Using DCFIFO for Data Transfer between Asynchronous Clock Domains

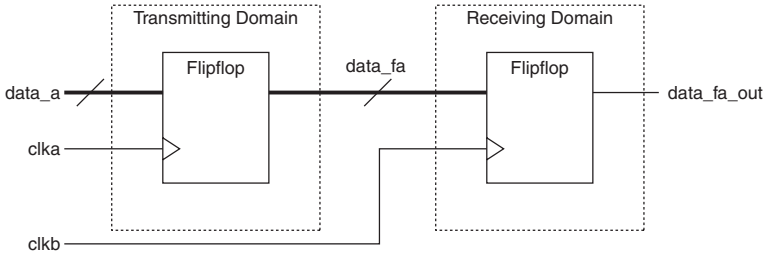
December 2007, version 1.0 Application Note 473

Introduction

In the design world, there are very few designs with a single clock domain. With increasingly complex designs, most applications have interfaces that involve asynchronous clock domains. Asynchronous clock domain crossing refers to the interfaces between clock domains with variable phase and no time relationship. This includes clock domains with different or similar frequencies but with unknown or varying phase relationships.

Figure 1 shows an example of asynchronous clock domain crossing. The transmitting domain and receiving domain are feed by `clka` and `clkb`, respectively. The clocks have variable phase and no time relationship. You can change the `data_fa` signal that crosses over from the transmitting domain to the receiving domain at any time, irrespective to `clkb`. This is considered asynchronous to the receiving domain.

Figure 1. Asynchronous Clock Domain Crossing



Verification for asynchronous clock domain paths is challenging. Designing with asynchronous clock domain crossing requires special handling to ensure the data is properly transferred from one domain to another. Because the time relationship between the two clocks is unknown for asynchronous clock domain crossing, you cannot ensure that you will meet the setup and hold time requirements of flipflops. When setup or hold times are violated, flipflop outputs become metastable. Metastable output causes unstable and illegal signal values that can propagate through the rest of the design at the receiving domain.

For more information about metastability, refer to *Application Note 42: Metastability in Altera Devices*.

There are two techniques commonly used when transferring multiple data bits across asynchronous clock domain paths. The first technique is based on a type of "handshake check" protocol. The second technique uses the dual-clock FIFO (DCFIFO) as the interface of the asynchronous clock domain crossing.

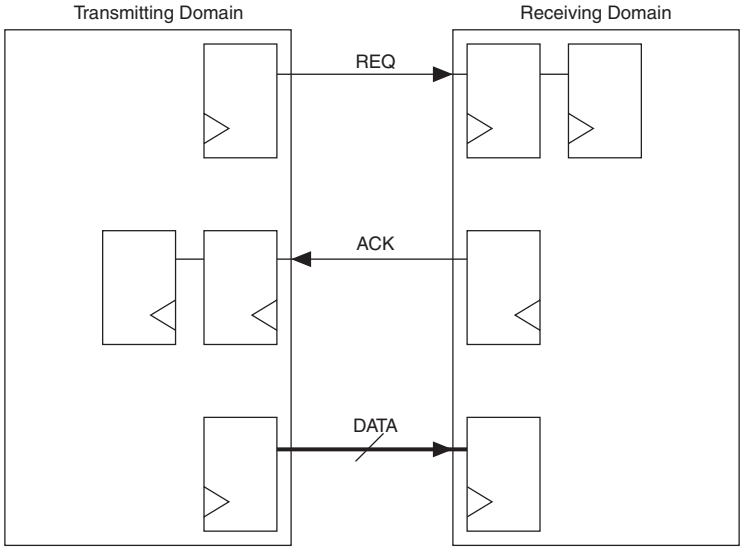
This application note covers the following topics:

- Handshake protocol and DCFIFO methods used for data transferring in asynchronous clock domain crossing.
- Using DCFIFO and a comparison with handshake protocol methodology.
- Design example for data transfer using DCFIFO that covers specification and implementation of the design example. Simulation results are discussed and detailed explanations are provided.

Data Transfer Using Handshake Protocol

The high-level block diagram in Figure 2 shows how the handshake protocol works. To prevent metastability issues, only the control signals (that is, REQ and ACK) require synchronization with cascaded flipflops. Based on the device hardware mean time between failures (MTBF) requirement, the control signals might need more synchronization stages to handle the metastable signal for the design to function properly. The ACK and REQ signals ensure that the data is always stable when the receiving domain samples it.

Figure 2. Handshake Protocol for Data Transfer between Asynchronous Clock Domains



You should implement state machines in both the transmitting domain and receiving domain to handle handshake protocol communication. The waveforms in Figure 3 show how the handshake protocol works.

The REQ signal is asserted when the data is ready and stable for transmitting. During the transmitting process, data must hold stable and can change only after both the REQ and ACK signals are de-asserted. The ACK signal is asserted to notify that the data is received successfully. Both REQ and ACK signals are de-asserted before the next data transfer. The same process applies for the next data transfer as well.

Figure 3. Handshake Protocol Timing Diagram

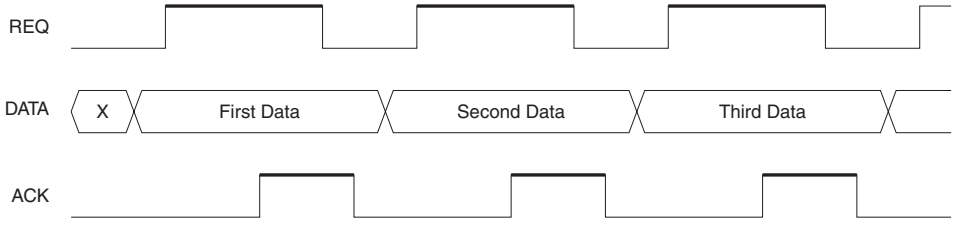


Figure 4 shows the chronological events for the data transferring process using a handshake protocol. A handshake protocol comprises the following steps:

1. Until the transfer is complete, the transmitting domain ensures the DATA signals are stable before a request signal (REQ) is asserted. The transfer is complete when the ACK signal is received and de-asserted.

2. The transmitting domain sends a request signal (REQ) to the receiving domain to inform that the data is ready to be read.

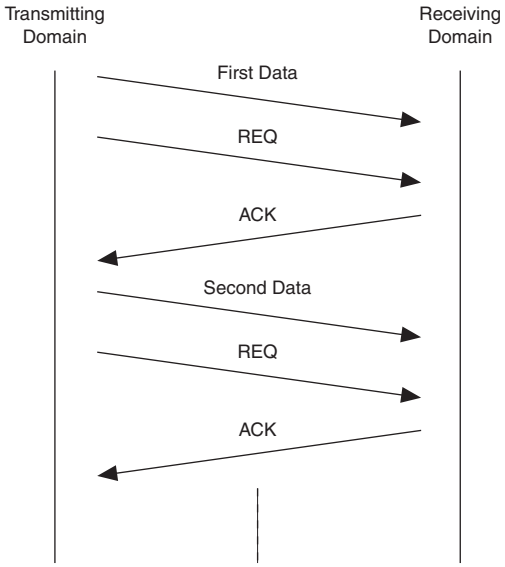
3. The REQ signal is synchronized in the receiving domain through a cascaded synchronizer. After receiving the signal, the receiving domain samples the data in the register where it has been placed by the transmitting domain.

4. The receiving domain sends an acknowledgement signal (ACK) to the transmitting domain to inform it that it has read the data.

5. The ACK signal is synchronized in the transmitting domain through a cascaded synchronizer. After receiving the signal, the transmitting domain de-activates the REQ signal.

6. When the receiving domain senses that the REQ signal has been de-activated in the transmitting domain, it de-activates the ACK signal. Only then is the cycle of the first data transfer considered complete. Subsequent data transfers repeat the same steps.

Figure 4. Chronological Events for Data Transfer Using Handshake Protocol



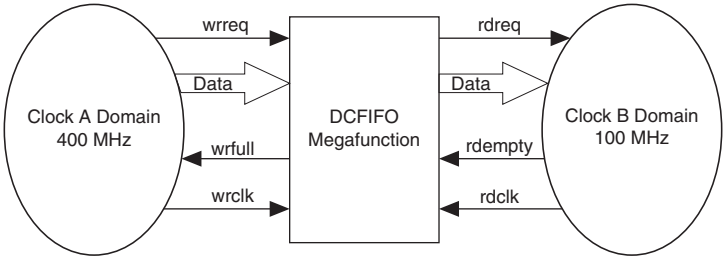
Data Transfer Using DCFIFO

To make sure the data is properly transferred, you can use a DCFIFO megafunction to interface between asynchronous clock domains A and B. Altera® provides the DCFIFO megafunction when you use the MegaWizard® Plug-In Manager in the Quartus® II software. You can configure the DCFIFO megafunction according to your design requirements and instantiate the megafunction in your design to perform the data transfer between the asynchronous clock domains. The DCFIFO megafunction saves you valuable design time and effort by your not having to hand code your own DCFIFO.

For more information about the DCFIFO megafunction, refer to the *First-In-First-Out Megafunction User Guide (FIFO)*.

The high-level block diagram in Figure 5 shows how the data is transferred from clock A domain to clock B domain through the DCFIFO megafunction. The DCFIFO megafunction interfaces between asynchronous clock domains A and B, which are running at different speeds with no time relationship.

Figure 5. Using the DCFIFO Megafunction for Asynchronous Clock Domain Crossing



The DCFIFO megafunction has some control and status signals. Among the signals, write request (`wrreq`), read request (`rdreq`), write full (`wrfull`), and read empty (`rdempty`), are commonly used for data transfer in asynchronous clock domain crossing. The DCFIFO megafunction performs two main operations together with the signals:

- Writing the data into the FIFO (write operation)
- Reading the data from the FIFO (read operation)

For write operations, assert the `wrreq` control signal only if the FIFO is not full (`wrfull` is de-asserted). Continue the write operation into the DCFIFO until it finishes the data transfer from the transmitting domain clock A. When the DCFIFO is full, you must wait until `wrfull` is de-asserted before you can start writing data into the DCFIFO.

For read operations, assert the `rdreq` control signal only if the FIFO is not empty (`rdempty` is de-asserted). Continue to read from the FIFO until the `rdempty` control signal is asserted. You can use a counter in the receiving domain to count the number of words that have read from the DCFIFO. Depending on the application, you may want to place another counter to capture the packet or block boundary (for example, one package consists of eight words).

You must consider the precautions discussed in this section when you design your write control logic and read control logic for write and read operations. "Design Example: Data Transfer Using the DCFIFO Megafunction," on page 8 provides an example of how to make use of the DCFIFO megafunction with status signals for data transfer in asynchronous clock domains.

Benefits of the DCFIFO Megafunction

Handshake protocol and DCFIFO are the two most common methods for data transfer between asynchronous clock domains. However, using DCFIFO (specifically, the Altera DCFIFO megafunction) has many benefits over the handshake protocol implementation.

You can obtain better performance for your design with the DCFIFO. As compared to the DCFIFO, handshake protocol handling takes a few more clock cycles to transfer a single data signal. The handshake protocol must wait for an ACK signal for each data transfer. However, for the DCFIFO, write and read operations can occur at the same time continuously without waiting for ACK signals. If your design requires high-speed data transfer, using DCFIFO is the better choice.

Using the DCFIFO megafunction saves you valuable design time and makes the debugging process easier and more efficient by not requiring you to code for the handshake protocol or design your own DCFIFO. Also, the DCFIFO megafunction reduces failure rate because the metastability issues have already been handled by the megafunction (for example, you do not have to decide how many cascaded synchronizers are required to handle metastability issues). With these advantages, you can focus on how to design your control logic to interface with the DCFIFO megafunction rather than to interface with the clock domain directly.

However, a drawback of the DCFIFO megafunction is resource use. In general, the resources used in the DCFIFO megafunction are higher than resources used in the handshake protocol. This is because the DCFIFO is a buffer storage and utilizes RAM resources in the device.

If your design has limited-resource concerns and if data transfer speed does not matter, the handshake protocol is the better choice. However, in common applications where high-speed data transfer is required and time-to-market is critical, using the DCFIFO megafunction is a better choice.

Table 1 compares the DCFIFO megafunction and handshake protocol implementations.

Table 1. DCFIFO Megafunction versus Handshake Protocol Implementation

Requirements	DCFIFO	Handshake Protocol
Data Transfer	Fast	Slow
Designer Effort	Low	High
Resource Usage	High	Low

Design Example: Data Transfer Using the DCFIFO Megafunction

Complex designs often involve data transfer in the asynchronous clock domain. "Benefits of the DCFIFO Megafunction," on page 7 explained that you should not transfer data in asynchronous clock domains directly because of metastability issues.

This design example shows you how to use the DCFIFO megafunction as an interface for successful data transfer in asynchronous clock domains. It also provides details about how to design the write control logic and read control logic for monitoring and controlling write and read operations of the DCFIFO.

Use the ModelSim®-Altera software to simulate the design example. You can download the full design example at:

www.altera.com/literature/an/an473_design_example.zip

Table 2 describes the **.do** scripts included in this design example.

Table 2. ModelSim-Altera .do Scripts for the Design Example (Part 1 of 2)

ModelSim-Altera .do Script	Description
fast_to_slow_rtl.do	RTL functional simulation scripts for data transfer between fast clock domains and slow clock domains. All important internal signals are shown in the simulation.
slow_to_fast_rtl.do	RTL functional simulation scripts for data transfer between slow clock domains and fast clock domains. All important internal signals are shown in the simulation.

Table 2. ModelSim-Altera .do Scripts for the Design Example (Part 2 of 2)

ModelSim-Altera .do Script	Description
fast_to_slow_gate.do	Gate-level timing simulation scripts for data transfer between fast clock domains and slow clock domains. Only top-level pins are shown in the simulation. No internal signals are shown.
slow_to_fast_gate.do	Gate-level timing simulation scripts for data transfer between slow clock domains and fast clock domains. Only top-level pins are shown in the simulation. No internal signals are shown. To avoid any setup or hold violations during timing simulation, change the SDC file to **slow_to_fast_an_dcfifo_top.sdc** in the TimeQuest Timing Analyzer settings.

The focus of this application note is only for **fast_to_slow_rtl.do** scripts that run RTL functional simulations for data transfer from fast-to-slow clock domains.

Design Specification and Implementation

This design example includes a ROM megafunction, a RAM megafunction, and a DCFIFO megafunction. You can configure and build these megafunctions with the MegaWizard Plug-In Manager in the Quartus II software. The ROM and RAM megafunction are for demonstration purposes only. Actual applications may be different or more complicated.

Besides the three megafunctions that can be easily configured and built with the MegaWizard Plug-In Manager, two hand-coded control logic blocks (write control logic and read control logic) are required to ensure the proper transfer of data. The source code for this control logic is provided in the design example.

Figure 6. High-Level Block Diagram for the Design Example

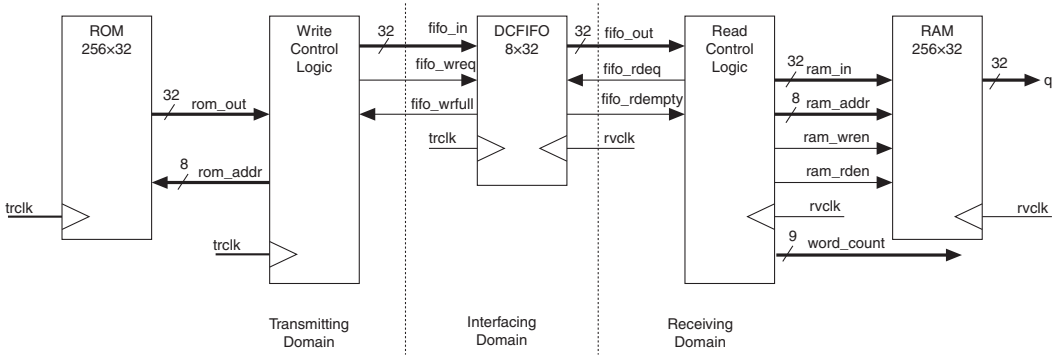


Figure 6 shows the high-level blocks of the design example. This design example is divided into three sections:

- Transmitting domain:
  - Consists of a ROM with 256-word depth and 32-bit data width, and a write control logic block.
  - The ROM stores the data that must eventually be transferred to the RAM in the receiving domain.
  - The write control logic block reads the data from ROM and writes into the DCFIFO. When 256 words of data have been read from ROM, the write operation is complete.
  - Clock `trclk` runs at 400 MHz. `trclk` is totally independent of `rvclk`.
- Interfacing domain:
  - Consists of a DCFIFO with 8-word depth and 32-bit data width.
  - The DCFIFO acts as the interface block between the ROM in the transmitting domain and the RAM in the receiving domain.
  - The DCFIFO asserts the `fifo_wrfull` signals to the write control logic to indicate the FIFO is full.
  - The DCFIFO sends the `fifo_rdempty` signal to the read control logic to indicate the FIFO is empty.
- Receiving domain:
  - Consists of a RAM and a read control logic block.
  - The RAM stores the data that is transferred from the ROM in the transmitting domain.
  - The read control logic block reads the data from DCFIFO and writes into the RAM. It also increases the counter for `word_count` each time a word is successfully read from the DCFIFO.
  - Clock `rvclk` runs at 100 MHz. `rvclk` is totally independent of `trclk`.

Megafunction Configuration Settings

Tables 3 through 5 show configuration options for the ROM megafunction, DCFIFO megafunction, and RAM megafunction, respectively. The megafunctions are configured in the MegaWizard Plug-In Manager as described in the tables.

Table 3. Configuration for ROM Megafunction

Function	Description
Currently selected device family	**Stratix III**
How wide should the 'q' output bus be?	**32 bits**
How many 32-bit words of memory?	**256 words**
What should the memory block type be?	**Auto**
Set the maximum block depth to	**Auto**
What clocking method would you like to use?	**Single Clock**
Which port should be registered?	**Only 'address' input port is registered**
Create one clock enable signal for each clock signal. All registered ports are controlled by the enable signal(s)	Disable
Create an 'aclr' asynchronous clear for the registered ports	Disable
Create a 'rden' read enable signal	Disable
Do you want to specify the initial content of the memory?	Yes. File name is **myrom.hex**
Allow In-System Memory Content Editor to capture and update content independently of the system clock	Disable

For more information, refer to the *Read Only Memory Megafunction User Guide (ROM)*.

Table 4. Configuration for DCFIFO Megafunction (Part 1 of 2)

Function	Description
Currently selected device family	**Stratix III**
How wide should be the FIFO be?	**32 bits**
Use a different output width and set to	Disable
How deep should the FIFO be?	**8 words**
Do you want a common clock for reading and writing FIFO?	**No**
Are the FIFO clocks synchronized?	**No**
Which optional output control signals do you want?	**Read-side**, select **empty** **Write-side**, select **full**

Table 4. Configuration for DCFIFO Megafunction (Part 2 of 2)

Function	Description
Asynchronous clear	Enable
Which kind of read access do you want with the 'rdreq' signal?	**Legacy synchronous FIFO mode**
What should the memory block type be?	Auto
Would you like to maximize performance but use more area?	**No (smallest area)**
Would you like to disable any circuitry protection?	Enable **Disable overflow checking** Enable **Disable underflow checking**
Implement FIFO function with logic cells only, even if the device contains EABs or ESBs	Disable

Disable overflow checking and underflow checking to reduce resource usage and improve performance. The write control logic and read control logic have already done this to prevent overflow or underflow of the DCFIFO. Overflow or underflow can cause data corruption.

For more information about single and dual-clock FIFOs, refer to the *First-In-First-Out Megafunction User Guide (FIFO)*.

Table 5. Configuration for RAM Megafunction (Part 1 of 2)

Function	Description
Currently selected device family	**Stratix III**
How wide should the 'q' output bus be?	**32 bits**
How many 32-bit words of memory?	**256 words**
What should the memory block type be?	**Auto**
What clocking method would you like to use?	**Single clock**
Which ports should be registered?	**Only 'data', 'wren', and 'address' input ports are registered**
Create one clock enable signal for each clock signal	Disable
Create a byte enable port	Disable
Create an 'aclr' asynchronous clear for the registered ports	Enable
Create a 'rden' read enable signal	Enable
What should the q output be when reading from a memory location being written to?	New Data

Table 5. Configuration for RAM Megafunction (Part 2 of 2)

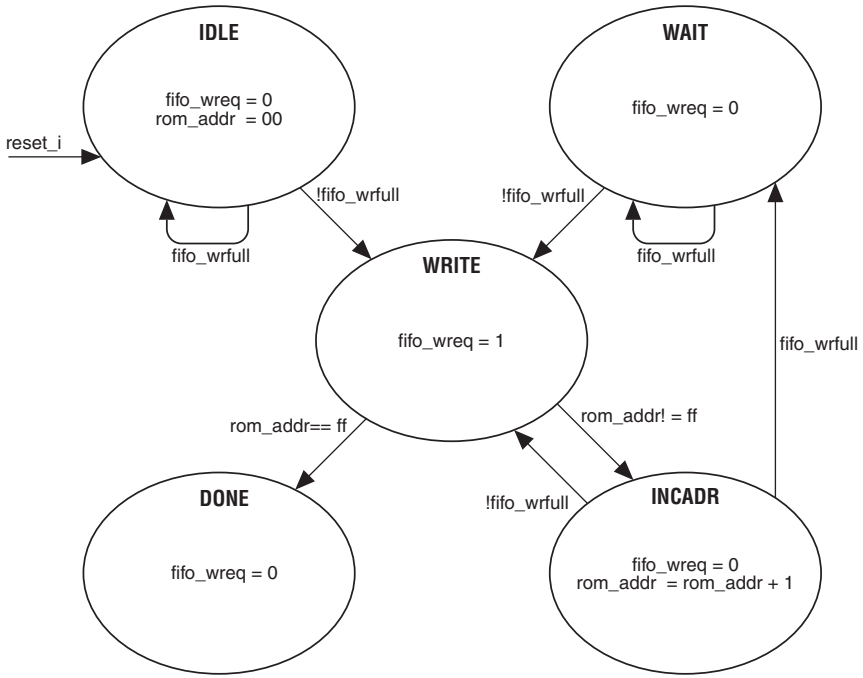
Function	Description
Get x's for write masked bytes instead of old data when byte enable is used	Enable
Do you want to specify the initial content of the memory?	No, leave it blank
Allow In-System Memory Content Editor to capture and update content independently of the system clock	Disable

For more information about the RAM megafunction, refer to the *Random Access Memory Megafunction User Guide (RAM)*.

Write Control Logic State Machine

Figure 7 shows the state machine for write control logic of the design example. The implementation of this state machine is written in **write_control_logic.v** in the design example.

Figure 7. State Machine for Write Control Logic



When the state machine is in the IDLE state, data at address 00 is ready to be read by the ROM. The state machine remains at this state if `reset_i` (asynchronous reset is pressed and held) or `fifo_wrfull` (the DCFIFO is full) are asserted. No operation is carried out. For other cases, the state machine transitions to the WRITE state at the next clocking edge.

At the WRITE state, `fifo_wreq` is asserted to allow write operation for the data from the respective ROM address into the DCFIFO. At the next clocking edge, the state machine transitions to the DONE state if it has written the last data (`rom_addr = ff`). Otherwise, the state machine transitions to the INCADR state.

At the INCADR state, `fifo_wreq` is de-asserted to abort the write operations. To write the subsequent data, the ROM address is increased by one. The increment of the address causes the data to be changed.

The ROM address is increased by one to prepare the next data to be written into the DCFIFO (`rom_addr` = `rom_addr` + 1). The `fifo_wrreq` is de-asserted to stop the write operation. At the next clocking edge, the

state machine transitions back to the WRITE state if the DCFIFO is not full (`!fifo_wrfull`). If the DCFIFO is full, the state machine transitions to the WAIT state.

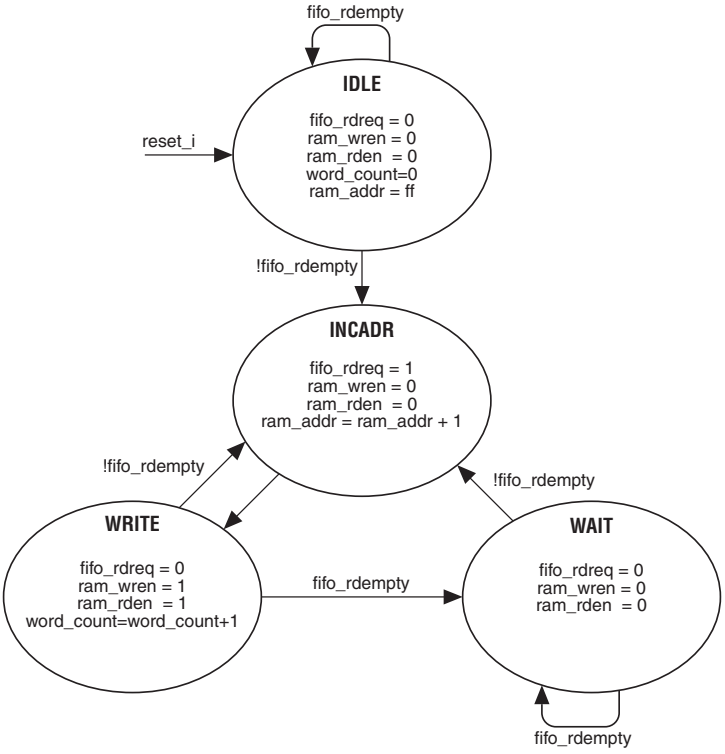
In the WAIT state, `fifo_wreq` remains de-asserted. The state machine transitions to the WRITE state only if the DCFIFO is not full (`!fifo_wrfull`) at the next clocking edge.

At the DONE state, all the data from the ROM has been written into the DCFIFO. The write operation from ROM to DCFIFO is complete.

Read Control Logic State Machine

Figure 8 shows the state machine for read control logic of the design example. The implementation of this state machine is written in **read_control_logic.v** in the design example.

Figure 8. State Machine for Read Control Logic



In the IDLE state, the RAM address is set to ff to accommodate the increment of the RAM address at the next state that results in the RAM address 00. It remains at this state if `reset_i` (asynchronous reset is pressed and held) or `fifo_rdempty` (DCFIFO is empty) are asserted and the `word_count` counter is also reset. No operation is carried out. For other cases, the state machine transitions to the INCADR state at the next clocking edge.

In the INCADR state, `fifo_rdreq` is asserted to allow the read operation from the DCFIFO. At the same time, the RAM address is increased by one (`ram_addr` = `ram_addr` + 1). If the previous state is IDLE, the RAM address is pointing at 00 so that the first data from the DCFIFO is written at the correct RAM address. At the next clocking edge, the state machine transitions to the WRITE state.

At the WRITE state, `fifo_rdreq` is de-asserted to abort the read operation. At this state, the respective data is shown on output of the DCFIFO and is ready to be written into the RAM. Upon aborting the read operation, `ram_wren` is asserted to allow the data to be written into the respective RAM address. At the same time, the `word_count` is increased by one to indicate that one word has been received in the receiving domain. In this design example, the RAM is configured to read the new data when read-during-write occurs and is targeted to the same memory location. Therefore, `ram_rden` is asserted to read the new written data. At the next clocking edge, the state machine transitions back to the INCADR state if the DCFIFO is not empty (`!fifo_rdempty`). If the DCFIFO is empty (`fifo_rdemtpy`), the state machine transitions to the WAIT state.

In the WAIT state, `fifo_rdreq` remains de-asserted. The state machine transitions to the INCADR state only if the DCFIFO is not empty (`!fifo_rdempty`) at the next clocking edge. If the last word is read out from DCFIFO (indication from `word_count`), it remains at this state.

Adding Timing Constraints to the Design Example

When you use the DCFIFO in this design example, apply the correct timing constraints so that the TimeQuest Timing Analyzer can analyze the design properly. The design example provides the **fast_to_slow_an_dcfifo_top.sdc** file as a reference for the settings in the timing constraints for clocks and false paths in this particular design.

The rvclk runs at 400 MHz and is independent of trclk, which runs at 100 MHz. To specify these clocks in the TimeQuest Timing Analyzer, use the following SDC command:

- `create_clock -name {rvclk} -period 10.000 -waveform { 0.000 5.000 } [get_ports {rvclk}] -add`
- `create_clock -name {trclk} -period 2.500 -waveform { 0.000 1.250 } [get_ports {trclk}] -add`

Because rvclk and trclk are independent or asynchronous, set the false path constraints on any path in between these two clock domains. The TimeQuest Timing Analyzer does not analyze these paths. To set the false path between rvclk and trclk, use the following SDC command:

- `set_false_path -from [get_clocks {rvclk}] -to [get_clocks {trclk}]`
- `set_false_path -from [get_clocks {trclk}] -to [get_clocks {rvclk}]`

However, if rvclk and trclk are synchronous, you cannot set the false path between the entire rvclk and trclk domain. There are some paths within the DCFIFO megafunction that must not be analyzed, especially if rvclk and trclk have a tight timing requirement. You should set false paths in the DCFIFO.

For write-to-read domains, set a false path between the delayed_wrptr_g and rs_dgwp registers:

```
set_false_path -from [get_registers {*dcfifo*delayed_wrptr_g[*]}] -to
[get_registers {*dcfifo*rs_dgwp*}]
```

For read-to-write domains, set a false path between the rdptr_g and ws_dgrp registers:

```
set_false_path -from [get_registers {*dcfifo*rdptr_g[*]}] -to
[get_registers {*dcfifo*ws_dgrp*}]
```

For more information about how to set timing constraints, refer to the *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the Quartus II Handbook.

Simulation Results

This section describes the functional simulation results of the design example. The design example uses Mentor Graphics® ModelSim-Altera to simulate the design.

To run the functional simulation with ModelSim-Altera software, invoke the ModelSim-Altera software and perform the following steps:

1. Change the directory name to <*project_dir*>**/simulation/modelsim**.
2. To run the <*project_dir*>**/simulation/modelsim/fast_to_slow_rtl.do** script, type `do fast_to_slow_rtl.do` in the ModelSim-Altera simulator console window.

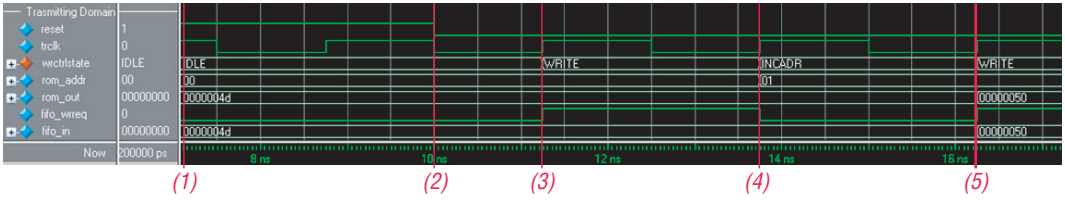
For more information about how to use the Mentor Graphics ModelSim simulator tool, refer to *Mentor Graphics ModelSim Support* chapter in volume 3 of the *Quartus II Handbook*.

The simulation results are divided into transmitting domain and receiving domain sections. The transmitting domain section consists of all the signals in the transmitting domain; the receiving domain section consists of all the signals in the receiving domain.

Initial Write Operation to the DCFIFO

Figure 9 shows the functional simulation results at the receiving domain for initial write operation to the DCFIFO while the DCFIFO is not full.

Figure 9. Initial Write Operation to the DCFIFO



Notes to Figure 9:

(1) Initially, the reset signal is asserted. The `wrctrlstate` signal is in the IDLE state. The rest of the input signals (`rom_addr` and `fifo_wrreq`) are initialized to 0. `rom_out` outputs the first data 4d from the `rom_addr 00`. The `fifo_in` signal shows the same value, 4d, because it is directly connected to the `rom_out`.

(2) At 10 ns, the reset signal is de-asserted.

(3) At 11.25 ns, `wrctrlstate` transitions to the WRITE state. During the WRITE state, `fifo_wrreq` is asserted.

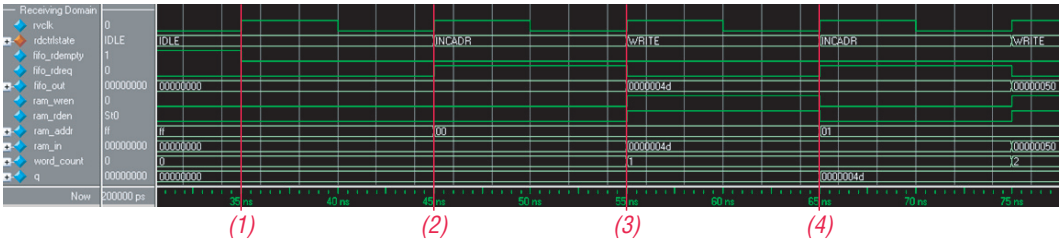
(4) At 13.75 ns, `wrctrlstate` transitions to the INCADR state. During the INCADR state, it de-asserts `fifo_wrreq` and increments the `rom_addr` to 01. The `rom_out` outputs the respective data (that is, 50) at the next rising edge of `trclk`.

(5) At 16.25 ns, `wrctrlsate` transitions to the WRITE state. The same state transition continues as stated in *Notes (3)* and *(4)* if `fifo_wrfull` is not asserted. Therefore, the WRITE state happens every two `trclk` cycles to write the data into the DCFIFO.

Initial Read Operation from the DCFIFO

Figure 10 shows the functional simulation results at the receiving domain for the initial read operation from the DCFIFO.

Figure 10. Initial Read Operation from the DCFIFO



Notes to Figure 10:

(1) At 35 ns, `fifo_rdempty` is de-asserted, indicating the DCFIFO is not empty and ready to be read. After the `fifo_rdempty` signal is de-asserted, `rdctrlstate` always transitions to the INCADR state at the next rising `rvclk` edge.

(2) At 45 ns, `rdctrlstate` transitions to the INCADR state. During the INCADR state, it increments `ram_addr` to 00 from ff. It asserts the `fifo_rdreq` signal to read the first data inside the DCFIFO. The `fifo_out` signal outputs the first data (that is, 4d) at the next rising edge of `rvclk`. The `ram_in` signal shows the same value, 4d, because it is directly connected to the `fifo_out`.

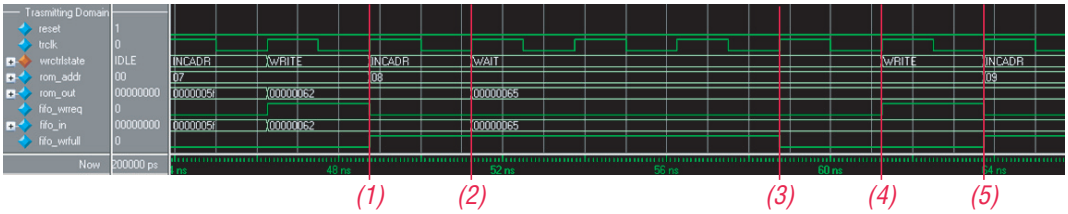
(3) At 55 ns, `rdctrlstate` transitions to the WRITE state. During the WRITE state, `fifo_rdreq` is de-asserted after the data has been successfully read out from the DCFIFO. During this state, it increments the `word_count` from 0 to 1. At the same time, it asserts both `ram_wren` and `ram_rden`. Therefore, q outputs the respective data (that is, 4d) at the next rising edge of `rvclk`.

(4) At 65 ns, `rdctrtlstate` transitions to the INCADR state. The same state transition continues as stated in *Notes (2)* and *(3)* if `fifo_rdempty` is not asserted. Therefore, the WRITE state happens every two `rvclk` cycles to write the data into the RAM.

Write Operation when DCFIFO is Full

Figure 11 shows the functional simulation results at the transmitting domain for the Write operation when the DCFIFO is full.

Figure 11. Write Operation when DCFIFO is Full



Notes to Figure 11:

(1) At 48.75 ns, `fifo_wrfull` is asserted to indicate the DCFIFO is full. After `fifo_wrfull` is asserted, `wrtcltrsate` always transitions from the INCADR state to the WAIT state at the next rising edge of `trclk`.

(2) At 51.25 ns, `wrctrlstate` transitions to the WAIT state. During the WAIT state, it holds the `rom_addr` (that is, 08) so that the respective data is written into the DCFIFO at the next WRITE state. The WAIT state continues until `fifo_wrfull` is de-asserted.

(3) At 58.75 ns, `fifo_wrfull` is de-asserted. After `fifo_wrfull` is de-asserted, `wrtcltrsate` always transitions from the WAIT state to the WRITE state at the next rising edge of `trclk`.

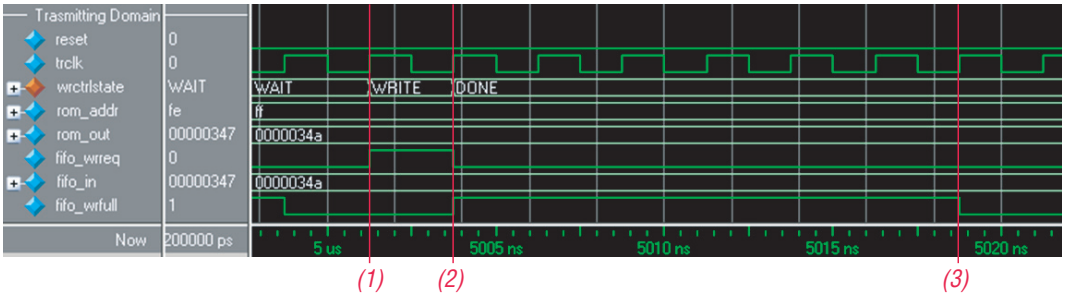
(4) At 61.25 ns, `wrctlstate` transitions to the WRITE state to perform the write operation.

(5) At 63.75 ns, the process is repeated as stated in *Notes (1)* through *(4)* after the `fifo_wrfull` is asserted.

Completion of Data Transfer from ROM to DCFIFO

Figure 12 shows the functional simulation results for completion of the data transfer from ROM to DCFIFO.

Figure 12. Completion of Data Transfer from ROM to DCFIFO



Notes to Figure 12:

(1) At ~5001 ns, the last data (that is, 34a) is successfully written into the DCFIFO. The `wrctrlstate` signal transitions to the DONE state at the next rising edge of `trclk` to indicate the completion of data transfer from the ROM to the DCFIFO.

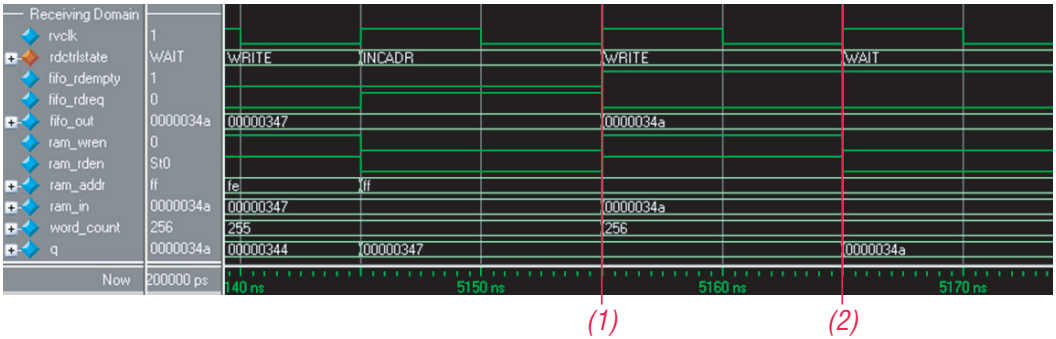
(2) At ~5003 ns, `wrctrlstate` transitions to the DONE state. `fifo_wrreq` is de-asserted to disable the write operation and `fifo_wrfull` is asserted to indicate the DCFIFO is full. Even though the data transfer is complete at the transmitting domain, the read operation continues at the receiving domain. However, `fifo_wrfull` is not de-asserted immediately because of the latency in the DCFIFO.

(3) At ~5018 ns, `fifo_wrfull` is de-asserted. The receiving domain continues to read out the data until `fifo_rdempty` is asserted.

Completion of Data Transfer from DCFIFO to RAM

Figure 13 shows the functional simulation results for completion of the data transfer from DCFIFO to RAM.

Figure 13. Completion of Data Transfer from DCFIFO to RAM



Notes to Figure 13:

(1) At 5155 ns, the last data (that is, 34a) is written into the RAM. The `fifo_rdempty` signal is asserted. The `word_count` increments to 256 and is the last word to be transferred in this design example.

(2) At 5165 ns, `rctrlstate` transitions to the WAIT state after the completion of data transfer from the DCFIFO to the RAM.

To verify the results, compare the q outputs with **myrom.hex** provided in the design example. Open **myrom.hex** with the Quartus II software and select the word size as 32 bit. The q outputs should display the same results as in the **myrom.hex** file.

Conclusion

Data transfers in asynchronous clock domains require special handling to ensure data is properly transferred from one domain to another. Improper handling in asynchronous clock domains can result in setup and hold time violations and cause the metastable data to propagate through the rest of your design. You can use the handshake protocol to build logic that handles the metastable issue, but it takes more design time and effort, and data transfer speed is slow.

With the DCFIFO megafunction, you can perform data transfers between two asynchronous clock domains easily because the megafunction prevents metastable issues and handles data transfer properly. Also, with the DCFIFO megafunction, you do not have to design a handshake protocol. You must have write control logic to control data writes to the DCFIFO from the transmitting domain, and you must have read control logic to control data reads from the DCFIFO to the receiving domain to

output the number words of data that has been received. Depending on the application, you can use the number of words to identify the packet or block boundary recognition.

This application note provides a design example with the source code for the control logic for the DCFIFO block. However, you might need to modify the code according to your design requirements. In conclusion, you can implement data transfer in asynchronous clock domains using the DCFIFO megafunction with proper control logic.

Referenced Documents

This application note references the following documents:

- *Application Note 42: Metastability in Altera Devices*
- *First-In-First-Out Megafunction User Guide (FIFO)*
- *Mentor Graphics ModelSim Support*
- *Quartus II TimeQuest Timing Analyzer*
- *Random Access Memory Megafunction User Guide (RAM)*
- *Read Only Memory Megafunction User Guide (ROM)*

Document Revision History

Table 6 shows the revision history for this application note.

Table 6. Document Revision History

Date and Document Version	Changes Made	Comment
December 2007, v1.0	Initial release.	

101 Innovation Drive
San Jose, CA 95134
www.altera.com
Technical Support:
www.altera.com/support
Literature Services:
www.altera.com/literature