write memory-mapped transactions require three main building blocks: an address decoder, a register file, and a read multiplexer.

Figure 9–1 shows how to implement a set of four output registers to support software read back from logic.

Figure 9–1. Example of Control and Status Registers (CSR) in a Slave Component



The decoder enables the appropriate 32-bit or 64-bit register for writes. For reads, the address bits drive the multiplexer selection bits. The `read` signal registers the data from the multiplexer, adding a pipeline stage so that the component can achieve a higher clock frequency. This component has write wait states and one read wait state. Alternatively, if you want high throughput, you might set both the read and write wait states to zero, and then specify a read latency of one, because the component also supports pipelined reads.

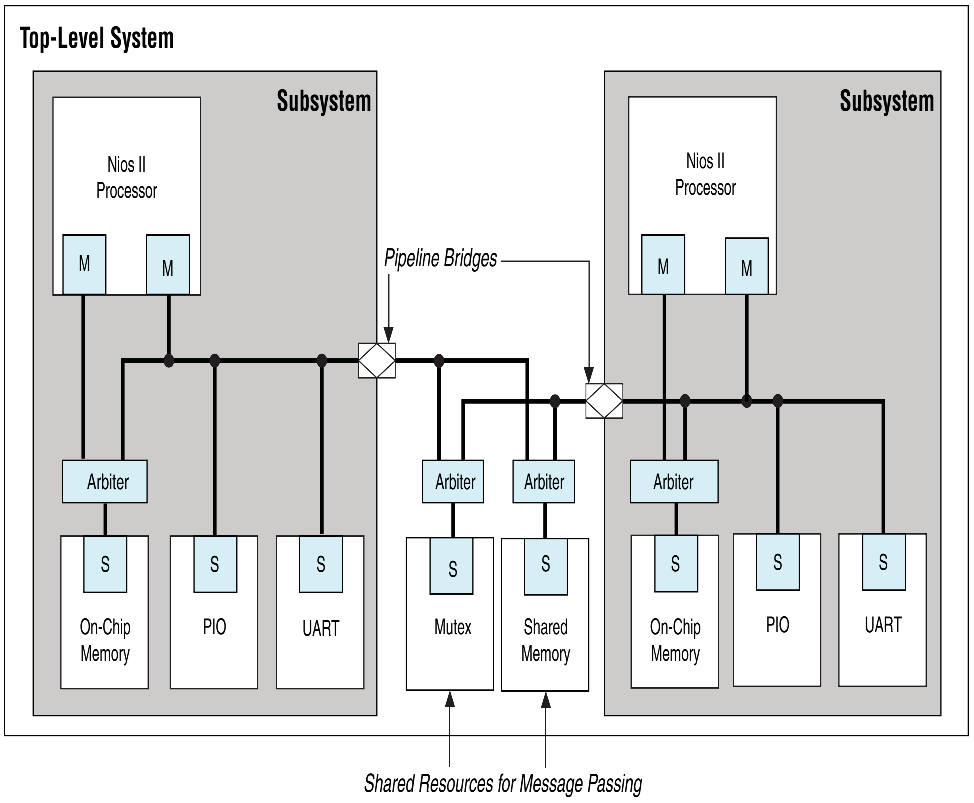
Using Hierarchy in Systems

You can use hierarchy to sub-divide a system into smaller subsystems that can be connected together in a top-level Qsys system. You can use hierarchy to simplify verification control of slaves connected to each master in a memory-mapped system. Before you begin implementing subsystems in your design, you should plan the system hierarchical blocks at the top level, using the following guidelines:

- **Plan shared resources**—For example, determine the best location for shared resources in the system hierarchy. For example, if two subsystems share resources, you should add the components that use those resources to a higher-level system for easy access.
- **Plan shared address space between subsystems**—Planning the address space ensures you can set appropriate sizes for bridges between subsystems.
- **Plan how much latency you might add to your system**—When you add a pipeline bridge between subsystems, you might add more latency to the overall system. You can reduce the added latency by parameterizing the pipeline bridge with zero cycles of latency.

Figure 9–2 shows an example of two Nios II processor subsystems with shared resources for message passing. Bridges in each subsystem export the Nios II data master to the top-level system that includes the mutex (mutual exclusion component) and shared memory component (which could be another on-chip RAM, or a controller for an off-chip RAM device).

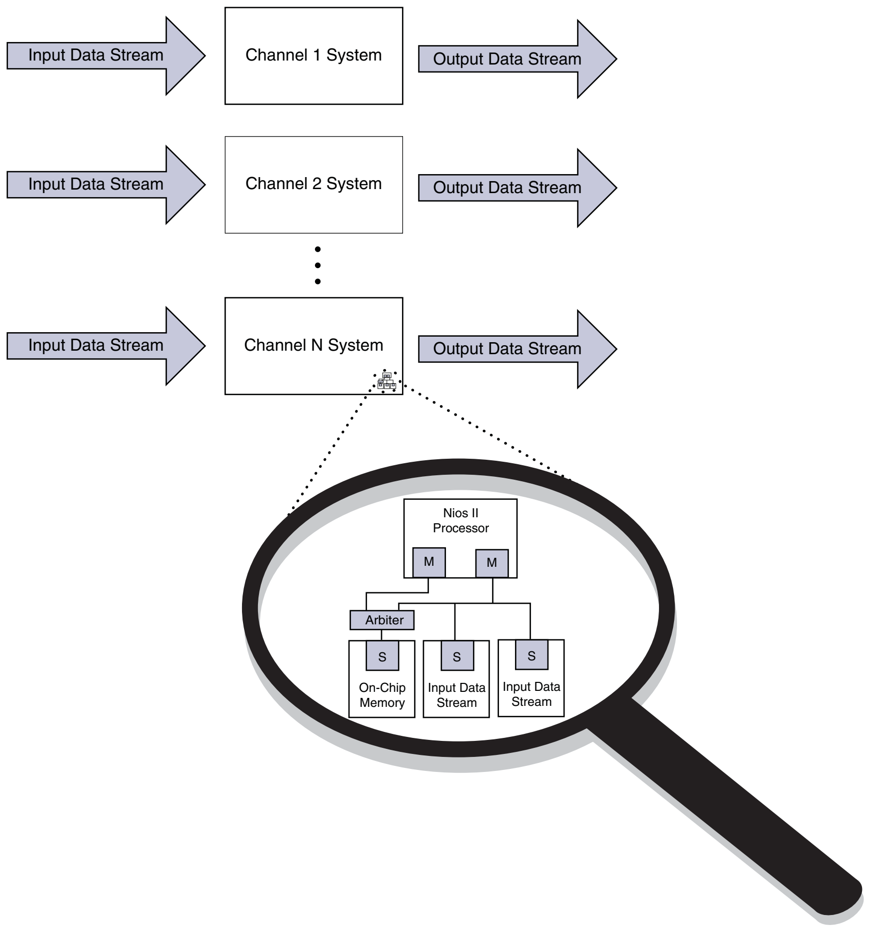
Figure 9–2. Message Passing Between Subsystems



If a design contains one or more identical functional units, the functional unit can be defined as a subsystem and instantiated multiple times within a top-level system. You can also design systems that process multiple data channels by instantiating the same subsystem for each channel. This approach is easier to maintain than a larger, non hierarchical system. In addition, such systems are easier to scale because you can calculate the required resources as a simple multiple of the subsystem requirements.

Figure 9–3 shows a design with three subsystems, each processing a unique channel.

Figure 9–3. Multi Channel System



Using Concurrency in Memory-Mapped Systems

Qsys interconnect takes advantage of parallel hardware in FPGAs, which allows you to design concurrency into your system and process multiple transactions simultaneously. The following sections describe design choices that can increase concurrency in your system.

Create Multiple Masters

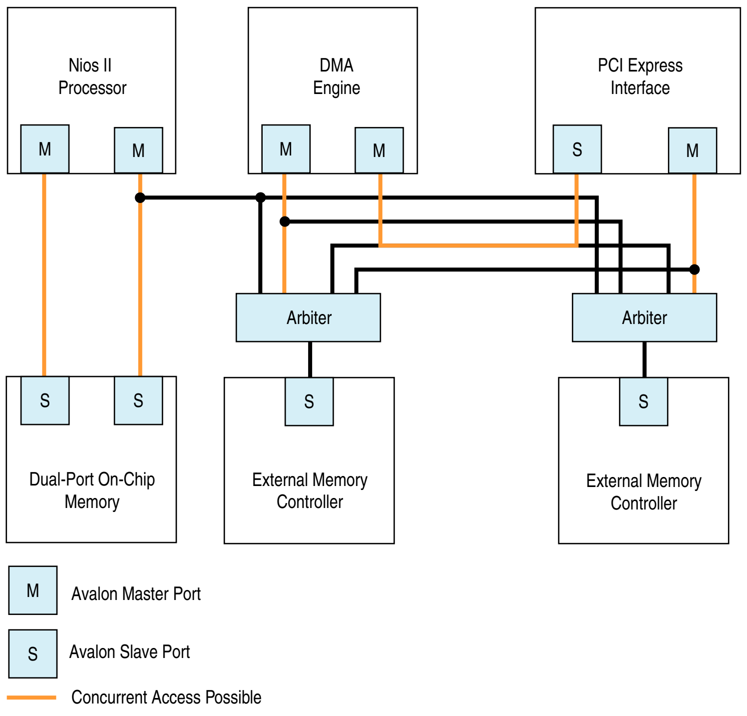
Implementing concurrency requires multiple masters in the system. Systems that include a processor contain at least two master interfaces because the processors include separate instruction and data masters. Master components can be categorized as follows:

- General purpose processors, such as Nios II processors
- DMA (direct memory access) engines
- Communication interfaces, such as PCI Express

Because Qsys generates an interconnect with slave-side arbitration, every master interface in your system can issue transfers concurrently. Masters in the system can issue transfers concurrently as long as they are not posting transfers to the same slave. Concurrency is limited by the number of master interfaces sharing any particular slave interface. If your design requires higher data throughput, you can increase the number of master and slave interfaces to increase the number of transfers that occur simultaneously. Refer to "Create Multiple Slave Interfaces" on page 9–8 for more information.

Figure 9–4 shows a system with three master interfaces. The lines are examples of connections that can be active simultaneously.

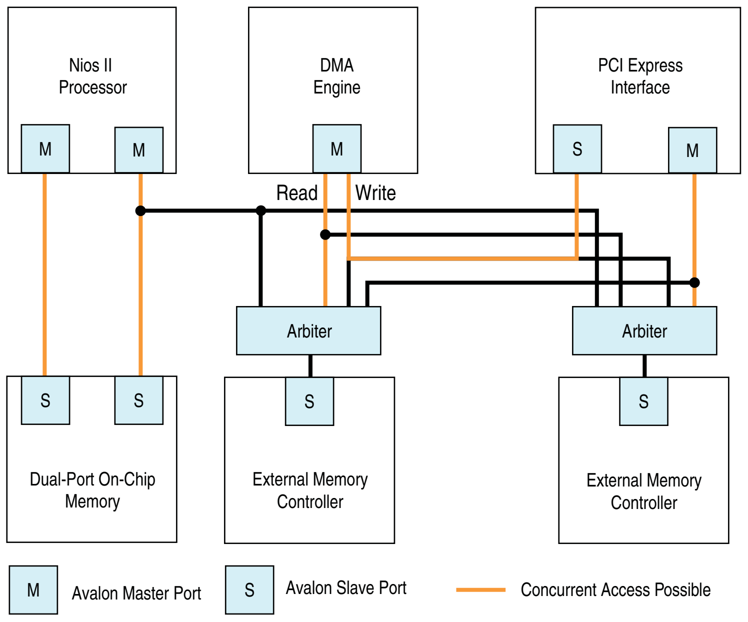
Figure 9–4. Avalon Multiple Master Parallel Access



In this Avalon example, the DMA engine operates with Avalon-MM read and write masters. However, an AXI DMA interface typically has only one master, because in the AXI standard the write and read channels on the master are independent and can process transactions simultaneously.

Figure 9–5 shows an AXI example where the DMA engine operates with a single master, because in AXI the write and read channels on the master are independent and can process transactions simultaneously. This example shows concurrency between the read and write channels, with the yellow lines representing concurrent data paths.

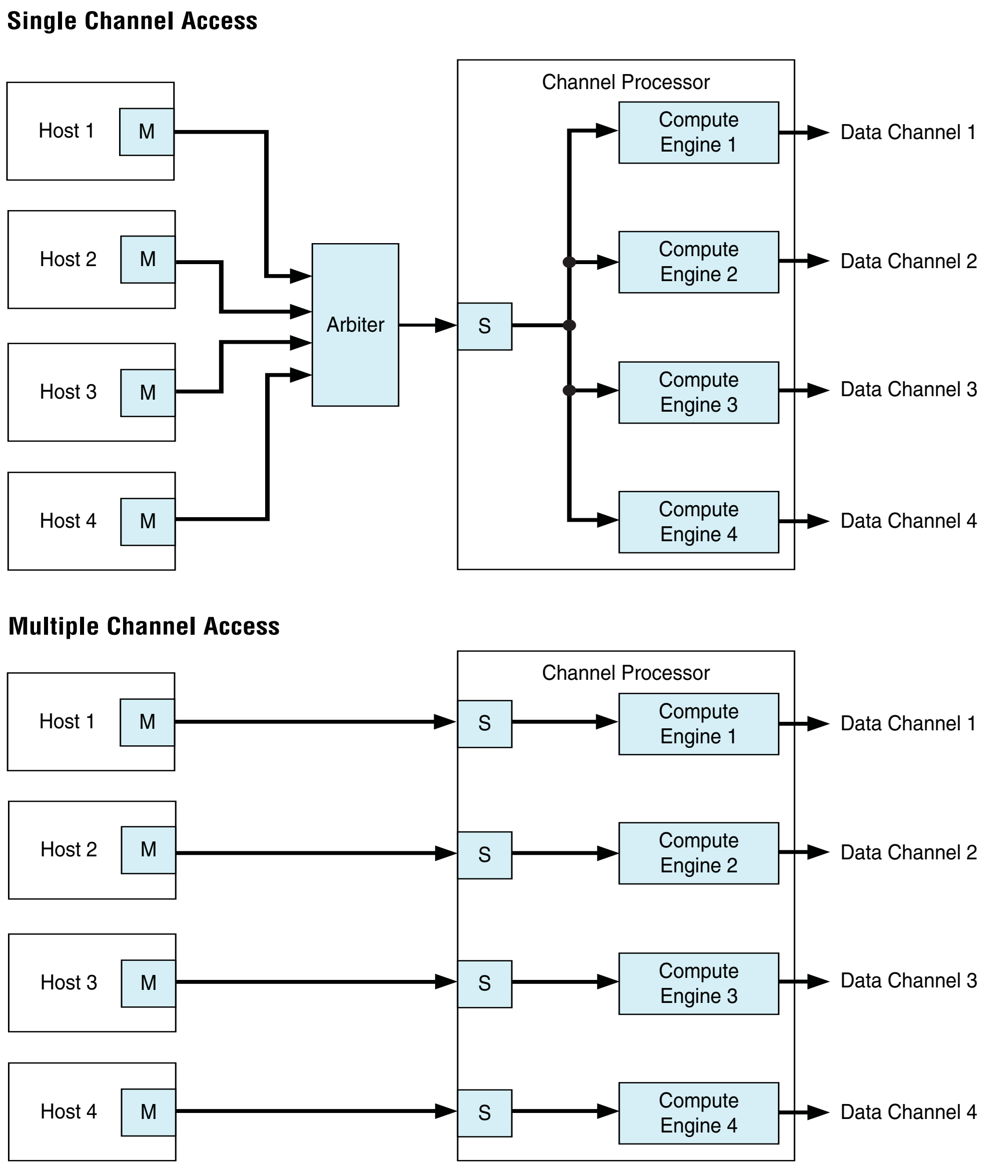
Figure 9–5. AXI Multi Master Parallel Access



Create Multiple Slave Interfaces

You can create multiple slave interfaces for a particular function to increase concurrency in your design. Figure 9–6 shows two channel processing systems. In the first, four hosts must arbitrate for the single slave interface of the channel processor. In the second, each host drives a dedicated slave interface, allowing all master interfaces to simultaneously access the slave interfaces of the component. Arbitration is not necessary when there is a single host and slave interface.

Figure 9–6. Single Interface Vs Multiple Interfaces

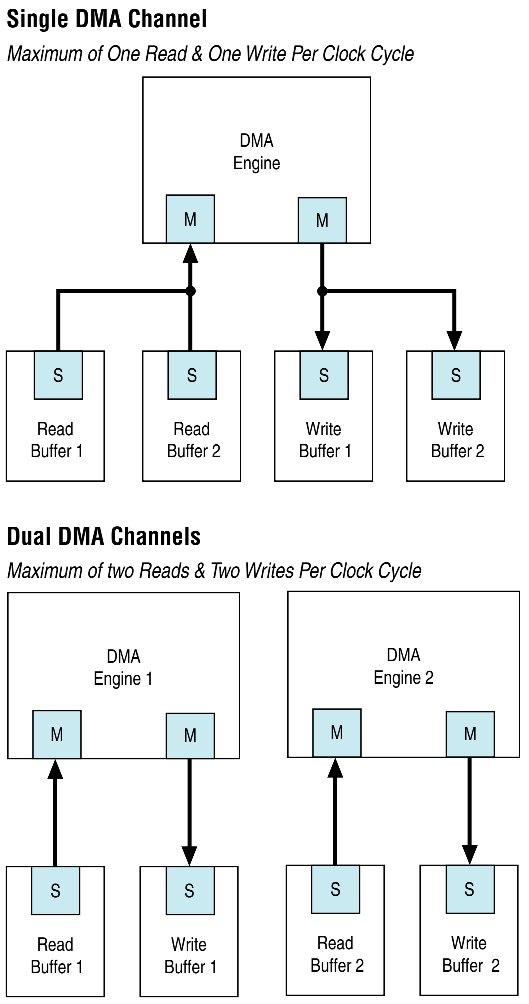


Use DMA Engines

In some systems, you can use DMA engines to increase throughput. You can use a DMA engine to transfer blocks of data between interfaces, which then frees the CPU from carrying out this routine task. A DMA engine transfers data between a programmed start and end address without intervention, and the data throughput is dictated by the components connected to the DMA. Factors that affect data throughput include data width and clock frequency. Figure 9–7 shows a system that can sustain more concurrent read and write operations by including more DMA engines, for the case that accesses to the read and write buffers in the top system can be split between two DMA engines, as shown in the Dual DMA Channels system at the bottom of the figure.

☞ In this example, the DMA engine operates with Avalon-MM write and read masters. An AXI DMA typically has only one master, because in AXI the write and read channels on the master are independent and can process transactions simultaneously.

Figure 9–7. Single or Dual DMA Channels



Insert Pipeline Stages to Increase System Frequency

Qsys provides the **Limit interconnect pipeline stages to** option on the **Project Settings** tab to automatically add pipeline stages to the Qsys interconnect when you generate your system. You can specify between 0 to 4 pipeline stages, where 0 means that the interconnect has a combinational data path. You can specify a unique interconnect pipeline stage value for each subsystem.

Adding pipeline stages might increase the f_{MAX} of your design by reducing the combinational logic depth, at the cost of additional latency and logic utilization.

The insertion of pipeline stages requires certain interconnect components. For example, in a system with a single slave interface, there is no multiplexer; therefore multiplexer pipelining does not occur. When there is an Avalon or AXI single-master to single-slave system, no pipelining occurs, regardless of the **Limit interconnect pipeline stages to** parameter.

☞ For more information about the **Limit interconnect pipeline stages to** parameter, refer to the *Qsys Interconnect* chapter in volume 1 of the *Quartus II Handbook*.

Using Avalon Bridges

You can use bridges to increase system frequency, minimize generated Qsys logic, minimize adapter logic, and to structure system topology when you want to control where Qsys adds pipelining. You can also use bridges with arbiters when there is concurrency in the system.

☞ AXI bridges are not supported in the Quartus II software, version 12.1; however, you can use Avalon bridges between AXI interfaces, and between Avalon domains. Qsys automatically creates interconnect logic between the AXI and Avalon interfaces, so you do not have to explicitly instantiate bridges between these domains. For more discussion about the benefits and disadvantages of shared and separate domains, refer to the *Qsys Interconnect* chapter in volume 1 of the *Quartus II Handbook*.

An Avalon bridge has an Avalon-MM slave interface and an Avalon-MM master interface. You can have many components connected to the bridge slave interface, or many components connected to the bridge master interface, or a single component connected to a single bridge slave or master interface. You can configure the data width of the bridge, which can affect how Qsys generates bus sizing logic in the interconnect. Both interfaces support Avalon-MM pipelined transfers with variable latency, and can also support configurable burst lengths.

Transfers to the bridge slave interface are propagated to the master interface, which connects to components downstream from the bridge. When you need greater control over the interconnect pipelining, you can use bridges instead of using the **Limit Interconnect Pipeline Stages to** parameter.

Increasing System Frequency

In Qsys, you can introduce interconnect pipeline stages or pipeline bridges to increase clock frequency in your system. Bridges control the system interconnect topology and allow you to subdivide the interconnect, giving you more control over pipelining and clock crossing functionality.

Insert Pipeline Bridges

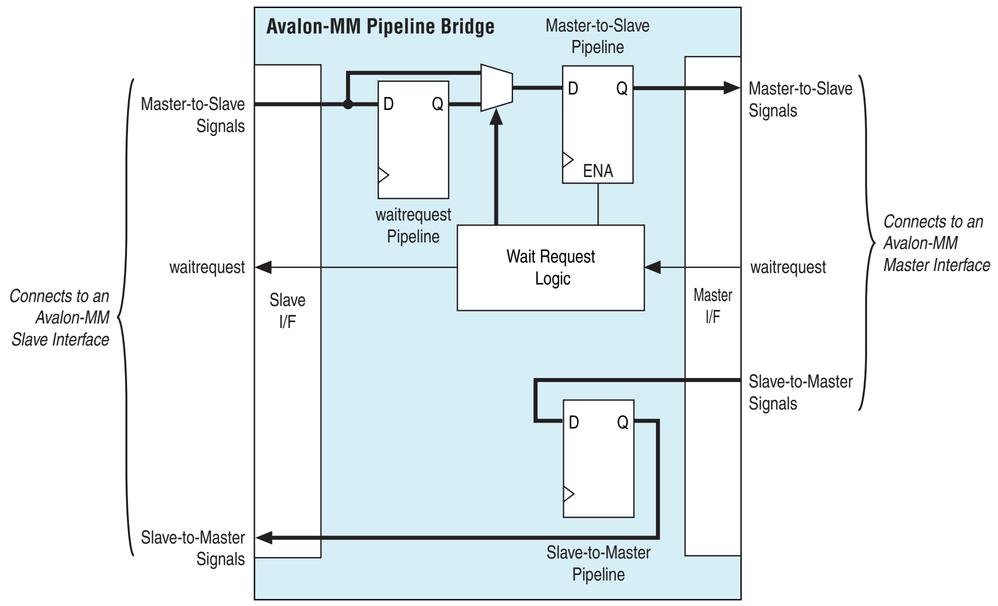
You can insert an Avalon-MM pipeline bridge to insert registers in the path between the bridges and its master and slaves. If a critical register-to-register delay occurs in the Qsys interconnect, a pipeline bridge can help reduce this delay and improve system f_{MAX}.

The Avalon-MM pipeline bridge component integrates into any Qsys system. The pipeline bridge options can increase logic utilization and read latency. The change in topology may also reduce concurrency if multiple masters arbitrate for the bridge.

You can use the Avalon-MM pipeline bridge to control topology without adding a pipeline stage. A pipeline bridge that does not add a pipeline stage is optimal in some latency-sensitive applications. For example, a CPU may benefit from minimal latency when accessing memory.

Figure 9–8 shows the architecture of an Avalon-MM pipeline bridge.

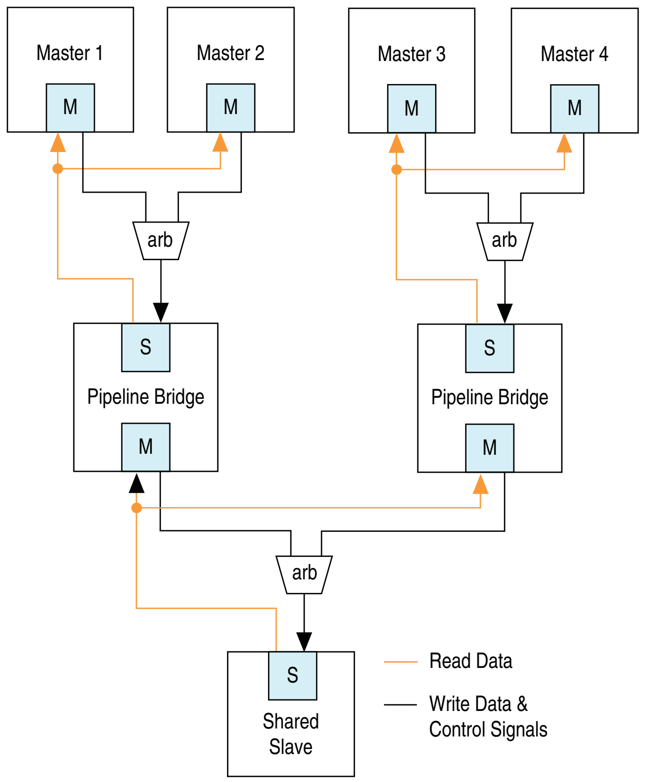
Figure 9–8. Avalon-MM Pipeline Bridge



Implement Command Pipelining (Master-to-Slave)

When many masters share a slave device, use command pipelining to improve performance. The arbitration logic for the slave interface must multiplex the `address`, `writedata`, and `burstcount` signals. The multiplexer width increases proportionally with the number of masters connecting to a single slave interface. The increased multiplexer width might become a timing critical path in the system. If a single pipeline bridge does not provide enough pipelining, you can instantiate multiple instances of the bridge in a tree structure to increase the pipelining and further reduce the width of the multiplexer at the slave interface, as Figure 9–9 shows.

Figure 9–9. Tree of Bridges



Response Pipelining (Slave-to-Master)

A system can benefit from slave-to-master pipelining for masters that connect to many slaves that support read transfers. The interconnect inserts a multiplexer for every read data path back to the master. As the number of slaves supporting read transfers connecting to the master increases, so does the width of the read data multiplexer. As with master-to-slave pipelining, if the performance increase is insufficient with one bridge, you can use multiple bridges in a tree structure to improve f_{MAX}.

Use Clock Crossing Bridges

Transfers to the slave interface are propagated to the master interface. The clock crossing bridge contains a pair of clock crossing FIFOs, which isolate the master and slave interfaces in separate, asynchronous clock domains.

When you use a FIFO clock crossing bridge for the clock domain crossing, you add data buffering. Buffering allows pipelined read masters to post multiple reads to the bridge, even if the slaves downstream from the bridge do not support pipelined transfers.

Separate Component Frequencies

You can use of a clock crossing bridge to place high and low frequency components in separate clock domains. If you limit the fast clock domain to the portion of your design that requires high performance, you might achieve a higher f_{MAX} for this portion of the design.

For example, the majority of processor peripherals included in embedded designs do not need to operate at high frequencies, therefore you do not need to use a high-frequency clock for these components. When you compile a design with the Quartus II software, compilation may take more time when the clock frequency requirements are difficult to meet because the Fitter needs more time to place registers to achieve the required f_{MAX}. To reduce the amount of effort that the Fitter uses on low priority and low performance components, you can place these behind a clock crossing bridge operating at a lower frequency, allowing the Fitter to increase the effort placed on the higher priority and higher frequency data paths.

Minimize Design Logic

Bridges can reduce the interconnect logic by reducing the amount of arbitration and multiplexer logic that Qsys generates. This reduction occurs because bridges limit the number of concurrent transfers that can occur. The following sections discuss how you can use bridges to minimize the logic generated by Qsys.

Avoid Speed Optimizations That Increase Logic

Adding an additional pipeline stage with a pipeline bridge between masters and slaves reduces the amount of combinational logic between registers, which can increase system performance, as described in the section “Increasing System Frequency” on page 9–10.

If you can increase the f_{MAX} of your design logic, you may be able to turn off the Quartus II optimization settings, such as the **Perform register duplication** setting. Register duplication creates duplicate registers to be placed in two or more physical locations in the FPGA to reduce register-to-register delays. You might also want to choose **Speed** for the optimization method, which typically results in higher logic utilization due to logic duplication. By making use of the registers or FIFOs available in the Avalon-MM bridges, you can increase the design speed and avoid needless logic duplication or speed optimizations, thereby reducing the logic utilization of the design.

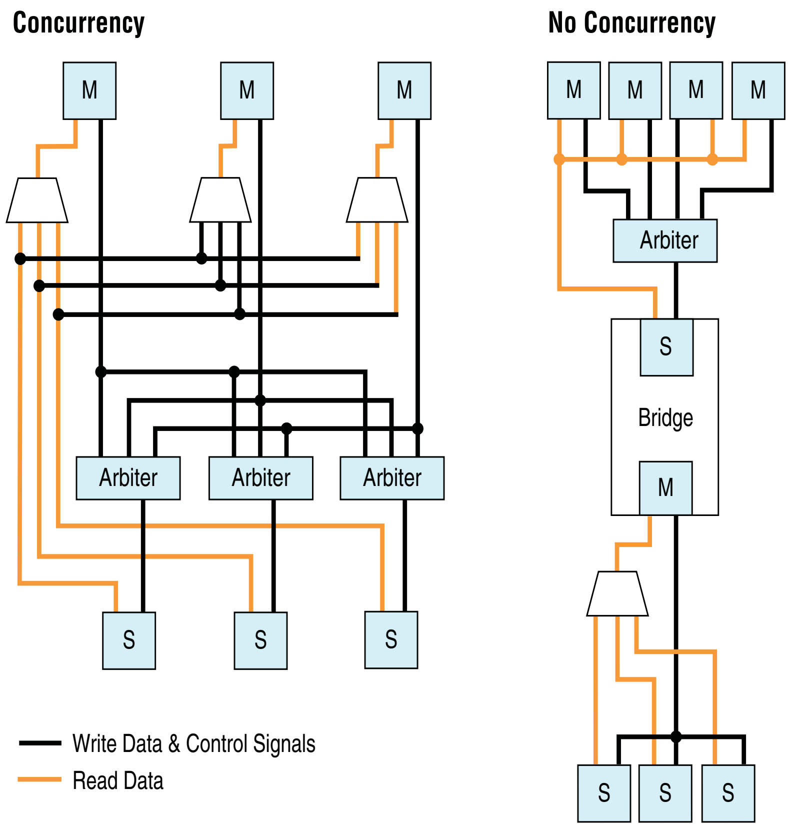
Reduced Concurrency

The amount of logic generated for the interconnect often increases as the system becomes larger because Qsys creates arbitration logic for every slave interface that is shared by multiple master interfaces. Qsys inserts multiplexer logic between master interfaces that connect to multiple slave interfaces if both support `read` data paths. Most embedded processor designs contain components that are either incapable of supporting high data throughput, or do not need to be accessed frequently. These components can contain Avalon-MM master or slave interfaces. Because the interconnect supports concurrent accesses, you might want to limit concurrency by inserting bridges into the datapath to limit the amount of arbitration and multiplexer logic generated.

For example, if your system contains three masters and three slave interfaces that are interconnected, Qsys generates three arbiters and three multiplexers for the read data path. If these masters do not require a significant amount of simultaneous throughput, you can reduce the resources that your design consumes by connecting the three masters to a pipeline bridge. The bridge masters the three slave interfaces, and reduces the interconnect into a bus structure. Qsys creates one arbitration block between the bridge and the three masters, and a single read data path multiplexer between the bridge and three slaves, and prevents concurrency; similar to that of a standard bus architecture. You should not use this method for high throughput data paths to ensure that you do not limit overall system performance.

Figure 9–10 shows the difference in architecture between systems with or without a pipeline bridge.

Figure 9–10. Switch Interconnect to Bus



Minimizing Adapter Logic

Qsys generates adapter logic for clock crossing, width adaptation, and burst support when there is a mismatch between the clock domains, widths, or bursting capabilities of the master and slave interface pairs. Qsys creates burst adapters when the maximum burst length of the master is greater than the master burst length of the slave. The adapter logic creates extra logic resources, which can be substantial when your system contains master interfaces connected to many components that do not share the same characteristics. By placing bridges in your design, you can reduce the amount of adapter logic that Qsys generates.

Effective Placement of Bridges

To determine the effective placement of a bridge, you should initially analyze each master in your system to determine if the connected slave devices support different bursting capabilities or operate in a different clock domain. The maximum `burstcount` of a component is visible as the `burstcount` signal in the HDL file of the component. The maximum burst length is 2 $^{(\text{width(burstcount -1)})}$, so that if the `burstcount` width is four bits, the maximum `burstcount` is eight. If no `burstcount` signal is present, the component does not support bursting or has a burst length of 1.

To determine if the system requires a clock crossing adapter between the master and slave interfaces, check the **clock** column beside the master and slave interfaces in Qsys. If the clock is different for the master and slave interfaces, Qsys inserts a clock crossing adapter between them. To avoid creating multiple adapters, you can place the components containing slave interfaces behind a bridge so that only one adapter is created. By placing multiple components with the same burst or clock characteristics behind a bridge, you limit concurrency and the number of adapters.

You can use a bridge to separate AXI and Avalon domains to minimize burst adaptation logic. For example, if there are multiple Avalon slaves that are connected to an AXI master, you can consider inserting a bridge to access the adaptation logic once before the bridge, instead once per slave. This costs latency, though, and you would also lose concurrency between reads and writes.

Changing the Response Buffer Depth

When you use automatic clock-crossing adapters, Qsys determines the required depth of FIFO buffering based on the slave properties. If a slave has a high **Maximum Pending Reads** parameter, the resulting deep response buffer FIFO that Qsys inserts between the master and slave can consume a lot of device resources. To control the response FIFO depth, you can use a clock crossing bridge and manually adjust its FIFO depth to trade off throughput with smaller memory utilization. For example, if you have masters that cannot saturate the slave, you do not need response buffering, so that using a bridge reduces the FIFO memory depth and reduces the **Maximum Pending Reads** available from the slave.

Consequences of Using Bridges

Before you use pipeline or clock crossing bridges in your design, you should carefully consider their effects. Bridges can have any combination of the following consequences on your design, which could be positive or negative. You can benchmark your system before and after inserting bridges to determine their impact. The following sections discuss the possible consequences of adding bridges to your system.

Increased Latency

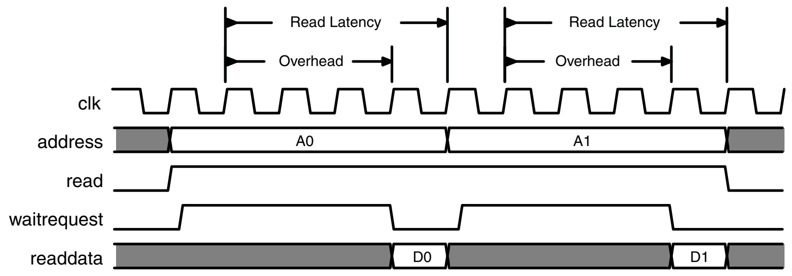
Adding a bridge to your design has an effect on the read latency between the master and the slave. Depending on the system requirements and the type of master and slave, this latency increase may or may not be acceptable in your design.

Acceptable Latency Increase

For a pipeline bridge, a cycle of latency is added for each pipeline option that is enabled. The buffering in the clock crossing bridge also adds latency. If you use a pipelined or burst master that posts many read transfers, the increase in latency does not impact performance significantly because the latency increase is very small compared to the length of the data transfer.

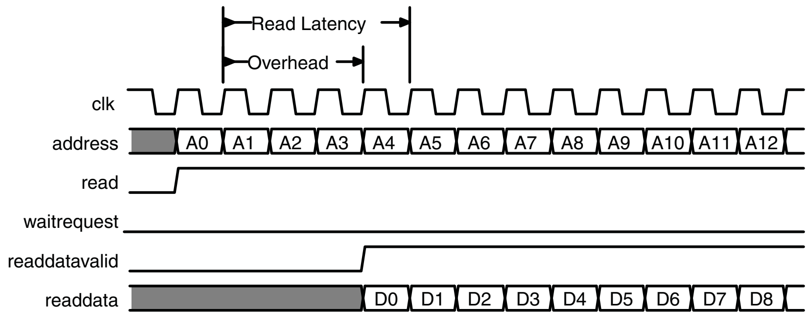
For example, if you use a pipelined read master such as a DMA controller to read data from a component with a fixed read latency of four clock cycles, but only perform a single word transfer, the overhead is three clock cycles out of the total four, assuming there is no additional pipeline latency in the Qsys interconnect. The read throughput is only 25%. Figure 9–11 shows this type of low-efficiency read transfer.

Figure 9–11. Low-Efficiency Read Transfer



However, if 100 words of data are transferred without interruptions, the overhead is three cycles out of the total of 103 clock cycles, corresponding to a read efficiency of approximately 97% when there is no additional pipeline latency in the interconnect. Adding a pipeline bridge to this read path adds two extra clock cycles of latency. The transfer requires 105 cycles to complete, corresponding to an efficiency of approximately 94%. Although the efficiency decreased by 3%, adding the bridge might increase the f_{MAX} by 5%, for example, and in that case, if the clock frequency can be increased, the overall throughput would improve. As the number of words transferred increases, the efficiency increases to nearly 100%, whether or not a pipeline bridge is present. Figure 9–12 shows this type of high-efficiency read transfer.

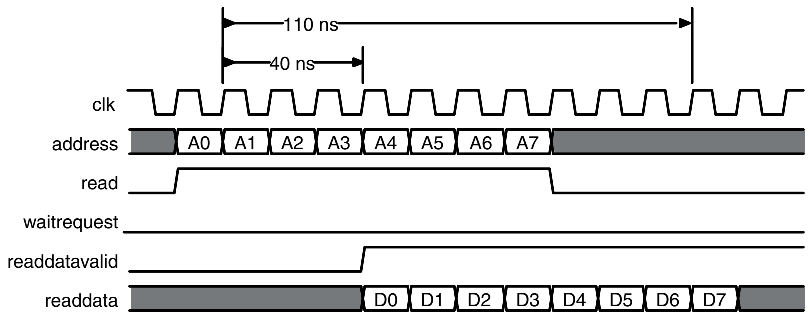
Figure 9–12. High Efficiency Read Transfer



Unacceptable Latency Increase

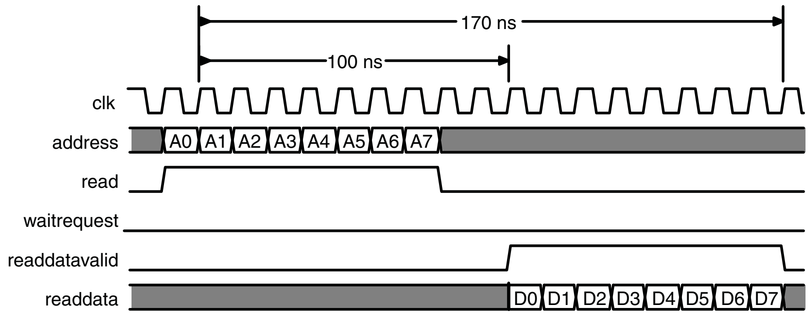
Processors are sensitive to high latency read times and typically fetch data for use in calculations that cannot proceed until the data arrives. Before adding a bridge to the data path of a processor instruction or data master, determine whether the clock frequency increase justifies the added latency. Figure 9–13 shows the performance of a Nios II processor and memory operating at 100 MHz. The Nios II processor instruction master has a cache memory with a read latency of four cycles, that is eight sequential words of data return for each read. At 100 MHz, the first read takes 40 ns to complete. Each successive word takes 10 ns so that eight reads complete in 110 ns.

Figure 9–13. Processor System: Eight Reads with Four Cycles Latency



Adding a clock crossing bridge allows the memory to operate at 125 MHz in this example. However, this increase in frequency is negated by the increase in latency for the following reasons, as shown in Figure 9–14. If the clock crossing bridge adds six clock cycles of latency at 100 MHz, then the memory continues to operate with a read latency of four clock cycles; consequently, the first read from memory takes 100 ns, and each successive word takes 10 ns because reads arrive at the frequency of the processor, which is 100 MHz. In total, eight reads complete after 170 ns. Although the memory operates at a higher clock frequency, the frequency at which the master operates limits the throughput.

Figure 9–14. Processor System: Eight Reads with Ten Cycles Latency



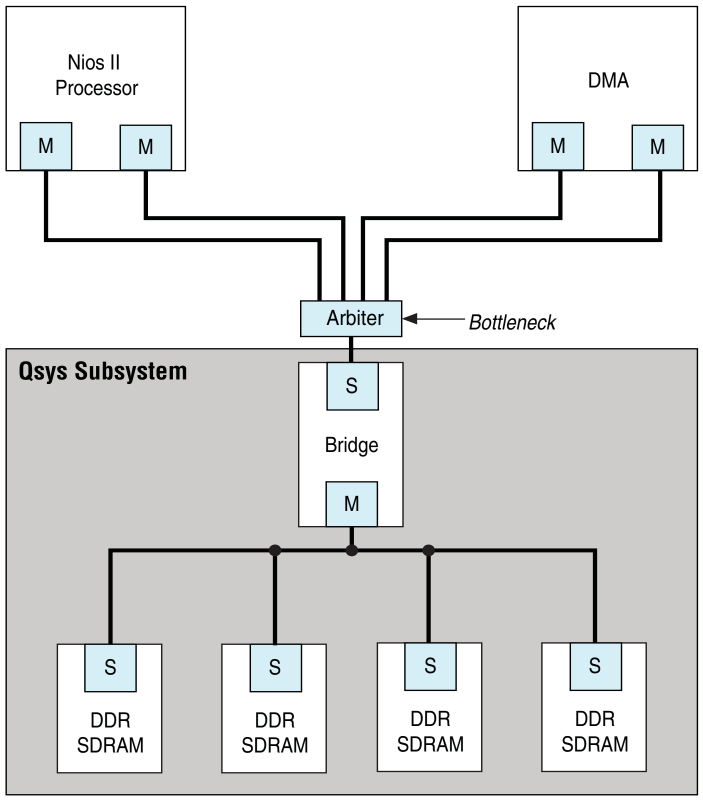
Limited Concurrency

Placing an bridge between multiple master and slave interfaces limits the number of concurrent transfers your system can initiate. This limitation is the same as connecting multiple master interfaces to a single slave interface. The slave interface of the bridge is shared by all the masters and, as a result, Qsys creates arbitration logic. If the components placed behind a bridge are infrequently accessed, this concurrency limitation might be acceptable.

Bridges can have a negative impact on system performance if you use them inappropriately. For example, if multiple memories are used by several masters, you should not place the memory components behind a bridge. The bridge limits memory performance by preventing concurrent memory accesses. Placing multiple memory components behind a bridge can cause the separate slave interfaces to appear as one large memory to the masters accessing the bridge; all masters must access the same slave interface.

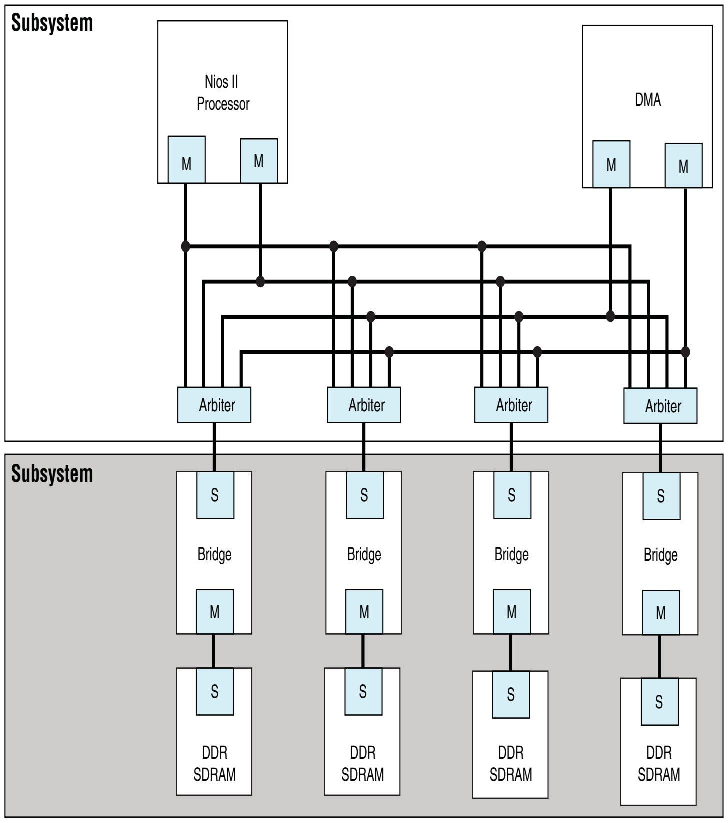
Figure 9–15 shows a memory subsystem with one bridge that acts as a single slave interface for the Avalon-MM Nios II and DMA masters, which results in a bottleneck architecture. The bridge acts as a bottleneck between the two masters and the memories. An AXI DMA typically has only one master, because in the AXI standard the write and read channels on the master are independent and can process transactions simultaneously.

Figure 9–15. Inappropriate Use of a Bridge in a Hierarchical System



If the f_{MAX} of your memory interfaces is low and you want to use a pipeline bridge between subsystems, you can place each memory behind its own bridge, which increases the f_{MAX} of the system without sacrificing concurrency, as Figure 9–16 shows.

Figure 9–16. Efficient Memory Pipelining Without a Bottleneck in a Hierarchical System



Address Space Translation

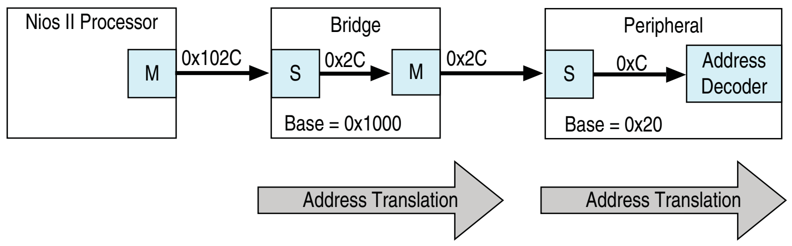
The slave interface of a pipeline or clock crossing bridge has a base address and address span. You can set the base address or allow Qsys to set it automatically. The address of the slave interface is the base offset address of all the components connected to the bridge. The address of components connected to the bridge is the sum of the base offset and the address of that component.

Address Shifting

The master interface of the bridge drives only the address bits that represent the offset from the base address of the bridge slave interface. Any time a master accesses a slave through a bridge, both addresses must be added together, otherwise the transfer fails. The **Address Map** tab in Qsys displays the addresses of the slaves connected to each master and includes address translations caused by system bridges.

Figure 9–17 shows how address translation functions. In this example, the Nios II processor connects to a bridge located at base address 0x1000, a slave connects to the bridge master interface at an offset of 0x20, and the processor performs a write transfer to the fourth 32-bit or 64-bit word within the slave. Nios II drives the address 0x102C to interconnect, which is within the address range of the bridge. The bridge master interface drives 0x2C, which is within the address range of the slave, and the transfer completes.

Figure 9–17. Avalon Bridge Address Translation

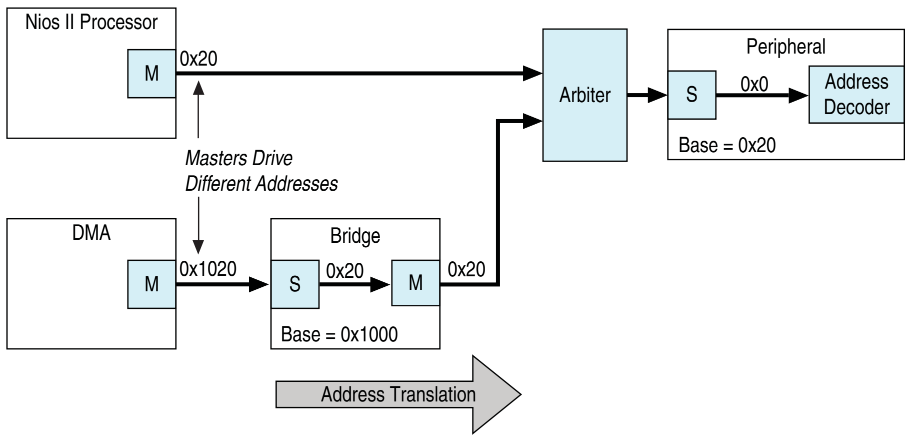


Address Coherency

To simplify the system design, all masters should access slaves at the same location. In many systems, a processor passes buffer locations to other mastering components, such as a DMA controller. If the processor and DMA controller do not access the slave at the same location, Qsys must compensate for the differences.

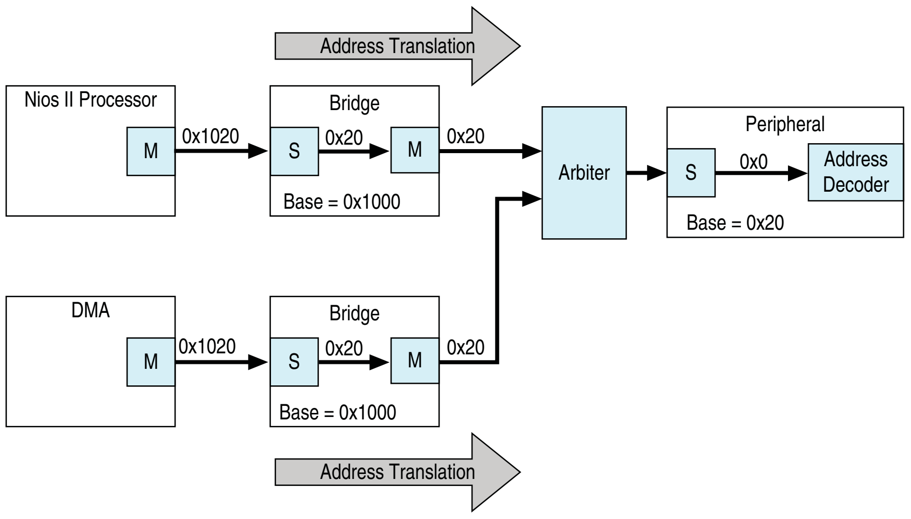
In Figure 9–18, a Nios II processor and DMA controller access a slave interface located at address 0x20. The processor connects directly to the slave interface. The DMA controller connects to a pipeline bridge located at address 0x1000, which then connects to the slave interface. Because the DMA controller accesses the pipeline bridge first, it must drive 0x1020 to access the first location of the slave interface. Because the processor accesses the slave from a different location, you must maintain two base addresses for the slave device.

Figure 9–18. Slave at Different Addresses, Complicating the Software



To avoid the requirement for two addresses, you can add an additional bridge to the system, set its base address to 0x1000, and then disable all the pipelining options in the second bridge so that the bridge has minimal impact on system timing and resource utilization. Because this second bridge has the same base address as the original bridge, the DMA controller connects to both the processor and DMA controller and accesses the slave interface with the same address range, as shown in Figure 9–19.

Figure 9–19. Address Translation Corrected With Bridge



Increasing Transfer Throughput

Increasing the transfer efficiency of the master and slave interfaces in your system increases the throughput of your design. Designs with strict cost or power requirements benefit from increasing the transfer efficiency because you can then use less expensive, lower frequency devices. Designs requiring high performance also benefit from increased transfer efficiency because increased efficiency improves the performance of frequency–limited hardware.

Throughput is the number of symbols (such as bytes) of data that can be transferred in a given clock cycle of time period. Read latency is the number of clock cycles between the address and data phase of a transaction. For example, a read latency of two means that the data is valid two cycles after the address is posted. If the master has to wait for one request to finish before the next begins, such as with a processor, then the read latency is very important to the overall throughput.

You can measure throughput and latency in simulation by observing the waveforms, or using the verification IP monitors. For more information, refer to the *Avalon Verification IP Suite User Guide* or the *Mentor Graphics AXI Verification IP Suite - Altera Edition* on the Altera website.

Using Pipelined Transfers

Pipelined transfers increase the read efficiency by allowing a master to post multiple reads before data from an earlier read returns.

Masters that support pipelined transfers post transfers continuously, relying on the `readdatavalid` signal to indicate valid data. Avalon-MM slaves support pipelined transfers by including the `readdatavalid` signal or operating with a fixed read latency.

AXI masters declare how many outstanding writes and reads it can issue with the `writeIssuingCapability` and `readIssuingCapability` parameters. In the same way, a slave can declare how many reads it can accept with the `readAcceptanceCapability` parameter.

AXI masters with a read issuing capability greater than one are pipelined in the same way as Avalon masters and the `readdatavalid` signal.

Using the Maximum Pending Reads Parameter

If you create a custom component with a slave interface supporting variable-latency reads, you must specify the **Maximum Pending Reads** parameter in the Component Editor. Qsys uses the **Maximum Pending Reads** parameter to generate the appropriate interconnect, and represents the maximum number of read transfers that your pipelined slave component can process. If the number of reads presented to the slave interface exceeds the **Maximum Pending Reads** parameter, then the slave interface must assert `waitrequest`.

Optimizing the value of the **Maximum Pending Reads** parameter requires a good understanding of the latencies of your custom components. This parameter should be based on the component's highest read latency for the various logic paths inside the component. For example, if your pipelined component has two modes, one requiring two clock cycles and the other five, set the **Maximum Pending Reads** parameter to 5, which allows your component to pipeline five transfers, eliminating dead cycles after the initial five-cycle latency.

You can also determine the correct value for the **Maximum Pending Reads** parameter by monitoring the number of reads that are pending during system simulation or while running the hardware. To use this method, set the **Maximum Pending Reads** to a very high value and use a master that issues read requests on every clock. You can use a DMA for this task as long as the data is written to a location that does not frequently assert `waitrequest`. If you implement this method with the hardware, you can observe your component with a logic analyzer or built-in monitoring hardware.

Choosing the correct value for the **Maximum Pending Reads** parameter of your custom pipelined read component is important. If you underestimate the **Maximum Pending Reads** value, you might cause a master interface to stall with a `waitrequest` until the slave responds to an earlier read request and frees a FIFO position.

The **Maximum Pending Reads** parameter controls the depth of the response FIFO inserted into the interconnect for each master connected to the slave. This FIFO does not use significant hardware resources. Overestimating the **Maximum Pending Reads** parameter for your custom component results in a slight increase in hardware utilization. For these reasons, if you are not sure of the optimal value, you should overestimate this value.

If your system includes a bridge, you must set the **Maximum Pending Reads** parameter on the bridge as well. To allow maximum throughput, this value should be equal to or greater than the **Maximum Pending Reads** value for the connected slave that has the highest value. As described in "Changing the Response Buffer Depth" on page 9–15, you can limit the maximum pending reads of a slave and reduce the buffer depth by reducing the parameter value on the bridge if the high throughput is not required. If you do not know the **Maximum Pending Reads** value for all your slave components, you can monitor the number of reads that are pending during system simulation while running the hardware. To use this method, set the **Maximum Pending Reads** parameter to a high value and use a master that issues read requests on every clock, such as a DMA. Then, reduce the number of maximum pending reads of the bridge until the bridge reduces the performance of any masters accessing the bridge.

Arbitration Shares and Bursts

Arbitration shares provide control over the arbitration process. By default, the arbitration algorithm allocates evenly, with all masters receiving one share.

You can adjust the arbitration process to your system requirements by assigning a larger number of shares to the masters that need greater throughput. The larger the arbitration share, the more transfers are allocated to the master to access a slave. The masters gets uninterrupted access to the slave for its number of shares, as long as the master is transacting (reading or writing).

If a master cannot post a transfer and other masters are waiting to gain access to a particular slave, the arbiter grants another master access. This mechanism prevents a master from wasting arbitration cycles if it cannot post back-to-back transfers. A bursting transaction contains multiple beats (or words) of data, starting from a single address. Bursts allow a master to maintain access to a slave for more than a single word transfer. If a bursting master posts a write transfer with a burst length of eight, it is guaranteed arbitration for eight write cycles.

You can assign arbitration shares to Avalon-MM bursting master and AXI masters (which are always considered a bursting master). Each share consists of one burst transaction (such as multi-cycle write), and allows a master to complete a number of bursts before arbitration switches to the next master.

For more information about arbitration shares and bursts, refer to the *Avalon Interface Specifications*, or the *AMBA Protocol Specification* on the ARM website.

Differences Between Arbitration Shares and Bursts

The following three key characteristics distinguish arbitration shares and bursts:

- Arbitration lock
- Sequential addressing
- Burst adapters

Arbitration Lock

When a master posts a burst transfer, the arbitration is locked for that master; consequently, the bursting master should be capable of sustaining transfers for the duration of the locked period. If, after the fourth write, the master deasserts the write (Avalon-MM `write` or AXI `wvalid`) signal for fifty cycles, all other masters continue to wait for access during this stalled period.

To avoid wasted bandwidth, your master designs should wait until a full burst transfer is ready before requesting access to a slave device. Alternatively, you can avoid wasted bandwidth by posting `burstcounts` equal to the amount of data that is ready. For example, if you create a custom bursting write master with a maximum `burstcount` of eight, but only three words of data are ready, you can simply present a `burstcount` of three. This strategy does not result in optimal use of the system bandwidth if the slave is capable of handling a larger burst; however, this strategy prevents stalling and allows access for other masters in the system.

Avalon-MM Sequential Addressing

An Avalon-MM burst transfer includes a base address and a `burstcount`. The `burstcount` represents the number of words of data to be transferred, starting from the base address and incrementing sequentially. Burst transfers are common for processors, DMAs, and buffer processing accelerators; however, sometimes when a master must access non-sequential addresses. Consequently, a bursting master must set the `burstcount` to the number of sequential addresses, and then reset the `burstcount` for the next location.

The arbitration share algorithm has no restrictions on addresses; therefore, your custom master can update the address it presents to the interconnect for every read or write transaction.

AXI has different burst types than the Avalon interface. For more information about AXI burst types, refer to the *Qsys Interconnect* chapter in volume 1 of the *Quartus II Handbook*, and the *AMBA AXI Protocol Specification* on the ARM website.

Burst Adapters

Qsys allows you to create systems that mix bursting and non-bursting master and slave interfaces. This design strategy allows you to connect bursting master and slave interfaces that support different maximum burst lengths, and Qsys generates burst adapters when appropriate.

Qsys inserts a burst adapter whenever a master interface burst length exceeds the burst length of the slave interface, or if the master issues a burst type that the slave cannot support. For example, if you connect an AXI master to an Avalon slave, a burst adapter is inserted.

Qsys assigns non-bursting masters and slave interfaces a burst length of one. The burst adapter divides long bursts into shorter bursts. As a result, the burst adapter adds logic to the address and `burstcount` paths between the master and slave interfaces.

Choosing Avalon-MM Interface Types

To avoid inefficient Avalon-MM transfers, custom master or slave interfaces must use the appropriate simple, pipelined, or burst interfaces. The three possible transfer types are described below.

Simple Avalon-MM Interfaces

Simple interface transfers do not support pipelining or bursting for reads or writes; consequently, their performance is limited. Simple interfaces are appropriate for transfers between masters and infrequently used slave interfaces. In Qsys, the PIO, UART, and Timer include slave interfaces that use simple transfers.

Pipelined Avalon-MM Interfaces

Pipelined read transfers allow a pipelined master interface to start multiple read transfers in succession without waiting for the prior transfers to complete. Pipelined transfers allow master-slave pairs to achieve higher throughput, even though the slave port might require one or more cycles of latency to return data for each transfer.

In many systems, read throughput becomes inadequate if simple reads are used and pipelined transfers can increase throughput. If you define a component with a fixed read latency, Qsys automatically provides the pipelining logic necessary to support pipelined reads. Altera recommends using fixed latency pipelining as the default design starting point for slave interfaces. If your slave interface has a variable latency response time, use the `readdatavalid` signal to indicate when valid data is available. The interconnect implements read response FIFO buffering to handle the maximum number of pending read requests.

To use components that support pipelined read transfers, and to use a pipelined system interconnect efficiently, your system must contain pipelined masters. Refer to the “Avalon Pipelined Read Master Example” on page 9–39 for an example of a pipelined read master. Altera recommends using pipelined masters as the default starting point for new master components. Use the `readdatavalid` signal for these master interfaces.

Because master and slaves often have mismatched pipeline latency, interconnect often contains logic to reconcile the differences. Many cases of pipeline latency are possible, as shown in Table 9–1.

Table 9–1. Various Cases of Pipeline Latency in a Master-Slave Pair (Part 1 of 2)

Master	Slave	Pipeline Management Logic Structure
No pipeline	No pipeline	The Qsys interconnect does not instantiate logic to handle pipeline latency.
No pipeline	Pipelined with fixed or variable latency	The Qsys interconnect forces the master to wait through any slave-side latency cycles. This master-slave pair gains no benefits from pipelining, because the master waits for each transfer to complete before beginning a new transfer. However, while the master is waiting, the slave can accept transfers from a different master.
Pipelined	No pipeline	The Qsys interconnect carries out the transfer as if neither master nor slave were pipelined, causing the master to wait until the slave returns data. An example of a non-pipeline slave is an asynchronous off-chip interface.

Table 9–1. Various Cases of Pipeline Latency in a Master-Slave Pair (Part 2 of 2)

Master	Slave	Pipeline Management Logic Structure
Pipelined	Pipelined with fixed latency	The Qsys interconnect allows the master to capture data at the exact clock cycle when data from the slave is valid, to enable maximum throughput. An example of a fixed latency slave is an on-chip memory.
Pipelined	Pipelined with variable latency	The slave asserts a signal when its `readdata` is valid, and the master captures the data. The master-slave pair can achieve maximum throughput if the slave has variable latency. Examples of variable latency slaves include SDRAM and FIFO memories.

Burst Avalon-MM Interfaces

Burst transfers are commonly used for latent memories such as SDRAM and off-chip communication interfaces such as PCI Express. To use a burst-capable slave interface efficiently, you must connect to a bursting master. Components that require bursting to operate efficiently typically have an overhead penalty associated with short bursts or non-bursting transfers.

Altera recommends that you design a burst-capable slave interface if you know that your component requires sequential transfers to operate efficiently. Because SDRAM memories incur a penalty when switching banks or rows, performance improves when SDRAM memories are accessed sequentially with bursts.

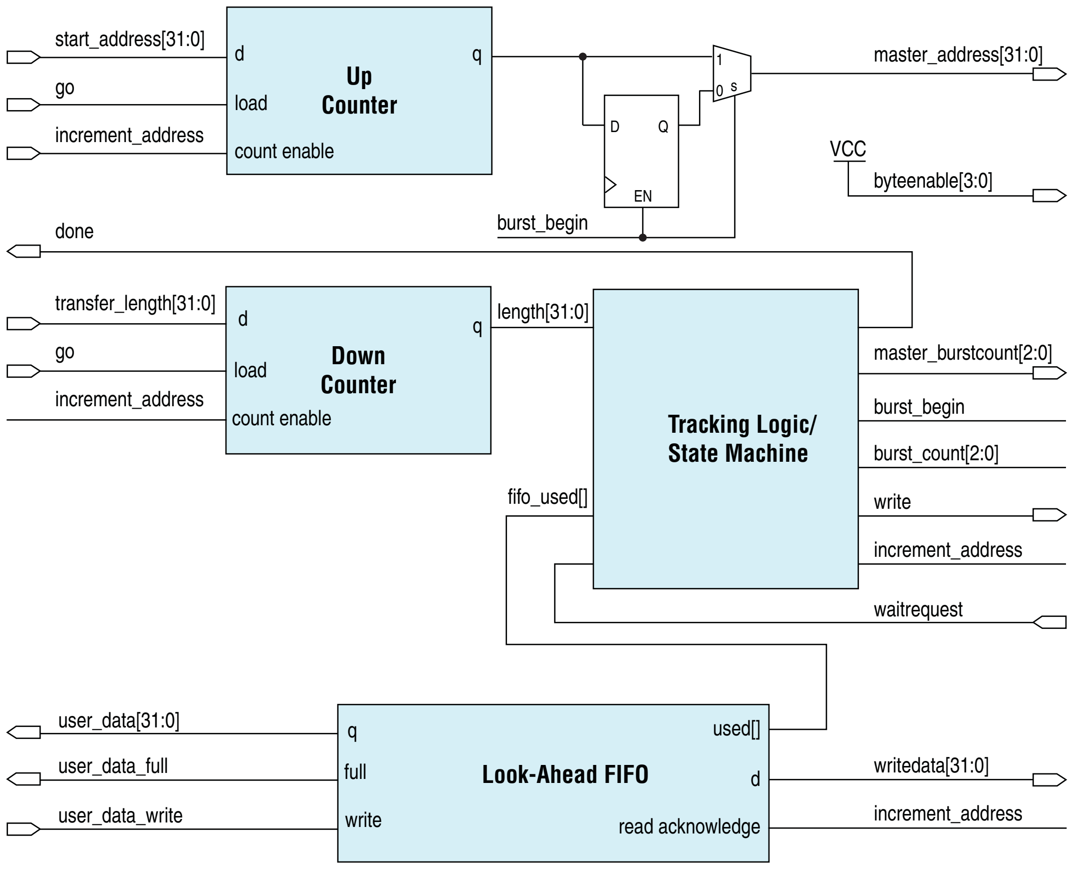
Architectures that use the same signals to transfer address and data also benefit from bursting. Whenever an address is transferred over shared address and data signals, the throughput of the data transfer is reduced. Because the address phase adds overhead, using large bursts increases the throughput of the connection.

Avalon-MM Burst Master Example

Figure 9–20 shows the architecture of a bursting write master that receives data from a FIFO and writes the contents to memory. You can use this master as a starting point for your own bursting components, such as custom DMAs, hardware accelerators, or off-chip communication interfaces. In Figure 9–20, the master performs word accesses and writes to sequential memory locations.

For more information about the example in Figure 9–20, refer to the write master design in the *Avalon Memory-Mapped Master Templates* on the Altera website.

Figure 9–20. Avalon Bursting Write Master



When `go` is asserted, the `start_address` and `transfer_length` are registered. On the next clock cycle, the control logic asserts `burst_begin`. The `burst_begin` signal synchronizes the internal control signals in addition to the `master_address` and `master_burstcount` presented to the interconnect. The timing of these two signals is important because during bursting write transfers `address`, `byteenable`, and `burstcount` must be held constant for the entire burst.

To avoid inefficient writes, the master only posts a burst when enough data has been buffered in the FIFO. To maximize the burst efficiency, the master should stall only when a slave asserts `waitrequest`. In this example, the FIFO's `used` signal tracks the number of words of data that are stored in the FIFO and determines when enough data has been buffered.

The `address` register increments after every word transfer, and the `length` register decrements after every word transfer. The address remains constant throughout the burst. Because a transfer is not guaranteed to complete on burst boundaries, additional logic is necessary to recognize the completion of short bursts and complete the transfer.

Reducing Logic Utilization

This section describes how to minimize logic size of Qsys systems. Typically, there is a trade-off between logic utilization and performance. Information in this section applies to both Avalon and AXI interfaces.

Minimize Interconnect Logic

In Qsys, changes to the connections between master and slave reduce the amount of interconnect logic required in the system.

Create Dedicated Master and Slave Connections

You might be able to create a system so that a master interface connects to a single slave interface. This configuration eliminates address decoding, arbitration, and return data multiplexing, which simplifies the interconnect. Dedicated master-to-slave connections attain the same clock frequencies as Avalon-ST connections.

Typically, these one-to-one connections include an Avalon memory-mapped bridge or hardware accelerator. For example, if you insert a pipeline bridge between a slave and all other master interfaces, the logic between the bridge master and slave interface is reduced to wires. Figure 9–16 on page 9–19 shows this technique. If a hardware accelerator connects only to a dedicated memory, no system interconnect logic is generated between the master and slave pair.

Removing Unnecessary Connections

The number of connections between master and slave interfaces affects the f_{MAX} of your system. Every master interface that you connect to a slave interface increases the width of the multiplexer width. As a multiplexer width increases, so does the logic depth and width that implements the multiplexer in the FPGA. To improve your system performance, connect masters and slaves only when necessary.

When you connect a master interface to many slave interfaces, the multiplexer for the read data signal grows. Avalon typically uses a `readdata` signal, and AXI read data signals add a response status and last indicator to the read response channel using the commands `rdata`, `rresp`, and `rlast`. Use bridges to help control the depth of multiplexers, as shown in Figure 9–9.

Simplifying Address Decode Logic

If address code logic is in the critical path, you may be able to change the address map to simplify the decode logic. Experiment with different address maps, including a one-hot encoding, to see if results improve.

Minimize Arbitration Logic by Consolidating Multiple Interfaces Into One

As the number of components in your design increases, the amount of logic required to implement the interconnect also increases. The number of arbitration blocks increases for every slave interface that is shared by multiple master interfaces. The width of the read data multiplexer increases as the number of slave interfaces supporting read transfers increases on a per master interface basis. For these reasons, consider implementing multiple blocks of logic as a single interface to reduce interconnect logic utilization.

Logic Consolidation Trade-Offs

You should consider the following trade-offs before making modifications to your system or interfaces.

☞ Refer to "Using Concurrency in Memory-Mapped Systems" on page 9–5 for additional discussion on concurrency trade-offs.

First, consider the impact on concurrency that results when you consolidate components. When your system has four master components and four slave interfaces, it can initiate four concurrent accesses. If you consolidate the four slave interfaces into a single interface, then the four masters must compete for access. Consequently, you should only combine low priority interfaces such as low speed parallel I/O devices if the combination does not impact the performance.

Second, determine whether consolidation introduces new decode and multiplexing logic for the slave interface that the interconnect previously included. If an interface contains multiple read and write address locations, the interface already contains the necessary decode and multiplexing logic. When you consolidate interfaces, you typically reuse the decoder and multiplexer blocks already present in one of the original interfaces; however, combining interfaces might simply move the decode and multiplexer logic, rather than eliminate duplication.

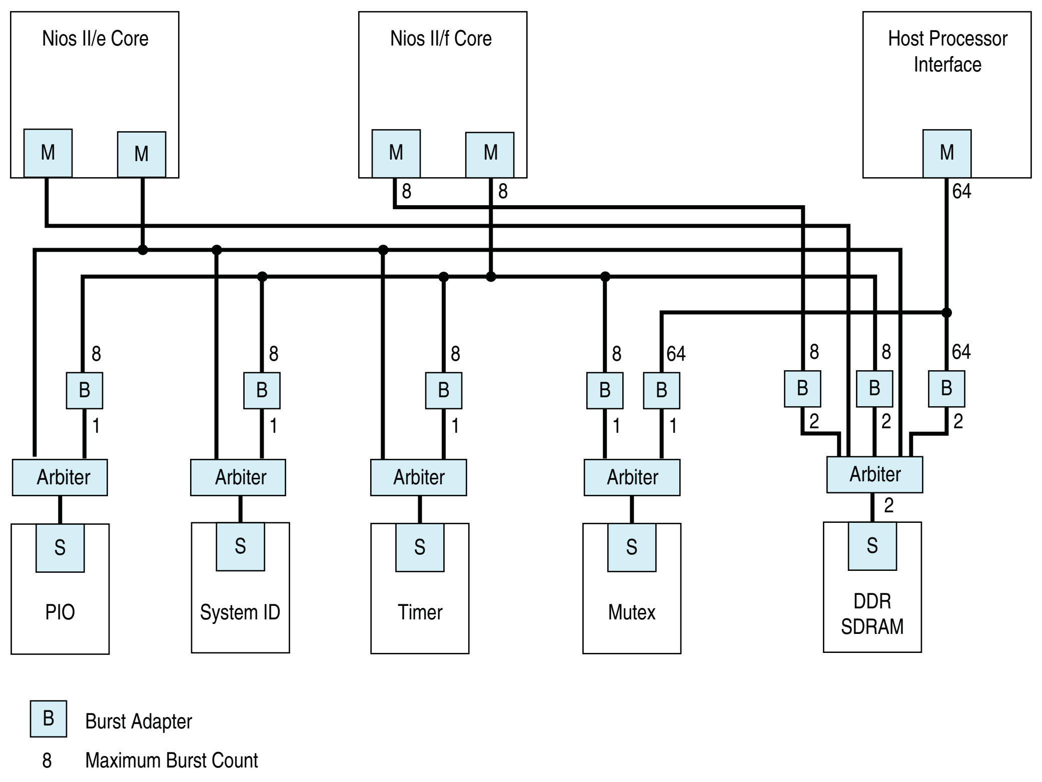
Finally, consider whether consolidating interfaces makes the design complicated. If so, Altera recommends that you do not consolidate interfaces.

System Example of Consolidating Interfaces

In this example, the Nios II/e core maintains communication between the Nios II /f core and external processors. The Nios II/f core supports a maximum burst size of eight. The external processor interface supports a maximum burst length of 64. The Nios II/e core does not support bursting. The only memory in the system is SDRAM with an Avalon maximum burst length of two.

Figure 9–21 shows a system with a mix of components with different burst capabilities. It includes a Nios II/e core, a Nios II/f core, and an external processor, which off-loads some processing tasks to the Nios II/f core.

Figure 9–21. Mixed Bursting System

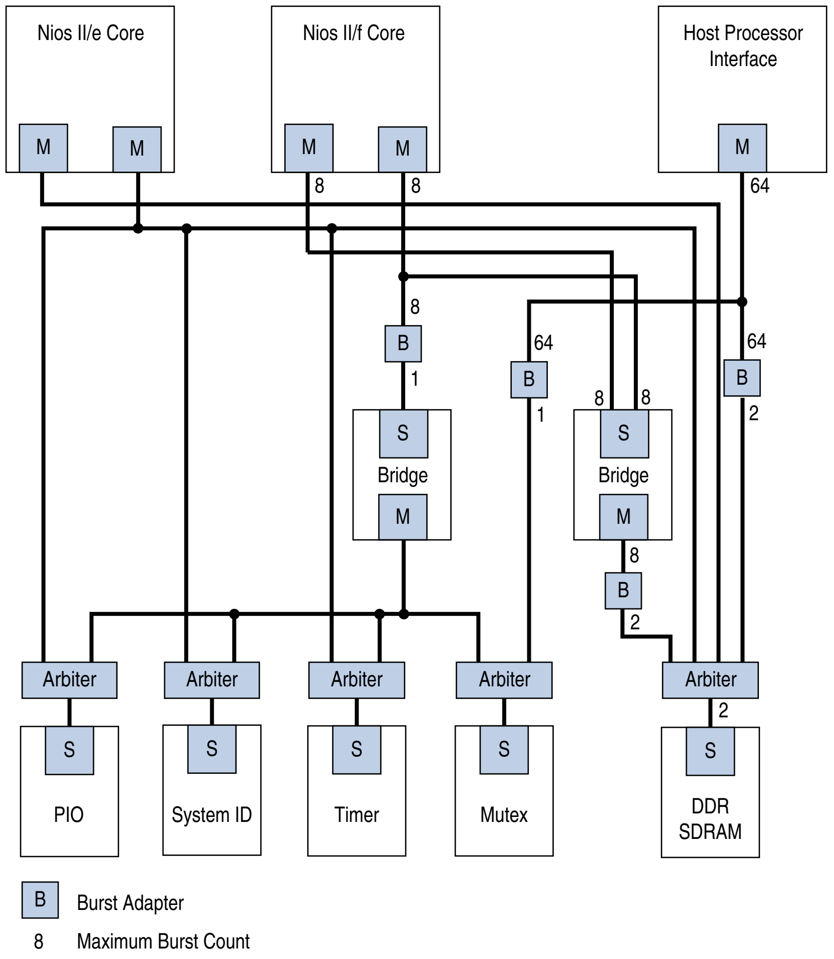


Qsys automatically inserts burst adapters to compensate for burst length mismatches. The adapters reduce bursts to a single transfer, or the length of two transfers. For the external processor interface connecting to DDR SDRAM, a burst of 64 words is divided into 32 burst transfers, each with a burst length of two.

When you generate a system, Qsys inserts burst adapters based on maximum `burstcount` values; consequently, the interconnect logic includes burst adapters between masters and slave pairs that do not require bursting, if the master is capable of bursts. In Figure 9–21, Qsys inserts a burst adapter between the Nios II processors and the timer, system ID, and PIO peripherals. These components do not support bursting and the Nios II processor performs only single word read and write accesses to these components.

To reduce the number of adapters, you can add pipeline bridges, as Figure 9–22 shows. The pipeline bridge between the Nios II/f core and the peripherals that do not support bursts eliminates three burst adapters from Figure 9–21. A second pipeline bridge between the Nios II/f core and the DDR SDRAM, with its maximum burst size set to eight, eliminates another burst adapter.

Figure 9–22. Mixed Bursting System with Bridges



Implementing Multiple Clock Domains

You specify clock domains in Qsys on the **System Contents** tab. Clock sources can be driven by external input signals to Qsys, or by PLLs inside Qsys. Clock domains are differentiated based on the name of the clock. You may create multiple asynchronous clocks with the same frequency.

Clock Domain Crossing Logic

Qsys generates Clock Domain Crossing Logic (CDC) that hides the details of interfacing components operating in different clock domains. The system interconnect supports the memory-mapped protocol with each port independently, and therefore masters do not need to incorporate clock adapters in order to interface to slaves on a different domain. Qsys interconnect logic propagates transfers across clock domain boundaries automatically.

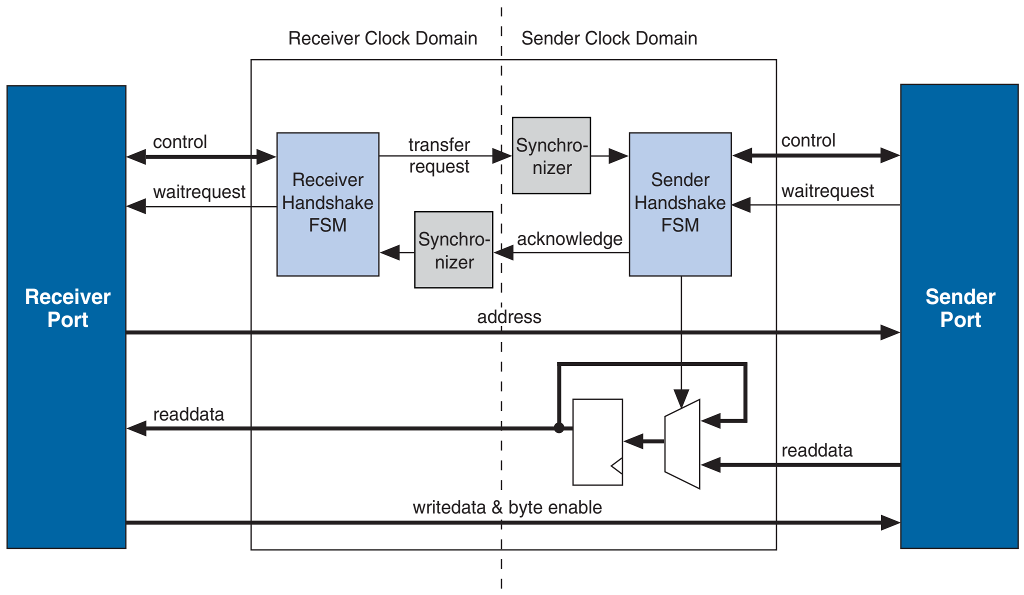
Clock-domain adapters provide the following benefits:

- Allow component interfaces to operate at different clock frequencies.
- Eliminates the need to design CDC hardware.
- Allows each memory-mapped port to operate in only one clock domain, which reduces design complexity of components.
- Enable masters to access any slave without communication with the slave clock domain.
- Allows you to focus performance optimization efforts on components that require fast clock speed.

A clock domain adapter consists of two finite state machines (FSM), one in each clock domain, that use a simple hand-shaking protocol to propagate transfer control signals (`read_request`, `write_request`, and the master `waitrequest` signals) across the clock boundary.

Figure 9–23 shows illustrates a clock domain adapter between one master and one slave.

Figure 9–23. Block Diagram of Clock Crossing Adapter



The synchronizer blocks in Figure 9–23 use multiple stages of flipflops to eliminate the propagation of metastable events on the control signals that enter the handshake FSMs. The CDC logic works with any clock ratio.

The typical sequence of events for a transfer across the CDC logic is described as follows:

1. Master asserts address, data, and control signals.
2. The master handshake FSM captures the control signals, and immediately forces the master to wait.

   ☞ The FSM uses only the control signals, not address and data. For example, the master simply holds the address signal constant until the slave side has safely captured it.

3. Master handshake FSM initiates a transfer request to the slave handshake FSM.
4. The transfer request is synchronized to the slave clock domain.
5. The slave handshake FSM processes the request, performing the requested transfer with the slave.
6. When the slave transfer completes, the slave handshake FSM sends an acknowledge back to the master handshake FSM.
7. The acknowledge is synchronized back to the master clock domain.
8. The master handshake FSM completes the transaction by releasing the master from the wait condition.

Transfers proceed as normal on the slave and the master side, without a special protocol to handle crossing clock domains. From the perspective of a slave, there is nothing different about a transfer initiated by a master in a different clock domain. From the perspective of a master, a transfer across clock domains simply requires extra clock cycles. Similar to other transfer delay cases (for example, arbitration delay or wait states on the slave side), the Qsys forces the master to wait until the transfer terminates. As a result, pipeline master ports do not benefit from pipelining when performing transfers to a different clock domain.

Qsys automatically determines where to insert CDC logic based on the system contents and the connections between components, and places CDC logic to maintain the highest transfer rate for all components. Qsys evaluates the need for CDC logic for each master and slave pair independently, and generates CDC logic wherever necessary.

Duration of Transfers Crossing Clock Domains

CDC logic extends the duration of master transfers across clock domain boundaries. In the worst case which is for reads, each transfer is extended by five master clock cycles and five slave clock cycles. Assuming the default value of 2 for the Master domain synchronizer length and the Slave domain synchronizer length, the components of this delay are the following:

- Four additional master clock cycles, due to the master-side clock synchronizer
- Four additional slave clock cycles, due to the slave-side clock synchronizer
- One additional clock in each direction, due to potential metastable events as the control signals cross clock domains

Systems that require a higher performance clock should use the Avalon-MM clock crossing bridge instead of the automatically inserted CDC logic. The clock crossing bridge includes a buffering mechanism, so that multiple reads and writes can be pipelined. After paying the initial penalty for the first read or write, there is no additional latency penalty for pending reads and writes, increasing throughput by up to four times, at the expense of added logic resources.

For more information, refer to *Avalon Memory-Mapped Design Optimizations* in the *Embedded Design Handbook.*

Reducing Power Consumption

This section describes various low power design changes that you can make to reduce the power consumption of the interconnect and your custom components.

Qsys does not support AXI standard low power extensions in the current version of the QII software.

Use Multiple Clock Domains

When you use multiple clock domains, you should put non-critical logic in the slower clock domain. Qsys automatically reconciles data crossing over asynchronous clock domains by inserting clock crossing logic (handshake or FIFO).

You can use clock crossing in Qsys to reduce the clock frequency of the logic that does not require a high frequency clock, allowing you to reduce power consumption. You can use either handshaking clock crossing bridges or handshaking clock crossing adapters to separate clock domains.

Clock Crossing Bridge

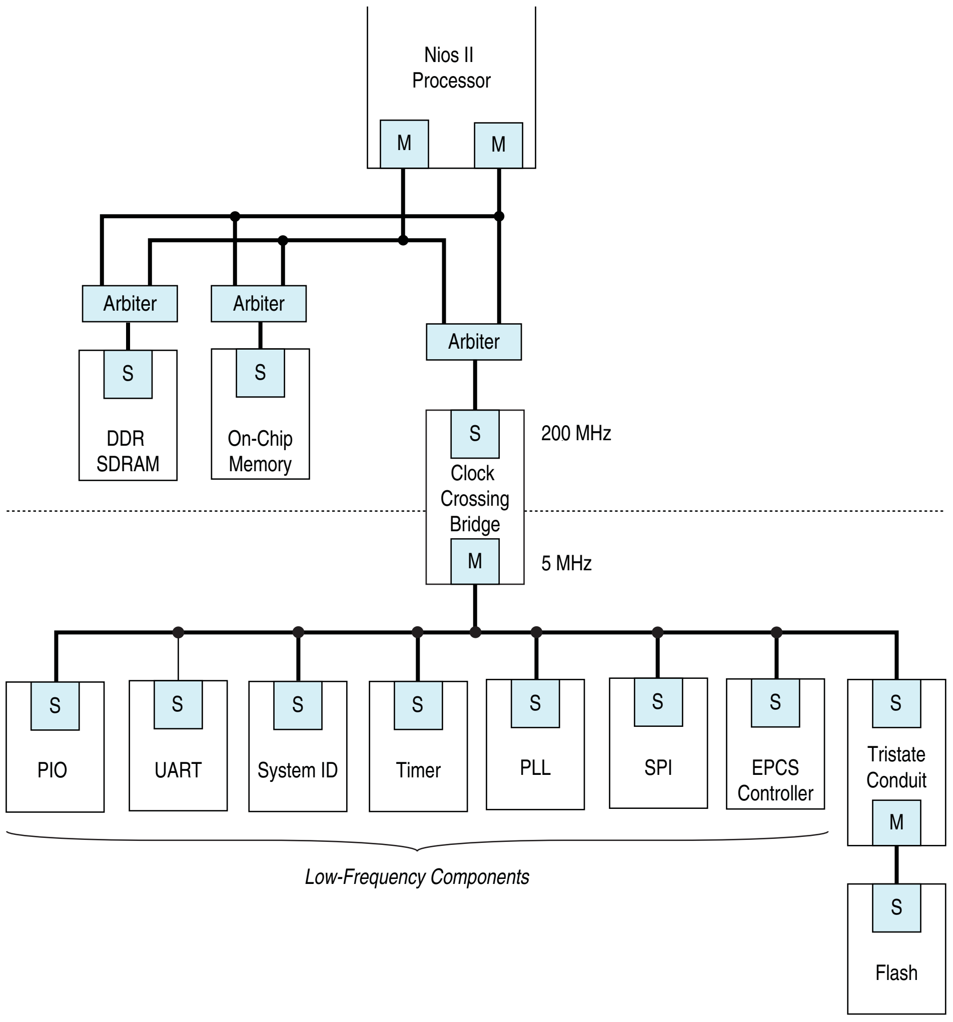
You can use the clock crossing bridge to connect master interfaces operating at a higher frequency to slave interfaces running a a lower frequency. Only connect low throughput or low priority components to a clock crossing bridge that operates at a reduced clock frequency. The following are examples of low throughput or low priority components:

- PIOs
- UARTs (JTAG or RS-232)
- System identification (SysID)
- Timers
- PLL (instantiated within Qsys)
- Serial peripheral interface (SPI)
- EPCS controller
- Tristate bridge and the components connected to the bridge

By reducing the clock frequency of the components connected to the bridge, you reduce the dynamic power consumption of your design. Dynamic power is a function of toggle rates and decreasing the clock frequency decreases the toggle rate.

Figure 9–24 shows a system where a bridge reduces power consumption.

Figure 9–24. Reducing Power Utilization Using a Bridge to Separate Clock Domains



Qsys automatically inserts clock crossing adapters between master and slave interfaces that operate at different clock frequencies. You can choose the type of clock crossing adapter in the Qsys **Project Settings** tab. There are three types of clock crossing adapter types available in Qsys, as described below. Adapters do not appear in the Qsys **Connection** column because you do not insert them.

Clock Crossing Adapter Types

Specifies the default implementation for automatically inserted clock crossing adapters. The following adapter types are available:

- **Handshake—**Uses a simple handshaking protocol to propagate transfer control signals and responses across the clock boundary. This adapter uses fewer hardware resources because each transfer is safely propagated to the target domain before the next transfer can begin. The Handshake adapter is appropriate for systems with low throughput requirements.

- **FIFO**—Uses dual-clock FIFOs for synchronization. The latency of the FIFO adapter is approximately two clock cycles more than the handshake clock crossing component, but the FIFO-based adapter can sustain higher throughput because it can support multiple transactions simultaneously. The FIFO adapter requires more resources. The FIFO adapter is appropriate for memory-mapped transfers requiring high throughput across clock domains.
- **Auto**—Qsys specifies the appropriate FIFO adapter for bursting links and the Handshake adapter for all other links.

Throughput

Because the clock crossing bridge uses FIFOs to implement the clock crossing logic, it buffers transfers and data. Clock crossing adapters are not pipelined, so that each transaction is blocking until the transaction completes. Blocking transactions may lower the throughput substantially; consequently, if you want to reduce power consumption without limiting the throughput significantly, you should use the clock crossing bridge or the FIFO clock crossing adapter. However, if the design simply requires single read transfers, a clock crossing adapter is preferable because the latency is lower.

Resource Utilization

The clock crossing bridge requires few logic resources besides on-chip memory. The number of on-chip memory blocks used is proportional to the address span, data width, buffering depth, and bursting capabilities of the bridge. The clock crossing adapter does not use on-chip memory and requires a moderate number of logic resources. The address span, data width, and the bursting capabilities of the clock crossing adapter determine the resource utilization of the device.

Throughput versus Memory Trade-Offs

When you decide to use a clock crossing bridge or clock crossing adapter, you must consider the effects of throughput and memory utilization in your design. If on-chip memory resources are limited, you may be forced to choose the clock crossing adapter. Using the clock crossing bridge to reduce the power of a single component may not justify using more resources. However, if you can place all your low priority components behind a single clock crossing bridge, you reduce power consumption in your design.

Minimizing Toggle Rates

Your design consumes power whenever logic transitions between on and off states. When the state is held constant between clock edges, no charging or discharging occurs. This section discusses the following three design techniques that you can use to reduce the toggle rates of your system:

- Registering component boundaries
- Using clock enable signals
- Inserting bridges

Registering Component Boundaries

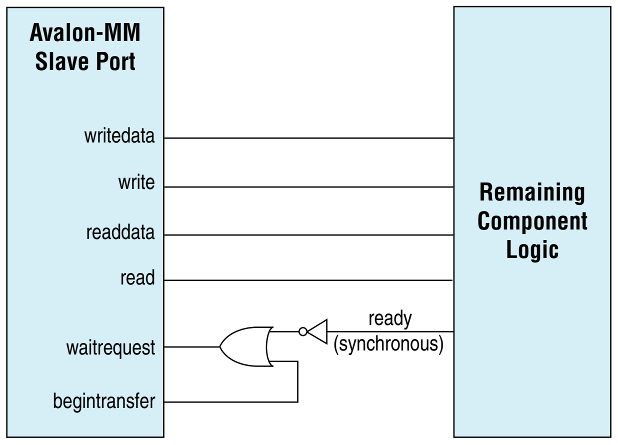
Qsys interconnect is uniquely combinational when no adapters or bridges are present and there is no interconnect pipelining. When a slave interface is not selected by a master, various signals may toggle and propagate into the component. By registering the boundary of your component at the master or slave interface, you can minimize the toggling of the interconnect and your component. In addition, registering boundaries can improve operating frequency. When you register the signals at the interface level, you must ensure that the component continues to operate within the interface standard specification.

Avalon-MM `waitrequest` is a difficult signal to synchronize when you add registers to your component. `waitrequest` must be asserted during the same clock cycle that a master asserts read or write to, in order to prolong the transfer. A master interface may read the `waitrequest` signal too early and post more reads and writes prematurely.

☞ There is no direct AXI equivalent for `waitrequest` and `burstcount`, though the *AMBA Protocol Specification* implies that `ready` (the equivalent of Avalon-MM `waitrequest`) cannot depend combinatorially on AXI `valid`. Therefore, Qsys typically buffers AXI component boundaries (at least for the `ready` signal).

For slave interfaces, the interconnect manages the `begintransfer` signal, which is asserted during the first clock cycle of any `read` or `write` transfer. If your `waitrequest` is one clock cycle late, you can logically `OR` your `waitrequest` and the `begintransfer` signals to form a new `waitrequest` signal that is properly synchronized, as shown in Figure 9–25.

Figure 9–25. Variable Latency



Alternatively, your component can assert `waitrequest` before it is selected, guaranteeing that the `waitrequest` is already asserted during the first clock cycle of a transfer.

Using Clock Enables

You can use clock enables to hold your logic in a steady state. You can use the write and read signals as clock enables for slave components. Even if you add registers to your component boundaries, your interface can potentially toggle without the use of clock enables.

You can also use the clock enable to disable combinational portions of your component. For example, you can use an active high clock enable to mask the inputs into your combinational logic to prevent it from toggling when the component is inactive. Before preventing inactive logic from toggling, you must determine if the masking causes your circuit to function differently. If masking causes a functional failure, it might be possible to use a register stage to hold the combinational logic constant between clock cycles.

Inserting Bridges

You can use bridges to reduce toggle rates, if you do not want to modify the component by using boundary registers or clock enables. A bridge acts as a repeater where transfers to the slave interface are repeated on the master interface. If the bridge is not accessed, the components connected to its master interface are also not accessed. The master interface of the bridge remains idle until a master accesses the bridge slave interface.

Bridges can also reduce the toggle rates of signals that are inputs to other master interfaces. These signals are typically `readdata`, `readdatavalid`, and `waitrequest`. Slave interfaces that support read accesses drive the `readdata`, `readdatavalid`, and `waitrequest` signals. A bridge inserts either a register or clock crossing FIFO between the slave interface and the master to reduce the toggle rate of the master input signals.

Disabling Logic

There are typically two types of low power modes: volatile and non-volatile. A volatile low power mode holds the component in a reset state. When the logic is reactivated, the previous operational state is lost. A non-volatile low power mode restores the previous operational state. This section discusses using either software-controlled or hardware-controlled sleep modes to disable a component in order to reduce power consumption.

Software-Controlled Sleep Mode

To design a component that supports software controlled sleep mode, create a single memory mapped location that enables and disables logic, by writing a zero or one. Use the register’s output as a clock enable or reset, depending on whether the component has non-volatile requirements. The slave interface must remain active during sleep mode so that the `enable` bit can be set when the component needs to be activated.

If multiple masters can access a component that supports sleep mode, you can use the mutex core available in Qsys to provide mutually exclusive accesses to your component. You can also build in the logic to re-enable the component on the very first access by any master in your system. If the component requires multiple clock cycles to re-activate, then it must assert wait request to prolong the transfer as it exits sleep mode.

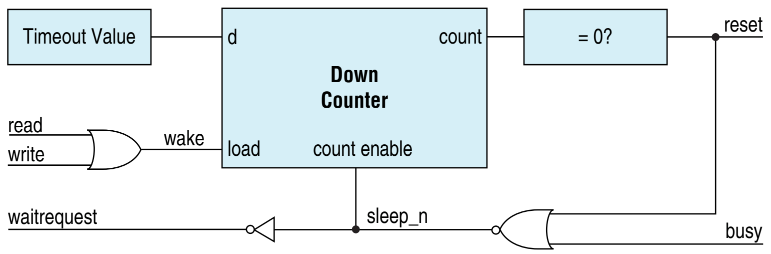
For more information about the mutex core, refer to the *Mutex Core* chapter of the *Embedded Peripherals IP User Guide*.

Hardware-Controlled Sleep Mode

You can implement a timer in your component that automatically causes the component to enter a sleep mode based on a timeout value specified in clock cycles between read or write accesses. Each access resets the timer to the timeout value. Each cycle with no accesses decrements the timeout value by one. If the counter reaches zero, the hardware enters sleep mode until the next access. Figure 9–26 provides a schematic for this logic. If restoring the component to an active state takes a long time, use a long timeout value so that the component is not continuously entering and exiting sleep mode.

The slave interface must remain functional while the rest of the component is in sleep mode. When the component exits sleep mode, the component must assert the `waitrequest` signal until it is ready for read or write accesses.

Figure 9–26. Hardware-Controlled Sleep Components



For more information on reducing power utilization, refer to *Power Optimization* in the *Quartus II Handbook.*

Design Examples

The following examples illustrate the resolution of Qsys system design challenges.

Avalon Pipelined Read Master Example

For a high throughput system using the Avalon-MM standard, you can design a pipelined read master that allows your system to issue multiple read requests before data returns. Pipelined read masters hide the latency of read operations by posting reads as frequently as every clock cycle. You can use this type of master when the address logic is not dependent on the data returning.

Design Requirements

You must carefully design the logic for the control and data paths of pipelined read masters. The control logic must extend a read cycle whenever the `waitrequest` signal is asserted. This logic must also control the master `address`, `byteenable`, and `read` signals. To achieve maximum throughput, pipelined read masters should post reads continuously as long as `waitrequest` is deasserted. While `read` is asserted, the address presented to the interconnect is stored.

The data path logic includes the readdata and readdatavalid signals. If your master can accept data on every clock cycle, you can register the data with the readdatavalid as an enable bit. If your master cannot process a continuous stream of read data, it must buffer the data in a FIFO. The control logic must stop issuing reads when the FIFO reaches a predetermined fill level to prevent FIFO overflow.

Refer to the *Avalon Interface Specifications* to learn more about the signals that implement an Avalon pipelined read master.

Expected Throughput Improvement

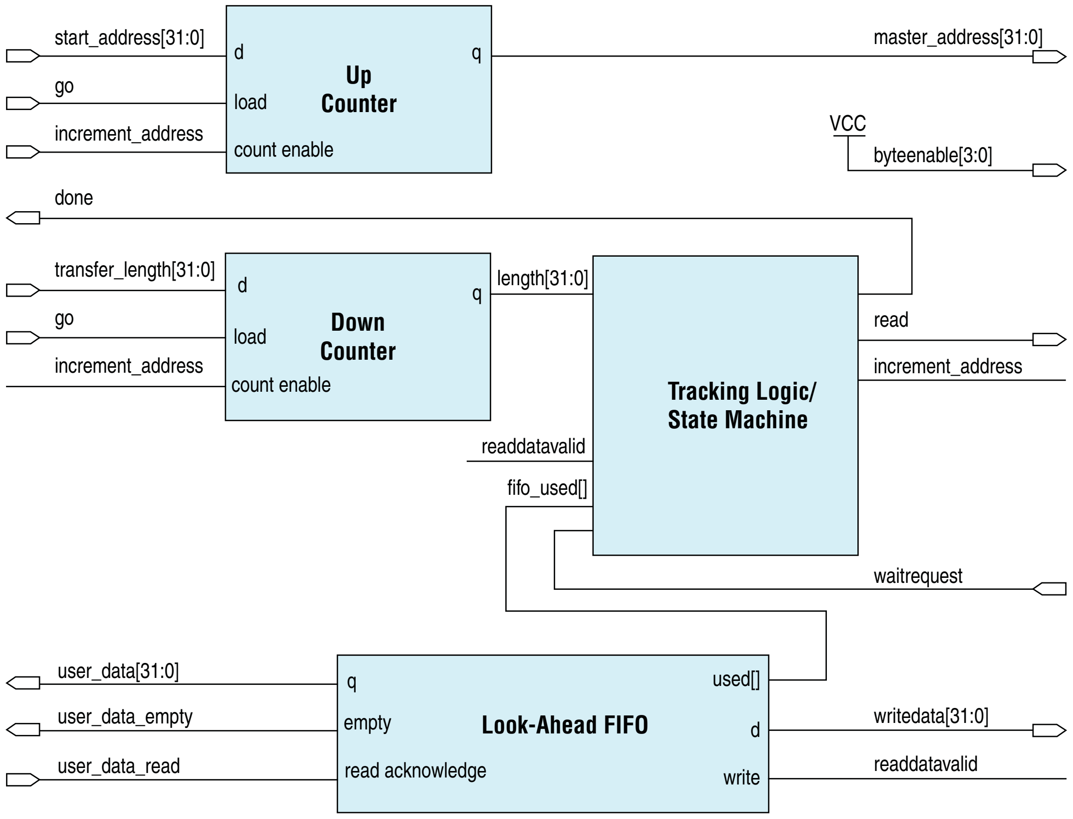
The throughput improvement that you can achieve with a pipelined read master is typically directly proportional to the pipeline depth of the interconnect and the slave interface. For example, if the total latency is two cycles, you can double the throughput by inserting a pipelined read master, assuming the slave interface also supports pipeline transfers. If either the master or slave does not support pipelined read transfers, then the interconnect asserts waitrequest until the transfer completes. You can also gain throughput when there are some cycles of overhead before a read response.

The “Increased Latency” on page 9–15 describes an example in which both the master and slave interfaces support pipelined read transfers. In this example, data can flow on a continuous stream after the initial latency. Where reads are not pipelined, the throughput is reduced. When both the master and slave interfaces support pipelined read transfers, data flows in a continuous stream after the initial latency. Figure 9–27 illustrates reads that are not pipelined. The system uses three cycles of latency for each read, achieving an overall throughput of 25%. Figure 9–20 shows reads that are pipelined. After the three cycles of latency, the data flows continuously.

You can use a pipelined read master that stores data in a FIFO to implement a custom DMA, hardware accelerator, or off-chip communication interface. Figure 9–27 shows a pipeline read master that stores data in a FIFO. The master performs word accesses that are word-aligned and reads from sequential memory addresses. The transfer length is a multiple of the word size. In Figure 9–27, the master performs word accesses that are word-aligned and reads from sequential memory addresses. The transfer length is a multiple of the word size.

Figure 9–27 shows a pipeline read master that stores data in a FIFO.

Figure 9–27. Pipelined Read Master



When the `go` bit is asserted, the master registers the `start_address` and `transfer_length` signals. The master begins issuing reads continuously on the next clock until the `length` register reaches zero. In this example, the word size is four bytes so that the address always increments by four and the length decrements by four. The `read` signal remains asserted unless the FIFO fills to a predetermined level. The `address` register increments and the `length` register decrements if the `length` has not reached 0 and a `read` is posted.

The master posts a read transfer every time the `read` signal is asserted and the `waitrequest` is deasserted. The master issues reads until the entire buffer has been read or `waitrequest` is asserted. An optional tracking block monitors the `done` bit. When the `length` register reaches zero, some reads are outstanding. The tracking logic prevents assertion of done until last `read` completes. The tracking logic monitors the number of reads posted to the interconnect so that it does not exceed the space remaining in the `readdata` FIFO. This logic includes a counter that verifies the following conditions are met:

- If a read is posted and `readdatavalid` is deasserted, the counter increments.
- If a read is not posted and `readdatavalid` is asserted, the counter decrements.

When the `length` register and the tracking logic counter reach zero, all the reads have completed and the `done` bit is asserted. The `done` bit is important if a second master overwrites the memory locations that the pipelined read master accesses. This bit guarantees that the reads have completed before the original data is overwritten.

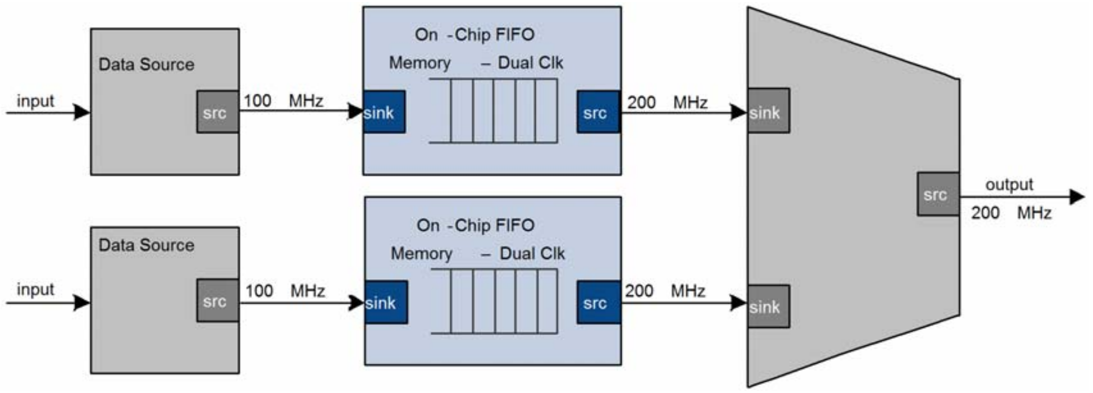
Multiplexer Examples

You can combine adapters with streaming components to create datapaths whose input and output streams have different properties. The following sections provide examples of datapaths in which the output stream is higher performance than the input stream. Figure 9–28 shows an output with double the throughput of each interface with a corresponding doubling of the clock frequency. Figure 9–29 doubles the data width. Figure 9–30 boosts the frequency of a stream by 10% by multiplexing input data from two sources.

Example to Double Clock Frequency

Figure 9–28 illustrates a datapath that uses the dual clock version of the on-chip FIFO memory and Avalon-ST channel multiplexer to merge the 100 MHz input from two streaming data sources into a single 200 MHz streaming output. This example increases throughput by increasing the frequency and combining inputs.

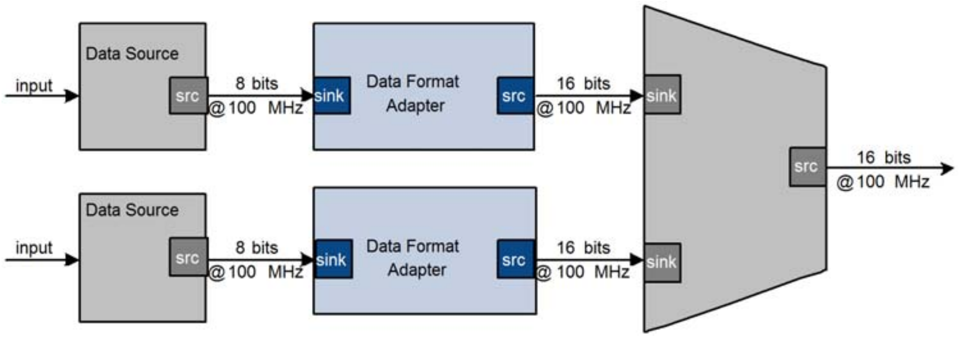
Figure 9–28. Datapath that Doubles the Clock Frequency



Example to Double Data Width and Maintain Frequency

Figure 9–29 illustrates a datapath that uses the data format adapter and Avalon-ST channel multiplexer to convert two, 8-bit inputs running at 100 MHz to a single 16-bit output at 100 MHz.

Figure 9–29. Datapath to Double Data Width and Maintain Original Frequency

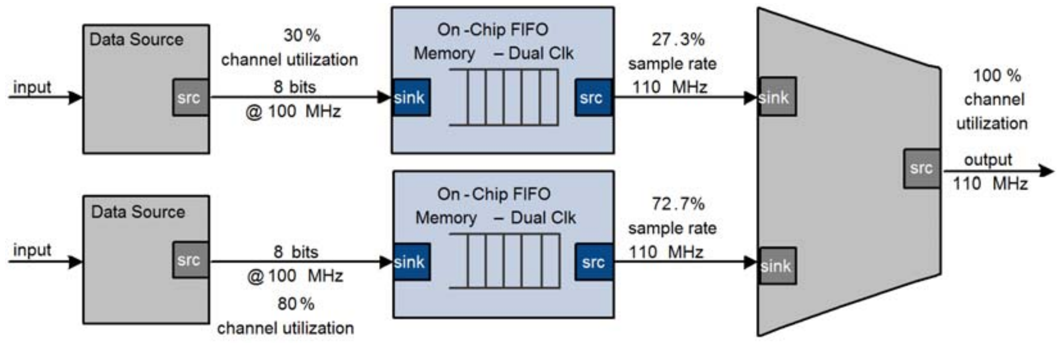


Example to Boost the Frequency

Figure 9–30 illustrates a datapath that uses the dual clock version of the on-chip FIFO memory to boost the frequency of input data from 100 MHz to 110 MHz by sampling two input streams at differential rates. In this example, the on-chip FIFO memory has an input clock frequency of 100 MHz and an output clock frequency of 110 MHz. The channel multiplexer runs at 110 MHz and samples one input stream 27.3 percent of the time and the second 72.7 percent of the time.

You do not need to know what the typical and maximum input channel utilizations are before this type of design. For example, if the first channel hits 50% utilization, the output stream exceeds 100% utilization.

Figure 9–30. Datapath to Boost the Clock Frequency



Conclusion

Recommendations presented in this chapter may improve your system's maximum clock frequency, concurrency and throughput, logic utilization, or even power utilization. When you design a Qsys system, use your knowledge of the design intent and goals to further optimize system performance beyond the automated optimization available within Qsys.

Document Revision History

Table 9–2 shows the revision history for this document.

Table 9–2. Document Revision History

Date	Version	Changes
May 2013	13.0.0	■ Added AMBA APB support.
November 2012	12.1.0	■ Added AMBA AXI4 support.
June 2012	12.0.0	■ Added AMBA AXI3 support.
November 2011	11.1.0	■ New document release.

For previous versions of the *Quartus II Handbook*, refer to the Quartus II Handbook Archive