Speedster7t Soft IP User Guide (UG103)

Speedster FPGAs

Preliminary Data

Copyrights, Trademarks and Disclaimers

Preliminary Data

This document contains preliminary information and is subject to change without notice. Information provided herein is based on internal engineering specifications and/or initial characterization data.

Achronix Semiconductor Corporation

2903 Bunker Hill Lane
Santa Clara, CA 95054
USA

Website: www.achronix.com
E-mail : info@achronix.com

Table of Contents

Chapter - 1: Introduction

There are a number of available soft IP cores for the Speedster®7t family of devices. Each of these cores has an IP configurator within ACE that allows configuration of the soft IP core. When configured, the generated wrapper for the core can be instantiated within a user project enabling both synthesis and simulation of the design.

This document describes the available soft IP cores and the methods for configuration and instantiation of each. Soft IP cores are primarily implemented using the components present in the FPGA programmable fabric. For details of these components, refer to the *Speedster7t IP Component Library User Guide* (UG086).

Chapter - 2: Instructions

Within ACE, all IP cores are accessed using the IP perspective (see the "Perspectives" chapter in the *ACE User Guide* (UG070). The flow and method for generating user IP cores is fully detailed in the "Creating an IP Configuration" chapter in the ACE User Guide. Unless directed otherwise below, follow the instructions in the ACE User Guide.

IO Ring and Core

Within the Speedster7t device there are two categories of IP core. These are listed within the IP Libraries pane as **IO Ring** and **Core**. For the Speedster7t family of devices, the following soft IP cores are available:

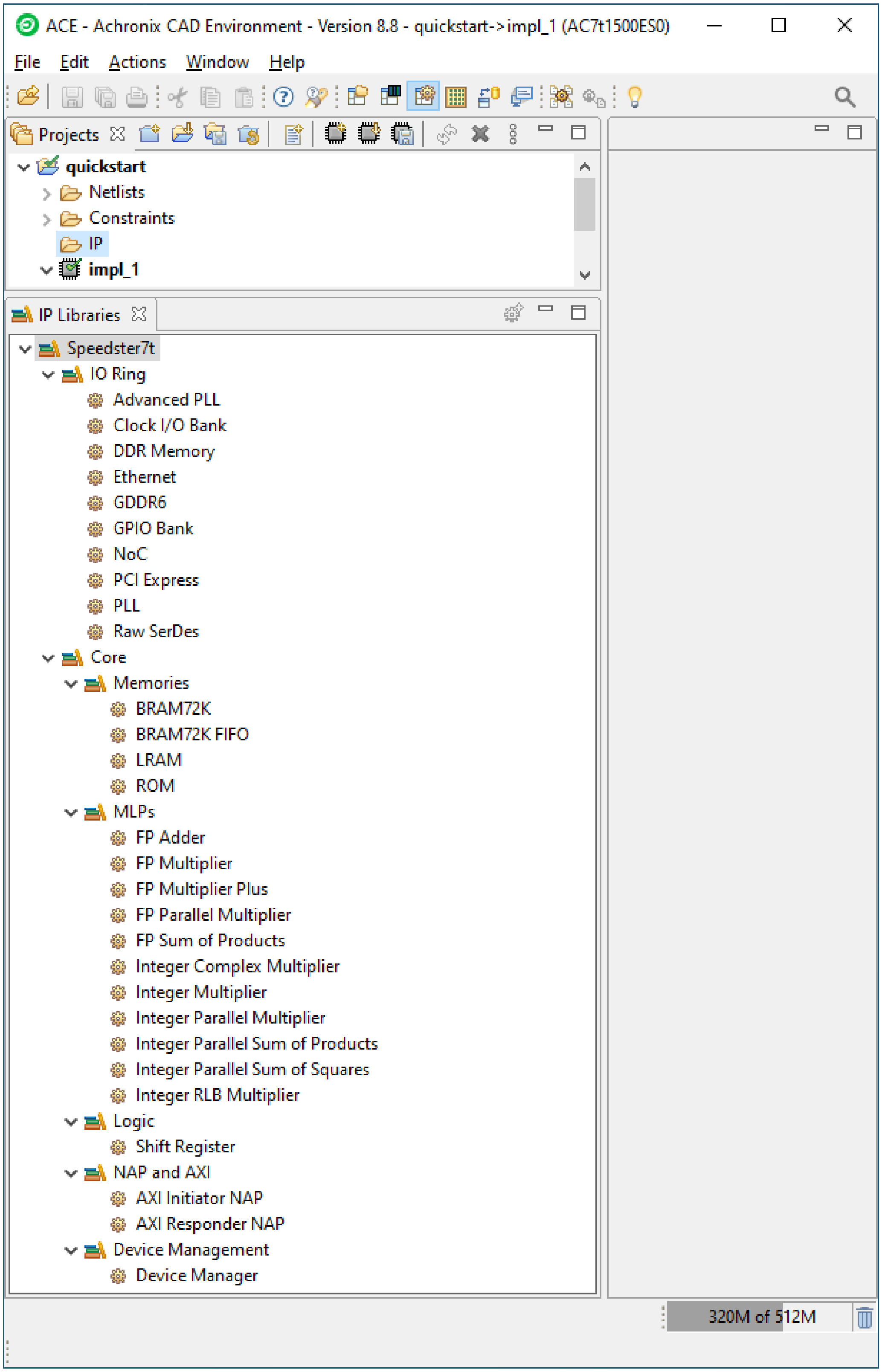

Figure 1: ***Speedster7t IP Libraries View***

IO Ring

The IO Ring contains the configuration for each of the IP cores located in the IO ring of the device such as the Clock Banks, PLLs, GDDR, DDR, GPIO, PCIe, NoC, SerDes and Ethernet. The configuration of each of these IP cores is detailed in its respective User Guide.

Core

The **Core** view contains the available IP configurators for the selected target device used in the project. For the Speedster7t devices, the available soft cores are shown above (see page 9), and listed in the following chapter (see page 11). The details of how to configure each of these cores are given in the appropriate chapter.

Chapter - 3: Available Configurators

Memories

- BRAM72K Soft IP (see page 12) – for creating large block RAM memory arrays
- LRAM Soft IP (see page 16) – for creating large logic RAM memory arrays
- ROM Soft IP (see page 20) – for creating ROMs constructed of either block RAM or logic RAM

MLPs

- Integer Complex Multiplier Soft IP (see page 24) – for creating a complex multiplier with a single machine learning processing block.
- Integer Multiplier Soft IP (see page 28) – for creating a single multiplier of up to 32 × 32
- Integer Parallel Multiplier Soft IP (see page 32) – for creating parallel multipliers of up to 32 × 32
- Integer Parallel Sum of Products Soft IP (see page 37) – for integer sum of products from up to 24 multipliers
- Integer Parallel Sum of Squares Soft IP (see page 42) – for integer sum of squares inputs
- Integer RLB Multiplier Soft IP (see page 47) – for creating a single multiplier using RLBs in the fabric logic

Logic

- Shift Register Soft IP (see page 51) – for creating DFF-based shift registers

NAP and AXI

- AXI Initiator NAP Soft IP (see page 55) – for creating a NAP that initiates AXI traffic
- AXI Responder NAP Soft IP (see page 60) – for creating a NAP that responds to AXI traffic

Device Management

- Device Manager Soft IP (see page 65) – for creating the Achronix Device Manager to run GDDR6 training and used in all builds to manage JTAG communication

Chapter - 4: BRAM72K Soft IP

Description

The Speedster7t BRAM72K soft IP core creates an arbitrary sized memory array, comprised of ACX_BRAM72K primitives. The macro employs the embedded data and address cascade paths between ACX_BRAM72K primitives enabling fast connections for both address and data paths.

If only a single ACX_BRAM72K is required, this primitive can be inferred or instantiated in the code directly. However, if a memory array comprising multiple BRAM72K blocks is required, it is recommended to use the soft IP configuration to enable the optimum architecture.

Configuration

The user macro has the following configuration options:

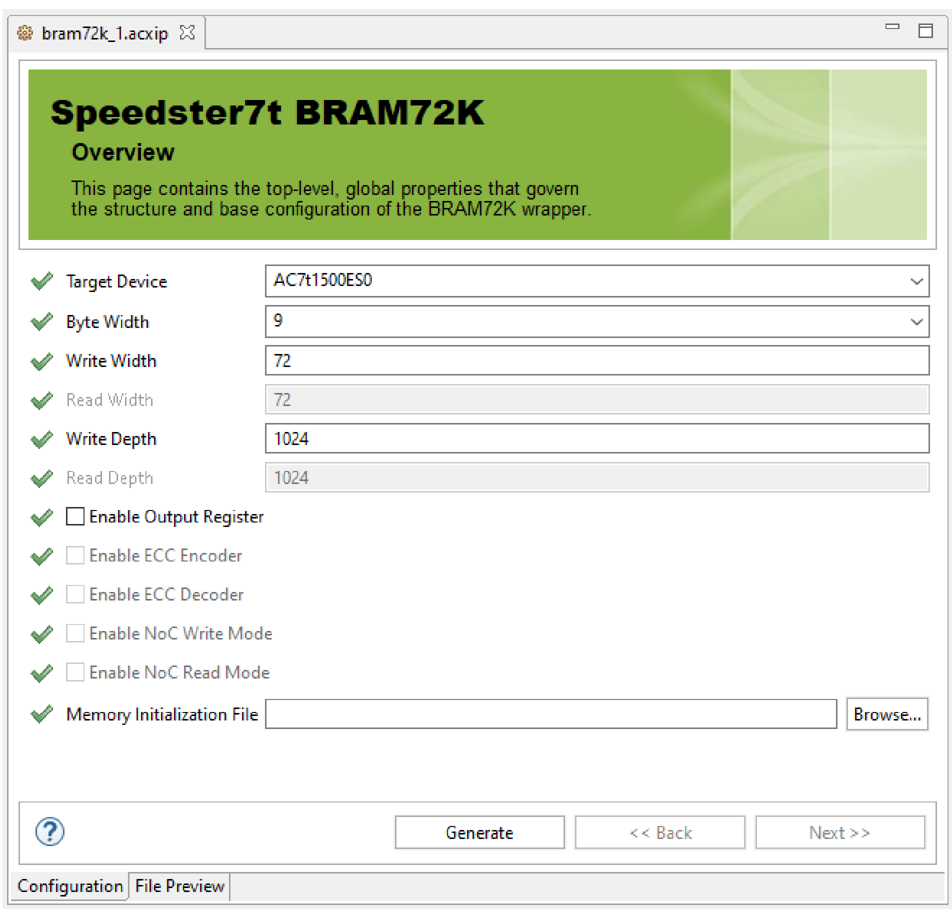

Figure 2: ***BRAM72K Soft IP Configuration***

Table 1: *Configuration Options*

Name	Default	Range	Description
Target Device	AC7t1500ES0	All Speedster7t devices	Set to match the target device of the project.
Byte Width	9	8, 9	Determines whether fields should be set to 8-bit or 9-bit.
Write Width	72	1 to 9216	Write port data width. Values greater than 144 limit the write depth to 16K words.
Read Width	72	1 to 9216	Read port data width. Currently set to match the write width. This value cannot be changed by the user. Future releases of this user macro are planned to allow configuring different write and read widths.
Write Depth	1024	512 to 1048576	Write port address depth. The maximum value is limited by the number of BRAM72K blocks in a column in the target device. The maximum value is also dependant on the write width as detailed in Write and Read Depths versus Data Width (see page 14)
Read Depth	1024	512 to 1048576	Read port address depth. Currently set to match the write depth. This value cannot be changed by the user. Future releases of this user macro are planned to allow configuring different write and read depths.
Enable Output Register	Off	On, Off	Determines whether the output register in each of the BRAM72K primitives is enabled. Adds an additional cycle of latency to any read operation.
Enable ECC Encoder	Off	On, Off	Determines whether the ECC encoder is enabled for writes to the memory array. This option is currently disabled and cannot be set by the user.
Enable ECC Decoder	Off	On, Off	Determines whether the ECC encoder is enabled for reads from the memory array. This option is currently disabled and cannot be set by the user.
Enable NoC Write Mode	Off	On, Off	Determines whether the BRAM can be written directly from the NoC. This option is currently disabled and cannot be set by the user.
Enable NoC Read Mode	Off	On, Off	Determines whether the BRAM can be read directly from the NoC. This option is currently disabled and cannot be set by the user.
Use Memory Initialization File	Off	On, Off	Determines whether a memory initialization file(1) is used to initialize the memory contents. This initialization occurs for both synthesis and simulation. When this option is enabled, entry of the file location in the associated file browser dialog is permitted.

Table Notes

1. If relative paths are used for the memory initialization file location, the same relative paths must be valid from both the ACE project directory and the simulation directory. It is recommended to locate both of these directories at the same relative depth in the project tree, and to use relative paths that navigate up the tree to the first common directory, before descending the tree to the location of the files.

   For example: The Achronix reference designs locate the ACE project in <project root>/src/ace. The simulation directories are located in <project root>/sim/vcs or <project root>/sim/questa. The memory initialization files are located in <project root>/src/mem_init_files. A relative path correct for both simulation and ACE is "../../src/mem_init_files/filename.txt"

Write and Read Depth

Absolute Limits

The write and read depths are related to the write and read widths. The absolute limit on these values is detailed in the table below:

Table 2: ***Write and Read Depths Versus Data Width***

Memory Width	Maximum Memory Depth
1 to 144	1048576
145 to 9216	16384

Device Specific Limits

Within a device, the largest memory that can be generated is limited by the number of ACX_BRAM72K primitives in a column. For example, if there are 64 ACX_BRAM72K primitives in a column, then the maximum memory size is 64 × 72K bits = 4,718,592 bits. This would support a configuration of 72-bits × 64K depth, or 36-bits × 128K depth.

Examples

The following figure shows the macro configured for a 4096-bit by 16,384 entry memory with the memory output register enabled:

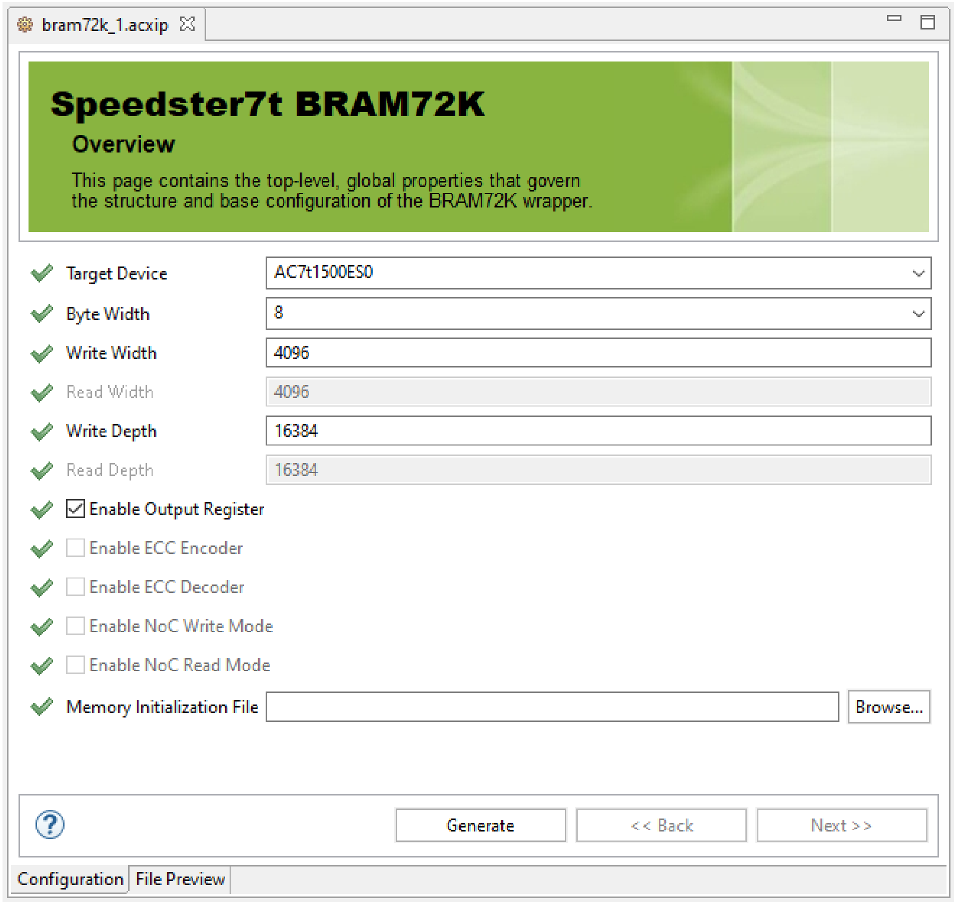

Figure 3: ***4096 × 16K Memory Configuration***

The following figure shows the IP diagram for the above configuration:

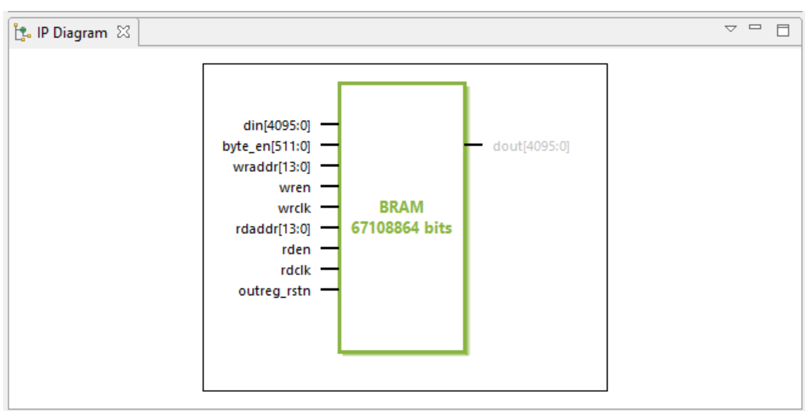

Figure 4: ***4096 × 16K Memory IP Diagram***

Chapter - 5: LRAM Soft IP

Description

The LRAM soft IP core creates an arbitrary sized memory array comprised of LRAM primitives.

If only a single LRAM is required, this primitive can be inferred or instantiated into the code directly. However, if a memory array consisting of multiple LRAM primitives is required, it is recommended to use the soft IP configurator to achieve the optimum architecture.

Utilization

> **Note**
>
> Within the Speedster7t family, the LRAM and MLP primitives share a site. Therefore if a site is allocated for use as an LRAM primitive, the MLP on that same site cannot be used.

Configuration

The LRAM2K soft IP configurator has the following configuration options:

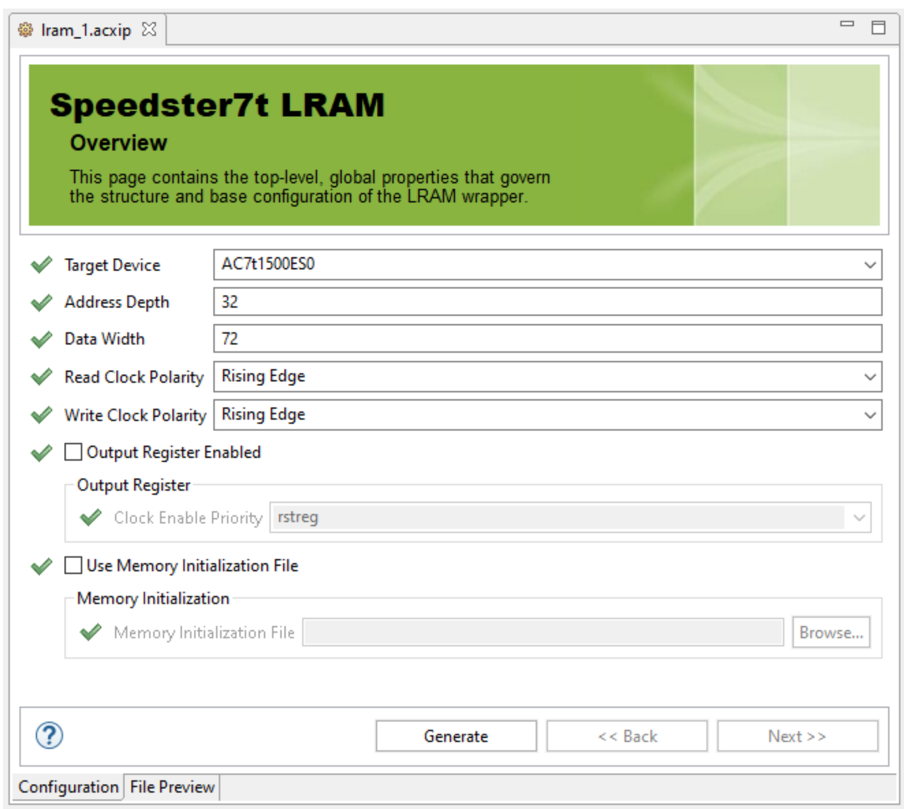

Figure 5: ***LRAM Soft IP Configurator***

Table 3: *Configuration Options*

Name	Default	Range	Description
Target Device	AC7t1500ES0	All Speedster7t devices	Set to match the target device of the project.
Address Depth	32	4 to 4096	Address depth of the memory array in words. The depth imposes limitations on the maximum data width. The limits are detailed in Read and Write Depths Versus Data Widths (see page 17).
Data Width	72	1 to 184320	Data port width in bits of both `din` and `dout`.
Read Clock Polarity	Rising Edge	Falling or Rising Edge	The `rdclk` active edge on which all read transactions will occur.
Write Clock Polarity	Rising Edge	Falling or Rising Edge	The `wrclk` active edge on which all write transactions will occur.
Output Register Enabled	Off	On, Off	Determines whether the output register in each of the LRAM primitives is enabled. This adds an additional cycle of latency to any read operation. If the output register is disabled, the memory array is combinatorial. The output changes when the input address changes.
Output Register Clock Enable Priority	rstreg	rstreg, rstce	Controls the clock enable input of the output register. • **rstreg:** The `outregce` input is ignored when `rstregn` = 1'b0. The output register is reset on the next active `rdclk` edge. • **rstce:**`outregce` must be equal to 1'b1 and `rstregn` = 1'b0 for the output register to be reset on the next active `rdclk` edge.
Use Memory Initialization File	Off	On, Off	Determines whether a memory initialization file[1] is used to initialize the memory contents. This initialization occurs for both synthesis and simulation. When this option is enabled, entry of the file location in the associated file browser dialog is permitted.

Table Notes

1. If relative paths are used for the memory initialization file location, the same relative paths must be valid from both the ACE project directory and the simulation directory. It is recommended to locate both these directories at the same relative depth in the project tree, and to use relative paths that navigate up the tree to the first common directory, before descending the tree to the location of the files.
   For example, the Achronix reference designs locate the ACE project in `<project root>/src/ace`. The simulation directories are located in `<project root>/sim/vcs` or `<project root>/sim/questa`. The memory initialization files are located in `<project root>/src/mem_init_files`. A relative path correct for both simulation and ACE is `"../../src/mem_init_files/filename.txt"`.

Write and Read Depth

Absolute Limits

The write and read depths are related to the write and read widths. The absolute limit on these values is detailed in the table below:

Table 4: ***Write and Read Depths versus Data Width***

Memory Width	Maximum Memory Depth
4 to 32	184320
33 to 64	92160
65 to 96	61416
97 to 128	46080
129 to 144	36864

Device Specific Limits

Within a device, the largest memory that can be generated is limited by the number of ACX_LRAM primitives in a column. For example, if there are 64 ACX_LRAM2K primitives in a column, the maximum memory size is 64 × 2K bits = 131,027 bits. This would support a configuration of 64-bits x 2K depth, or 32-bits × 4K depth. Alternatively, if there are 64 ACX_LRAM4K in a column, the maximum memory size is 64 × 4K bits = 262,144 bits. This would support a configuration of 64-bits × 4K depth, or 32-bits × 8K depth.

The type of LRAM used by the Speedster7t LRAM configurator is dependent upon the device chosen and the available LRAM types within the fabric.

Examples

The following figure shows the soft IP configured for a 128-bit × 4096-word LRAM memory with the memory output register enabled:

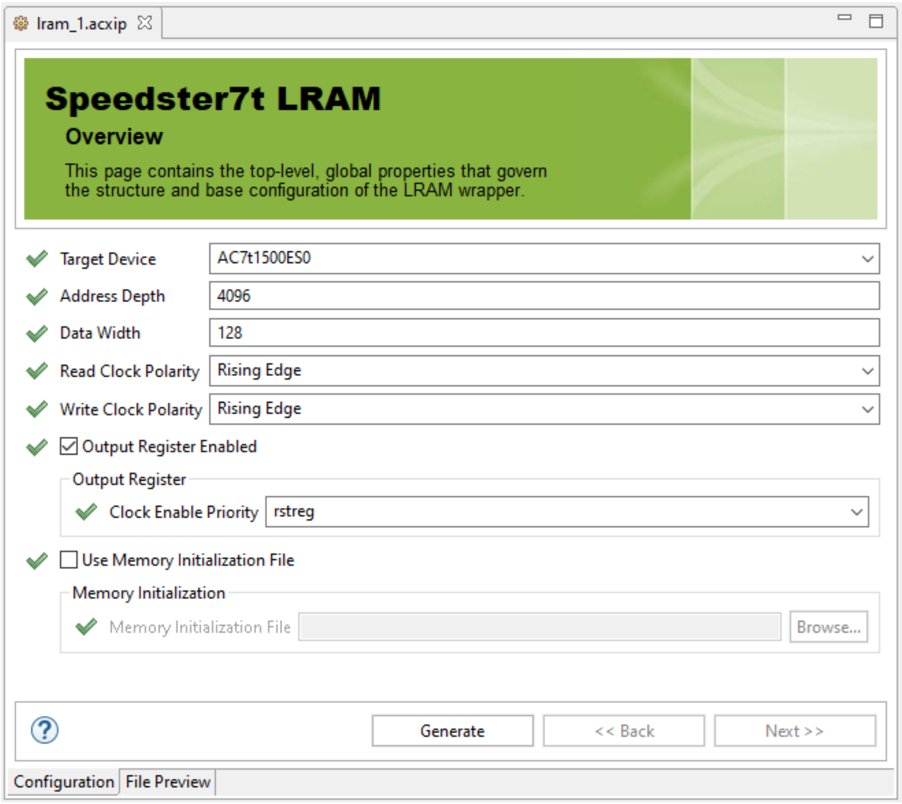

Figure 6: ***128-Bit x 4096-Word LRAM Memory Configuration***

The following figure shows the IP diagram for the above configuration:

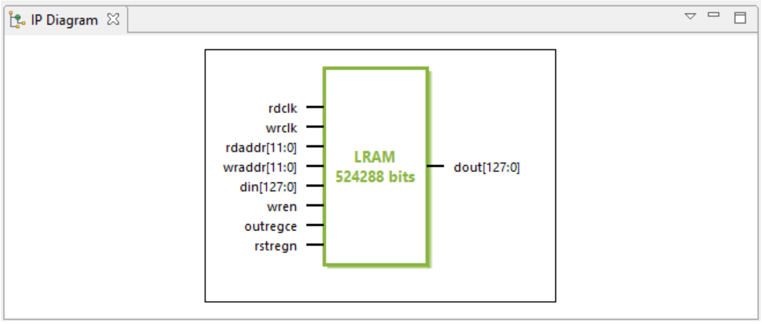

Figure 7: ***128-Bit x 4096-Word LRAM Memory IP Diagram***

Chapter - 6: ROM Soft IP

Description

The Speedster7t ROM soft IP core creates an arbitrary sized ROM, using either BRAM or LRAM primitives.

Utilization

Note

Within the Speedster7t family, the LRAM and MLP primitives share a site. Therefore, if using a Speedster7t device, and if the ROM soft IP configuration selects an LRAM to implement the ROM, the MLPs on the sites used by the ROM cannot be used.

Configuration

The soft IP has the following configuration options:

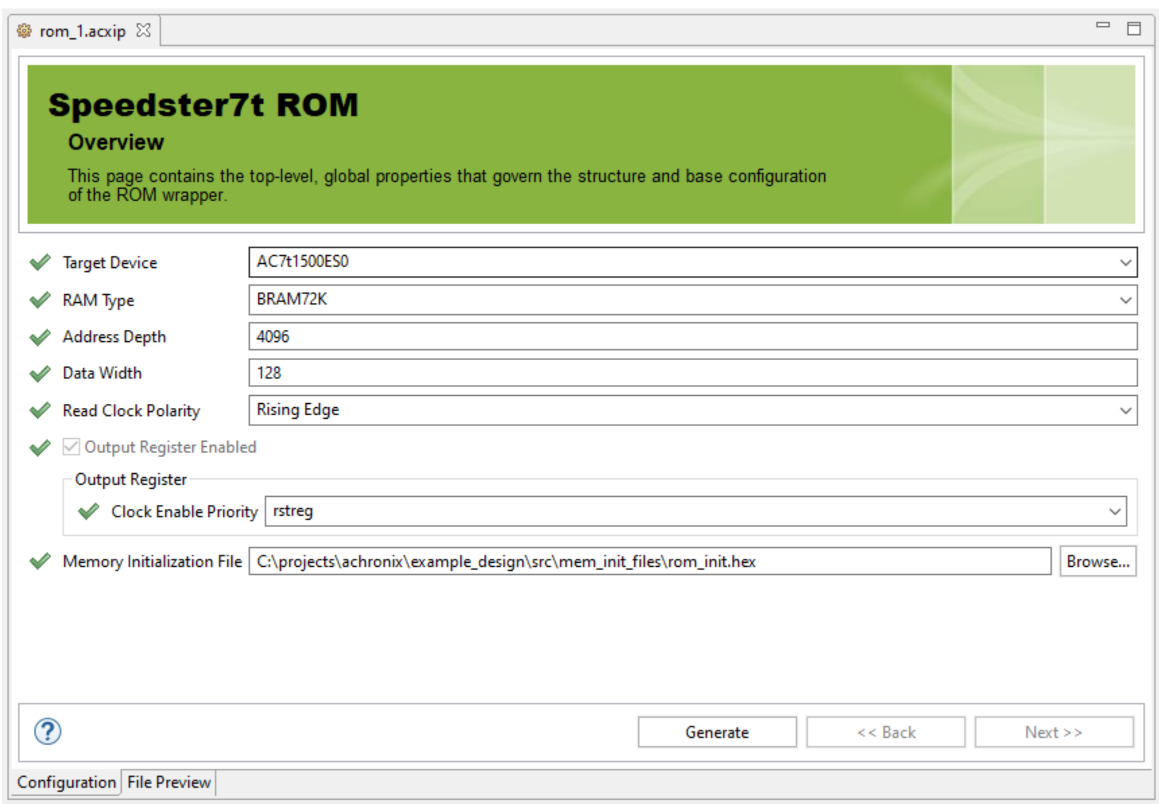

Figure 8: ***ROM Soft IP Configuration***

Table 5: *Configuration Options*

Name	Default	Range	Description
Target Device	AC7t1500ES0	All Speedster7t devices	Set to match the target device of the project.
RAM Type	BRAM72K	BRAM72K, LRAM2K	Determines which type of RAM primitives used to form the ROM.
Address Depth	1024	4 to 16384	Address depth of the ROM in words.
Data Width	20	1 to 184320	Port width in bits of the `dout` port. The width selected affects the available address depth. The maximum values of width versus depth are detailed in Read and Write Depths Versus Data Widths (see page 22).
Read Clock Polarity	Rising Edge	Falling or Rising Edge	The `rdclk` active edge on which all read transactions will occur.
Enable Output Register	Off	On, Off	Determines whether the output register in each of the LRAM2K primitives is enabled. Adds an additional cycle of latency to any read operation. If the output register is disabled, the memory array is combinatorial. The output changes when the input address changes.
Output Register Clock Enable Priority	rstreg	rstreg, rstce	Controls the clock enable input of the output register. • **rstreg**: The `outregce` input is ignored when `rstregn` = 1'b0. The output register is reset on the next active `rdclk` edge. • **rstce**: `outregce` must be equal to 1'b1 and `rstregn` = 1'b0 for the output register to be reset on the next active `rdclk` edge.
Use Memory Initialization File	On	On	Location of the memory initialization file[1] used to initialize the memory contents. This initialization occurs for both synthesis and simulation.

Table Notes

1. If relative paths are used for the memory initialization file location, the same relative paths must be valid from both the ACE project directory and the simulation directory. It is recommended to locate both of these directories at the same relative depth in the project tree, and to use relative paths that navigate up the tree to the first common directory, before descending the tree to the location of the files.

   For example, the Achronix reference designs locate the ACE project in `<project root>/src/ace`. The simulation directories are located in `<project root>/sim/vcs` or `<project root>/sim/questa`. The memory initialization files are located in `<project root>/src/mem_init_files`. A relative path correct for both simulation and ACE is `"../../src/mem_init_files/filename.txt"`.

Write and Read Depth

Absolute Limits

The write and read depths are related to the write and read widths. The absolute limit of these values is detailed in the table below:

Table 6: *Write and Read Depths Versus Data Width*

RAM Type	Memory Width	Maximum Memory Depth
ACX_BRAM72K	1 to 11520	16384
	11521 to 23040	8096
	23041 to 46080	4096
	46081 to 92160	2048
	92161 to 184320	1024
ACX_LRAM2K	1–1440	4096
	1441 to 2880	2048
	2881 to 5760	1024
	5761 to 11520	512
	11521 to 23040	256
	23041 to 46080	128
	46081 to 92160	64
	92161 to 184320	32

Device Specific Limits

Within a device, the largest memory that can be generated is limited by the number of RAM primitives in a column, and the choice of RAM type. The overall ROM limit in bits is equivalent to the number of RAM primitives in a column multiplied by the number of bits in the primitive, (72K for ACX_BRAM72K, and 2K for ACX_LRAM2K).

Examples

The following figure shows the soft IP configured for a 128-bit × 4096-word ROM formed of BRAM72K primitives with the memory output register enabled.

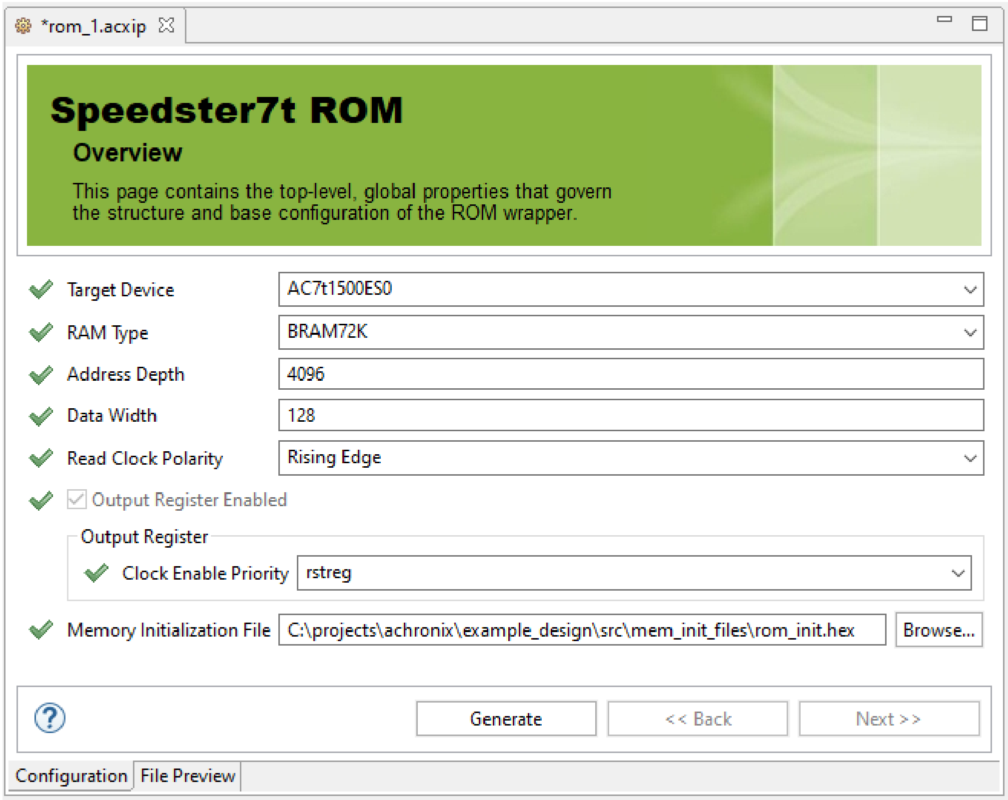

Figure 9: ***128-bit × 4096-word ROM Configuration***

The following figure shows the soft IP I/O diagram for the above configuration.

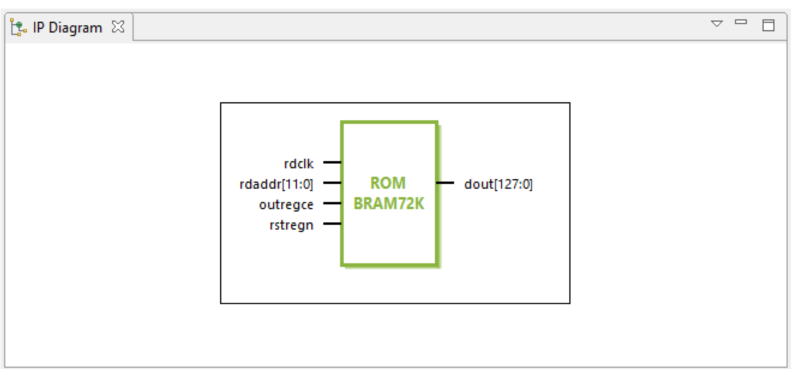

Figure 10: ***128-Bit × 4096-Word ROM IP Diagram***

Chapter - 7: Integer Complex Multiplier Soft IP

Description

The Integer Complex Multiplier Soft IP implements a complex multiplication with a single machine learning processing block.

Configuration

The integer complex multiplier soft IP configurator has the following options:

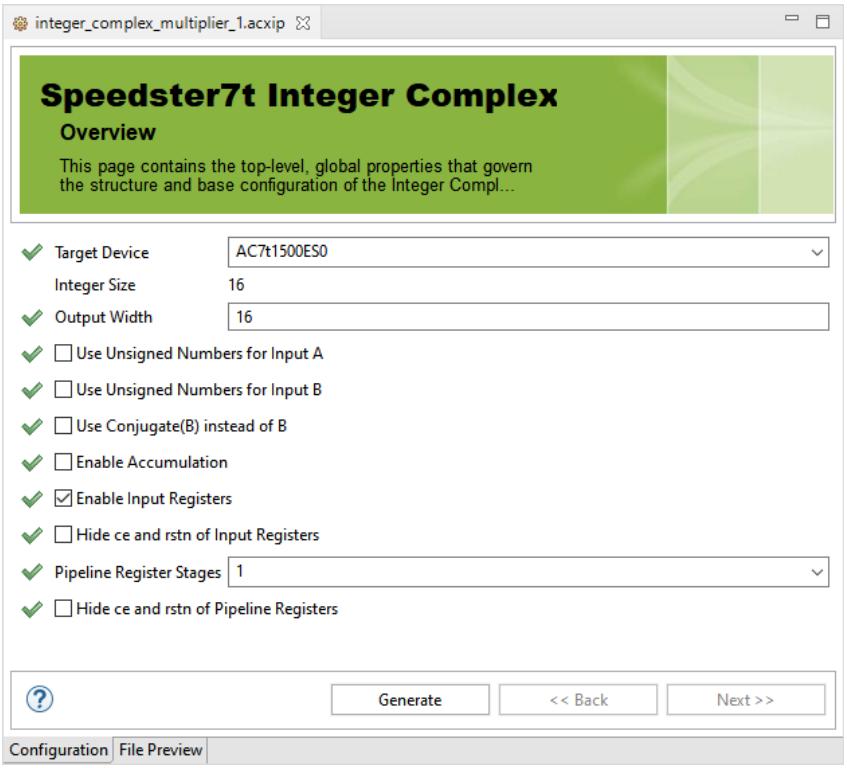

Figure 11: ***Complex Integer Multiplier Soft IP Configurator***

Table 7: *Configuration Options*

Name	Default	Range	Description
Target Device	AC7t1500ES0	All Speedster7t devices	Set to match the target device of the project.
Use Unsigned Numbers for Input A	No	Yes, No	0 – `i_din_a_re` and `i_din_a_im` are signed (two's complement). 1 – `i_din_a_re` and `i_din_a_im` are unsigned.
Use Unsigned Numbers for Input B	No	Yes, No	0 – `i_din_b_re` and `i_din_b_im are` signed (two's complement). 1 – `i_din_b_re` and `i_din_b_im` are unsigned.
Use Conjugate(B) instead of B	No	Yes, No	Compute A × conjugate(B) instead of A × B.
Enable Accumulation	No	Yes, No	0 – No accumulation: (`dout_re` + `dout_im` · `j`) = (`i_din_a_re` + `i_din_a_im` · `j`) × (`i_din_b_re` + `i_din_b_im` · `j`). 1 – Accumulation: `dout_re` + `dout_im` · `j` is the accumulated value. The start of accumulation is signaled by asserting `i_load` = `1`.
Enable Input Registers	No	Yes, No	0 – No input registers. 1 – `i_din_a_re`, `i_din_a_im`, `i_din_b_re`, and `i_din_b_im` are registered. The input registers are controlled by the `i_in_reg_a_ce`, `i_in_reg_b_ce`, and `i_in_reg_rstn` inputs. Enabling the input register adds one cycle of latency.
Hide `ce` and `rstn` of Input Registers	No	Yes, No	If selected, the `i_in_reg_a_ce`, `i_in_reg_b_ce`, and `i_in_reg_rstn` inputs will be automatically tied high (`1'b1`).
Pipeline Register Stages	0	0, 1, 2	The number of pipeline registers, not counting the input register. The total latency is `pipeline_regs` + `in_reg_enable`.
Hide `ce` and `rstn` of Pipeline Registers	No	Yes, No	If selected, the `i_pipeline_ce` and `i_pipeline_rstn` inputs will be automatically tied high (`1'b1`).

Table 8: *Ports*

Name	Direction	Description
`i_clk`	Input	Clock input, used for the (optional) registers and accumulator.
`i_din_a_re`[(Integer Size – 1):0]	Input	Real coefficient of A data input to multiplier.
`i_din_a_im`[(Integer Size – 1):0]	Input	Imaginary coefficient of A data input to multiplier.
`i_din_b_re`[(Integer Size – 1):0]	Input	Real coefficient of B data input to multiplier.
`i_din_b_im`[(Integer Size – 1):0]	Input	Imaginary coefficient of B data input to multiplier.
`i_load`	Input	Resets the accumulator to A × B, ignoring the previous value. This signal is internally pipelined to have the same latency as A × B.
`i_in_reg_a_ce`	Input	(Optional) Clock enable for `i_din_a_re` and `i_din_a_im`. Present when **Enable Input Registers** is set to **On**.
`i_in_reg_b_ce`	Input	(Optional) Clock enable for `i_din_b_re` and `i_din_b_im`. Present when **Enable Input Registers** is set to **On**.
`i_in_reg_rstn`	Input	(Optional) Synchronous active-low reset for input registers. Present when **Enable Input Registers** is set to **On**.
`i_pipeline_ce`	Input	(Optional) Clock enable for pipeline and accumulator registers. Present when **Pipeline Register Stages** is greater than 0.
`i_pipeline_rstn`	Input	(Optional) Synchronous active-low reset for pipeline and accumulator registers. Present when **Pipeline Register Stages** is greater than 0.
`o_dout_re`[(Output Width – 1):0]	Output	Real coefficient of the result of multiplication and accumulation. Always signed (2's complement).
`o_dout_im`[(Output Width – 1):0]	Output	Imaginary coefficient of the result of multiplication and accumulation. Always signed (2's complement).

Examples

The following example shows the integer complex multiplier configured for 16-bit unsigned inputs with input registers, accumulation and two pipeline stages, and 32-bit output:

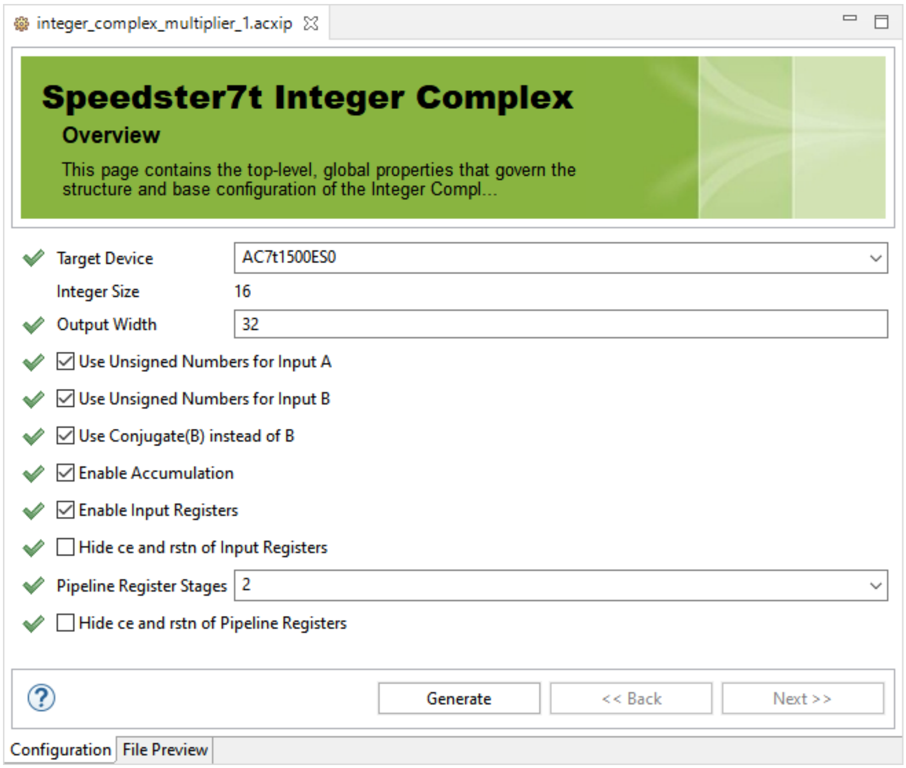

Figure 12: ***Two Pipeline Stage Integer Multiplier Configuration***

The following figure shows the IP diagram for the above configuration:

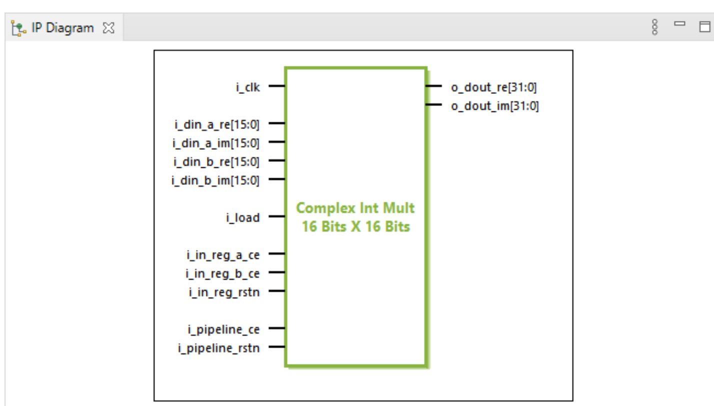

Figure 13: ***Two Pipeline Stage Integer Multiplier I/O***

Chapter - 8: Integer Multiplier Soft IP

Description

The Integer Multiplier soft IP core configures a two-input integer multiplier using either RLB (logic) based multipliers or MLP primitives. The multiplier also supports optional result accumulation. The configurator supports sizes of up to 32 × 32 integers, with both signed and unsigned numerical formats.

Configuration

The Integer Multiplier soft IP configurator has the following options:

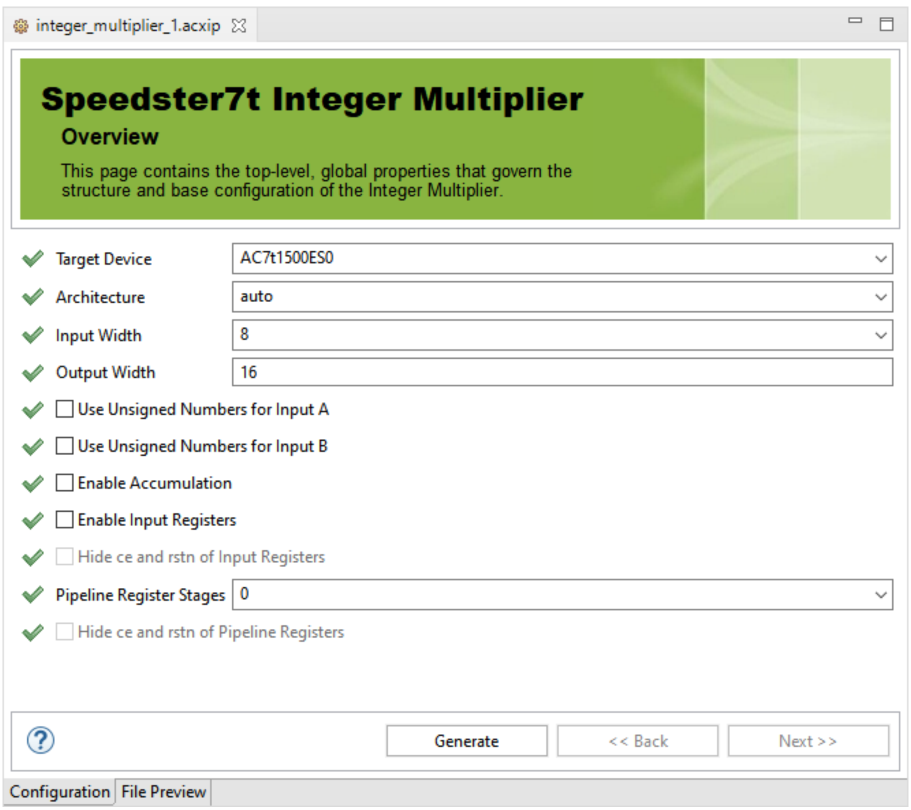

Figure 14: ***Integer Multiplier Soft IP Configurator***

Table 9: *Configuration Options*

Name	Default	Range	Description
Target Device	AC7t1500ES0	All Speedster7t devices	Set to match the target device of the project.
Architecture	auto	auto, rlb, mlp	Determines which primitives should be used to implement the multiplier. • **auto**: Allow the tool to choose the most appropriate primitive based on the number formats and sizes selected. • **rlb**: Implement the multiplier in fabric logic. The Speedster7t FPGA has a unique MLUT structure which supports very efficient multiplier arrays using logic. • **mlp**: Use the MLP primitive to implement the multiplier.
Input Width	8	3, 4, 5, 6, 7, 8, 16 or 32	Width of both a and b inputs.
Use Unsigned Numbers for Input A/B	Off	On, Off	When set, configures the appropriate input to use unsigned numbers. By default, the inputs are set to signed.
Enable Input Registers	Off	On, Off	When set, enables a register stage for both A and B inputs. This stage adds a cycle of latency to all results. Enabling the input registers adds the inputs `i_in_reg_a_ce`, `i_in_reg_b_ce` and `i_in_reg_rstn` to the resultant soft IP.
Hide `ce` and `rstn` of Input Registers	No	Yes, No	If selected, the `i_in_reg_a_ce`, `i_in_reg_b_ce`, and `i_in_reg_rstn` inputs are automatically tied high (`1'b1`).
Enable Accumulator	Off	On, Off	The output is the accumulation of the result from each clock cycle. Enabling accumulation adds the input `i_load` to the resultant soft IP. The accumulation is cleared when `i_load` is asserted, the output is reset to `i_din_a` × `i_din_b`. The `i_load` signal has the same pipeline delay to the accumulator as the `i_din_a` and `i_din_b` inputs. Therefore it should be applied on the same cycle as the `i_din_a` and `i_din_b` inputs that are to start a new accumulation cycle.
Pipeline Register Stages	0	0, 1, 2, 3	Add pipeline register stages through the multiplication process. Enabling pipeline registers improves timing performance at the cost of an additional cycle of latency for each stage enabled. When any pipeline stages are enabled, the inputs `i_pipeline_ce` and `i_pipeline_rstn` are added to the resultant soft IP.
Hide `ce` and `rstn` of Pipeline Registers	No	Yes, No	If selected, the `i_pipeline_ce` and `i_pipeline_rstn` inputs are automatically tied high (`1'b1`).
Output Width	16	8 to 128	Width of the data output. Automatically updated by the configurator when Input Width is updated. In addition, the value can be modified to meet requirements. The valid range changes dependent upon the Input Width and Architecture. (1)

Table Notes

1. When accumulation is enabled, it might be necessary to increase the data output width to account for the growth in the result over multiple accumulation cycles. The minimum output width can be calculated as (2 × Input Width) + (number of accumulation cycles).

Table 10: *Ports*

Name	Direction	Description
i_clk	Input	Clock input, used for the (optional) registers and accumulator.
i_din_a[(Input Width – 1):0]	Input	'A' data input to the multiplier.
i_din_b[(Input Width – 1):0]	Input	'B' data input to the multiplier.
i_in_reg_a_ce	Input	(Optional) Clock enable for i_din_a. Present when **Enable Input Registers** is set to **On**.
i_in_reg_b_ce	Input	(Optional) Clock enable for i_din_b. Present when **Enable Input Registers** is set to **On**.
i_in_reg_rstn	Input	(Optional) Synchronous active-low reset for input registers. Present when **Enable Input Registers** is set to **On**.
i_pipeline_ce	Input	(Optional) Clock enable for pipeline and accumulator registers. Present when **Pipeline Register Stages** is greater than 0.
i_pipeline_rstn	Input	(Optional) Synchronous active-low reset for pipeline and accumulator registers. Present when **Pipeline Register Stages** is greater than 0.
i_load(1)	Input	(Optional) When asserted to 1'b1, resets the accumulator to i_din_a × i_din_b, ignoring the previous value. Present when **Enable Accumulator** is set to **On**.
o_dout[(Output Width – 1):0]	Output	Result of multiplication and accumulation.

Table Notes

1. This signal is internally pipelined to have the same latency as i_din_a and i_din_b.

Examples

The following example shows the integer multiplier configured for signed 32 × 32 inputs, with accumulation and a single pipeline stage:

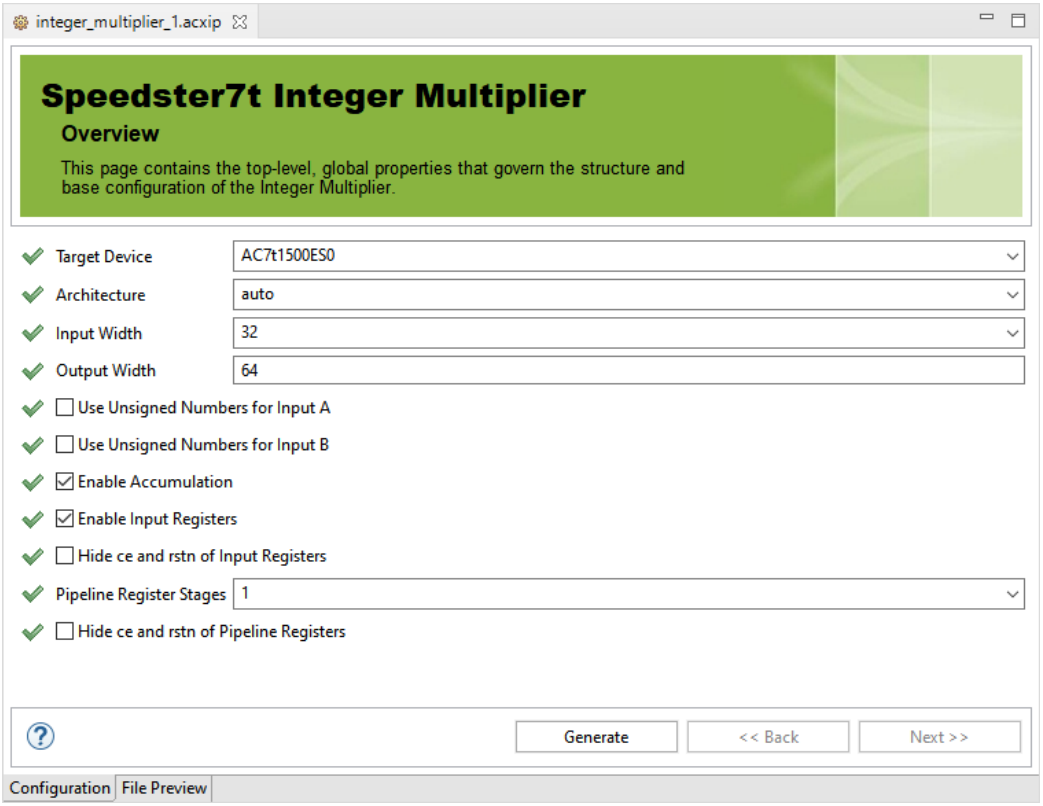

Figure 15: ***32 × 32 Signed Integer Multiplier Configuration***

The following figure shows the IP diagram for the above configuration:

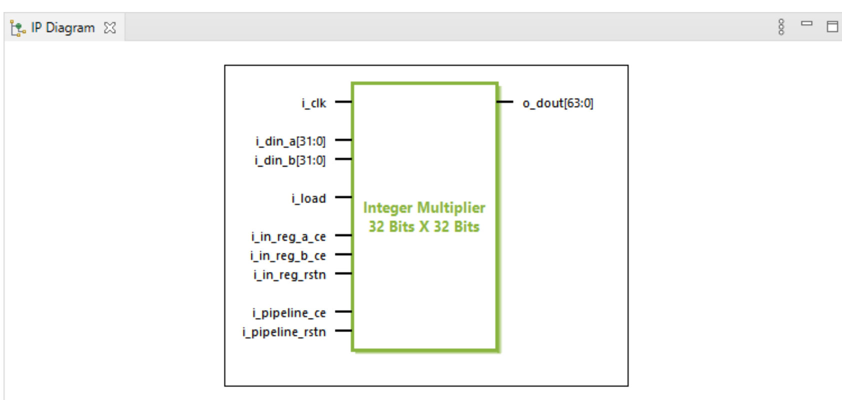

Figure 16: ***32 x 32 Signed Integer Multiplier I/O***

Chapter - 9: Integer Parallel Multiplier Soft IP

Description

The Integer Parallel Multiplier soft IP core configures multiple integer multipliers. This soft IP core is implemented with the MLP primitive which contains an array of integer multipliers. There can be up to 8 separate multipliers ranging from 3 × 3 to 16 × 16 bits. The multipliers support both signed and unsigned arithmetic.

Configuration

The Integer Parallel Multiplier soft IP configurator has the following options:

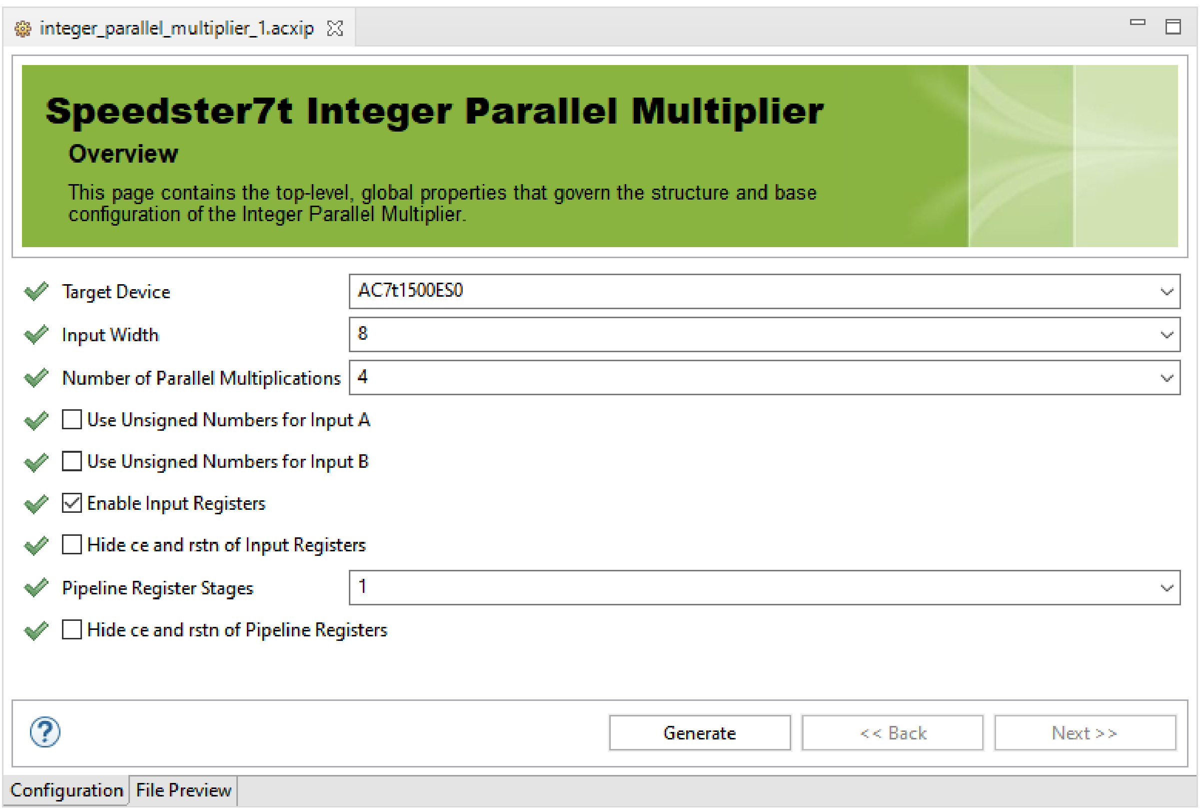

Figure 17: *Integer Parallel Multiplier Soft IP Configurator*

Table 11: *Configuration Options*

Name	Default	Range	Description
Target Device	AC7t1500ES0	All Speedster7t devices	Set to match the target device of the project.
Input Width	8	3, 4, 5, 6, 7, 8 or 16	Width of the two inputs to each multiplier
Number of Parallel Multiplications	4	2 to 8	The number of parallel multipliers to be implemented. The maximum number of multipliers is determined by the Input Width. Refer to the table Number of Multipliers Per Input Width (see page 34), below, for details.
Use Unsigned Numbers for Input A/B	Off	On, Off	When set, configures the appropriate inputs to use unsigned numbers. By default, the inputs are set to signed.
Enable Input Registers	Off	On, Off	When set, enables a register stage for all multiplier inputs. This adds a cycle of latency to all results. Enabling the input registers adds these inputs to the resultant soft IP core: • `i_in_reg_a_ce` • `i_in_reg_b_ce` • `i_in_reg_rstn`
Hide `ce` and `rstn` of Input Registers	No	Yes, No	If selected, the `i_in_reg_a_ce`, `i_in_reg_b_ce`, and `i_in_reg_rstn` inputs are automatically tied high (`1'b1`).
Pipeline Register Stages	0	0, 1	Adds a pipeline register stage to the multiplication process. Enabling pipeline registers improves timing performance at the cost of an additional cycle of latency. When any pipeline stages are enabled, these inputs are added to the resultant soft IP core: • `i_pipeline_ce` • `i_pipeline_rstn`
Hide `ce` and `rstn` of Pipeline Registers	No	Yes, No	If selected, the `i_pipeline_ce` and `i_pipeline_rstn` inputs are automatically tied high (`1'b1`).

Table 12: ***Number of Multipliers Per Input Width***

Input Width	Maximum Number of Multipliers
3	8
4	8
5	4
6	4
7	4
8	4
16	2

Table 13: ***Ports***

Name	Direction	Description
`i_clk`	Input	Clock input to drive the (optional) registers and accumulator.
`i_din_a`[(Input Width – 1):0]	Input	Packed (see page 35) vector of data to 'A' inputs of multipliers.
`i_din_b`[(Input Width – 1):0]	Input	Packed (see page 35) vector of data to 'B' inputs of multipliers.
`i_in_reg_a_ce`	Input	(Optional) Clock enable for `i_din_a`. Present when **Enable Input Registers** is set to **On**.
`i_in_reg_b_ce`	Input	(Optional) Clock enable for `i_din_b`. Present when **Enable Input Registers** is set to **On**.
`i_in_reg_rstn`	Input	(Optional) Synchronous active-low reset for input registers. Present when **Enable Input Registers** is set to **On**.
`i_pipeline_ce`	Input	(Optional) Clock enable for pipeline registers. Present when **Pipeline Register Stages** is set to **1**.
`i_pipeline_rstn`	Input	(Optional) Synchronous active-low reset for pipeline registers. Present when **Pipeline Register Stages** is set to **1**.
`o_dout`[(Output Width – 1):0]	Output	Output bus consisting of the results from all the multipliers in parallel. Output Width is dynamically calculated by the configurator. See Output Format (see page 35) for details of how the results are assembled in the single output bus.

Input Format

Each multiplier input is formed from an array of the individual inputs packed in a single input vector:

Code

```
i_din_a/b(i) = i_din_a/b[i * int_size +: int_size];
```

Output Format

For each multiplier, the result width in bits is equal to 2 × **Input Width**.

The results from all the parallel multiplications are output as a concatenation on the `o_dout` output. The width of this output is calculated as follows:

Output Width = **Number of Parallel Multiplications** × 2 × **Input Width**.

The bit lanes used for the result of an individual multiplier are found by multiplying the number of the multiplier (starting at 0) by the result width.

Example

If four 8 × 8 multipliers are configured:

Result Width = 2 × 8 = 16 bits

Output Width = 4 × 16 = 64 bits

Each individual multiplier result appears in the lanes detailed in the table below.

Table 14: *Output Bus Organization*

Multiplier Number	Result
0	`o_dout[15:0]`
1	`o_dout[31:16]`
2	`o_dout[47:32]`
3	`o_dout[63:48]`

Examples

The following example shows the integer parallel multiplier configured for two signed 16 × 16 multiplications, with input registers and and an internal pipeline stage:

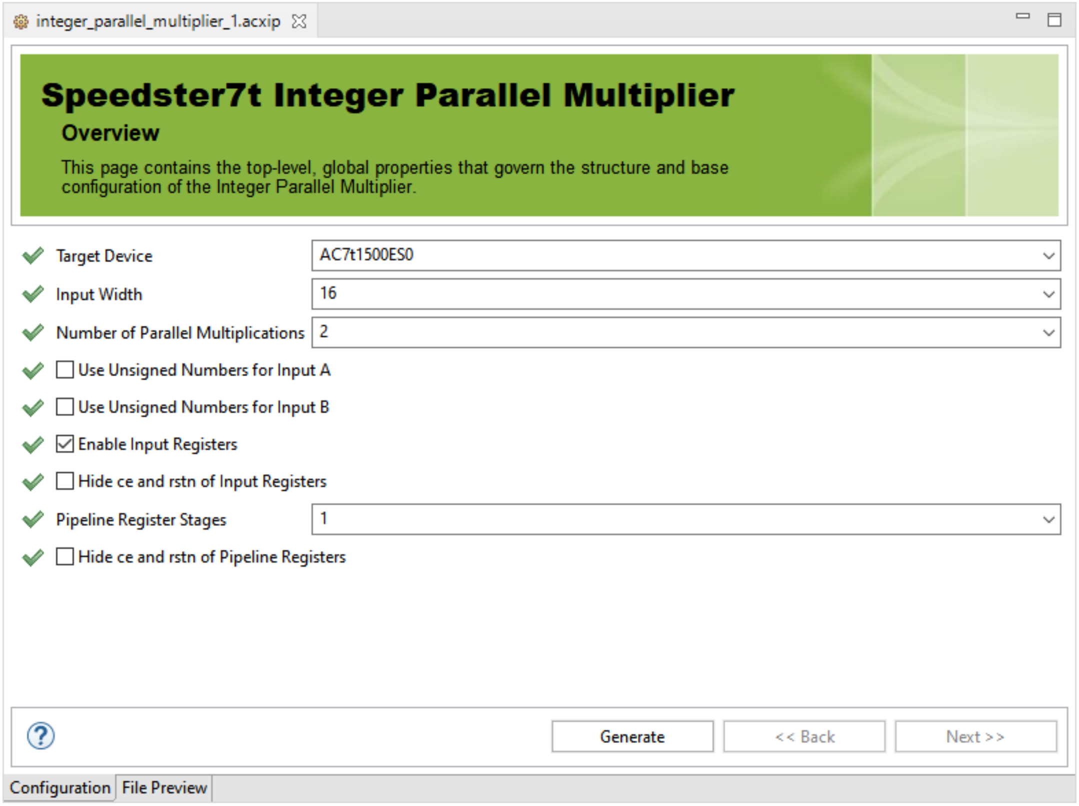

Figure 18: ***Two 16 × 16 Signed Parallel Integer Multiplier Configuration***

The following figure shows the IP diagram for the above configuration:

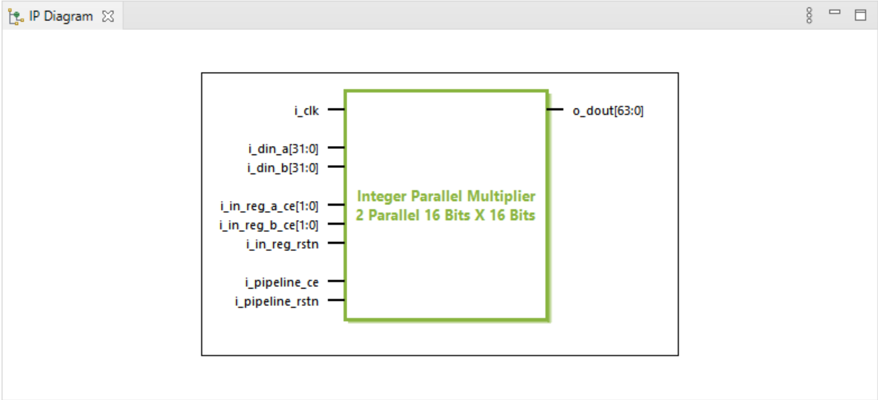

Figure 19: ***Two 16 × 16 Signed Parallel Integer Multiplier I/O***

Chapter - 10: Integer Parallel Sum of Products Soft IP

Description

The Integer Parallel Sum of Products soft IP core configures multiple parallel integer multipliers with a single summed result. This soft IP core is implemented with the MLP primitive which contains an array of integer multipliers and associated adders. Up to 24 parallel multipliers, ranging from 3 × 3 to 16 × 16 bits can be used. The multipliers support both signed and unsigned arithmetic. The final output can optionally be accumulated.

Configuration

The integer parallel sum of products soft IP configurator has the following options:

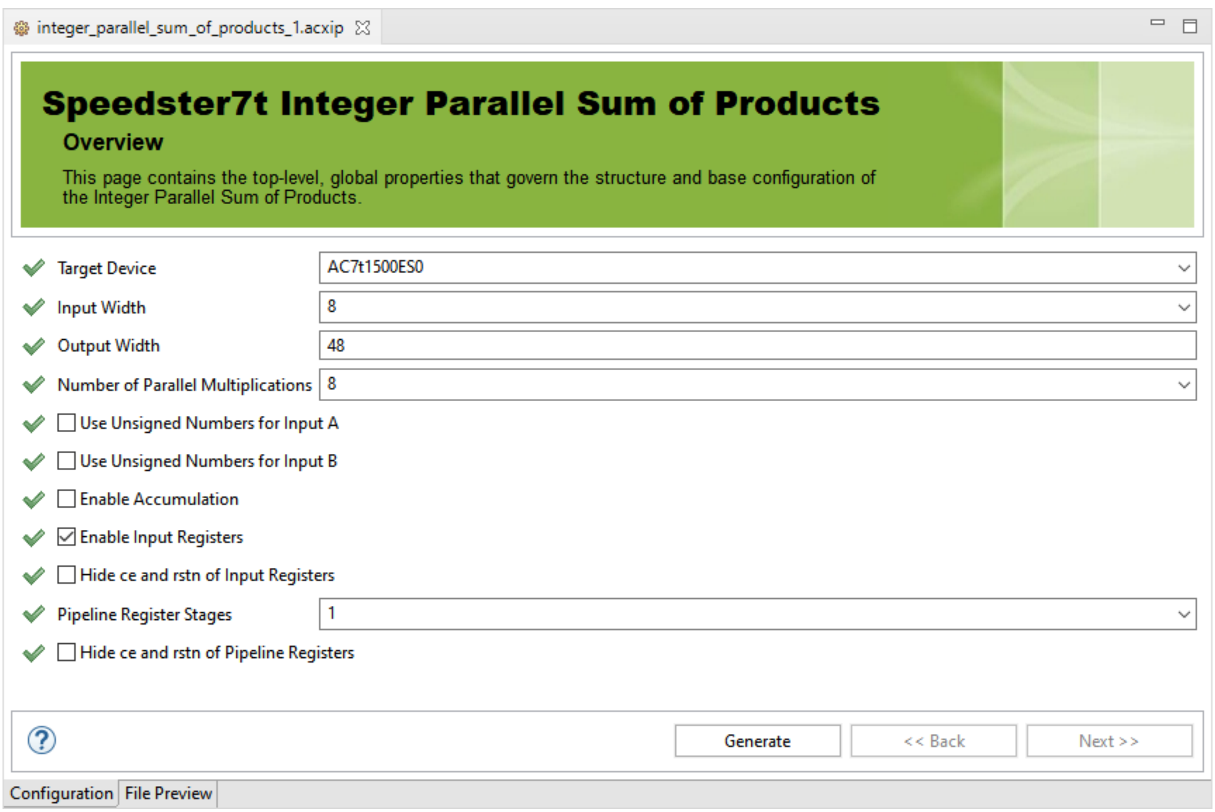

Figure 20: ***Integer Parallel Sum of Products Configurator***

Table 15: *Configuration Options*

Name	Default	Range	Description
Target Device	AC7t1500ES0	All Speedster7t devices	Set to match the target device of the project.
Input Width	8	3, 4, 5, 6, 7, 8 or 16	Width of the two inputs to each multiplier.
Number of Parallel Multiplications	8	1 to 24	The number of parallel multipliers to be implemented and their results summed. The maximum number of multipliers is determined by the Input Width. Refer to the Maximum Number of Multipliers Per Input Width table for details.
Use Unsigned Numbers for Input A/B	Off	On, Off	When set, configures the appropriate inputs to use unsigned numbers. The inputs are signed by default.
Enable Input Registers	Off	On, Off	When set, enables a register stage for all multiplier inputs. This adds a cycle of latency to all results. Enabling the input registers adds these inputs to the resultant soft IP core: • `i_in_reg_a_ce` • `i_in_reg_b_ce` • `i_in_reg_rstn`
Hide ce and rstn of Input Registers	No	Yes, No	If selected, the `i_in_reg_a_ce`, `i_in_reg_b_ce`, and `i_in_reg_rstn` inputs are automatically tied high (`1'b1`).
Enable Accumulator	Off	On, Off	The output is the accumulation of the result from each clock cycle. Enabling accumulation adds the input `i_load` to the resultant soft IP core. The accumulation is cleared when `i_load` is asserted.
Pipeline Register Stages	0	0, 1 or 2	Adds pipeline register stages to the multiplication process. Enabling pipeline register stages improves timing performance at the cost of an additional cycle of latency per stage. When any pipeline stages are enabled, these inputs are added to the resultant soft IP core: • `i_pipeline_ce` • `i_pipeline_rstn`
Hide `ce` and `rstn` of Pipeline Registers	No	Yes, No	If selected, the `i_pipeline_ce` and `i_pipeline_rstn` inputs are automatically tied high (`1'b1`).
Output Width	48	3 to 48	Width of the data output. By default, this value is set to 48 bits. This value can be reduced if required. (1)

Table Notes

1. Ensure that **Output Width** is sufficient to represent the maximum result that can be accumulated. In the event of overflow, the higher order bits of any result are truncated.

   When accumulation is enabled, it might be necessary to increase the data output width to account for the growth in the result over multiple accumulation cycles. The minimum output width can be calculated as:

   (2 × **Input Width** × **Number of Parallel Multiplications**) + (number of accumulation cycles)

Table 16: ***Maximum Number of Multipliers Per Input Width***

Input Width	Maximum Number of Multipliers
3	24
4	16
5	12
6	12
7	10
8	8
16	4

Table 17: *Ports*

Name	Direction	Description
i_clk	Input	Clock input, used for the (optional) registers and accumulator.
i_din_a[(*data width* – 1):0]	Input	Packed (see page 40) data vector to the 'A' inputs of the multipliers where *data width* = **Input Width** × **Number of Parallel Multiplications**.
i_din_b[(*data width* – 1):0]	Input	Packed (see page 40) data vector to the 'B' inputs of the multipliers where *data width* = **Input Width** × **Number of Parallel Multiplications**.
i_in_reg_a_ce	Input	(Optional) Clock enable for i_din_a. Present when **Enable Input Registers** is set to **On**.
i_in_reg_b_ce	Input	(Optional) Clock enable for i_din_b. Present when **Enable Input Registers** is set to **On**.
i_in_reg_rstn	Input	(Optional) Synchronous active-low reset for input registers. Present when **Enable Input Registers** is set to **On**.
i_pipeline_ce	Input	(Optional) Clock enable for pipeline registers. Present when **Pipeline Register Stages** is greater than 0.
i_pipeline_rstn	Input	(Optional) Synchronous active-low reset for pipeline registers. Present when **Pipeline Register Stages** is greater than 0.
i_load[(1)]	Input	(Optional) When asserted to 1'b1, resets the accumulator to i_din_a × i_din_b, ignoring the previous value. Present when **Enable Accumulator** is set to On.
o_dout[(Output Width – 1):0]	Output	Output bus consisting of the sum of products from all of the multipliers in parallel.

Table Notes

1. This signal is internally pipelined to have the same latency as i_din_a and i_din_b.

Input Format

Each multiplier input is formed from an array of the individual inputs, packed in a single input vector.

Code

```
i_din_a / b(i) = i_din_a / b[i*int_size +: int_size];
```

Examples

The following example shows the integer parallel sum of products configured for four signed 16 × 16 multiplications with input registers, accumulation and a single internal pipeline stage.

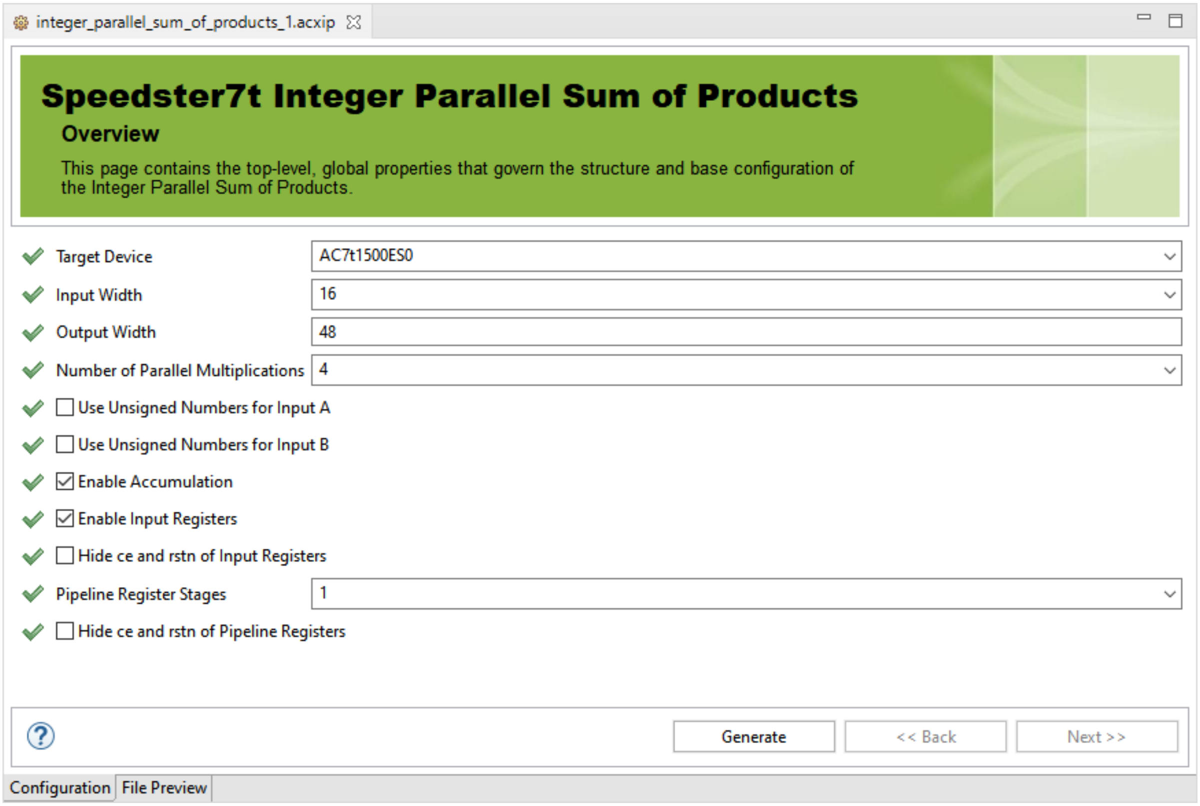

Figure 21: ***Four 16 × 16 Signed Multiplier Sum of Products Configuration***

The following figure shows the IP diagram for the above configuration:

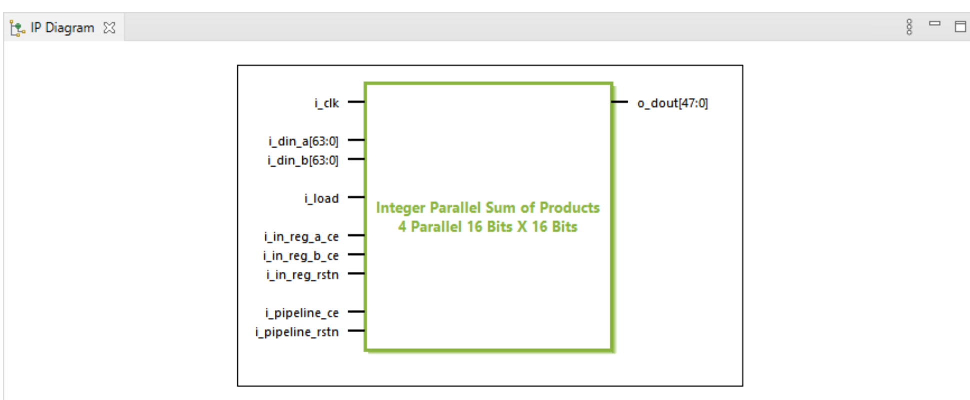

Figure 22: ***Four 16 × 16 Signed Multiplier Sum of Products I/O***

Chapter - 11: Integer Parallel Sum of Squares Soft IP

Description

The Integer Parallel Sum of Squares soft IP core configures multiple parallel integer square (n^2) multipliers with a single summed result. This soft IP core is implemented with the MLP primitive which contains an array of integer multipliers and associated adders. Up to 32 parallel multipliers, ranging from 3 × 3 to 16 × 16 bits can be configured. The multipliers support both signed and unsigned arithmetic. The final output can optionally be accumulated.

Configuration

The integer parallel sum of squares soft IP configurator has the following options:

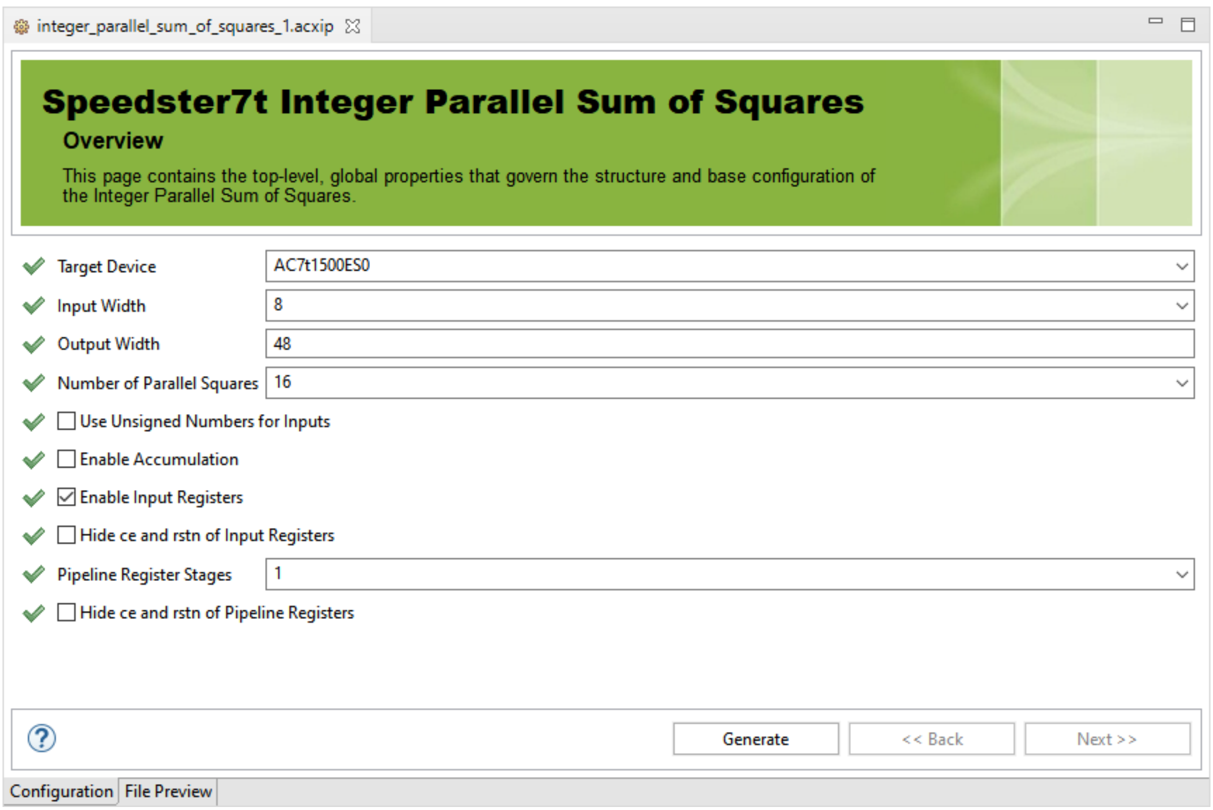

Figure 23: ***Integer Parallel Sum of Squares Configurator***

Table 18: *Configuration Options*

Name	Default	Range	Description
Target Device	AC7t1500ES0	All Speedster7t devices	Set to match the target device of the project.
Input Width	8	3, 4, 5, 6, 7, 8 or 16	Width of the input to be squared.
Number of Parallel Squares	8	1 to 32	The number of parallel squaring multipliers to be implemented and their results summed. The maximum number of multipliers is determined by the Input Width. Refer to the Maximum Number of Multipliers Per Input Width table, below, for details.
Use Unsigned Numbers for Input	Off	On, Off	When set, configures the input to use unsigned numbers. The input is signed by default.
Enable Input Registers	Off	On, Off	When set, enables a register stage for the input. Adds a cycle of latency to all results. Enabling the input registers adds these inputs to the resultant soft IP core: • `i_in_reg_ce` • `i_in_reg_rstn`
Hide `ce` and `rstn` of Input Registers	No	Yes, No	If selected, the `i_in_reg_ce` and `i_in_reg_rstn` inputs are automatically tied high (`1'b1`).
Enable Accumulator	Off	On, Off	Output is the accumulation of the result from each clock cycle. Enabling accumulation adds these inputs to the resultant soft IP core: • `i_load` • `i_pipeline_ce` • `i_pipeline_rstn` The accumulation is cleared when `i_load` is asserted.
Pipeline Register Stages	0	0, 1 or 2	Adds pipeline register stages to the multiplication process. Enabling pipeline register stages improves timing performance at the cost of an additional cycle of latency per stage. When any pipeline stages are enabled, these inputs are added to the resultant soft IP core: • `i_pipeline_ce` • `i_pipeline_rstn`
Hide `ce` and `rstn` of Pipeline Registers	No	Yes, No	If selected, the `i_pipeline_ce` and `i_pipeline_rstn` inputs are automatically tied high (`1'b1`).
Output Width	48	3 to 48	Width of the data output. By default, this value is set to 48 bits. This value can be reduced if required. (1)

Table Notes

1. Ensure that **Output Width** is sufficient to represent the maximum result that can be accumulated. In the event of overflow, the higher order bits of any result are truncated.

   When accumulation is enabled, it might be necessary to increase the data output width to account for the growth in the result over multiple accumulation cycles. The minimum output width can be calculated as:

   (2 × **Input Width** × **Number of Parallel Squares**) + (number of accumulation cycles)

Table 19: ***Maximum Number of Multipliers Per Input Width***

Input Width	Maximum Number of Multipliers
3	32
4	32
5	16
6	16
7	16
8	16
16	4

Table 20: *Ports*

Name	Direction	Description
`i_clk`	Input	Clock input to drive the (optional) registers and accumulator.
`i_din`[(*data width* – 1):0]	Input	Packed (see page 45) vector of data input to the squaring multipliers where *data width* = **Input Width** × **Number of Parallel Squares**.
`i_in_reg_ce`	Input	(Optional) Clock enable for `i_din`. Present when **Enable Input Registers** is set to **On**.
`i_in_reg_rstn`	Input	(Optional) Synchronous active-low reset for input register. Present when **Enable Input Registers** is set to **On**.
`i_pipeline_ce`	Input	(Optional) Clock enable for pipeline registers. Present when **Pipeline Register Stages** is greater than 0.
`i_pipeline_rstn`	Input	(Optional) Synchronous active-low reset for pipeline registers. Present when **Pipeline Register Stages** is greater than 0.
`i_load`(1)	Input	(Optional) When asserted to `1'b1`, resets the accumulator to `i_din2` ignoring the previous value. Present when **Enable Accumulator** is set to **On**.
`o_dout`[(Output Width – 1):0]	Output	Output bus consisting of the sum of squares from all of the multipliers in parallel.

Table Notes

1. This signal is internally pipelined to have the same latency as `i_din`.

Input Packing

Inputs are packed in a single input vector.

Code

```
din(i) = i_din[i * int_size +: int_size];
```

Examples

The following example shows the integer parallel sum of squares configured for four signed 16 bit inputs, with input registers, accumulation and a single internal pipeline stage:

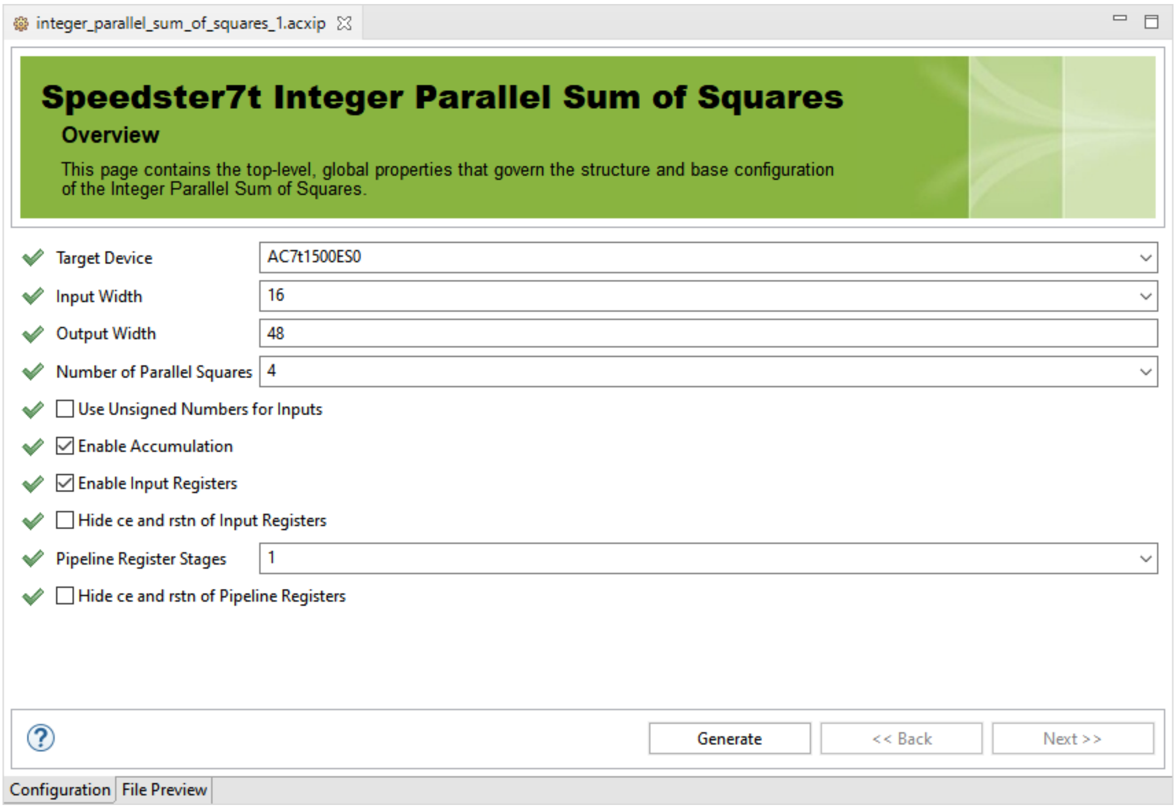

Figure 24: ***Four 16 × 16 Signed Multiplier Sum of Squares Configuration***

The following figure shows the IP diagram for the above configuration:

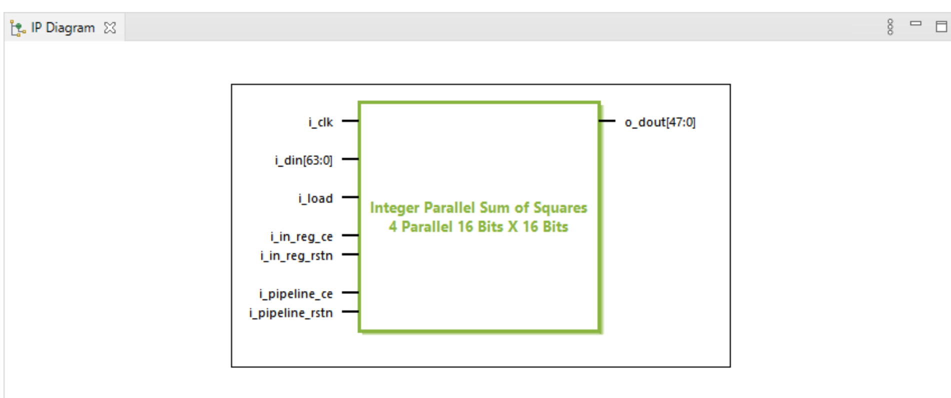

Figure 25: ***Four 16 × 16 Signed Multiplier Sum of Squares I/O***

Chapter - 12: Integer RLB Multiplier Soft IP

Description

The Integer RLB Multiplier soft IP core configures a two-input integer multiplier using RLB (logic) based multipliers. The multiplier can also support optional result accumulation. The configurator supports sizes of up to 9 × 9, with signed inputs and outputs.

Configuration

The integer RLB multiplier soft IP configurator has the following options:

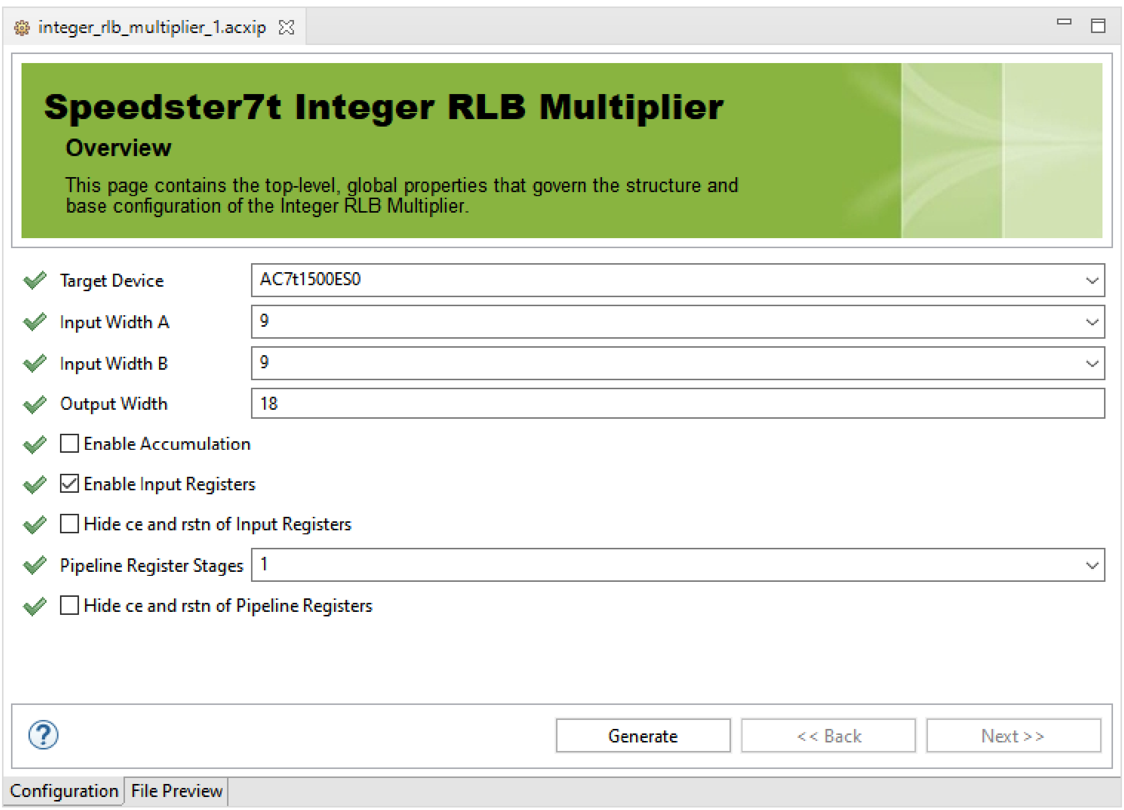

Figure 26: *Integer RLB Multiplier Soft IP Configurator*

Table 21: *Configuration Options*

Name	Default	Range	Description
Target Device	AC7t1500ES0	All Speedster7t devices	Set to match the target device of the project.
Input Width A	9	3 to 9	Width of 'A' inputs.
Input Width B	9	3 to 9	Width of 'B' inputs.
Enable Input Registers	Off	On, Off	When set, enables a register stage for both 'A' and 'B' inputs. Adds a cycle of latency to all results. Enabling the input registers adds these inputs to the resultant soft IP core: • `i_in_reg_a_ce` • `i_in_reg_b_ce` • `i_in_reg_rstn`
Hide `ce` and `rstn` of Input Registers	No	Yes, No	If selected, the `i_in_reg_a_ce`, `i_in_reg_b_ce`, and `i_in_reg_rstn` inputs are automatically tied high (`1'b1`).
Enable Accumulator	Off	On, Off	Output is the accumulation of the result from each clock cycle. Enabling accumulation adds the input `i_load` to the resultant soft IP core. The accumulation is cleared when `i_load` is asserted. The output is reset to `i_din_a` × `i_din_b`.
Pipeline Register Stages	0	0, 1, 2	Adds pipeline register stages through the multiplication process. Enabling pipeline registers improves timing performance at the cost of an additional cycle of latency for each stage enabled. When any pipeline stages are enabled, these inputs are added to the resultant soft IP core: • `i_pipeline_ce` • `i_pipeline_rstn`
Hide `ce` and `rstn` of Pipeline Registers	No	Yes, No	If selected, the `i_pipeline_ce` and `i_pipeline_rstn` inputs are automatically tied high (`1'b1`).
Output Width	18	2 to 48	Width of the data output. Automatically updated by the configurator when **Input Width A/B** is updated. In addition, the value can be modified to match requirements. (1)

Table Notes

1. When accumulation is enabled, it might be necessary to increase the data output width to account for the growth in the result over multiple accumulation cycles. The minimum output width can be calculated as:
   (2 × **Input Width**) + (number of accumulation cycles)

Table 22: *Ports*

Name	Direction	Description
i_clk	Input	Clock input to drive the (optional) registers and accumulator.
i_din_a[(Input Width A – 1):0]	Input	'A' data input to the multiplier.
i_din_b[(Input Width B – 1):0]	Input	'B' data input to the multiplier.
i_in_reg_a_ce	Input	(Optional) Clock enable for i_din_a. Present when **Enable Input Registers** is set to **On**.
i_in_reg_b_ce	Input	(Optional) Clock enable for i_din_b. Present when **Enable Input Registers** is set to **On**.
i_in_reg_rstn	Input	(Optional) Synchronous active-low reset for the input registers. Present when **Enable Input Registers** is set to **On**.
i_pipeline_ce	Input	(Optional) Clock enable for the pipeline and accumulator registers. Present when **Pipeline Register Stages** is greater than 0.
i_pipeline_rstn	Input	(Optional) Synchronous active-low reset for the pipeline and accumulator registers. Present when **Pipeline Register Stages** is greater than 0.
i_load[1]	Input	(Optional) When asserted to 1'b1, resets the accumulator to i_din_a × i_din_b ignoring the previous value. Present when **Enable Accumulator** is set to **On**.
o_dout[(Output Width – 1):0]	Output	Result of multiplication and accumulation.

Table Notes

1. This signal is internally pipelined to have the same latency as i_din_a and i_din_b.

Examples

The following example shows the integer multiplier configured for signed 9 × 9 inputs with input registers, accumulation and a single pipeline stage:

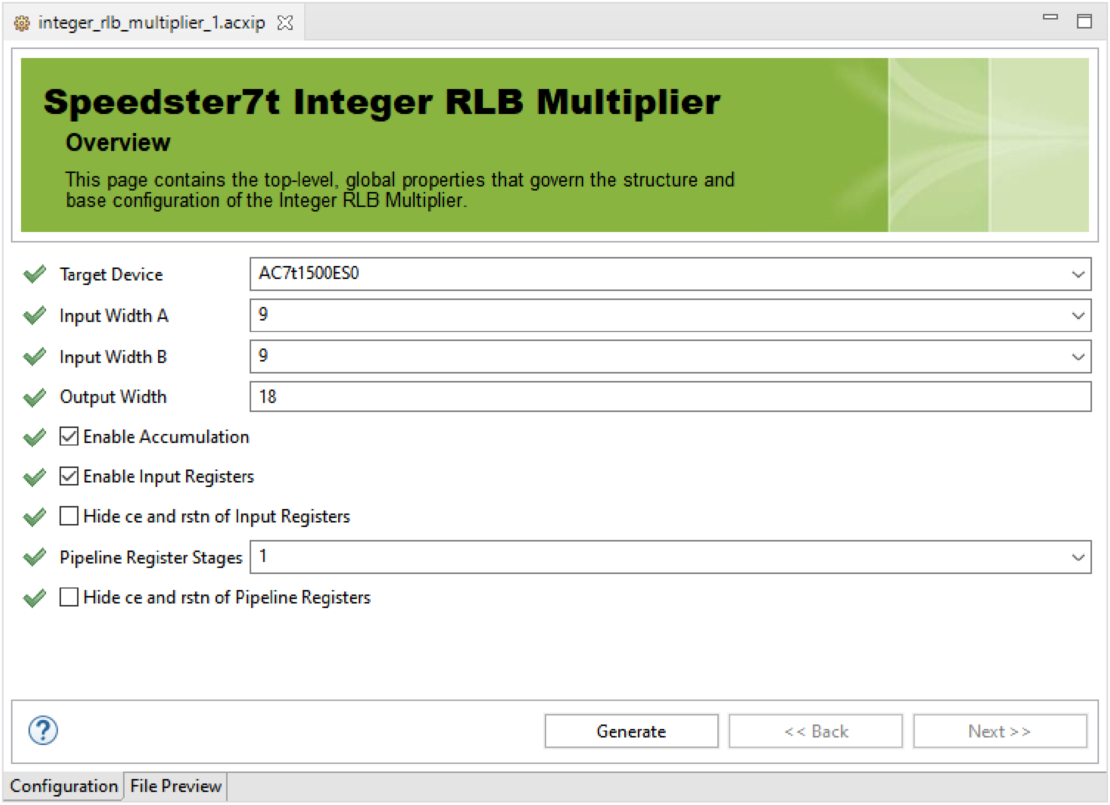

Figure 27: ***9 × 9 Signed Integer RL Multiplier Configuration***

The following figure shows the IP diagram for the above configuration:

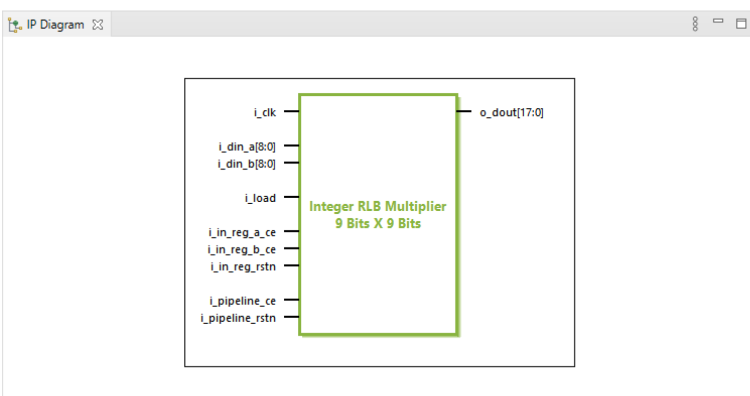

Figure 28: ***9 × 9 Signed Integer RLB Multiplier I/O***

Chapter - 13: Shift Register Soft IP

Description

The Shift Register soft IP core implements a shift register using the fabric flip-flops. The soft IP core can be configured to support varying widths and depths of shift functions. The generated shift register includes a clock enable and reset.

Configuration

The shift register soft IP configurator has the following options:

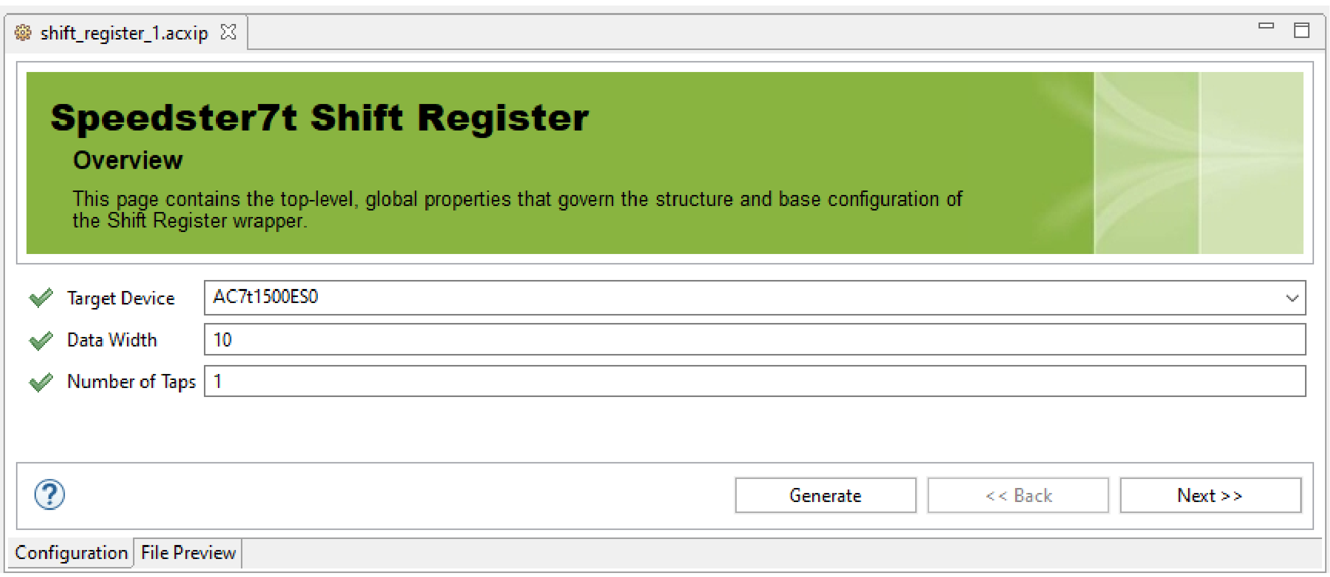

Figure 29: ***Shift Register Configurator***

Table 23: *Configuration Options*

Name	Default	Range	Description
Target Device	AC7t1500ES0	All Speedster7t devices	Set to match the target device of the project.
Data Width	10	1 to 65,536	Width of the input and output data bus.
Number of Taps	1	1 to 256	Number of taps in the shift register. All intermediate taps are output from the soft IP.

Note

The number of flip-flops used is calculated by:

Data Width × Number of Taps

For very wide or deep shift registers, a large number of flip-flops may be used which might result in sub-optimal timing closure. In these circumstances, it is recommended to use a BRAM or LRAM to implement the shift register function. However, it should be noted that a BRAM or LRAM implemented shift register only provides access to the end tap of the shift register, and not the intermediate stages.

Table 24: *Ports*

Name	Direction	Description
`clk`	Input	Clock input to each flip-flop.
`din`[(Data Width – 1):0]	Input	Input data bus.
`en`	Input	Clock enable: `en` = `1'b0`: Shift register is stopped. No data is input. Current output is maintained. `en` = `1'b1`: Shift register transfers data from tap to tap on each rising edge of `clk`.
[(Number of Taps – 1):0] `dout`[(Data Width – 1):0]	Output	Output data bus and intermediate taps. The final tap of the shift register (the input delayed by **Number of Taps** clock cycles) is output on [(Number of Taps – 1):0] `dout`.

Examples

The following example shows the shift register configured for 72 bits wide, by 5 stages deep:

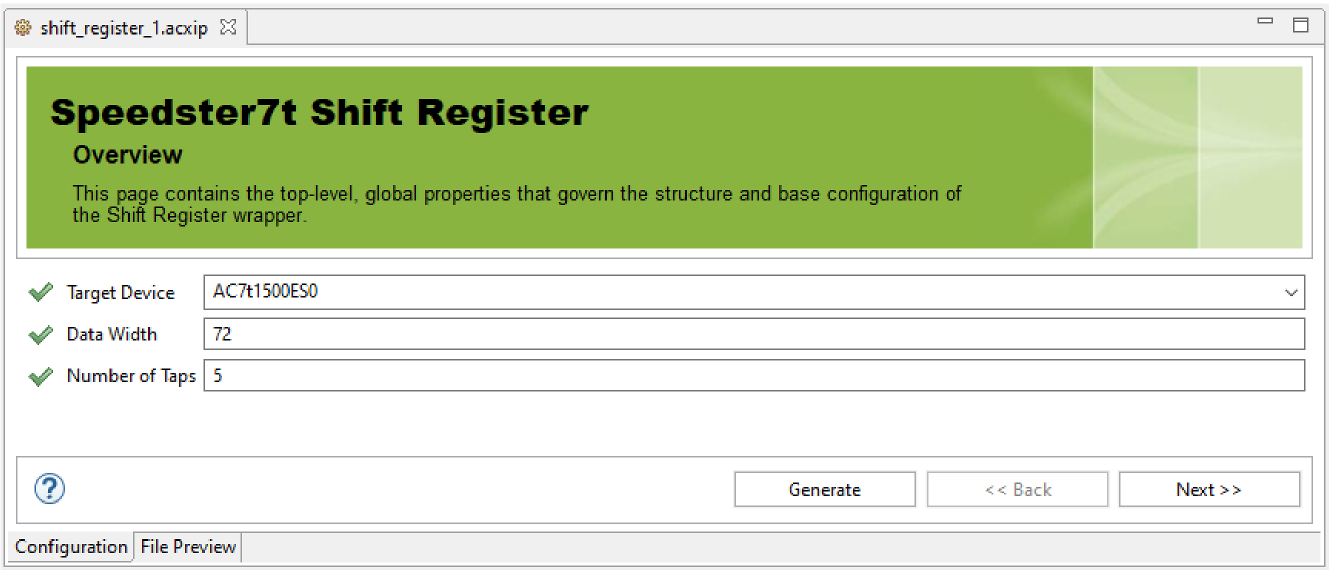

Figure 30: ***72 Bit, 5 Stage Shift Register Configuration***

The following figure shows the IP diagram for the above configuration:

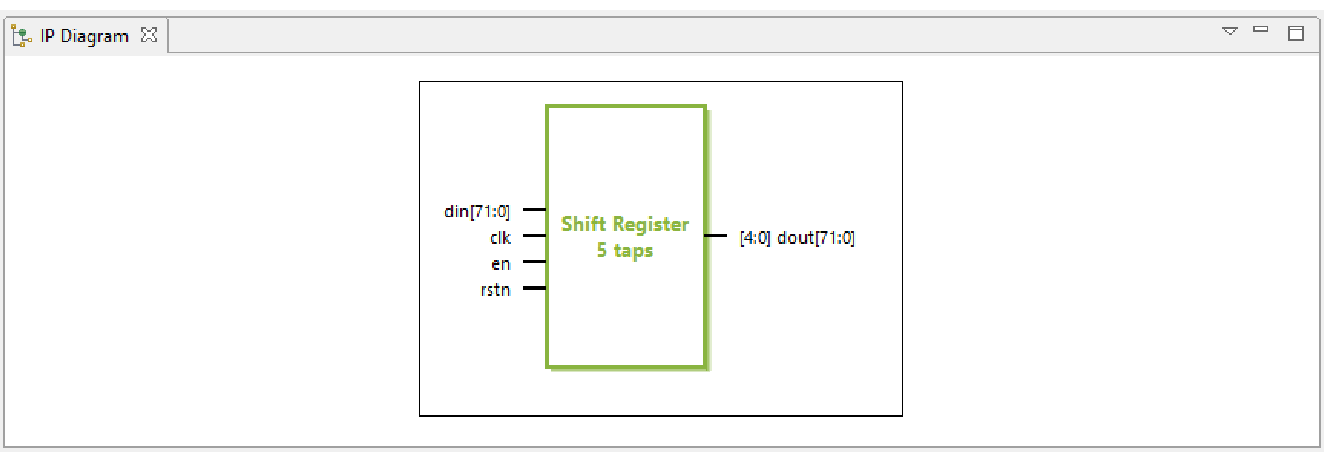

Figure 31: ***72 bit, 5 stage Shift Register I/O***

Chapter - 14: Speedster7t AXI Initiator NAP

Description

The AXI Initiator NAP soft IP core implements a NoC access point (NAP) that connects to user logic in the fabric, which responds to read and write AXI transactions from the 2D NoC. The soft IP configurator allows choosing the NAP location and setting the south-to-north arbitration weight for the NAP. The AXI Initiator NAP is comprised of the ACX_NAP_AXI_MASTER primitive. More details on the primitive can be found in the ACX_NAP_AXI_MASTER page of the *Speedster7t Component Library User Guide* (UG086).

Configuration

The AXI Initiator NAP soft IP configurator has the following options:

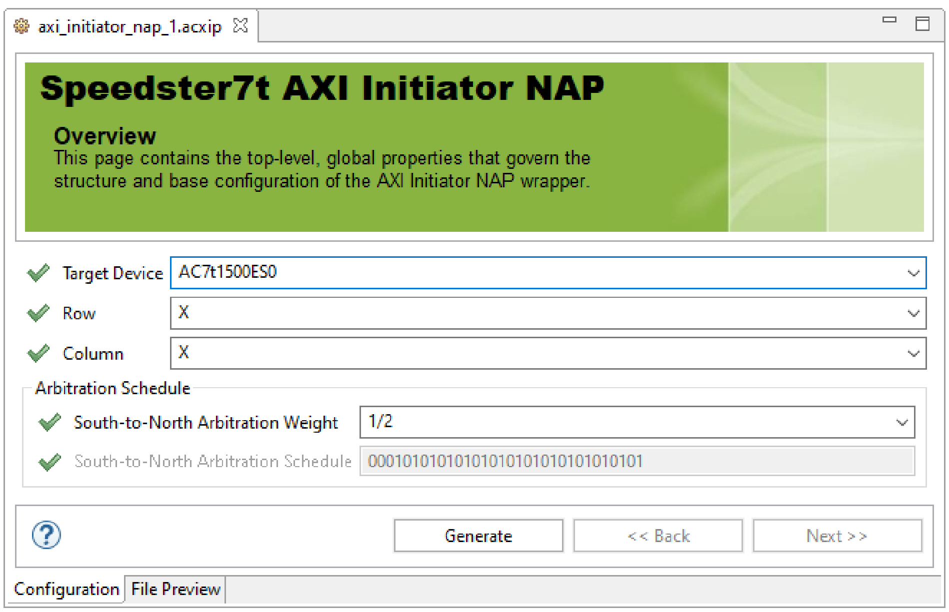

Figure 32: *AXI Initiator NAP*

Table 25: *Configuration Options*

Name	Default	Range	Description
Target Device	AC7t1500ES0	All Speedster7t devices	Set to match the target device of the project.
Row	X	1 to 8	Set to match the row number location of the NAP.
Column	X	1 to 10	Set to match the column number location of the NAP.
South-to-north arbitration weight	1/2	0, 1/2, 1/3, ... , 1/10, 1, custom	Sets the fraction of remaining bandwidth this NAP is guaranteed in the south-to-north direction. "Remaining bandwidth" is the bandwidth not guaranteed to the downstream NAPs on the same column.
South-to-north arbitration schedule	00010101010101010101010101010101	00000000000000000000000000000000 to 11111111111111111111111111111111	A 32-bit number that allows the specific setting of a custom arbitration schedule in the south-to-north direction, where "1" means this NAP gets priority and "0" means the downstream NAPs get priority. Only available when "custom" is chosen for the arbitration weight.

Files

The above configuration settings prompt the generation of an AXI Initiator NAP design file in the selected location as shown below:

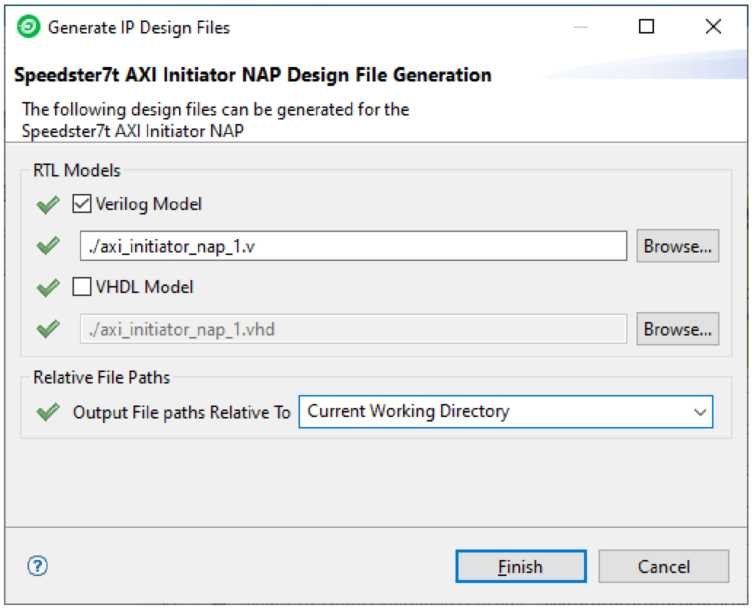

Figure 33: ***AXI Initiator NAP File Generation***

Examples

Below is an example configuration of the AXI Initiator NAP located in Row 6, Column 3, using an arbitration weight of 1/2:

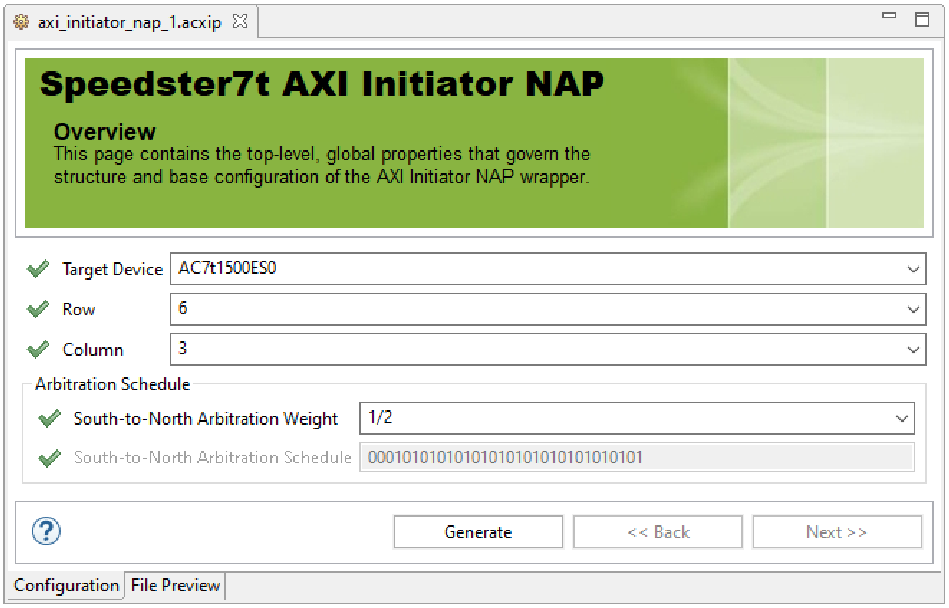

Figure 34: ***AXI Initiator NAP in Row 6, Column 3***

The IP diagram of the above configuration follows:

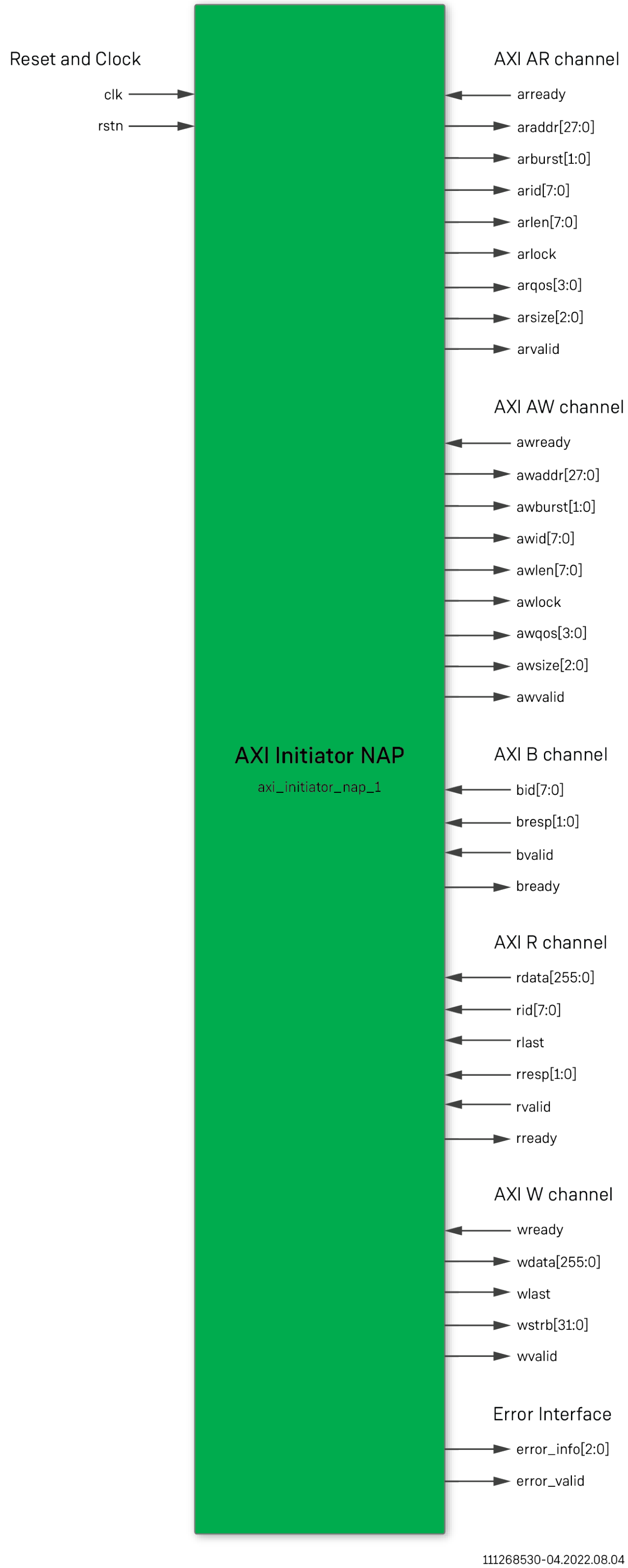

Figure 35: ***AXI Initiator NAP IP Diagram***

Chapter - 15: Speedster7t AXI Responder NAP

Description

The AXI Responder NAP soft IP core implements a NoC access point (NAP) that connects to user logic in the fabric, which initiates read and write AXI transactions to the 2D NoC. The soft IP allows choosing the NAP location and setting the east-to-west and west-to-east arbitration weights for the NAP. The AXI Responder NAP is comprised of the ACX_NAP_AXI_SLAVE primitive. More details on the primitive can be found in the ACX_NAP_AXI_SLAVE page of the *Speedster7t Component Library User Guide* (UG086).

Configuration

The AXI Responder NAP soft IP configuration GUI has the following options:

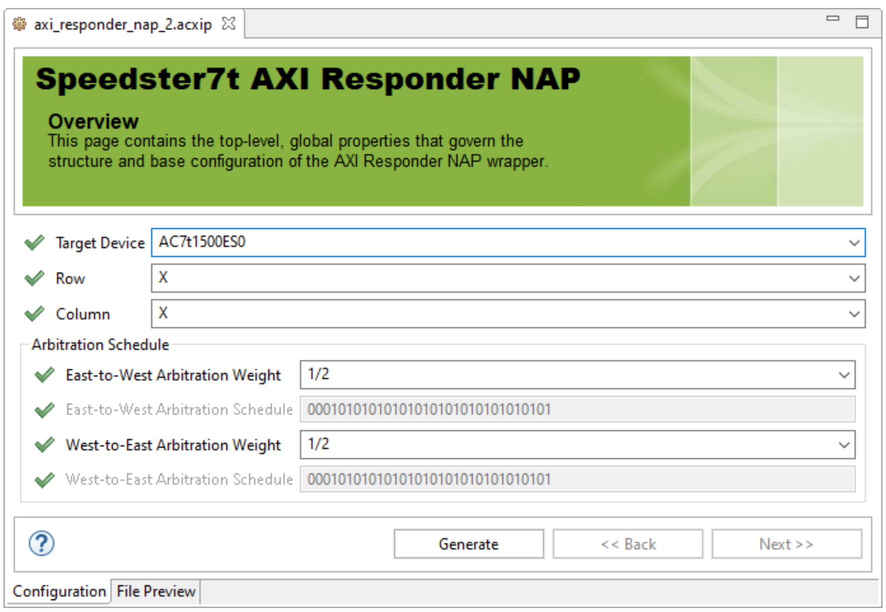

Figure 36: ***AXI Responder NAP Configuration***

Table 26: *Configuration Options*

Name	Default	Range	Description
Target Device	AC7t1500ES0	All Speedster7t devices	Set to match the target device of the project.
Row	X	1 to 8	Set to match the row number location of the NAP.
Column	X	1 to 10	Set to match the column number location of the NAP.
East-to-west arbitration weight	1/2	0, 1/2, 1/3, ... , 1/10, 1, custom	Sets the fraction of remaining bandwidth this NAP is guaranteed in the east-to-west direction. "Remaining bandwidth" is the bandwidth not guaranteed to the downstream NAPs on the same row.
East-to-west arbitration schedule	00010101010101010101010101010101	00000000000000000000000000000000 to 11111111111111111111111111111111	A 32-bit number that allows the specific setting of a custom arbitration schedule in the east-to-west direction, where "1" means this NAP gets priority and "0" means the downstream NAPs get priority. Only available when "custom" is chosen for the arbitration weight.
West-to-east arbitration weight	1/2	0, 1/2, 1/3, ... , 1/10, 1, custom	Sets the fraction of remaining bandwidth this NAP is guaranteed in the west-to-east direction. "Remaining bandwidth" is the bandwidth not guaranteed to the downstream NAPs on the same row.
West-to-east arbitration schedule	00010101010101010101010101010101	00000000000000000000000000000000 to 11111111111111111111111111111111	A 32-bit number that allows the specific setting of a custom arbitration schedule in the west-to-east direction, where "1" means this NAP gets priority and "0" means the downstream NAPs get priority. Only available when "custom" is chosen for the arbitration weight.

Files

The above configuration settings prompt the generation of an AXI Initiator NAP design file in the selected location as shown below:

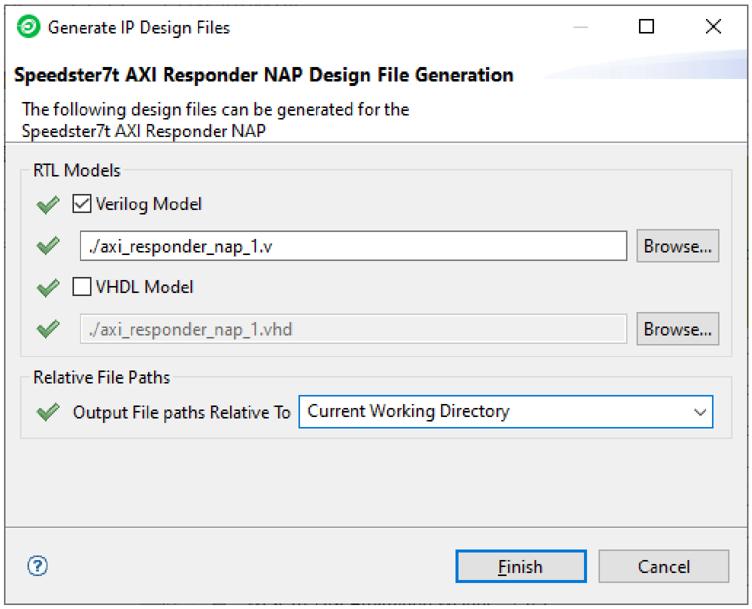

Figure 37: ***AXI Responder NAP File Generation***

Examples

An example configuration of the AXI Responder NAP located in Row 8, Column 2, using a 1/3 east-to-west and 1/4 west-to-east arbitration weight is shown below:

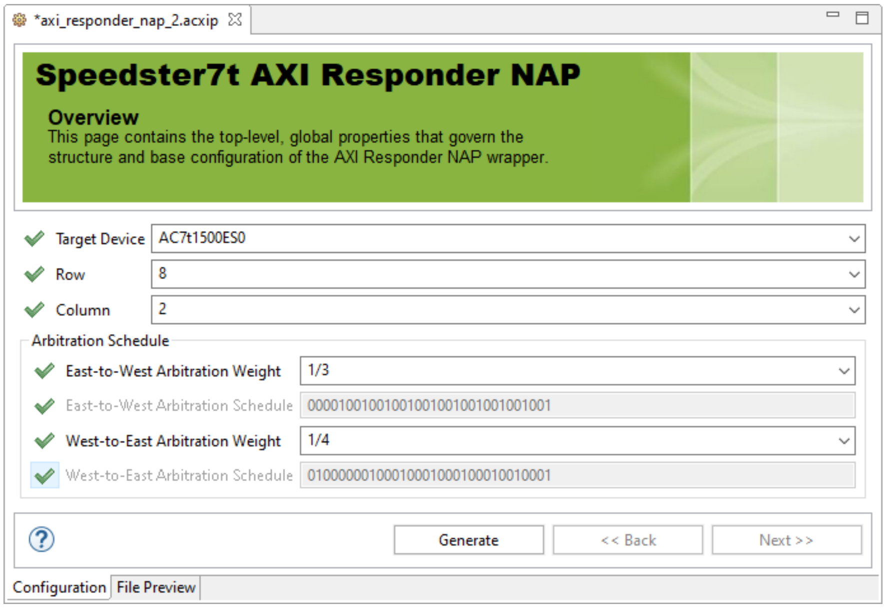

Figure 38: ***AXI Responder NAP Configuration***

The IP diagram of the above configuration follows:

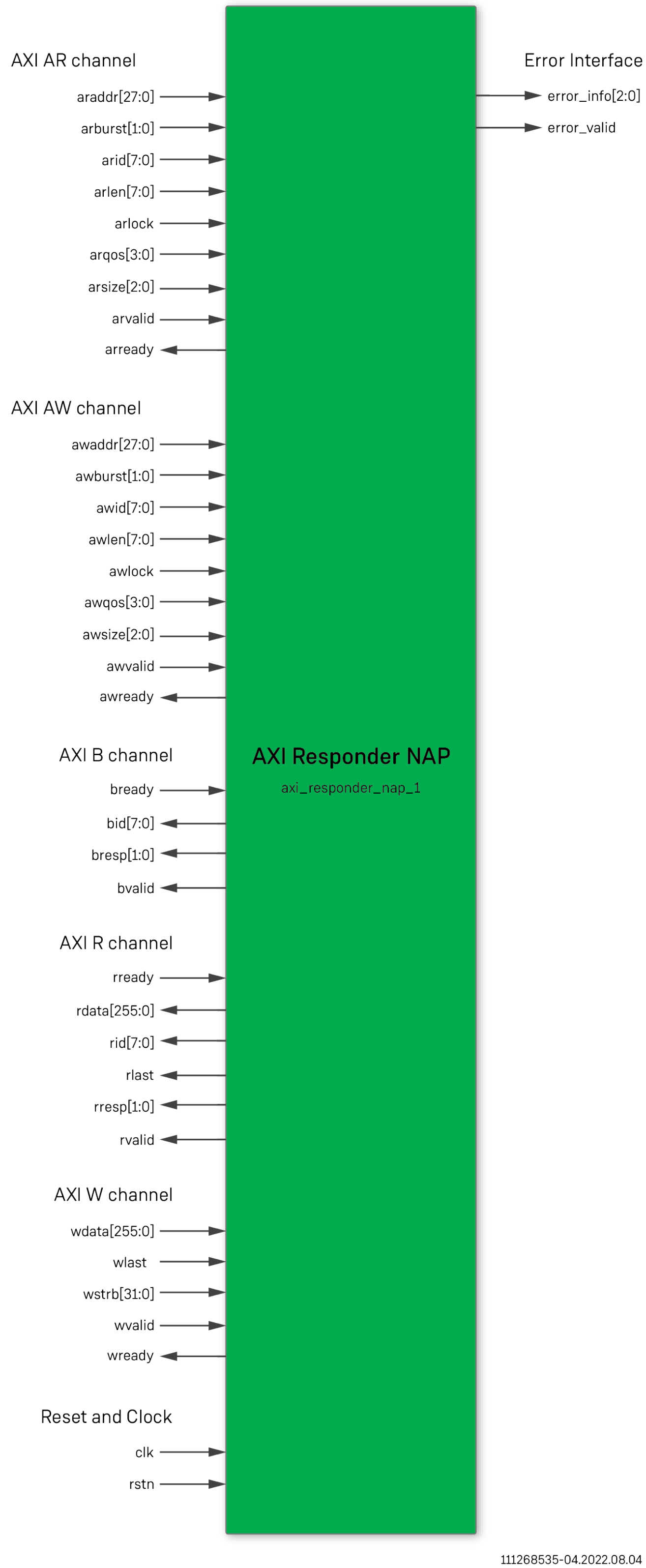

Figure 39: ***AXI Responder NAP IP Diagram***

Chapter - 16: Speedster7t Device Manager

Description

The ACX_DEVICE_MANAGER component can provide automatic control of the device IP components such as GDDR6 and DDR4, where the hardened control is complex for typical production systems. The device manager contains a soft MCU (micro-controller unit), which is embedded in the FPGA core and performs the management funtions. In addition, the device manager acts as a bridge between the JTAG interface and the NoC, allowing access to CSR registers with Tcl commands. The ACX_DEVICE_MANAGER itself can also be queried via the JTAG interface, using commands in ACE. A block diagram of the ACX_DEVICE_MANAGER is shown below.

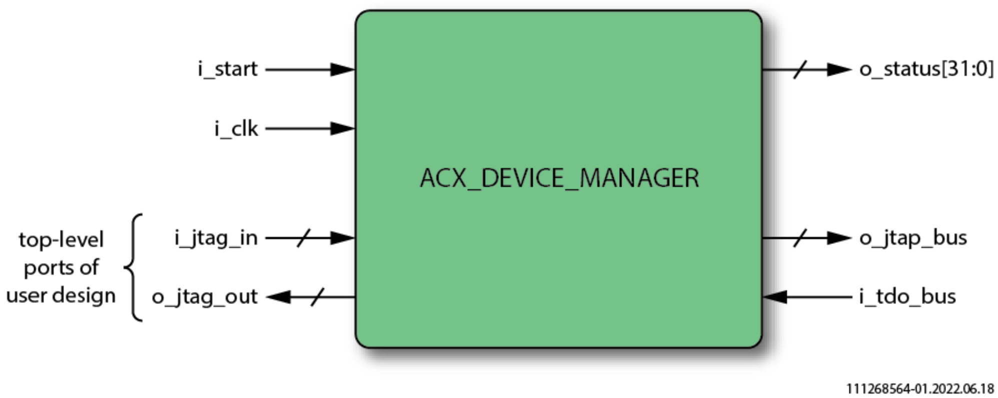

Figure 40: *ACX_DEVICE_MANAGER Block Diagram*

The ACX_DEVICE_MANAGER operates in two phases, startup and background. In the startup phase, the device manager performs steps necessary to configure an IP interface (currently only GDDR6). Many interfaces require some form of pre-calibration or link training. During the startup phase, communication to these interfaces is not possible. The user design must wait until the startup phase is complete before accessing the interface. The end of the startup phase is indicated by an output status signal, `o_status`.

Following the startup phase, the device manager enters the background phase. In this phase, the device manager acts as bridge between the JTAG interface and the NoC, and may perform periodic maintenance tasks such as temperature sensor monitoring, etc. The ACX_DEVICE_MANAGER contains an embedded NAP to communicate via the NoC. The core of the ACX_DEVICE_MANAGER is a pre-programmed 32-bit MCU.

Starting with ACE 8.8, the ACX_DEVICE_MANAGER controls the startup and operation of the GDDR6 interface. Hence, any design that utilizes GDDR6 must include an ACX_DEVICE_MANAGER instance. When the device manager instance is integrated into the GDDR6 user design, the GDDR PLL locks can drive the start of the device manager. While at the startup phase, the device manager feeds off of the bitstream in ACE that consists of the desired configuration for the GDDR IP. It is recommended that all user designs include the ACX_DEVICE_MANAGER to support periodic monitoring tasks such as the reading of the temperature sensor. Designs running on Speedster7t AC7t1550 devices must include an ACX_DEVICE_MANAGER if Tcl access to the CSR register is required.

Configuration

The Device Manager soft IP configurator has the following options:

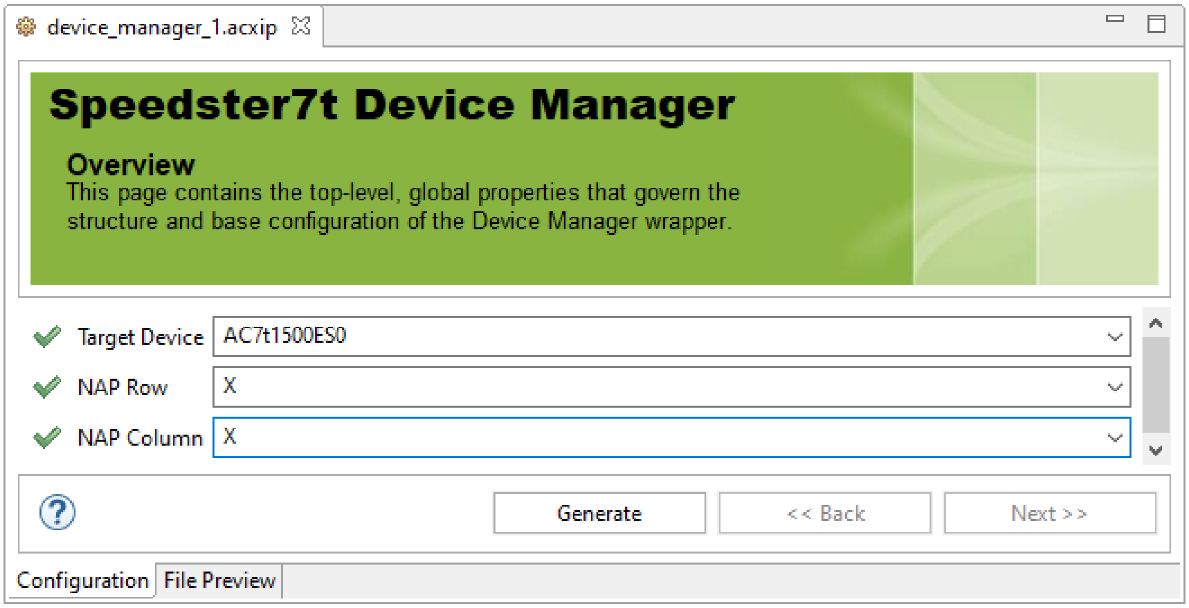

Figure 41: ***Speedster7t Device Manager Soft IP Configuration***

Table 27: ***Speedster7t Device Manager Configuration Options***

Name	Default	Range	Description
Target Device	AC7t1500ES0	All Speedster7t devices	Set to match the target device of the project.
NAP Row (1)	4'bx	1–8	Row coordinate for NAP. (2)
NAP Column (1)	4'bx	1–10	Column coordinate for NAP.

Table Notes

1. The ADM must have the row and column parameters set in the RTL. This differs from for all other IP where it is recommended that the row and column location is set via the `.pdc` file. This requirement results from the fact that the device manager is encrypted. Thus, the path to the row and column parameters is also encrypted.
2. For ES0 devices (AC7t1500ES0, AC7t1550ES0), NAP_ROW must be set from rows 5–8.

File generation

The above configuration settings prompt the generation of a device manager template design file in the selected location as shown below:

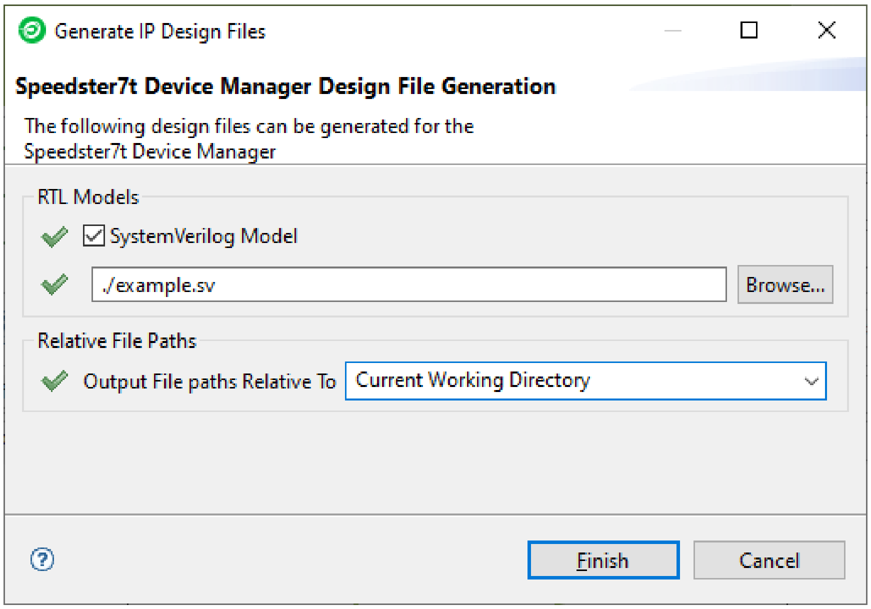

Figure 42: ***Device Manager Design File Generation***

Ports

Table 28: *ACX_DEVICE_MANAGER Ports*

Name	Direction	Description
`i_clk`	Input	Clock input, must be driven with a 100 MHz or slower clock.
`i_start`	Input	Start signal, must be kept high for the ACX_DEVICE_MANAGER to start. It is recommended to simply tie this high (`1'b1`), or to drive it with a PLL lock signal.
`o_status[31:0]`	Output	Ready and error status.
`t_JTAG_INPUT i_jtag_in`	Input	JTAG input: must be connected to a top-level port of the same `t_JTAG_INPUT` type.
`t_JTAG_OUTPUT o_jtag_out`	Output	JTAG output: must be connected to a top-level port of the same `t_JTAG_OUTPUT` type.
`t_JTAP_BUS o_jtap_bus`	Output	JTAP bus output, to share the JTAG interface.
`i_tdo_bus`	Input	JTAP bus input, to share the JTAG interface. When the JTAG interface is not shared, tie this input to `1'b0`.

Status Signals

The o_status output signal shows the status when the startup phase is complete, and indicates if there were any errors during startup. Most designs can only monitor o_status[0] or o_status[1:0] to decide when the design can be started. The o_status value can also be read in ACE Tcl console with the mcu_status command. If errors occur, other ACE commands may give additional information.

Table 29: *ACX_DEVICE_MANAGER Status Output*

Status bits	Meaning when asserted
0	Startup is complete and successful. This signal can be used as the start signal for the user design.
1	Startup is complete. If there were no errors, o_status[0] is set as well; otherwise, an error bit is also set.
3:2	Device manager internal error; These bits should always read 2'b00 to indicate no error. If either bit is asserted, contact Achronix for assistance.
6:4	Reserved for future use. (1)
14:7	GDDR_7...GDDR_0 startup error.
31:15	Reserved for future use. (1)

Table Notes

1. Unused bits are set to 1'b0.

Examples

The following example shows the Device Manager configured for NAP Row 6 and NAP Column 3:

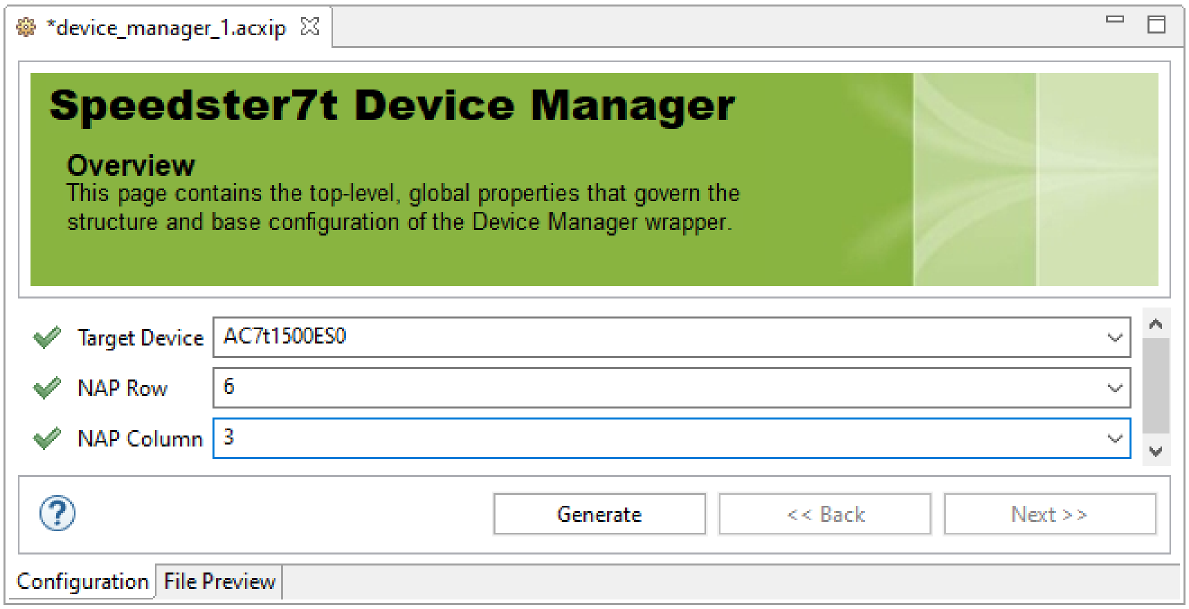

Figure 43: ***Device Manager Configured for NAP Row 6 and NAP Column 3***

The **Generate** option generates a device manager design file in the selected path. In this example, the design file is generated with the name 'example.sv'.

The ACE IP diagram from the above configuration is shown below. The diagram shows the ACE-generated design instance "example":

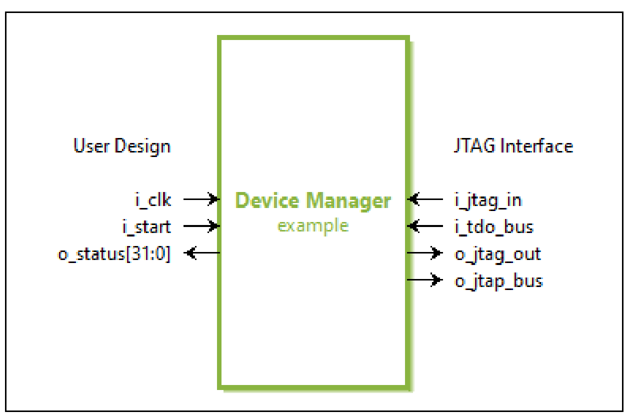

Figure 44: ***Device Manager IP Diagram***

Using the JTAG Interface

JTAG Connection

The ACX_DEVICE_MANAGER connects to the JTAG interface via top-level ports in the design. These ports have a type (i.e., `t_JTAG_INPUT`, `t_JTAG_OUTPUT`) that must be used in the port and wire declarations. Declarations are illustrated in the instantiation example (see page 75) below. The placement of the JTAG interface is built into the macro, and no additional placement statements are required. ACE requires that the JTAG ports are top-level ports of the design. For this reason, it is most convenient to instantiate the ACX_DEVICE_MANAGER at the top-level of the design.

The Speedster7t AC7t1500 FPGA supports two JTAG communication models, depending on whether the design includes an ACX_DEVICE_MANAGER instance. Both models are used in exactly the same manner, but the correct model must be selected with the `use_acx_device_manager` command defined below. This command must be issued before programming the FPGA whenever a design has an ACX_DEVICE_MANAGER instance.

Accessing the ACX_DEVICE_MANAGER with ACE

ACE has several Tcl commands that can be used to access the device manager via the JTAG interface. To enable these commands, use the command `package require mcu`. Before these commands can be used, the JTAG connection must be established, and the device must be programmed. A typical sequence of instructions is shown in the table below:

Table 30: *Device Manager JTAG Interface Tcl Command Sequence Example*

Line	Command	Description
1.	`use_device_namespace AC7t1500`	Some commands, such as `program_hex_file`, are device-specific, and are, therefore, in a special `ac7t1500::` namespace. Hence, the command should be called as: `ac7t1500::program_hex_file` As a convenience, the `use_device_namespace` command imports commands from the namespace, so that the namespace prefix can be omitted.
2.	`use_acx_device_manager`	This command establishes that the ACX_DEVICE_MANAGER is used for JTAG communication. This command must be given before programming the FPGA.
3.	`set_jtag_id`	JTAG commands must know the name of the JTAG device driver. For most commands, this name is assumed to be in the global variable `$jtag_id`. The `set_jtag_id` command finds the name of the device and sets the global variable. If there is more than one device (not common), it lists the possible choices. In such cases, or alternatively, the `jtag_id` variable can be assigned manually.
4.	`open_jtag`	The `open_jtag` command establishes the connection with the JTAG driver. A `close_jtag` command is also provided.
5.	`program_hex_file <design>.hex`	The `program_hex_file` command programs the FPGA via the JTAG interface. The full path name of the hex file must be specified, typically: `<userdesign>/impl_1/output/<design>.hex`
6.	`# use commands, for instance`	The above sequence can be used for any design, independent of the ACX_DEVICE_MANAGER (except for the `use_acx_device_manager` command on line 2).
7.	`package require mcu`	The `package` command loads a set of Tcl macros in ACE. It must be given at least once, but using it multiple times is harmless.
8.	`mcu_status`	This command displays the status of the device manager `o_status` output. See the following section for details on this command.

ACX_DEVICE_MANAGER commands

The following ACE commands interact with the ACX_DEVICE_MANAGER. They are enabled with the Tcl command shown on line 7, above.

- `use_acx_device_manager [-enable] [-disable]`
  This command specifies that the ACX_DEVICE_MANAGER handles JTAG communication with the NoC. It must be specified at least once, before programming the FPGA. If this command is not used, the startup phase of the ACX_DEVICE_MANAGER does not complete (if JTAG is never used, as is typical in a production environment, this command is not required).

Table 31: ***use_acx_device_manager Arguments***

Argument	Optional	Description
`-enable`	Y	Specifies that the ACX_DEVICE_MANAGER handles JTAG communication with the NoC (default).
`-disable`	Y	Specifies legacy mode, where the FCU handles JTAG communication with the NoC. This argument should not be used with the ACX_DEVICE_MANAGER, because it interferes with the startup phase.

- `mcu_status [-wait] [-quiet] [-timeout <seconds>] [-debug]`
  This command prints the value of the ACX_DEVICE_MANAGER o_status output, with interpretation:

Table 32: ***mcu_status Arguments***

Argument	Optional	Description
`-wait`	Y	Waits for the startup phase to complete.
`-quiet`	Y	Instead of printing the result, this argument returns the status value (an integer).
`-timeout int`	Y	Specifies a timeout (in seconds) for the `-wait` option (default is 50 seconds). If a timeout occurs, most likely the `use_acx_device_manager` command was not issued.
`-debug`	Y	Prints additional information that might be useful when filing a support ticket.

- `mcu_info [-quiet]`
  Prints ACX_DEVICE_MANAGER version information.

Table 33: ***mcu_info Arguments***

Argument	Optional	Description
`-quiet`	Y	Instead of printing the result, this argument returns the information as a list.

- `noc_delay`
  Prints the roundtrip delay to access a CSR register. This command confirms that the ACX_DEVICE_MANAGER can communicate with the NoC. However, its main purpose is to print whether the CSR clock is the system clock or the JTAG clock. For the ACX_DEVICE_MANAGER to operate correctly, the clock should be the system clock. If the clock is reported as the JTAG clock, most likely the `use_acx_device_manager` command was omitted. The clock can be switched with a `set_csr_clock` command, but a `use_acx_device_manager` command must also be used.
- `set_csr_clock [-system] [-jtag]`
  Sets the clock used for CSR registers. For the ACX_DEVICE_MANAGER to operate correctly, the CSR clock should be the system clock. When the `use_acx_device_manager` command is specified, the clock selection should be automatic. If the CSR clock remains the JTAG clock even after specifying the system clock, the board is misconfigured (the `FCU_CFG_CLKSEL` pin of the FPGA must be 0). This condition must be corrected for the design to work.

Table 34: *set_csr_clock Arguments*

Argument	Optional	Description
`-system`	Y	Sets the system clock as CSR clock (default).
`-jtag`	Y	Sets the JTAG clock to the CSR clock.

Sharing the JTAG Interface with Snapshot

The Snapshot debug tool, used to observe signals in a design, is independent of the ACX_DEVICE_MANAGER, but also uses the JTAG interface to interact with ACE. The ACX_DEVICE_MANAGER has two ports (`o_jtap_bus` and `i_tdo_bus`) that pass the JTAG signals through so that the interface can be shared. The ACX_SNAPSHOT_UNIT module has matching ports (`i_jtap_bus` and `o_tdo_bus`) that should be connected to the ACX_DEVICE_MANAGER.

Caution!

1. It is necessary to use the ACX_SNAPSHOT_UNIT when using the ADM. ACX_SNAPSHOT cannot be connected.
2. `o_jtap_bus` is not a simple wire, but instead, is type `t_JTAP_BUS`. This type must be used in the wire declaration. When connected in this manner, Snapshot operates normally but with the caveat that follows in point 3 (This caveat may change in future versions of ACE).
3. To use Snapshot, the ACE JTAG connection must be closed using the `<device_namespace>::close_jtag` Tcl command. This is because Snapshot establishes its own connection to the JTAG driver in a different way, and the driver cannot have both connections open simultaneously. When a Snapshot has been taken, the JTAG interface connected can be opened again with `<device_namespace>::open_jtag`, to allow use of Tcl commands via JTAG. The JTAG connection can be opened and closed repeatedly without affecting the running design.

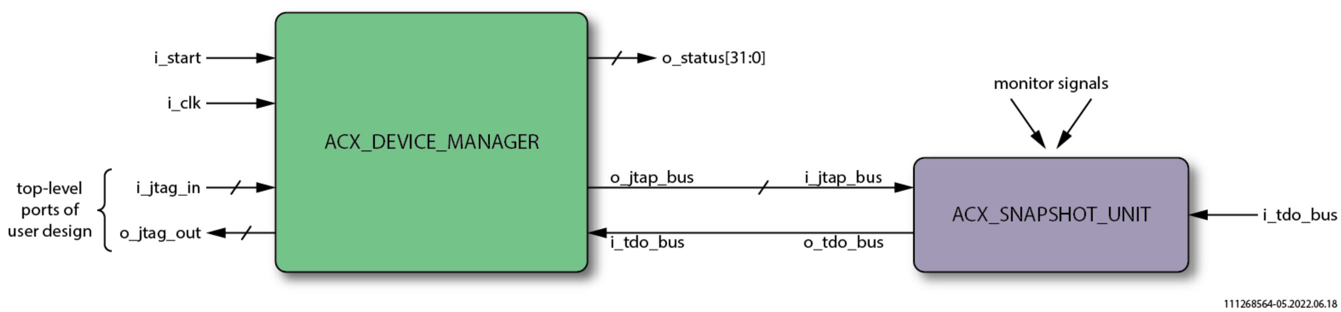

Figure 45: *Sharing the JTAG connection between ACX_DEVICE_MANAGER and Snapshot*

Simulation

Currently, in full RTL simulation mode, the ACX_DEVICE_MANAGER simply waits a few cycles, then sets `o_status` to `0x3` to indicate successful completion of the startup phase. This is appropriate for most simulations, which typically do not include full simulation models for all of the interfaces and external devices. Future ACE releases are planned to include more simulation options.

Instantiation Example

The following code shows the ACE generated device manager template:

Example template:

```
`include "speedster7t/macros/ACX_DEVICE_MANAGER.svp"

module device_manager_test
(
   // JTAG Interface
   input  t_JTAG_INPUT  i_jtag_in,      // Should be connected to top-level ports with the same
declaration
   input                i_tdo_bus,      // Pass-through the JTAG bus to connect to Snapshot. If
not used, this input should be tied to 1'b0
   output t_JTAG_OUTPUT o_jtag_out,     // Should be connected to top-level ports with the same
declaration
   output t_JTAP_BUS    o_jtap_bus,     // Pass-through of the JTAG bus to connect to Snapshot
(or other JTAG components)

   // User Design
   input                i_clk,          // 100 MHz Clock input for Device Manager block.
   input                i_start,        // A high input starts the Device Manager. In most cases
this signal is tied to 1'b1,
                                        // but it can also be tied to a PLL lock signal if
necessary.
   output [31:0]        o_status        // Progress indication, error status, alarms
);

   wire [1023:0] not_used;

  ACX_DEVICE_MANAGER #
  (
  .NAP_ROW      (6),            // NAP Row
  .NAP_COLUMN   (3)             // NAP Column
  )
  x_dev_mgr
  (

   // JTAG Interface
   .i_jtag_in   (i_jtag_in),    // Should be connected to top-level ports with the same
declaration
   .i_tdo_bus   (i_tdo_bus),    // Pass-through the JTAG bus to connect to Snapshot. If not used,
this input should be tied to 1'b0
   .o_jtag_out  (o_jtag_out),   // Should be connected to top-level ports with the same
declaration
   .o_jtap_bus  (o_jtap_bus),   // Pass-through of the JTAG bus to connect to Snapshot (or other
JTAG components)

   // User Design

   .i_clk       (i_clk),        // 100 MHz Clock input for Device Manager block.
   .i_start     (i_start),      // A high input starts the Device Manager. In most cases this
signal is tied to 1'b1,
                                // but it can also be tied to a PLL lock signal if necessary.
   .o_status    (o_status)      // Progress indication, error status, alarms
  );

endmodule : device_manager_test
```

The following example shows how the ACE-generated device manager template can be utilized in a design:

Using Without Snapshot:

```
module top_level
(
   // JTAG Interface
   input  t_JTAG_INPUT   i_jtag_in,     // Should be connected to top-level ports with the same
declaration
   output t_JTAG_OUTPUT  o_jtag_out,    // Should be connected to top-level ports with the same
declaration

   // User Design
   input  wire           i_clk          // 100 MHz Clock input for Device Manager block.
);

   // signals for shared JTAG bus
   wire  t_JTAP_BUS      jtap_bus;      // shared JTAG bus
   wire                  tdo_bus;       // tie to 1'b0 if unused

   // Other ADM signals
   logic [32 -1:0]       adm_status;    // Status from the ADM

   device_manager_test # ()
   i_acx_device_manager
   (
        // JTAG Interface
       .i_jtag_in   (i_jtag_in),        // Should be connected to top-level ports with the same
declaration
       .i_tdo_bus   (tdo_bus),          // Pass-through the JTAG bus to connect to Snapshot. If
not used, this input should be tied to 1'b0
       .o_jtag_out  (o_jtag_out),       // Should be connected to top-level ports with the same
declaration
       .o_jtap_bus  (jtap_bus),         // Pass-through of the JTAG bus to connect to Snapshot
(or other JTAG components)

       // User Design
       .i_clk       (i_clk),            // 100 MHz Clock input for Device Manager block.
       .i_start     (1'b1),             // A high input starts the Device Manager. In most cases
this signal is tied to 1'b1,
                                        // but it can also be tied to a PLL lock signal if
necessary.
       .o_status    (adm_status)        // Progress indication, error status, alarms
   );

endmodule : top_level
```

Using With Snapshot:

```
`include "speedster7t/common/speedster7t_snapshot_v3.sv"

module top_level
(
   // JTAG Interface
   input  t_JTAG_INPUT   i_jtag_in,      // Should be connected to top-level ports with the same
declaration
   output t_JTAG_OUTPUT  o_jtag_out,     // Should be connected to top-level ports with the same
declaration

   // User Design
   input                 i_clk           // 100 MHz Clock input for Device Manager block.
);

   // signals for shared JTAG bus
   wire   t_JTAP_BUS     jtap_bus;       // shared JTAG bus
   wire                  tdo_bus;        // tie to 0 if unused

   // Other ADM signals
   logic [32 -1:0]       adm_status;     // Status from the ADM

   device_manager_test # ()
   i_acx_device_manager
   (
        // JTAG Interface
        .i_jtag_in   (i_jtag_in),        // Should be connected to top-level ports with the same
declaration
        .i_tdo_bus   (tdo_bus),          // Pass-through the JTAG bus to connect to Snapshot. If
not used, this input should be tied to 1'b0
        .o_jtag_out  (o_jtag_out),       // Should be connected to top-level ports with the same
declaration
        .o_jtap_bus  (jtap_bus),         // Pass-through of the JTAG bus to connect to Snapshot
(or other JTAG components)

        // User Design
        .i_clk       (i_clk),            // 100 MHz Clock input for Device Manager block.
        .i_start     (1'b1),             // A high input starts the Device Manager. In most cases
this signal is tied to 1'b1,
                                         // but it can also be tied to a PLL lock signal if
necessary.
        .o_status    (adm_status)        // Progress indication, error status, alarms
   );

   ACX_SNAPSHOT_UNIT #(....)
   x_snapshot
   (
        .i_jtap_bus  (jtap_bus),
        .i_tdo_bus   (1'b0),             // Tie to 1'b0 if not used
        .o_tdo_bus   (tdo_bus),
        ... other Snapshot ports ...
   );

endmodule : top_level
```

Chapter - 17: Revision History

Version	Date	Description
1.0	13 Sep 2021	• Initial release.
2.0	20 Sep 2022	• Add device manager.