LogiCORE IP Divider Generator v5.0

Product Guide for Vivado Design Suite

PG151 March 20, 2013

XILINX®

XILINX®

Table of Contents

Appendix C: Additional Resources

Introduction

The Xilinx LogiCORE IP Divider Generator core 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 DSP 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 DSP slices.
- Optional operand widths, synchronous controls, and selectable latency.

LogiCORE IP Facts Table	
Core Specifics	
Supported Device Family(1)	Zynq™-7000, Virtex®-7, Kintex™-7, Artix™-7
Supported User Interfaces	AXI4-Stream
Resources	See Table 2-4 through Table 2-7.
Provided with Core	
Design Files	Encrypted RTL
Example Design	Not Provided
Test Bench	VHDL
Constraints File	Not Provided
Simulation Model	VHDL Behavioral Verilog and VHDL
Supported S/W Driver	N/A
Tested Design Flows(2)	
Design Entry	Vivado Design Suite System Generator for DSP
Simulation	Mentor Graphics Questa SIM Vivado Simulator
Synthesis	Vivado Synthesis
Support	
Provided by Xilinx @ www.xilinx.com/support	

Notes:

1. For a complete list of supported devices, see Vivado IP catalog.
2. For the supported versions of the tools, see the Xilinx Design Tools: Release Notes Guide.

Overview

Two implementations of division are supported by Divider Generator:

- **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 DSP slices.

A detailed explanation of each implementation is provided in Radix-2 Options in Chapter 4 and High Radix Options in Chapter 4.

Feature Summary

Radix-2 Solution

- 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

High Radix Solution

- 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 DSP Slices

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 DSP slices makes this the preferred option for larger operand widths.

Licensing and Ordering Information

This Xilinx LogiCORE IP module is provided at no additional cost with the Xilinx Vivado Design Suite under the terms of the Xilinx End User License. Information about this and other Xilinx LogiCORE IP modules is available at the Xilinx Intellectual Property page. For information about pricing and availability of other Xilinx LogiCORE IP modules and tools, contact your local Xilinx sales representative.

Product Specification

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 DSP slice 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 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 2-1). In the fractional case, the result is an M-bit-wide quotient with an F-bit-wide fractional remainder (Equation 2-2). 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 2-3).

Integer remainder case:

$$Dividend = Quotient \times Divisor + IntRmd \quad \text{Equation 2-1}$$

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

$$FractRmd = \frac{IntRmd \times 2^{F}}{Divisor} \quad \text{Equation 2-2}$$

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

$$FractRmd = \frac{IntRmd \times 2^{F-1}}{Divisor} \quad \text{Equation 2-3}$$

For signed mode with integer remainder, the sign of the quotient and remainder correspond exactly to Equation 2-1.

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 in Chapter 3) outputs until new results appear.

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 DSP (multiply-add) slices, providing an efficient, low-latency iteration.

Performance

This section details the performance information for various core configurations.

Maximum Frequencies

Maximum frequencies are listed in Table 2-4 through Table 2-7 with the resources used for the configurations measured.

Latency

Radix-2

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 2-1 provides a list of the latency formula for divider selections.

Table 2-1: **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

Table 2-1: **Latency of Radix-2 Solution Based on Divider Parameters *(Cont'd)***

Signed	Fractional	Clocks Per Division	Latency[1]
True	True	>1	M+F+A+5

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

High Radix Solution

Tables 2-2 and 2-3 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 2-2: **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 2-3: **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

Throughput

Radix-2 Solution

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

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.

Resource Utilization

Table 2-4 through Table 2-7 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 2-4 defines performance characteristics for Radix-2 cases on Virtex®-7 FPGA, speed grade -1 using speedfile "ADVANCED 1.01h 2011-03-07".

Note: These benchmarks were obtained using ISE Design Suite. The performance is expected to be similar when using Vivado Design Suite.

Table 2-4: **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 2-5 defines performance characteristics for Radix-2 cases on Kintex™-7 FPGA, speed grade -1, using speedfile "ADVANCED 1.01f 2011-03-23".

Note: These benchmarks were obtained using ISE Design Suite. The performance is expected to be similar when using Vivado Design Suite.

Table 2-5: **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.

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 2-6 defines performance characteristics for cases run on a Virtex-7 FPGA, speed grade -1 using speedfile "ADVANCED 1.01h 2011-03-07".

Note: These benchmarks were obtained using ISE Design Suite. The performance is expected to be similar when using Vivado Design Suite.

Table 2-6: **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 2-7 defines performance characteristics for cases run on a Kintex-7 FPGA, speed grade -1, using speedfile "ADVANCED 1.01f 2011-03-23".

Note: These benchmarks were obtained using ISE Design Suite. The performance is expected to be similar when using Vivado Design Suite.

Table 2-7: **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

Table 2-7: **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
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.

Port Descriptions

The core pinout and signal names are shown in Figure 2-1 and defined in Table 2-8

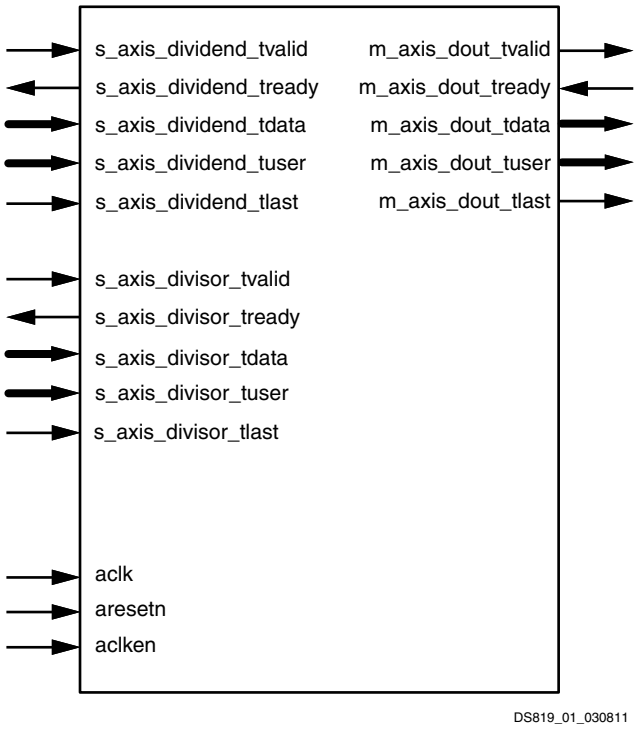


Figure 2-1: **Core Pinout Diagram**

Table 2-8: **Signal Pinout**

Signal	Direction[(1)]	Optional	Description
aclk	Input	No	Rising edge clock.
aclken	Input	Yes	Active high clock enable.

Table 2-8: **Signal Pinout *(Cont'd)***

Signal	Direction[(1)]	Optional	Description
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 in Chapter 3 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 in Chapter 3 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 in Chapter 3 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.
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 in Chapter 3 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.

Designing with the Core

This chapter includes guidelines and additional information to facilitate designing with the core.

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 3-1 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.

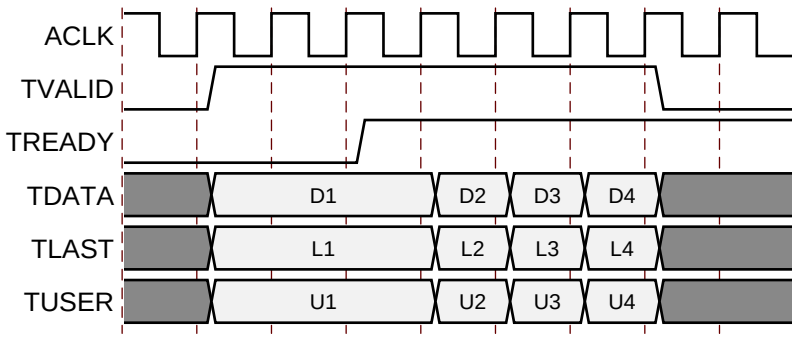


Figure 3-1: **Data Transfer in an AXI-Stream Channel**

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 3-2 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.

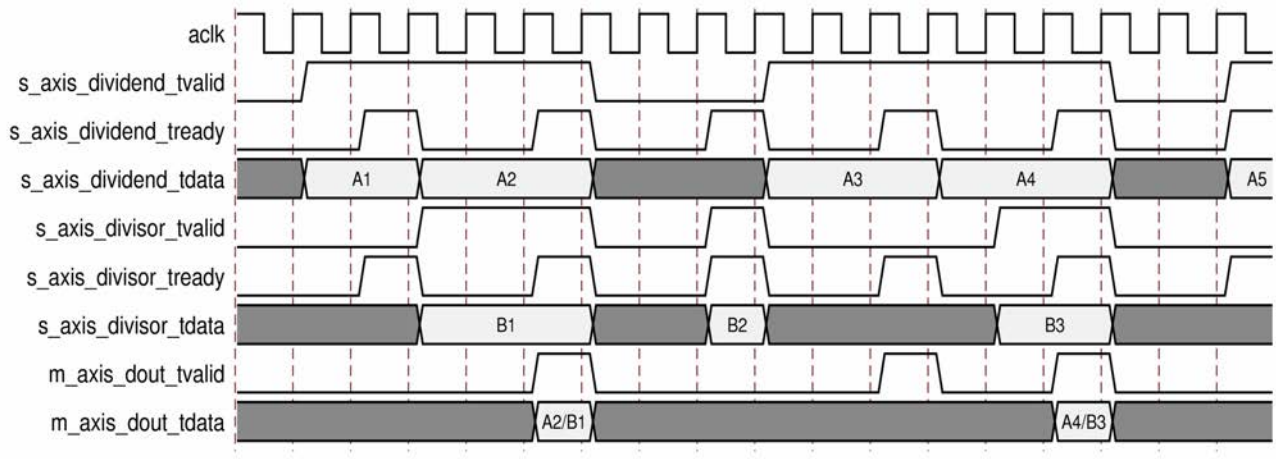


Figure 3-2: **Non Blocking Mode**

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 3-3 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 3-3 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.

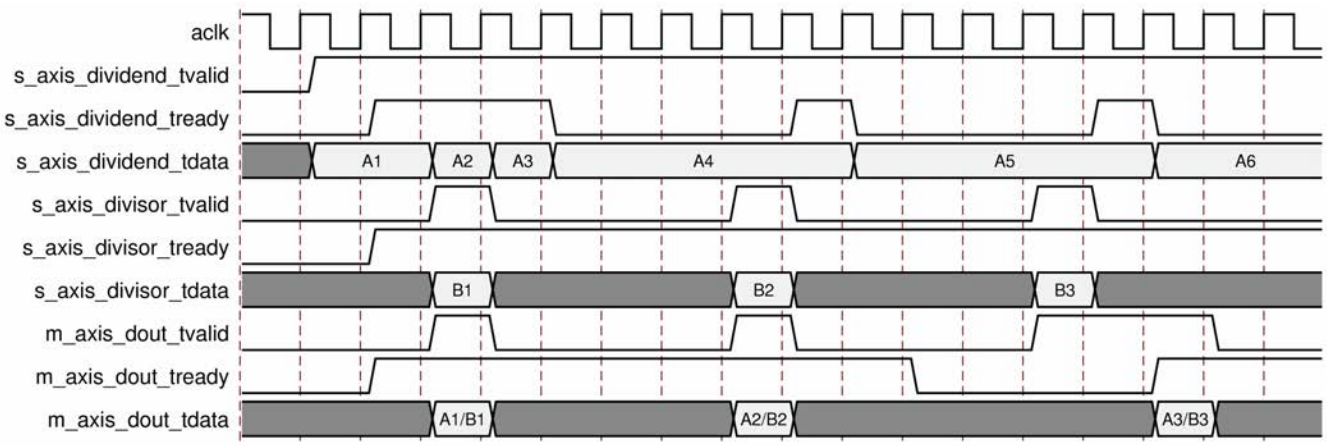


Figure 3-3: **Blocking Mode**

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 3-4.

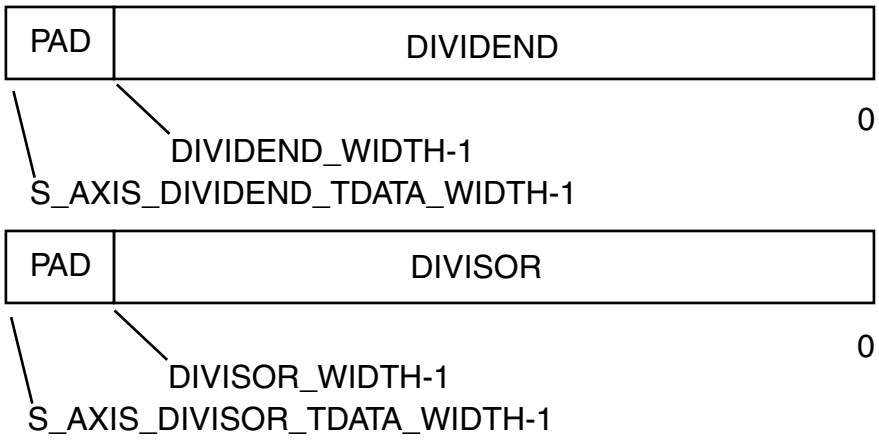


Figure 3-4: **Input Data TDATA Structure**

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.

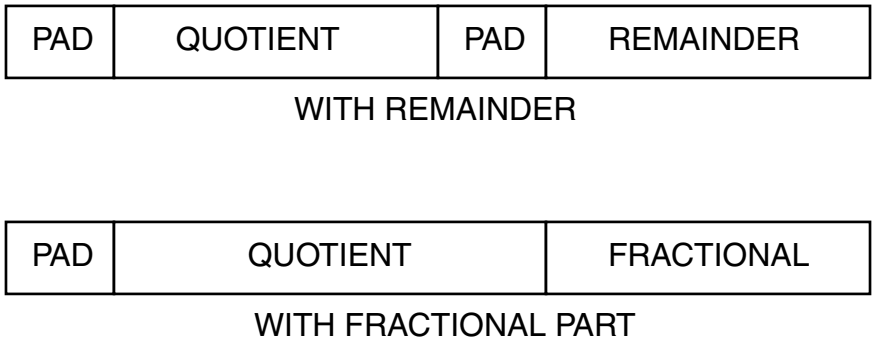


Figure 3-5: **Data Out TDATA Structure**

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.

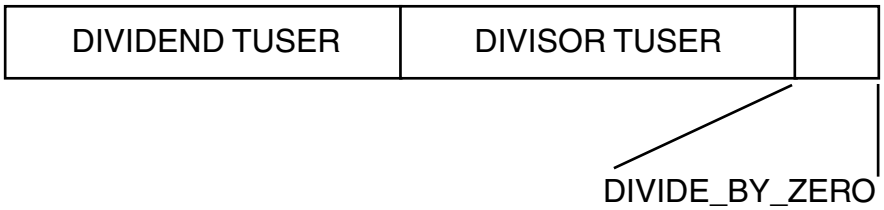


Figure 3-6: **Data Out TUSER Structure**

Clocking

This core has only one clock and there are no special considerations nor clock domain crossing considerations.

Resets

The core may be reset using the ARESETn pin. This is a global synchronous, active-Low reset that must be asserted for at least two ACLK cycles. All control circuitry is returned to the power-on state. Data registers may or may not be reset, but outputs are qualified by

TVALID. Any residue output in the ACLK cycles following reset and before TVALID is asserted can be ignored.

Protocol Description

This core adheres to the AXI4-Stream specification. For more details, see AXI4-Stream Considerations.

Customizing and Generating the Core

This chapter includes information about using Xilinx tools to customize and generate the core in the Vivado™ Design Suite environment.

GUI

The Divider Generator core can be found in the Xilinx Blockset in the Math section. The block is called "Divider Generator 5.0." 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.

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 in Chapter 3 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 in Chapter 3. 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 Resource Utilization in Chapter 2.
- **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 TREADY, 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.

Constraining the Core

There are no specific constraints for this core.

Detailed Example Design

When the core is generated using the Vivado™ Design Suite, 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 Vivado output directory. The source code is comprehensively commented.

Using the Demonstration Test Bench

The demonstration test bench instantiates the generated Divider Generator core.

Compile the netlist and the demonstration test bench into the work library (see your simulator documentation for more information on how to do this). 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 input data
- Generates a clock signal
- Drives the input signals of the core to demonstrate core features
- Checks that the output signals of he core obey AXI4-Stream protocol rules (data values are not checked to keep the test bench simple)
- Provides signals showing the separate fields of AXI4-Stream TDATA and TUSER signals

The demonstration test bench drives the core input signals to demonstrate the features and modes of operation of the core. The operations performed by the demonstration test bench are appropriate for the configuration of the generated core, and are a subset of the following operations:

1. An initial phase where the core is initialized and no operations are performed.

2. Perform a single operation, and wait for the result.
3. Perform 100 consecutive operations with incrementing data.
4. Perform operations while demonstrating the AXI4-Stream control signals' use and effects.
5. If ACLKEN is present: Demonstrate the effect of toggling `aclken`.
6. If ARESETn is present: Demonstrate the effect of asserting `aresetn`.
7. Demonstrate the handling of special floating-point values (NaN, zero, infinity).

Customizing the Demonstration Test Bench

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

For instructions on simulating your core, see the *Vivado Design Suite User Guide: Logic Simulation (UG900)* [Ref 6].

For instruction on implementing your core, see *Vivado Design Suite User Guide: Implementation (UG904)* [Ref 6].

Appendix A

Migrating

This appendix describes the changes from older versions of the IP to the current IP release.

Migrating from v4.0 to v5.0

Port Changes

There are no changes to ports, parameters or behavior from v4.0 to v5.0.

Latency Changes

There is no change to latency between v4.0 and v5.0.

Functionality Changes

There are no functional changes between v4.0 and v5.0.

Migrating from v3.0 to v5.0

Port Changes

Table A-1 details the changes to port naming, additional or deprecated ports and polarity changes.

Table A-1: **Port Changes from Version 3.0 to Version 5.0**

Version 3.0	Version 5.0	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.

Table A-1: **Port Changes from Version 3.0 to Version 5.0 *(Cont'd)***

Version 3.0	Version 5.0	Notes
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 in Chapter 3 for details.
FRACTIONAL		
ND		Deprecated. However this is analogous to the TVALID signals. See Instructions for Minimum Change Migration (v3.0 to v5.0).
RDY		Deprecated. However, this is analogous to TVALID on the output channel. See Instructions for Minimum Change Migration (v3.0 to v5.0).
RFD		Deprecated. However, this is analogous to TREADY on the input channels. See Instructions for Minimum Change Migration (v3.0 to v5.0)
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 in Chapter 3 for details.
	s_axis_dividend_tvalid	TVALID (AXI4-Stream channel handshake signal) for each channel
	s_axis_divisor_tvalid	
	m_axis_dout_tvalid	
	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 v5.0 is different compared to v3.0 for AXI Blocking mode. Latency is the same as v3.0 in v5.0 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 v5.0)

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.

Functionality Changes

From v3.0 to v5.0, there are functional changes due to the adoption of AXI4-Stream Interfaces. See Instructions for Minimum Change Migration (v3.0 to v5.0) and section on AXI4-Stream Considerations in Chapter 3.

Debugging

This appendix includes details about resources available on the Xilinx Support website and debugging tools. The following topics are included in this appendix:

Finding Help on Xilinx.com

To help in the design and debug process when using the Divider Generator, the Xilinx Support web page (www.xilinx.com/support) contains key resources such as product documentation, release notes, answer records, information about known issues, and links for opening a Technical Support WebCase.

Documentation

This product guide is the main document associated with the Divider Generator. This guide, along with documentation related to all products that aid in the design process, can be found on the Xilinx Support web page (www.xilinx.com/support) or by using the Xilinx Documentation Navigator.

Download the Xilinx Documentation Navigator from the Design Tools tab on the Downloads page (www.xilinx.com/download). For more information about this tool and the features available, open the online help after installation.

Answer Records

Answer Records include information about commonly encountered problems, helpful information on how to resolve these problems, and any known issues with a Xilinx product. Answer Records are created and maintained daily ensuring that users have access to the most accurate information available.

Answer Records for this core are listed below, and can also be located by using the Search Support box on the main Xilinx support web page. To maximize your search results, use proper keywords such as:

- Product name
- Tool message(s)
- Summary of the issue encountered

A filter search is available after results are returned to further target the results.

Master Answer Record for Divider Generator Core

AR 54499

Contacting Technical Support

Xilinx provides technical support at www.xilinx.com/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.

To contact Xilinx Technical Support:

1. Navigate to www.xilinx.com/support.
2. Open a WebCase by selecting the WebCase link located under Support Quick Links.

When opening a WebCase, include:

- Target FPGA including package and speed grade.
- All applicable Xilinx Design Tools and simulator software versions.
- Additional files based on the specific issue might also be required. See the relevant sections in this debug guide for guidelines about which file(s) to include with the WebCase.

Debug Tools

There are many tools available to address Divider Generator design issues. It is important to know which tools are useful for debugging various situations.

Example Design

The Divider Generator is delivered with an example design that can be synthesized, complete with functional test benches. Information about the example design can be found in *Chapter 6, Detailed Example Design*.

Vivado Lab Tools

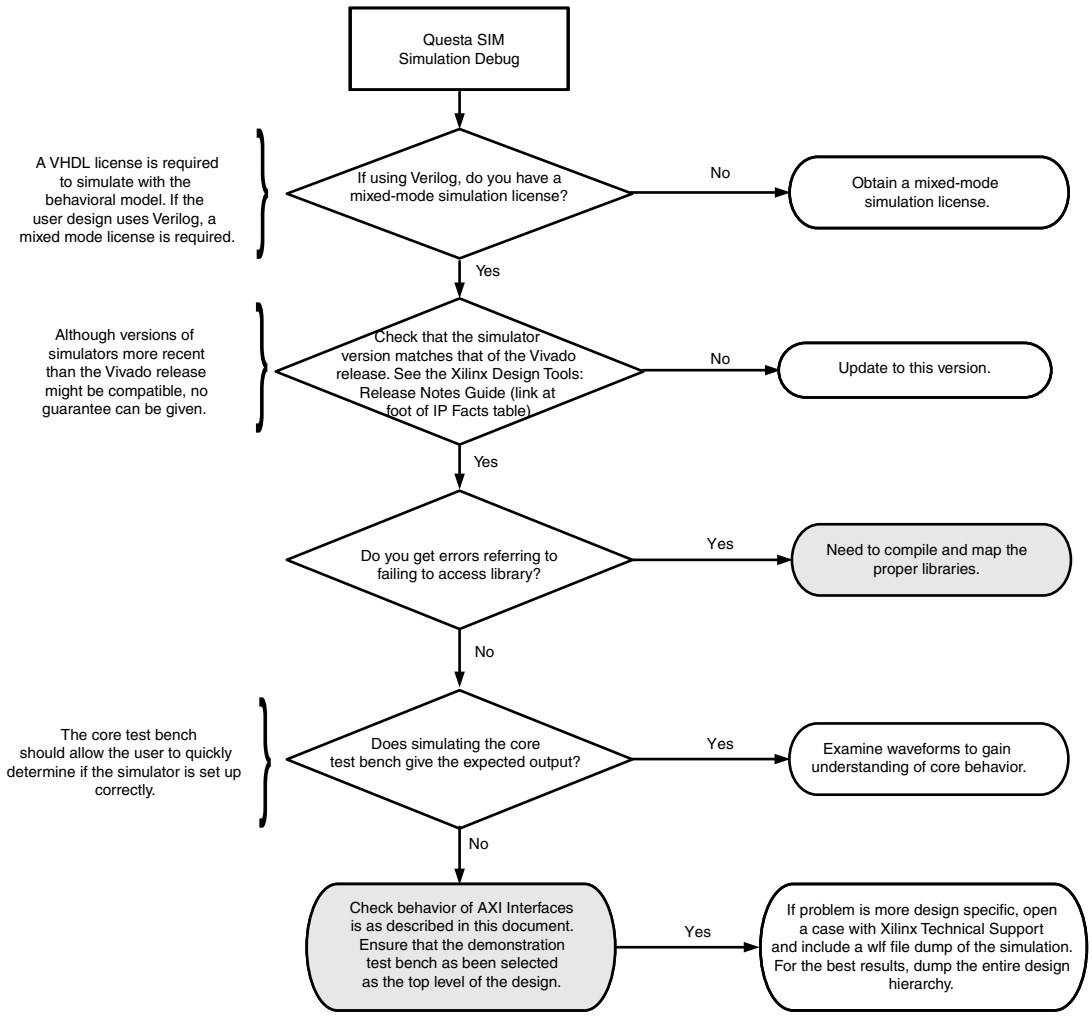
Vivado inserts logic analyzer and virtual I/O cores directly into your design. Vivado Lab Tools allows you to set trigger conditions to capture application and integrated block port signals in hardware. Captured signals can then be analyzed. This feature represents the functionality in the Vivado IDE that is used for logic debugging and validation of a design running in Xilinx FPGA devices in hardware.

The Vivado logic analyzer is used to interact with the logic debug LogiCORE IP cores, including:

- ILA 2.0 (and later versions)
- VIO 2.0 (and later versions)

Simulation Debug

The simulation debug flow for Questa SIM is illustrated in Figure B-1. A similar approach can be used with other simulators.



Interface Debug

AXI4-Stream Interfaces

If data is not being transmitted or received, check the following conditions:

- If transmit `<interface_name>_tready` is stuck low following the `<interface_name>_tvalid` input being asserted, the core cannot send data.
- If the receive `<interface_name>_tvalid` is stuck low, the core is not receiving data.
- Check that the `ACLK` inputs are connected and toggling.
- Check that the AXI4-Stream waveforms are being followed as described in AXI4-Stream Considerations in Chapter 3.
- Check core configuration.

Appendix C

Additional Resources

Xilinx Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see the Xilinx Support website at:

www.xilinx.com/support.

For a glossary of technical terms used in Xilinx documentation, see:

www.xilinx.com/company/terms.htm.

References

These documents provide supplemental material useful with this product guide:

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
6. Vivado Design Suite Documentation

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
03/20/2013	1.0	Initial Xilinx as a product guide. Replaces DS819, *LogiCORE IP Divider Generator Data Sheet*.

Notice of Disclaimer