6. Avalon Memory-Mapped Design Optimizations

ED51007-1.1

The Avalon® Memory-Mapped (Avalon-MM) system interconnect fabric is a flexible, partial crossbar fabric that connects master and slave components. Understanding and optimizing this system interconnect fabric in can help you create higher performance designs. When you use the Altera® system-on-a-programmable-chip (SOPC) design tool, SOPC builder automatically generates optimized interconnect logic to your specifications, saving you from time-consuming and error-prone task.

This chapter provides recommendations to optimize the performance, resource utilization, and power consumption of your Avalon-MM design. The following topics are discussed:

- Selecting Hardware Architecture
- Understanding Concurrency
- Increasing Transfer Throughput
- Increasing System Frequency
- Reducing Logic Utilization
- Reducing Power Utilization

One of the key advantages of FPGAs for system design is the high availability of parallel resources. SOPC Builder uses the parallel resources inherent in the FPGA fabric to maximize concurrency. You use the SOPC Builder GUI to specify the connectivity between blocks of your design. SOPC Builder automatically generates the optimal HDL from your specification.

Selecting Hardware Architecture

Hardware systems typically use one of four architectures to connect the blocks of a design:

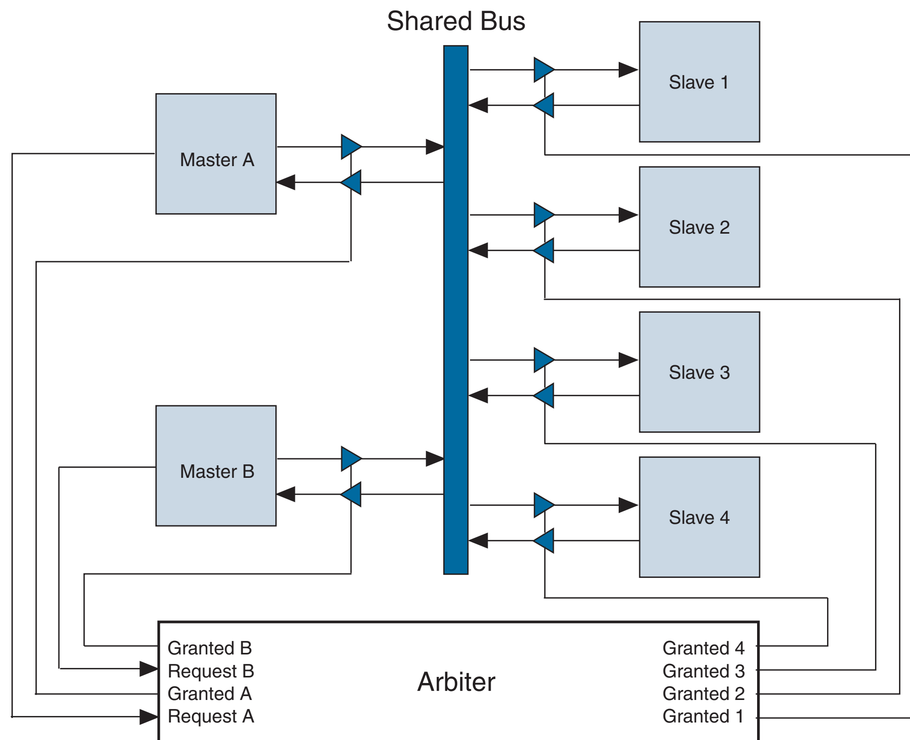
- Bus
- Full Crossbar Switch
- Partial Crossbar Switch
- Streaming

No single architecture can be used efficiently for all systems. The following sections discuss the characteristics, advantages and disadvantages of each of these interconnect architectures.

Bus

Bus architectures can achieve relatively high clock frequencies at the expense of little or no concurrency. Bus architectures connect masters and slaves using a common arbitration unit. The arbiter must grant a master access to the bus before a data transfer can occur. A shared bus architecture can lead to a significant performance penalty in systems with many bus masters because all masters compete for access to the shared bus rather than a particular slave device.

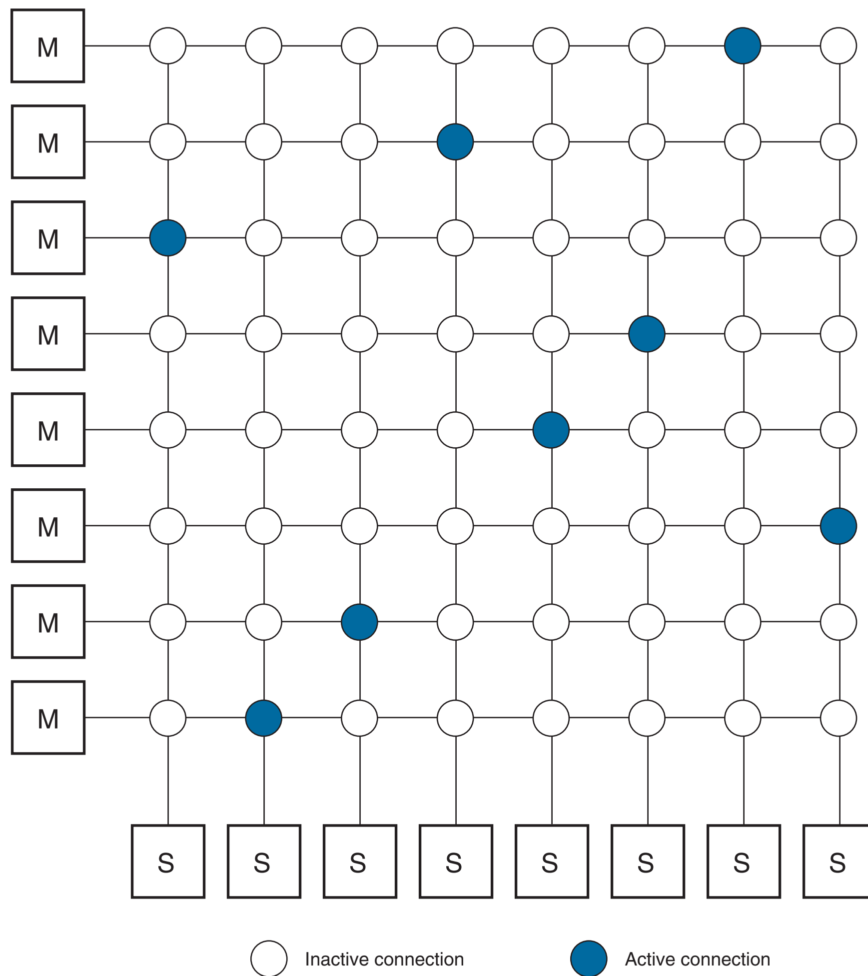
Figure 6–1. Bus Architecture



Full Crossbar Switch

Crossbar switches, unlike bus architectures, support concurrent transactions. A crossbar switch allows any number of masters to connect to any number of slaves. Networking and high performance computing applications typically use crossbars because they are flexible and provide high throughput. Crossbars are implemented with large multiplexers. The crossbar switch includes the arbitration function. Crossbars provide a high degree of concurrency. The hardware resource utilization grows exponentially as more masters and slaves are added; consequently, FPGA designs avoid large crossbar switches because logic utilization must be optimized.

Figure 6–2. Crossbar Switch



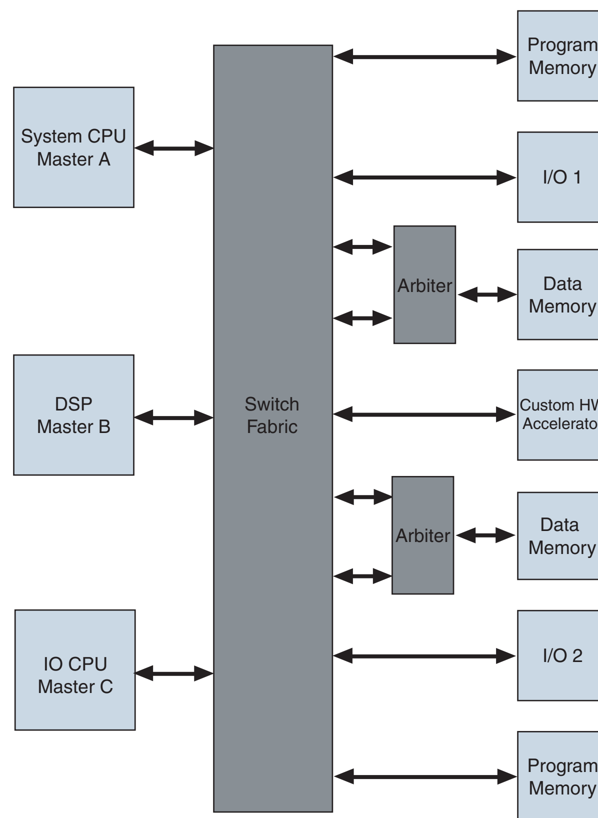
Partial Crossbar Switch

In many embedded systems, individual masters only require connectivity to a subset of the slaves so that a partial crossbar switch provides the optimal connectivity. There are two significant advantages to the partial crossbar switch:

- The reduction in connectivity results in system interconnect fabric that operates at higher clock frequencies
- The system interconnect fabric consumes fewer resources.

These two advantages make partial crossbar switches ideal for ASIC or FPGA interconnect structures. Figure 6–3 illustrates an SOPC Builder system with masters and slaves connected by a partial crossbar switch.

Figure 6–3. Partial Crossbar Switch – SOPC Builder System Interconnect



SOPC Builder generates logic that implements the partial crossbar system interconnect fabric using slave side arbitration. An arbiter included in the system interconnect fabric performs slave side arbitration. This architecture significantly reduces the contention for resources that is inherent in a shared bus architecture. The arbiter selects among all requesting masters so that unless two or more masters are requesting access in the same cycle, there is no contention. In contrast, a shared bus architecture requires all masters to arbitrate for the bus, regardless of the actual slave device to which the masters requires access.

In Figure 6–3, the system CPU has its own program memory and I/O; there is never any contention for these two slave resources. The system CPU shares a memory with the DSP master; consequently, there is a slave-side arbiter to control access. The DSP is the only master that accesses the custom hardware accelerator; there is no contention for this device. The DSP and I/O CPU master share a memory, and access is controlled by a slave-side arbiter. The I/O CPU master has it own program memory and I/O device.

The partial crossbar switch that SOPC Builder generates is ideal in FPGA designs because SOPC Builder only generates the logic necessary to connect masters and slaves that communicate with each other. Using SOPC Builder, you gain the performance of a switch fabric with the flexibility of an automatically generated interconnect architecture. Because SOPC Builder automatically generates the system interconnect fabric, you can regenerate it automatically if your system design changes.

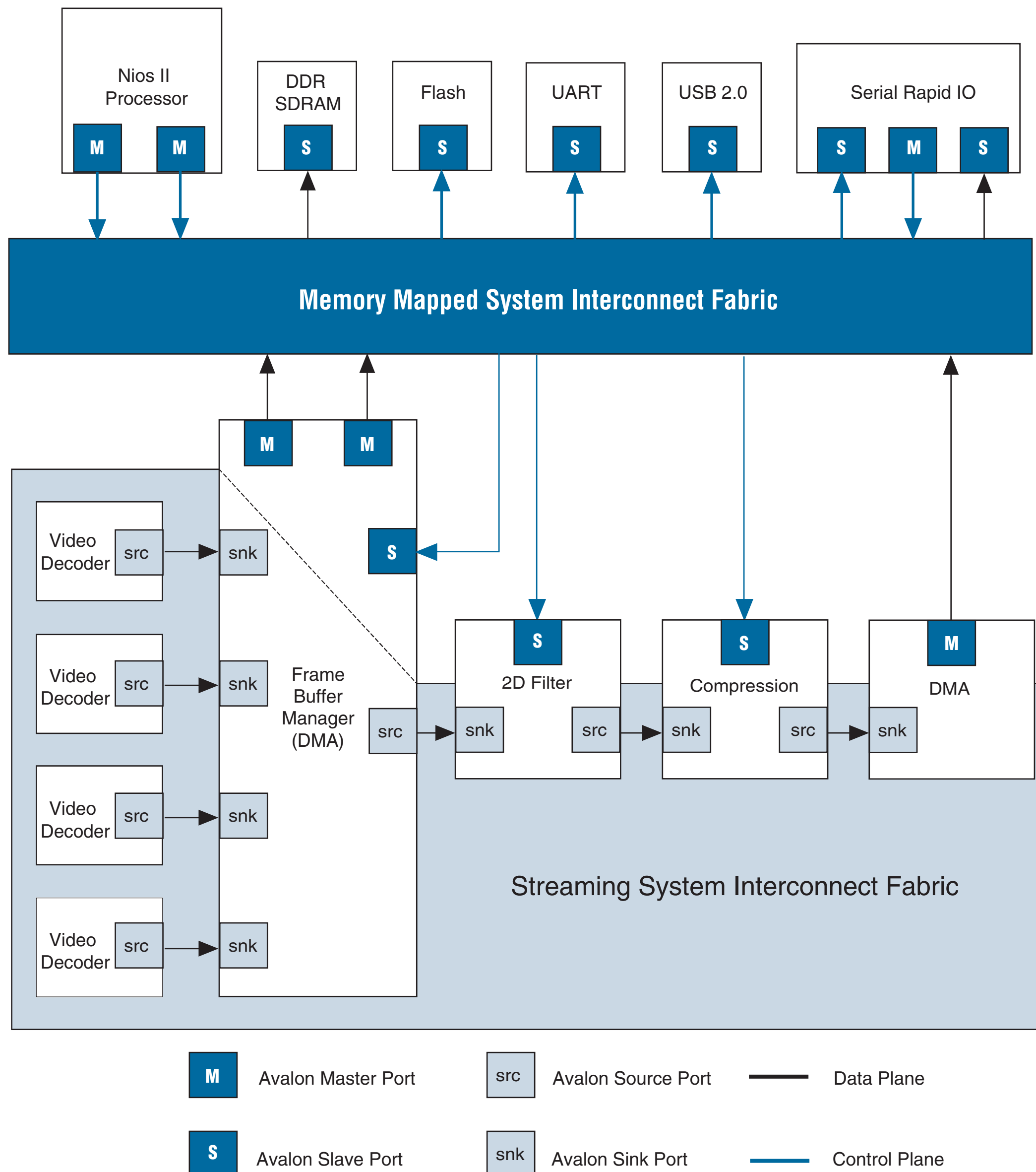
Streaming

Applications that require high speed data transfers use streaming interconnect. SOPC Builder supports Avalon Streaming (Avalon-ST) which creates point-to-point connections between source and sink components. Each streaming connection includes a single source and sink pair, eliminating arbitration. Because SOPC Builder supports both partial crossbar and streaming connections you can design systems that require the partial crossbar for the control plane, typically used to program registers and set up data transfers, and streaming for the data plane, typically used for high speed data transfers.

Full and partial crossbar switches and streaming architectures are all commonly used to implement data planes. The control plane usually includes a processor or state machine to control the flow of data. Full or partial crossbar switches or a shared bus architecture implement control planes.

SOPC Builder generates interconnect logic for both data and control planes. The system interconnect fabric connects Avalon-MM and Avalon-ST interfaces automatically based on connectivity information that you provide. Figure 6–4 shows a video processing application designed using SOPC Builder. This application uses Avalon-MM interfaces for control and Avalon-ST interfaces to transfer data. The video imaging pipeline includes five hardware blocks with Avalon-ST interfaces: a video decoder, a frame buffer, a two-dimensional filter, a compression engine and a direct memory access (DMA) master. All of the hardware blocks, with the exception of the video decoders, also include an Avalon-MM interface for control.

Figure 6–4. Video Data and Control Planes



Dynamic Bus Sizing

A common issue in system design is integration of hardware blocks of different data widths. SOPC Builder automatically adapts mixed width Avalon-MM components by inserting the correct adapter logic between masters and slaves of different widths. For example, if you connect a 32-bit master to a 16-bit slave, the system interconnect fabric creates an adapter that segments the 32-bit transfer into two, separate 16-bit transfers. In creating adapters, the system interconnect fabric employs byte enables to qualify each byte lane.

© June 2008 Altera Corporation

Understanding Concurrency

One of the key benefits of designing with FPGAs is the re-programmable parallel hardware. Because the underlying FPGA structure supports massive parallelism, the system interconnect fabric is tailored to utilize parallel hardware. You can use parallel hardware to create *concurrency* so that several computational processes are executing at the same time.

The following sections outline other design choices you can make to increase the concurrency in your system.

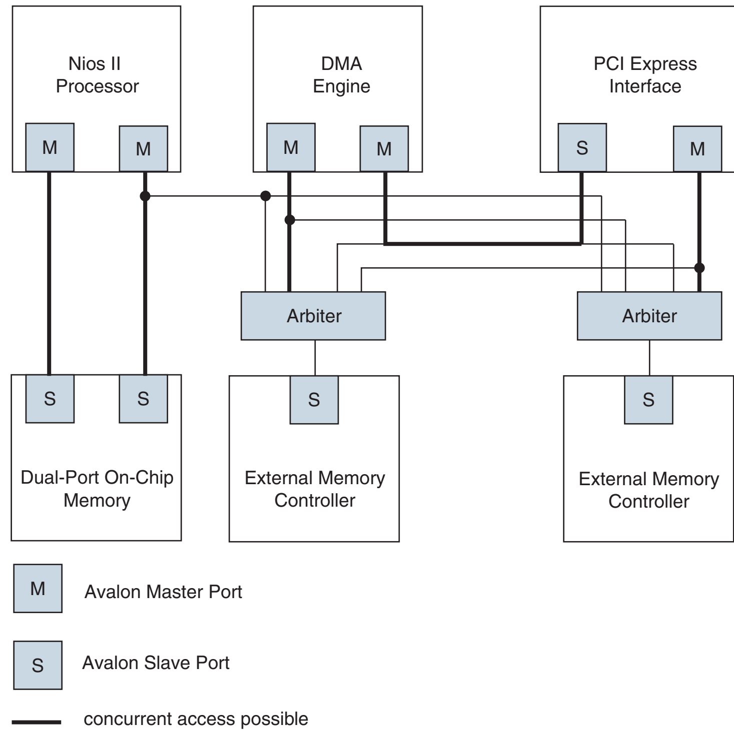
Create Multiple Masters

Your system must have multiple masters to take advantage of the concurrency that the system interconnect fabric supports. Systems that include a Nios® II processor always contain at least two masters, because the Nios II processor includes separate instruction and data masters. Master components typically fall into three main categories:

- General purpose processors, such as Nios II
- DMA engines
- Communication interfaces, such as PCI Express

Because SOPC Builder generates system interconnect fabric with slave side arbitration, every master in your system can issue transfers concurrently. As long as two or more masters are not posting transfers to a single slave, no master stalls. The system interconnect fabric contains the arbitration logic that determines wait states and drives the `waitrequest` signal to the master when a slave must stall. Figure 6–5 illustrates a system with three masters. The bold wires in this figure indicate connections that can be active simultaneously.

Figure 6–5. Multi Master Parallel Access



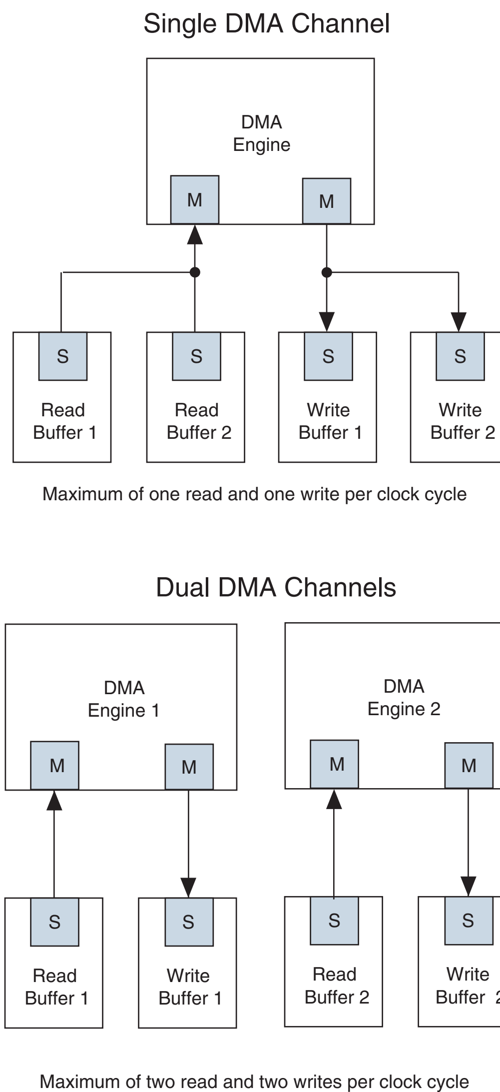
Create Separate Datapaths

Concurrency is limited by the number of masters sharing any particular slave because the system interconnect fabric uses slave side arbitration. If your system design requires higher data throughput, you can increase the number of masters and slaves to increase the number of transfers that occur simultaneously.

Use DMA Engines

DMA engines also increase data throughput. Because a DMA engine transfers all data between a programmed start and end address without any programmatic intervention, the data throughput is dictated by the components connected to the DMA. The factors that affect data throughput include data width and clock frequency. By including more DMA engines, a system can sustain more concurrent read and write operations as Figure 6–6 illustrates.

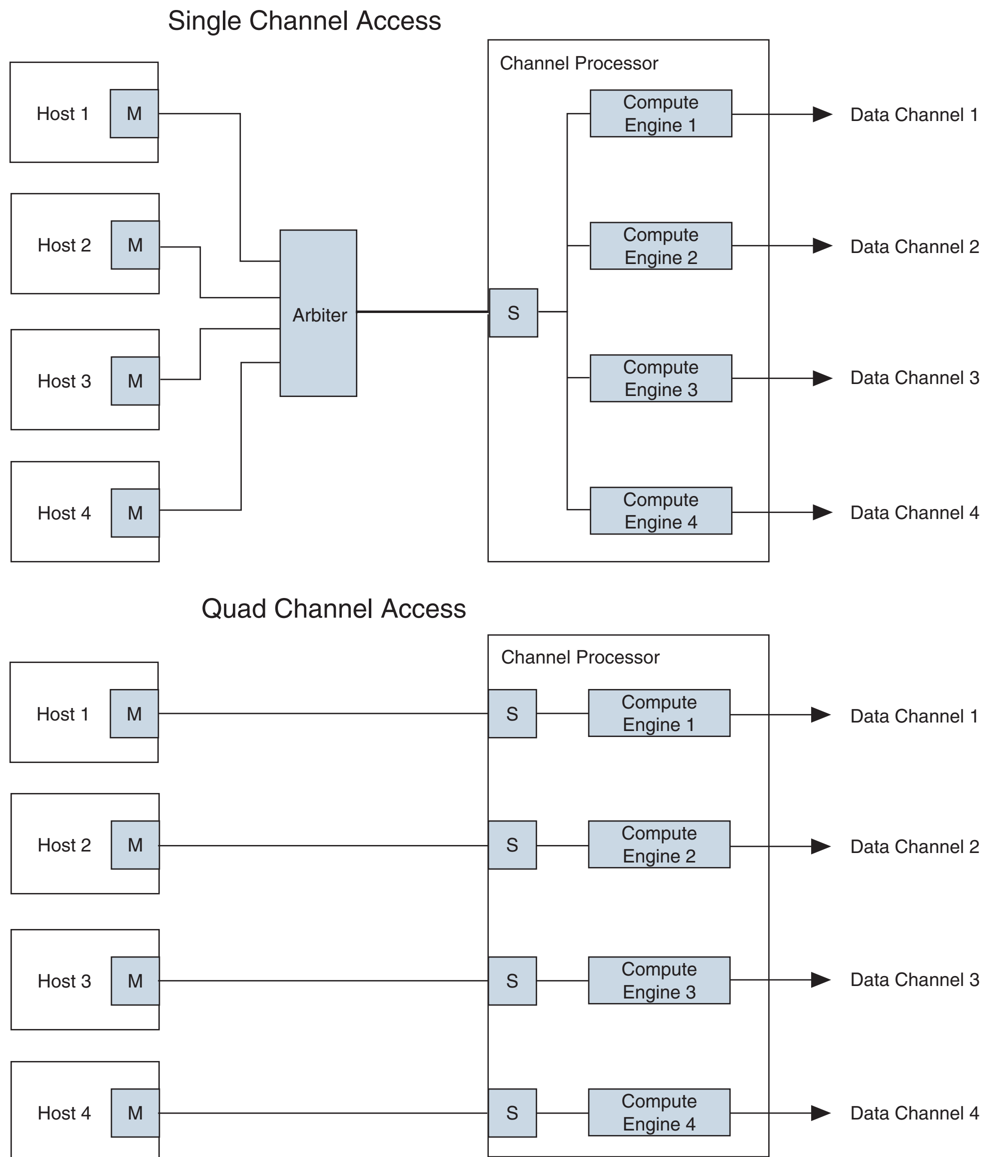
Figure 6–6. Single or Dual DMA Channels



Include Multiple Master or Slave Ports

Creating multiple slave ports for a particular function increases the concurrency in your design. Figure 6–7 illustrates two channel processing

Figure 6–7. Before and After Separate Slaves



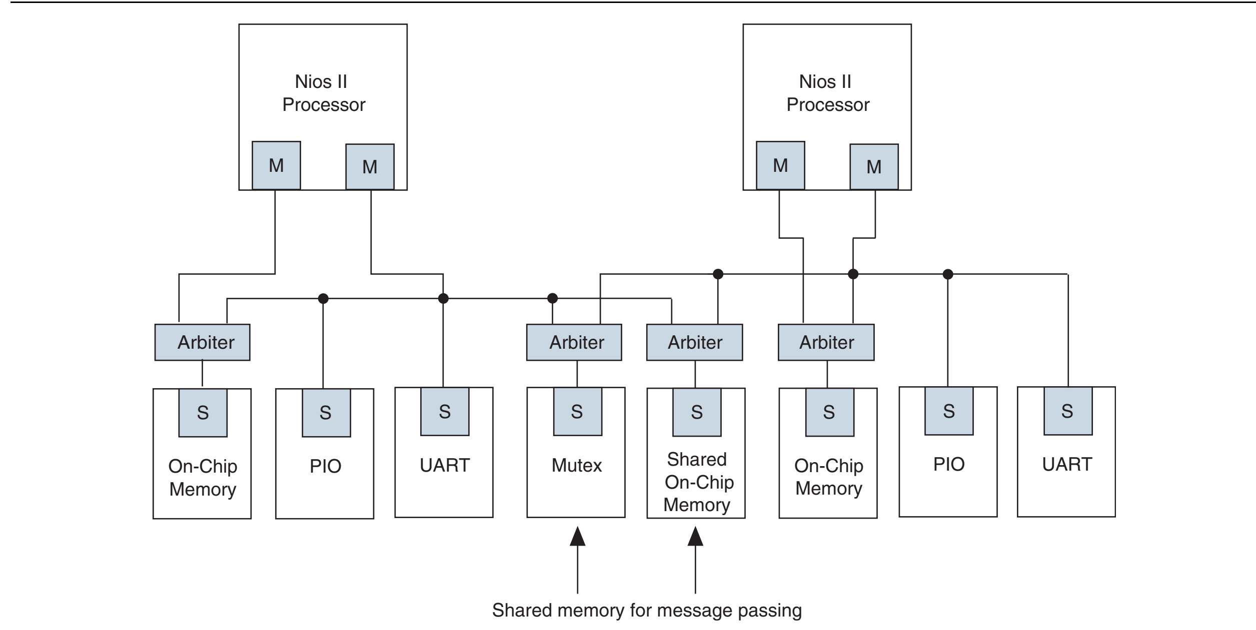
systems. In the first, four hosts must arbitrate for the single slave port of the channel processor. In the second, each host drives a dedicated slave port, allowing all masters to access the component's slave ports simultaneously.

Create Separate Sub-Systems

You can also use hierarchy to sub-divide a system into smaller, more manageable sub-systems. This form of concurrency is implemented by limiting the number of slaves to which a particular master connects. You can create multiple independent sub-systems within a single SOPC Builder system. When communication between sub-systems is necessary, you can use shared memory, message passing, or FIFOs to transfer information.

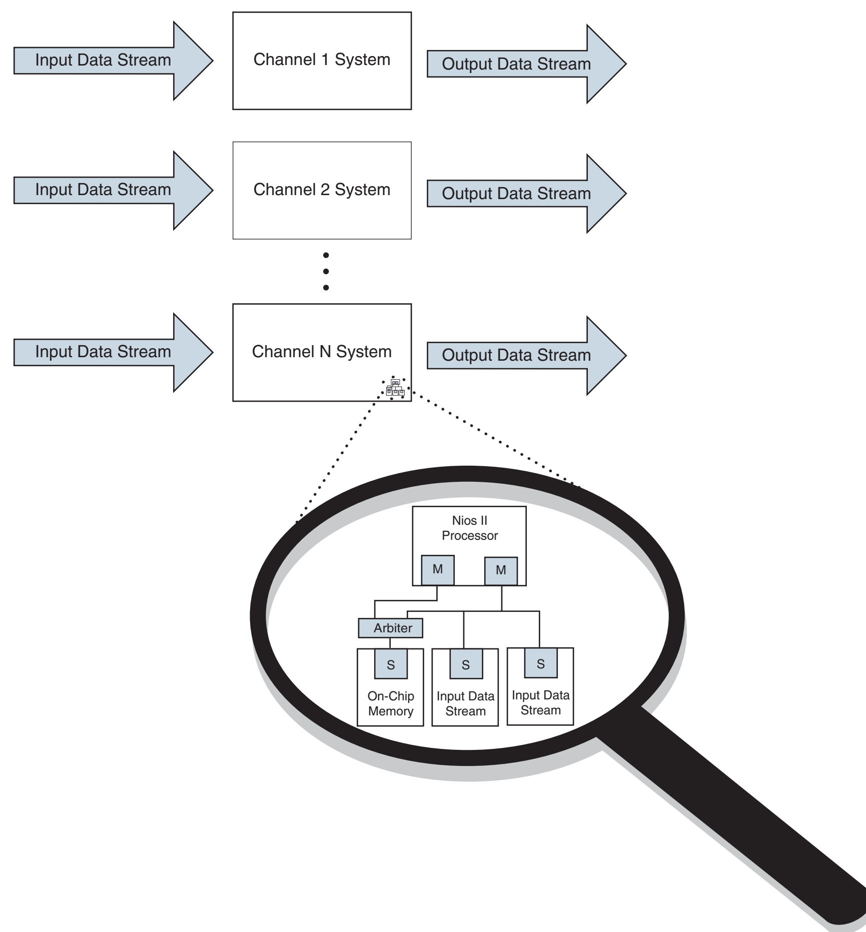
For more information, refer to *Creating Multiprocessor Nios II Systems Tutorial* and *Multiprocessor Coordination Peripherals*.

Figure 6–8. Message Passing



Alternatively, if the subsystems are identical, you can design a single SOPC Builder system and instantiate it multiple times in your FPGA design. This approach has the advantage of being easier to maintain than a heterogeneous system. In addition, such systems are easier to scale because once you know the logic utilization and efficiency of a single instance, you can estimate how much logic is necessary for multiple subsystems. Systems that process multiple data channels are frequently designed by instantiating the same sub-system for each channel.

Figure 6–9. Multi-Channel System



Increasing Transfer Throughput

Increasing the transfer efficiency of the master and slave ports in your system increases the throughput of your design. Designs with strict cost or power requirements benefit from increasing the transfer efficiency because less expensive, lower frequency devices can be used. At the other end of the spectrum, designs requiring high performance also benefit from increased transfer efficiency because it improves the performance of frequency–limited hardware.

Using Pipelined Transfers

Pipelined transfers increase the read efficiency by allowing a master to post multiple reads before data from the earlier reads returns. Masters that support pipelined transfers post transfers continuously, relying on the `readdatavalid` signal to indicate valid data. Slaves support pipelined transfers by including the `readdatavalid` signal or operating with a fixed read latency.

Maximum Pending Reads

SOPC Builder updates the maximum pending reads property when it generates the system interconnect fabric. If you create a custom component with a slave port supporting reads, you must specify this value in the Component Editor. This value represents the maximum number of read transfers your pipelined slave component can handle. If the number of reads presented to the slave port exceeds the maximum pending reads parameter your component must assert `waitrequest`.

Selecting the Maximum Pending Reads Value

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 longest delay. For example, if your pipelined component has two modes, one requiring two clock cycles and the other five, set the maximum pending reads value to five. Doing so allows your component to pipeline five transfers, eliminating dead cycles after the initial five-cycle latency.

Another way to determine the correct value for the maximum pending reads parameter is to monitor the number of reads that are actually pending during system simulation or while running the actual 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 assert `waitrequest` frequently. If you run this experiment with the actual hardware, you can use a logic analyzer or built-in monitoring hardware to observe your component.

Overestimating Versus Underestimating the Maximum Pending Reads Value

Choosing the correct value for the maximum pending reads value of your custom pipelined read component is very important. If you underestimate the maximum pending reads value you either loose data or cause a master port to stall indefinitely. The system interconnect fabric has no timeout mechanism to handle long delays.

The maximum pending reads value dictates the depth of the `readdatavalid` FIFO inserted into the system interconnect fabric for each master connected to the slave. Because this FIFO is only one bit wide it does not consume significant hardware resources. Overestimating the maximum pending reads value for your custom component results in a slight increase in hardware utilization. For these reasons, if you are not sure of the optimal value, it is better to overestimate this value.

Pipelined Read Masters

A pipelined read master can post multiple reads before data returns. Pipelined read masters hide the latency of read operations by posting reads as frequently as every clock cycle. A pipelined read master can prefetch data when no data dependencies are present. Examples of common pipelined read masters include the following:

- DMA engines
- Nios II processor (with a cache line size greater than four bytes)
- C2H read masters

Requirements

The logic for the control and datapaths of pipelined read masters must be carefully designed. 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 system interconnect fabric is stored.

The datapath logic includes the `readdata` and `readdatavalid` signals. If your master can return data on every clock cycle, you can register the data using `readdatavalid` as the enable bit. If your master cannot handle 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.

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

Throughput Improvement

The throughput gain that you achieve by using a pipelined read master is typically directly proportional to the latency of the slave port. For example, if the read latency is two cycles, you can double your throughput using a pipelined read master, assuming the slave port also supports pipeline transfers. If either the master or slave does not support pipelined read transfers then the system interconnect fabric asserts `waitrequest` until the transfer completes.

When both the master and slave ports support pipelined read transfers, data flows in a continues stream after the initial latency. Figure 6–10 illustrates the case where reads are not pipelined. The system pays a penalty of 3 cycles latency for each read, making the overall throughput 25 percent. Figure 6–11 illustrates the case where reads are pipelined. After the initial penalty of 3 cycles latency, the data flows continuously.

Figure 6–10. Low Efficiency Read Transfer

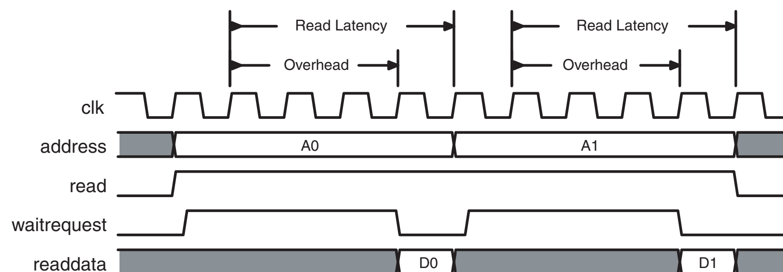
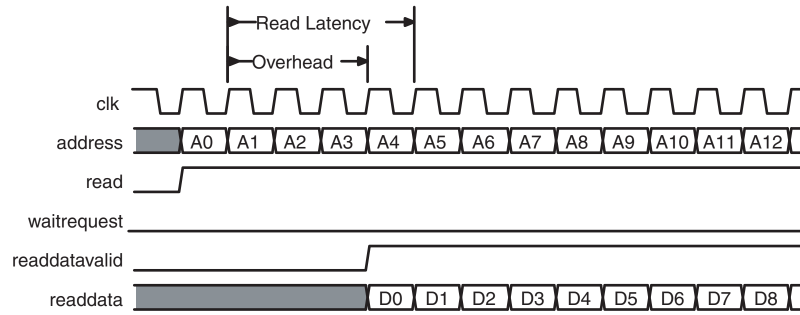


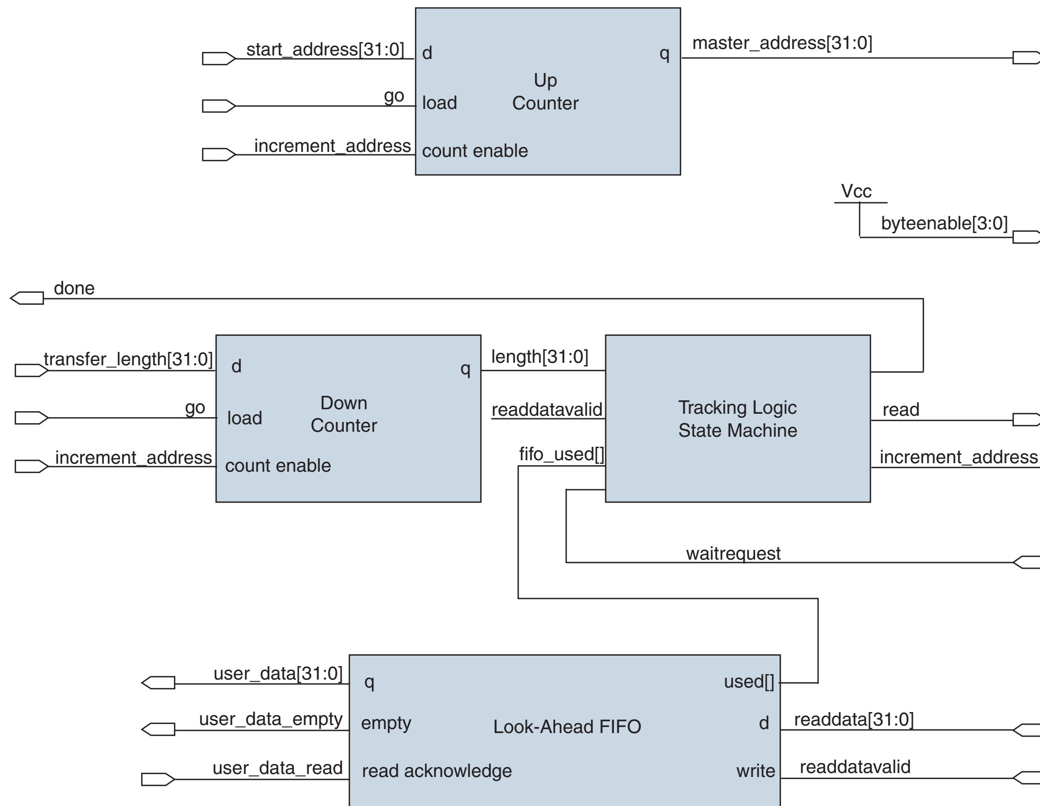
Figure 6–11. High Efficiency Read Transfer



Pipelined Read Master Example

Figure 6–12 illustrates a pipelined read master that stores data in a FIFO that can be used to implement a custom DMA, hardware accelerator, or off-chip communication interface. To simplify the design, the control and data logic are separate. 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 6–12. Latency Aware Master



When the go bit is asserted, the master registers the start_address and transfer_length signals. The master begins issuing reads on the next clock and does not stop until the length register reaches zero. In this example, the word size is 4 bytes so that the address always increments by 4 and the length decrements by 4. 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 times the done bit. When the length register reaches 0, some reads will be outstanding. The tracking logic guarantees that done is not asserted until the last read completes. The tracking logic monitors the number of reads posted to the system interconnect fabric so that is does not exceed the space remaining in the readdata FIFO. This logic includes a counter that counts if 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 0, 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 all the reads have completed before the original data is overwritten.

To learn more about creating Avalon-MM masters refer to the following design examples and documentation:

- *Nios II Embedded Processor Design Examples*
- *Developing Components for SOPC Builder* in volume 4 of the *Quartus II Handbook*.

Arbitration Shares and Bursts

Arbitration shares provide control over the arbitration process. By default, the arbitration algorithm provides equal fairness, with all masters receiving one share. You can tune 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.

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 in a round robin fashion. This mechanism prevents a master from using arbitration cycles if it cannot post back-to-back transfers.

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 8, it is guaranteed arbitration for 8 write cycles.

Differences between Arbitration Shares and Bursts

The following three key characteristics distinguish between 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` signal for fifty cycles, all other masters continue to wait for access during this stalled period.

To avoid wasting 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 burst counts equal to the amount of data that is ready. For example, if you have created a custom bursting write master with a maximum burst count of 8, but only 3 words of data are ready, you can simply present a burst count of 3. This strategy does not result in optimal use of the system bandwidth; however, it prevents starvation for other masters in the system.

Sequential Addressing

By definition, a burst transfer includes a base address and a burst count. The burst count 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, there are occasions when a master must access non-sequential addresses. Consequently, a bursting master must set the burst count to the number of sequential addresses and then reset the burst count 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 system interconnect fabric for every read or write transaction.

Burst Adapters

SOPC Builder allows you to create systems that mix bursting and non-bursting master and slave ports. It also allows you to connect bursting master and slave ports that support different maximum burst lengths. In order to support all these cases, SOPC Builder generates burst adapters when appropriate.

SOPC Builder inserts a burst adapter whenever a master port burst length exceeds the burst length of the slave port. SOPC Builder assigns non-bursting masters and slave ports 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 ports.

Choosing Interface Types

To avoid inefficient transfers, custom master or slave ports must use the appropriate interfaces. The are three possible interface types: simple, pipelined and burst. Each is described below:

Simple

Simple interfaces 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 ports. In SOPC Builder, the PIO, UART, and Timer include slave ports that operate at peak efficiency using simple transfers.

When designing a custom component, Altera recommends that you start with a simple interface. If performance becomes an issue, you can modify the component to support either pipelined reads or bursting.

Pipelined

In many systems, read throughput becomes inadequate if simple reads are used. If your system requires high read throughput and is not overly sensitive to read latency, then your component can implement pipelined transfers. If you define a component with a fixed read latency, SOPC Builder automatically provides the logic necessary to support pipelined reads. If your component has a variable latency response time, use the readdatavalid signal to indicate valid data. SOPC Builder implements a `readdatavalid` FIFO to handle the maximum number of pending read requests.

To use components that support pipelined read transfers efficiently, your system must contain pipelined masters. Refer to the "Pipelined Read Master Example" on page 6–15 for an example of a pipelined read master.

Burst

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

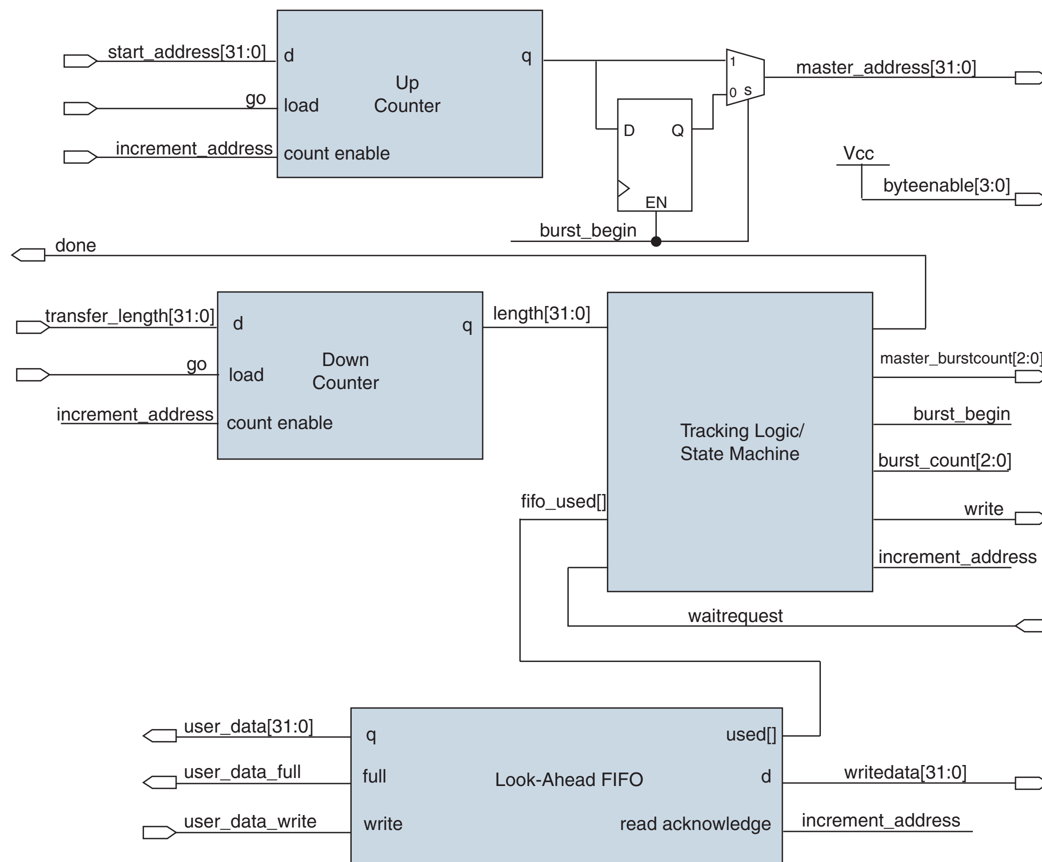
Altera recommends that you design a burst-capable slave port if you know that your component requires sequential transfers to operate efficiently. Because DDR SDRAM memories incur a penalty when switching banks or rows, performance improves when they are accessed sequentially using bursts.

Any shared address and data bus architecture also benefits from bursting. Whenever an address is transferred over a shared address and data bus, the throughput of the data transfer is reduced. Because the address phase adds overhead, using large bursts increases the throughput of the bus.

Burst Master Example

Figure 6–13 illustrates 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 6–13, the master performs word accesses and writes to sequential memory locations.

Figure 6–13. Bursting Write Master



When `go` is asserted, the `address` and `length` are registered. On the following 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 system interconnect fabric. 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 the master should only stall 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.

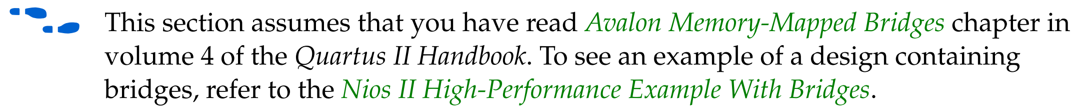
The `address` register increments after every word transfer, and the `length` register decrements after every word transfer. The address remains constant throughout the burst. Because a transfer is not guaranteed to complete on burst boundaries, additional logic is necessary to recognize the completion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

The *Accelerated FIR with Built-in Direct Memory Access Example*, includes a pipelined read master and bursting write master similar to those used in Figure 6–13.

Increasing System Frequency

In SOPC Builder, you can introduce bridges to reduce the amount of logic that SOPC Builder generates and increase the clock frequency.

In SOPC Builder, you can use bridges to control the system interconnect topology. Bridges allow you to subdivide the system interconnect fabric, giving you more control over pipelining and clock crossing functionality.



This section assumes that you have read *Avalon Memory-Mapped Bridges* chapter in volume 4 of the *Quartus II Handbook*. To see an example of a design containing bridges, refer to the *Nios II High-Performance Example With Bridges*.

Use Pipeline Bridges

The pipeline bridge contains Avalon-MM master and slave ports. Transfers to the bridge slave port are propagated to the master port which connects to components downstream from the bridge. You have the option to add the following pipelining features between the bridge ports:

- Master-to-Slave Pipelining
- Slave-to-Master Pipelining
- waitrequest Pipelining

The pipeline bridge options can increase your logic utilization and read latency. As a result, you should carefully consider the effects of the following options described in this section.

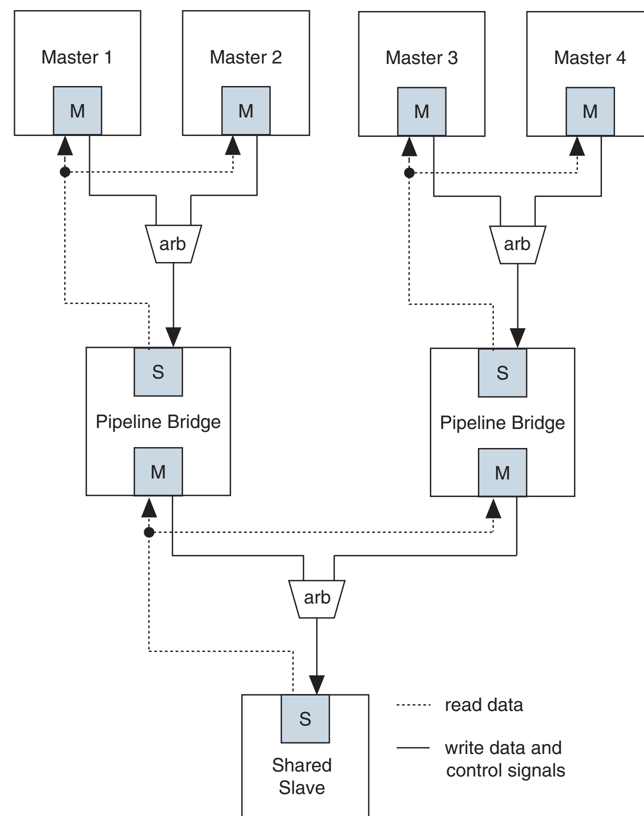
Master-to-Slave Pipelining

Master-to-slave pipelining is advantageous when many masters share a slave device. The arbitration logic for the slave port must multiplex the `address`, `writedata`, and `burstcount` signals. The multiplexer width increases as the number of masters connecting to a single slave port increases, causing timing problems if the slave component does not register the input signals.

This option is helpful if the `waitrequest` signal becomes part of a critical timing path. Because `waitrequest` is dependent on arbitration logic, which is driven by the master address, enabling master-to-slave pipelining helps pipeline this path. If the `waitrequest` signal remains a part of a critical timing path, you should explore using the `waitrequest` pipelining feature of the pipelined bridge.

If a single pipeline bridge provides insufficient improvement, you can instantiate this bridge multiple times, in a binary tree structure, to increase the pipelining and further reduce the width of the multiplexer at the slave port as Figure 6–14 illustrates.

Figure 6–14. Tree of Bridges



Slave-to-Master Pipelining

Slave-to-master pipelining is advantageous for masters that connect to many slaves that support read transfers. The system interconnect fabric inserts a multiplexer for every read datapath 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, you can use multiple bridges to improve f_{MAX} using a binary tree structure.

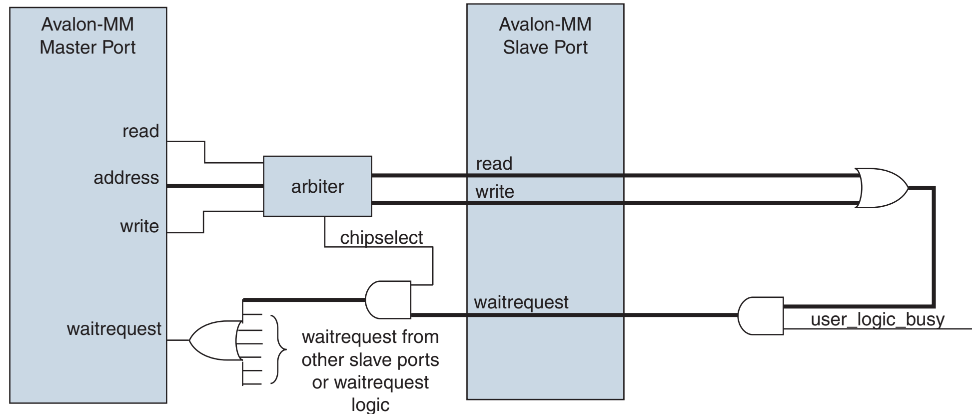
waitrequest Pipelining

`waitrequest` pipelining can be advantageous in cases where a single master connects to many slaves. Because slave components and system interconnect fabric drive `waitrequest`, the logic depth grows as the number of slaves connected to the master increases. `waitrequest` pipelining is also useful when multiple masters connect to a single slave, because it pipelines the arbitration logic.

In many cases `waitrequest` is a combinational signal because it must be asserted during the same cycle that a `read` or `write` transaction is posted. Because `waitrequest` is typically dependent on the master `read` or `write` signals, it creates a timing path from the master to the slave and back to the master. Figure 6–15 illustrates this round-trip path with the thick wire.

To prevent this round-trip path from impacting the f_{MAX} of your system, you can use master-to-slave pipelining to reduce the path length. If the `waitrequest` signal remains part of the critical timing path, you can consider using the `waitrequest` pipelining feature. Another possibility is to register the `waitrequest` signal and keep it asserted, even when the slave is not selected. When the slave is selected, it has a full cycle to determine whether it can respond immediately.

Figure 6–15. Typical Slow Waitrequest Path



Use a Clock Crossing Bridge

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

Because FIFOs are used for the clock domain crossing, you gain the added benefit of data buffering when using the clock crossing bridge. The 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.

Increasing Component Frequencies

One of the most common uses of the clock crossing bridge is 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 a high performance, you can achieve a higher f_{MAX} for this portion of the design.

Reducing Low-Priority Component Frequencies

The majority of components included in embedded designs do not benefit from operating at higher frequencies. Examples components that do not require high frequencies are timers, UARTs, and JTAG controllers.

 © June 2008 Altera Corporation

When you compile a design using the Quartus II design software, the fitter places your logic into regions of the FPGA. The higher the clock frequency of your system, the longer a compilation takes. The compilation takes more time 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 datapaths.

Consequences of Using Bridges

Before using the pipeline or clock crossing bridges in your design, you should carefully consider their effects. The bridges can have any combination of the following effects on your design:

- Increased Latency
- Limited Concurrency
- Address Space Translation

Depending on your system, these effects could be positive or negative. You can use benchmarks to test your system before and after inserting bridges to determine their effects. The following sections discuss the effects of adding bridges.

Increased Latency

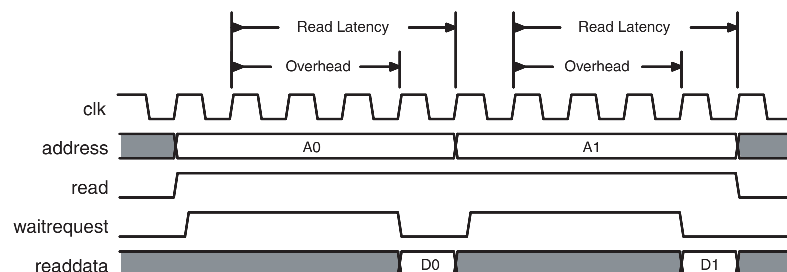
Adding either type of bridge to your design has an effect on the read latency between the master and the slave. Depending on the system requirements and the type of master and slave this latency increase may or may not be acceptable in your design.

Acceptable Latency Increase

For the 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 to a significant degree because it is very small compared to the length data transfer.

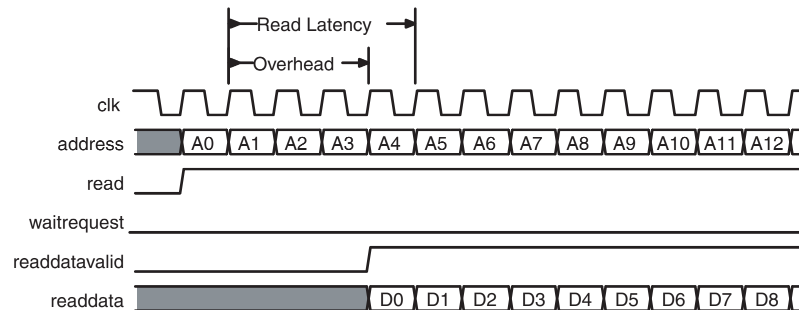
For example, if you use a pipelined read master such as a DMA controller to read data from a component with a fixed read latency of 4 clock cycles but only perform a single word transfer, the overhead is 3 clock cycles out of the total 4. The read throughput is only 25%.

Figure 6–16. Low Efficiency Read Transfer



On the other hand, if 100 words of data are transferred without interruptions, the overhead is 3 cycles out of the total of 103 clock cycles, corresponding to a read efficiency of approximately 97%. 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 increases the f_{MAX} by 5%. The overall throughput improves. As the number of words transferred increases the efficiency will increase to approach 100%, whether or not a pipeline bridge is present.

Figure 6–17. High Efficiency Read Transfer

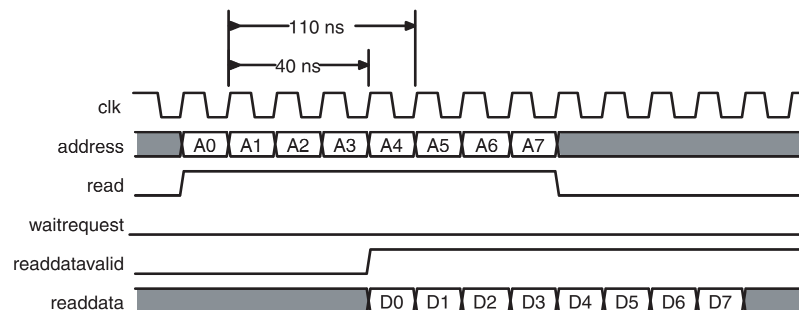


Unacceptable Latency Increase

Processors are sensitive to high latency read times. They typically fetch data for use in calculations that cannot proceed until it arrives. Before adding a bridge to the datapath of a processor instruction or data master, determine whether the clock frequency increase justifies the added latency.

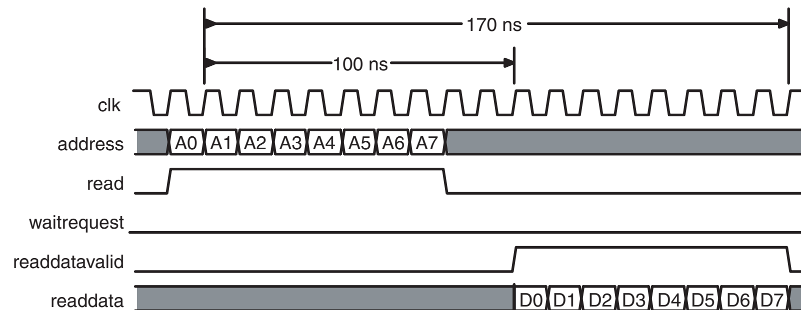
The following example design illustrates this point. The original design contains a Nios II processor and memory operating at 100 MHz. The Nios II processor instruction master has a cache memory with a read latency of 4 cycles. Eight sequential words of data return for each read. At 100 MHz, the first read takes 40 ns to complete. Each successive word takes 10 ns so that 8 reads complete in 110 ns.

Figure 6–18. Eight Reads with Four Cycles Latency



Adding a clock crossing bridge allows the memory to operate 125 MHz. However, this increase in frequency is negated by the increase in latency for the following reasons. Assume that the clock crossing bridge adds 6 clock cycles of latency at 100 MHz. The memory still operates with a read latency of 4 clock cycles; consequently, the first read from memory takes 100 ns and each successive word takes 10 ns because reads arrive at the processor's frequency, 100 MHz. In total, all 8 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 6–19. Eight Reads with Ten Cycles latency

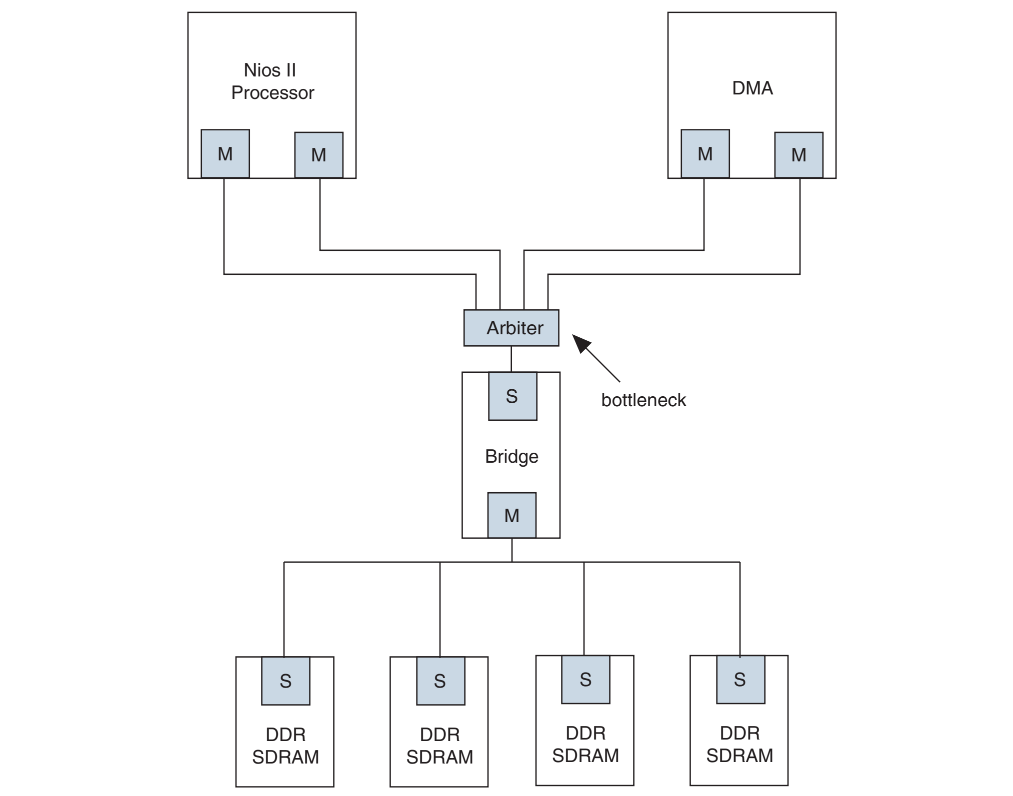


Limited Concurrency

Placing an Avalon bridge between multiple Avalon-MM master and slave ports limits the number of concurrent transfers your system can initiate. This limitation is no different than connecting multiple master ports to a single slave port. The bridge's slave port is shared by all the masters and arbitration logic is created as a result. If the components placed behind a bridge are infrequently accessed, this concurrency limitation may be acceptable.

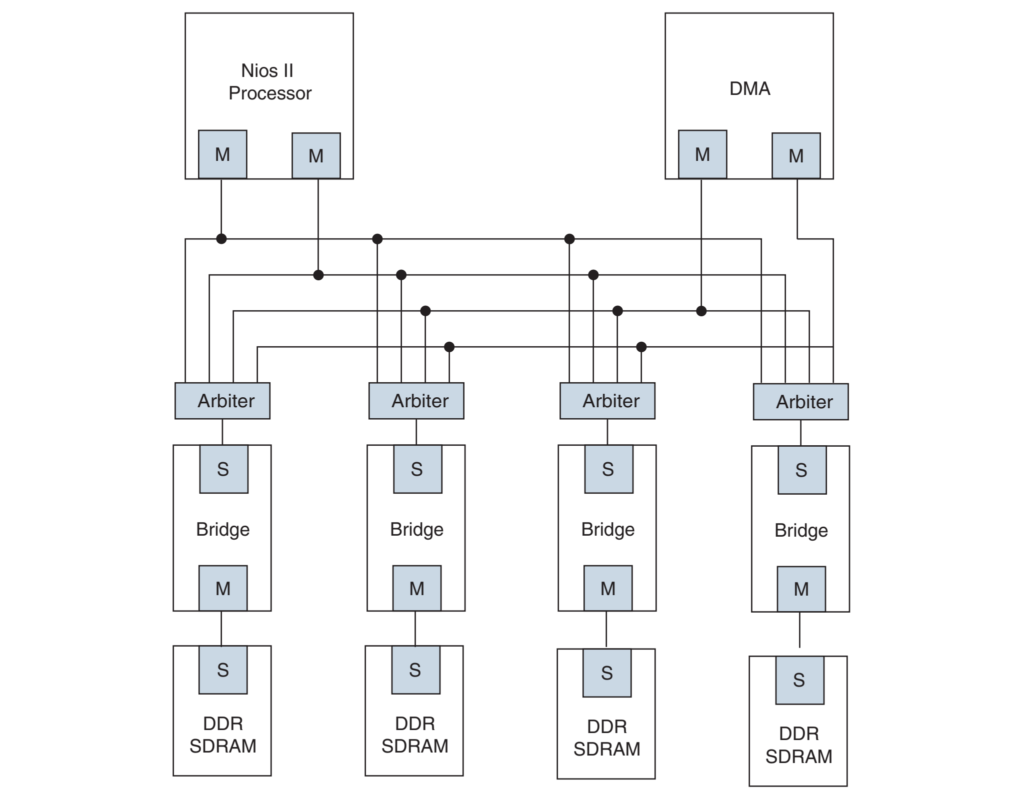
Bridges can have a severe negative impact on system performance if used inappropriately. For instance, if multiple memories are used by several masters, you should not place them all behind a bridge. The bridge limits the memory performance by preventing concurrent memory accesses. Placing multiple memory components behind a bridge, causes the separate slave interfaces to appear as one monolithic memory to the masters accessing the bridge; they must all access the same slave port. Figure 6–20 illustrates this configuration.

Figure 6–20. Poor Memory Pipelining



If the f_{MAX} of your memory interfaces is low, you can place each memory behind its own bridge, which increases the f_{MAX} of the system without sacrificing concurrency as Figure 6–21 illustrates.

Figure 6–21. Efficient Memory Pipelining



Address Space Translation

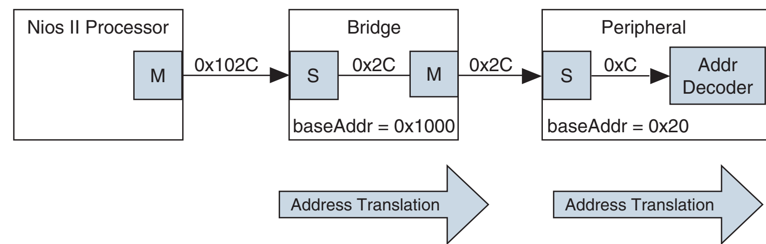
The slave port of a pipeline or clock crossing bridge has a base address and address span. You can set the base address or allow SOPC Builder to set it automatically. The slave port's address is the base offset address of all the components connected to the bridge. The address of components connected to the bridge is the sum of the base offset and that component's address. The address span of the bridge is automatically calculated by SOPC Builder based on the address range of all the components connected to it.

Address Shifting

The master port of the bridge only drives the address bits that represent the offset from the base address of the bridge slave port. Any time an Avalon-MM master accesses a slave through a bridge, both addresses must be added together, otherwise the transfer fails. Clicking the **Address Map** button in SOPC Builder displays the addresses of the slaves connected to each master taking into account any address translations caused by bridges in the system.

Figure 6–22 illustrates how this address translation takes place. In this example, the Nios II processor connects to a bridge located at base address 0x1000. A slave connects to the bridge master port at an offset of 0x20 and the processor performs a write transfer to the fourth 32-bit word within the slave. Nios II drives the address 0x102C to system interconnect fabric which lies within the address range of the bridge. The bridge master port drives 0x2C which lies within the address range of the slave and the transfer completes

Figure 6–22. Avalon Bridge Address Translation

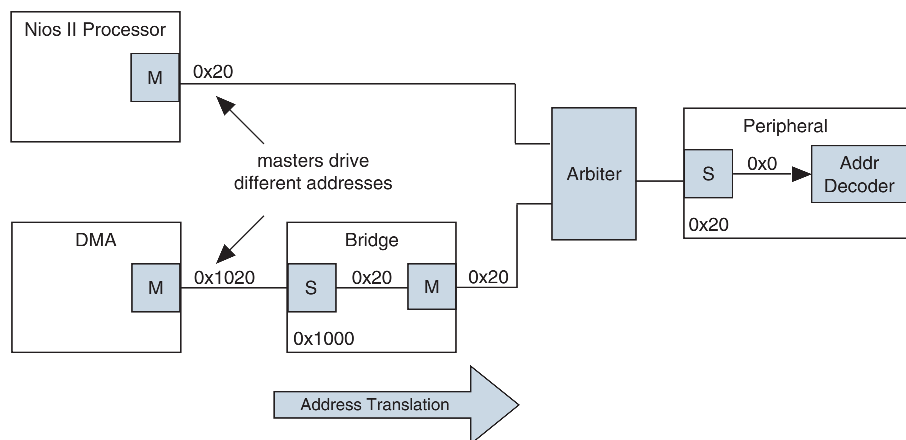


Address Coherency

To avoid unnecessary complications in software, 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, software must compensate for the differences.

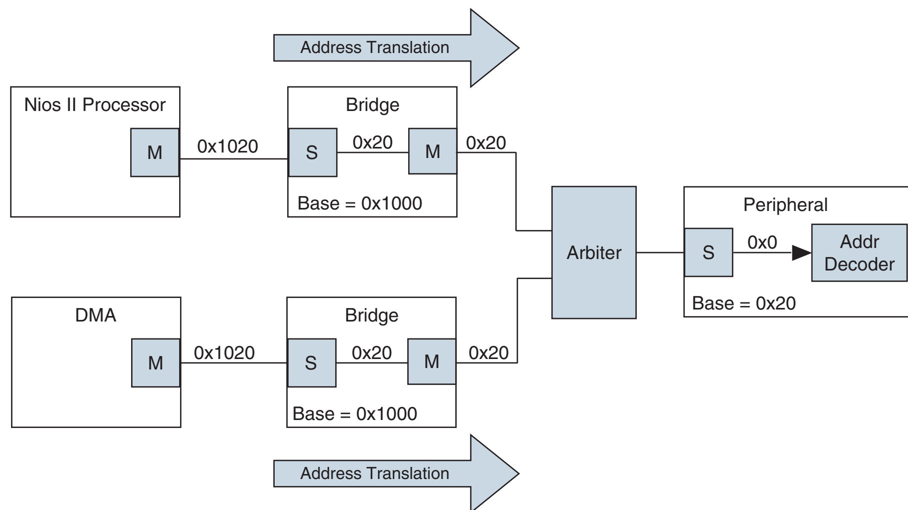
In the following example, a Nios II processor and DMA controller access a slave port located at address 0x20. The processor connects directly to the slave port. The DMA controller connects to a pipeline bridge located at address 0x1000 which then connects to the slave port. Because the DMA controller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

Figure 6–23. Slave at Different Addresses, Complicating the Software



To avoid this issue, you can add an additional bridge to the design and set its base address to 0x1000. You can disable all the pipelining options in this second bridge so that it has a minimal impact on the system timing and resource utilization. Because this second bridge has the same base address as the bridge the DMA controller connects to, both the processor and DMA controller access the slave port using the same address range.

Figure 6–24. Address Translation Corrected Using Bridge



Minimize System Interconnect Logic

In SOPC Builder, you have control over the address space of your system, as well as the connections between master sand slaves. This control allows you to make minor changes to your system in order to increase the overall system performance. The following sections explain design changes you can make to improve the f_{MAX} of your system.

- Use Unique Address Bits
- Create Dedicated Master and Slave Connections
- Remove Unnecessary Connections

Use Unique Address Bits

For every slave in your system, SOPC Builder inserts comparison logic to drive the arbiter to select a slave. This comparison logic determines if the master is performing an access to the slave port by determining if the address presented is in the slave port's address range. This result is ANDed with the master read and write signals to determine if the transfer is destined for the slave port.

The comparison logic can become part of a failing timing path because the result is used to drive the slave port. To reduce this path length, you can move the slave port base address to use unique MSBs for the comparison. Frequently, you can reduce the comparison logic to a single logic element if you avoid using a base address that shares MSBs with other slave ports.

Consider a design with 4 slave ports each having an address range of 0x10 bytes connected to a single master. If you use the **Auto-Assign Base Addresses** option in SOPC Builder, the base addresses for the 4 slaves is set to 0x0, 0x10, 0x20, and 0x30, which corresponds to the following binary numbers: 6b'000000, 6b'010000, 6'b100000, and 6b'110000. The two MSBs must be decoded to determine if a transfer is destined for any of the slave ports.

If the addresses are located at 0x10, 0x20, 0x40, and 0x80, no comparison logic is necessary. These binary locations are: 6’b00010000, 6b’00100000, 6b’01000000, and 6b’10000000. This technique is referred to as *one-hot* encoding because a single asserted address bit replaces comparison logic to determine if a slave transfer is taking place. In this example, the performance gained by moving the addresses would be minimal; however, when you connect many slave ports of different address spans to a master this technique can result in a significant improvement.

Create Dedicated Master and Slave Connections

In some circumstances it is possible to modify a system so that a master port connects to a single slave port. This configuration eliminates address decoding, arbitration, and return data multiplexing, greatly simplifying the system interconnect fabric. Dedicated master-to-save connections attain the same clock frequencies as Avalon-ST connections with the added benefits offered by Avalon-MM.

Typically these one-to-one connections include an Avalon-MM bridge or hardware accelerator. For example, if you insert a pipeline bridge between a slave and all other master ports, the logic between the bridge master and slave port is reduced to wires. Figure 6–21 illustrates this technique. If a hardware accelerator only connects to a dedicated memory, no system interconnect logic is generated between the master and slave pair.

Remove Unnecessary Connections

The number of connections between master and slave ports has a great influence on the f_{MAX} of your system. Every master port that you connect to a slave port increases the multiplexer select 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, only connect masters and slaves when necessary.

In the case of a master port connecting to many slave ports, the multiplexer for the `readdata` signal grows as well. Use bridges to help control this depth of multiplexers, as Figure 6–14 illustrates.

Reducing Logic Utilization

The system interconnect fabric supports the Avalon-MM and Avalon-ST interfaces. Although the system interconnect fabric for Avalon-ST interfaces is lightweight, the same is not always true for the Avalon-MM. This section describes design changes you can make to reduce the logic footprint of the system interconnect fabric.

Minimize Arbitration Logic by Consolidating Components

As the number of components in your design increases, so does the amount logic required to implement the system interconnect fabric. The number of arbitration blocks increases for every slave port that is shared by multiple master ports. The width of the `readdata` multiplexer increases as the number of slave ports supporting `read` transfers increases on a per master port basis. For these reasons you should consider implementing multiple blocks of logic as a single component to reduce the system interconnect fabric logic utilization.

Logic Consolidation Tradeoffs

Consider the following two tradeoffs before making any modifications to your system or components. First, consider the impact on concurrency that consolidating components has. When your system has four master components and four slave components, it can initiate four concurrent accesses. If you consolidate all four slave components into a single component, all four masters must compete for access. Consequently, you should only combine low priority components such as low speed parallel I/O devices where the combination will not impact the performance.

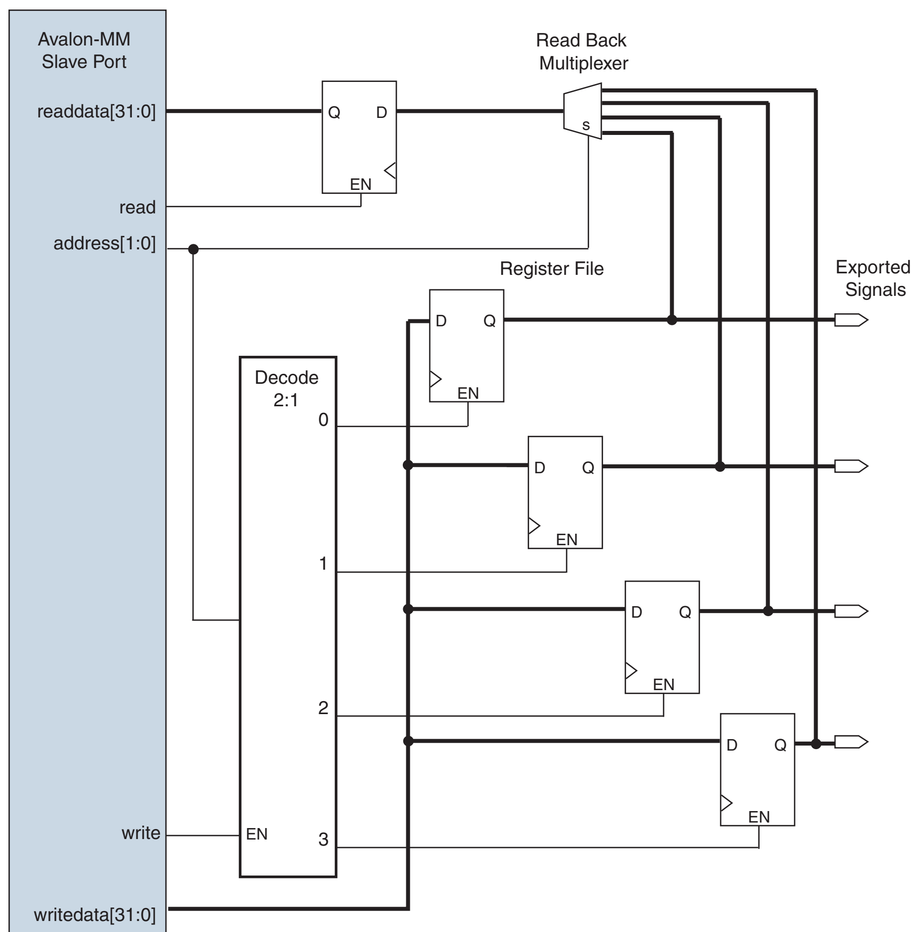
Second, determine whether consolidation introduces new decode and multiplexing logic for the slave port that the system interconnect fabric previously included. If a component contains multiple read and write address locations it already contains the necessary decode and multiplexing logic. When you consolidate components, you typically reuse the decoder and multiplexer blocks already present in one of the original components; however, it is possible that combining components will simply move the decode and multiplexer logic, rather than eliminating duplication.

Combined Component Example

Figure 6–25 illustrates set of four output registers that support software read back. The registers can be implemented using four PIO components in SOPC Builder; however, this example provides a more efficient implementation of the same logic. You can use this example as a template to implement any component that contains multiple registers mapped to memory locations.

Components that implement reads and writes require three main building blocks: an address decoder, a register file, and a read multiplexer. In this example, the read data is a copy of the register file outputs. The read back functionality may seem redundant; however, it is useful for verification.

Figure 6–25. Four PIOs



The decoder enables the appropriate 32-bit PIO register for writes. For reads, the address bits drive the multiplexer selection bits. The `read` signal registers the data from the multiplexer so that the component can achieve a high clock frequency. In the SOPC Builder component editor, this component would be described as having 0 write wait states and 1 read wait state. Alternatively, you could set both the read and write wait states to 0 and specify a read latency of 1 because this component also supports pipelined reads.

Use Bridges to Minimize System Interconnect Fabric Logic

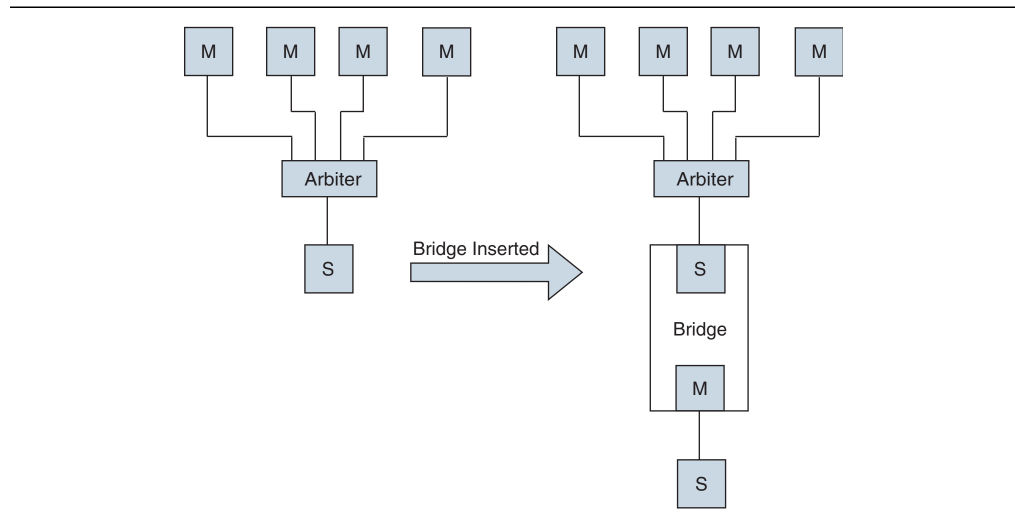
Bridges reduce the system interconnect fabric logic by reducing the amount of arbitration and multiplexer logic that SOPC Builder generates. This reduction occurs because bridges limit the number of concurrent transfers that can occur. The following sections discuss how you can use bridges to minimize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

SOPC Builder Speed Optimizations

The system interconnect fabric SOPC Builder generates supports slave-side arbitration. As a result, SOPC Builder creates arbitration logic for every Avalon-MM slave port that is shared by multiple Avalon-MM master ports. SOPC Builder inserts multiplexer logic between master ports that connect to multiple slave ports if both support read datapaths. The amount of logic generated for the system interconnect fabric grows as the system grows.

Even though the interconnect fabric supports multiple concurrent transfers, the master and slave ports in your system can only handle one transfer at a time. If four masters connect to a single slave, the arbiter grants each access in a round robin sequence. If all four masters connect to an Avalon bridge and the bridge masters the slave port, the arbitration logic moves from the slave port to the bridge.

Figure 6–26. Four Masters to Slave Four Masters to Bridge



In Figure 6–26 a pipeline bridge registers the arbitration logic's output signals, including `address` and `writedata`. A multiplexer in the arbitration block drives these signals. Because a logic element (LE) includes both combinational and register logic, this additional pipelining has little or no effect on the logic footprint. And, the additional pipeline stage reduces the amount of logic between registers, increasing system performance.

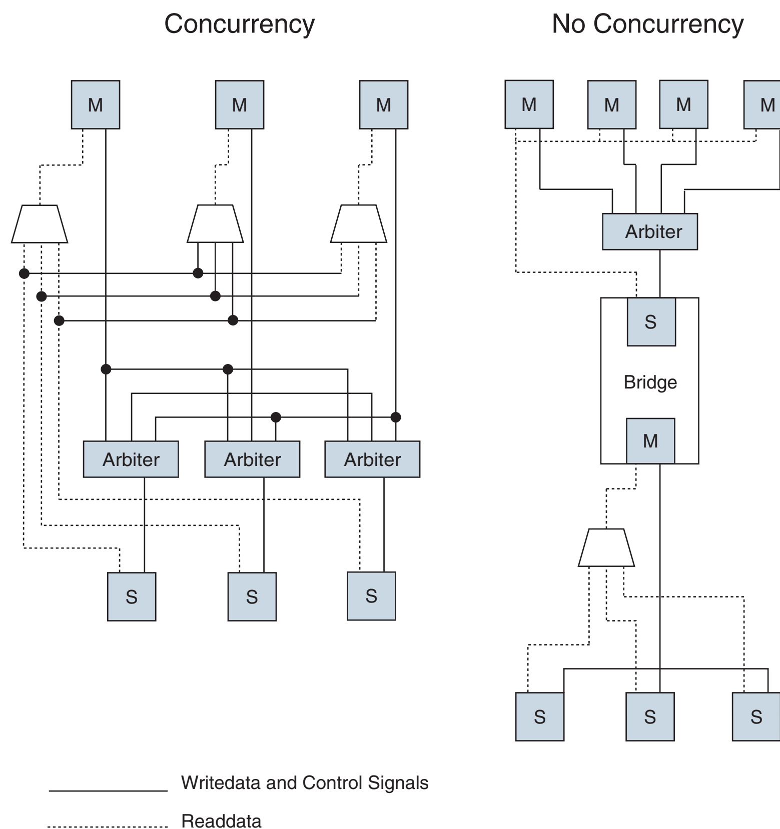
If you can increase the f_{MAX} of your design, you may be able to turn off **Perform register duplication** on the **Physical Synthesis Optimizations** page in the **Settings** dialog box of the Quartus II software. Register duplication duplicates logic in two or more physical locations in the FPGA in an attempt to reduce register-to-register delays. You may also avoid selecting **Speed** for the **Optimization Technique** on the **Analysis & Synthesis Settings** page in the **Settings** dialog box of the Quartus II software. This setting typically results in larger hardware footprint. 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

Most embedded designs contain components that are either incapable of supporting high data throughput or simply do not need to be accessed frequently. These components can contain Avalon-MM master or slave ports. Because the system interconnect fabric supports concurrent accesses, you may wish to limit this 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 ports that are all interconnected, SOPC Builder generates three arbiters and three multiplexers for the read datapath.

Assuming these masters do not require a significant amount of throughput, you can simply connect all three masters to a pipeline bridge. The bridge masters all three slave ports, effectively reducing the system interconnect fabric into a bus structure. SOPC Builder creates one arbitration block between the bridge and the three masters and single read datapath multiplexer between the bridge and three slaves. This architecture prevents concurrency, just as standard bus structures do. Therefore, this method should not be used for high throughput datapaths. Figure 6–27 illustrates the difference in architecture between system with and without the pipeline bridge.

Figure 6–27. Switch Fabric to Bus



Use Bridges to Minimize Adapter Logic

SOPC Builder generates adapter logic for clock crossing and burst support when there is a mismatch between the clock domains or bursting capabilities of the master and slave port pairs. Burst adapters are created 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 Avalon-MM master ports 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 SOPC Builder generates.

Effective Placement of Bridges

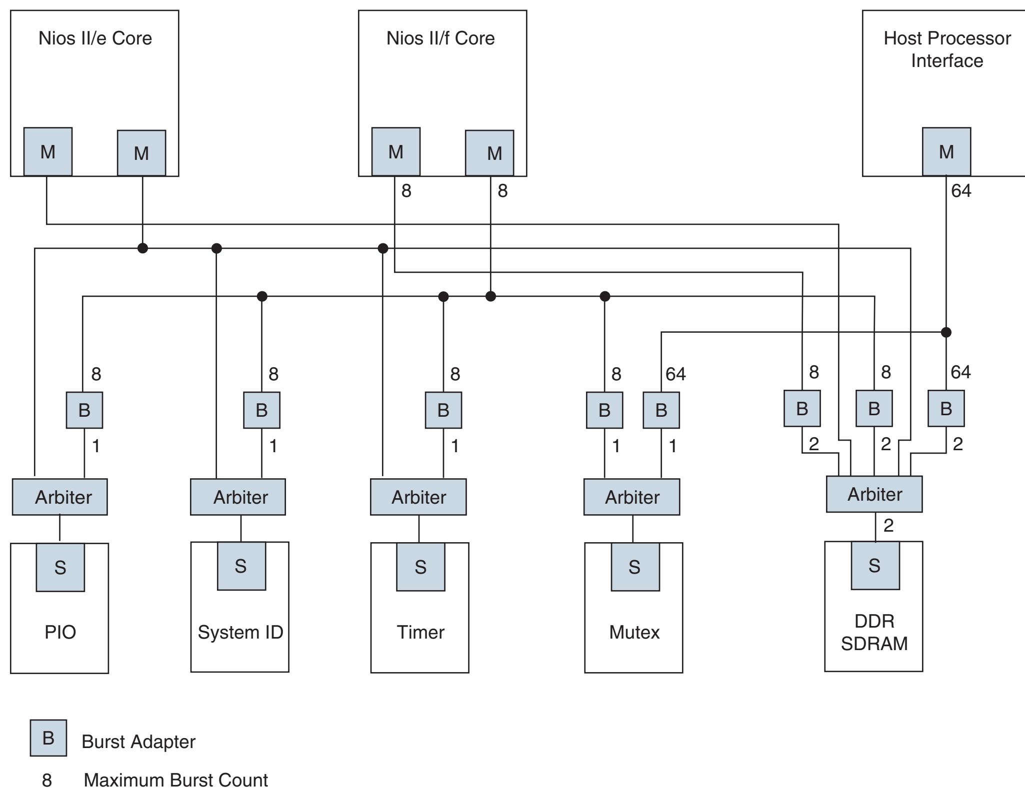
First, analyze each master in your system to determine if the connected slave devices support different bursting capabilities or operate in a different clock domain. The maximum burstcount of a device may be visible as the **burstcount** parameter in the GUI. If it is not, check the width of the `burstcount` signal in the component's HDL file. The maximum burst length is $2^{(width(burstcount -1))}$, so that if the width is 4 bits, the burstcount is 8. If no `burstcount` signal is present, the component does not support bursting or has a burst length of 1.

To determine if a clock crossing adapter is required between the master and slave ports, check the **clock** column beside the master and slave ports in SOPC Builder. If the clock shown is different for the master and slave ports, SOPC Builder inserts a clock crossing adapter between them. To avoid creating multiple adapters, you can place the components containing slave ports 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.

Compact System Example

Figure 6–28 illustrates 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 offloads some processing tasks to the Nios II/f core. 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 DDR SDRAM with an Avalon maximum burst length of two.

Figure 6–28. Mixed Bursting System

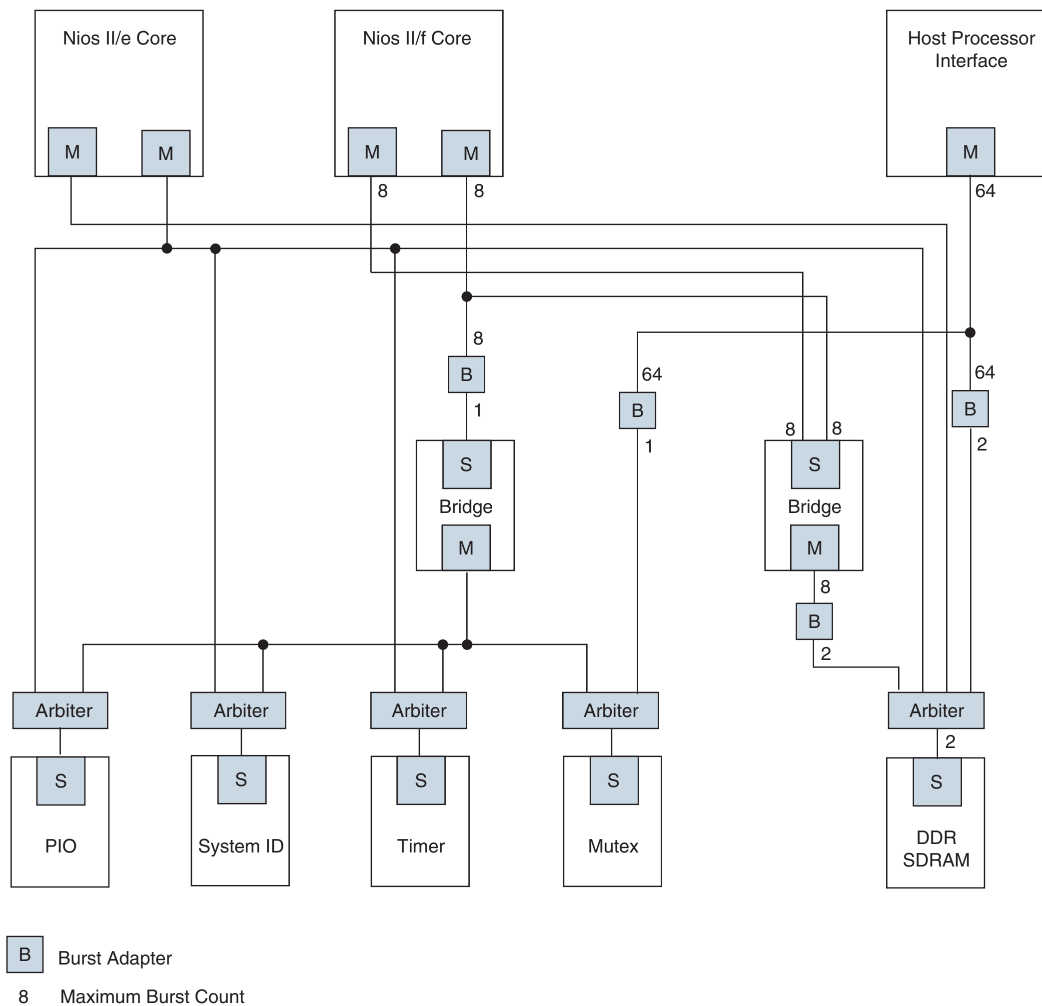


SOPC Builder automatically inserts burst adapters to compensate for burst length mismatches. The adapters reduce bursts to a length of two or single transfers. For the external processor interface connecting to DDR SDRAM, a burst of 64 words is divided into 32 burst transfers, each with a burst length of 2.

At system generation time, SOPC Builder inserts burst adapters based on maximum burstcount values; consequently, the system interconnect fabric includes burst adapters between masters and slave pairs that do not require bursting, if the master is capable of bursts. In Figure 6–28, SOPC Builder 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 only performs single word read and write accesses to these devices.

To reduce the number of adapters, you can add pipeline bridges, as Figure 6–29 illustrates. The pipeline bridge between the Nios II/f core and the peripherals that do not support bursts eliminates three burst adapters from Figure 6–28. 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 6–29. Mixed Bursting System with Bridges



Reducing Power Utilization

Although SOPC Builder does not provide specific features to support low power modes, there are opportunities for you to reduce the power of your system. This section explores the various low power design changes that you can make in order to reduce the power consumption of the system interconnect fabric and your custom components.

Reduce Clock Speeds of Non-Critical Logic

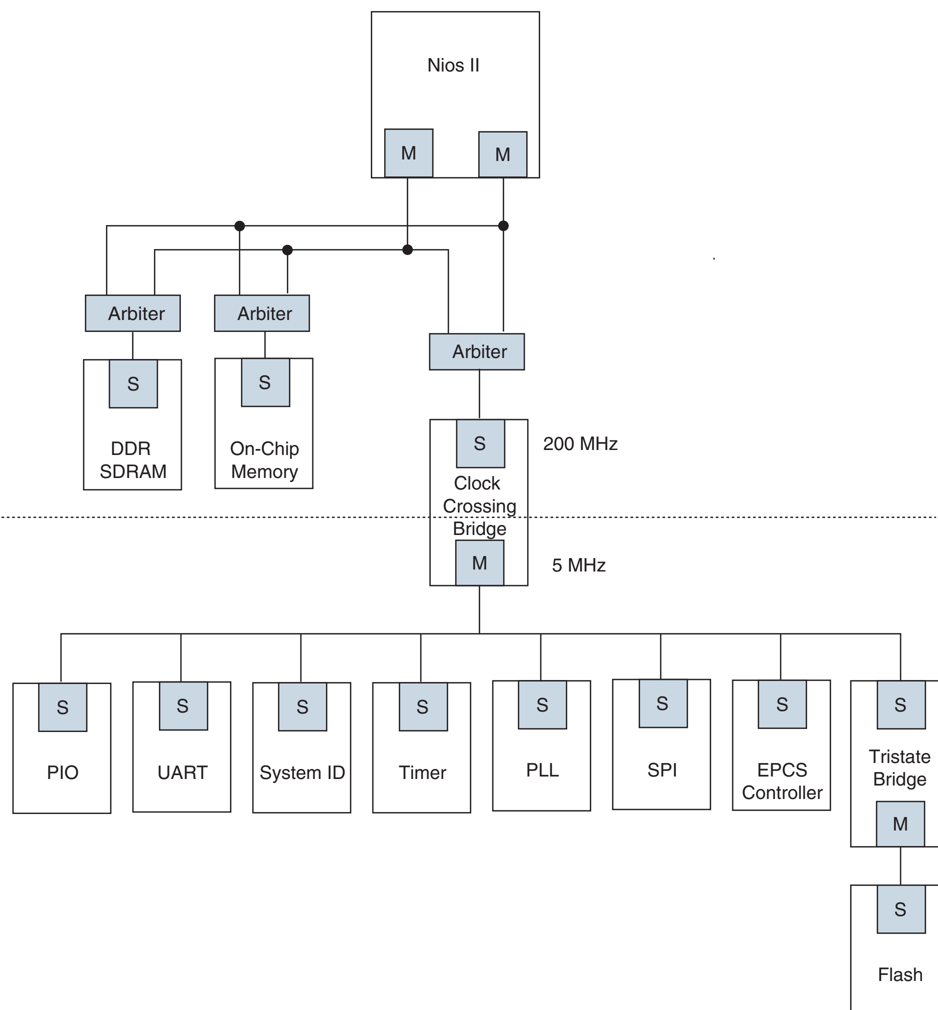
Reducing the clock frequency reduces power consumption. Because SOPC Builder supports clock crossing, you can reduce the clock frequency of the logic that does not require a high frequency clock, allowing you to reduce power consumption. You can use either clock crossing bridges or clock crossing adapters to separate clock domains.

Clock Crossing Bridge

You can use the clock crossing bridge to connect Avalon-MM master ports operating at a higher frequency to slave ports running a a lower frequency. Only low throughput or low priority components should be placed behind a clock crossing bridge that operates at a reduced clock frequency. Examples of typical components that can be effectively placed in a slower clock domain are:

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

Figure 6–30. Low Power Using Bridge



Placing these components behind a clock crossing bridge increases the read latency; however, if the component is not part of a critical section of your design the increased latency is not an issue. 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.

Clock Crossing Adapter

SOPC Builder automatically inserts clock crossing adapters between Avalon-MM master and slave ports that operate at different clock frequencies. The clock crossing adapter uses a handshaking state-machine to transfer the data between clock domains. The HDL code that defines the clock crossing adapters resembles that of other SOPC components. Adapters do not appear in the SOPC Builder **Connection** column because you do not insert them. The differences between clock crossing bridges and clock crossing adapters should help you determine which are appropriate for your design.

Throughput

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

Resource Utilization

The clock crossing bridge requires very 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 any on-chip memory and requires a moderate number of logic resources. The address span, data width, and bursting capabilities of the clock crossing adapter and also determine the resource utilization of the device.

Throughput versus Memory Tradeoffs

The choice between the clock crossing bridge and clock crossing adapter is between throughput and memory utilization. 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 the additional resources required. However, if you can place all your low priority components behind a single clock crossing bridge you reduce power consumption in your design. In contrast, SOPC Builder inserts a clock crossing adapter between each master and slave pair that run at different frequencies if you have not included a clock crossing bridge, increasing the logic utilization in your design.

Minimize 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 three design techniques you can use to reduce the toggle rates of your system:

- Registering Component Boundaries
- Enabling Clocks
- Inserting Bridges

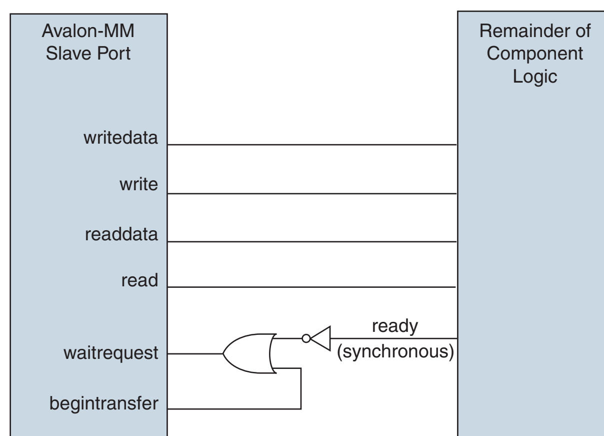
Registering Component Boundaries

The system interconnect fabric is purely combinational when no adapters or bridges are present. When a slave port is not selected by a master, various signals may toggle and propagate into the component. By registering the boundary of your component at the Avalon-MM master or slave interface you can minimize the toggling of the system interconnect fabric and your component. When you register the signals at the port level you must ensure that the component continues to operate within the Avalon-MM specification.

`waitrequest` is usually the most 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 prolong the transfer. A master interface may read the `waitrequest` signal too early and post more reads and writes prematurely.

For slave interfaces, the system interconnect fabric 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.

Figure 6–31. Variable Latency



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

Enabling Clocks

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 Avalon-MM slave components. Even if you add registers to your component boundaries, your interface can still 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 this masking causes a functional failure, it may be possible to use a register stage to hold the combinational logic constant between clock cycles.

Inserting Bridges

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

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

Disable 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 covers two ways to disable a component to reduce power using either software- or hardware-controlled sleep modes.

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 0 or 1. Use the register's output as a clock enable or reset depending on whether the component has non-volatile requirements. The slave port 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 SOPC Builder to provide mutual 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 `waitrequest` to prolong the transfer as it exits sleep mode.

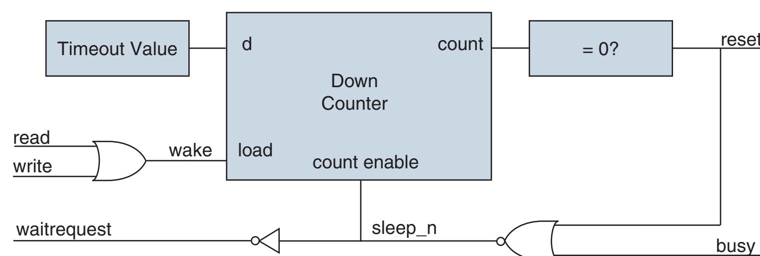
To learn more about the mutex core refer to the *Mutex Core* in volume 5 of the *Quartus II Handbook*.

Hardware Controlled Sleep Mode

You can implement a timer in your component that automatically causes it to enter a sleep mode based upon 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 1. If the counter reaches 0, the hardware enters sleep mode until the next access. Figure 6–32 provides a schematic for this logic. If it takes a long time to restore the component to an active state, use a long timeout value so that the component is not continuously entering and exiting sleep mode.

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

Figure 6–32. Hardware Controlled Sleep Components



For more information on reducing power utilization, refer to *Power Optimization in volume 2 of the Quartus II Handbook*.

Referenced Documents

This chapter references the following documents:

- *Accelerated FIR with Built-in Direct Memory Access Example*
- *Avalon Interfaces Specifications*
- *Avalon Memory-Mapped Bridges* chapter in volume 4 of the *Quartus II Handbook*
- *Creating Multiprocessor Nios II Systems Tutorial*
- *Developing Components for SOPC Builder* in volume 4 of the *Quartus II Handbook*
- *Multiprocessor Coordination Peripherals*
- *Mutex Core* in volume 5 of the *Quartus II Handbook*
- *Nios II Embedded Processor Design Examples*
- *Nios II High-Performance Example With Bridges*
- *Power Optimization* in volume 2 of the *Quartus II Handbook*

 © June 2008 Altera Corporation

Document Revision History

Table 6–1 shows the revision history for this chapter.

Table 6–1. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
June 2008 v1.1	Corrected Table of Contents	—
March 2008 v1.0	Initial release	—