

Overview

This document describes the efficient use of memories in SOPC Builder embedded systems. Efficient memory use increases the performance of FPGA-based embedded systems. Embedded systems use memories for a range of tasks, such as the storage of software code and lookup tables (LUTs) for hardware accelerators.

Your system's memory requirements depend heavily on the nature of the applications which you plan to run on the system. Memory performance and capacity requirements are small for simple, low cost systems. In contrast, memory throughput can be the most critical requirement in a complex, high performance system. The following general types of memories can be used in embedded systems.

Volatile Memory

A primary distinction in memory types is volatility. *Volatile* memories only hold their contents while power is applied to the memory device. As soon as power is removed, the memories lose their contents; consequently, volatile memories are unacceptable if data must be retained when the memory is switched off. Examples of volatile memories include static RAM (SRAM), synchronous static RAM (SSRAM), synchronous dynamic RAM (SDRAM), and FPGA on-chip memory.

Non-volatile Memory

Non-volatile memories retain their contents when power is switched off, making them good choices for storing information that must be retrieved after a system power-cycle. CPU boot-code, persistent application settings, and FPGA configuration data are typically stored in non-volatile memory. Although non-volatile memory has the advantage of retaining its data when power is removed, it is typically much slower to write to than volatile memory, and often has more complex writing and erasing procedures. Non-volatile memory is also usually only guaranteed to be erasable a given number of times, after which it may fail. Examples of non-volatile memories include all types of flash, EPROM, and EEPROM. Most modern embedded systems use some type of flash memory for non-volatile storage.

Many embedded applications require both volatile and non-volatile memories because the two memory types serve unique and exclusive purposes. The following sections discuss the use of specific types of memory in embedded systems.

On-Chip Memory

On-chip memory is the simplest type of memory for use in an FPGA-based embedded system. The memory is implemented in the FPGA itself; consequently, no external connections are necessary on the circuit board. To implement on-chip memory in your design, simply select the **On-Chip Memory** from the **System Contents** tab in SOPC Builder. You can then specify the size, width, and type of on-chip memory, as well as special on-chip memory features such as dual-port access.

- For details about the **On-Chip Memory** SOPC Builder component, refer to the *Building Memory Subsystems Using SOPC Builder* chapter in volume 4 of *Quartus II Handbook*.

Advantages

On-chip memory is the highest throughput, lowest latency memory possible in an FPGA-based embedded system. It typically has a latency of only one clock cycle. Memory transactions can be pipelined, making a throughput of one transaction per clock cycle typical.

Some variations of on-chip memory can be accessed in dual-port mode, with separate ports for read and write transactions. Dual-port mode effectively doubles the potential bandwidth of the memory, allowing the memory to be written over one port, while simultaneously being read over the second port.

Another advantage of on-chip memory is that it requires no additional board space or circuit-board wiring because it is implemented on the FPGA directly. Using on-chip memory can often save development time and cost.

Finally, some variations of on-chip memory can be automatically initialized with custom content during FPGA configuration. This memory is useful for holding small bits of boot code or LUT data which needs to be present at reset.

- For more information about which types of on-chip memory can be initialized upon FPGA configuration, refer to the *Building Memory Subsystems Using SOPC Builder* chapter of the *Quartus II Handbook*.

Disadvantages

While on-chip memory is very fast, it is somewhat limited in capacity. The amount of on-chip memory available on an FPGA depends solely on the particular FPGA device being used, but capacities range from around 15 KBytes in the smallest Cyclone II device to just under 2 MBytes in the largest Stratix III device.

Because most on-chip memory is volatile, it loses its contents when power is disconnected. However, some types of on-chip memory can be initialized automatically when the FPGA is configured, essentially providing a kind of non-volatile function. For details, refer to the embedded memory chapter of the device handbook for the particular FPGA family you are using or Quartus® II Help.

Best Applications

The following sections describe the best uses of on-chip memory.

Cache

Because it is low latency, on-chip memory functions very well as cache memory for microprocessors. The Nios II processor uses on-chip memory for its instruction and data caches. The limited capacity of on-chip memory is usually not an issue for caches because they are typically relatively small.

Tightly Coupled Memory

The low latency access of on-chip memory also makes it suitable for tightly-coupled memories. Tightly coupled memories are memories which are mapped in the normal address space, but have a dedicated interface to the microprocessor, and possess the high-speed, low-latency properties of cache memory.



For more information regarding tightly-coupled memories, refer to the [Using Nios II Tightly Coupled Memory Tutorial](#).

Look Up Tables

For some software programming functions, particularly mathematical functions, it is sometimes fastest to use a LUT to store all the possible outcomes of a function, rather than computing the function in software. On-chip memories work well for this purpose as long as the number of possible outcomes fits reasonably in the capacity of on-chip memory available.

FIFO

Embedded systems often need to regulate the flow of data from one system block to another. FIFOs can buffer data between processing blocks that run most efficiently at different speeds. Depending on the size of the FIFO your application requires, on-chip memory can serve as very fast and convenient FIFO storage.



For more information regarding FIFO buffers, refer to the [On-Chip FIFO Memory Core](#) chapter in volume 5 of the *Quartus II Handbook*.

Poor Applications

On-chip memory is poorly suited for applications which require large memory capacity. Because on-chip memory is relatively limited in capacity, avoid using it to store large amounts of data; however, some tasks can take better advantage of on-chip memory than others. If your application utilizes multiple small blocks of data, and not all of them fit in on-chip memory, you should carefully consider which blocks to implement in on-chip memory. If high system performance is your goal, place the data which is accessed most often in on-chip memory.

On-Chip Memory Types

Depending on the type of FPGA you are using, there are several types of on-chip memory available. For details on the different types of on-chip memory available to you, refer to the device handbook for the particular FPGA family you are using.

Best Practices

To optimize the use of the on-chip memory in your system, follow these guidelines:

- Set the on-chip memory data width to match the data-width of its primary system master. For example, if you are connecting the on-chip memory to the data master of a Nios II processor, you should set the data width of the on-chip memory to 32 bits, the same as the data-width of the Nios II data master. Otherwise, the access latency could be longer than one cycle because the system interconnect fabric performs width translation.

- If more than one master connects to an on-chip memory component, consider enabling the dual-port feature of the on-chip memory. The dual-port feature removes the need for arbitration logic when two masters access the same on-chip memory. In addition, dual-ported memory allows concurrent access from both ports, which can dramatically increase efficiency and performance when the memory is accessed by two or more masters. However, writing to both slave ports of the RAM can result in data corruption if there is not careful coordination between the masters.

To minimize FPGA logic and memory utilization, follow these guidelines:

- Choose the best type of on-chip memory for your application. Some types are larger capacity; others support wider data-widths. The embedded memory section in the device handbook for the appropriate FPGA family provides details on the features of on-chip memories.
- Choose on-chip memory sizes that are a power of 2 bytes. Implementing memories which are not a power of 2 can result in inefficient memory and logic use.

External SRAM

The term *external SRAM* refers to any static RAM (SRAM) device that you connect externally to a FPGA. There are several varieties of external SRAM devices. The choice of external SRAM and its type depends upon the nature of the application. Designing with SRAM memories presents both advantages and disadvantages.

Advantages

External SRAM devices provide larger storage capacities than on-chip memories, and are still quite fast, although not as fast as on-chip memories. Typical external SRAM devices have capacities ranging from around 128 KBytes to 10 MBytes. Specialty SRAM devices can even be found in smaller and larger capacities. SRAMs are typically very low latency and high throughput devices, slower than on-chip memory only because they connect to the FPGA over a shared, bidirectional bus. The SRAM interface is very simple, making connecting to an SRAM from an FPGA a simple design task. You can also share external SRAM buses with other external SRAM devices, or even with external memories of other types, such as flash or SDRAM.



See *Building Memory Subsystems Using SOPC Builder* in volume 4 of the *Quartus II Handbook*, for more information regarding shared external buses.

Disadvantages

The primary disadvantages of external SRAM in an FPGA-based embedded system are cost and board real estate. SRAM devices are more expensive per MByte than other high-capacity memory types such as SDRAM. They also consume more board space per MByte than both SDRAM and FPGA on-chip memory which consumes none.

Best Applications

External SRAM is quite effective as a fast buffer for medium-size blocks of data. You can use external SRAM to buffer data that does not fit in on-chip memory and requires lower latency than SDRAM provides. You can also group multiple SRAM memories to increase capacity.

SRAM is also optimal for accessing random data. Many SRAM devices can access data at non-sequential addresses with the same low-latency as sequential addresses, an area where SDRAM performance suffers. SRAM is the ideal memory type for a large LUT holding the data for color conversion algorithm that is too large to fit in on-chip memory.

External SRAM performs relatively well when used as execution memory for a CPU with no cache. The low latency properties of external SRAM help improve CPU performance if the CPU has no cache to mask the higher latency of other types of memory.

Poor Applications

Poor uses for external SRAM include systems which require large amounts of storage and systems which are cost-sensitive. If your system requires a block of memory larger than 10 MBytes, you may want to consider a different type of memory, such as SDRAM, which is less expensive.

External SRAM Types

There are several types of SRAM devices. The most popular types are listed below.

- Asynchronous SRAM—This is the slowest type of SRAM because it is not dependent on a clock.
- Synchronous SRAM (SSRAM)—Synchronous SRAM operates synchronously to a clock. It is faster than asynchronous SRAM but also more expensive.
- Pseudo-SRAM—Pseudo-SRAM (PSRAM) is a type of dynamic RAM (DRAM) which has an SSRAM interface.
- ZBT SRAM—ZBT (zero bus turnaround) SRAM can switch from read to write transactions with zero turn around cycles, making it a very low-latency. ZBT SRAM typically requires a special controller to take advantage of its low-latency features.

Best Practices

To get the best performance from your external SRAM devices, follow these guidelines:

- Use SRAM interfaces which are the same data width as the data width of the primary system master which accesses the memory.
- If pin utilization or board real estate is a larger concern than the performance of your system, you can use SRAM devices with a smaller data width than the masters that will access them to reduce the pincount of your FPGA and possibly the number of memory devices on the PCB. However, this change results in reduced performance of the SRAM interface.

Flash

Flash memory is a non-volatile memory type used frequently in embedded systems. In FPGA-based embedded systems, flash is always external because FPGAs do not contain flash memory. Because flash memory retains its contents after power is removed, it is commonly used to hold microprocessor boot code as well as any data which needs to be preserved in the case of a power failure. Flash memories are available with either a parallel or a serial interface. The fundamental storage technology for parallel and serial flash devices is the same.

Unlike SRAM, flash cannot be updated with a simple write transaction. Every write to a flash device uses a write command consisting of a fixed sequence of consecutive read and write transactions. Before flash can be written, it must be erased. All flash devices are divided into some number of erase blocks, or sectors, which vary in size, depending on the flash vendor and device size. Entire sections of flash must be erased as a unit; individual words cannot be erased. These requirements sometimes make flash devices difficult to use.

Advantages

The primary advantage of flash memory is that it is non-volatile. Modern embedded systems use flash extensively to store not only boot code and settings, but large blocks of data such as audio or video streams. Many embedded systems use flash memory as a low-power, high-reliability substitute for a hard drive.

Among other non-volatile types of memory, flash is the most popular for four reasons:

- It is durable
- It is erasable
- It permits a large number of erase cycles
- It is low-cost

You can share flash buses with other flash devices, or even with external memories of other types, such as external SRAM or SDRAM.



Refer to *Building Memory Subsystems Using SOPC Builder* in volume 4 of the *Quartus II Handbook* for more information regarding shared external buses.

Disadvantages

A major disadvantage of flash is its write-speed. Because you can only write to flash devices using special commands, multiple bus transactions are required for each flash write. Furthermore, the actual write time, once the write command has been sent, can be several microseconds. Depending on clock speed, the actual write time can be in the hundreds of clock cycles. Because of the sector-erase restriction, if you need to change a data word in the flash, you must complete the following steps:

1. Copy the entire contents of the sector into a temporary buffer
2. Erase the sector
3. Change the single data word in the temporary buffer
4. Write the temporary buffer back to flash.

This procedure contributes to the poor write-speed of flash memory devices. Because of its poor write-speed, flash is typically only used for storing data which must be preserved after power is turned off.

Typical Applications

Flash memory is effective for storing any data that you wish to preserve if power is removed from the system. Common uses of flash include storage of the following items:

- Microprocessor boot code
- Microprocessor application code to be copied to RAM upon system startup
- Persistent system settings, including:
 - Network MAC address
 - Calibration data
 - User preferences
- FPGA configuration images
- Media (audio, video)

Poor Applications

Because of flash memory's slow write speeds, do not use it for anything that does not need to be preserved after power-off. SRAM is a much better alternative if volatile memory is an option. Systems which use flash memory usually also include some SRAM as well.

One particularly poor use of flash is direct execution of microprocessor application code. If any of the code's writeable sections are located in flash, the software simply will not work, because flash cannot be written without using its special write commands. Systems which store application code in flash usually copy the application to SRAM before executing it.

Flash Types

There are several types of flash devices. The most popular types are listed below:

- CFI flash – This is the most common type of flash memory. It has a parallel interface. CFI stands for common flash interface, a standard to which all CFI flash devices adhere. SOPC Builder and the Nios II processor have built-in support for CFI flash.



For more details, refer to the following documentation: *Common Flash Interface Controller Core* in volume 5 of the *Quartus II Handbook* and the *Nios II Flash Programmer User Guide*.

- Serial flash – This flash has a serial interface to preserve device pins and board space. Because many serial flash devices have their own specific interface protocol, it is best to thoroughly read a serial flash device's datasheet before choosing it. Altera EPCS configuration devices are a type of serial flash.

 For more information about EPCS configuration devices, refer to the *Altera Configuration Devices* chapter in volume 2 of Altera's *Configuration Handbook*.


- NAND flash – NAND flash is a newer type of flash memory which has recently begun to gain popularity. NAND flash can achieve very high capacities, up to multiple GBytes per device. The interface to NAND flash is a bit more complicated than that of CFI flash. It requires either a special controller or intelligent low-level driver software. You can use NAND Flash with Altera FPGAs; however, Altera does not provide any built-in support.

SDRAM

SDRAM is another type of volatile memory. It is similar to SRAM, except that it is dynamic and must be refreshed periodically to maintain its content. The dynamic memory cells in SDRAM are much smaller than the static memory cells used in SRAM. This difference in size translates into very high-capacity and low-cost memory devices.

In addition to the refresh requirement, SDRAM has other very specific interface requirements which typically necessitate the use of special controller hardware. Unlike SRAM which has a static set of address lines, SDRAM divides up its memory space into banks, rows, and columns. Switching between banks and rows incurs some overhead, so that efficient use of SDRAM involves the careful ordering of accesses. SDRAM also multiplexes the row and column addresses over the same address lines, which reduces the pin count necessary to implement a given size of SDRAM. Higher speed varieties of SDRAM such as DDR, DDR2, and DDR3 also have strict signal integrity requirements which need to be carefully considered during the design of the PCB.

SDRAM devices are among the least expensive and largest-capacity types of RAM devices available, making them one of the most popular. Most modern embedded systems use SDRAM. A major part of an SDRAM interface is the SDRAM controller. The SDRAM controller manages all the address-multiplexing, refresh and row and bank switching tasks, allowing the rest of the system to access SDRAM without knowledge of its internal architecture.

 For information on the SDRAM controllers available for use in Altera FPGAs, refer to the following documents:

- *DDR and DDR2 SDRAM High-Performance Controller User Guide*,
- *DDR3 SDRAM High-Performance Controller User Guide*
- *AN 398: Using DDR/DDR2 SDRAM with SOPC Builder*.

Advantages

SDRAM's most significant advantages are its capacity and cost. No other type of RAM combines the low-cost and large capacity of SDRAM, which makes it a very popular choice. SDRAM also makes efficient use of pins. Because row and column addresses are multiplexed over the same address pins, fewer pins are required to implement a given capacity of memory. Finally, SDRAM generally consumes less power than an equivalent SRAM device.

In some cases, you can also share SDRAM buses between multiple SDRAM devices, or even with external memories of other types, such as external SRAM or flash.



Refer to *Building Memory Subsystems Using SOPC Builder* in volume 4 of the *Quartus II Handbook* for more information regarding shared external buses.

Disadvantages

Along with the high-capacity and low-cost of SDRAM, comes additional complexity and latency. The complexity of the SDRAM interface requires that you must always use an SDRAM controller to manage SDRAM refresh cycles, address multiplexing, and interface timing. Such a controller consumes FPGA logic elements which would normally be available for other logic.

SDRAM suffers from a significant amount of access latency. Most SDRAM controllers take measures to minimize the amount of latency, but the nature of SDRAM dictates that latency is always greater than that of regular external SRAM or FPGA on-chip memory. However, while first-access latency is high, SDRAM throughput can actually be quite high once the initial access latency is overcome because consecutive accesses can be pipelined. Some types of SDRAM can achieve higher clock frequencies than SRAM, further improving throughput. The SDRAM interface specification also employs a burst feature to help improve overall throughput.

Best Applications

SDRAM is generally a good choice in the following circumstances:

- Storing large blocks of data—SDRAM's large capacity makes it the best choice for buffering any large blocks of data such as network packets, video frame buffers, and audio data.
- Executing microprocessor code—SDRAM is commonly used to store instructions and data for microprocessor software, particularly when the program being executed is large. Instruction and data caches improve performance for large programs. Depending on the system topography and the SDRAM controller used, the sequential reads typical of cache line fills can potentially take advantage of SDRAM's pipeline and burst capabilities.

Poor Applications

SDRAM may not be the best choice in the following situations:

- Whenever low-latency memory access is required—Although high throughput is possible using SDRAM, its first-access latency is quite high. If low latency access to a particular block of data is a requirement of your application, SDRAM is probably not a good candidate for holding that block of data.

- Small blocks of data—When only a small amount of storage is needed, SDRAM may be unnecessary. An on-chip memory may be able to meet your memory requirements without adding another memory device to the PCB.
- Small, simple embedded systems—If your system uses a small FPGA in which logic resources are scarce and your application does not require the capacity that SDRAM provides, you may prefer to use a small external SRAM or on-chip memory rather than devoting FPGA logic elements to an SDRAM controller.

SDRAM Types

There are a several types of SDRAM devices. The most common types are listed below:

- SDR SDRAM—Single data rate (SDR) SDRAM is the original type of SDRAM. It is either referred to as just SDRAM or SDR SDRAM to distinguish it from newer, double data rate (DDR) types. The name *single data rate* refers to the fact that a maximum of a single word of data can be transferred per clock cycle. SDR SDRAM is still in wide use, although newer types of DDR SDRAM are becoming more common.
- DDR SDRAM—Double data rate (DDR) SDRAM is a newer type of SDRAM that supports higher data throughput by transferring a data word on both the rising and falling edge of the clock. DDR SDRAM uses 2.5 V SSTL signaling. The use of DDR SDRAM requires a custom memory controller.
- DDR2 SDRAM—DDR2 SDRAM is a newer variation of standard DDR SDRAM memory which builds on the success of DDR by implementing slightly improved interface requirements such as lower power 1.8 V SSTL signaling and on-chip signal termination.
- DDR3 SDRAM—DDR3 is another variant of DDR SDRAM which again improves the potential bandwidth of the memory by improving signal integrity and increasing clock frequencies.

SDRAM Controller Types Available From Altera

Table 7-1 lists the SDRAM controllers that Altera provides. They are available without licenses.

Table 7-1. Memory Controller Available from Altera (Part 1 of 2)

Controller Name	Description
SDR SDRAM Controller	This is the only SDR SDRAM controller Altera offers. It is a simple, easy-to-use controller that works with most available SDR SDRAM devices. For more information refer to <i>SDRAM Controller Core</i> chapter in volume 5 of the <i>Quartus II Handbook</i> .
DDR/DDR2 Controller Megacore Function	This controller is a legacy component which is maintained for existing designs only. Altera does not recommend it for new designs.

Table 7-1. Memory Controller Available from Altera (Part 2 of 2)

Controller Name	Description
High Performance DDR/DDR2 Controller	<p>This is the DDR/DDR2 controller that Altera recommends for new designs. It supports two primary clocking modes, full-rate and half-rate.</p> <ul style="list-style-type: none">■ Full-rate mode presents data to the SOPC Builder system at twice the width of the actual DDR SDRAM device at the full SDRAM clock rate.■ Half-rate mode presents data to the SOPC Builder system at four times the native SDRAM device data width at half the SDRAM clock rate. <p>For more information about this controller, refer to the <i>DDR and DDR2 SDRAM High-Performance Controller User Guide</i>.</p>
High Performance DDR3 Controller	<p>This is the DDR3 controller that Altera recommends for new designs. It is similar to the high performance DDR/DDR2 controller. It also supports full- and half-rate clocking modes.</p> <p>For more information about this controller, refer to the <i>DDR3 SDRAM High-Performance Controller User Guide</i>.</p>

Best Practices

When using the high performance DDR or DDR2 SDRAM controller, it is important to determine whether full-rate or half-rate clock mode is optimal for your application.

Half-Rate Mode

Half-rate mode is optimal in cases where you require the highest possible SDRAM clock frequency, or when the complexity of your system logic means that you are not able to achieve the clock frequency you need for the DDR SDRAM. In half-rate mode, the internal Avalon interface to the SDRAM controller is half of the external SDRAM frequency.

In half-rate mode, the local data width (the data width inside the SOPC Builder system) of the SDRAM controller is four times the data width of the physical DDR SDRAM device. For example, if your SDRAM device is 8 bits wide, the internal Avalon data port of the SDRAM controller is 32 bits. This design choice facilitates bursts of four accesses to the SDRAM device.

Full-Rate Mode

In full-rate mode, the internal Avalon interface to the SDRAM controller runs at the full external DDR SDRAM clock frequency. Use full-rate mode if your system logic is simple enough that it can easily achieve DDR SDRAM clock frequencies, or when running the system logic at half the clock rate of the SDRAM interface is too slow for your requirements.

When using full-rate mode, the local data width of the SDRAM controller is two times the data width of the physical DDR SDRAM itself. For example, if your SDRAM device is 16-bits wide, the internal Avalon data port of the SDRAM controller in full-rate mode is 32 bits. Again, this choice facilitate bursts to the SDRAM device

Sequential Access

SDRAM performance benefits from sequential accesses. When access is sequential, data is written or read from consecutive addresses and it may be possible to increase throughput by using bursting. In addition, the SDRAM controller can optimize the accesses to reduce row and bank switching. Each row or bank change incurs a delay, so that reducing switching increases throughput.

Bursting

SDRAM devices employ bursting to improve throughput. Bursts group a number of transactions to sequential addresses, allowing data to be transferred back-to-back without incurring the overhead of requests for individual transactions. If you are using the high performance DDR/DDR2 SDRAM controller, you may be able to take advantage of bursting in the system interconnect fabric as well. Bursting is only useful if both the master and slave involved in the transaction are burst-enabled. Refer to the documentation for the master in question to see if bursting is supported.

Selecting the burst size for the high performance DDR/DDR2 SDRAM controller depends on the mode in which you use the controller. In half-rate mode, the Avalon-MM data port is four times the width of the actual SDRAM device; consequently, four transactions are initiated to the SDRAM device for each single transfer in the system interconnect fabric. A burst size of four is used for those four transactions to SDRAM. This is the maximum size burst supported by the high performance DDR/DDR2 SDRAM controller. Consequently, using bursts for the high performance DDR/DDR2 SDRAM controller in half-rate mode does not increase performance because the system interconnect fabric is already using its maximum supported burst-size to carry out each single transaction.

However, in full-rate mode, you can use a burst size of two with the high performance DDR/DDR2 SDRAM controller. In full-rate mode, each Avalon transaction results in two SDRAM device transactions, so two Avalon transactions can be combined in a burst before the maximum supported SDRAM controller burst size of four is reached.

SDRAM Minimum Frequency

Many SDRAM devices, particularly DDR, DDR2, and DDR3 devices have minimum clock frequency requirements. The minimum clock rate depends on the particular SDRAM device. Refer to the datasheet of the SDRAM device you are using to find the device's minimum clock frequency.

SDRAM Device Speed

SDRAM devices, both SDR and DDR, come in several speed grades. When using SDRAM with FPGAs, the operating frequency of the FPGA system is usually lower than the maximum capability of the SDRAM device. Therefore, it is typically not worth the extra cost to use fast speed-grade SDRAM devices. Before committing to a specific SDRAM device, consider both the expected SDRAM frequency of your system, and the maximum and minimum operating frequency of the particular SDRAM device.

Memory Optimization

This section presents tips and tricks that can be helpful when implementing any type of memory in your SOPC Builder system. These techniques can help improve system performance and efficiency.

Isolate Critical Memory Connections

For many systems, particularly complex ones, isolating performance-critical memory connections is beneficial. To achieve the maximum throughput potential from memory, connect it to the fewest number of masters possible and share those masters with the fewest number of slaves possible. Minimizing connections reduces the size of the data multiplexers required, increasing potential clock speed and also reduces the amount of arbitration necessary to access the memory.



You can use bridges to isolate memory connections. For more information on efficient system topology refer to the following documents:

- *Avalon Memory-Mapped Bridges* chapter in volume 4 of the *Quartus II Handbook*.
- *Avalon Memory-Mapped Design Optimizations* chapter of the *Embedded Design Handbook*.

Match Master and Slave Data Width

Matching the data widths of master and slave pairs in SOPC Builder is advantageous. Whenever a master port is connected to a slave of a different data width, SOPC Builder inserts adapter logic to translate between them. This logic can add additional latency to each transaction, reducing throughput. Whenever possible, try to keep the data width consistent for performance-critical master and slave connections. In cases where masters are connected to multiple slaves, and slaves are connected to multiple masters, it may be impossible to make all the master and slave connections the same data width. In these cases, you should concentrate on the master-to-slave connections which have the most impact on system performance.

For instance, if Nios II CPU performance is critical to your overall system performance, and the CPU is configured to run all its software from an SDRAM device, you should use a 32-bit SDRAM device because that is the native data width of the Nios II processor, and it delivers the best performance. Using a narrower or wider SDRAM device can negatively impact CPU performance because of greater latency and lower throughput. However, if you are using a 64-bit DMA to move data to and from SDRAM, the overall system performance may be more dependent on DMA performance. In these cases, it may be advantageous to implement a 64-bit SDRAM interface.

Use Separate Memories to Exploit Concurrency

Any time multiple masters in your system access the same memory, each master is only granted access some fraction of the time. Shared access may hurt system throughput if a master is starved for data.

If you create separate memory interfaces for each master, they can access memory concurrently at full speed, removing the memory bandwidth bottleneck. Separate interfaces are quite useful in systems which employ a DMA, or in multiprocessor systems where the potential for parallelism is significant.

In SOPC Builder, it is easy to create separate memory interfaces. Simply instantiate multiple on-chip memory components instead of one. You can also use this technique with external memory devices such as external SRAM and SDRAM by adding more, possibly smaller, memory devices to the board and connecting them to separate interfaces in SOPC Builder. Adding more memory devices presents tradeoffs between board real estate, FPGA pins, and FPGA logic resources, but can certainly improve system throughput. Your system topology should reflect your system requirements.



For more information regarding topology tradeoffs refer to *Avalon Memory-Mapped Design Optimizations* chapter of the *Embedded Design Handbook*.

Understand the Nios II Instruction Master Address Space

This Nios II CPU instruction master cannot address more than a 256 MByte span of memory; consequently, providing more than 256 MBytes to run Nios II software wastes memory resources. This restriction does not apply to the Nios II data master that can address up to 2 GBytes.

Test Memory

You should rigorously test the memory in your system to ensure that it is physically connected and setup properly before relying on it in an actual application. The Nios II Development Kit ships with a memory test example which is a good starting point for building a thorough memory test for your system.

Case Study

The section describes the optimization of memory partitioning in a video processing application to illustrate the concepts discussed earlier in this document.

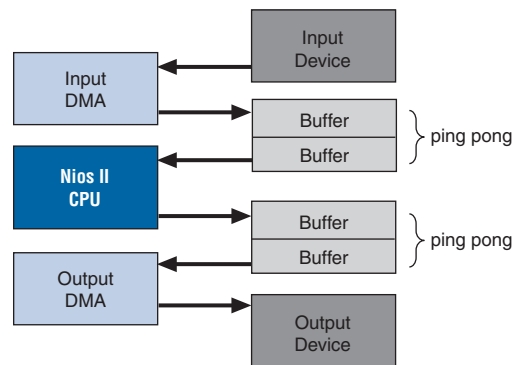
Application Description

This video processing application employs an algorithm that operates on a full frame of video data, line by line. Other details of the algorithm do not impact design of the memory subsystem. The data flow includes the following steps:

1. A dedicated DMA engine copies the input data from the video source into a buffer.
2. A Nios II CPU operates on that buffer, performing the video processing algorithm and writing the result to another buffer.
3. A second dedicated DMA engine copies the output from the CPU result buffer to the video output device.
4. The two DMAs provide an element of concurrency by copying input data to the next input buffer, and copying output data from the previous output buffer at the same time the CPU is processing the current buffer, a technique commonly called ping-ponging.

Figure 7-1 shows the basic architecture of the system.

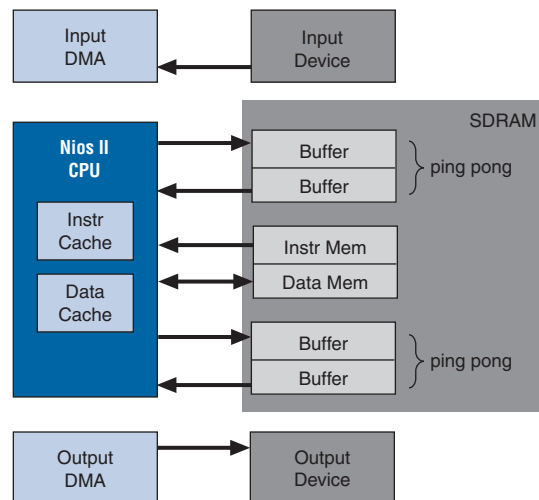
Figure 7-1. Sample Application Architecture



Initial Memory Partitioning

As a starting point, the application uses SDRAM for all of its storage and buffering, a commonly used memory architecture. The input DMA copies data from the video source to an input buffer in SDRAM. The CPU performs its processing by reading from that SDRAM input buffer, writing its result to an output buffer, also located in SDRAM. In addition, the CPU uses SDRAM for both its instruction and data memory. (Refer to [Figure 7-2.](#))

Figure 7-2. All Memory Implemented in SDRAM



Functionally, there is nothing wrong with this implementation. It is a frequently used, traditional type of embedded system architecture. It is also relatively inexpensive, because it uses only one external memory device; however, it is somewhat inefficient, particularly regarding its use of SDRAM. As [Figure 7-2](#) illustrates, there are 6 different channels of data being accessed in the SDRAM.

1. CPU instruction
2. CPU data
3. Input data from DMA
4. Input data to CPU

5. Output data from CPU
6. Output data to DMA

With this many channels moving in and out of SDRAM simultaneously, especially at the high data-rates required by video, the SDRAM bandwidth is easily the most significant performance bottleneck in the design.

Optimized Memory Partitioning

This design can be optimized to operate more efficiently. These optimizations are described in the following sections.

Add An External SRAM for input buffers

The first optimization to improve efficiency is to move the input buffering from the SDRAM to an external SRAM device. This technique creates performance gains for three reasons:

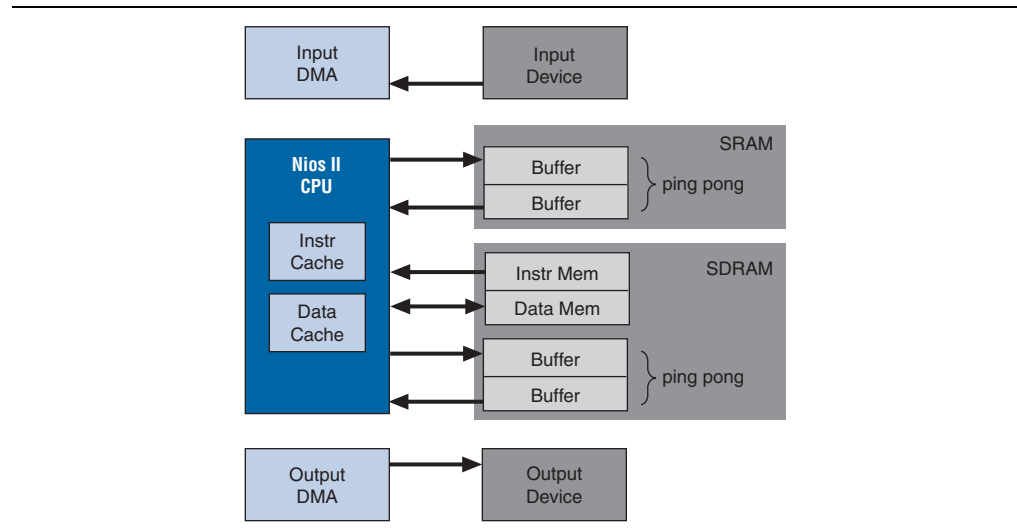
- First, the input side of the application achieves higher throughput because it now uses its own dedicated external SRAM to bring in video data.
- Second, two of the high-bandwidth channels from the SDRAM are eliminated, allowing the remaining SDRAM channels to achieve higher throughput.
- Third, because eliminating two channels reduces the number of accesses to the SDRAM memory, there are fewer row changes in the SDRAM, leading to higher throughput.

The redesigned system processes data faster, at the expense of more complexity and higher cost. [Figure 7-3](#) illustrates the redesigned system.



If the video frames are small enough to fit in FPGA on-chip memory, you can use on-chip memory for the input buffers, saving the expense and complexity of adding an external SRAM device.

Figure 7-3. Input Channel Moved to External SSRAM



Notice that there are still four channels connected to SDRAM:

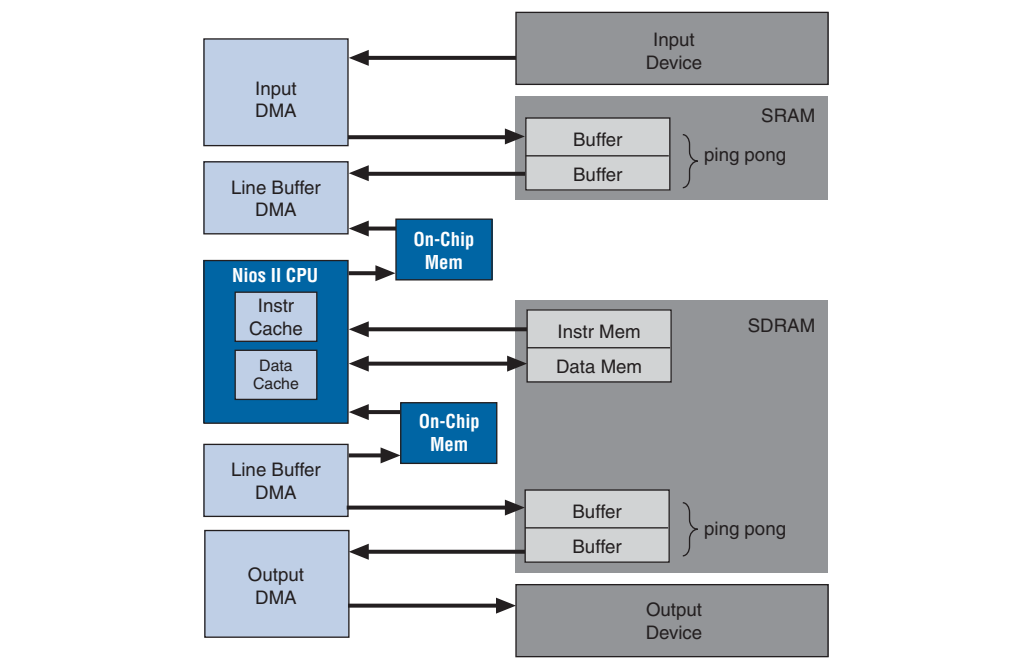
1. CPU instruction
2. CPU data
3. Output data from CPU
4. Output data to DMA

While we could probably achieve some additional performance benefit by adding a second external SRAM for the output channel, the benefit would not likely be significant enough to outweigh the added cost and complexity. The reason is that only two of the four remaining channels require significant bandwidth from the SDRAM, the two video output channels. Assuming our CPU contains both instruction and data caches, the SDRAM bandwidth required by the CPU is likely to be relatively small. Therefore, sharing the SDRAM for CPU instruction and data, and the video output channel is probably acceptable. If necessary, increasing the CPU cache sizes can further reduce the CPU's reliance on SDRAM bandwidth.

Add On-Chip Memory for Video Line Buffers

The final optimization is to add small on-chip memory buffers for input and output video lines. Because the processing algorithm operates on the video input one line at a time, buffering entire lines of input data in an on-chip memory improves performance. It enables the CPU to read all its input data from on-chip RAM—the fastest, lowest latency type of memory available.

The DMA fills these buffers ahead of the CPU in a ping-pong scheme, in a manner analogous to the input frame buffers used for the external SRAM. The same on-chip memory line buffering scheme is used for CPU output. The CPU writes its output data to an on-chip memory line buffer, which is copied to the output frame buffer by a DMA once both the input and output ping-pong buffers flip, and the CPU begins processing the next line. [Figure 7-4](#) illustrates this memory architecture.

Figure 7-4. On-Chip Memories Added As Line Buffers

Referenced Documents

This chapter references the following documents:

- *Altera Configuration Devices* chapter in volume 2 of the *Configuration Handbook*
- *AN 398: Using DDR/DDR2 SDRAM with SOPC Builder*
- *Avalon Memory-Mapped Bridges* in volume 4 of the *Quartus II Handbook*
- *Avalon Memory-Mapped Design Optimizations* chapter of the *Embedded Design Handbook*
- *Building Memory Subsystems Using SOPC Builder* in volume 4 of *Quartus II Handbook*
- *Common Flash Interface Controller Core* in volume 5 of the *Quartus II Handbook*
- *DDR and DDR2 SDRAM High-Performance Controller User Guide*
- *DDR3 SDRAM High-Performance Controller User Guide*
- *Nios II Flash Programmer User Guide*
- *On-Chip FIFO Memory Core* in volume 5 of the *Quartus II Handbook*
- *SDRAM Controller Core* in volume 5 of the *Quartus II Handbook*
- *Using Nios II Tightly Coupled Memory Tutorial*

Document Revision History

Table 7-1 shows the revision history for this chapter.

Table 7-2. 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.	—

