LogiCORE IP Divider Generator v4.0

DS819 June 22, 2011 **Product Specification**

Introduction

The Xilinx® LogiCORE™ IP Divider Generator core v4.0 creates a circuit for integer division based on Radix-2 non-restoring division, or High-Radix division with prescaling. The Radix-2 algorithm exploits fabric to achieve a range of throughput options that includes single cycle, and the high Radix algorithm exploits XtremeDSP™ slices at lower throughput, but with reuse to reduce resources.

Features

- AXI4-Stream-compliant interfaces
- Integer division with operands of up to 64 bits wide.
- Performs Radix-2 integer division for fabric-only implementation or High Radix division with prescaling to take advantage of XtremeDSP slices.
- Optional operand widths, synchronous controls, and selectable latency.
- For use with Xilinx CORE Generator™ and Xilinx System Generator for DSP 13.2.

LogiCORE IP Facts	
Core Specifics	
Supported Device Family(1)	Virtex-7, Kintex-7, Artix™-7, Zynq™-7000, Virtex-6, Spartan-6
Supported User Interfaces	AXI4-Stream
Configuration	See Tables 8 to 15
Provided with Core	
Documentation	Product Specification
Design Files	Netlist
Example Design	Not Provided
Test Bench	VHDL
Constraints File	N/A
Simulation Model	VHDL and Verilog
Tested Design Tools	
Design Entry Tools	CORE Generator 13.2 System Generator for DSP 13.2
Simulation(2)	Mentor Graphics ModelSim Cadence Incisive Enterprise Simulator (IES) Synopsys VCS and VCS MX ISim
Synthesis Tools	XST 13.2
Support	
Provided by Xilinx, Inc.	

1. For the complete list of supported devices, see the release notes for this core.
2. For the supported version of the tools, see the ISE Design Suite 13: Release Notes Guide

Functional Overview

Two implementations of division are supported by Divider Generator v4.0:

- **Radix-2.** Radix-2 non-restoring integer division using integer operands, allowing either a fractional or integer remainder to be generated. This is recommended for operand widths less than around 16 bits or for applications requiring high throughput. The implementation uses fabric primitives (registers and LUTs).
- **High Radix**. High Radix division with prescaling. This is recommended for operand widths greater than around 16 bits. This implementation uses XtremeDSP slices.

A detailed explanation of each implementation is provided in Radix-2 Options and High Radix Options.

Applications

Division is the most complex of the four basic arithmetic operations. Because hardware solutions are correspondingly larger and more complex than the solutions for other operations, it is best to minimize the number of divisions in any algorithm. There are many forms of division implementation, each of which can offer the optimal solution in different circumstances.

The Divider Generator core provides two division algorithms, offering solutions targeted at small operands and large operands.

The Radix-2 non-restoring algorithm solves one bit of the quotient per cycle using addition and subtraction. The design is fully pipelined, and can achieve a throughput of one division per clock cycle. If full throughput is not required, the divisions per clock parameter can be set to 2, 4 or 8. This causes an iterative solution to be generated which uses less resource by re-using the calculation engine. This algorithm naturally generates a remainder, so is the choice for applications requiring integer remainders or modulus results.

The High Radix with prescaling algorithm resolves multiple bits of the quotient at a time. It is implemented as an iterative engine and so throughput is a function of the number of iterations required. The prescaling prior to the iterative operation causes an overhead of resource which makes this algorithm less suitable for smaller operands. Although the iterative calculation is more complex than for Radix-2, taking more cycles to perform, the number of bits of quotient resolved per iteration and its use of XtremeDSP slices makes this the preferred option for larger operand widths.

Functional Description

The Divider Generator core uses one of two implementations as selected by the user. The Radix 2 solution is recommended for smaller operand widths, for high throughput or situations where XtremeDSP slices use must be minimized. The High Radix solution is recommended for larger operand widths. Because the two solutions differ in so many aspects of parameter ranges, throughput, latency, etc. they are described in this section separately.

Radix-2 Feature Summary

- Provides quotient with integer or fractional remainder
- Pipelined, parallel architecture for increased throughput
- Pipeline reduction for size versus throughput selections
- Dividend width from 2 to 64 bits
- Divisor width from 2 to 64 bits
- Independent dividend, divisor and fractional bit widths
- Fully synchronous design using a single clock

- Supports unsigned or two's complement signed numbers
- Can implement 1/X (reciprocal) function

Radix-2 Solution Overview

This parameterized solution divides an M-bit-wide variable dividend by an N-bit-wide variable divisor. The output consists of the quotient and either an integer remainder or fractional result (quotient continued past the binary point). In the integer remainder case, the result of the division is an M-bit-wide quotient with an N-bit-wide integer remainder (Equation 1). In the fractional case, the result is an M-bit-wide quotient with an F-bit-wide fractional remainder (Equation 2). When signed operation is selected, all operands and results employ a two's complement sign bit, resulting in one less bit of magnitude result (Equation 3).

Integer remainder case:

$$Dividend = Quotient * Divisor + IntRmd$$

Equation 1

F-bit-wide fractional remainder in the unsigned case:

$$FractRmd = \frac{IntRmd * 2^{F}}{Divisor}$$

Equation 2

F-bit-wide fractional remainder in the signed case:

$$FractRmd = \frac{IntRmd * 2^{(F-1)}}{Divisor}$$

Equation 3

For signed mode with integer remainder, the sign of the quotient and remainder correspond exactly to Equation 1.

Thus,

6/-4 = -1 REMD 2

whereas

-6/4 = -1 REMD –2

For signed mode with fractional remainder, the sign bit is present both in the quotient and the fractional remainder. For example, for a five-bit dividend, divisor and fractional remainder we have:

-9/4 = 9/-4 = -(2 1/4)

This corresponds to:

10111/00100 or 01001/11100

Giving the result:

Quotient = 11110 (= -2)

Remainder = 11100 (= -1/4)

For division by zero, the quotient, remainder, and fractional results are undefined.

The core is highly pipelined. The throughput of the core is configurable and can be reduced from 1 clock cycle per division to 2, 4 or 8 clock cycles per division to reduce resources.

The dividend and divisor bit widths can be set independently. The bit width of the quotient is equal to the bit width of the dividend. The bit width of the integer remainder is equal to the width of the divisor. For fractional output, the remainder bit width is independent of the dividend and divisor. The core handles data ranges of 2 to 64 bits for dividend, divisor, and fractional outputs.

The divider can be used to implement the reciprocal of X; that is the 1/X function. To do this, the dividend bit width is set to 2 and fractional mode is selected. The dividend input is then tied to 01 for both unsigned or signed operation, and the X value is provided via the divisor input.

Following a power-on reset or `aresetn`, the core outputs zeros on `QUOTIENT` and `FRACTIONAL` (see TDATA Structure for Output (DOUT) Channel) outputs until new results appear.

Radix-2 Latency and Throughput

The total latency (number of enabled clock cycles required before the core generates the first valid output) is a function of the bit width of the dividend. If fractional output is required, the latency is also a function of the fractional bit width. In general:

- Latency is of the order M for integer remainder dividers, where M is the width of the Quotient
- Latency is of the order M + F for fractional remainder dividers where F is the width of the Fractional output

Table 1 provides a list of the latency formula for divider selections.

Table 1: **Latency of Radix-2 Solution Based on Divider Parameters**

Signed	Fractional	Clocks Per Division	Latency(1)
False	False	1	M+A+2
False	False	>1	M+A+3
False	True	1	M+F+A+2
False	True	>1	M+F+A+3
True	False	1	M+A+4
True	False	>1	M+A+5
True	True	1	M+F+A+4
True	True	>1	M+F+A+5

Notes:

1. M = Dividend and Quotient Width, F = Fractional Width, A = total Latency of AXI interfaces.

The Clocks per Division parameter allows a range of choices of throughput versus resources. With Clocks per Division set to 1, the core is fully pipelined, so it has maximal throughput of one division per clock cycle, but uses the most resources. Clock per Division settings of 2, 4, and 8 reduce the throughput by those respective factors for smaller core sizes.

AXI interfaces give an additional latency of 0 for Non-Blocking, 1 for Blocking with no Output TREADY and 3 for Blocking with Output TREADY (M_AXIS_DOUT_TREADY). However, when Blocking mode is selected, latency varies by run time.

High Radix Solution Feature Summary

- High Radix division enabled by prescaling
- Provides quotient and, optionally, fractional outputs
- Configurable widths, synchronous controls, selectable latency and detection of division by zero
- Uses XtremeDSP slices

High Radix Solution Overview

The High Radix implementation performs division by prescaling operands before employing an accelerated High Radix division algorithm. The design is fully pipelined for maximum clock frequency. First, the divisor is normalized, then an estimate of its reciprocal is made. Both operands are multiplied by this estimate to bring the divisor closer to 1. The precision and accuracy of the prescale determines how many bits of quotient can be resolved on each subsequent iteration. The fact that the prescaled divisor is close to one allows the estimate of new quotient bits to be just the top bits of the residue left from the previous iteration. The iterative operation itself is performed in carry-save notation, so that no long carry chains limit performance. Because only the top bits of the residue are used as the estimate and the divisor is not exactly 1, errors do occur in the internal result of each iteration; thus, the quotient bits resolved on each iteration overlap slightly with the previously resolved bits to allow correction of errors in subsequent iterations.

Because the iteration calculation consists of a carry-save multiplication and subtraction, it is ideally suited to the XtremeDSP (multiply-add) slices, providing an efficient, low-latency iteration.

Throughput Considerations

The iterative process is implemented as a loop rather than an unrolled data pipeline to reduce resources. This means that new input must be held off until previous calculations are finished within the iterative circuit. The maximum possible throughput is therefore 1/N divisions per clock, where N is the number of iterations required. However, to achieve this maximum throughput the input might be required to be bursty. This is because the iterative engine can be pipelined with each stage of the pipe offering a carousel place for interlaced divisions.

With the addition of AXI4-Stream interfaces, average throughput is unchanged. The Blocking modes provide an element of FIFO buffering to the data, so it is not possible to make deterministic predictions of when the core is ready to accept new data. For NonBlocking mode timing is more predictable. The GUI provides feedback of the rate (1 in N) at which the divider can accept input on a continuous basis with a constant interval. This is expressed on the "Throughput" field of the GUI and is expressed in terms of 1 input every N enabled clock cycles.

Tables 2 and 3 show latency for the High Radix solution. To this, add 0 for NonBlocking mode, 1 for Blocking mode with no output TREADY and 3 for Blocking mode with output TREADY.

Table 2: **Minimum Latency of High-Radix Solution Based on Divider Parameters**

Dividend and Quotient Width + Fractional Width					
4 to 12	**13 to 26**	**27 to 40**	**41 to 54**	**55 to 68**	**69 to 82**
2	3	4	5	6	7

Table 3: **Maximum Latency of High-Radix Solution Based on Divider Parameters**

Divisor Width	Dividend and Quotient Width + Fractional Width					
	4 to 12	**13 to 26**	**27 to 40**	**41 to 54**	**55 to 68**	**69 to 82**
4 to 8	16	20	24	29	33	37
9 to 18	17	21	25	30	34	38
19 to 32	18	22	26	31	35	39
33 to 35	19	23	27	32	36	40
36 to 48	20	24	28	33	37	41
49 to 52	22	26	30	35	39	43
53 to 54	23	27	31	36/	40	44

Pinout

The core pinout and signal names are shown in Figure 1 and defined in Table 4

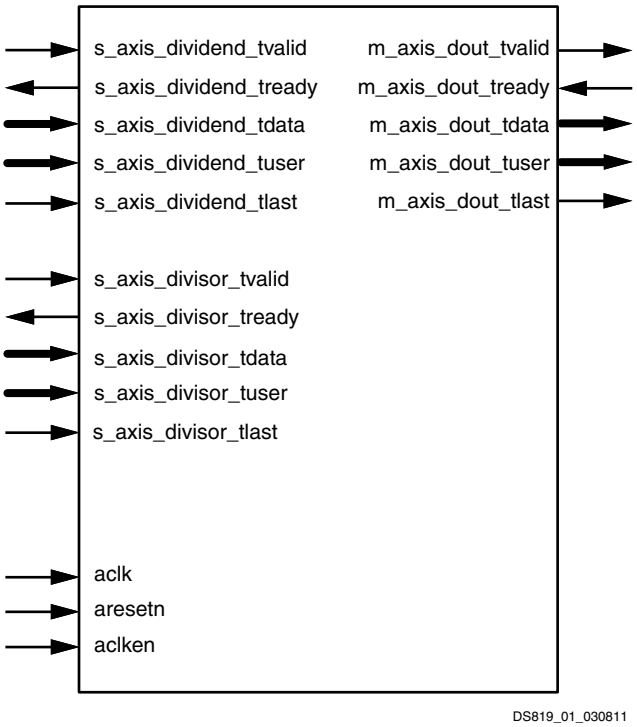

Figure 1: **Core Pinout Diagram**

Table 4: **Signal Pinout**

Signal	Direction[1]	Optional	Description
aclk	Input	No	Rising edge clock.
aclken	Input	Yes	Active high clock enable.
aresetn	Input	Yes	Active low synchronous clear (optional, always take priority over aclken) aresetn should be asserted or de-asserted for not less than two aclk cycles.
s_axis_dividend_tvalid	Input	No	TVALID for s_axis_dividend channel. See AXI4-Stream Considerations for protocol.
s_axis_dividend_tready	Output	Yes	TREADY for s_axis_dividend channel.
s_axis_dividend_tdata	Input	No	TDATA for s_axis_dividend channel. See TDATA Packing for internal structure and width.
s_axis_dividend_tuser	Input	Yes	TUSER for s_axis_dividend channel.
s_axis_dividend_tlast	Input	Yes	TLAST for s_axis_dividend channel.
s_axis_divisor_tvalid	Input	No	TVALID for s_axis_divisor channel.
s_axis_divisor_tready	Output	Yes	TREADY for s_axis_divisor channel.
s_axis_divisor_tdata	Input	No	TDATA for s_axis_divisor channel. See TDATA Packing for internal structure and width.
s_axis_divisor_tuser	Input	Yes	TUSER for s_axis_divisor channel.
s_axis_divisor_tlast	Input	Yes	TLAST for s_axis_divisor channel.

Table 4: **Signal Pinout *(Cont'd)***

Signal	Direction[1]	Optional	Description
m_axis_dout_tvalid	Output	No	TVALID for m_axis_dout channel.
m_axis_dout_tready	Input	Yes	TREADY for m_axis_dout channel.
m_axis_dout_tdata	Output	No	TDATA for m_axis_dout channel. See TDATA Packing for internal structure and width.
m_axis_dout_tuser	Output	Yes	TUSER for m_axis_dout channel.
m_axis_dout_tlast	Output	Yes	TLAST for m_axis_dout channel.

Notes:

1. Dividend and Quotient Width must be set to satisfy the largest possible quotient result. Due to the non-symmetry of two's complement representation bit growth from the dividend to quotient is possible, but only for the single combination of the most negative number divided by negative one (that is, $-2^{(M-1)}/-1$). The width of dividend and quotient can be extended by 1 bit should this situation need to be accommodated.

CORE Generator Graphical User Interface

The Divider Generator core GUI provides one page split into sections to set parameter values for the particular instantiation required. This section provides a description of each GUI field. These fields are grouped as follows:

- **Component Name**: The base name of the output files generated for the core. Names must begin with a letter and be composed of any of the following characters: a to z, 0 to 9 and "_".

Common Options

Describes parameters common to both implementations and allows the selection of the divider implementation.

- **Algorithm Type**: This selects between Radix-2 and High Radix division solutions.
- **Operand Sign**: Signed or unsigned. Determines the interpretation of the input and output buses. Operand sign for the High Radix solution is always signed, and for this solution the FRACTIONAL field of the output channel is always unsigned.

Dividend Channel

- **Dividend Width**: Specifies the number of integer bits provided on the DIVIDEND (S_AXIS_DIVIDEND_TDATA) and QUOTIENT fields (subfield of M_AXIS_DOUT_TDATA). This must be set to satisfy the largest possible quotient result. Due to the non-symmetry of two's complement representation bit growth from the dividend to quotient is possible, but only for the single combination of the most negative number divided by negative one (that is, $-2^{(M-1)}/-1$). The width of dividend (and hence quotient) can be extended by 1 bit should this situation need to be accommodated
- **Has TLAST**: Specifies whether the this channel has a TLAST port. The Divider Generator core does not use this information. The facility is provided to ease system design. TLAST information is conveyed to the output channel with the same latency as the datapath.
- **Has TUSER**: Specifies whether this channel has a TUSER port. As with TLAST, the Divider Generator core does not use this information. TUSER exists to ease system design. TUSER bits are conveyed to the output with the same latency as the datapath.
- **TUSER Width**: Available when Has TUSER is true, this sets the width of the TUSER port for this channel.

Divisor Channel

- **Divisor Width**: Specifies the number of integer bits provided on the DIVISOR field of `s_axis_divisor_tdata`. When the core is configured for Radix-2 with remainder output, the width of the remainder is also equal to the value of this parameter.

- **Has TLAST**: Specifies whether the this channel has a TLAST port. The Divider Generator core does not use this information. The facility is provided to ease system design. TLAST information is conveyed to the output channel with the same latency as the datapath.
- **Has TUSER**: Specifies whether this channel has a TUSER port. As with TLAST, the Divider Generator core does not use this information. TUSER exists to ease system design. TUSER bits are conveyed to the output with the same latency as the datapath.
- **TUSER Width**: Available when Has TUSER is true, this sets the width of the TUSER port for this channel.

Output Channel

- **Remainder Type**: This selects between remainder types Fractional and Remainder presented on the FRACTIONAL field of the output TDATA port (`m_axis_dout_tdata`). Fractional remainder type is the only option for High Radix.
- **Fractional Width**: If Fractional remainder type is selected, this determines the number of bits provided on the FRACTIONAL field of the output channel (`m_axis_dout_tdata`). When High Radix is selected, the total output width (quotient part plus fractional part) is limited to 82.

The width of the quotient is equal to the width of the dividend and is set in the Dividend channel section.

The width of the TUSER port is the sum of the present input channel TUSER fields plus one if divide_by_zero detect is active. See AXI4-Stream Considerations for the internal structure of the TUSER port.

This channel also has a TLAST port if either of the input channels has a TLAST port.

Radix-2 Options

- **Clocks Per Division**: Determines the throughput of the Radix 2 solution (interval in clocks between inputs (or outputs)). A low value for this parameter results in high throughput, but also in greater resource use.

High Radix Options

- **Detect Divide-by-Zero**: Check box. Determines if the core has a DIVIDE_BY_ZERO field in the output TUSER port (`m_axis_dout_tuser`) to signal when a division by zero has been performed.
- **Number of iterations:** Read-only text field that reports the number of iterations performed by the High-Radix engine for each divide. This sets the maximum throughput of the divider. To achieve this throughput, the operands must be supplied as soon as requested by the core S_AXIS_DIVIDEND_TREADY and S_AXIS_DIVISOR_TREADY outputs.
- **Throughput:** Read-only text field that reports the maximum throughput that can be sustained by the divider when operands are supplied at a constant rate. In AXI blocking modes, throughput might be slightly higher due to buffering. This rate applies when FlowControl is set to NonBlocking and the output channel DOUT has no TREADY.

AXI4-Stream Options

- **Flow Control**: Blocking or NonBlocking. This is more fully explained in AXI4-Stream Considerations. NonBlocking mode provides an easier migration path from the previous version of Divider Generator core. Blocking mode eases data flow management to/from other AXI4-Stream Blocking mode cores at the expense of some additional resource and latency.
- **Optimize Goal**: This applies only to Blocking mode. When ACLKEN is selected and Optimize Goal is set to Resources, performance might be reduced. See Performance and Resource Utilization.
- **Output has TREADY**: Selects whether the output channel has a TREADY signal. This is required to allow back pressure from downstream, for example, if connected to another AXI4-Stream Blocking core. Without OutTready, downstream circuitry cannot halt dataflow from the divider, but some resource is saved.

- **Output TLAST Behavior**: Selects the source of the output channel TLAST signal. When neither or only one input channel has a TLAST then the output TLAST is not present or derives from the input TLAST appropriately. When both input channels have TLAST, the output channel TLAST can derive from either alone, the logical OR of both inputs, or the logical AND of both inputs.

Latency Options

- **Latency Configuration**: Automatic (fully pipelined) or manual (determined by following field). Latency Configuration for Radix-2 solution is always Automatic.
- **Latency**: When Latency Configuration is set to Automatic, this field provides the latency from input to output in terms of clock enabled clock cycles. When Manual, this field is used to specify the latency required. When high performance (clock frequency) is not required, a lower value in this field can save resources.

Control Signals

- **ACLKEN**: Determines if the core has a clock enable input (`aclken`).
- **ARESETN**: Determines if the core has an active low synchronous clear input (`aresetn`).

  Note:

  a. The signal `aresetn` always takes priority over `aclken`, that is, `aresetn` takes effect regardless of the state of `aclken`.

  b. The signal `aresetn` is active low.

  c. The signal `aresetn` should be held active for at least 2 clock cycles. This is because, for performance, `aresetn` is internally registered before being fed to the reset port of primitives.

System Generator For DSP Graphical User Interface

This section describes the System Generator for DSP GUI and details the parameters that differ from the CORE Generator GUI.

The Divider Generator core can be found in the Xilinx Blockset in the Math section. The block is called "Divider Generator 4.0."

See the System Generator for DSP Help page for the "Divider Generator 4.0" block for more information on parameters not mentioned here.

The System Generator for DSP GUI offers the same parameters as the CORE Generator GUI, with the exception that the Operand Sign is inferred from the input operands.

Using the Divider Generator IP Core

The CORE Generator GUI performs error-checking on all input parameters. Optimum latency information is also available.

Several files are produced when a core is generated, and customized instantiation templates for Verilog and VHDL design flows are provided in the `.veo` and `.vho` files, respectively. For detailed instructions, see the CORE Generator software documentation.

Simulation Models

The core has a number of options for simulation models:

- VHDL UNISIM-based structural simulation model
- Verilog UNISIM-based structural simulation model

The models required can be selected in the CORE Generator project options.

Xilinx recommends that simulations utilizing UNISIM-based structural models are run using a resolution of 1 ps. Some Xilinx library components require a 1 ps resolution to work properly in either functional or timing simulation. The UNISIM-based structural simulation models can produce incorrect results if simulated with a resolution other than 1 ps. See the "Register Transfer Level (RTL) Simulation Using Xilinx Libraries" section in *Chapter 6* of the Synthesis and Simulation Design Guide [Ref 3]. This document is part of the ISE® Software Manuals set available at www.xilinx.com/support/software_manuals.htm.

XCO Parameters

Table 5 defines the mapping between GUI parameters and XCO parameters.

Table 5: **GUI and XCO Parameter Mapping**

GUI Parameter	Default Value	XCO Values	XCO Parameter
Common Parameters			
Algorithm Type	Radix2	Radix2, High_Radix	algorithm_type
Operand Sign	Signed	Unsigned, Signed Must be Signed for High_Radix	operand_sign
Dividend Channel			
Dividend Width(1)(2)	16	2 to 64 for Radix2, 4 to 64 for High_Radix	dividend_and_quotient_width
Has TLAST	false	false, true	dividend_has_tlast
Has TUSER	false	false, true	dividend_has_tuser
TUSER Width	1	1 to 256	dividend_tuser_width
Divisor Channel			
Divisor Width	16	2 to 64 for Radix2, 4 to 64 for High_Radix	divisor_width
Has TLAST	false	false, true	divisor_has_tlast
Has TUSER	false	false, true	divisor_has_tuser
TUSER Width	1	1 to 256	divisor_tuser_width
Output Channel			
Remainder Type	Remainder	Remainder, Fractional	remainder _type
Fractional Width(2)	16	2 to 64 for Radix2 0 to 64 for High Radix	fractional_width
Radix 2 Options			
Clocks per Division	1	1, 2, 4, 8	clocks_per_division
High Radix Divider Parameter			
Detect Divide-By-Zero	false	false, true	divide_by_zero_detect

Table 5: **GUI and XCO Parameter Mapping *(Cont'd)***

GUI Parameter	Default Value	XCO Values	XCO Parameter
AXI4-Stream Options			
Flow Control	NonBlocking	NonBlocking, Blocking	FlowControl
Optimize Goal	Resource	Resources, Performance	OptimizeGoal
Output has TREADY	false	false, true	OutTready
Output TLAST Behavior	null	null, Pass_dividend_tlast, Pass_divisor_tlast, Or_all_tlasts, And_all_Tlasts	OutTLASTBehv
Latency Options			
Latency Configuration	Automatic	Automatic, Manual Must be Automatic for Radix2	latency_configuration
Latency	18	Range is a function of other parameters as summarized in Table 1, Table 2 and Table 3.	latency
Control Signals			
ACLKEN	false	false, true	aclken
ARESETN	false	false, true	aresetn

Notes:

1. Dividend Width also determines the Quotient width. It must be set to satisfy the largest possible quotient result. Due to the non-symmetry of two's complement representation bit growth from the dividend to quotient is possible, but only for the single combination of the most negative number divided by negative one (that is, $-2^{(M-1)}/-1$). The width of dividend and quotient can be extended by 1 bit should this situation need to be accommodated.
2. When High Radix the total output width (Quotient width plus Fractional Width) is limited to 82.

Demonstration Test Bench

When the core is generated using CORE Generator, a demonstration test bench is created. This is a simple VHDL test bench that exercises the core.

The demonstration test bench source code is one VHDL file: `demo_tb/tb_<component_name>.vhd` in the CORE Generator output directory. The source code is comprehensively commented.

Using the Demonstration Test Bench

The demonstration test bench instantiates the generated Divider Generator core. Either the behavioral model or the netlist can be simulated within the demonstration test bench.

- Behavioral model: Ensure that the CORE Generator project options are set to generate a behavioral model. After generation, this creates a behavioral model wrapper named `<component_name>.vhd`. Compile this file into the work library (see your simulator documentation for information on how to do this).
- Netlist: If the CORE Generator project options were set to generate a structural model, a VHDL or Verilog netlist named `<component_name>.vhd` or `<component_name>.v` was generated. If this option was not set, generate a netlist using the netgen program, for example:

  ```
  netgen -sim -ofmt vhdl <component_name>.ngc <component_name>_netlist.vhd
  ```

  Compile the netlist into the work library (see your simulator documentation for more information).

Compile the demonstration test bench into the work library. Then simulate the demonstration test bench. View the test bench signals in your simulator's waveform viewer to see the operations of the test bench.

Demonstration Test Bench in Detail

The demonstration test bench performs the following tasks:

- Instantiates the core
- Generates two input data tables containing ramps of different frequencies
- Generates a clock signal
- Drives the core's clock enable and reset input signals (if present)
- Drives the core's input signals to demonstrate core features
- Checks that the core output signals obey AXI protocol rules (data values are not checked in order to keep the test bench simple)
- Provides signals showing the separate fields of AXI TDATA and TUSER signals

The demonstration test bench drives the core's input signals to demonstrate the features and modes of operation of the core. The Divider Generator core is driven with two simple data ramps of different periods to stimulate the core with a wide range of positive and negative values, including zero. The input data is pre-generated and stored in data tables, and the test bench drives the core data inputs with the ramp data throughout the operation of the test bench.

The demonstration test bench drives the AXI handshaking signals in different ways, split into three phases. The operations depend on whether Blocking Mode or NonBlocking Mode is selected:

- Blocking Mode:
  - Phase 1: full throughput, all TVALID and TREADY signals are tied high
  - Phase 2: apply increasing amounts of back pressure by deasserting the master channel's TREADY signal
  - Phase 3: deprive slave dividend channel of valid transactions at an increasing rate by deasserting its TVALID signal
- NonBlocking Mode:
  - Phase 1: full throughput, all TVALID and TREADY signals are tied high
  - Phase 2: deprive slave dividend channel of valid transactions at an increasing rate by deasserting its TVALID signal
  - Phase 3: deprive all slave channels of valid transactions at different rates by deasserting each of their TVALID signals

Customizing the Demonstration Test Bench

It is possible to modify the demonstration test bench to drive the core's inputs with different data or to perform different operations.

Input data is pre-generated in the `create_ip_dividend_table` and `create_ip_divisor_table` functions and stored in the `IP_dividend_DATA` and `IP_divisor_DATA` constants. New input data frames can be added by defining new functions and constants. Make sure that each input data frame is of an appropriate type, similar to the `T_IP_dividend_TABLE` and `T_IP_divisor_TABLE` array types.

All operations performed by the demonstration test bench to drive the core's inputs are done in the `stimuli` process. This process is comprehensively commented, to explain clearly what is being done. New input data or different ways of driving AXI handshaking signals can be added by modifying sections of this process.

The total run time of the test can be modified by changing the `TEST_CYCLES` constant: this controls the number of clock cycles before the simulation is stopped.

The clock frequency of the core can be modified by changing the `CLOCK_PERIOD` constant.

AXI4-Stream Considerations

The conversion to AXI4-Stream interfaces brings standardization and enhances interoperability of Xilinx IP LogiCORE solutions. Other than general control signals such as `aclk`, `aclken` and `aresetn`, all inputs and outputs to the Divider Generator core are conveyed via AXI4-Stream channels. A channel consists of TVALID and TDATA always, plus several optional ports and fields. In the Divider Generator core, the optional ports supported are TREADY, TLAST and TUSER. Together, TVALID and TREADY perform a handshake to transfer a message, where the payload is TDATA, TUSER and TLAST. The Divider Generator core operates on the operands contained in the TDATA fields and outputs the result in the TDATA field of the output channel. The Divider Generator core does not use inputs, TUSER and TLAST as such, but the core provides the facility to convey these fields with the same latency as for TDATA. The Divider Generator core does use the output TUSER to hold the divide_by_zero indication signal. This facility of passing TLAST and TUSER from input to output is intended to ease use of the Divider Generator core in a system. For example, the Divider Generator core might operate on streaming packetized data. In this example, the core could be configured to pass the TLAST of the packetized data channel, thus saving the system designer the effort of constructing a bypass path for this information.

For further details on AXI4-Stream Interfaces see [Ref 4] and [Ref 5].

Basic Handshake

Figure 2 shows the transfer of data in an AXI4-Stream channel. TVALID is driven by the source (master) side of the channel and TREADY is driven by the receiver (slave). TVALID indicates that the value in the payload fields (TDATA, TUSER and TLAST) is valid. TREADY indicates that the slave is ready to receive data. When both TVALID and TREADY are true in a cycle, a transfer occurs. The master and slave set TVALID and TREADY respectively for the next transfer appropriately.

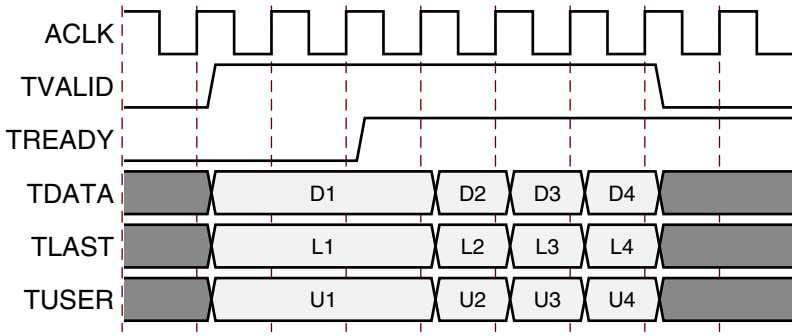

Figure 2: **Data Transfer in an AXI-Stream Channel**

Non Blocking Mode

The Divider Generator core provides a mode intended to ease the migration from previous, non-AXI versions of this core. The term 'Non-Blocking' is used to indicate that lack of data on one input channel does not cause incoming data on the other channel to be buffered. Also, back pressure from the output is not possible because in NonBlocking mode the output channel does not have a TREADY signal. The full flow control of AXI4-Stream is not always required. Blocking or Non-Blocking behavior is selected via the FlowControl parameter or GUI field. The choice of Blocking or NonBlocking applies to the whole core, not each channel individually. Channels still have the non-optional TVALID signal, which is analogous to the New Data (ND) signal on many cores prior to the adoption of AXI4-Stream. Without the facility to block dataflow, the internal implementation is much simplified, so fewer

resources are required for this mode. This mode is recommended for users wishing to move to this version from a pre-AXI version with minimal change.

When all of the present input channels receive an active TVALID (and TREADY, if present, is asserted), an operation is validated and the output TVALID (suitably delayed by the latency of the core) is asserted to qualify the result. This is to allow a minimal migration from v3.0. In the event that one channel receives TVALID and the other does not, then an operation does not occur, even if TREADY is present and asserted. Hence, unlike Blocking mode which is fully AXI4-Stream compliant, valid transactions on an individual channel can be ignored in NonBlocking mode.

For performance, `aresetn` is registered internally, which delays its action by a clock cycle. The effect is that the cycle following the deassertion of ARESETN the core is still reset and does not accept input. TVALID is also inactive on the output channel for this cycle.

Figure 3 shows the NonBlocking mode in operation. For simplicity of illustration, the latency of the core is zero. As indicated by `s_axis_dividend_tready` and `s_axis_divisor_tready`, which are ultimately the same signal, the core can accept data on every third cycle. Data A1 in the dividend channel is ignored because `s_axis_divisor_tvalid` is de-asserted. Data inputs A2 and B1 are accepted because both TVALIDs and TREADY are asserted.

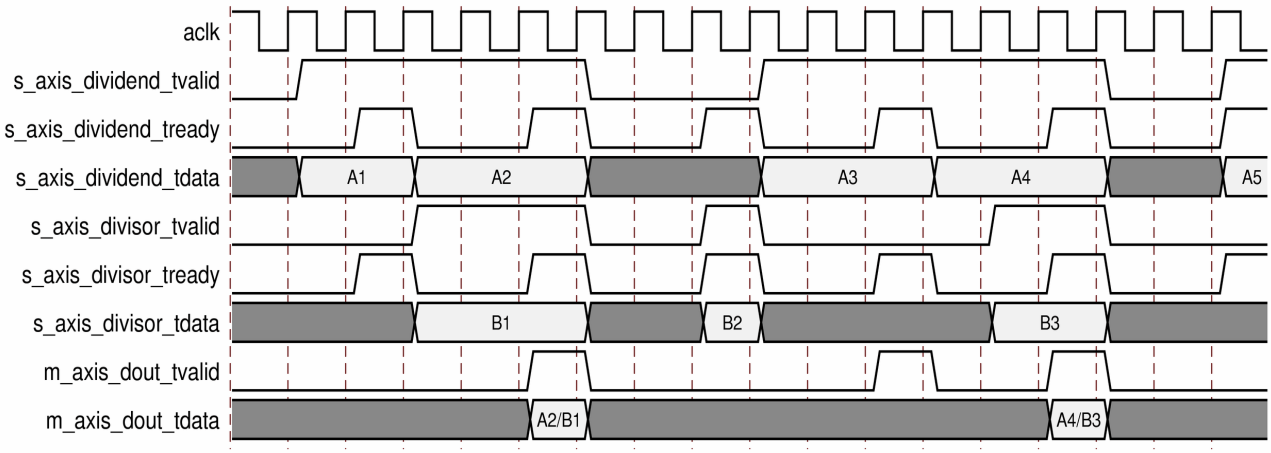

Figure 3: **Non Blocking Mode**

Blocking Mode

The term 'Blocking' means that each channel with TREADY buffers data for use. The full flow control of AXI4-Stream aids system design because the flow of data is self-regulating. Blocking or Non-Blocking behavior is selected via the FlowControl parameter GUI field. Data loss is prevented by the presence of back pressure (TREADY), so that data is only propagated when the downstream datapath is ready to process the data.

The Divider Generator core has two input channels and one output channel. When all input channels have validated data available, an operation occurs and the result becomes available on the output. If the output is prevented from off-loading data because `m_axis_dout_tready` is low then data accumulates in the output buffer internal to the core. When this output buffer is nearly full the core stops further operations. This prevents the input buffers from off-loading data for new operations so the input buffers fill as new data is input. When the input buffers fill, their respective TREADYs (`s_axis_divisor_tready` and `s_axis_dividend_tready`) are de-asserted to prevent further input. This is the normal action of back pressure.

The two input channels are tied in the sense that each must receive validated data before an operation can proceed. Therefore, there is an additional blocking mechanism, where one input channel does not receive validated data

while the other does. In this case, the validated data is stored in the input buffer of the channel. After a few cycles of this scenario, the buffer of the channel receiving data fills and TREADY for that channel is de-asserted until the starved channel receives some data. Figure 4 shows both blocking behavior and back pressure. The first data on channel S_AXIS_DIVIDEND is paired with the first data on channel S_AXIS_DIVISOR, the second with the second and so on. This demonstrates the 'blocking' concept. The channel names S_AXIS_DIVIDEND and S_AXIS_DIVISOR are used conceptually. Either can be taken to mean the divisor or dividend channel. Figure 4 further shows how data output is delayed not only by latency, but also by the handshake signal `m_axis_dout_tready`. This is 'back pressure'. Sustained back pressure on the output along with data availability on the inputs eventually leads to a saturation of the core's buffers, leading the core to signal that it can no longer accept further input by de-asserting the input channel TREADY signals. The minimum latency in this example is two cycles, but it should be noted that in Blocking operation latency is not a useful concept. Instead, as the diagram shows, the important idea is that each channel acts as a queue, ensuring that the first, second, third data samples on each channel are paired with the corresponding samples on the other channels for each operation.

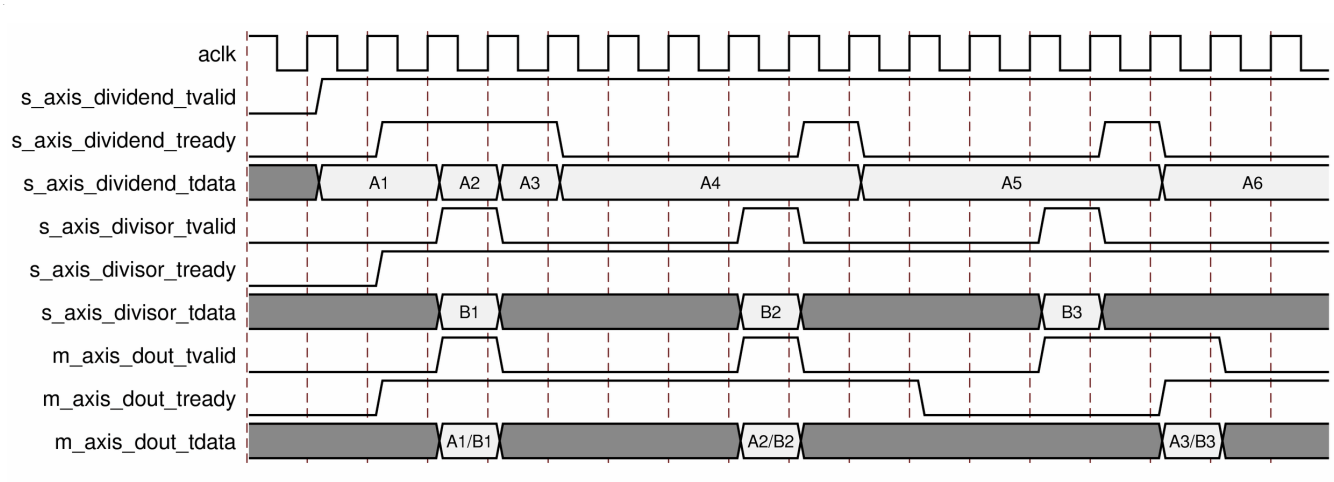

Figure 4: **Blocking Mode**

Note: This diagram is for illustration of the blocking behavior and handshake protocol. Latency of core is zero in diagram which in reality will not be the case.

TDATA Packing

Fields within an AXI4-Stream interface follow a specific naming nomenclature. In this core the operands are both passed to or from the core via the channel's TDATA port. To ease interoperability with byte-oriented protocols, each subfield within TDATA which could be used independently is first extended, if necessary, to fit a bit field which is a multiple of 8 bits. For the output DOUT channel, result fields are sign extended to the byte boundary. The bits added by byte orientation are ignored by the core and do not result in additional resource use.

TDATA Structure for Dividend and Divisor Channels

Input channels Dividend and Divisor carry their operands only in their TDATA field. For each, the operand occupies the least significant bits. The TDATA port width itself is the minimum multiple of bytes wide required to contain the operand. See Figure 5.

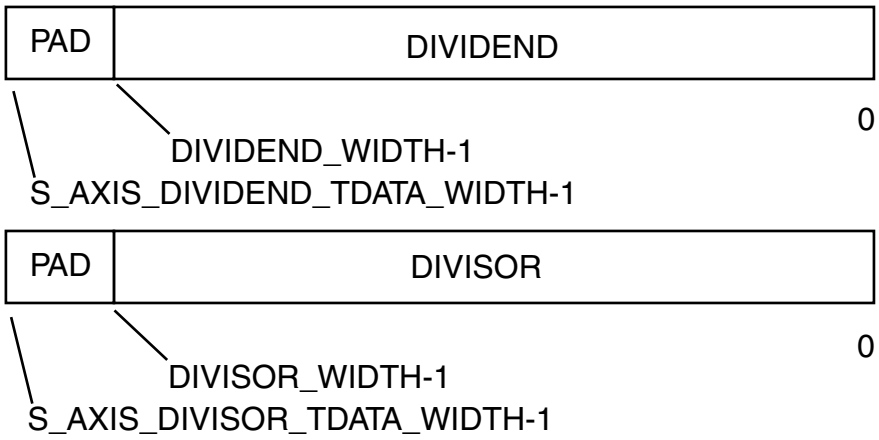

Figure 5: **Input Data TDATA Structure**

TDATA Structure for Output (DOUT) Channel

The structure of `m_axis_dout_tdata` is more complex. This port contains both quotient and, if present, remainder or fractional outputs. When the remainder type is set to remainder, the two outputs are considered separate and so are byte-oriented before being concatenated to make the `m_axis_dout_tdata` signal. When remainder type is fractional, the fractional part is considered an extension of the quotient so these two fields are concatenated before being padded to the next byte boundary.

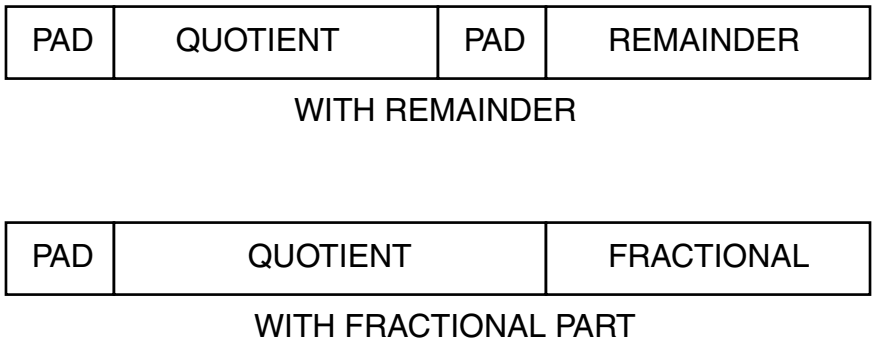

Figure 6: **Data Out TDATA Structure**

TLAST and TUSER Handling

TLAST in AXI4-Stream is used to denote the last transfer of a block of data. TUSER is for ancillary information which qualifies or augments the primary data in TDATA. The Divider Generator core operates on a per-sample basis where each operation is independent of any before or after. Because of this, there is no need for TLAST on a divider. The TLAST and TUSER signals are supported on each input channel purely as an optional aid to system design for the scenario in which the data stream being passed through the Divider Generator core does indeed have some packetization or ancillary field, but which is not relevant to the divider. The facility to pass TLAST and/or TUSER removes the burden of matching latency to the TDATA path, which can be variable, through the divider.

When Divide_by_zero detect is selected, the signal indicating a division by zero is output on the least significant bit of the output channel TUSER port.

TLAST Options

TLAST for each input channel is optional. Each, when present, can be passed via the divider, or, when more than one channel has TLAST enabled, can pass a logical AND or logical OR of the TLASTs input. When no TLASTs are present on any input channel, the output channel does not have TLAST either.

TUSER Options

TUSER for each input channel is optional. Each has user-selectable width. The Divider Generator core might also generate a TUSER bit. This is when divide_by_zero detection is selected. These fields are concatenated, without any byte-orientation or padding, to form the output channel TUSER field. The divide_by_zero bit occupies the least significant position, followed by the TUSER field from the Divisor channel then TUSER from the Dividend channel in the most significant position.

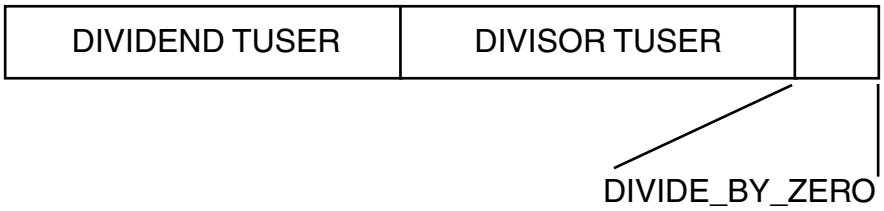

Figure 7: **Data Out TUSER Structure**

Migrating to Divider Generator v4.0 from Earlier Versions

XCO Parameter Changes

The CORE Generator core update functionality can be used to update an existing XCO file from v3.0 to Divider Generator v4.0, but it should be noted that the update mechanism alone does not create a core compatible with v3.0. See Instructions for Minimum Change Migration (v3.0 to v4.0).

Table 6 shows the changes to XCO parameters from version 3.0 to version 4.0.

Table 6: **XCO Parameter Changes from v3.0 to v4.0**

Version 3.0	Version 4.0	Notes
Component_Name	Component_Name	Unchanged
dividend_and_quotient_width	dividend_and_quotient_width	Unchanged
divisor_width	divisor_width	Unchanged
remainder_type	remainder_type	Unchanged
fractional_width	fractional_width	Unchanged
operand_sign	operand_sign	Unchanged
clocks_per_division	clocks_per_division	Unchanged
divide_by_zero_detect	divide_by_zero_detect	Unchanged
latency_configuration	latency_configuration	Unchanged
latency	latency	Unchanged in function. Unchanged for AXI4-Stream NonBlocking mode. Because AXI4-Stream Blocking modes add latency, retaining the same value reduces pipelining in the core and can reduce performance
ce	ACLKEN	Renamed only

Table 6: **XCO Parameter Changes from v3.0 to v4.0 *(Cont'd)***

Version 3.0	Version 4.0	Notes
sclr	ARESETn	Renamed only. While the sense of the aresetn signal has changed, this XCO determined whether or not the signal exists and has not changed. Note also that a minimum length of 2 cycles is recommended when aresetn is asserted.
SclrCEPriority		Deprecated. aresetn always overrides aclken in accordance with AXI4-Stream protocol.
	dividend_has_tlast	Introduced in version 4.0
	dividend_has_tuser	Introduced in version 4.0
	dividend_tuser_width	Introduced in version 4.0
	divisor_has_tlast	Introduced in version 4.0
	divisor_has_tuser	Introduced in version 4.0
	divisor_tuser_width	Introduced in version 4.0
	FlowControl	Introduced in version 4.0
	OptimizeGoal	Introduced in version 4.0
	OutTready	Introduced in version 4.0
	OutTLASTBehv	Introduced in version 4.0

For more information on this upgrade feature, see the CORE Generator software documentation.

Port Changes

Table 7 details the changes to port naming, additional or deprecated ports and polarity changes from v3.0 to v4.0.

Table 7: **Port Changes from Version 3.0 to Version 4.0**

Version 3.0	Version 4.0	Notes
CLK	aclk	Rename only
CE	aclken	Rename only
SCLR	aresetn	Rename and change of sense (now active low). Note recommendation that aresetn should be asserted for a minimum of 2 cycles.
DIVIDEND	s_axis_dividend_tdata(N-1:0)	
DIVISOR	s_axis_divisor_tdata(M-1:0)	
QUOTIENT	m_axis_dout_tdata(S-1:0)	Both Quotient and Fractional (or remainder) map to m_axis_dout_tdata. See TDATA Structure for Output (DOUT) Channel for details.
FRACTIONAL		
ND		Deprecated. However this is analogous to the TVALID signals. See Instructions for Minimum Change Migration (v3.0 to v4.0).
RDY		Deprecated. However, this is analogous to TVALID on the output channel. See Instructions for Minimum Change Migration (v3.0 to v4.0).
RFD		Deprecated. However, this is analogous to TREADY on the input channels. See Instructions for Minimum Change Migration (v3.0 to v4.0)
DIVIDE_BY_ZERO	m_axis_dout_tuser(0)	When this signal is selected to appear, it occupies the LSB of the output TUSER port. See TUSER Options for details.
	s_axis_dividend_tvalid	TVALID (AXI4-Stream channel handshake signal) for each channel
	s_axis_divisor_tvalid	
	m_axis_dout_tvalid	

Table 7: **Port Changes from Version 3.0 to Version 4.0 *(Cont'd)***

Version 3.0	Version 4.0	Notes
	s_axis_dividend_tready	TREADY (AXI4-Stream channel handshake signal) for each channel.
	s_axis_divisor_tready	
	m_axis_dout_tready	
	s_axis_dividend_tlast	TLAST (AXI4-Stream packet signal indicating the last transfer of a data structure) for each channel. The Divider Generator core does not use TLAST, but provides the facility to pass TLAST with the same latency as TDATA.
	s_axis_divisor_tlast	
	m_axis_dout_tlast	
	s_axis_dividend_tluser	TUSER (AXI4-Stream ancillary field for application-specific information) for each channel. The Divider Generator core does not use TUSER, but provides the facility to pass TUSER with the same latency as TDATA.
	s_axis_divisor_tuser	
	m_axis_dout_tuser	

Latency Changes

With the addition of AXI4-Stream interfaces, the latency of the Divider Generator core v4.0 is different compared to v3.0 for AXI Blocking mode. Latency is the same as v3.0 in v4.0 for AXI Non-Blocking mode.

Importantly, when in Blocking Mode, the latency of the core is variable due to the FIFO nature of the AXI4-Stream protocol, so only the minimum possible latency can be determined. Relative to v3.0, with Blocking and Output TREADY present, minimum latency is 3 cycles greater. With no output TREADY, minimum latency is increased by one cycle only.

Instructions for Minimum Change Migration (v3.0 to v4.0)

To configure the Divider Generator core v4.0 to most closely mimic the behavior of v3.0 the translation is as follows:

Parameters

- Set FlowControl to NonBlocking.

All other new parameters default to false and can be ignored.

Ports

- Rename and map signals as detailed in Port Changes.
- Map ND to both `s_axis_dividend_tvalid` and `s_axis_divisor_tvalid`.
- Map RFD to `s_axis_dividend_tready` (`s_axis_divisor_tready` can be used equally).
- Map RDY to `m_axis_dout_tvalid`.

Performance and resource use is mostly unchanged compared with Divider Generator v3.0 other than small changes due to the use of a different version of ISE tools.

Performance and Resource Utilization

Tables 8 to 15 provide performance and resource usage information for a number of different Divider Generator core configurations.

The maximum clock frequency results were obtained by double-registering input and output ports to reduce dependence on I/O placement. The inner level of registers used a separate clock signal to measure the path from the input registers to the first output register through the core.

The resource usage results do not include the aforementioned wrapping registers and hence represent the true logic used by the core to implement a single Divider. LUT counts include SRL32s and LUTs used as route-throughs.

The map options used were: `"map -ol high"`

The par options used were: `"par -ol high"`

Clock frequency does not take clock jitter into account and should be derated by an amount appropriate to the clock source jitter specification. Performance figures are achieved using default arguments to placement and route tools (other than high effort), so as to obtain realistic rather than best-case results.

Radix 2 Performance tables

For all Radix 2 cases the Operand Sign is unsigned and ACLKEN and ARESETN are disabled.

Table 8 defines performance characteristics for Radix-2 cases on Virtex®-7 FPGA, speed grade -1 using speedfile "ADVANCED 1.01h 2011-03-07"

Table 8: **Radix-2 Solution Performance Characteristics on Virtex-7 FPGA (Part = xc7v285t)**

Parameter/Result	Case 1	Case 2	Case 3	Case 4	Case 5	Case 6	Case 7	Case 8	Case9
Dividend and Quotient Width	8	8	8	8	8	8	8	32	64
Divisor Width	8	8	8	8	8	8	8	32	64
Remainder Type	remd	rem	rem	rem	rem	rem	rem	rem	frac
Fractional Width	8	8	8	8	8	8	8	32	64
Clocks per Division	1	1	1	1	2	2	8	1	1
Flow Control (NonBlocking/ Blocking)	NonBlock	Blocking	Blocking	Blocking	NonBlock	NonBlock	NonBlock	NonBlock	NonBlock
OutTready	no	yes	yes	no	no	no	no	no	no
Optimize Goal(Speed/Area)	either	Speed	Area	either	either	either	either	either	either
Input TUSER widths	0/0	0/0	0/0	0/0	0/0	8/8	0/0	0/0	0/0
LUT6-FF Pairs	170	216	223	187	161	194	87	2196	16473
LUTs	153	203	197	165	114	136	46	2068	16286
FFs	226	287	288	247	161	195	91	3202	26723
Block RAMs	0	0	0	0	0	0	0	0	0
DSP48 Blocks	0	0	0	0	0	0	0	0	0
Max Clock Freq(1)(2)	497	497	491	497	389	395	497	375	247

Notes:
1. Area and maximum clock frequencies are provided as a guide and might vary with new releases of the Xilinx implementation tools.
2. Maximum clock frequencies are shown in MHz. Clock frequency does not take jitter into account and should be de-rated by an amount appropriate to the clock source jitter specification.
3. Case 9 is approximately 8 times larger than case 8 due to 3 doubling factors: dividend width, divisor width and fractional output rather than remainder output.

Table 9 defines performance characteristics for Radix-2 cases on Kintex™-7 FPGA, speed grade -1, using speedfile "ADVANCED 1.01f 2011-03-23"

Table 9: **Radix-2 Solution Performance Characteristics on Kintex-7 FPGA (Part = xc7k325t)**

Parameter/Result	Case 1	Case 2	Case 3	Case 4	Case 5	Case 6	Case 7	Case 8
Dividend and Quotient Width	8	8	8	8	8	8	8	32
Divisor Width	8	8	8	8	8	8	8	32
Remainder Type	remd	rem	rem	rem	rem	rem	rem	rem
Fractional Width	8	8	8	8	8	8	8	32
Clocks per Division	1	1	1	1	2	2	8	1
Flow Control (NonBlocking/Blocking)	NonBlock	Blocking	Blocking	Blocking	NonBlock	NonBlock	NonBlock	NonBlock
OutTready	no	yes	yes	no	no	no	no	no
Optimize Goal(Speed/Area)	either	Speed	Area	either	either	either	either	either
LUT6-FF Pairs	172	218	216	187	161	188	81	2209
LUTs	155	203	205	155	119	135	49	2060
FFs	226	287	288	247	161	195	91	3202
Block RAMs	0	0	0	0	0	0	0	0
DSP48 Blocks	0	0	0	0	0	0	0	0
Max Clock Freq(1)(2)	>452	>452	>452	>452	410	438	>452	366

Notes:

1. Area and maximum clock frequencies are provided as a guide and might vary with new releases of the Xilinx implementation tools.
2. Maximum clock frequencies are shown in MHz. Clock frequency does not take jitter into account and should be de-rated by an amount appropriate to the clock source jitter specification.

Table 10 defines performance characteristics for Radix-2 cases on Virtex-6 FPGA, speed grade -1 using speedfile "PRODUCTION 1.14a 2011-03-07".

Table 10: **Radix-2 Solution Performance Characteristics on Virtex-6 FPGA (Part = XC6VLX75T-1)**

Parameter/Result	Case 1	Case 2	Case 3	Case 4	Case 5	Case 6	Case 7	Case 8
Dividend and Quotient Width	8	8	8	8	8	8	8	32
Divisor Width	8	8	8	8	8	8	8	32
Remainder Type	remd	rem	rem	rem	rem	rem	rem	rem
Fractional Width	8	8	8	8	8	8	8	32
Clocks per Division	1	1	1	1	2	2	8	1
Flow Control (NonBlocking/Blocking)	NonBlock	Blocking	Blocking	Blocking	NonBlock	NonBlock	NonBlock	NonBlock
OutTready	no	yes	yes	no	no	no	no	no
Optimize Goal(Speed/Area)	either	Speed	Area	either	either	either	either	either
LUT6-FF Pairs	168	216	217	184	148	185	81	2195
LUTs	150	202	203	152	124	139	52	2126
FFs	226	287	288	247	161	195	91	3202
Block RAMs	0	0	0	0	0	0	0	0
DSP48 Blocks	0	0	0	0	0	0	0	0
Max Clock Freq(1)(2)	>491	>491	>491	>491	422	429	460	361

Notes:

1. Area and maximum clock frequencies are provided as a guide and might vary with new releases of the Xilinx implementation tools.
2. Maximum clock frequencies are shown in MHz. Clock frequency does not take jitter into account and should be de-rated by an amount appropriate to the clock source jitter specification.

Table 11 defines performance characteristics for Radix-2 cases on Spartan®-6 FPGA, speed grade 2, using speedfile "PRODUCTION 1.18a 2011-03-07"

Table 11: **Radix-2 Solution Performance Characteristics on Spartan-6 FPGA (Part = XC6SLX16-2)**

Parameter/Result	Case 1	Case 2	Case 3	Case 4	Case 5	Case 6	Case 7	Case 8
Dividend and Quotient Width	8	8	8	8	8	8	8	32
Divisor Width	8	8	8	8	8	8	8	32
Remainder Type	remd	rem	rem	rem	rem	rem	rem	rem
Fractional Width	8	8	8	8	8	8	8	32
Clocks per Division	1	1	1	1	2	2	8	1
Flow Control (NonBlocking/Blocking)	NonBlock	Blocking	Blocking	Blocking	NonBlock	NonBlock	NonBlock	NonBlock
OutTready	no	yes	yes	no	no	no	no	no
Optimize Goal(Speed/Area)	either	Speed	Area	either	either	either	either	either
LUT6-FF Pairs	163	211	215	191	156	171	77	2185
LUTs	149	197	189	149	118	132	56	2130
FFs	226	287	288	247	161	195	91	3202
Block RAMs	0	0	0	0	0	0	0	0
DSP48 Blocks	0	0	0	0	0	0	0	0
Max Clock Freq(1)(2)	>334	313	329	>334	277	267	319	236

Notes:

1. Area and maximum clock frequencies are provided as a guide and might vary with new releases of the Xilinx implementation tools.
2. Maximum clock frequencies are shown in MHz. Clock frequency does not take jitter into account and should be de-rated by an amount appropriate to the clock source jitter specification.

High Radix Performance Tables

Optional control signals `aclken` and `aresetn` are disabled. The High Radix treats inputs and outputs as signed numbers.

Note: When the core does not have aresetn, use can be made of SRL16 primitives, leading to a substantial reduction in circuit size. For this reason, the use of aresetn is not recommended.

Table 12 defines performance characteristics for cases run on a Virtex-7 FPGA, speed grade -1 using speedfile "ADVANCED 1.01h 2011-03-07".

Table 12: **High Radix Solution Performance Characteristics on Virtex-7 FPGA (Part = xc7v285t)**

Parameter/Result	Case 1	Case 2	Case 3	Case 4	Case 5	Case 6	Case 7	Case 8
Dividend and Quotient Width	10	10	36	36	54	54	37	64
Divisor Width	14	14	36	36	50	50	24	64
Remainder Type	frac	frac	frac	frac	frac	frac	frac	frac
Fractional Width	2	2	2	2	28	28	0	2
Latency Configuration (latency)	Auto (17)	2	Auto (28)	4	Auto (43)	8	Auto (26)	Auto(40)
LUT-FF Pairs	290	208	793	537	1156	798	603	1349
LUTs	263	206	748	506	1114	774	532	1303
FFs	392	58	1062	184	1594	261	795	1837
RAMB18E1	1	1	1	1	1	1	1	1
DSP48E1s	7	7	13	13	16	16	11	19
Max Clock Freq(1)(2)	395	79	395	59	355	59	263	323

Notes:

1. Resources and maximum clock frequencies are provided as a guide and might vary with new releases of the Xilinx implementation tools.
2. Maximum clock frequencies are shown in MHz. Clock frequency does not take jitter into account and should be de-rated by an amount appropriate to the clock source jitter specification.

Table 13 defines performance characteristics for cases run on a Kintex-7 FPGA, speed grade -1, using speedfile "ADVANCED 1.01f 2011-03-23".

Table 13: **High Radix Solution Performance Characteristics on Kintex-7 FPGA (Part = xc7k325t)**

Parameter/Result	Case 1	Case 2	Case 3	Case 4	Case 5	Case 6	Case 7	Case 8
Dividend and Quotient Width	10	10	36	36	54	54	37	64
Divisor Width	14	14	36	36	50	50	24	64
Remainder Type	frac	frac	frac	frac	frac	frac	frac	frac
Fractional Width	2	2	2	2	28	28	0	2
Latency Configuration (latency)	Auto (17)	2	Auto (28)	4	Auto (43)	8	Auto (26)	Auto(40)
LUT6-FF Pairs	290	209	797	536	1162	807	603	1354
LUTs	265	206	744	506	1110	769	532	1297
FFs	392	58	1062	184	1594	261	795	1837
RAMB16BWERs	1	1	1	1	1	1	1	1
DSP48A1s	7	7	13	13	16	16	11	19
Max Clock Freq(1)(2)	395	76	395	61	350	61	261	358

Notes:

1. Resources and maximum clock frequencies are provided as a guide and might vary with new releases of the Xilinx implementation tools.
2. Maximum clock frequencies are shown in MHz. Clock frequency does not take jitter into account and should be de-rated by an amount appropriate to the clock source jitter specification.

Table 14 defines performance characteristics for cases run on a Virtex-6 FPGA, speed grade -1 using speedfile "PRODUCTION 1.14a 2011-03-07".

Table 14: **High Radix Solution Performance Characteristics on Virtex-6 FPGA (Part = XC6VLX75T-1)**

Parameter/Result	Case 1	Case 2	Case 3	Case 4	Case 5	Case 6	Case 7	Case8
Dividend and Quotient Width	10	10	36	36	54	54	37	64
Divisor Width	14	14	36	36	50	50	24	64
Remainder Type	Frac	Frac	Frac	Frac	Frac	Frac	Frac	frac
Fractional Width	2	2	2	2	28	28	0	2
Latency Configuration (latency)	Auto (17)	2	Auto (28)	4	Auto (43)	8	Auto (26)	Auto(40)
LUT-FF Pairs	286	208	790	540	1162	807	594	1335
LUTs	271	207	745	502	1103	757	541	1305
FFs	392	58	1062	184	1594	261	795	1838
RAMB18E1	1	1	1	1	1	1	1	1
DSP48E1s	7	7	13	13	16	16	11	19
Max Clock Freq(1)(2)	399	78	399	62	399	62	262	377

Notes:

1. Resources and maximum clock frequencies are provided as a guide and might vary with new releases of the Xilinx implementation tools.
2. Maximum clock frequencies are shown in MHz. Clock frequency does not take jitter into account and should be de-rated by an amount appropriate to the clock source jitter specification.

Table 15 defines performance characteristics for cases run on a Spartan-6 FPGA, speed grade 2, using speedfile "PRODUCTION 1.18a 2011-03-07".

Table 15: **High Radix Solution Performance Characteristics on Spartan-6 FPGA (Part = XC6SLX16-2)**

Parameter/Result	Case 1	Case 2	Case 3	Case 4	Case 5	Case 6	Case 7	Case 8
Dividend and Quotient Width	10	10	36	36	54	54	37	64
Divisor Width	14	14	36	36	50	50	24	64
Remainder Type	Frac	Frac	Frac	Frac	Frac	Frac	Frac	Frac
Fractional Width	2	2	2	2	28	28	0	2
Latency Configuration (latency)	Auto (17)	2	Auto (30)	4	Auto (45)	8	Auto (27)	Auto(40)
LUT6-FF Pairs	279	189	798	528	1191	794	558	1371
LUTs	256	175	730	504	1075	749	510	1280
FFs	421	70	1138	184	1719	261	809	1997
RAMB16BWERs	1	1	1	1	1	1	1	1
DSP48A1s	7	7	15	15	18	18	11	22
Max Clock Freq(1)(2)	257	61	195	30	175	30	169	112

Notes:

1. Resources and maximum clock frequencies are provided as a guide and might vary with new releases of the Xilinx implementation tools.
2. Maximum clock frequencies are shown in MHz. Clock frequency does not take jitter into account and should be de-rated by an amount appropriate to the clock source jitter specification.

References

1. "*Computer Arithmetic Algorithms and Hardware Designs*," Behrooz Parhami. Oxford Press © 2000.
2. "*Proceedings 12th Symposium on Computer Arithmetic*," IEEE Computer Society Press © 1995.
3. Synthesis and Simulation Design Guide
4. *Xilinx AXI Design Reference Guide* (UG761)
5. AMBA 4 AXI4-Stream Protocol Version: 1.0 Specification

Support

Xilinx provides technical support for this LogiCORE IP product when used as described in the product documentation. Xilinx cannot guarantee timing, functionality, or support of product if implemented in devices that are not defined in the documentation, if customized beyond that allowed in the product documentation, or if changes are made to any section of the design labeled *DO NOT MODIFY*.

See the IP Release Notes Guide (XTP025) for further information on this core.

For each core, there is a master Answer Record that contains the Release Notes and Known Issues list for the core being used. The following information is listed for each version of the core:

- New Features
- Bug Fixes
- Known Issues

Ordering Information

This LogiCORE IP module is included at no additional cost with the Xilinx ISE Design Suite software and is provided under the terms of the Xilinx End User License Agreement. Use the CORE Generator software included with the ISE Design Suite to generate the core. For more information, visit the core page.

Information about additional Xilinx LogiCORE IP modules is available at the Xilinx IP Center. For pricing and availability of other Xilinx LogiCORE IP modules and software, contact your local Xilinx sales representative.

Revision History

The following table shows the revision history for this document:

Date	Version	Description of Revisions
06/22/11	1.0	Initial Xilinx release. Previous data sheet was DS530.

Notice of Disclaimer