SoC Blockset™
User's Guide

MATLAB®&SIMULINK®

MathWorks®

How to Contact MathWorks

Latest news: www.mathworks.com

Sales and services: www.mathworks.com/sales_and_services

User community: www.mathworks.com/matlabcentral

Technical support: www.mathworks.com/support/contact_us

Phone: 508-647-7000

The MathWorks, Inc.
1 Apple Hill Drive
Natick, MA 01760-2098

Contents

Simulate SoC Applications

2

Generate Code and Deploy on SoC Device

3

Analyze Performance on SoC Device

4

Create SoC Models

What is Task Execution?

A task is a unit of execution or unit of work in a software application. Typically, task execution in an embedded processor is managed by the operating system (OS). When deployed to the embedded processor, a task corresponds to an OS thread. The SoC Blockset defines the execution life cycle and relation to OS threads as follows.

Task Execution Life Cycle

The life cycle of a task can be divided into five states:

*   *Created* – The system creates all the tasks when the application starts and immediately moves them to the waiting state.
*   *Waiting* – The task waits for the associated trigger signal, such as an OS timer or I/O device. After receiving the trigger signal, the task starts to run. If the task has the highest priority, it enters the running state. Otherwise, the task continues to wait until it becomes the highest priority, triggered task.
*   *Running* – The task executes its code. When the code completes execution, the task immediately moves to the waiting state. If a trigger for a higher-priority task occurs, the running task moves to the preempted state.
*   *Preempted* – The task is preempted and waiting to run. A task runs based on a combination of the task priority and the order the task entered the *Preempted* state. Assuming equal task priorities of all other tasks in *Ready to Resume* state, tasks run based on first-in-first-out (FIFO) ordering.
*   *Terminated* – Tasks terminate when the application ends.

This figure shows the state diagram of a task execution life cycle for an application using an OS. For simplicity, the terminated state is not shown, but a task can reach the terminated state from any of the other states.
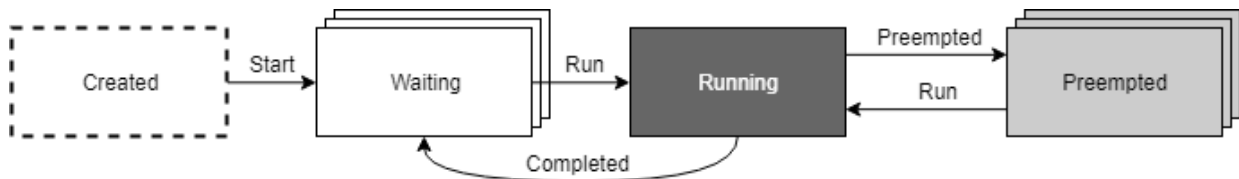
Task and Thread

A task is a conceptual unit of work in an algorithm. In an application executing on a device, a task is a section of code that executes in a thread within an operating system (OS). The OS thread determines the state of execution of the task. Within the SoC Blockset, a task specifically refers to the portion of the Simulink® model contained within a rate or function-call subsystem. The trigger signal for that subsystem comes from a Task Manager block. When deployed to hardware, an OS thread uses the task properties. The thread executes the code generated from the subsystem. Conceptually, a *Task* in simulation is equivalent to a *thread* in generated code.

See Also

Task Manager

More About

- "Timer-Driven Task" on page 1-8
- "Event-Driven Tasks" on page 1-4

External Websites

- Task (computing)

Event-Driven Tasks

Event-driven tasks start executing when triggered by an external event. Events can include internal events, such as memory stream or register writes, or external events, such as receiving a UDP data packet from a network connection. Assuming no other tasks are executing at the time of the event or the task has the highest priority, the event-driven task can respond immediately to the event. The task can then process the received data, and potentially generate other events in the model.

Create a Simulink Model with an Event Driven Task

This example shows how to create and configure a Simulink® model to use the event driven task feature of the SoC Blockset.

Create a Software Reference Model

This section shows how to create a reference model of the software for an SoC application model. The software contains an event driven task subsystem that reacts to receiving UDP packets.

1   Create a new blank model.

2   In the Simulink editor, add a Function-Call Subsystem block to the model. Connect an Inport block to the input port of the Function-Call Subsystem block. Connect the output port to a Termclcinator block.

3   Add an Asynchronous Task Specification block to the model. On the Block Parameters dialog box, set the **Task priority** to 41.

4   Connect the output port of Asynchrnous Task Specification block to the function() input of the Function-Call Subsystem block.

5   Add an Inport block and open the Block parameters dialog box. On the **Signal Attributes** tab, check **Output function call**. Connect the Inport block to the input port of the Asynchronous Task Specification block.

6   Open the Function-Call subsystem model.

7   Add a UDP Read block to model. Open the Block Parameters dialog box, set **Maximum data length (elements)** to 1024 and check **Enable event-based execution**.

8   Connect the Inport block to the UDP Read block **UDP Data** port. Connect the **Data** port to the Outport block. Connect the **Length** port to a Terminator block.

9 Open the Configuration Parameters dialog box, select the **Solver** pane. Set **Solver selection > Type** to Fixed-step and check **Tasking and sample timer options > Higher priority value indicates higher task priority**.

10 Select the **Hardware Implementation** pane, set **Hardware board** to Zedboard.

11 Save the model as soc_task_createeventdriventask_software.slx.

The completed model should look similar to the following model.
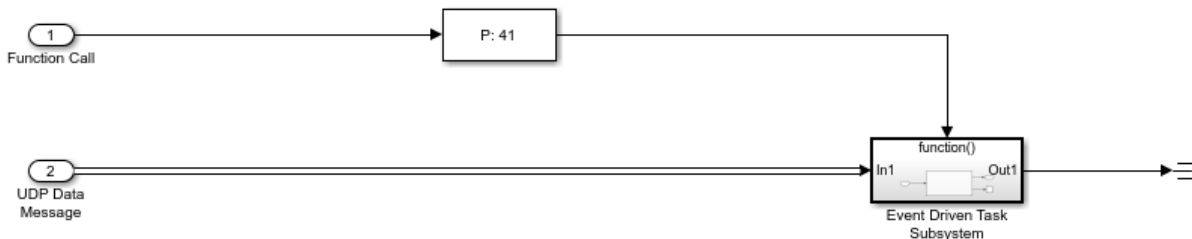
Create the SoC Application Model

This section shows how to create the top level SoC application model that contains the software reference model developed in the previous section.

1 Create a new blank model.

2 In the Simulink editor, add a Model block. On the Block Parameters dialog box, set **Model name** to soc_task_createeventdriventask_software.slx.

3 Add a Task Manager block and open the Block Parameters dialog box. Set the **Main > Type** to Event-driven and **Main > Priority** to 41. Each newly added event-driven task exposes an event message input port on the Task Manager block.

4 (Optional) On the **Simulation** tab, you specify the task duration for that task. For more information on setting task duration, see Task Duration.

5 In the editor, add an IO Data Source block to the model. Open the Block Parameters dialog box and enable **Show event port**.

6 Connect the IO Data Source block **Event** port to the Task Manager and the **UDP Data** port to the UDP Data Message port on the Model reference block.

7 Open the Configuration Parameters dialog box, select the **Solver** pane. Set **Solver selection > Type** to Fixed-step and check **Tasking and sample timer options > Higher priority value indicates higher task priority**.

8 Select the **Hardware Implementation** pane, set **Hardware board** to Zedboard.

9 Update the diagram, press **Ctrl+D**.

10 Save the model as soc_task_createeventdriventask_application.slx.

The completed model should look similar to the following model.
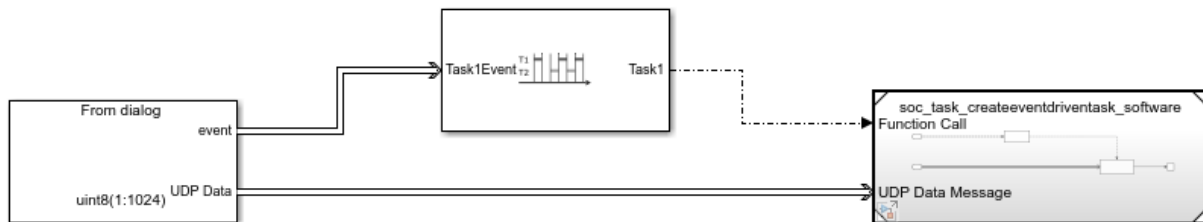
Run the Model with Event Driven Task

In the Simulink editor, run the
soc_task_createeventdriventask_application.slx model. When the run
completes, open the Simulation Data Inspector (SDI) and select **Task1**. The SDI display
shows that **Task1** triggers and executes each time a new UDP packet arrives. Although
superficially the task execution appears periodic, this is only a byproduct of the current
default settings of the IO Data Source block that generates the event with a time step of
0.1.
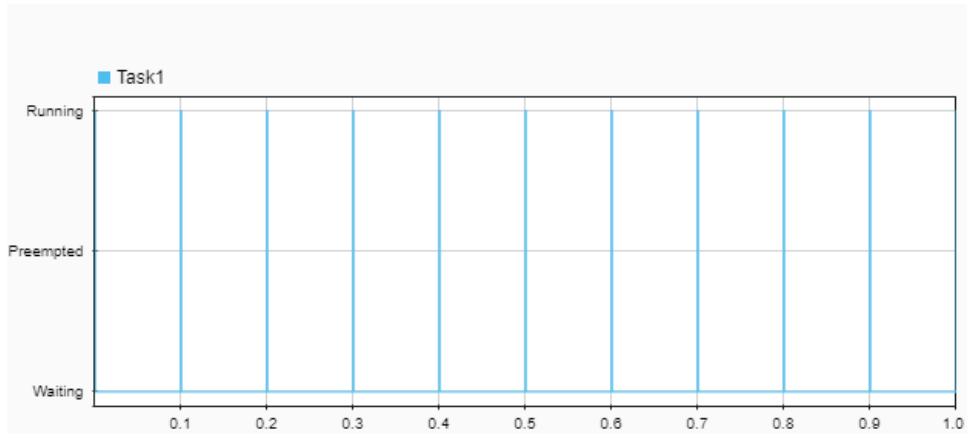
See Also

I/O Data Source | Task Manager

More About

- "What is Task Execution?" on page 1-2
- "Timer-Driven Task" on page 1-8

Timer-Driven Task

Timer-driven tasks execute at a periodic rate equal to an integer multiple of the Simulink model fundamental sample time.

To create a timer-driven task, connect the task port of a Task Manager block to a periodic event port on a Model block. Each rate in a Model block generates a unique model periodic event port with the time step for the rate shown on the block icon. In the Model block dialog mask, use the **Schedule rates** parameter to enable model periodic event ports.

Note A timer-driven task requires a lower priority than an event-driven task.

Create a Simulink Model with an Timer Driven Task

This example shows how to create and configure a Simulink(r) model to use the timer driven task feature of the SoC Blockset.

Create a Software Reference Model

This section shows how to create a reference model of the software for an SoC application model. The software contains one timer driven task subsystem that reacts to receiving UDP packets.

1 Create a new blank model.
2 In the Simulink editor, add a Subsystem block to the model. Add a Sine block and connect it to the Subsystem block. Connect the output of the Subsystem block to a Terminator block.
3 Open the Function-Call subsystem model.
4 Open the Block parameteres dialog box of the Inport block, set the **Sample Time** to `0.1`.
5 In the Simulink editor, open the Configuration Parameters dialog box.
6 Select the **Hardware Implementation** pane, set **Hardware board** to `Zedboard`.
7 Save the model as `soc_task_createtimerdriventask_software.slx`.

The completed model should look similar to the following model.

Create the SoC Application Model

This section shows how to create the top level SoC application model that contains the software reference model developed in the previous section.

1 Create a new blank model.

2 In the Simulink editor, add a Model block and open the Block Parameters dialog box.

3 Check **Main > Schedule Rates** and set **Main > Model name** to
   `soc_task_createtimerdriventask_software.slx`.

4 In the editor, add a Task Manager block to the model.

5 (Optional) Open the Block Parameters dialog box of the Task Manager block. By
   default, the task **Type** is `Timer-driven` with a **Period** of `0.1`. On the **Simulation**
   tab, you specify the task duration for that task. For more information on setting task
   duration, see Task Duration.

6 In the editor, connect the **Task1** port to the **D1[0.1]** port of the Model block.

7 Open the Configuration Parameters dialog box, select the **Hardware
   Implementation** pane, set **Hardware board** to `Zedboard`.

8 Update the diagram, press **Ctrl+D**.

9 Save the model as `soc_task_createtimerdriventask_application.slx`.

The completed model should look similar to the following model.
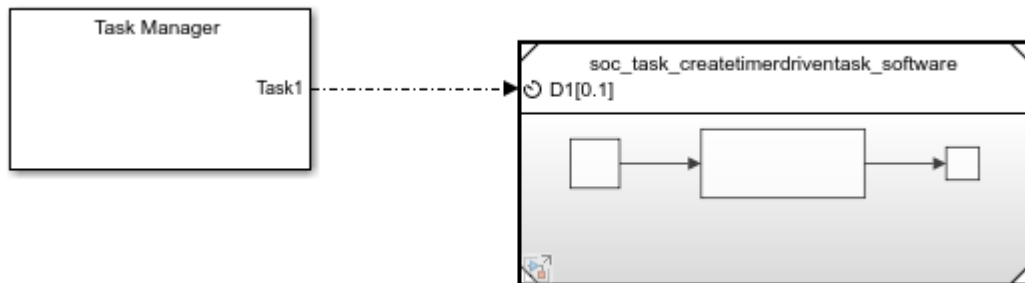
Run the Model with Timer Driven Task

In the Simulink editor, run the
`soc_task_createtimerdriventask_application.slx` model. When the run completes, open the Simulation Data Inspector (SDI) and select **Task1**. The SDI display shows that **Task1** triggers each `0.1` time steps.
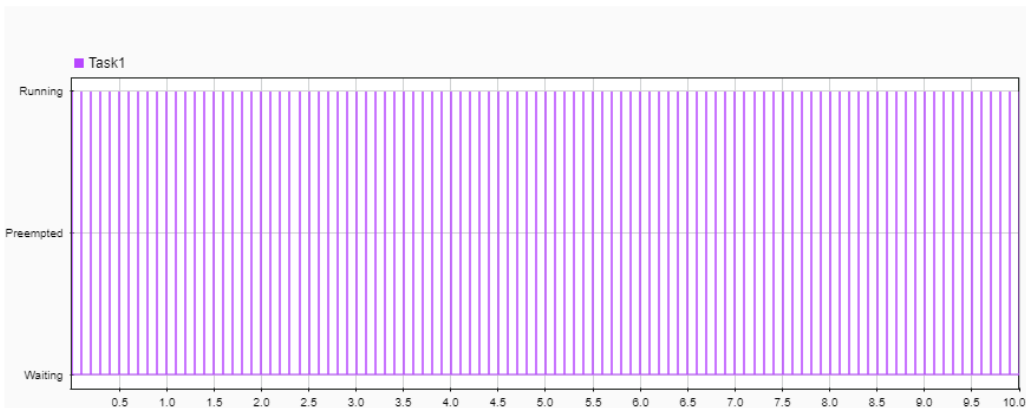
See Also
Task Manager

More About
- "What is Task Execution?" on page 1-2
- "Event-Driven Tasks" on page 1-4

Kernel Latency

In a deployed application, switching between threads requires a finite amount of time depending on the current state of the thread, embedded processor, and OS. *Kernel latency* defines the time required for the operating system to respond to a trigger signal, stop execution of any running threads, and start the execution of the thread responsible for the trigger signal.

SoC Blockset models simulate *Kernel latency* as a delay at the start of execution of a task the first time the task moves from the waiting to running state. The following diagram shows the execution timing of a high-priority and low-priority task on a system that simulates a single processor core.
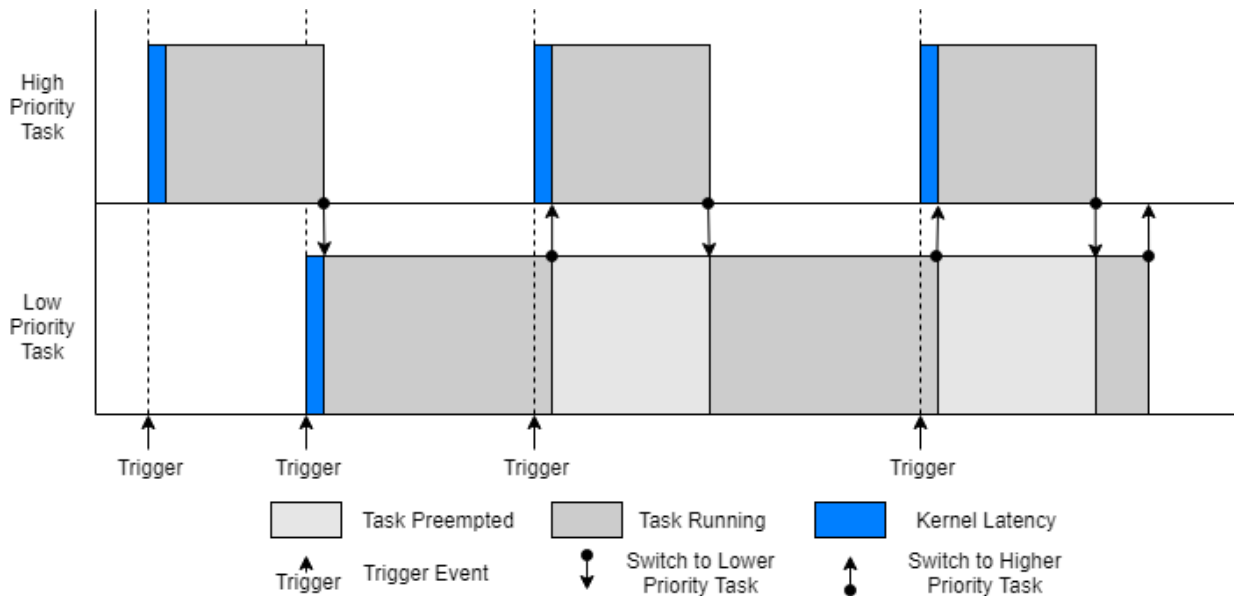
Other factors affecting kernel latency, such as context switch times, can be considered negligible compared to other effects and are not modeled in simulation.

Note Kernel latency requires advanced knowledge of the processor specifications and can be generally set to 0 without impact to the simulation.

Effect Kernel Latency on Task Execution

This example shows the effect of kernel latency on the behavior and timing of two timer driven tasks in an SoC application.

The following model simulates a software application with two timer driven tasks. The task characteristics, specified in the Task Manager block, are as follows:

Name	Period	Mean Duration
HighPriorityTask	0.01	0.004
LowPriorityTask	0.03	0.007

With these timing conditions, the high priority task preempts the low priority task. In the model Configuration Parameters dialog box, the **Hardware Implementation > Operating system/scheduler > Kernel latency** is set to `0.002`.
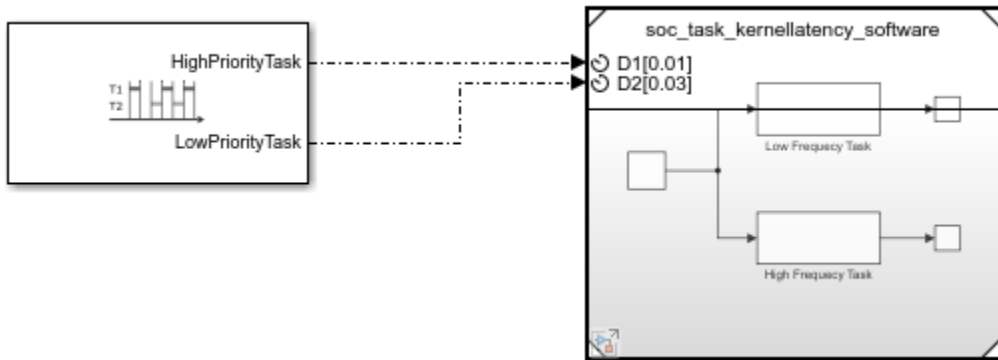
Run the model and open the Simulation Data Inspector (SDI) display. Selecting the two task signal produces the following display.
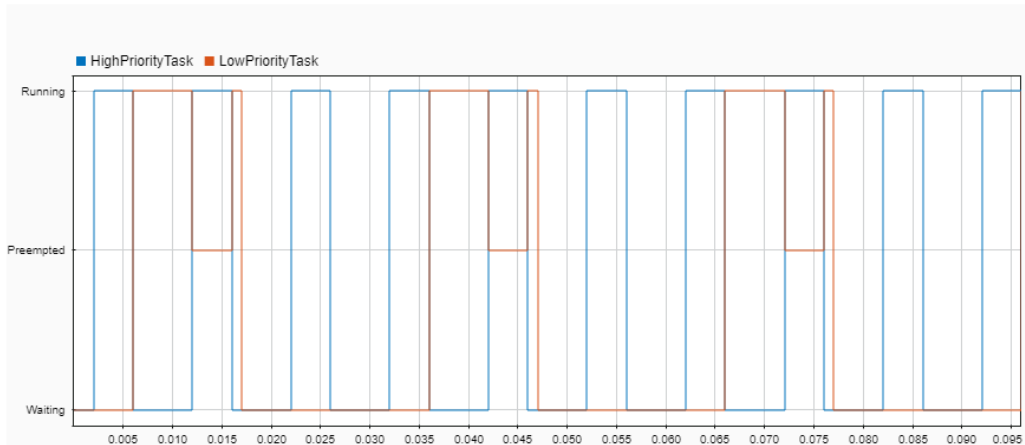
Inspecting the SDI display, a change in task state state from *Waiting* to *Running* shows a latency of `0.002` seconds. However, when the task changes from *Preempted* to *Running*, no latency occurs. This timing matches with the expected behavior of task, experiencing a latency in startup of that task execution instance, but not when the task instance already exists.

See Also

Task Manager

More About

- "What is Task Execution?" on page 1-2
- "Task Duration" on page 1-14

Task Duration

The total time an instance of a task spends in the running state defines the task duration. Task duration can vary due to multiple sources, in particular:

- Conditional branching in the task algorithm
- Dependence on signal values from other tasks
- Dependence on signals from external sources, such as I/O devices or hardware user logic
- Compiler settings and SoC device processor architecture

As a result, task duration for any given task instance can be nondeterministic.

The Task Manager block provides four ways to simulate the nondeterministic task duration: approximation using a parameterized probability distribution, approximation using a calculated probability distribution, and replay of recorded task execution timing data.

Approximation Using Parameterized Probability Distribution

In simulation, the Task Manager block can define the task duration as random variable expressed as the weighted sum of truncated normal distributions. For example, this diagram shows the probability distribution of a task that executes with a short task duration, but can occasionally execute with a longer durations.
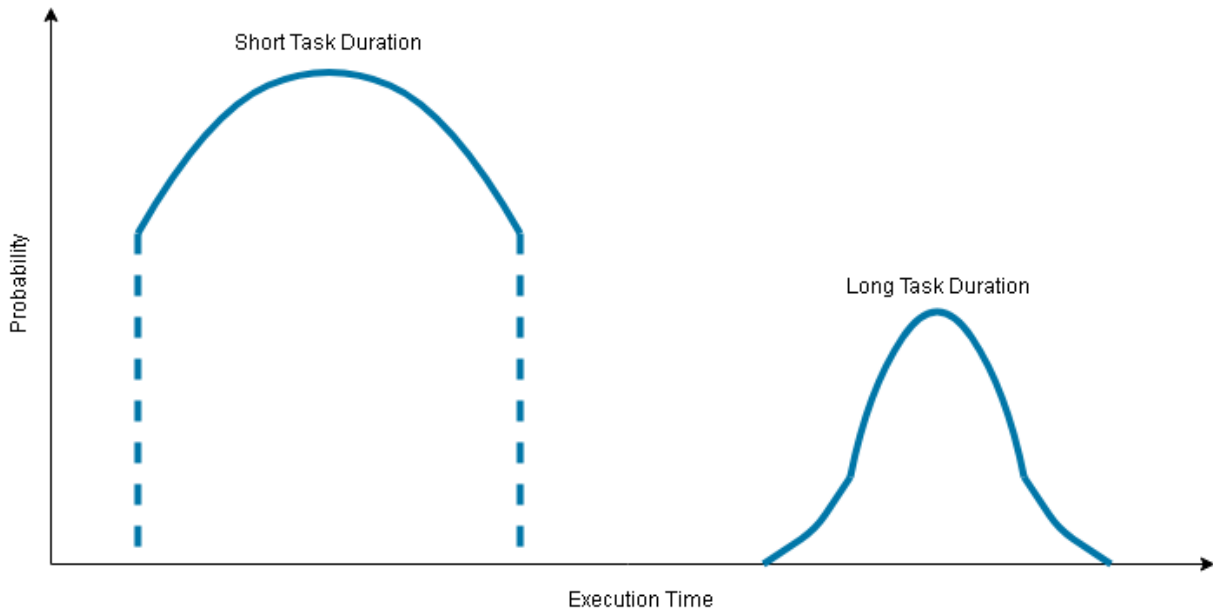
To create a probability distribution for a task duration, first open the Task Manager block dialog. Then, on the **Simulation** tab, set **Specify task duration via:** to `Dialog`. In the **Task duration settings** section, you can set the properties of each distribution by editing the text of that property. You can also add and delete probability distributions from the sum of distributions by clicking the **Add distribution** and **Delete distribution** buttons, respectively.

Task duration settings

Define probability distribution of the task duration times as a combination of up to five normal distributions, each with the specified occurrence probability.

	Percent	Mean	SD	Min	Max
1	70	1e-06	0	1e-06	1e-06
2	10	4e-06	1e-07	4e-06	5e-06
3	20	5e-06	1e-06	4e-06	6e-06

Add distribution   Delete distribution

Note

- The sum of the `Percent` weights must equal 100.
- Each task can use a maximum of 5 distributions.

Approximation Using Calculated Probability Distribution

Each recording of task execution data, either from a previous simulation or from execution on an SoC device, generates several profiling files. The `metadata.csv` file contains the calculated mean and standard deviation for each task in that recording. To configure a task in the Task Manager block to use the derived statistical data for task duration, follow these steps:

1 Open the Task Manager block dialog mask.

2 On the **Simulation** tab, set **Specify task duration via** to `Recorded task diagnostics file`.

3 Specify the location and name of the `metadata.csv` file. The **Mean** and **Deviation** parameters are automatically updated with the data from the file.

4 Click **OK**.

Specification from Task Manager Input Port

An input port on the Task Manager block dynamically specifies the task duration. To expose this task duration input port, follow these steps:

1 Open the Task Manager block dialog mask.

2 On the **Simulation** tab, set **Specify task duration via** to `Input port`.

3 Click **OK** to expose a new input port, named **TaskNameDur**, on the block.

Replay of Recorded Task Execution Timing Data

A data file provides exact task duration for each task execution instance. A task execution data file can come from a previous or independent model simulation or directly from the task execution on a processor in an SoC device. For more information on replaying recorded task execution timing data, see "Task Execution Playback using Recorded Data" on page 2-9.

See Also

Task Manager

More About

• "What is Task Execution?" on page 1-2
• "Task Execution Playback using Recorded Data" on page 2-9

External Websites

• Truncated Normal Distribution

Value and Caching of Task Subsystem Signals

In SoC Blockset, a task subsystem can be treated as an independent model with the task duration simulating the expected execution time on an SoC device. When the Task Manager block executes a task, input signals connected to that task subsystem can either be sampled and cached at the start of the task execution or sampled at the end of the task execution instance. The task subsystem then executes using either the cached or latest value. The value of signals and buses output from the subsystem change at the end of the task execution instance.

To enable task subsystem input signal caching, first open the Simulink configuration parameters on the processor reference model. On the **Hardware Implementation** pane, select **Hardware board settings > Task and memory simulation > Cache input data at task start**.

See Also
Task Manager

More About
- "What is Task Execution?" on page 1-2
- "Task Duration" on page 1-14
- "Kernel Latency" on page 1-11

Memory and Register Data Transfers

An SoC application is composed of one or more algorithms. When an algorithm transfers data to another algorithm, the data is represented as a signal line in Simulink. For behavioral models, the data transfer is instantaneous.

This diagram shows a behavioral model of a datapath between two algorithms.

In the physical world, the algorithms can be on two separate devices, and data transfers do not happen instantaneously. Furthermore, the algorithms can run at different rates, and therefore require buffering and control logic for handshaking. For example, a simple handshake such as "data is valid" from the producer of the data and "ready to accept data" from the consumer serve as control logic.

If one processing element executes in an FPGA or ASIC, and the next processing element executes on an embedded processor, then a simple signal line represents more than just a complex hardware datapath. The data transfer also represents a processor interrupt handler, an operating system task scheduler, and a software driver stack.

In SoC Blockset, you model data transfers and handshake protocols through shared memory. Use a Memory Channel block for external memory or a Register Channel block for registers.

Modeling Datapath with Memory Channel Block

The Memory Channel block represents an abstraction to a complex datapath through external memory and supports different handshake protocols. It facilitates a refinement of the communication between processing elements from an instantaneous, protocol-less wire to a full direct memory access (DMA) connection between a processor and an FPGA.

By adding a Memory Channel block, you can model data movement from one part of the algorithm to another.

The block provides a model of the communication pipeline. The channel also provides a signaling interface.
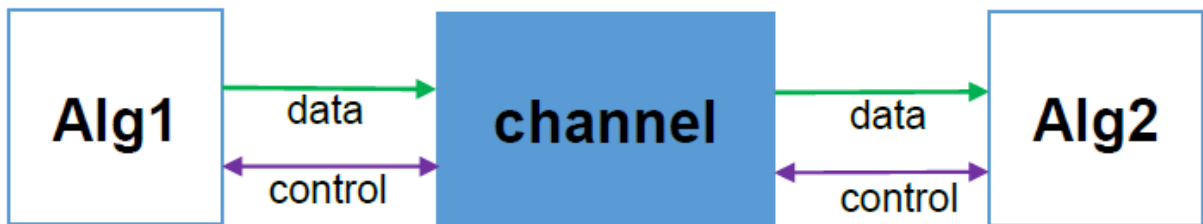
The interface protocol depends on where the processing is executed. An FPGA or ASIC algorithm can perform data transfers by using standard protocols such as AXI4-Stream or AXI4. An embedded CPU algorithm can use a driver-interface exported to the user space.

This figure shows a model of the datapath from an FPGA algorithm streaming data to a processor algorithm.
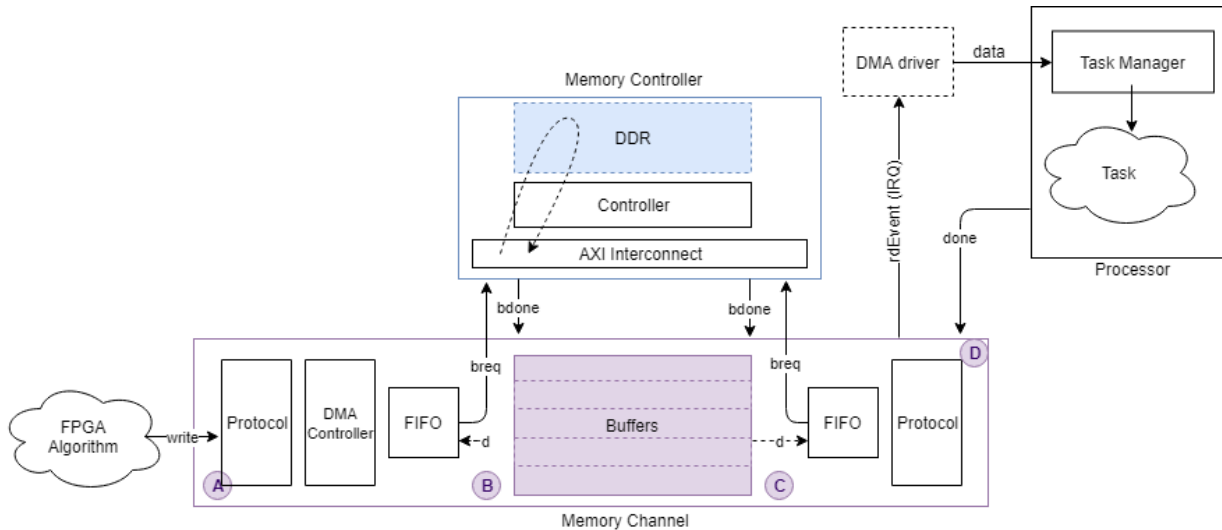
Other Memory Channel type selections model additional common datapaths through external memory. For more information about Memory Channel configurations, see Memory Channel.

The writer and reader are connected to the memory and request access to the external memory from a memory controller. For more information about the Memory Controller block, see Memory Controller.

Modeling Datapath with Register Channel Block

The Register Channel block represents the serialization of the processor reads or writes through a common configuration bus such as AXI-Lite.

The Register Channel block provides a timing model for the transfer of register values between processor and hardware algorithms through a common configuration bus. Use this block when the processor writes a command or configuration register or when the processor reads a status register.

See Also
Memory Channel | Register Channel

More About

- "External Memory Channel Protocols" on page 2-41

Simplified AXI4 Master Interface

In this section...

For designs that require accessing large data sets from an external memory, model your algorithm with a simplified AXI4 Master protocol. When you run the `IP Core Generation` workflow, HDL Coder™ generates an IP core with AXI4 Master interfaces. The AXI4 Master interface can communicate between your design and the external memory controller IP by using the AXI4 Master protocol.

Simplified AXI4 Master Protocol - Write Channel

You can use the simplified AXI4 Master protocol to map to AXI4 Master interfaces. Use the simplified AXI4 Master write protocol for a write transaction and the simplified AXI4 Master read protocol for a read transaction.

This figure shows the timing diagram for the signals that you model at the DUT input and output interfaces for an AXI4 Master write transaction.
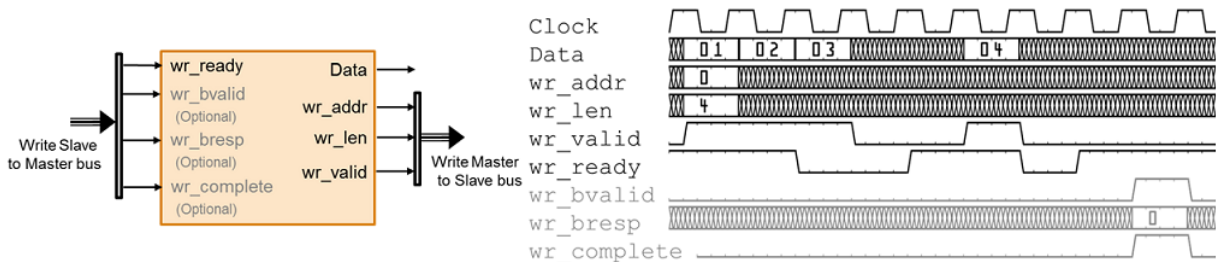
The DUT waits for `wr_ready` to become high to initiate a write request. When `wr_ready` becomes high, the DUT can send out the write request. The write request consists of the `Data` and `Write Master to Slave` bus signals. This bus consists of `wr_len`, `wr_addr`, and `wr_valid`. `wr_addr` specifies the starting address that DUT wants to write to. The `wr_len` signal corresponds to the number of data elements in this write transaction. `Data` can be sent as long as `wr_valid` is high. When `wr_ready` becomes low, the DUT must stop sending data within one clock cycle, and the `Data` signal becomes invalid. If the DUT continues to send data after one clock cycle, the data is ignored.

Output Signals

Model the `Data` and `Write Master to Slave bus` signals at the DUT output interface.

- `Data`: The data that you want to transfer, valid each cycle of the transaction.
- `Write Master to Slave bus` that consists of:
  - `wr_addr`: Starting address of the write transaction that is sampled at the first cycle of the transaction.
  - `wr_len`: The number of data values that you want to transfer, sampled at the first cycle of the transaction.
  - `wr_valid`: When this control signal becomes high, it indicates that the `Data` signal sampled at the output is valid.

Input Signals

Model the `Write Slave to Master bus` that consists of:

- `wr_ready`: This signal corresponds to the backpressure from the slave IP core or external memory. When this control signal goes high, it indicates that data can be sent. When `wr_ready` is low, the DUT must stop sending data within one clock cycle. You can also use the `wr_ready` signal to determine whether the DUT can send a second burst signal immediately after the first burst signal has been sent. Multiple burst signals are supported, which means that the `wr_ready` signal remains high to accept the second burst immediately after the last element of the first burst has been accepted.
- `wr_bvalid` (optional signal): Response signal from the slave IP core that you can use for diagnosis purposes. The `wr_bvalid` signal becomes high after the AXI4 interconnect accepts each burst transaction. If `wr_len` is greater than `256`, the AXI4 Master write module splits the large burst signal into 256-sized bursts. `wr_bvalid` becomes high for each 256-sized burst.
- `wr_bresp` (optional signal): Response signal from the slave IP core that you can use for diagnosis purposes. Use this signal with the `wr_bvalid` signal.
- `wr_complete` (optional signal): Control signal that when remains high for one clock cycle indicates that the write transaction has completed. This signal asserts at the last `wr_bvalid` of the burst.

Simplified AXI4 Master Protocol - Read Channel

This figure shows the timing diagram for the signals that you model at the DUT input and output interfaces for an AXI4 Master read transaction. These signals include the `Data`, `Read Master to Slave Bus`, and `Read Slave to Master Bus`.
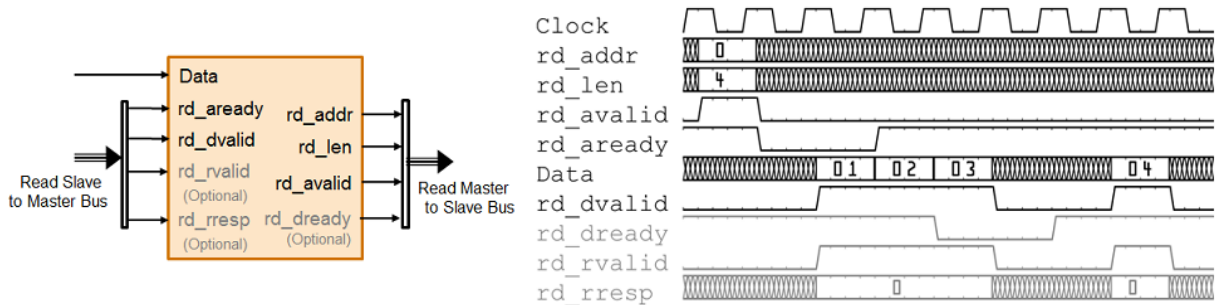
The DUT waits for `rd_aready` to become high to initiate a read request. When `rd_aready` is high, the DUT can send out the read request. The read request consists of the `rd_addr`, `rd_len`, and `rd_avalid` signals of the `Read Master to Slave bus`. The slave IP or the external memory responds to the read request by sending the `Data` at each clock cycle. The `rd_len` signal corresponds to the number of data values to read. The DUT can receive `Data` as long as `rd_dvalid` is high.

Read Request

To model a read request, at the DUT output interface, model the `Read Master to Slave bus` that consists of:

- `rd_addr`: Starting address for the read transaction that is sampled at the first cycle of the transaction.
- `rd_len`: The number of data values that you want to read, sampled at the first cycle of the transaction.
- `rd_avalid`: Control signal that specifies whether the read request is valid.

At the DUT input interface, implement the `rd_aready` signal. This signal is part of the `Read Slave to Master bus` and indicates when to accept read requests. You can monitor the `rd_aready` signal to determine whether the DUT can send consecutive burst requests. When `rd_aready` becomes high, it indicates that the DUT can send a read request in the next clock cycle.

Read Response

At the DUT input interface, model the `Data` and `Read Slave to Master bus` signals.

- `Data`: The data that is returned from the read request.
- `Read Master to Slave bus` that consists of:
  - `rd_dvalid`: Control signal which indicates that the `Data` returned from the read request is valid.
  - `rd_rvalid` (optional signal): response signal from the slave IP core that you can use for diagnosis purposes.
  - `rd_rresp` (optional signal): Response signal from the slave IP core that indicates the status of the read transaction.

At the DUT output interface, you can optionally implement the `rd_dready` signal. This signal is part of the `Read Master to Slave bus` and indicates when the DUT can start accepting data. By default, if you do not map this signal to the AXI4 Master read interface, the generated HDL IP core ties `rd_dready` to logic high.

AXI4-Stream Interface

Using SoC Blockset, you can model a simplified, streaming protocol in your model. Use HDL Coder to generate AXI4-Stream interfaces in the IP core.

Simplified Streaming Protocol

When you want to generate an AXI4-Stream interface in your IP core, in your DUT interface, implement the following signals:

- Data
- Valid

When you map scalar DUT ports to an AXI4-Stream interface, you can optionally model the following signals and map them to the AXI4-Stream interface:

- Ready
- Other protocol signals, such as:

    - TSTRB
    - TKEEP
    - TLAST
    - TID
    - TDEST
    - TUSER

Data and Valid Signals

When the Data signal is valid, the Valid signal is asserted.
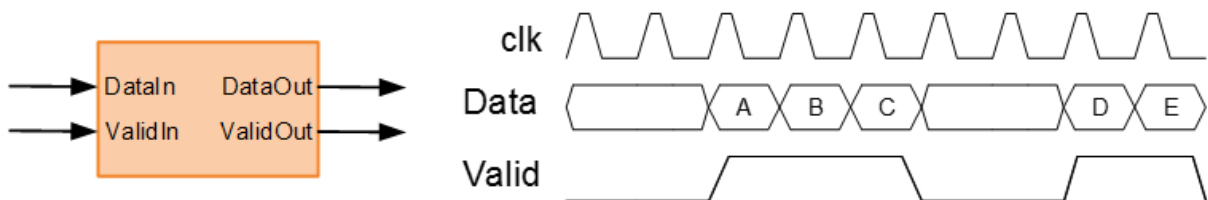
Ready Signal (Optional)

The AXI4-Stream interfaces in your DUT can optionally include a Ready signal. In a Slave interface, the Ready signal enables you to apply back pressure. In a Master interface, the Ready signal enables you to respond to back pressure.

If you model the Ready signal in your AXI4-Stream interfaces, your Master interface ignores the Data and Valid signals one clock cycle after the Ready signal is deasserted. You can start sending Data and Valid signals once the Ready signal is asserted. You can send one more Data and Valid signal after the Ready signal is deasserted.

If you do not model the Ready signal, HDL Coder generates the signal and the associated back pressure logic.
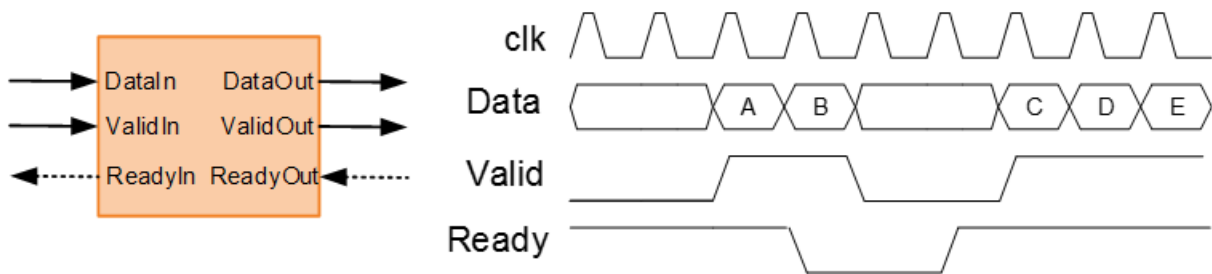
For example, if you have a FIFO in your DUT to store a frame of data, to apply back pressure to the upstream component, you can model the Ready signal based on the FIFO Full signal.

AXI4-Stream Video Interface

In this section...

Using SoC Blockset, you can implement a simplified, streaming pixel protocol in your model. Use HDL Coder to generate an HDL IP core with AXI4-Stream Video interfaces.

Streaming Pixel Protocol

You can use the streaming pixel protocol for AXI4-Stream Video interface mapping. Video algorithms process data serially and generate video data as a serial stream of pixel data and control signals.

To generate an IP core with AXI4-Stream Video interfaces, in your DUT interface, implement these signals:

- Pixel Data
- Pixel Control Bus

The **Pixel Control Bus** is a bus that has these signals:

- hStart
- hEnd
- vStart
- vEnd
- valid

The signals **hStart** and **hEnd** represent the start of an active line and the end of an active line respectively. The signals **vStart** and **vEnd** represent the start of a frame and the end of a frame.

You can optionally model the backpressure signal, **Ready**, and map it to the AXI4-Stream Video interface.

Protocol Signals and Timing Diagrams

This figure is a 2–by–3 pixel image. The active image area is the rectangle with a dashed line around it and the inactive pixels that surround it. The pixels are labeled with their grayscale values.
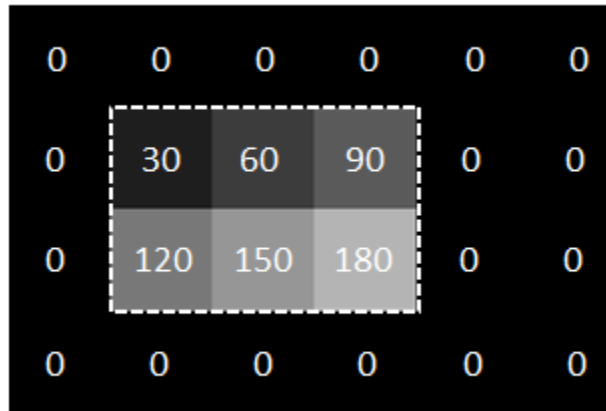
Pixel Data and Pixel Control Bus

This figure shows the timing diagram for the **Pixel Data** and **Pixel Control Bus** signals that you model at the DUT interface.
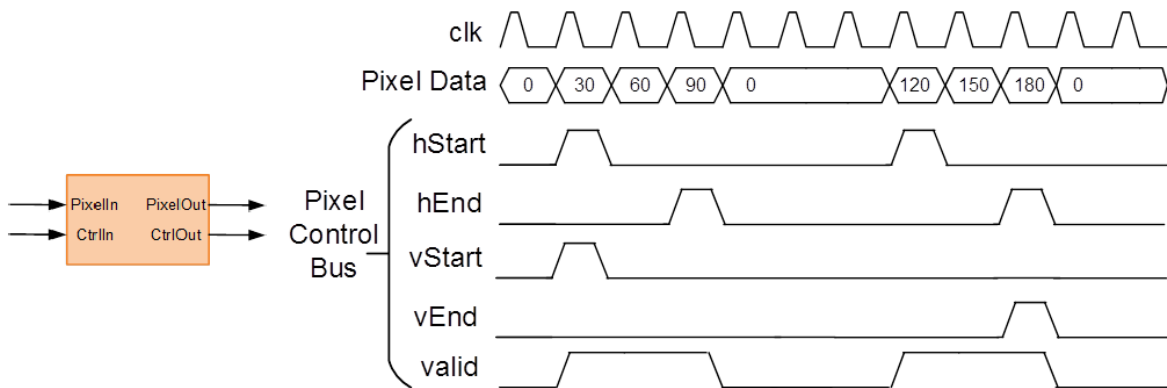
The **Pixel Data** signal is the primary video signal that is transferred across the AXI4-Stream Video interface. When the **Pixel Data** signal is valid, the **valid** signal is asserted.

The **hStart** signal becomes high at the start of the active lines. The **hEnd** signal becomes high at the end of the active lines.

The **vStart** signal becomes high at the start of the active frame in the second line. The **vEnd** signal becomes high at the end of the active frame in the third line.

Optional Ready Signal

This figure shows the timing diagram for the **Pixel Data**, the **Pixel Control Bus**, and the **Ready** signal that you model at the DUT interface.
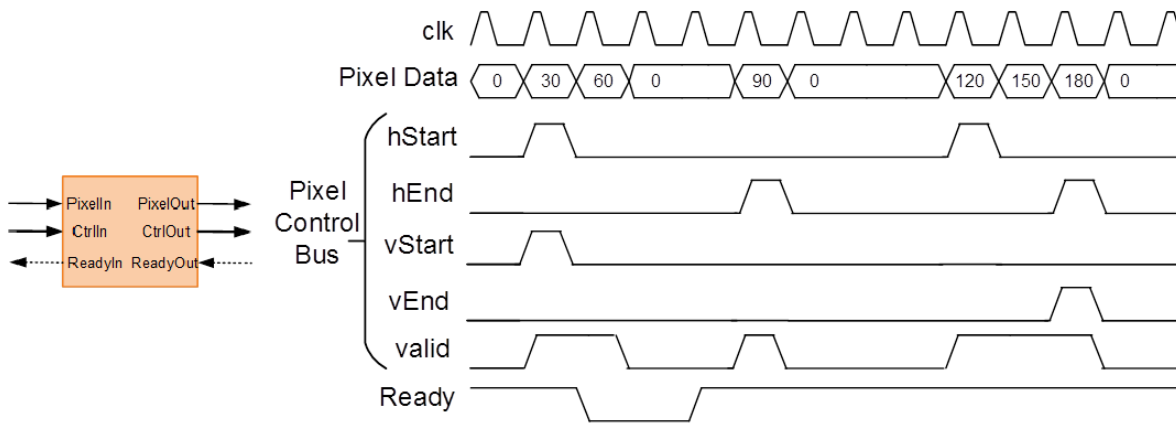
When you map the DUT ports to an AXI4-Stream Video interface, you can optionally model the backpressure signal, **Ready**, and map it to the AXI4-Stream Video interface.

In a Slave interface, with the **Ready** signal, you can apply back pressure. In a Master interface, with the **Ready** signal, you can respond to back pressure.

If you model the **Ready** signal in your AXI4-Stream Video interfaces, your Master interface must deassert its **valid** signal one cycle after the **Ready** signal is deasserted.

If you do not model the **Ready** signal, HDL Coder generates the associated backpressure logic.

Models with I/O Device Blocks

The **Processor I/O** sublibrary in the SoC Blockset contains blocks that simulate the data transfer between the processor system and other I/O devices in the SoC application. **Processor I/O** blocks, including the Register Read, Register Write, and Stream Read, can read and write data to internal I/O, such as DDR memory or registers, on the SoC device. Similarly, the TCP Read, TCP Write, UDP Read, and UDP Write blocks can read and write data to external communication I/O devices.

Processor I/O with Memory Model

Processor I/O blocks, including the Register Read, Register Write, and Stream Read, simulate reading and writing of data to registers and shared memory.

Processor I/O with Network Device Model

Processor I/O blocks, including the TCP Read, TCP Write, UDP Read, and UDP Write blocks, simulate the exchange of data to external communication protocols supported by an SoC device.

See Also

Use Template to Create SoC Model

SoC Blockset model templates provide design patterns and best practices for models intended for simulation, HDL code generation, or SoC deployment. Models created from any one of SoC Blockset model templates have their configuration parameters set up for simulation and code generation.

Create SoC Model Using SoC Blockset Template

To efficiently model hardware for SoC design, create a project by using an SoC Blockset template.

1   In the MATLAB® Home tab, select the **Simulink** button. Alternatively, at the command line, enter:

    ```
    simulink
    ```

2   On the Simulink Start Page, scroll down to the **SoC Blockset** section, which contains links to SoC templates for common workflows. Select a template and save the project. A project folder opens in your workspace containing:

    • A model with the name `soc_*_top.slx` – The top-level model for the SoC project.

    • `referencedmodels` – A folder containing the models referenced from the top model. Some templates include an FPGA model and a processor model. Other templates only include one referenced model: an FPGA model or a processor model.

    • `utilities` – A folder containing utility functions or testbench data used by the model.

    To open the top-level model in Simulink, on the **Project Shortcuts** tab, click **Open Top model**.
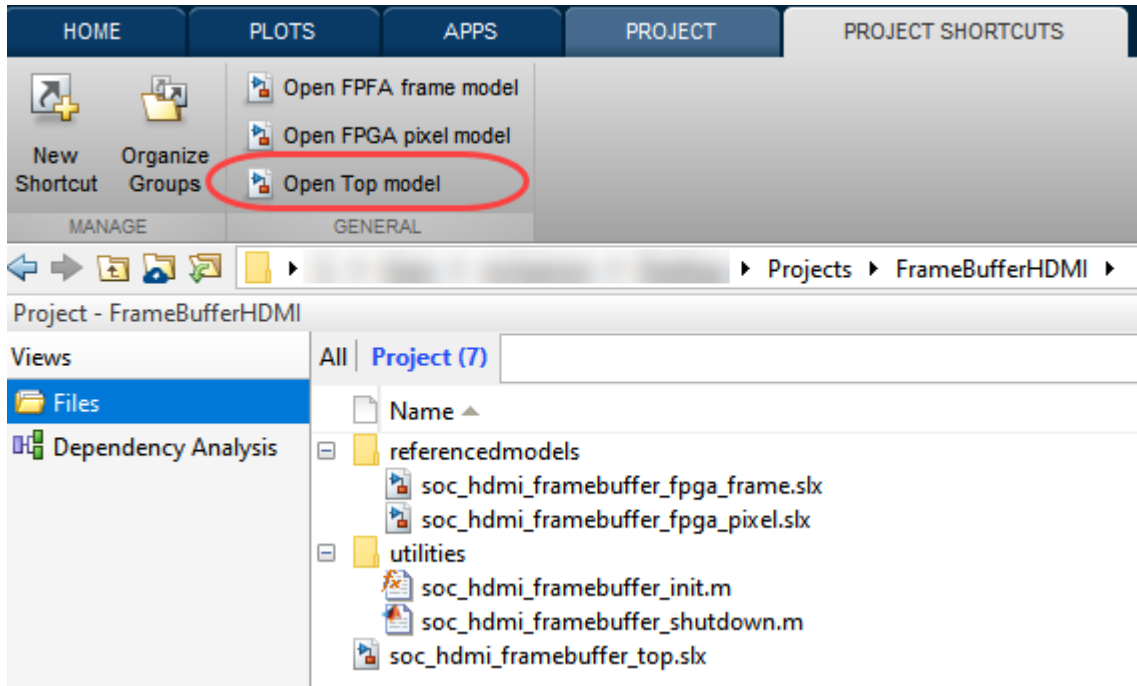
3   In each template, navigate to the blocks marked **FPGA Algorithm** in the FPGA model, or **Processor Algorithm** in the processor model. These blocks are highlighted for easy detection. Replace the template blocks with your own algorithm model.

Tip To easily find the algorithm blocks, follow the annotations throughout the model hierarchy.

4   To open the **SoC Blockset** Block Library, select the Library Browser button, then select **SoC Blockset** in the left pane. Alternatively, at the command line, enter:

```
soclib
```

This library includes blocks for creating SoC models and testbenches.

Template Structure

The top model in an SoC Blockset template includes an FPGA subsystem, which represents the logic intended to program the FPGA. The FPGA subsystem includes two Simulink model variants:

- Frame-based FPGA model – For enhanced simulation performance
- Sample-based FPGA model – For cycle accuracy and code generation

The top model also includes a processor subsystem, which represents the software program intended to run on the SoC processor. Both the FPGA and the top subsystems contain blocks marked as FPGA Algorithm or Processor Algorithm. Replace these algorithms with your own logic. The top model of the template also includes a memory system, with a memory controller and memory channels. These blocks represent the physical memory system on the board. The model often includes a register channel (to enable communication between the processor and FPGA), testbench, or I/O blocks.

Modify Project

Modify the FPGA Model

From the top model, open the FPGA model by clicking the arrow at the bottom left of the FPGA block:
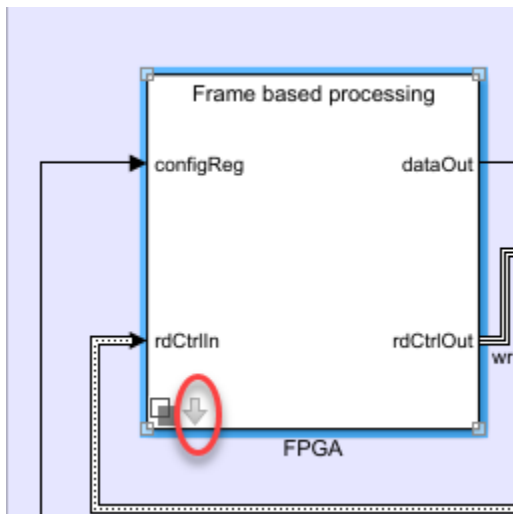
The FPGA model contains two model variants: a frame-based variant and a sample-based variant. Double-click the model variant you want to modify. The FPGA model typically includes two main subsystems for you to modify:

- FPGA Algorithm Wrapper – Double-click to open the model. The algorithm wrapper contains a green-highlighted subsystem named FPGA Algorithm. This block has two inputs and one output and is implemented as a multiplier. Replace this block with your own FPGA algorithm. Add inputs and outputs as required.
- Test Source Wrapper – This block includes a test source and is intended to generate stimulus as an input to the FPGA algorithm. Modify the test source to your needs, or replace it with an alternative source block. If the input to your FPGA algorithm is routed from an I/O block, such as HDMI or SDR, consider using a specific application template.

Note Not all templates include a Test Source block in the FPGA model.

Modify the Processor Model

The processor model includes a Task Manager block and a processor wrapper. The template implements the processor algorithm as a "pass through" wire. Open the processor algorithm wrapper, and replace the Processor Algorithm block (highlighted in blue) with your desired algorithm.

Modify the Register Channel

The top model of a template also includes a register channel to communicate between the processor and the FPGA model. Use the register channel to configure the FPGA mode,l or to read and check status registers. The Register Channel block in the template includes one register. To add additional registers you must modify the register channel block parameters, the FPGA algorithm, and the processor algorithm:

1   Add registers to the register channel – Double-click the Register Channel block to open the block mask and add additional registers as needed. Adding registers creates additional ports on the Register Channel block. For additional information, see Register Channel.

2   Add ports to the processor model – Navigate to the Processor Algorithm Wrapper model. To navigate to the processor model, click **Open Processor model** on the **Project Shortcuts** tab. Double-click Processor Algorithm Wrapper to modify it.

    For write registers, add an output port to the module and add logic to drive a value to the added output port. For read registers, add an input port and logic to process the

information returned from a read register. From the top model, wire the port to the Register Channel block.

3 Add ports to the FPGA model – Navigate to the FPGA Algorithm Wrapper model. To navigate to the FPGA/Frame based processing model, click **Open FPGA sample model** on the **Project Shortcuts** tab. Double-click FPGA Algorithm Wrapper to modify it.

For write registers, add an input port to the module and logic to process the information returned from a read register. For read registers, add an output port and logic to drive a value to the added output port.

For equivalent behavior when using a Simulink sample-based variant, repeat this step for the sample-based processing model in the FPGA wrapper.

4 From the top model, wire the new port to the Register Channel block.

See Also

Memory Controller | Memory Channel | Register Channel | Task Manager

More About

- "Stream from FPGA to Processor Template" on page 1-47
- "SDR Template" on page 1-51
- "HDMI Template" on page 1-40
- "Frame Buffer with HDMI Template" on page 1-43

HDMI Template

The High-Definition Multimedia Interface (HDMI) template provides a simulation model for SoC video streaming using SoC Blockset Support Package for Xilinx® Devices. Use this template to simulate and analyze the effects of internal and external connectivity, such as HDMI I/O behavior on a vision processing algorithm.
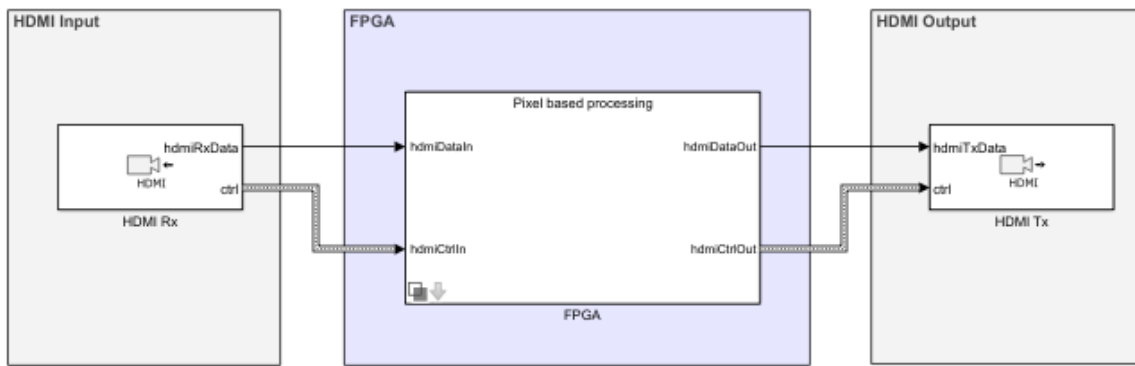
Required Products

- Computer Vision Toolbox™
- Vision HDL Toolbox™
- SoC Blockset Support Package for Xilinx Devices

Template Structure

HDMI video streams from an HDMI Rx block into the FPGA, which implements a video data processing algorithm. The processed images stream to the HDMI Tx block.

FPGA pixel model uses Video Stream Connector blocks to connect different subsystems and to connect to the HDMI I/O blocks. VideoStream Connector required to be able to generate each subsystem as a separate IP in the implemented reference design from the model. Since the FPGA frame model is for simulation purposes only and is not used for implementation, the Video stream connector blocks are not modeled.

Modify Project

In MATLAB, on the **Project Shortcuts** tab, click **Open FPGA pixel model**. Open the `FPGA Algorithm Wrapper`, as shown highlighted in green.
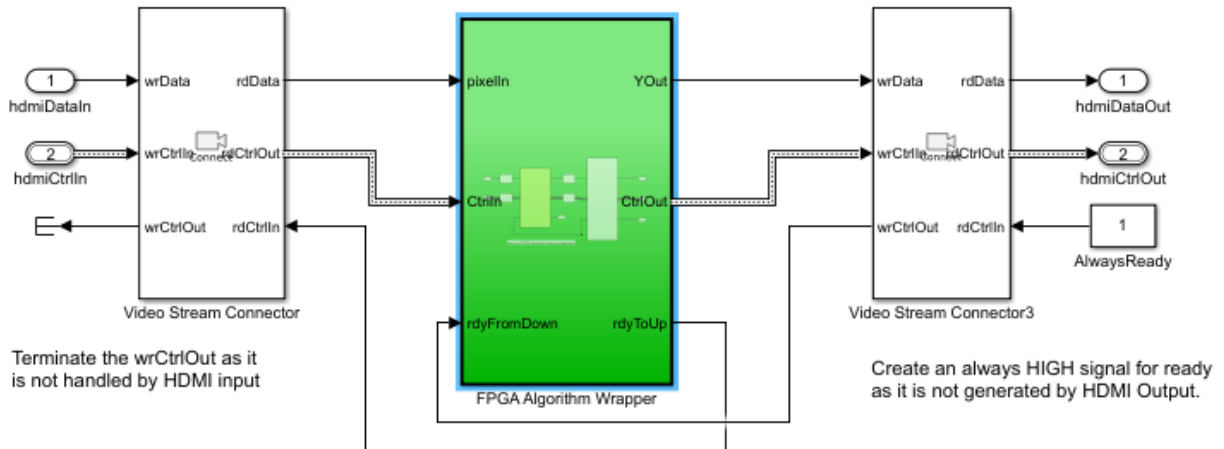
The `FPGA Algorithm`, also highlighted in green, contains feedthrough ports and signals.

Add your FPGA Algorithm in above subsystem   Video Stream FIFO
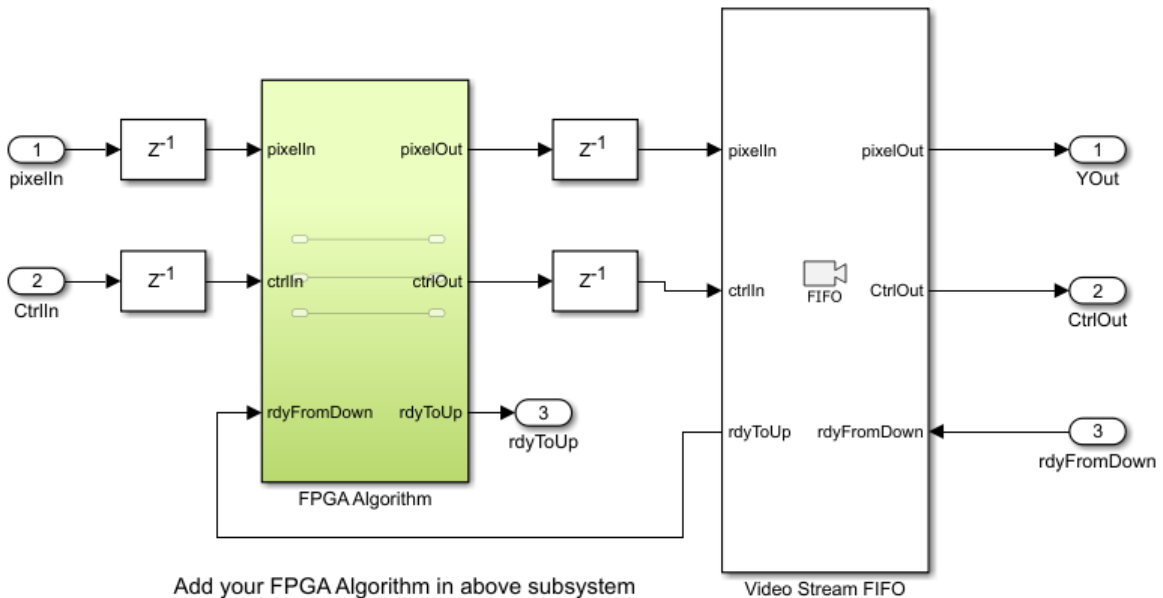
You can modify the content of the FPGA algorithm model to incorporate your desired vision processing algorithm, with complete simulation and code generation of the surrounding video memory system. For pure algorithm design and investigation, click **Open FPGA frame model** in the **Project Shortcuts** tab, and repeat this step.

See Also

"Use Template to Create SoC Model" on page 1-35 | "Create a New Project Using Templates" (Simulink)

More About

• "What Are Projects?" (Simulink)

Frame Buffer with HDMI Template

The Frame Buffer with High-Definition Multimedia Interface (HDMI) template creates a Simulink project with models to simulate and generate a video application with external memory frame buffer. This template forms the base for the "Histogram Equalization Using Video Frame Buffer" on page 5-24 example. Use this template to simulate the full reference design of a video processing application on an FPGA with HDMI I/O and connection to an external memory frame buffer for advanced image processing designs.
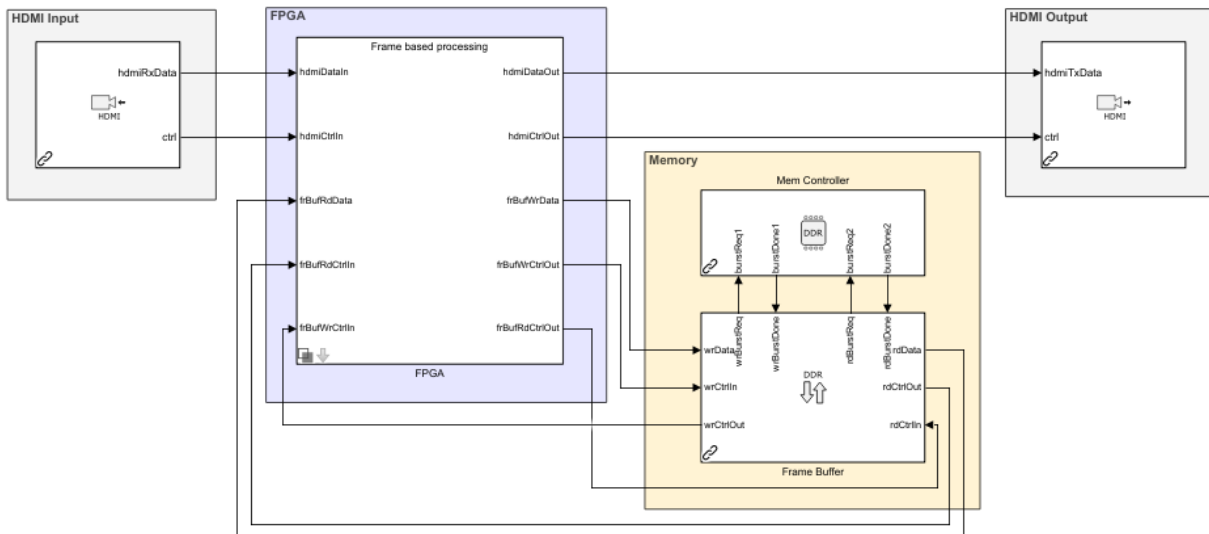
Required Products

- Vision HDL Toolbox
- Computer Vision Toolbox
- SoC Blockset Support Package for Xilinx Devices

Template Structure

HDMI video streams video data from an HDMI Rx block into the FPGA. The FPGA implements a color-space transformation and your image processing algorithm. The processed images then undergo the inverse color-space transformation and stream to the

HDMI Tx block. The FPGA algorithm is connected to the external memory frame buffer Memory Channel block configured in `ALI-Stream Video Frame Buffer` mode.

The FPGA pixel model uses Video Stream Connector blocks to connect different subsystems and to connect to HDMI I/O blocks. This is required to be able to generate each subsystem as a separate IP in the implemented reference design from the model. Since the FPGA frame model is for simulation purposes only and is not used for implementation, the Video Stream Connector blocks are not modeled.

Modify Project

In MATLAB, on the **Project Shortcuts** tab, click **Open FPGA pixel model**. Double-click to open the `FPGA Algorithm Wrapper`.
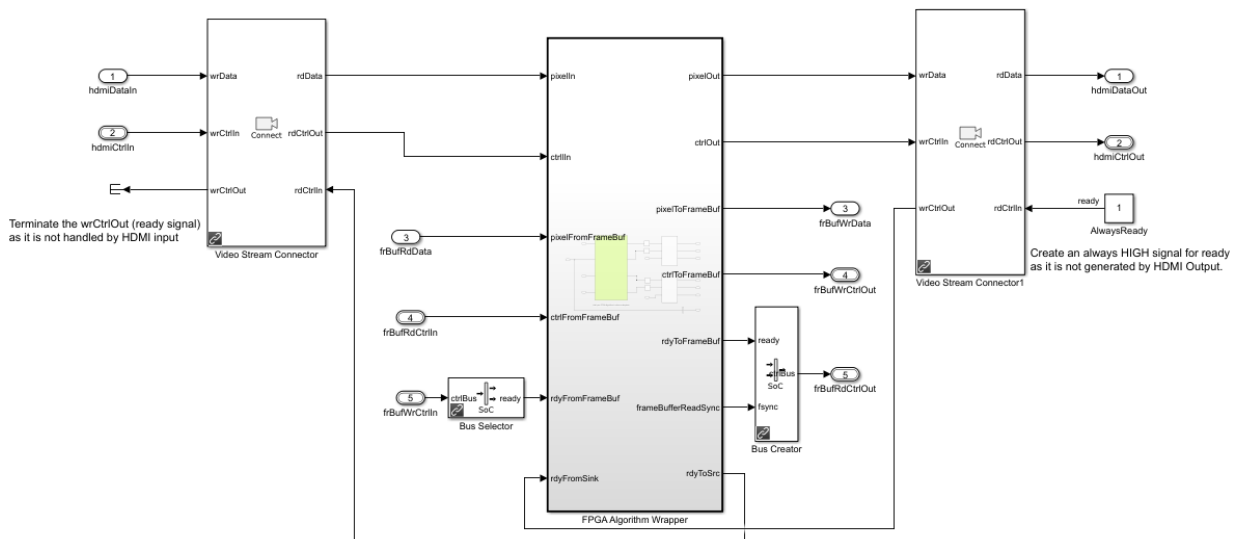
The `FPGA Algorithm`, highlighted in green, contains feedthrough ports and signals.
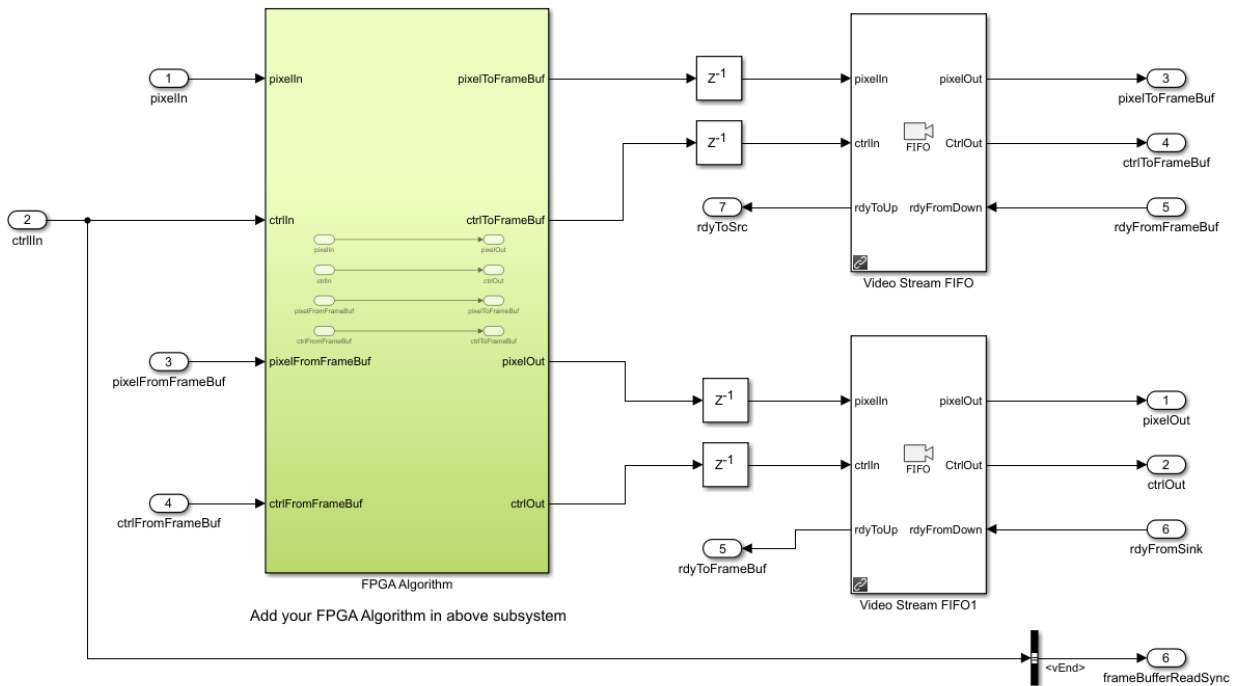
Modify the content of the `FPGA Algorithm` subsystem to incorporate your desired vision processing algorithm, with complete simulation and code generation of the surrounding video memory system. The **pixelToFrameBuf** and **pixelFromFrameBuf** ports provide access to the external memory channel, `Frame Buffer`. For pure algorithm design and investigation, in the **Project Shortcuts** tab, click **Open FPGA frame model**, and repeat this step.

See Also

"Use Template to Create SoC Model" on page 1-35 | "Create a New Project Using Templates" (Simulink)

More About

- "What Are Projects?" (Simulink)

Stream from FPGA to Processor Template

Use the *Stream from FPGA to Processor* template to create an SoC Blockset model for designing a datapath from hardware (FPGA) to software (Processor). To create a project using the "Stream to Processor" template, follow the steps to "Create SoC Model Using SoC Blockset Template" on page 1-35.
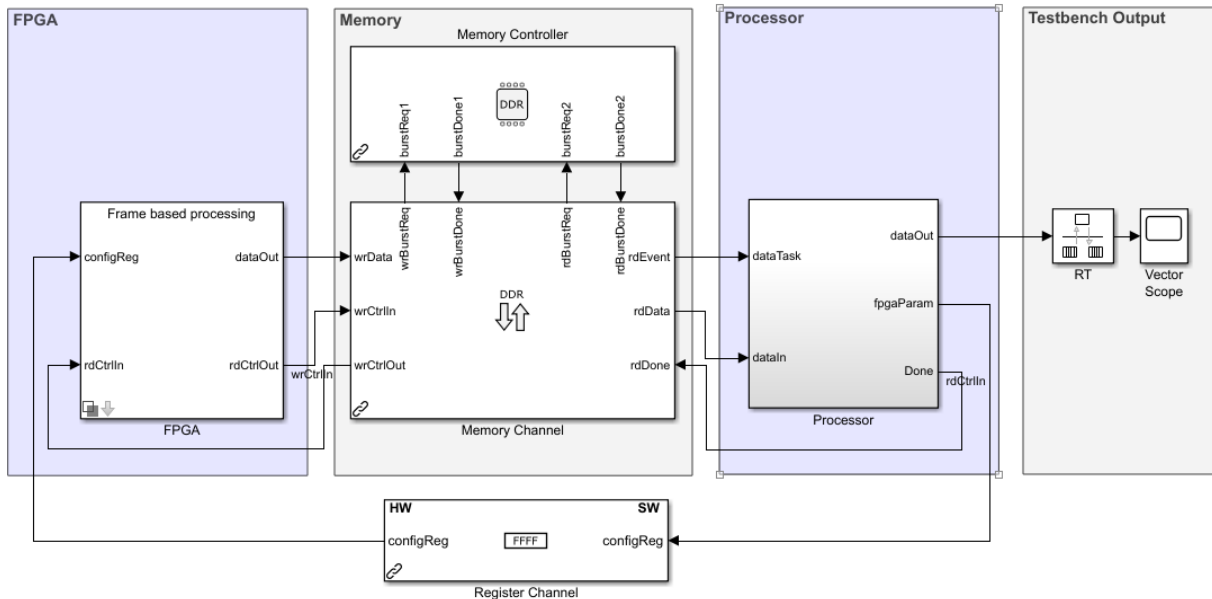
Required Products

For *sample-based* processing, no additional products are required.

For *frame-based* processing, DSP System Toolbox™ is required.

Template Structure

This template models a counter as the test data source and minimal logic for the FPGA and processor algorithms. Use this template as a guide and replace the FPGA algorithm and Processor algorithm with your own functionality. The FPGA algorithm is a simple multiplication performed on input data from the test source and from a **configReg**

parameter. The processor writes the **configReg**. This parameter is modeled using the Register Channel block. Data from the FPGA algorithm is passed to the processor through a Memory Channel block. The memory **Channel Type** parameter is set to `AXI4-Stream to Software via DMA`, which models the DMA data transfer through shared external memory.

The processor reads the computed data from the memory and performs additional computing, which is implemented in the template as a pass-through wire. You can view the simulation results by double-clicking the Vector Scope block in the testbench sink.

Modify Project

Modify the FPGA Model

In the MATLAB toolstrip, on the **Project Shortcuts** tab, click **Open FPGA sample model** to open the FPGA model. In the model, two areas are highlighted green, which represents user code: one in the FPGA Algorithm Wrapper block and one in the Test Source Wrapper block.
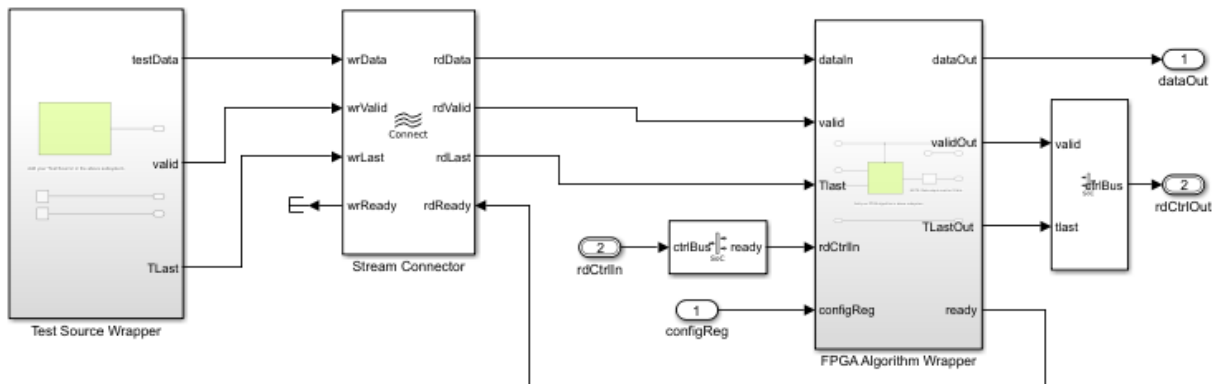
- FPGA Algorithm Wrapper – Double-click to open the model. The algorithm wrapper contains a green-highlighted subsystem named FPGA Algorithm. This block has two inputs and one output and is implemented as a multiplier. Replace this block with your own FPGA algorithm. Add inputs and outputs as required.
- Test Source Wrapper – This block includes a test source and is intended to generate stimulus as an input to the FPGA algorithm. This block is implemented as a counter in this template. Modify the test source to your needs, or replace it with an alternative source block.

> **Tip** When your FPGA model includes more than one IP, you must define each IP as a subsystem and connect the subsystems using a Stream Connector or Video Stream Connector block. For additional information, see "Considerations for Multiple IPs in FPGA Model" on page 1-56.

To enable consistent simulation behavior, click **Open FPGA frame model** in the **Project Shortcuts** tab and repeat this step. To simulate frame-based processing, you must have a DSP System Toolbox license.

Modify the Processor Model

In the MATLAB toolstrip, on the **Project Shortcuts** tab, click **Open Processor model**. The processor wrapper contains a blue highlighted subsystem representing the user code for the processor algorithm. Open the Processor Algorithm wrapper and replace the Processor Algorithm block with your desired algorithm.

Modify the Register Channel

The top model of a template also includes a register channel to communicate between the processor and the FPGA model. Use the register channel to configure the FPGA model, or to read and check status registers. The Register Channel block in the template includes one register. To add additional registers you must modify the register channel block parameters, the FPGA algorithm, and the processor algorithm:

1   Add registers to the register channel – Double-click the Register Channel block to open the block mask and add additional registers as needed. Adding registers creates additional ports on the Register Channel block. For additional information, see Register Channel.

2   Add ports to the processor model – Navigate to the Processor Algorithm Wrapper model. To navigate to the processor model, click **Open Processor model** on the **Project Shortcuts** tab. Double-click Processor Algorithm Wrapper to modify it.

    For write registers, add an output port to the module and add logic to drive a value to the added output port. For read registers, add an input port and logic to process the information returned from a read register. From the top model, wire the port to the Register Channel block.

3   Add ports to the FPGA model – Navigate to the FPGA Algorithm Wrapper model. To navigate to the FPGA/Frame based processing model, click **Open FPGA sample model** on the **Project Shortcuts** tab. Double-click FPGA Algorithm Wrapper to modify it.

    For write registers, add an input port to the module and logic to process the information returned from a read register. For read registers, add an output port and logic to drive a value to the added output port.

    For equivalent behavior when using a Simulink sample-based variant, repeat this step for the sample-based processing model in the FPGA wrapper.

4   From the top model, wire the new port to the Register Channel block.

See Also

More About

SDR Template

The software defined radio (SDR) template provides a simulation model for an SoC reference design available from Communications Toolbox™ Support Package for Xilinx Zynq®-Based Radio. Use this template to simulate the full reference design and analyze the effects of internal and external connectivity on and SDR algorithm, such as memory behavior and Radio Frequency (RF) I/O behavior.

To get started with SoC Blockset model for designing an SDR system, follow the steps to "Create SoC Model Using SoC Blockset Template" on page 1-35.
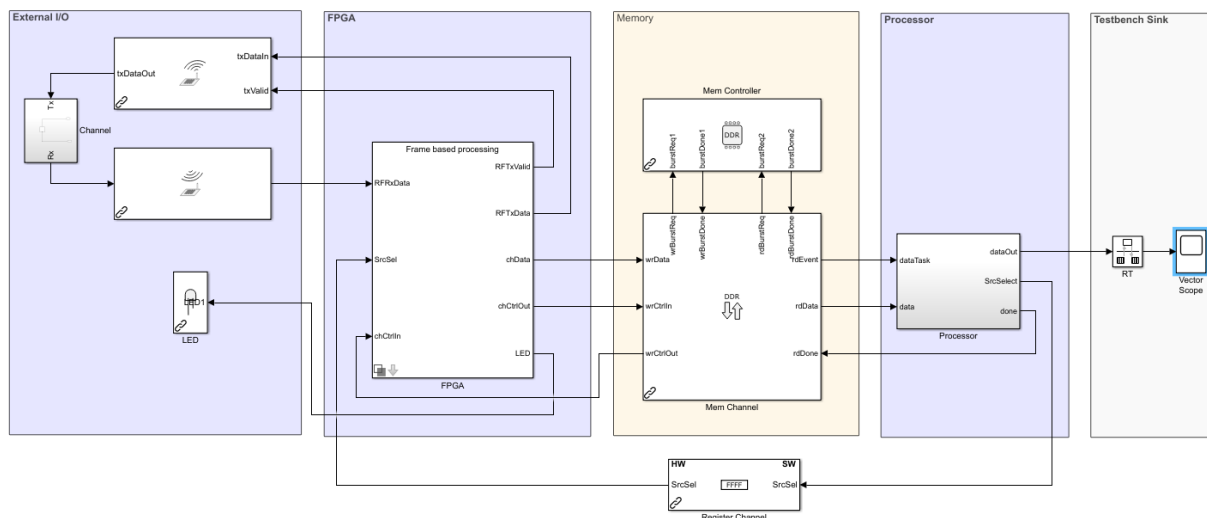
Required Products

- Communications Toolbox
- SoC Blockset Support Package for Xilinx Devices

Template Structure

This template models an SDR transceiver composed of AD9361 transmitter and receiver blocks. The transceiver connects an RF channel to the FPGA, which implements a receiver and a transmitter algorithm. The FPGA algorithm includes a Test Source block,

which generates a sinusoid signal and drives it to the transmitter. The FPGA algorithm also includes a Tx algorithm, implemented as simple pass-through wires, and an Rx algorithm, implemented as a gain block. A configuration register **srcSel** is modeled in the FPGA to select the source of data for the Rx algorithm. The processor writes the **srcSel** register to select either the test source from the FPGA or RF data from the transceiver. This register is modeled using the Register Channel block. Data from the FPGA algorithm is passed to the processor through a Memory Channel block.

Use this template as a guide and replace the Rx Algorithm and Tx Algorithm in the FPGA and the Processor Algorithm in the processor with your own functionality. The memory **Channel Type** parameter is set to `AXI4-Stream to software via DMA`, which models the direct memory access (DMA) data transfer through shared external memory.

The processor reads the computed data from the memory, and performs additional computing (implemented in the template as a pass-through wire). You can view the simulation results by double-clicking the Vector Scope block in the testbench sink.

Modify Project

Modify the FPGA Model

In MATLAB, on the **Project Shortcuts** tab, click **Open FPGA sample model**. Then, open the FPGA Transceiver Algorithm Wrapper. Notice three areas highlighted in green. These areas represent user code and are located in the Receiver Algorithm block, in the Transmitter Algorithm block, and the Test Source block.
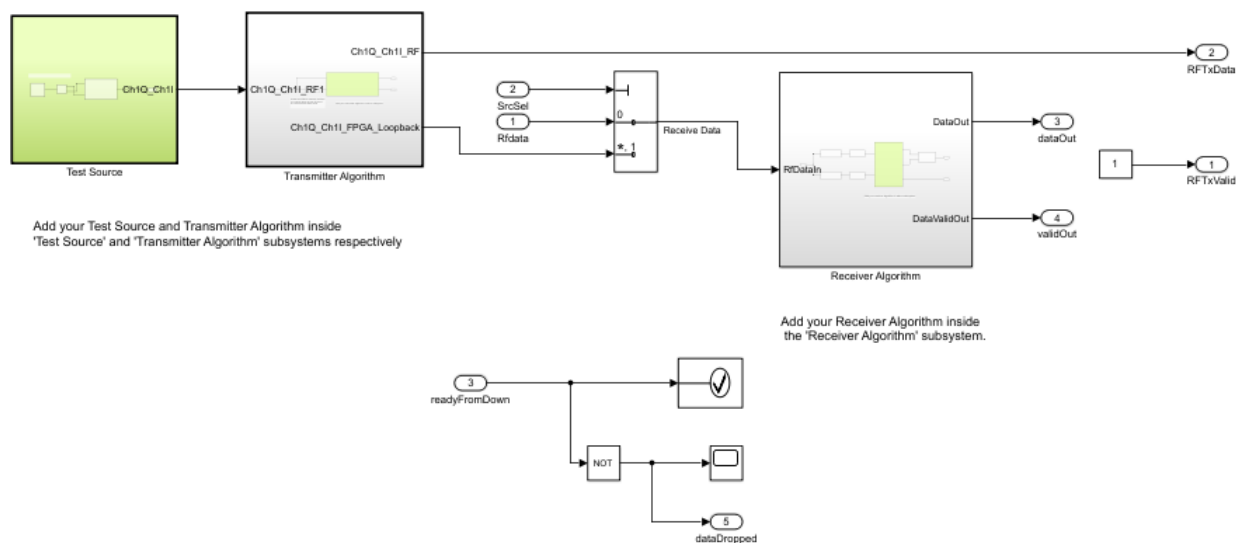
The FPGA model includes the following sections for you to modify (highlighted in green):

- Test Source block – This block generates a 10-kHz sinusoid signal and drives it to the transmitter algorithm. Modify the test source to your needs or replace it with an alternative source block.

- Receiver Algorithm subsystem – Inside the green-highlighted subsystem named Rx Algorithm, there is a block labeled Algorithm. The algorithm takes I/Q data as input and output with a valid signal. Replace this block with your own Rx algorithm.

- Transmitter Algorithm – Inside the green-highlighted subsystem named Tx Algorithm, the algorithm has an input from the test source and two output signals: one to the RF channel and one to the FPGA. Replace this block with your own Tx algorithm.

To enable consistent simulation behavior, in the **Project Shortcuts** tab, click **Open FPGA frame model** and repeat this step.

Modify the Processor Model

In MATLAB, on the **Project Shortcuts** tab, click **Open processor model**. The subsystem highlighted in blue represents the user code for the processor algorithm. Open the Processor Algorithm wrapper and replace the internal Processor Algorithm block (also highlighted in blue) with your desired algorithm.

Modify the Register Channel

The top model of a template also includes a register channel to communicate between the processor and the FPGA model. Use the register channel to configure the FPGA mode,l or to read and check status registers. The Register Channel block in the template includes one register. To add additional registers you must modify the register channel block parameters, the FPGA algorithm, and the processor algorithm:

1   Add registers to the register channel – Double-click the Register Channel block to open the block mask and add additional registers as needed. Adding registers creates additional ports on the Register Channel block. For additional information, see Register Channel.

2   Add ports to the processor model – Navigate to the Processor Algorithm Wrapper model. To navigate to the processor model, click **Open Processor model** on the **Project Shortcuts** tab. Double-click Processor Algorithm Wrapper to modify it.

    For write registers, add an output port to the module and add logic to drive a value to the added output port. For read registers, add an input port and logic to process the information returned from a read register. From the top model, wire the port to the Register Channel block.

3   Add ports to the FPGA model – Navigate to the FPGA Algorithm Wrapper model. To navigate to the FPGA/Frame based processing model, click **Open FPGA sample**

model on the **Project Shortcuts** tab. Double-click FPGA Algorithm Wrapper to modify it.

For write registers, add an input port to the module and logic to process the information returned from a read register. For read registers, add an output port and logic to drive a value to the added output port.

For equivalent behavior when using a Simulink sample-based variant, repeat this step for the sample-based processing model in the FPGA wrapper.

4 From the top model, wire the new port to the Register Channel block.

Considerations for Multiple IPs in FPGA Model

When your FPGA model includes more than one block for which you'd like to generate HDL using HDL Coder, you must use a connector model to connect your blocks.

For additional information, see Stream Connector and Video Stream Connector blocks.

Create an SoC Project Application

A system-on-chip (SoC) project developed using the SoC Blockset typically contains many diverse systems that make up a the complete application. These systems can include:

- Embedded processors with timer-driven and event-driven tasks.
- FPGAs with custom IP logic and timing.
- External memory systems with interaction to embedded processors and FPGAs.
- I/O device interaction, such as TCP/IP and UDP connections.

This example shows the steps to create an SoC application, using the features of the SoC Blockset, as a Simulink project. To begin, see "Project and Top-Level Model" on page 1-58.

Note This project is equivalent to the project automatically created by the "Stream from FPGA to Processor Template" on page 1-47. Templates are the recommended and preferred method for creating new projects. This example should be used for information purposes only.

See Also

"Use Template to Create SoC Model" on page 1-35 | "Stream from FPGA to Processor Template" on page 1-47

Project and Top-Level Model

An SoC application model developed using the SoC Blocksetcombines multiple subsystems and reference models. Each subsystem and reference model maps to a particular feature of an SoC device. Organization of the models and shared configuration settings requires a Simulink project.

1 Create a new SoC Blockset project named `SampleSoCApplication`. Creating a new project automatically creates a new project folder with the same name. For more information on creating projects, see "Create a New Project From a Folder" (Simulink).

2 Open a new Simulink model. Save the model as `soc_hwsw_top.slx` into the project folder.

3 Open the configuration parameters of the model. Select the **Hardware Implementation** tab and set the **Hardware board** to one of the supported boards for SoC Blockset, such as `Xilinx Zynq ZC706 evaluation kit`. Selecting the **Hardware board** automatically sets several parameters in the model related to that hardware board.

4 In the **Solver** tab, set **Solver selection** > **Type** to `Variable-step`.

5 Create three box areas and label them as `FPGA`, `Memory`, and `Processor`. For more information on creating box areas, see "Box and Label Areas of a Model" (Simulink). In the following sections, these areas are populated for various aspects of your SoC application.
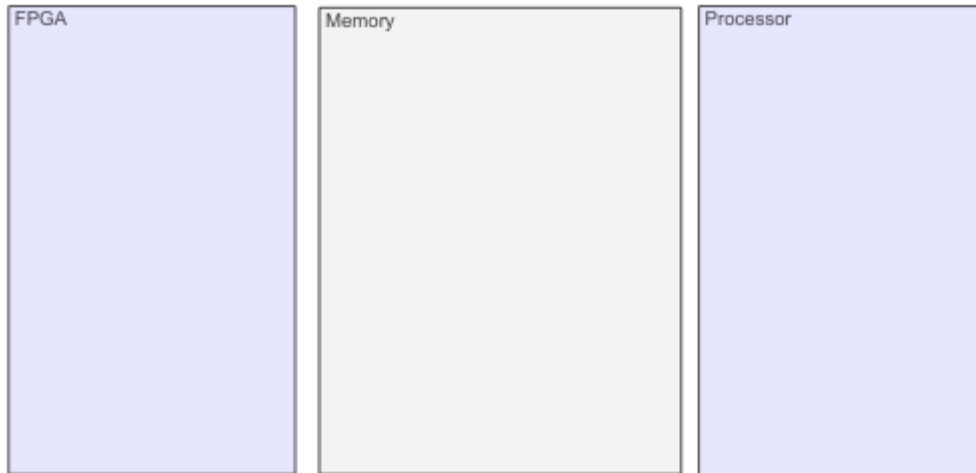
6 Create a new MATLAB function to initialize variables used throughout the project.

```matlab
function soc_hwsw_init
% Initialize the model wide variables and set them in base workspace.

SourceSTime = 1e-7;

FrameSize = 1000;
ProcSTime = SourceSTime*FrameSize;
FPGASTime = SourceSTime;
FPGAFrameSize = 1;

assignin('base','ProcSTime',ProcSTime);
assignin('base','FPGASTime',FPGASTime);
assignin('base','SourceSTime',SourceSTime);
assignin('base','FPGAFrameSize',FPGAFrameSize);
assignin('base','FrameSize',FrameSize);

end
```

In the project folder, save the file as soc_hwsw_init.m in a new subfolder, utilities.

See Also

"Software and Task Management on Processor" on page 1-61

More About

- "Create a New Project From a Folder" (Simulink)
- "Box and Label Areas of a Model" (Simulink)

Software and Task Management on Processor

The processor system in this SoC application reads data from the external memory following a write from the FPGA to that memory. Since FPGA writes and interaction with external memory are asynchronous, the processor uses an event-driven task to read from memory. The software also manages a register on the FPGA that specifies a multiplication factor to be used in the FPGA algorithm.

Processor Model

1 Open a new Simulink model. Save the model as `soc_hwsw_proc.slx` into a new subfolder, named `processor`, in the project folder.

2 Open the configuration parameters of the model. Select the **Hardware Implementation** tab and set the **Hardware board** to one of the supported boards for SoC Blockset, such as `Xilinx Zynq ZC706 evaluation kit`.

Note The processor model must use the same hardware board and solver configuration parameter settings as the top level model.

3 In the model, using a Function-Call Subsystem block, Asynchronous Task Specification block, Inport block, and Outport blocks, create the following system.
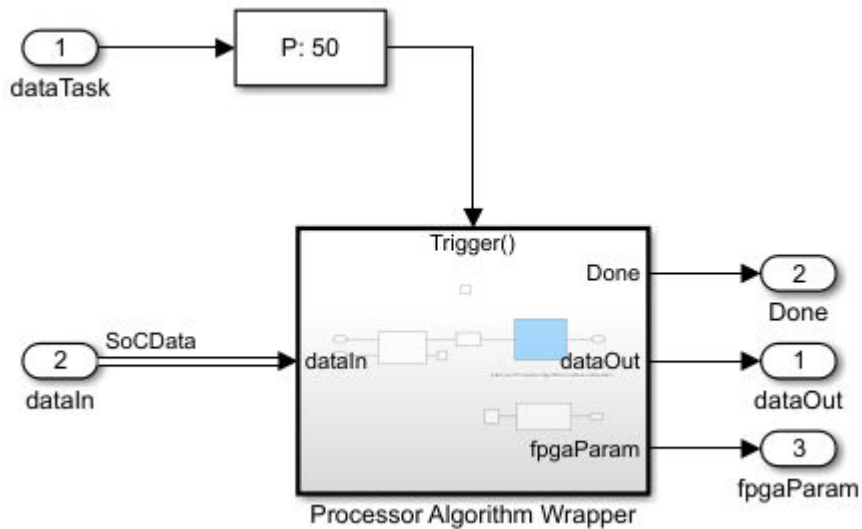
4 In the `dataTask` block dialog mask, check **Signal Attributes** > **Output function call** to expose a function call port on the outside model.

5 In the Asynchronous Task Specification block dialog mask, set **Task priority** to 50.

Note The task priority of the Asynchronous Task Specification block must match the priority of task in the Task Manager block driving this task.

Task Processing

The `Processor Algorithm Wrapper` subsystem reads data from the external memory only after each write to the external memory by the FPGA.

1 Open the `Processor Algorithm Wrapper` block.

2 Using a Stream Read block, Constant block, Data Type Conversion block, and Subsystem blocks, create the following model.

NOTE: Data input is always 32 bits.
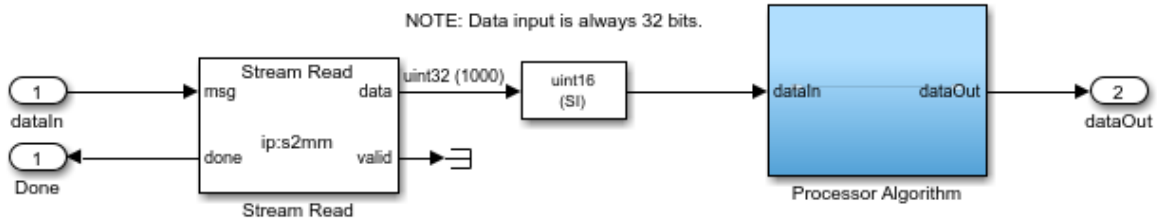
Add your Processor Algorithm in above subsystem

3 Open the Stream Read block dialog mask. Set **Number of buffers** to 6.

4 Open the Data Type Conversion block dialog mask and set **Output data type** to `uint16`.

5 The `Processor Algorithm` subsystem serves as a base to develop your own processing algorithm.

6 Open the `Register Channel Write` subsystem block.

7 Add a Register Write block to create the following model.
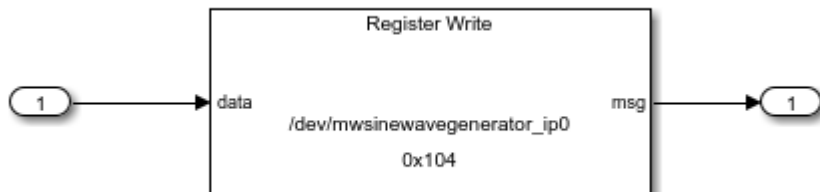
8 Open the Register Write block dialog mask. Set **Device name** to /dev/
mwsinewavegenerator_ip0 and **Offset address** to hex2dec('100').

Top Model

1 In the project folder, open the model soc_hwsw_top.slx.

2 Add a Subsystem block into the Processor area and label the block Processor.

3 In the Processor subsystem, using the Task Manager block and Model block, create
the following system.
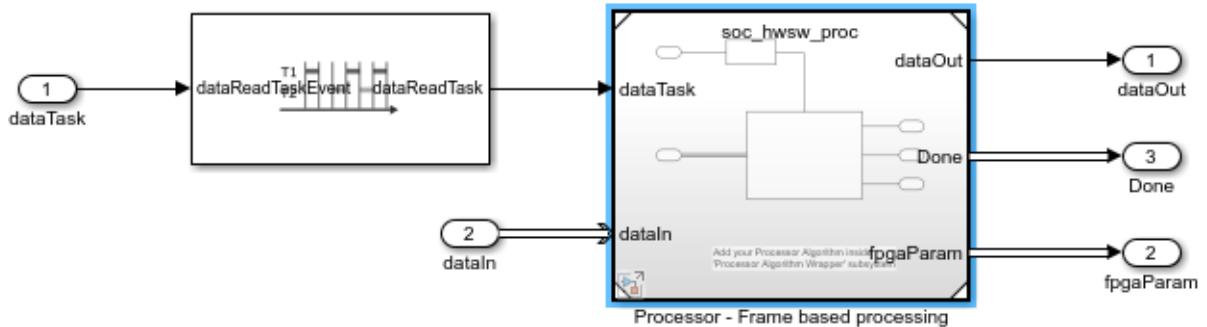
4 Open the Model block dialog mask and set **Model name** to soc_hwsw_proc.slx.

5 Open the Task Manager block dialog mask. Set the task **Name** to dataReadTask and
set the **Priority** to 50. In the **Simulation** tab, set the **Mean**, **Min**, and **Max** to
8e-05. Click **OK**.

See Also
Task Manager

More About
- "What is Task Execution?" on page 1-2
- "Event-Driven Tasks" on page 1-4
- "Task Duration" on page 1-14

User Logic on FPGA

In this SoC project example, the FPGA generates test data and process it in FPGA algorithm before passing it to processor using shared memory.

Sample Based Model

1. Open a new Simulink model. Save the model as soc_hwsw_fpga_sample.slx into the subfolder, named referencedmodels, in the project folder.

2. Open the Simulink configuration parameters to the **Hardware Implementation** panel. Set **Hardware board** to None and set **Device vendor** to ASIC/FPGA. In the **Solver** tab, set **Solver selection > Type** to Fixed-step. Click **OK** to apply the changes and close the configuration parameters.

Note SoC Blockset requires that the FPGA reference models specify the intended deployment hardware, in this case an FPGA.

---

3. In the new model, using Stream Connector block, SoC Bus Selector block, SoC Bus Creator block, and Subsystem blocks, create the following system.
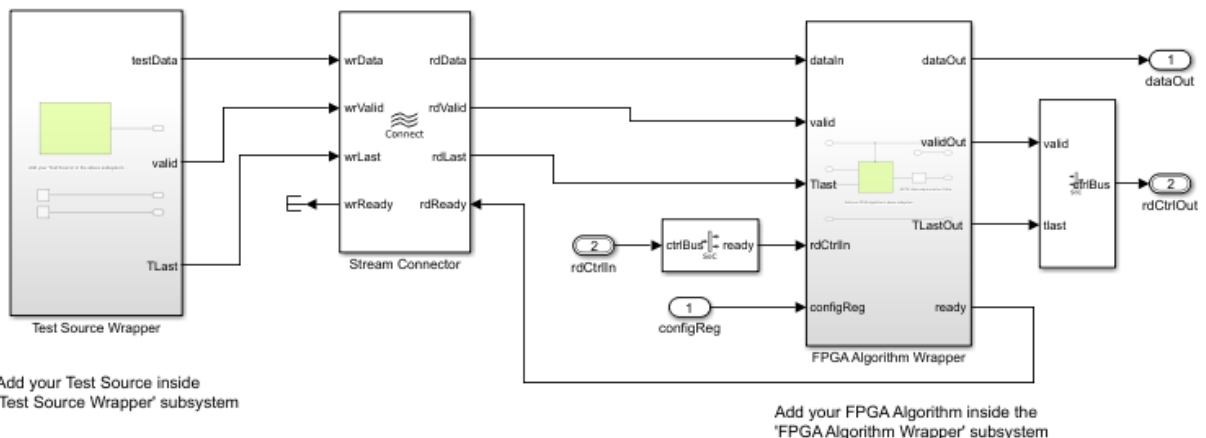
Tip When your FPGA model includes more than one IP, you must define each IP as a subsystem and connect the subsystems using a Stream Connector or Video Stream

Connector block. For additional information, see "Considerations for Multiple IPs in FPGA Model" on page 1-56.

In the SoC Bus Creator block dialog mask, set **Control type** to Valid.

4   The Test Source subsystem simulates a free-running counter. Open the Test Source subsystem and create the following system.
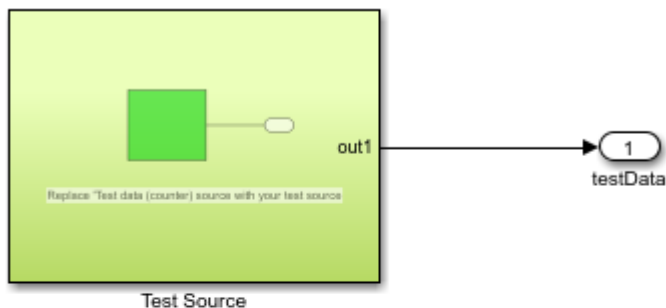
Add your 'Test Source' in the above subsystem
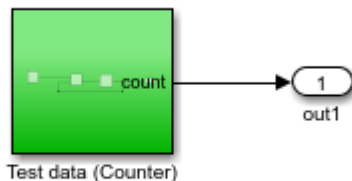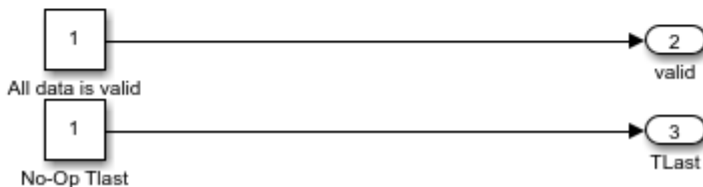
Replace 'Test data (counter) source with your test source
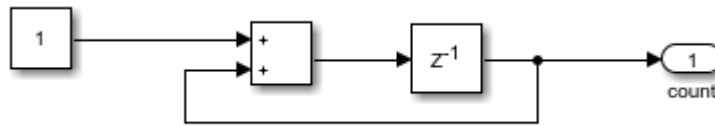
5   The FPGA Algorithm subsystem simulates the multiplication of streamed data. Open the FPGA Algorithm subsystem and using an Enabled Subsystem, Logical Operator, and Data Type Conversion blocks, create the following system.
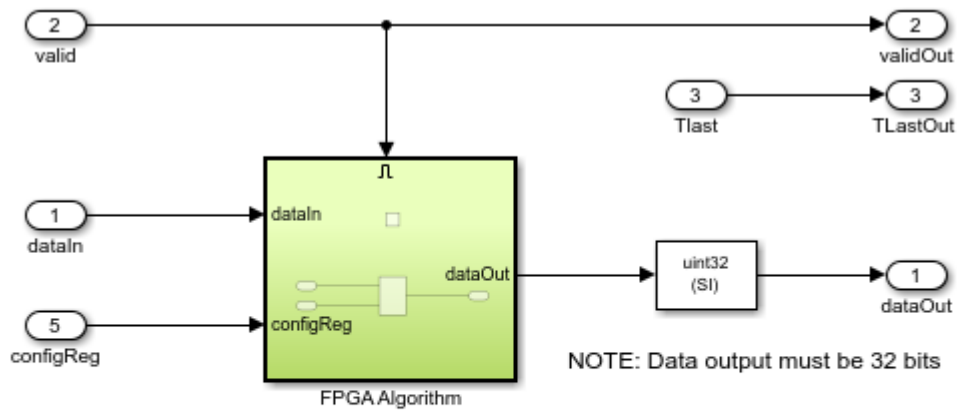
Top Model

1   In the project folder, open the model soc_hwsw_top.slx.

2   Add a Subsystem block into the FPGA area and label the block FPGA.

3   In the FPGA subsystem, using the Model block, create the following system.

Sample based processing
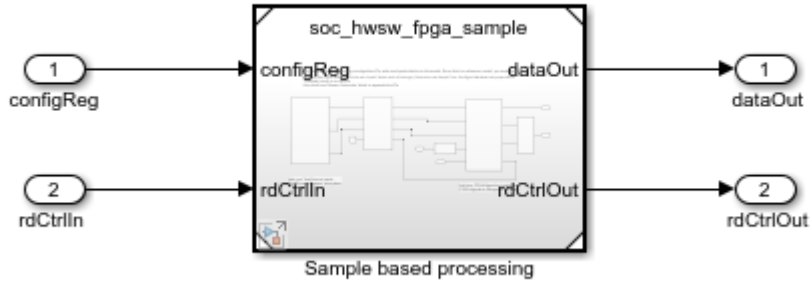
4 Open the Model block dialog mask and set **Model name** to
   `soc_hwsw_fpga_sample.slx`.

The "Stream from FPGA to Processor Template" on page 1-47, the FPGA subsystem uses
a model variant to select between the sample based model developed in this section and a
frame based model. The frame based model allows faster simulations but does not
support code generation.

See Also
SoC Bus Creator | SoC Bus Selector | Stream Connector

More About
• "AXI4-Stream Interface" on page 1-27

Memory and Register Channel Connections

The memory channel models the data transfer from FPGA to processor using shared external memory. The register channel models the control of FPGA logic from processor. You can both configure the FPGA logic and read the status of FPGA logic from processor. The following sections show how to connect create these channel connections.

Memory Channel Connection

1. Open the `soc_hwsw_top.slx` model.
2. Add a Memory Channel block and a Memory Controller block to the `Memory` area. Together, these blocks model the memory connection through `DDR` between the processor and FPGA sides of your application.
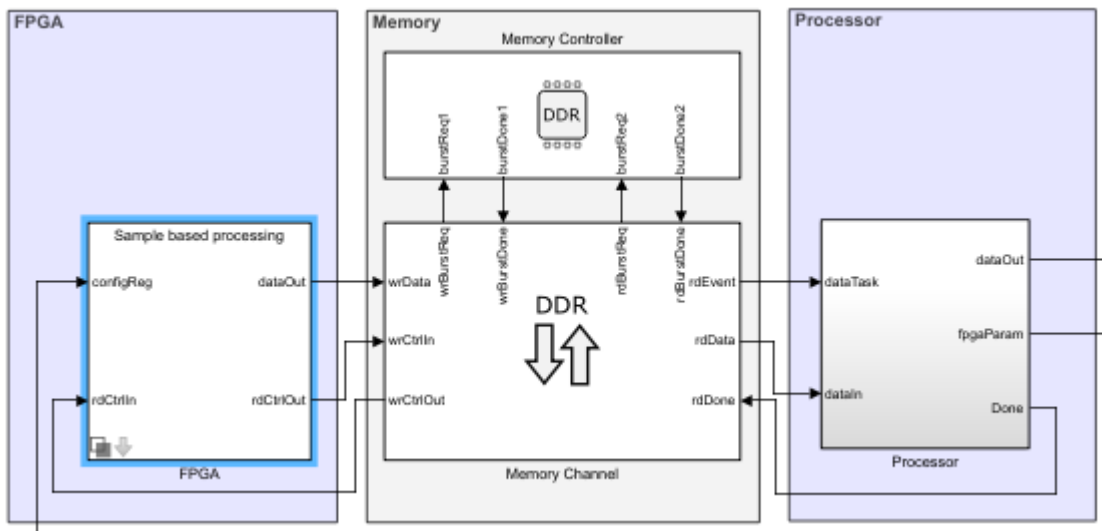
3. Open the Memory Controller block dialog mask. Set **Number of masters** to 2. In the **Advanced** tab, the Memory Controller automatically inherits parameters from the **Hardware board** specified in the model configurations.
4. Connect the pair of Memory Controller burst ports, `burstReq` and `burstDone`, to the read and write burst request ports of the Memory Channel block.
5. In the model, open the Memory Channel block dialog mask. Set **Channel type** to `AXI4-Stream to Software via DMA`. Set **Buffersize (bytes)** to `FrameSize*4` and **Number of buffers** to 6. Click **OK**.

Register Channel Connection

1 Add a Register Channel block to the model and connect the block to the `Processor` and `FPGA` subsystems as shown in the following image.
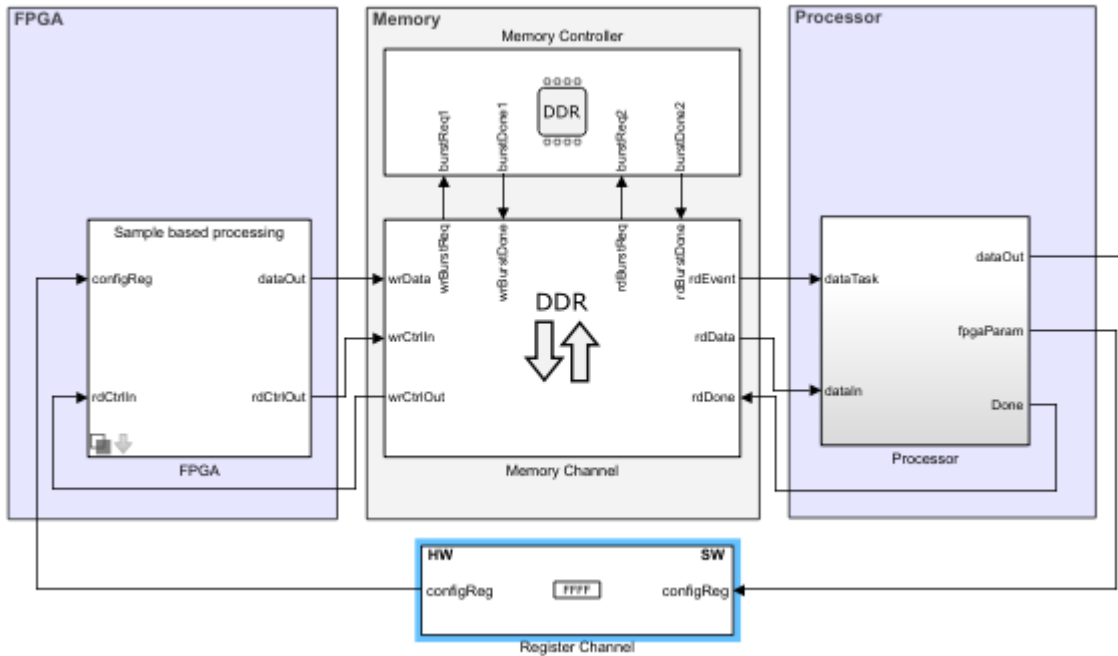
2 Open the Register Channel block dialog mask. Add two new registers with these properties.

Register	Direction	Data type	Dimension
configReg	Write	uint8	1

Set **Register write sample time** to `FPGASSTime`. Click **OK**. This sample time is set in the file `soc_hwsw_init.m`.

See Also

Memory Controller | Memory Channel | Register Channel

More About

- "Memory and Register Channel Connections" on page 1-69

Simulation and Analysis

This set of steps runs the `soc_hwsw_top.slx` model created in the previous steps. A visual of the processor output data shows the complete SoC application.

1. In the project folder, open the model `soc_hwsw_top.slx`.
2. Using a Scope block and Rate Transition block, update the model as shown in this diagram.
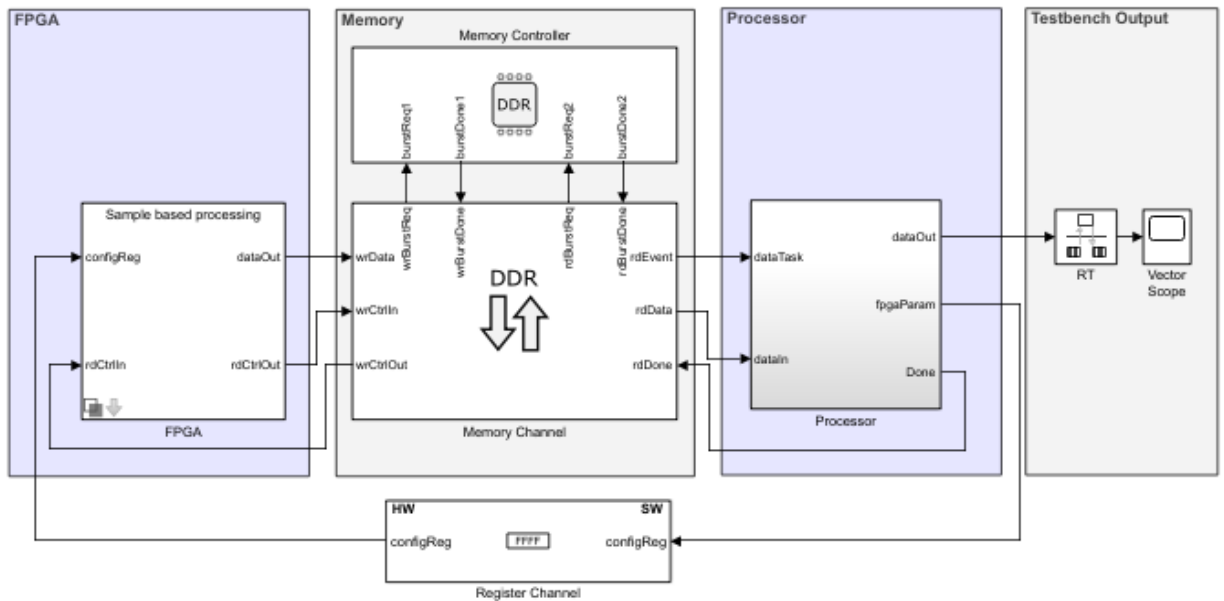
3. Run the model and open the `Vector Scope`.

The display in the `Vector Scope` shows the counter output.

See Also

"Use Template to Create SoC Model" on page 1-35 | "Stream from FPGA to Processor Template" on page 1-47

Entities in an SoC Blockset Model

Entities are discrete items of interest in a discrete-event simulation. By definition, these items are called entities in the SoC Blockset. Entities can pass from memory blocks, such as the Memory Channel or Memory Controller block, to processor blocks, such as the Stream Read or Task Manager block during a simulation. Entities can carry data, information, messages, and signals through a system.

Visualize Entities

Entities do not appear explicitly in the model window. A graphical block can represent a component that processes entities, but entities themselves do not have a graphical representation. However, you can gather information about entities using Simulink scopes. You cannot branch an entity connection line.

Generators and Terminators

In the SoC Blockset, many blocks that can produce entities, known as Generators, and consume entities, known as Terminators. A entity generator and terminator require matching data types to pass entities. This table shows the matched pairs of generator and terminator blocks.

Generator	Terminator	Data Type
IO Data Source	TCP Read	
	UDP Read	
	Register Read	`SoCData`
	Stream Read	`SoCData`
	Task Manager	`rteEvent`
Memory Channel		
Memory Controller		
Memory Traffic Generator		
Register Write		
Register Channel		

See Also

Custom Hardware Board Configuration

A custom hardware board is a hardware board that not explicitly supported as a default selection in SoC Blockset. To create an SoC project to simulate a custom hardware board, configure a Simulink project as follows:

1 Create or open an existing SoC project. For more information on creating SoC projects, see "Use Template to Create SoC Model" on page 1-35.

2 In the top level model, open the Simulink configuration parameters dialog. In the **Hardware Implemention** panel, set **Hardware board** to `Custom Hardware Board`.

3 In the **Hardware Implemention** panel, open the **Target hardware resources > Processor** group. Set Number of cores to match the number of cores available on your SoC processor. The cores available in your processor can be found from the SoC manufacturer's data sheet.

4 Open the **Target hardware resources > FPGA design (mem controllers)** group and set the "FPGA design (mem controllers)" configuration parameters according to your SoC specifications. For information on deriving "FPGA design (mem controllers)" parameters, see the Memory Controller block which shares these parameters.

5 Open the **Target hardware resources > FPGA design (mem channel)** group and set the "FPGA design (mem channels)" configuration parameters according to your SoC specifications. For information on deriving "FPGA design (mem channels)" parameters, see the Memory Channel block which shares these parameters.

Note The `Custom hardware board` selection only supports simulation. For code generation, use one of the provided SoC Blockset hardware board selections.

See Also

"Hardware Implementation Pane"

Build Error for Accelerator Mode and Rapid Accelerator Mode

SoC Blockset does not support "Accelerator Mode" (Simulink) or "Rapid Accelerator Mode" (Simulink) simulation of models. Attempting to use SoC Blockset blocks and features in model running accelerator mode or rapid accelerator mode results in undefined behavior.

In SoC Blockset models, set the simulation mode to normal mode or external mode.

See Also

More About

- "Accelerator Mode" (Simulink)
- "Rapid Accelerator Mode" (Simulink)

2

Simulate SoC Applications

Task Overruns and Countermeasures

With finite processing resources available in a system, an execution instance of a task might not be able to complete before the start of the next task instance. This task overrun results in the start of the next instance of the task execution to be delayed. As a result, the next task must catch-up to avoid another overrun. This diagram shows a simplified execution of two tasks: a high-priority event-driven task and a low priority timer-driven task.
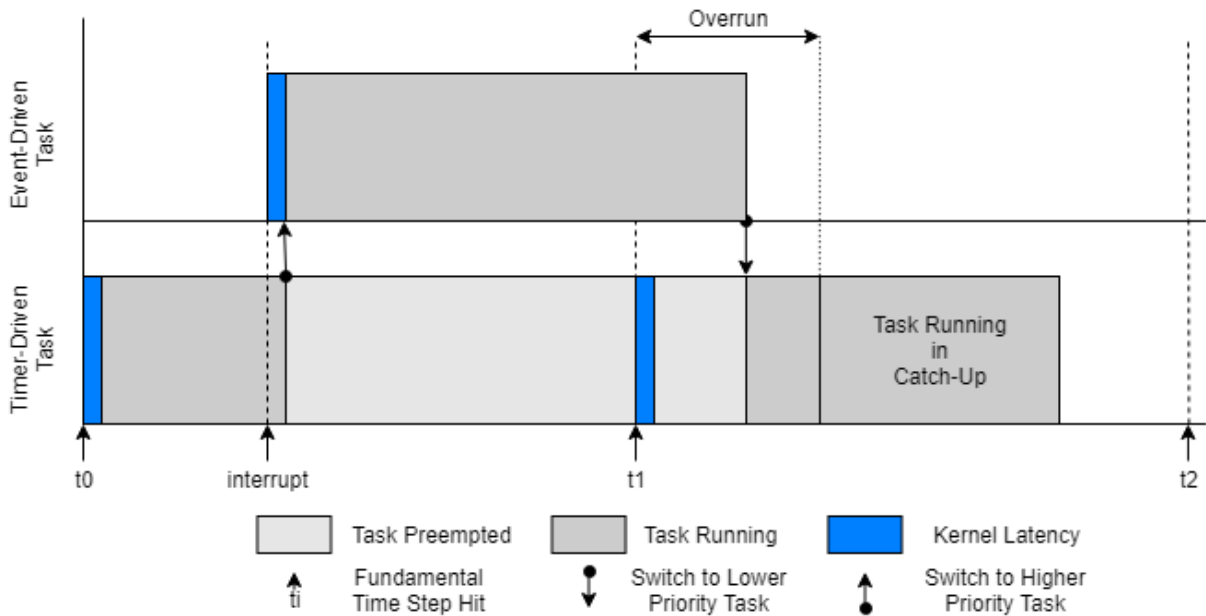
Due to the long execution time of the event-driven task, the first execution instance of the timer-driven task overruns into the start of the next execution instance. This overrun puts the second execution instance into catch-up mode.

When tasks overrun repeatedly, an execution backlog can develop in the application, potentially breaking the system. These sections discuss typical countermeasures to either reduce the chance of task overruns or handle situations when tasks overrun, preventing an execution backlog.

Reduction of Task Execution Interval

For timer-driven tasks, reduce the chance of overruns by providing the task with more execution time. Increase available execution time by decreasing the task rate, which is equivalent to increasing the time between task execution instances. This extra time provides each task execution instance a better chance of running to completion, even in the presence of other tasks. The rate of a timer-driven task can be adjusted in the Task Manager block by setting the **Period** parameter.

Reduction of the task execution interval cannot be guaranteed in all cases. Some of these cases include:

- For event-driven tasks, multiple events can occur at the same time, depending on the priority of the event-driven task. This case forces other tasks to overrun due to lack of resources.
- Real-time requirements where a task, timer or event driven, must respond to the latest event trigger signal and new data regardless of whether previous task instances completed. This case fixes the task execution interval to a value determined by the design requirements.

In these cases, distributing tasks across multiple processor cores or allowing tasks to drop can be advantageous depending on the design requirements.

Distribution of Tasks Across Multiple Processor Cores

Most modern embedded processors provide multiple cores where tasks can be executed. By distributing tasks across these multiple processor cores, tasks can run simultaneously without directly competing for processing resources and reducing the chance of task overruns. In SoC Blockset, a task can be set to run on a specific processor core in the Task Manager block by setting the **Core** parameter to the core number. For more information on the selection, execution, and visualization of tasks on multiple cores, see "Multicore Execution and Core Visualization" on page 2-14.

Dropping Overrunning Tasks

In some designs, a task must execute when the task trigger signal occurs or with the latest state of the system. If a task has been triggered and a new task trigger occurs, the new instance can be removed or *dropped*. After dropping the execution instance of the task that overran the next execution instance starts when the event trigger signal arrives.

To drop tasks when an overrun occurs, in the Task Manager block, enable the **Drop task that overrun** parameter.

Task Drops in Simulation

This example shows how to configure a task in the Task Manager block to drop when a task overrun occurs during simulation.

Task Overrun Without Task Drops

This model simulates a software application running on an ARM processor. A Task Manager block schedules the execution of the Timer Driven Subsystem, inside the Software Application Model Reference block. A Random Number block simulates a data source that the timer-driven task samples.
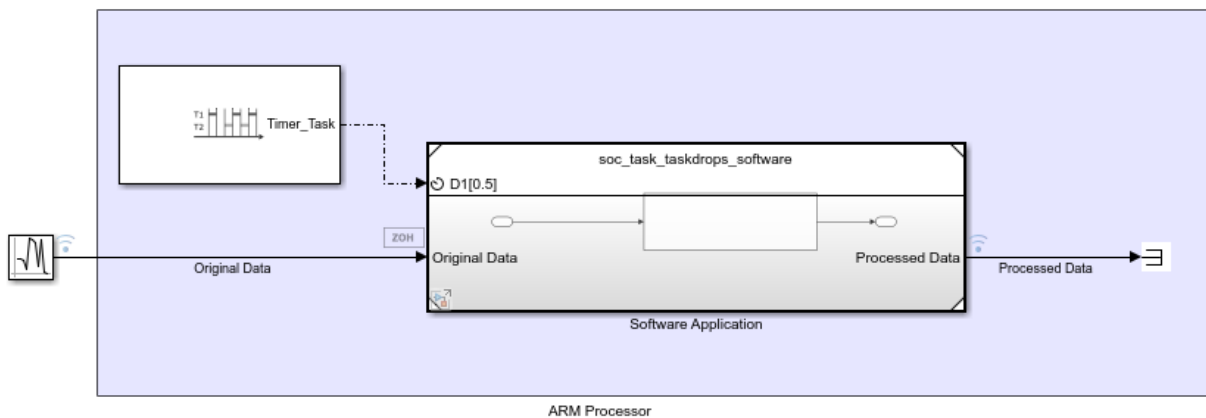
In this model, the task duration of 0.6 seconds exceeds the task period of `0.5` seconds causing the task to overrun. Click the Run button to build and run the model. When the model finishes running, the Simulation Data Inspector (SDI) display shows the task execution timing.
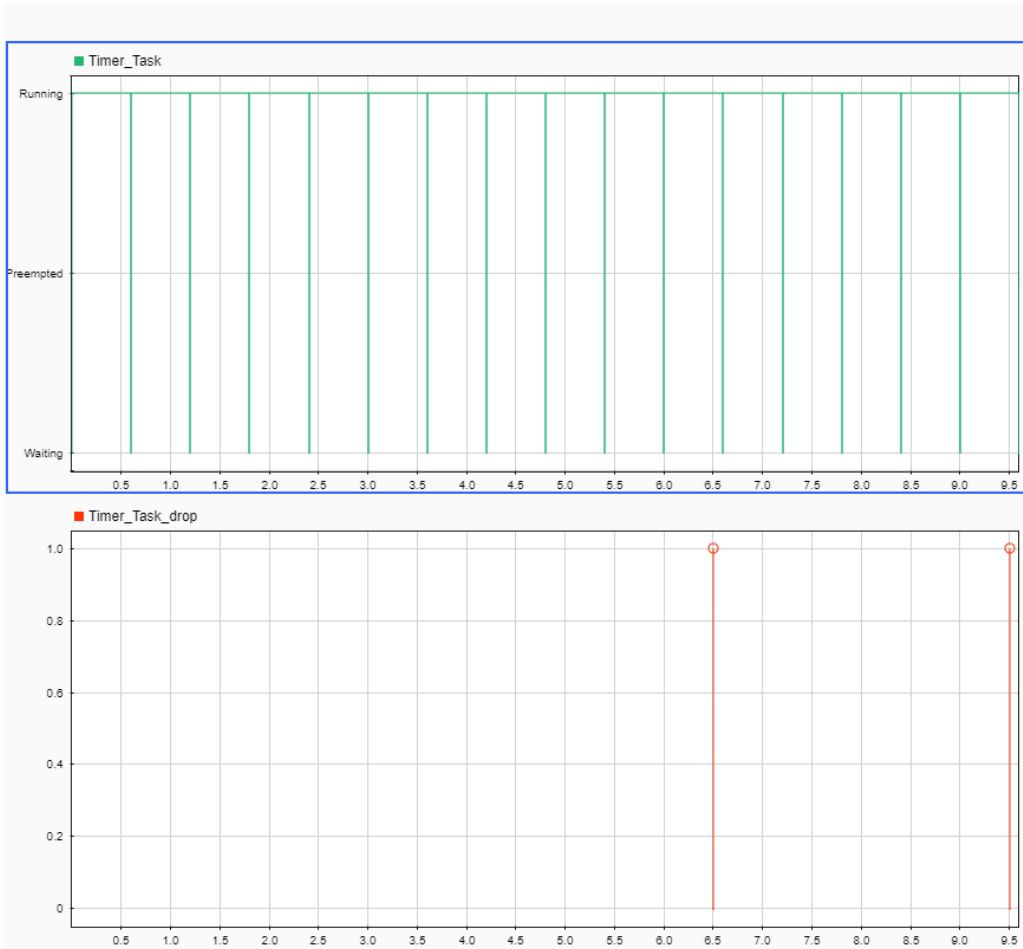
Inspecting the execution timing of the tasks shows that the start of each following task instance is delayed from the expected `0.5`-second interval by the overrun of the previous task.

Task Overrun With Task Drops

Using the same previously shown model, rather than overrunning the timer-driven task, the task drops so the next task instance starts at the 0.5-second interval. Open the Task Manager block dialog mask, and select **Drop tasks that overrun**. Run the model again. Open the SDI to view the task execution and dropped task instances.
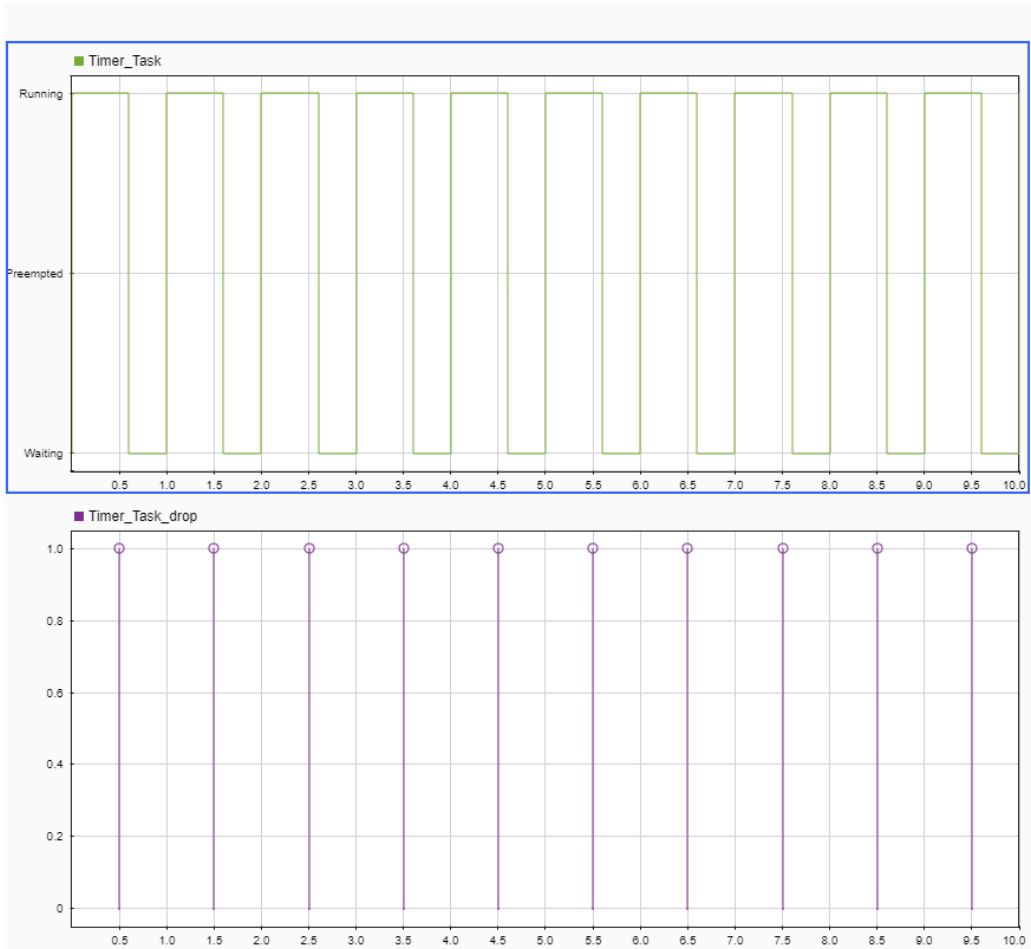
See Also
Task Manager

More About

Task Execution Playback using Recorded Data

The Task Manager block can replay the execution timing of a task recorded from either a previous simulation of that task or from the execution of a task on a processor in an SoC device. To replay a task timing data file, use the following procedure:

1   In a Simulink model, open the Task Duration block dialog box.
2   Select a task from the list of available tasks.
3   In the **Simulation** tab, select **Play back recorded task diagnostics file**.
4   Click **Browse** to select a `taskname.csv` file from a previous task simulation.

While using the data file for the task timing information, the Task Manager still manages individual tasks according the scheduling of the system and can be preempted by other higher priority tasks in the model. For more information on task priority and preemption, see "Task Priority and Preemption" on page 2-10.

See Also
Task Manager

Related Examples
•   "Task Execution" on page 5-74

More About
•   "Task Duration" on page 1-14

Task Priority and Preemption

Task priority informs the operating system of the importance of the task and the order in which a group of waiting tasks needs to execute. By setting the priorities of the tasks in the Task Manager block, tasks that need to react to critical or time-sensitive events can preempt lower priority and background tasks.

Tasks listed in the Task Manager block execute in a *rate monotonic* order. Rate-monotonic order requires the task with the highest static priority in the preempted state to immediately preempt all other tasks and enter the running state. Timer-driven tasks with shorter periods get higher static priorities. If two tasks with equal priority in the preempted state, when no other running task exists, then tasks execute in a first-in, first-out (FIFO) order.

Each event-driven task listed in the Task Manager block can be set with an explicit execution priority. Timer-driven tasks inherit their priority from the base rate task priority of the model. In the configuration parameters, the base rate task priority is set by the **Hardware Implementation** > **Hardware board settings** > **Operating system/ scheduler** > **Base rate task priority** parameter. The following example shows the interaction between a pair of competing tasks.

Preemption of Low Priority Task by High Priority Task

This example shows how the task manager changes the state of two tasks, preempting the lower priority task to allow the high priority task to run.

Task Manager with High and Low Priority Tasks

The following model simulates a software application with a high and low priority task. A Task Manager block schedules the execution of the task subsystems inside the Software Application Model Reference block.

ARM Processor
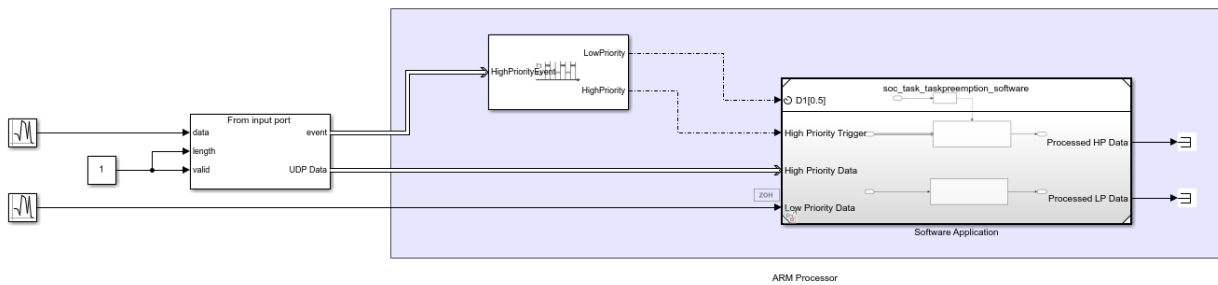
The low priority, timer driven, task is scheduled to run every 0.5 seconds with a duration of 0.2 seconds. The high priority, event driven, task is scheduled to run when a new UDP data packet arrives, which occurs every 1.1 seconds and requires a task duration of 0.5 seconds. As a result of these timing conditions, the low priority task gets preempted to allow the high priority task to run.

Simulation Showing Task Preemption

Click the Run button to build and run the model. When the model finishes running, open the SDI display to see the results of the simulation. Select the HighPriority and LowPriority task waveforms to see the task preemption.
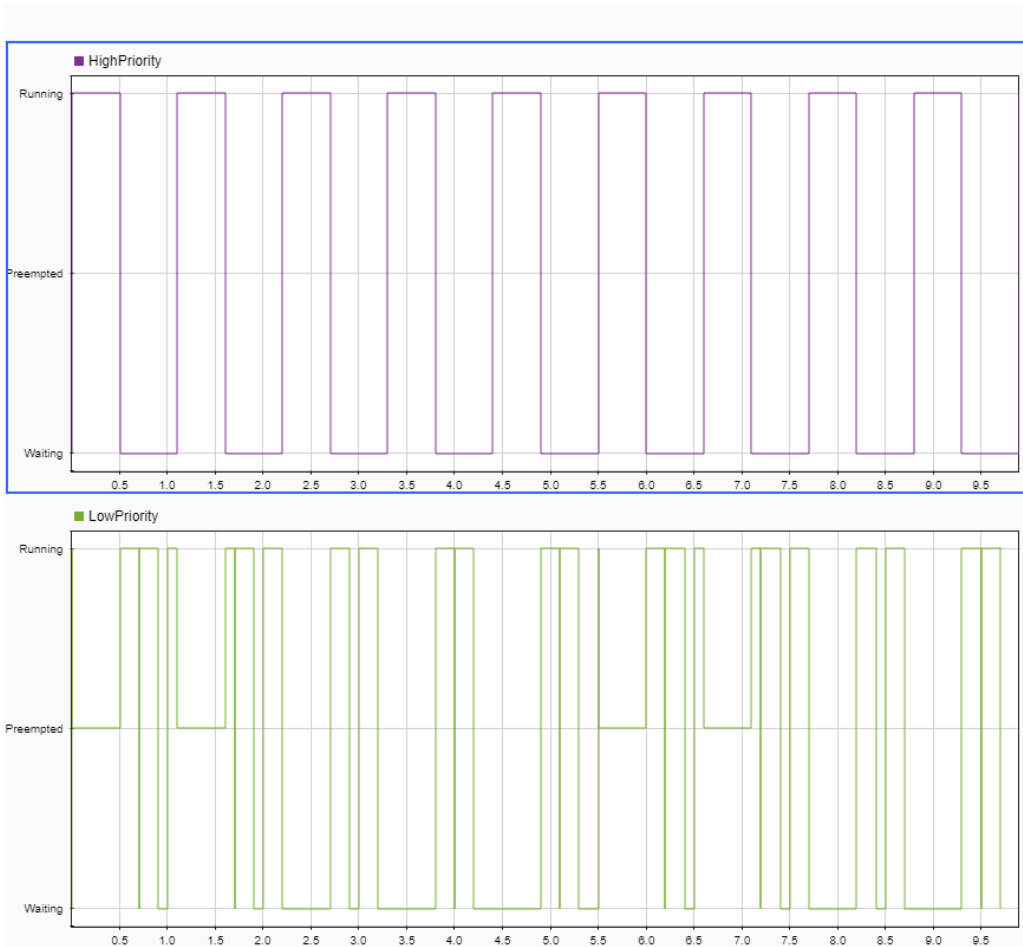
Inspecting the SDI display at time 1.0, the low priority task starts executing until time 1.1, getting preempted by high priority task. The low priority task then runs to completion

at 1.7 seconds, overrunning the next instance of the low priority task that should have started at 1.5 seconds.

See Also

Task Manager

More About

- "What is Task Execution?" on page 1-2
- "Task Overruns and Countermeasures" on page 2-2

Multicore Execution and Core Visualization

SoC Blockset enables simulation of task executions as they behave on a multicore processor. In multicore simulations, tasks can run simultaneously when assigned to different processor cores. Additionally, assigning lower-priority tasks to unique cores prevents these tasks from getting preempted, giving greater confidence to the final application.

Specify the Core for a Task

To set the processor core on which a task executes, open the Task Manager block dialog mask. Select a **Task** from the available tasks. In the task properties, set **Core** to a nonnegative integer value. During simulation, task instances execute on the specified core, subject to the preemption by other tasks executing on the same core. For more information on task preemption, see "Task Priority and Preemption" on page 2-10.

Core Visualization in Simulation Data Inspector

SoC Blockset provides a view of the processor cores on the Simulation Data Inspector. This diagram shows the visualization of the core activity relative to the task state.
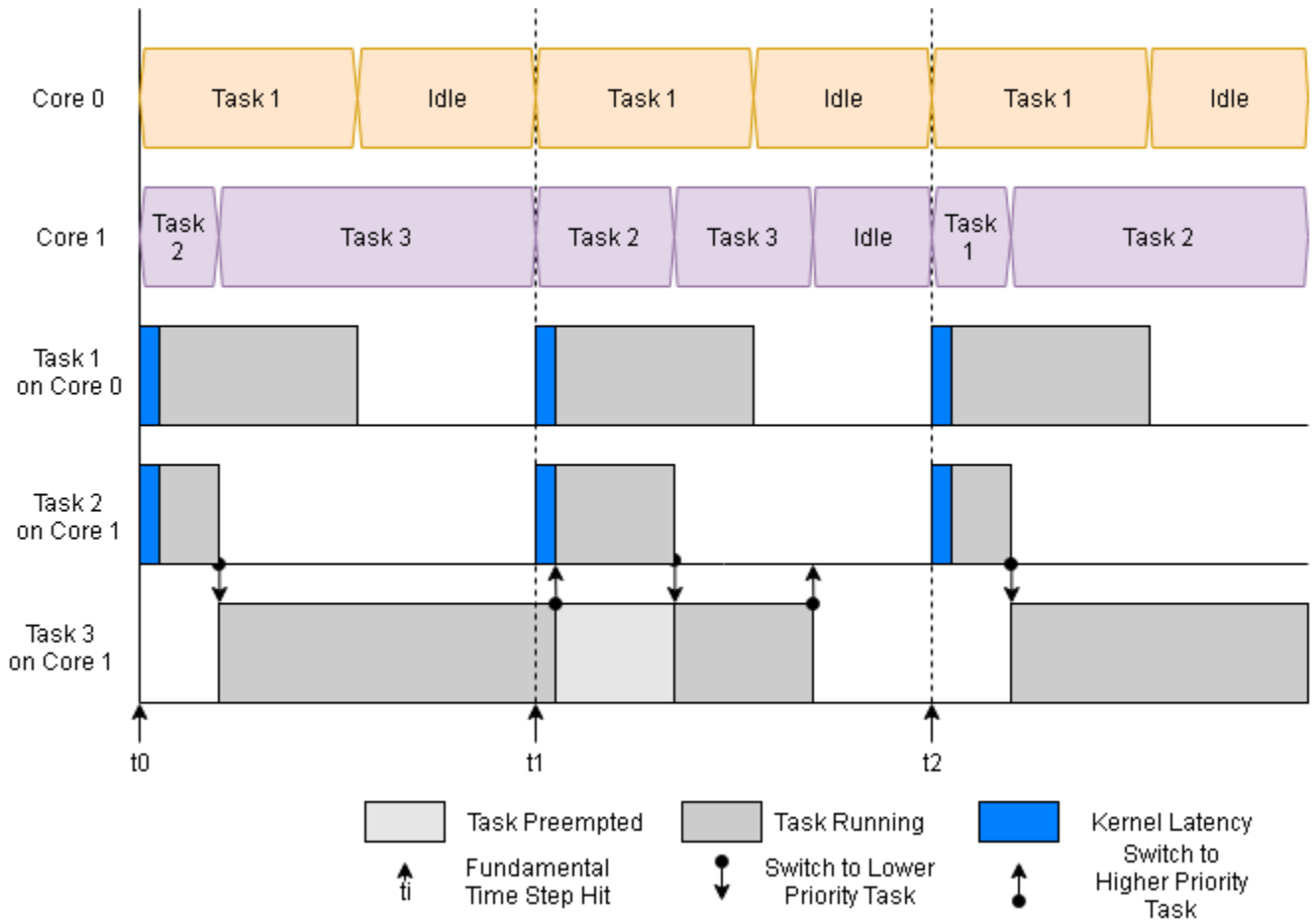
In the Simulation Data Inspector, the signal `corei` shows the current task executing on that core. When the core activity displays as idle, then that core has all tasks in the waiting state, and the kernel can use that core for background tasks that are not part of the main application.

Note If a task instance does not run to completion during the simulation time, the related core status over that instance appears empty in the SDI display.

Multi-Core Task Execution

This example shows the simulation of multiple tasks, managed by the Task Manager block, execute on multiple cores with display the core activity shown in the Simulation Data Inspector.

This model simulates a software application, running on an ARM processor, with 3 timer-driven tasks. A Task Manager block schedules the execution of the tasks, inside the `Software Application Model Reference` block. Tasks 1, with a period of `0.01` seconds, executes on `Core 0`. Tasks 2 and 3, with periods of `0.02` and `0.03` seconds, respectively, execute on `Core 1`.
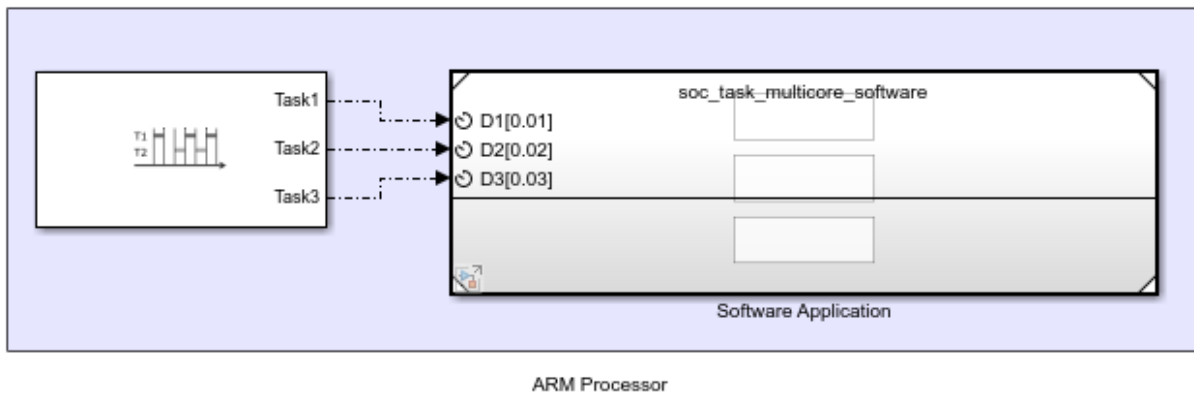
ARM Processor

Click the Run button to build and run the model. When the model finishes running, open the SDI display to see the results of the simulation. Select the **Core 0** and **Core 1** to view the core execution status.
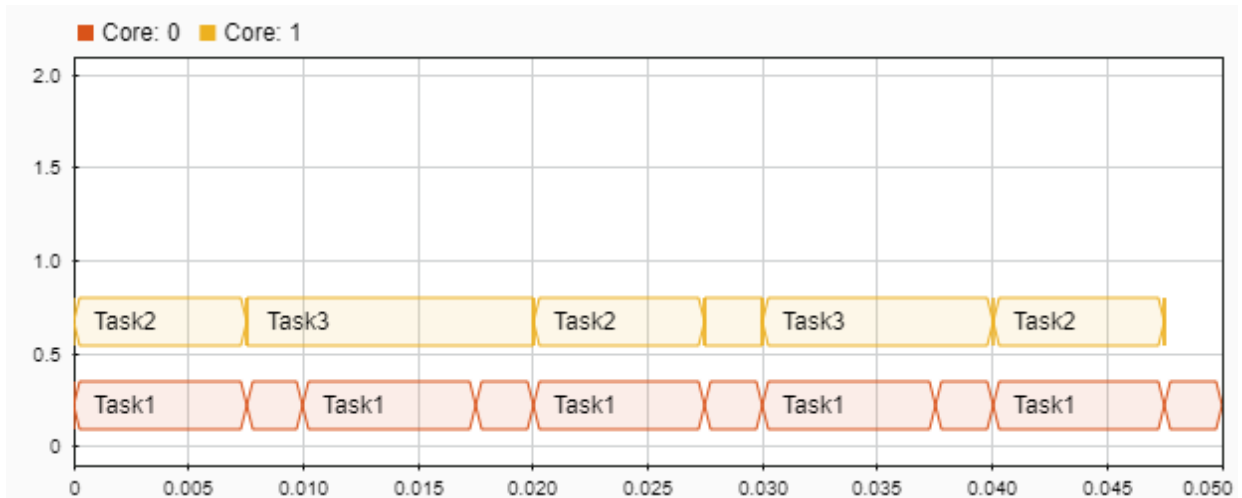
As shown in the SDI display, the core executes either the running task or moves to an idle state, to perform background kernel tasks. Additionally, as two cores are used in this application, high-priority, Task1 executes at the start of each trigger event. Similarly, Task2 and Task3 do not get preempted by Task1. As a result, the application makes better use of the available processor resources.

See Also

Simulation Data Inspector | Task Manager

More About

• "Task Priority and Preemption" on page 2-10

Recording Tasks for Use in Simulation

Each time a model containing a Task Manager block runs in simulation or on an embedded processor with external mode, Simulink records task execution data and statistics as a set of files. A diagnostics folder, with name *modelname*_diagnostics, contains two subfolders, sim and hw, for the data from simulations and recorded from hardware, respectively. Each run generates a unique folder, inside either the sim or hw folders, labelled by the date and time of the run. The folder name uses a time-date format, *YYYY_MM_DD_hh_mm_ss*, representing the year, month, day, hour, minute, and second, respectively.

Each run generates a set of metadata, statistics, and execution recording files, including:

- TaskInfo.mat – This file contains task information, including the task names and types, used internally by the SoC Blockset.

- metadata.csv – This file contains the derived mean and standard deviation for all tasks recorded in the profile.log data file. The metadata.csv file can be used directly in the Task Manager block to set task duration statistics. For more information on setting task duration, see "Task Duration" on page 1-14.

- *TaskName*.csv – This file contains the recorded task execution data as a comma-separated variable list. The first column contains the start time of each task instance. The second column contains the task durations for each task instance. If a task is dropped, lost, or corrupted, the start time and duration of that task execution instance are both replaced by -1. For more information on using recorded task execution timing in simulation, see "Task Execution Playback using Recorded Data" on page 2-9.

Note

- Tasks recorded from an embedded processor only start capturing task execution after successful connection of external mode. The lost start-up in task execution recordings from hardware should be considered when comparing timing results to recordings from simulation.

- When executing on an embedded processor, task execution recordings times will continue to run until the completion of all task instances scheduled in the Task Manager prior to the stop time of the model.

See Also

Task Manager

More About

- "Task Duration" on page 1-14
- "Task Execution Playback using Recorded Data" on page 2-9
- "Profile Task Execution on Processor" on page 4-4

Task Visualization in Simulation Data Inspector

The Simulation Data Inspector (SDI) display provides a direct view into the execution timing, the task state, and the execution of tasks in simulation and profiled from generated code running on hardware. Each model run, in simulation or on hardware using external mode, creates a separate **Run** for the task execution timing and data. This image shows the SDI display with a **Run** captured from an SoC Blockset model.

Each **Run** contains these signal types:

- *taskname* – The execution instance state for the task, with name *taskname*, defined in the Task Manager block. For more information on task execution states, see "What is Task Execution?" on page 1-2.

  Note If a task instance does not run to completion during the simulation time, the final task execution instance does not render in the SDI display.

- *taskname*_drop – An impulse indicating the scheduler dropped an execution instance of task, *taskname*_drop. For more information on task drops, see "Task Overruns and Countermeasures" on page 2-2.

- `Core: ` *n* – Execution activity on core *n* of the simulated processor. For more information on multicore execution and visualization, see "Multicore Execution and Core Visualization" on page 2-14.

Note If a task instance does not run to completion during the simulation time, the related core status over that instance does not render in the SDI display.

See Also

Simulation Data Inspector | Task Manager

More About

- "What is Task Execution?" on page 1-2
- "Task Overruns and Countermeasures" on page 2-2
- "Multicore Execution and Core Visualization" on page 2-14

Simulation Performance Plots

SoC Blockset enables post-simulation analysis of memory diagnostic data. These plots provide high-level performance diagnostics of the memory system of the model. These plots are calculated measurements from a simulation of your model. It considers the data type, sample time, and clock frequency to calculate the bandwidth of your memory model and considers the number of bursts executed per memory port.

To enable signal logging in simulation, select **Hardware Implementation** on the Configuration Parameters dialog box. Under **Hardware Board Settings > Target Hardware Resources > FPGA design (debug)**, select the desired **Memory channel diagnostic level**.

This figure shows the datapath from one FPGA algorithm to another FPGA algorithm through a memory channel.
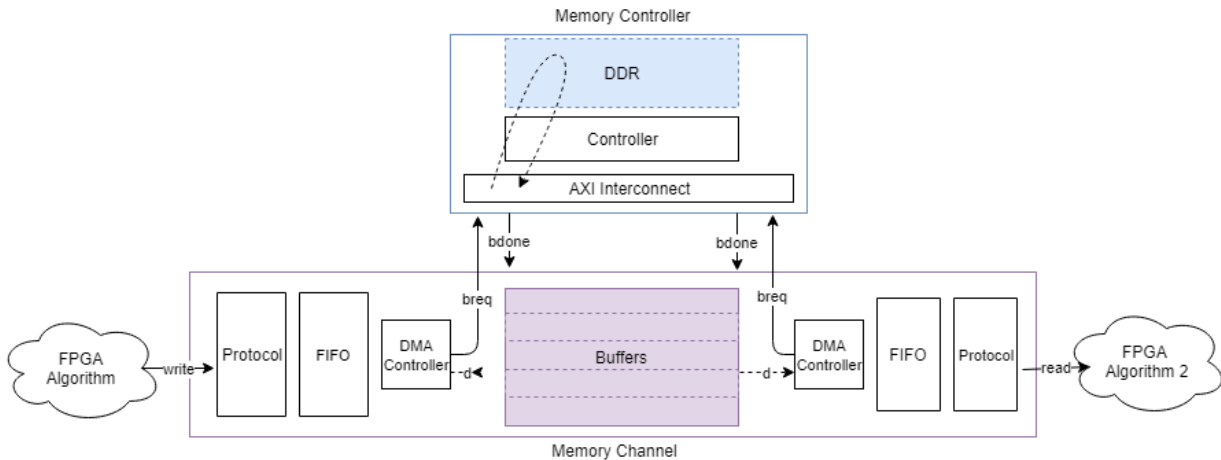
You can view channel latency plots for the datapath (represented by A, B, C, and D in the image) from the Memory Channel block mask. You can view memory bandwidth, burst count, and control-latency measurements (represented by 1, 2, 3, and 4 in the image) from the Memory Controller block mask.

The datapath from an FPGA algorithm to a processor is served through a DMA driver and a task processor and is illustrated in this image.
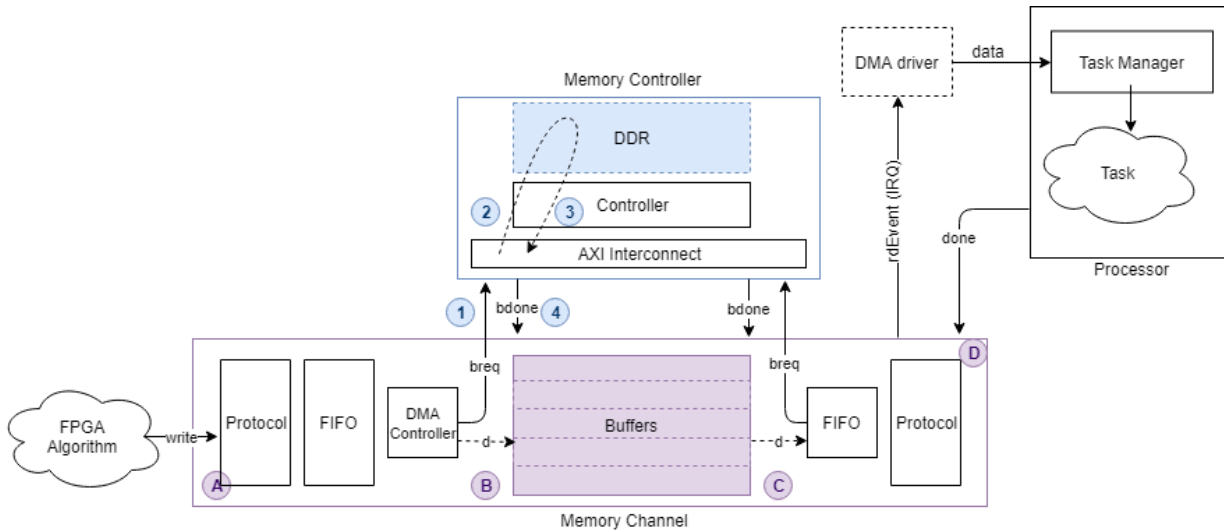
Memory Channel Latency Plots

Memory Channel latency information is available post simulation per channel. After simulating your model, open the Memory Channel block mask. On the **Performance** tab, click **Launch performance plots**. This action opens a new window with several control options to display these different latencies:

- **Buffer write complete** – This option shows the time it takes between issuing a write request to when the buffer is fully written. It is the path between A and B in the figure.
- **Buffer read complete** – This option shows the time it takes between issuing a read request to when the buffer is read and is available again for writing. It is the path between C and D in the figure. This option is only available if the reader is an FPGA algorithm (not a processor algorithm). If the reader is a processor algorithm, this time shows as zero.
- **Buffer task execution complete** – This option shows the time it takes between issuing a read request to when the buffer is read and is available again for writing. It is the path between C and D in the figure. This option is only available if the reader is a processor algorithm (not an FPGA algorithm). If the reader is an FPGA algorithm, this time shows as zero.

  The **Buffer task execution complete** shows the time it takes for these events to occur:

2-23

1   The write buffer is full.

2   The channel issued an interrupt request (IRQ) to the processor.

3   An interrupt service routine (ISR) is executed.

4   A task is scheduled.

5   The task started executing.

6   The task read data.

7   The task optionally processed the data.

8   The task sends a `done` signal back to the channel.

This following figure shows the latency path for a task execution to complete, as a red arrow from C to D.
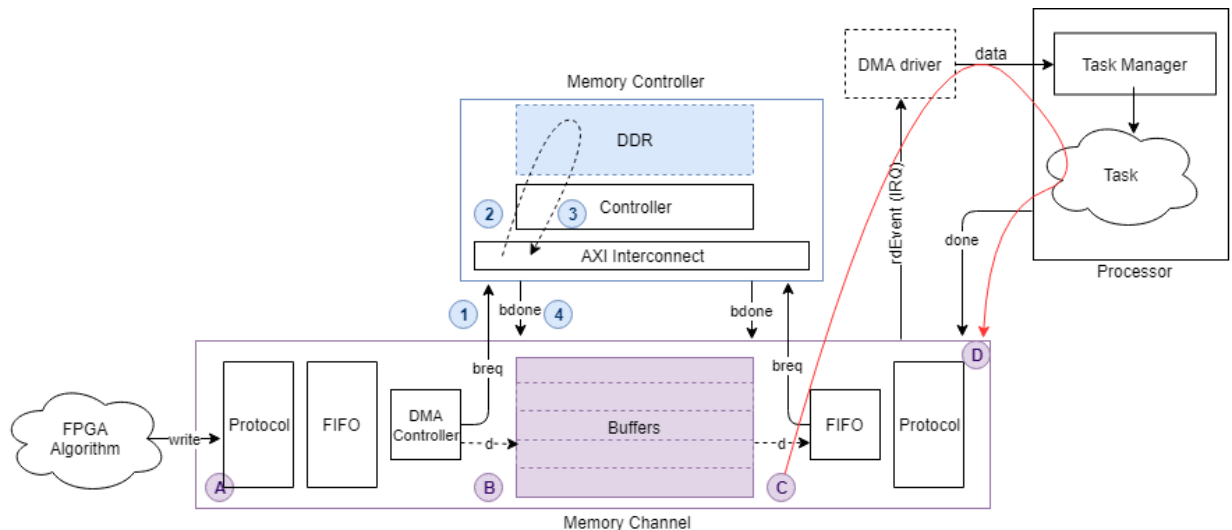
- **Averaging Window (s)** – Specify a time, in seconds, for the averaging window width. The plot is graphed as a moving average, using a time window with the width specified. You can also specify `min`, `max`, or `auto`.

  - `min` – Use this value to see data without any averaging. The total latency graph is aligned with the **Instantaneous Total Latency** marks.

  - `max` – Use this value to see the overall average for the entire simulation.

  - `auto` – Use this value to see averaging over the number of buffers in your channel.

- **Instantaneous Total Latency** – This shows discrete total latency measurements per buffer.

If you add **Buffer write complete** to **Buffer read complete** or **Buffer task execution complete**, the plot displays the full latency from writer to reader. This image shows the total latency plot for the "Streaming Data from Hardware to Software" on page 5-37 example.
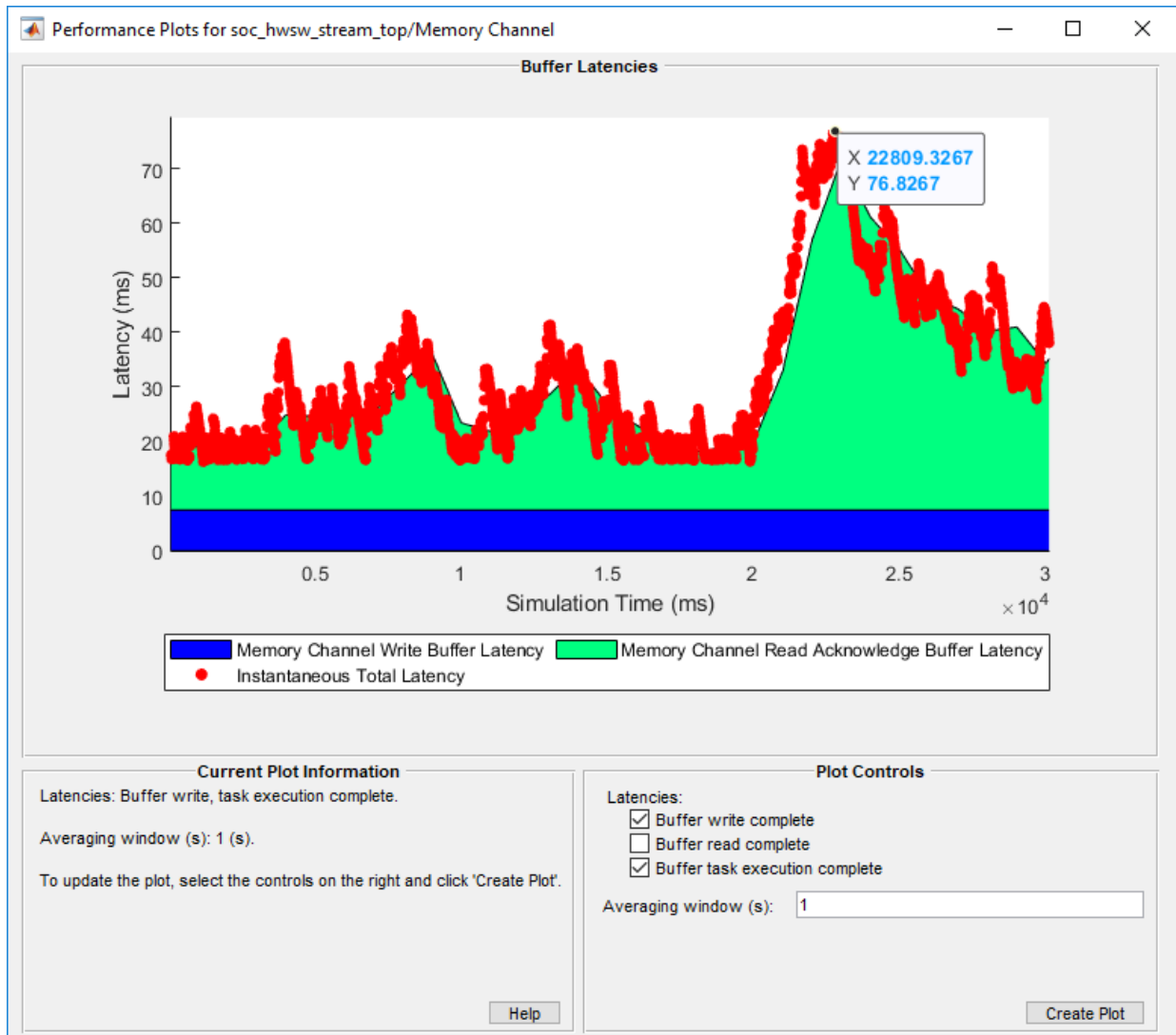
Note that the latencies are showing over an averaging window of one second. The instantaneous total latency shows a peak in latency as 76.8267 ms. Use this information to verify the model against the requirements.

Memory Controller Latency Plots

Memory Controller latency information is available post simulation. After simulating your model, open the Memory Controller block mask. On the **Performance** tab, click **Launch performance plots**. This action opens a new window with several control options to display performance metrics.

This figure shows the datapath from one FPGA algorithm to another FPGA algorithm through a memory channel.
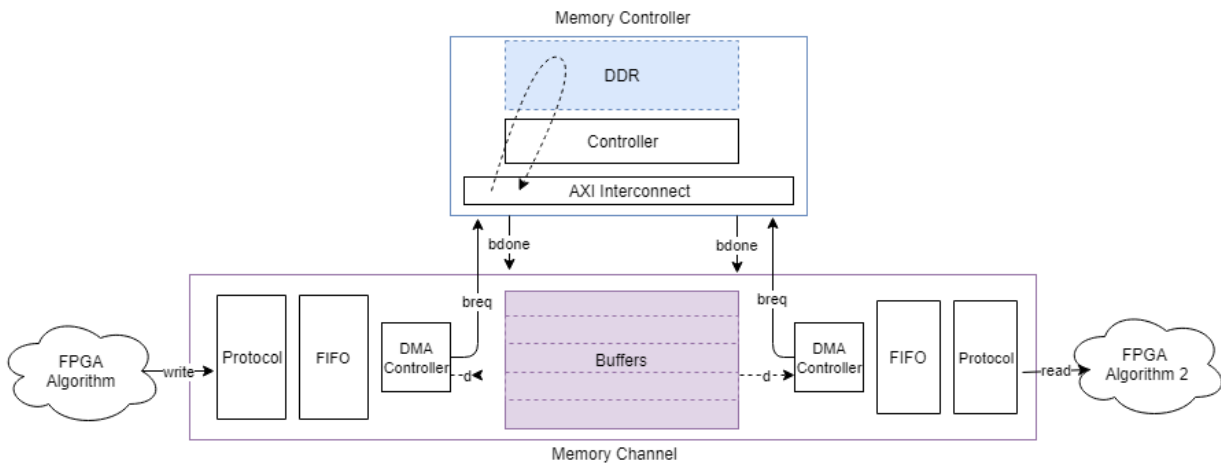
In the **Latencies** tab, select the master for which you want to graph latencies. Choose from any of these options:

- **Burst request to first transfer complete** – This option shows the time it takes from the moment the Memory Channel block issues a burst-write request to the first transfer of data. This latency accounts for arbitration or interconnect delays. It is the path between 1 and 2 in the figure.

- **Burst execution latency** – This option shows the time it takes from the first transfer of data to when a burst is written to memory. It is the path between 2 and 3 in the figure.

- **Burst last transfer to complete latency** – This option shows the time it takes from the moment a burst completes to when the Memory Controller block issues a `birst-done` signal to the Memory Channel block. It is the path between 3 and 4 in the figure.

- **Averaging Window (s)** – Specify a time, in seconds, for the averaging window width. The plot is graphed as a moving average, using a time window with the width specified. You can also specify `min`, `max`, or `auto`.

  - `min` – Use this value to see data without any averaging. The total latency graph is aligned with the **Instantaneous Total Latency** marks.
  - `max` – Use this value to see the overall average for the entire simulation.
  - `auto` – Use this value to see averaging over 1% of the bursts during the simulation.

- **Instantaneous Total Latency** – This option shows discrete total latency measurements per burst.

Click **Create Plot** to see the latency, for the selected masters over the duration of the simulation time. This image shows the total latency for `Master 2` in the "Analyze Memory Bandwidth Using Traffic Generators" on page 5-51 example.

Performance Plots for soc_memory_traffic_generator/Memory Controller

Latencies

Current Plot Information

Master to plot: Master 2.

Latencies: Burst request to first transfer complete, execution, last transfer to complete.

Averaging window (s): auto (2.431e-05 s).

To update the plot, select the controls on the right and click 'Create Plot'.

Latencies Plot Controls

Bandwidth \ Bursts \ Latencies \

Master to plot:
Master 2

Latencies:
☑ Burst request to first transfer complete
☑ Burst execution
☑ Burst last transfer to complete

Averaging window (s): auto

Note Memory controller latency plots are not available when the master is a processor.

You can then zoom in to analyze the peak instantaneous latency:
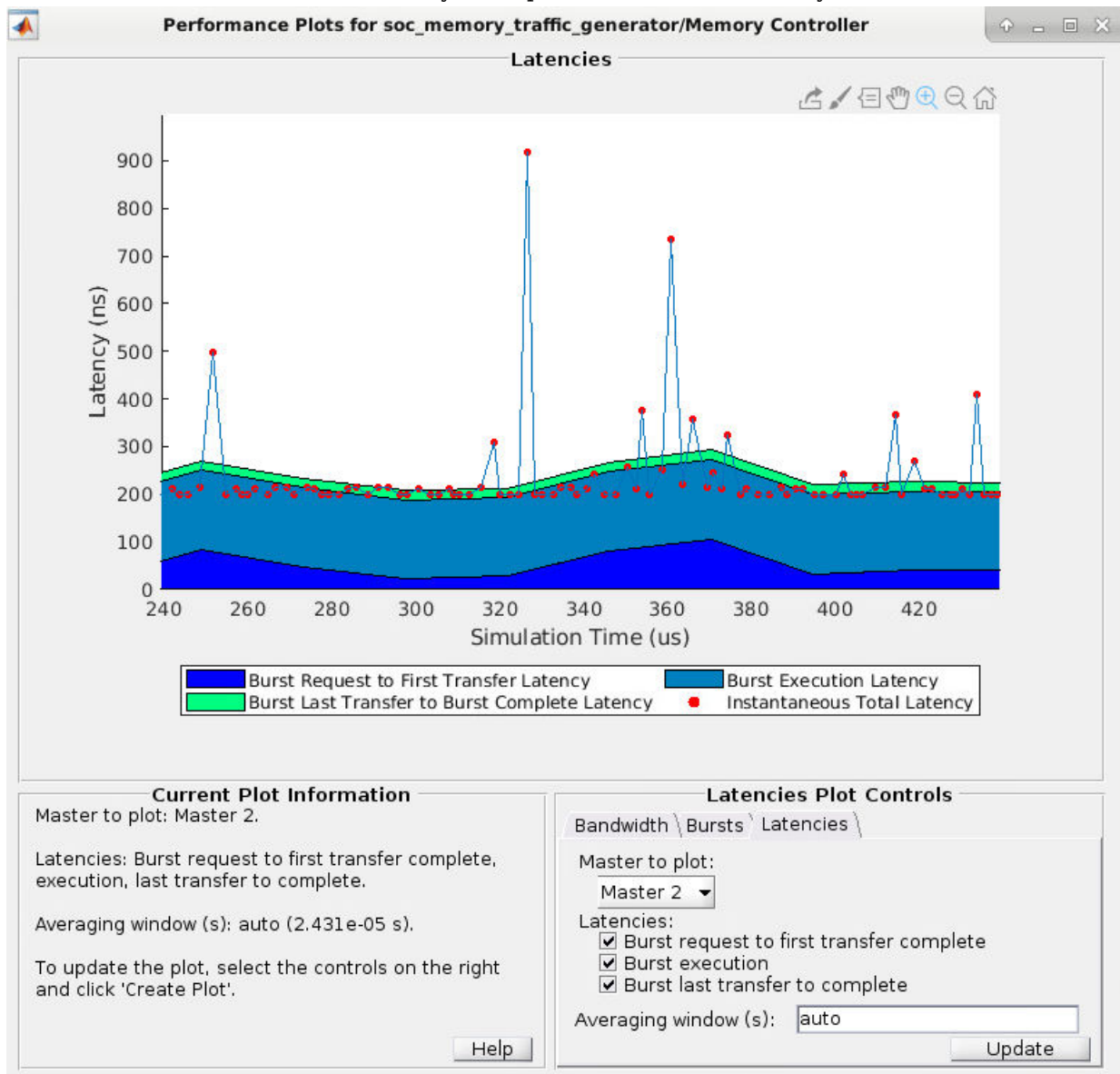
Memory Bandwidth Plots

In the **Bandwidth** tab, select the masters for which you want to graph bandwidth. Click **Create Plot** to see the bandwidth, in megabytes per second, for the selected masters over the duration of the simulation time. This image shows the bandwidth for the "Analyze Memory Bandwidth Using Traffic Generators" on page 5-51 example.

Memory Burst Plots

In the **Bursts** tab, select the masters for which you want to graph bursts. Click **Create Plot** to see the number of bursts executed for the selected master over the duration of the simulation time. This image shows the burst count for the "Analyze Memory Bandwidth Using Traffic Generators" on page 5-51 example.
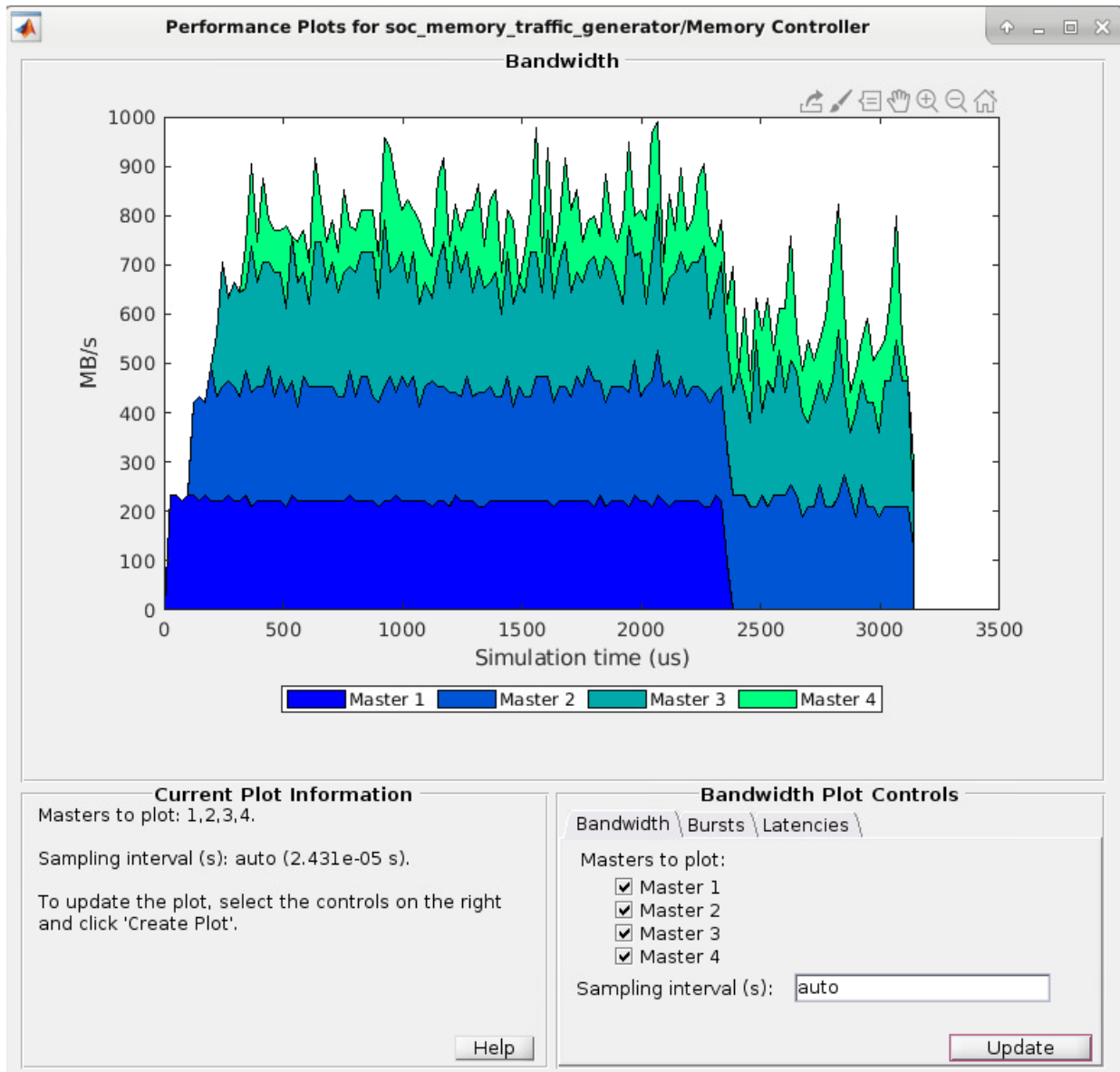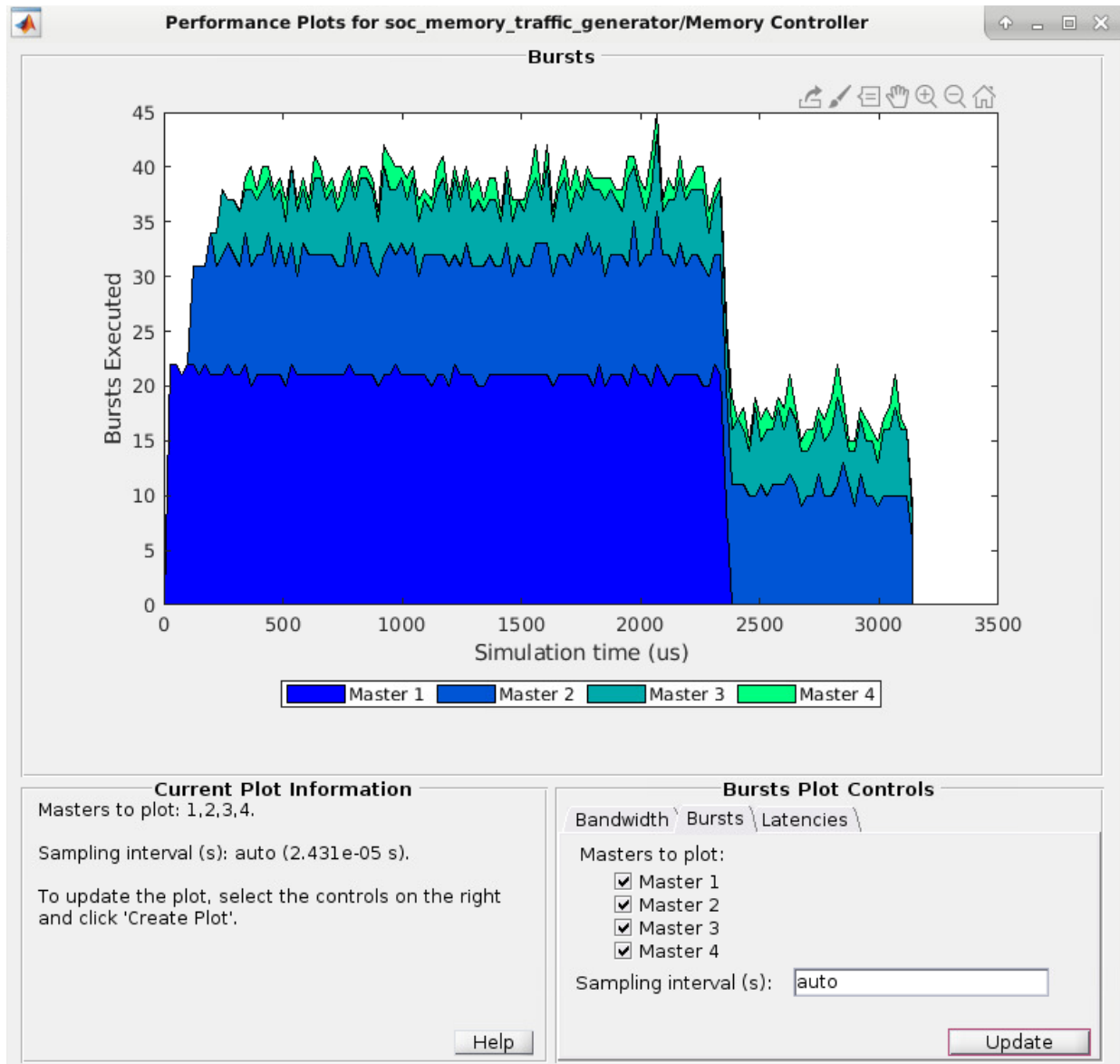
Note Bandwidth information is not displayed when a master is a processor.

See Also

Memory Controller | Memory Channel

More About

- "Memory Performance Information from FPGA Execution" on page 4-6

Simulation Diagnostics

SoC Blockset enables simulation and evaluation of memory transactions in Simulink without the need to deploy a model to an SoC device. Use this diagnostic information to analyze the performance of your models, and adjust as needed to meet the desired system performance requirements. The simulation generates two types of visualization of the memory traffic:

- "Simulation Performance Plots" on page 2-22 – Provides high level performance diagnostics of the model's memory system. Memory bandwidth, burst counts, and transaction latencies are calculated from a simulation of your model. You can view this information for each memory master in your model, or an overall view from the memory controller.
- "Buffer and Burst Waveforms" on page 2-36 – Provides burst transaction debug information from simulation, including the use of buffer regions.

You can also capture actual bandwidth, number of bursts, and latency measurements from the design running on the FPGA, and view information about individual burst transactions. This information is captured by including an AXI interconnect monitor IP in the FPGA design, and querying the data over a JTAG AXI master connection from the host. See "Memory Performance Information from FPGA Execution" on page 4-6.

Buffer and Burst Waveforms

SoC Blockset enables logging simulation signals, and visualizing the logged signals using the *Logic Analyzer* tool. To enable signal logging, Set **Memory diagnostics level** to `Basic diagnostic signals` in the configuration parameters of the model, under **Hardware Implementation** > **Target hardware resources** > **FPGA design (debug)**.

After simulating your model, locate the *Logic Analyzer* at the top of your Simulink window.
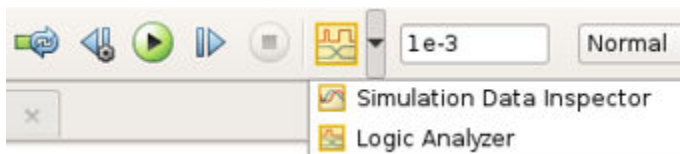
The **Logic Analyzer** tool provides visualization of signal waveforms to show timing of various events of the memory model. For more information about the tool, see **Logic Analyzer**.

The **Logic Analyzer** displays signals from the Memory Controller and from the Memory Channel blocks.

- **Burst Waveforms**

  Waveforms from the memory controller include information for bursts from the masters in the system. The waveforms are color coded to differentiate the different masters. These waveforms give insight into the sequencing of each of the masters through the shared memory. For each master, view the following signals:

  - BURST_EXECUTION_EVENT: State of the current burst request. Valid states are: none (idle), request, executing, done. For more information about the memory controller state, see Memory Controller.
  - ReqID: Identifier of the current burst request. An incrementing number that is unique throughout simulation.
  - burstTransfersCompleted: A running count of transferred bursts. If no bursts are dropped within the memory channel, the count of transferred bursts matches ReqID. If bursts are dropped, ReqID becomes larger than this count.
  - BytesTransferred: A running count of transferred bytes.

  The following figure shows the signals after simulating "Analyze Memory Bandwidth Using Traffic Generators" on page 5-51.
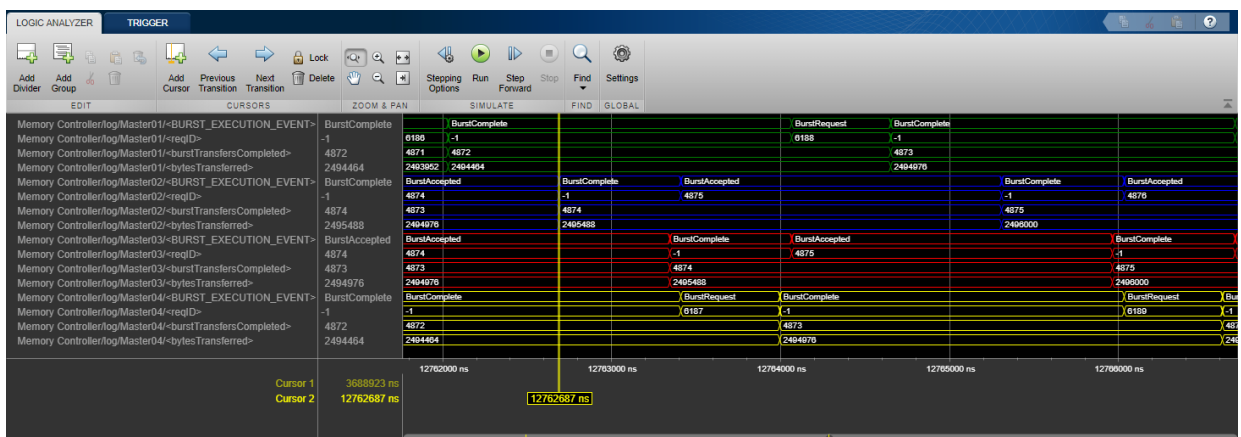
The waveforms include burst information for the four masters, displayed in different colors. This information correlates to the "Memory Controller Latency Plots" on page 2-27.

- **Buffer Waveforms**

  Waveforms from the memory channel include information for buffer read and write transactions in the channel. Each memory region is divided into several buffers specified by the **Number of buffers** parameter of the Memory Channel block. The writer fills the buffers, and the reader empties them. These waveforms give insight into the sequencing of the writer and reader for a given region. The buffer waveforms include the following signals:

  - `REGION_BUFFER_EVENT`: State of the current buffer request. Valid states are: none (idle), request, executing, done. For more information about the state of the memory channel, see Memory Channel.

  - `BufReqID`: Identifier of the current buffer request. An incrementing number that is unique throughout simulation.

  - `BufferAddress`: Starting address offset of the current buffer. The buffer address repeats as the simulation cycles through the buffers, reflecting the address boundaries of the buffers.

  - `BufGntCurrentBuf`: The currently active buffer specified from 1 to the number of buffers in the channel. `BufGntCurrentBuf` points to the buffer being written to (on the writer side), or the buffer being read from (on the reader side).

  - `BufRelCurrentBuf`: The buffer currently released by the reader or writer specified from 1 to the number of buffers in the channel. On the reader side, when a buffer is released it is available to the writer for writing. On the writer side, when a buffer is released it is available to the reader for reading.

  - `BufAvail`: The number of buffers currently available to the reader for reading. This value is identical on the reader and the writer side.

  - `BufTransfersCompleted`: A running count of transferred buffers. If no buffers are dropped within the memory region, the count of transferred buffers matches `BufReqID`. If buffers are dropped, `BufReqID` is larger than this count.

  - `icFIFOEntries`: Number of bursts written to the interconnect FIFO.

  - `icFIFODroppedCount`: Number of bursts dropped from the interconnect FIFO.

- The following figure shows the buffer signals after simulating "Histogram Equalization Using Video Frame Buffer" on page 5-24.
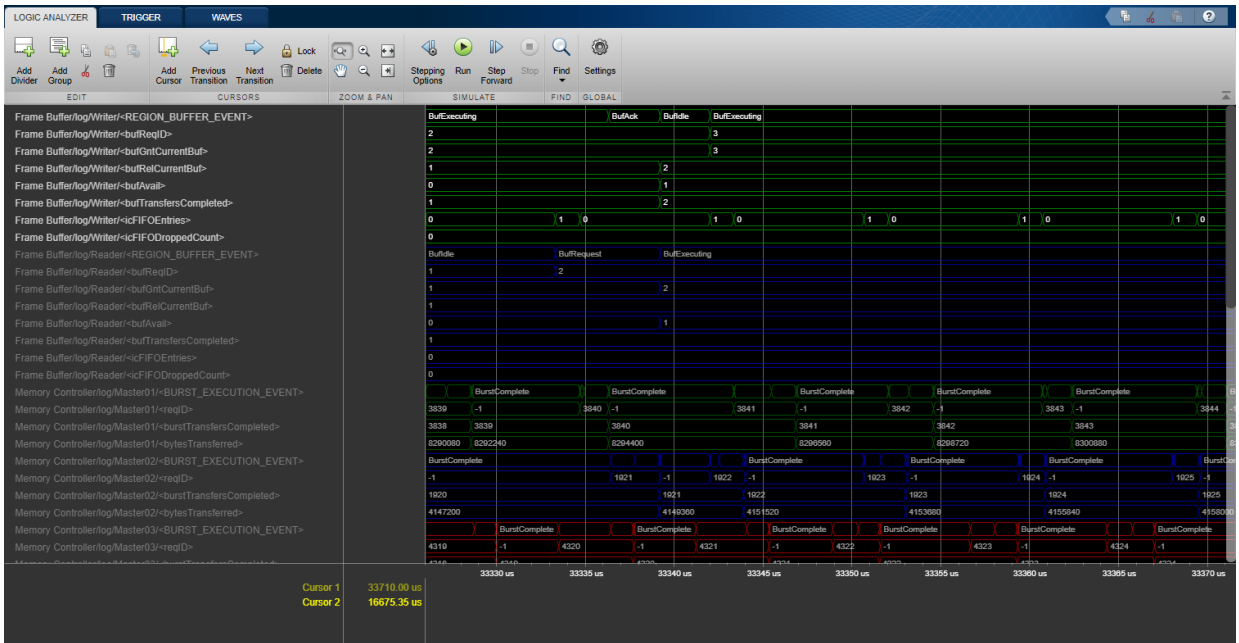
You can relate the memory model operation with the protocol interface to understand the performance of your model. The following figure shows how to relate the memory model operation with the protocol interface.

1. Backpressure from rdReady
2. Reader cannot finish buffer 2.
3. Writer's FIFO begins filling up.
4. Writer is blocked and must assert back-pressure upstream.
5. Reader gets to finish buffer. Everyone starts moving again.
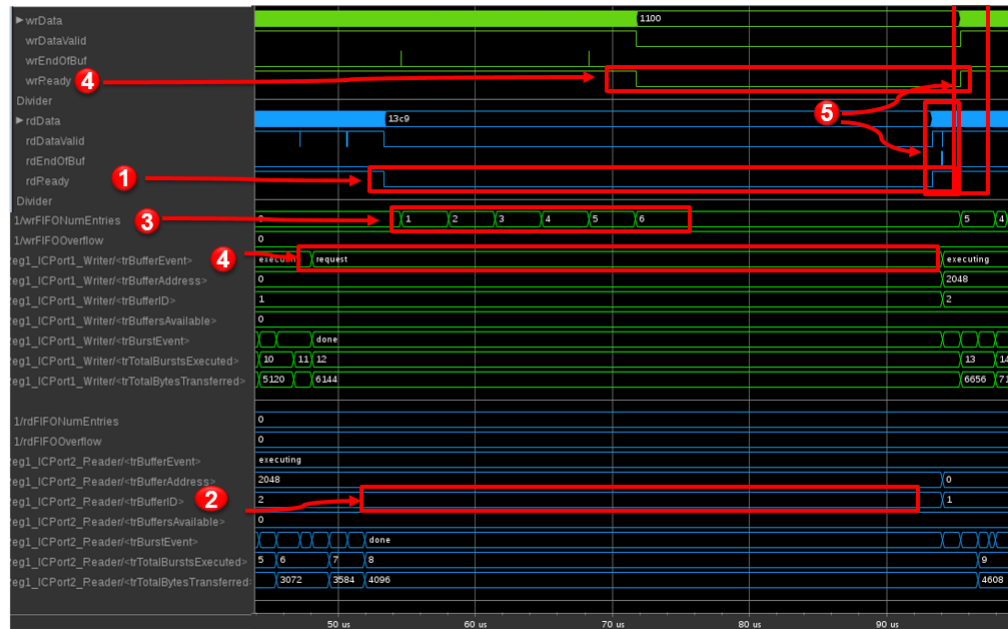
See Also

Logic Analyzer | Memory Controller | Memory Channel

More About

*   "Simulation Performance Plots" on page 2-22

External Memory Channel Protocols

The signal interfaces added to the channel model for the writer and reader are protocols that the algorithms use to communicate with the channel. Protocols do not change the core of the external memory channel model, which operates on *burst transactions*. They control only how the data gets in or out of those channels.

For FPGA or ASIC IPs, typical protocols include streaming data, streaming video data, and addressable data transfers. For software, typical protocols presented to an algorithm include simple data buffer, with details about interrupts, buffer management, and task scheduling left to the underlying OS.

Configure the Memory Channel block to support various protocols.

AXI4 Stream to Software via DMA

The AXI4-Stream Software configuration provides a software streaming protocol. Choose this configuration when a processor acts as a reader/writer to the memory. This protocol includes a trigger configuration, which the Task Manager block receives and reads. The trigger signals that the memory buffer is ready for writing or reading. For more information about AXI4-stream protocol, see "AXI4-Stream Interface" on page 1-27.

AXI4 Stream FIFO

The AXI4-Stream configuration provides a simple data valid and ready protocol for data streaming. You can generate a fully compliant AXI4-Stream interface from this protocol using HDL Coder.

For data stream channels, memory addressing is automatic. The channel is responsible for converting the stream to buffer addresses as a DMA core would. The relationship of the stream to the managed buffers in the external memory is through an 'end of buffer' signal, known as `tlast` for AXI4-Stream. For more information about AXI4-stream protocol, see "AXI4-Stream Interface" on page 1-27.

AXI4 Stream Video FIFO

The AXI4-Stream Video FIFO configuration provides a data valid and ready protocol similar to the AXI4 Stream FIFO. This protocol also has additional signaling to mark the start or the end of a video line and start or end of a video frame. This protocol is

compatible with the HDMI Rx and HDMI Tx blocks, available with the SoC Blockset Support Package for Xilinx Devices. You can generate a fully compliant AXI-Stream video streaming interface from this protocol using HDL Coder. For information about the HDMI blocks, see documentation for SoC Blockset support packages.

For streaming video data channels, memory addressing is automatic. The channel is responsible for converting the stream to buffer addresses as a DMA core would. The stream relates to the managed buffers in the external memory through the pixel control bus signals, which demarcate lines and frames. For more information, see "AXI4-Stream Video Interface" on page 1-30.

AXI4 Stream Video Frame Buffer

The AXI4-Stream Video Frame Buffer configuration provides The same signaling as the AXI4 Stream Video FIFO, with additional control signals for frame-buffer synchronization. This protocol is compatible with the HDMI Rx and HDMI Tx blocks, available with the SoC Blockset Support Package for Xilinx Devices. You can generate a fully compliant AXI-Stream video streaming interface from this protocol using HDL Coder. For information about the HDMI blocks, see documentation for SoC Blockset support packages.

For streaming video data channels, memory addressing is automatic. The channel is responsible for converting the stream to buffer addresses as a DMA core would. The stream's relationship to the managed buffers in the external memory is through the pixel control bus signals, which demarcate lines and frames.

AXI4 Random Access

The AXI4 configuration provides a simple, direct interface to the memory interconnect. Unlike the previous two streaming protocols, this protocol allows the algorithm to act as a memory master by providing the addresses and managing the burst transfer directly. This protocol represents a simplified master protocol. You can generate a fully compliant AXI-4 interface from this protocol using HDL Coder. For more information about the simplified AXI4 interface, see "Simplified AXI4 Master Interface" on page 1-23.

Channel Data and Memory Bursts

Regardless of the protocols being used, the memory model does not change. The channel data is always transferred to the memory model using burst transactions. For the AXI4 configuration, the algorithmic logic is responsible for defining the burst through the protocol signals.

For the streaming data configurations, the **Burst Length** parameter determines the burst size to the memory, and the channel **data** signal defines the size of each transfer on the interface.

There is a relationship between the channel data interface (its dimensions and data type) and the burst. The primary function of the protocol interface blocks is to chunk the data into byte buffers that are one burst in length. These bursts are forwarded to the memory model and it competes for the shared bandwidth. Each of the protocol blocks must specify the length of the burst so that the model can do the chunking. This length is expressed in terms of the scalar datum for the channel.

Guidelines for Setting Burst and Buffer Parameters

The following rules apply to setting up the burst and buffer sizes. A table of both good and bad examples follows.

1   The Burst Length of a given channel interface, calculated in bytes, must be less than 4096 bytes. To calculate the burst size in bytes, the channel interface scalar data type is converted to bytes and then multiplied by the Burst Length.

2   The Burst Length can be set above 256, but will warn if generating to an AXI-based target platform. AXI-based memory systems have a maximum burst beat count of 256.

3   The Channel Length must be an integer multiple of burst length or the burst length must be an integer multiple of channel length. That is, it must be possible to either chunk the incoming channel data to a whole number of bursts or to gather a whole number of incoming channel data to a single burst.

4   The Buffer Size must be a whole number of bursts. This must be true for both the writer's burst size (after conversion of its Burst Length to bytes) and the reader's burst size (after conversion of its Burst Length to bytes).

5   The calculated number of bursts in a buffer must not exceed 5000. This is a temporary restriction based on the event processing internal to the memory model. This can happen with shared memory regions that have large buffer sizes (such as for 1080p video frames) and channel interfaces that specify smaller burst sizes. Generally, with larger frames, bursts sizes near the 4096 byte limit must be used.

6   The scalar data type of the channel interface as converted to a flattened channel data width (i.e. *tdata* in the implementation) cannot exceed 32 bits. This is a temporary restriction based on current limitations of the byte conversions in the memory channel model.

Error messages are generated whenever any of these rules are violated. These messages give helpful information on the various parameters specified on the mask or derived from those parameters, and they describe how to fix the issue.

Set the burst and buffer parameters on the mask of the Memory Channel block.

The following table provides examples of good and bad parameter sets.

Burst and Buffer Parameter Examples

Channel Data Type	Channel Dimensions	Burst Length	Buffer Size	Good / Bad	Why?
uint8	[1 1]	1024	2048	Good	This is a simple 8-bit data transaction.
uint8	[1 3]	1024	2048	Good	This might represent an RGB pixel from a Vision HDL Toolbox block. It is converted to 24-bit packed data and padded with 8 bits to become a 32-bit *tdata* bus to the memory.
fixdt(0,10,0)	[1 3]	1024	1024	Good	This is converted to a 30-bit packed pixel with 2 bits of padding.
fixdt(0,12,0)	[1 3]	1024	1024	Bad	This results in a 36-bit packed pixel which violates the current limit of 32-bit *tdata*.
uint8	[120 160 3]	1024	1024	Bad	The scalar data is 24-bit, padded to a 32-bit *tdata*. The Channel Length is 120*160=19200. The burst length of 1024 does not evenly divide 19200.
uint8	[120 160 3]	120	76800	Good	The scalar data is 24-bit, padded to a 32-bit *tdata*. The Channel Length is 120*160, and since Burst Length is 120, Channel length is 160 bursts in size. The Buffer size is exactly 1 frame (120*160*4) as calculated in bytes.

See Also

Memory Channel

More About

- "Simplified AXI4 Master Interface" on page 1-23
- "AXI4-Stream Interface" on page 1-27
- "AXI4-Stream Video Interface" on page 1-30

Memory Channel Model Fidelity

- Handles any data types and dimensions
  1080p RGB frame or scalars
  Develop memory architecture *before* algorithm implementations

- Interface generates standard AXI protocols

- Model at fidelity of a DDR burst
  Represent thousands to millions of clocks with just a few events

- Event-based model
  Channel does not influence Simulink's fundamental sample time
  Only simulate when meaningful transitions happen

- Dummy masters to waste bandwidth

- High-level diagnostic performance data
  Match against deployed design performance

- Get full memory image dumps
  Debug data storage state over time
  Match against deployed design memory dumps

Record Data from Hardware I/O Devices

Models using recorded data in simulation can reproduce the behavior of the application when implemented onto a physical hardware or device. SoC Blockset provides a set of functions that can connect and record I/O device data directly from a hardware board. The recorded data file can then be used in an SoC Blockset model simulation.

Process to Record Data

To record I/O data from a hardware board, you can follow the general sequence of steps below.

1 *Configure Hardware* – Connect and configure your hardware board. You may need to install the hardware support package for your hardware board.

2 *Create Data Recorder* – A data recorder object manages the I/O hardware peripherals and stores the data during the data collection process.

3 *Choose I/O Devices* – Choose from the available I/O devices on the hardware board and add them to the data recorder object.

4 *Setup Recorder* – Prepare the hardware board for the data recording process. This setup includes any initialization and configuration of the hardware I/O devices to be recorded.

5 *Start Recording* – Start the data recorder on the hardware. The data recorder executes and collects data from the hardware I/O devices for the specified period.

6 *Execute Hardware Operations* – Run hardware operations on the hardware board that exercise the peripherals being recorded. Operations can include sending signals to an analog-to-digital converter or reading data received on a UDP channel.

7 *Save Data* – Save the data stored in the data recorder to a file on your development computer.

The resulting data file can now be used in the simulation of the hardware blocks.

Use Memory and I/O Device Data in Processor Simulation

The **Processor I/O** sub-library in SoC Blockset contains blocks that simulate the data transfer between the processor system and memory or I/O devices in the SoC application. **Processor I/O** blocks, including the Register Read, Register Write, and Stream Read, can read and write data to memory, such as DDR or hardware registers, on the SoC. Similarly, the TCP Read, TCP Write, UDP Read, and UDP Write blocks can read and write data to external I/O devices.

In simulation, an IO Data Source block sends data messages to the **Processor I/O** block using an entity. Together, this mechanism allows tasks simulate using either previously recorded or generated I/O data with timing accurate execution. For more information on entities, see "Entities in an SoC Blockset Model" on page 1-73.

The IO Data Source block and a **Processor I/O** block can be configured to simulate in one of three modes:

- Replay recorded data from file
- From input port
- Zeros

Event-Driven Task

For event-driven task, the recorded data The IO Data Source also sends event messages to the Task Manager block to start the task containing the **Processor I/O** block.

Timer-Driven Task

See Also
IO Data Source | Stream Read | TCP Read | Task Manager | UDP Read

More About
- "Entities in an SoC Blockset Model" on page 1-73

3

Generate Code and Deploy on SoC Device

Supported Third-Party Tools and Hardware

Third-Party Synthesis Tools and Version Support

SoC Blockset supports these third-party FPGA synthesis tools:

- Intel® Quartus® Prime Standard Edition 18.0
- Xilinx Vivado® Design Suite 2018.2

To use third-party synthesis tools with SoC Blockset, a supported synthesis tool must be installed, and the synthesis tool executable must be on the system path.

Third-Party Support for Software Generation

SoC Blockset supports this third-party software generation tool:

- Intel SoC FPGA Embedded Development Suite (EDS) 18.0

Supported Xilinx Devices

SoC Blockset supports execution on Xilinx devices shown in this table.

Device Family	Board
Xilinx Artix®-7	Artix-7 35T Arty FPGA Development Board
Xilinx Kintex®-7	Kintex-7 KC705
Xilinx Zynq	Zynq-7000 ZC706
	ZedBoard™
Xilinx Zynq UltraScale+™	Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit

Supported Intel Devices

SoC Blockset supports execution on Intel devices shown in this table.

Device Family	Board
Intel Arria® 10	Arria 10 SoC Development Kit

Device Family	Board
Intel Cyclone® V	Cyclone V SoC Development Kit

SoC Board Support Packages

The SoC Blockset support packages contain the definition files for all supported boards. You can download one or more vendor-specific support packages. To generate executables and execute on hardware, download at least one of these packages.

To see the list of SoC Blockset support packages, visit "SoC Blockset Supported Hardware" on page 6-2. To download an SoC Blockset support package, on the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Get Hardware Support Packages**.

See Also
"Hardware Implementation Pane Overview" | **SoC Builder**

More About
- "Generate SoC Design" on page 3-12
- "SoC Blockset Supported Hardware" on page 6-2

Configure Design for Generation

In this section...
"Create FPGA Model and Include It in Top-Level Model" on page 3-4
"Configure Reference Design" on page 3-4
"FPGA Model Guidelines" on page 3-5

To generate an FPGA reference design from an SoC Blockset design, you must create a top-level model and an FPGA model. The FPGA model must be included in the top-level model using model referencing. You can then generate the FPGA design directly from the top-level Simulink model using HDL Coder.

Before generating the reference design from your FPGA algorithm, perform these configuration steps:

1   Create an FPGA model and include it in the top-level model.

2   Configure the reference design.

3   Ensure the FPGA model meets recommended guidelines.

To learn more about model referencing, see "Model Reference Basics" (Simulink).

Create FPGA Model and Include It in Top-Level Model

Create your FPGA model by creating your FPGA algorithm in a separate Simulink model. Then, in the top-level model, create a single Model reference block and configure it to point to your FPGA model.

Configure Reference Design

To configure the reference design, open the configuration parameters for the model . Select `Hardware Implementation` on the left pane
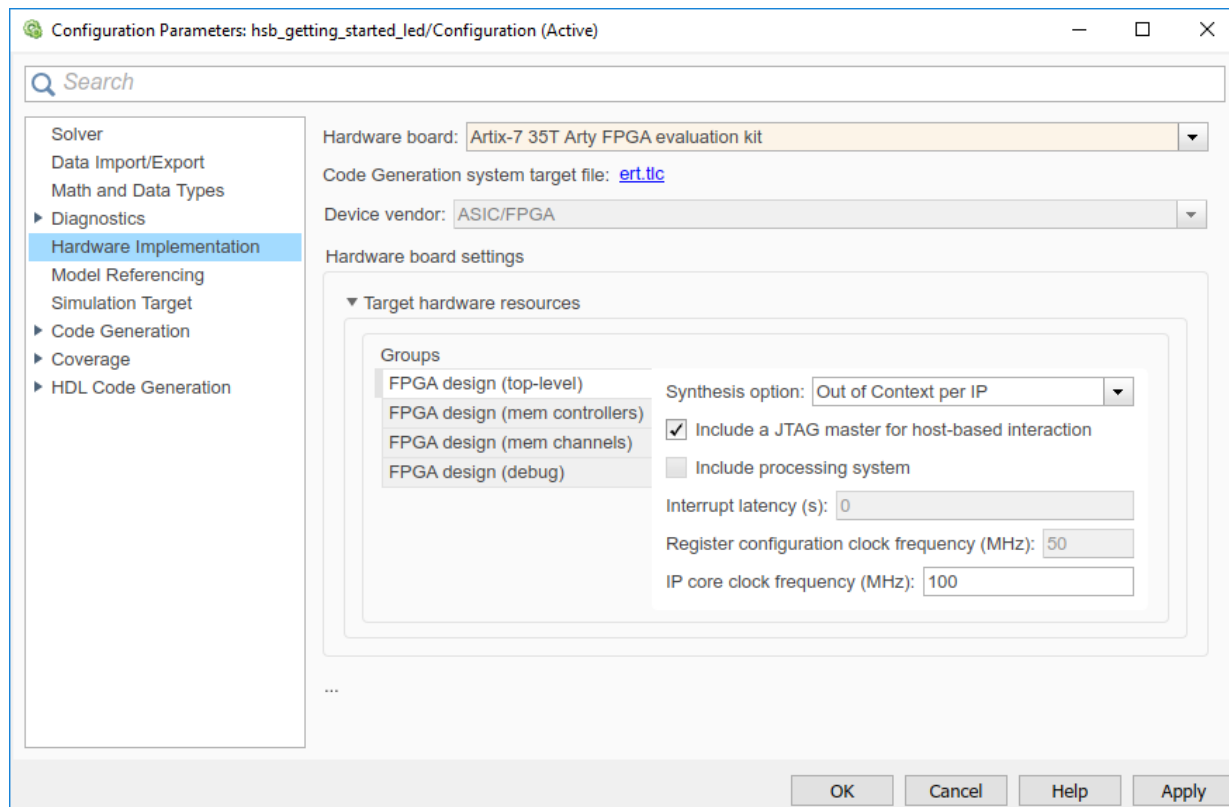
Select the target hardware board, and under **Target hardware resources**, configure the FPGA design tabs according to your implementation requirements. See more about these configuration parameters in "Hardware Implementation Pane".

FPGA Model Guidelines

When constructing the FPGA model within the Model Reference Subsystem, follow these guidelines.

- You can place any algorithm IP in an atomic subsystem.
- You can include multiple algorithm IP subsystems.
- The only SoC Blockset blocks that you can include in an algorithm IP subsystem are the Data Stream Backpressure and Video Stream Backpressure blocks.

- The backpressure blocks allow you to design your algorithm without thinking about properly handling the downstream-ready signal. To see an example of its use for a video application enter the following at the MATLAB command prompt:

  ```
  help soc
  ```

  Then click on the following example:

  Simulating and Generating a Histogram-based Equalization Algorithm with External Memory Frame Buffer

  ---

  Note The Data Stream Backpressure block does not handle FIFO-tlast. One workaround is to generate tlast from the output side of the backpressure block, using the output data valid signal to count transfers.

- To use inferred interfaces, for example AXI-Lite or AXI-Stream, you must connect your atomic subsystem ports to any of the following blocks:

  - Data Stream Wire Channel
  - Video Stream Wire Channel
  - BoardHMIInputs
  - Top-level model blocks, such as memory channels.

- At the top level of the FPGA reference model, you can include a Terminator, Scope and Constant blocks only. Including any other block at this level will cause a subsystem error.

Code Generation of Software Tasks

A Simulink model containing a Task Manager block simulates task execution. When a model gets deployed to an SoC hardware board, the SoC Blockset automatically creates and assigns the tasks to threads, links interrupts, messages, and system events to the generated code of the model.

Timer-Driven Tasks

An SoC Blockset model, when implemented onto hardware as generated and compiled code, uses an operating system (OS) timer to drive the base-rate time step of the model. All time based signals derive their time steps, known as sub-rates, from the base-rate time step of the model. A timer-driven task, created from the Task Manager block, uses a counter that increments at each base-rate timer step. When the counter reaches the integer multiple of the base-rate that defines the rate of the task, the task is triggered by posting to a semaphore associated with that task launches the thread.

Event-Driven Task

Each event-driven task created from the Task Manager block gets a unique semaphore. A unique event elsewhere in the system posts to that semaphore and puts the task thread into the running state. OS kernel handles the management of the task thread until it returns to the waiting state.

See Also

SoC Builder | Task Manager

More About

Configure and Run Design on Hardware

In this section...
"Load FPGA Bitstream" on page 3-8
"Configure and Run AXI Master JTAG Test bench Script" on page 3-8

Load FPGA Bitstream

The *HSB System Report* contains the location of the generated bitstream. To load the bitstream, call the `filProgramFPGA` function. For example:

```
>> filProgramFPGA('Xilinx Vivado', ...
'ProjectDirectory\BitstreamName.bit',1);
```

For Intel boards, change the vendor and tool accordingly. For example:

```
>> filProgramFPGA('Altera Quartus II', ...
'ProjectDirectory\BitstreamName.bit',1);
```

In these function calls, the `chainPosition` argument is set to 1. This argument defines the position of the FPGA on the JTAG programming chain. The value you provide here depends on your development board.

- Digilent® Arty or Xilinx KC705 boards: Set `chainPosition` to 1. For these boards, the FPGA is the only device on the JTAG chain.
- Xilinx ZC706 platform: Set `chainPosition` to 2. On this platform, the ARM processing system is the first device on the chain, and the FPGA is the second device.
- Altera® Cyclone V SoC Development Kit: Set `chainPosition` to 2.
- Altera Arria 10 SoC Development Kit: Set `chainPosition` to 1.

Configure and Run AXI Master JTAG Test bench Script

The generated JTAG test bench shows you how to set up MATLAB as AXI Master and configure your FPGA design over JTAG. You can customize the script to create your own test bench.

The generated script enables you to transition easily from simulating your design in Simulink, to verifying the implemented hardware system. Where possible, the script is populated with values from the corresponding simulation parameters.

The generated script is divided into several sections:

- **Register space for instantiated memory access IP**. This section provides information on any IP that is instantiated in your FPGA design, based on the Memory Channel blocks in your Simulink model. Possible IPs include DMA or VDMA cores when interfacing with AXI4-Stream and AXI4-Stream Video interfaces respectively.

  The helper function `hsb.util.get_ip_offsets`, returns the pertinent configuration register addresses for any memory access IP that is instantiated in your FPGA system. The base address of any IP is automatically populated in the generated JTAG script. An example from the Getting Started example is provided here:

  ```
  % S2MM_DMA DMA Component located at 0x40400000.
  % Generated from hsb_getting_started_led/Memory Channel Subsystem
  dma_s2mm_info = hsb.util.get_ip_offsets('s2mm_dma', '40400000');
  ```

  Here, a Stream-to-Memory-Mapped DMA component has been instantiated, located at address 0x40400000. The Memory Channel block in the example model instantiated the DMA automatically..

  The returned structure `dma_s2mm_info` contains register addresses of the relevant configuration registers for the setup and control of the DMA:

  ```
  dma_s2mm_info =
  struct with fields:
  BASE_ADDR: '40400000'
  S2MM_DMACR: '40400030'
  S2MM_DMASR: '40400034'
  S2MM_DA: '40400048'
  S2MM_LENGTH: '40400058'
  ```

  The register addresses are calculated based on the base address of the IP, provided as an input to the `hsb.util.get_ip_offsets` function.
- *Create AXIMASTER object.* This section creates an AXIMASTER object for using MATLAB as an AXI Master. This object will be used to issue read and write commands over JTAG for configuring your FPGA design.

  Note For additional information about the AXI Master object, see MATLAB AXI Master.
- *User-Defined Parameters.* Any relevant parameters from your Simulink model are included here. Examples include the base addresses of memory regions defined in the

Memory Channel blocks, and video frame size parameters (if configured as Streaming Video).

- *DMA/VDMA Configuration.* If memory access DMA or VDMA components have been instantiated from your system, then this section includes commands to configure the relevant components. The configuration commands use the returned information structure from the **Register space for instantiated memory access IP** section, along with values that have been automatically determined from your Simulink model.

Note If you plan to change the burst or frame size from the values used during simulation, update the configuration values accordingly.

- *Run test bench.* In this section, you can add a test bench stimulus for verifying your FPGA design. A loop is provided as a template, but you can modify this area as you see fit.
- *Release MATLAB as AXI Master object.* Releases the AXI Master object that was created.

Generated IP Core Information.   Along with the script, a MAT-file is generated that contains individual structures for each IP core within the generated FPGA design. Each structure contains:

- Base address of the IP core
- Addresses for configuration registers automatically generated for the relevant AXI4/ AXI4-Stream/AXI4-Stream Video interfaces during IP Core Generation
- Addresses for any registers that you added to your system via the Register Channel blocks

An example of the structure contents, as generated by the Getting Started example, is shown here.

```
led_counter_ip =
struct with fields:
BaseAddr: '43C10000'
IPCore_Reset: '43C10000'
IPCore_Enable: '43C10004'
AXI4_Stream_0_Master_PacketSize: '43C10008'
IPCore_Timestamp: '43C1000C'
start_counter: '43C10100'
```

This example generates a single IP core (`led_counter_ip`) with a single AXI4-Stream Master interface. The fields `IPCore_Reset`, `IPCore_Enable`, and `IPCore_Timestamp`

provide the absolute address of registers that are automatically generated for all generated IP cores. `AXI4_Stream_0_Master_PacketSize` provides the absolute address of the register that is automatically generated for AXI4-Stream Master interfaces. `start_counter` provides the absolute address of a custom register, modeled using a Register Channel block.

Generate SoC Design

In this section...

This tutorial outlines the steps to build hardware and software executables for your model and execute your application. Your SoC model can contain a processor model, an FPGA model, or both.

Step 1: Set Up FPGA Design Software Tools

To generate SoC binaries, you must include the path to Vivado or Quartus executables in your system path. If the executables are not already in your system path, use `hdlsetuptoolpath` function to add them to your path.

Xilinx Software

Use the `hdlsetuptoolpath` function to set up your system environment for accessing Xilinx tools from MATLAB. This function adds the specified installation folder to the MATLAB search path. The following example assumes that Xilinx Vivado is installed at `C:\Xilinx\Vivado\2018.2\bin`.

```
hdlsetuptoolpath('ToolName','Xilinx Vivado', ...
'ToolPath','C:\Xilinx\Vivado\2018.2\bin\vivado.bat')
```

Intel Software

Use the `hdlsetuptoolpath` function to set up your system environment for accessing Intel tools from MATLAB. This function adds the specified installation folder to the

MATLAB search path. The following example assumes that Intel FPGA design software is installed at `C:\Intel\18.0\quartus\bin64`.

```
hdlsetuptoolpath('ToolName','Altera Quartus II', ...
'ToolPath','C:\Intel\18.0\quartus\bin64')
```

Step 2: Start SoC Builder

In the model window, select **Tools > SoC Builder**. A supported board must be specified in the configuration parameters, in the **Hardware Implementation** pane.

Step 3: Prepare Model for Generation

Prepare your model by selecting a starting point for the build process, and then review the model information.
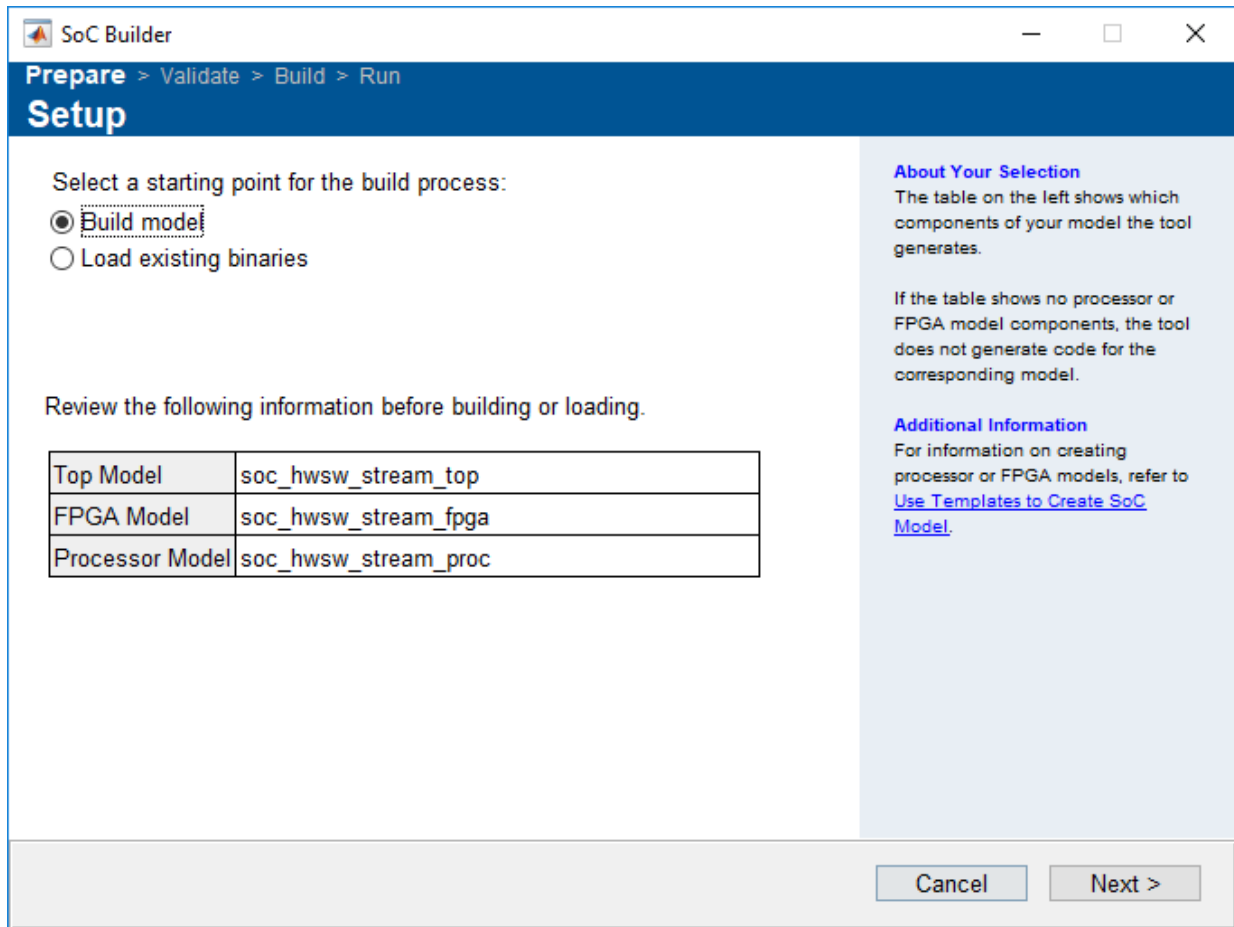
First, specify the starting point for the build process. If you are building a model that you have not built before, select **Build model**. If you previously completed the build process and saved the binaries in a folder, select **Load existing binaries**.

SoC Builder parses the model and displays the top model, the FPGA model (if one exists), and the ARM model (if one exists). Review this information for accuracy. If it seems incorrect, revise the model, save, and restart the **SoC Builder** tool.

Note If your FPGA model is set to a frame-based Simulink model variant, then the **SoC Builder** does not display the model in the table. To make it visible in the table, set the model variant to sample-based and recompile your design.

Click **Next**.

The next page of the **SoC Builder** provides information about the memory map of the model. To open the **Memory Mapper**, click **View/Edit**. Review the base addresses and offsets, and edit them if needed.

Note This memory map step of the **SoC Builder** is visible only if you have an FPGA model in your top model. If your FPGA model is set to frame-based modeling - then no FPGA model is visible, and therefore there is no access to the **Memory Mapper** tool.

Click **Next**.

Step 4: Select Project Folder

Specify a path to a project folder by entering the path in the **Project Folder** text box or by browsing to a folder location. The SoC Builder places all generated files, including reports, executables, and the bitstream, in this specified folder.

If you selected **Load existing binaries** as the starting point for the build process, specify the project folder location of the previous binaries and reports.

Click **Next**.

Step 5: Select Build Action

In the **Select Build Action** section, select one of these options:

- **Build, load and run** – Select this option to generate HDL and C code, build software executables and an FPGA programming file from your model. After building, **SoC Builder** loads the generated code to the FPGA board and executes the application.
- **Build only** – Select this option to generate HDL and C code, build software executables and an FPGA programming file from your model. **SoC Builder** saves the generated binaries in a folder, and you can continue execution later.

- **Build and load for external mode** – Select this option to build the design and run it in external mode. External mode enables you to tune parameters on the FPGA without having to rebuild the FPGA design. It also enables logging data from the FPGA and displaying it on the host. For more information about external mode, see "Tune and Monitor Models Running on Target Hardware" (Simulink).

Step 6: Validate Model

Check the model against the selected board and generate a report. Check the report to ensure that the design is generated as expected.

SoC Builder names the report `<project-folder>/html/`
`modelname_system_report.html` and saves it in the project folder. The report contains an overview section with information about the model, project folder, and generated files. The report also lists user IP cores and vendor-provided IP cores, with the address map of registers and memory blocks.

Step 7: Build Model

To generate a bitstream for your FPGA design and a compiled executable for your software, click **Build**.

Clicking **Build** opens an external shell and runs third-party tools for synthesis and implementation of the design. The generation time depends on the complexity of your model and your host computer. Once the generation is complete, the bitstream is generated with your model name. **SoC Builder** generates a JTAG testbench script if you selected the **Include MATLAB as AXI Master** option in the configuration parameters. The script shows how to set up MATLAB as an AXI Master and configure your FPGA design over JTAG. You can customize the script to create your own test bench. For more information about MATLAB as an AXI Master, see support package documentation: "SoC Blockset Supported Hardware" on page 6-2.

Step 8: Connect Hardware

Review the IPv4 address, SSH Port number, and login credentials. Edit any of these values if necessary. This step is critical if you have more than one board connected to the host computer, so that **SoC Builder** can identify the correct port connection. Verify that the displayed IP address matches the IP address for the board you intend to use.

Verify that the board is connected to the host with an Ethernet cable, and then click **Test Connection** to test the physical connection to the board.

> **Note** This step in the **SoC Builder** is visible only if your top model includes a processor model.

Step 9: Load and Run

Verify that your board is connected to the host computer.

- If a processor model is present in your top model, connect to the board with an Ethernet cable.
- If the top model includes an FPGA model, but no processor model, connect to the board with a JTAG cable.

Click **Load and Run**. This action loads the generated bitstream to the FPGA, programs the processor, and runs the application.

If you selected **Tune parameters and monitor signals in external mode** in step 5, this action loads the bitstream to the FPGA and opens the model in external mode. In this case, you can now choose signals for logging and monitoring or change tunable parameters.

> **Note** If your top model includes an FPGA model, but no processor model, the button shows as **Load**.

See Also
SoC Builder

4

Analyze Performance on SoC Device

Code Instrumentation Profiler

In a code instrumentation profiler, code gets added into the generated code to record the start and stop times of each task executing on the processor. The recorded start and stop times of each task are sent to the development computer to be saved, processed, and displayed. The instantaneous state of each task gets inferred from the combined start and stop times and priorities of all the tasks within the process.

Consider a simple model with two tasks, one high- and low-priority executing on an embedded processor and measured by a code instrumentation profiler. This diagram shows the measurements made by the code instrumentation profiler and the inference on the individual task states resulting from these measurements.
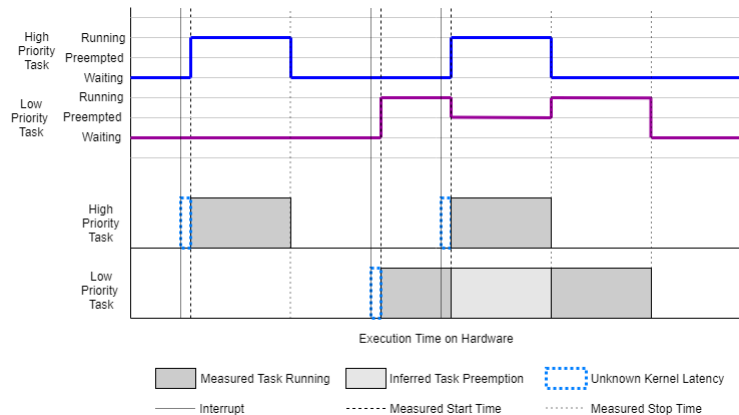
Inspecting the diagram, it shows that the state of the low-priority task gets inferred from the higher-priority task's execution. Since only the start and end times of task execution get measured, some pertinent data can be lost, specifically kernel latency. As kernel latency precedes the start of the task, the actual time of the interrupt event is not directly observed and the start time of the task can assumed to be delayed from the actual time of the interrupt. Furthermore, when a task moves from the preempted to the running states, the kernel latency gets added into the interpreted execution time of the lower-priority task.

Code instrumentation profiling benefits from easy generation and deployment. On models deployed to processors with operating systems running a single process in a single tasking mode, task execution timing measurements be made with sufficient accuracy and precision. As only a minimal amount of code to record the start and stop times of the task

get added to each task, the impact of the task execution timing by the code instrumentation profiler, inmost cases, can be considered negligible.

Limitations

Code instrumentation profiling provides lightweight measurement tooling of generated code. However, two limitations must be considered when measuring the task execution and duration times using the code instrumentation profiler. These limitations are as follows:

- Cannot measure kernel latency or components of kernel latency. Kernel latency can generally be treated as a constant. As kernel latency impacts all task start up time with approximately equal effect, an estimate of the kernel latency could be deduced with comparisons to the task timings in simulation. For more information on kernel latency, see "Kernel Latency" on page 1-11

- Cannot capture the effect of commands issued to the OS kernel from within the task. The code instrumentation profiler records the start time, end time, and preemption of a task by other tasks. However, when the task makes a call to the OS kernel, the code instrumentation profiler does not record the change of control between the task and the kernel as a preemption. As kernel calls, without detailed knowledge of the timing, can be treated as non-deterministic, the measured task duration cannot be reliably measured using this type of profiler. For more information on task duration, see "Task Duration" on page 1-14.

See Also
Task Manager

More About
- "Kernel Latency" on page 1-11
- "Task Duration" on page 1-14

Profile Task Execution on Processor

Task execution timing on an embedded processor can be measured using a code instrumentation profiler. Using external mode, a model with a Task Manager block executing on an SoC hardware board, the code instrumentation profiler streams the execution timing of the task to the host computer, displaying the data in the Simulation Data Inspector (SDI), in real time. Following the profiling of the hardware, the recorded data is saved with the generated code. The data can then be used with the Task Manager block to playback the task execution in simulation.

Task Profiling with External Mode

1   Open the Simulink model to profile.
2   In the **Model Configuration Parameters** > **Hardware Implementation**, set the **Hardware board** to an SoC Blockset supported hardware board.
3   In **Hardware board settings** > **Hardware diagnostics**, enable **Show in SDI**.

▼ Hardware diagnostics

    ☑ Show in SDI

    ☐ Save to file

    Instrumentation: | Kernel ▾

    Recording: | Continuous ▾

    Show: | Task manager tasks ▾

4   (Optional) You can log the measured data to file that gets saved into the `<model>_ert_rtw/instrumented/diagnostics` folder.
5   In the Simulink editor, set the **Simulation mode** to `External Mode`.
6   Click **Run** to deploy and execute the code on the hardware board. A *Simulation Data Inspector* automatically appears showing the streaming data for the tasks and processors on which they are executing.
7   Click **Stop** to end the model execution.

You can access and examine the logged data in the code generation folder used by the model. For more information on accessing the recorded streaming profiled data, see

"Recording Tasks for Use in Simulation" on page 2-18. To use the recorded data in simulation, see "Task Execution Playback using Recorded Data" on page 2-9.

See Also

Task Manager

More About

- "Recording Tasks for Use in Simulation" on page 2-18

Memory Performance Information from FPGA Execution

In this section...
"Performance Plots" on page 4-8
"Burst Waveforms" on page 4-12
"Configuring and Querying the AXI Interconnect Monitor" on page 4-13

Similar to the memory performance plots generated in simulation, you can collect memory interconnect traffic information from a design running on the FPGA. You can then generate similar performance and latency plots. You can also capture transaction information to view in the **Logic Analyzer** tool. Use these memory interconnect metrics to monitor real memory performance, debug and improve the design, and compare them against the memory performance obtained in simulation.

To include an AXI interconnect monitor (AIM) IP in your design, in the configuration parameters of the model, select the **Include AXI interconnect monitor** option under **Hardware Implementation > Target hardware resources > FPGA design (debug)**. The AXI interconnect monitor IP collects information from the design while it is running on the FPGA. This information can be queried from MATLAB by using the JTAG connection. All memory masters in your FPGA are connected to the AXI interconnect monitor IP. These masters can include Memory Channel and Memory Traffic Generator blocks that you generated HDL code for or other masters in your design.
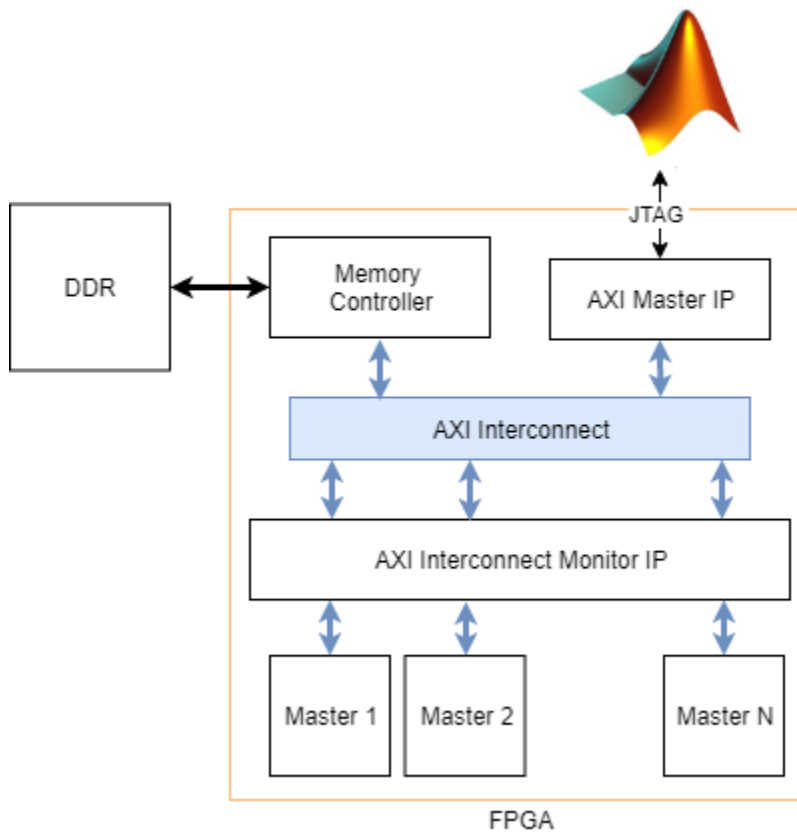
The SoC Builder generates a script that collects and displays the metrics returned by the AXI interconnect monitor. The generated plots are similar to the plots of memory performance in simulation, displaying bandwidth, number of bursts, and transaction latencies. You can also modify the script to collect and display memory control signal waveforms similar to the memory transaction waveforms from simulation. For information on the simulation memory performance tools, see "Simulation Performance Plots" on page 2-22 and "Buffer and Burst Waveforms" on page 2-36.

For an example, see "Analyze Memory Bandwidth Using Traffic Generators" on page 5-51, which shows how to monitor memory performance in both simulation and when running on the FPGA.

Performance Plots

The script generated by the **SoC Builder** tool uses the JTAG connection to enable any traffic generators in your design, and then samples the memory performance information from the AXI interconnect monitor IP as fast as it can. The sampling interval depends on the JTAG latency, which is typically from 10 ms to 20 ms. The script then displays plots similar to the performance plots from the Memory Controller block in your simulation. The plot displays the bandwidth, number of bursts, and transaction latency for each master.
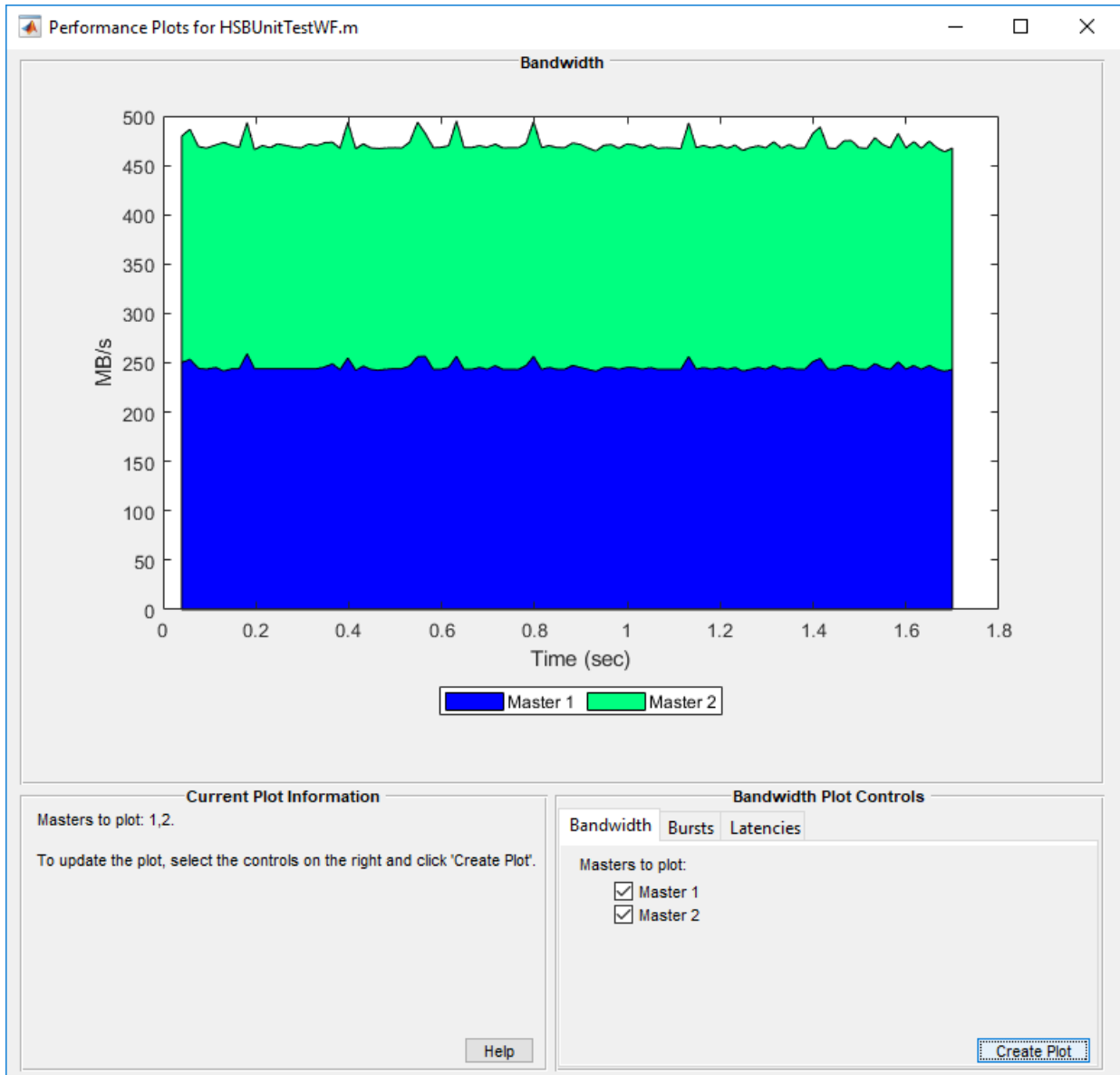
> **Note** The JTAG master itself is not connected to the AXI interconnect monitor. Therefore, the hardware diagnostics do not include the memory usage plots for test-bench-only masters that initialize the memory with predetermined data.

The latency is an average of the time from the start of the burst request to the first data transfer, over the sampling interval. It is expressed as an absolute time. To convert the latency to clock cycles, divide the time value by the clock period of the FPGA.
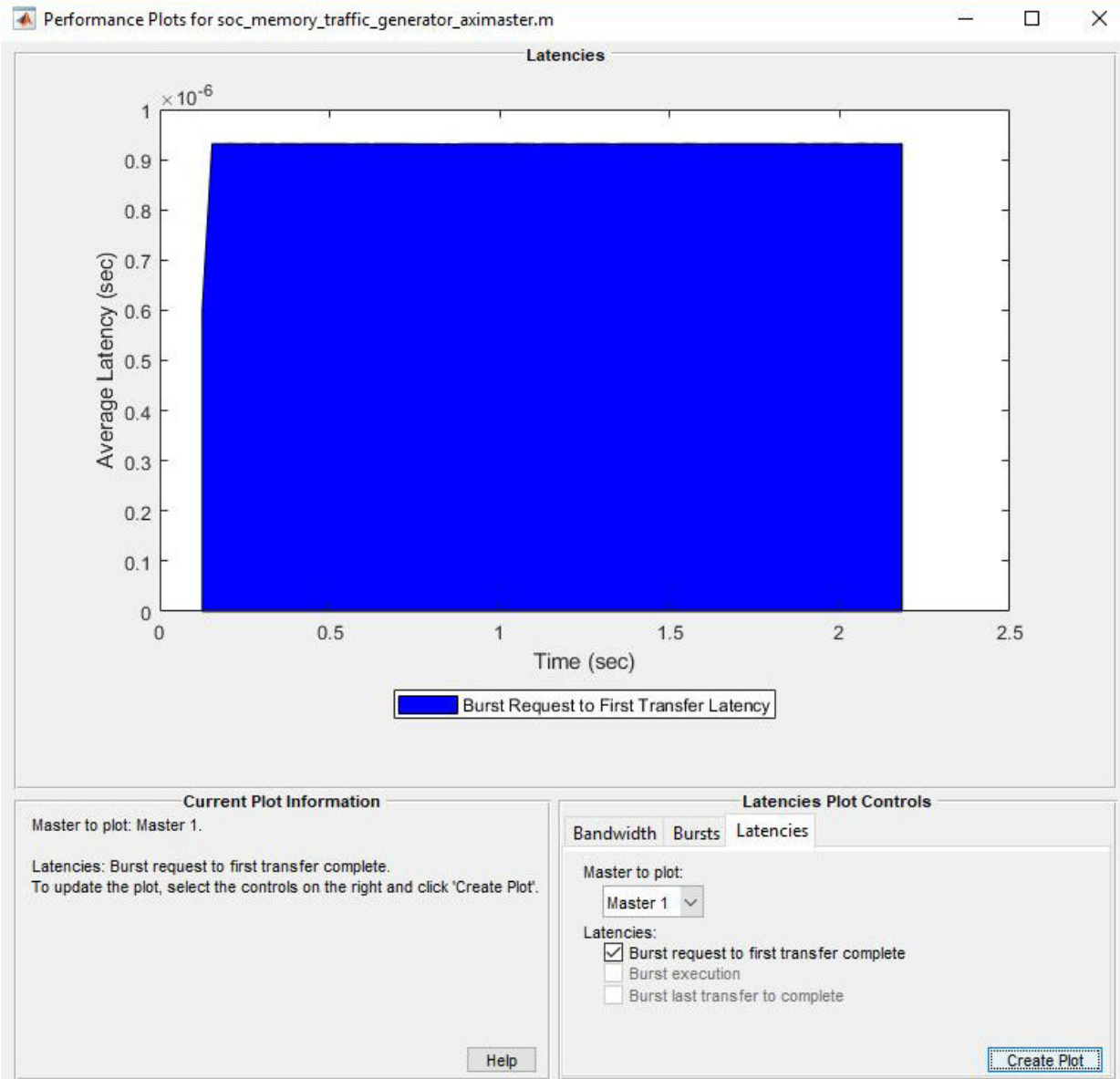
Burst Waveforms

You can also modify the generated script to configure the AXI interconnect monitor to collect event data for each burst transaction. You can view these events in the **Logic Analyzer** waveform viewer to examine arbitration behavior. Specify the number of transactions to capture, **Trace capture depth**, in the configuration parameters of the model, under **Hardware Implementation > Target hardware resources > FPGA design (debug)**.
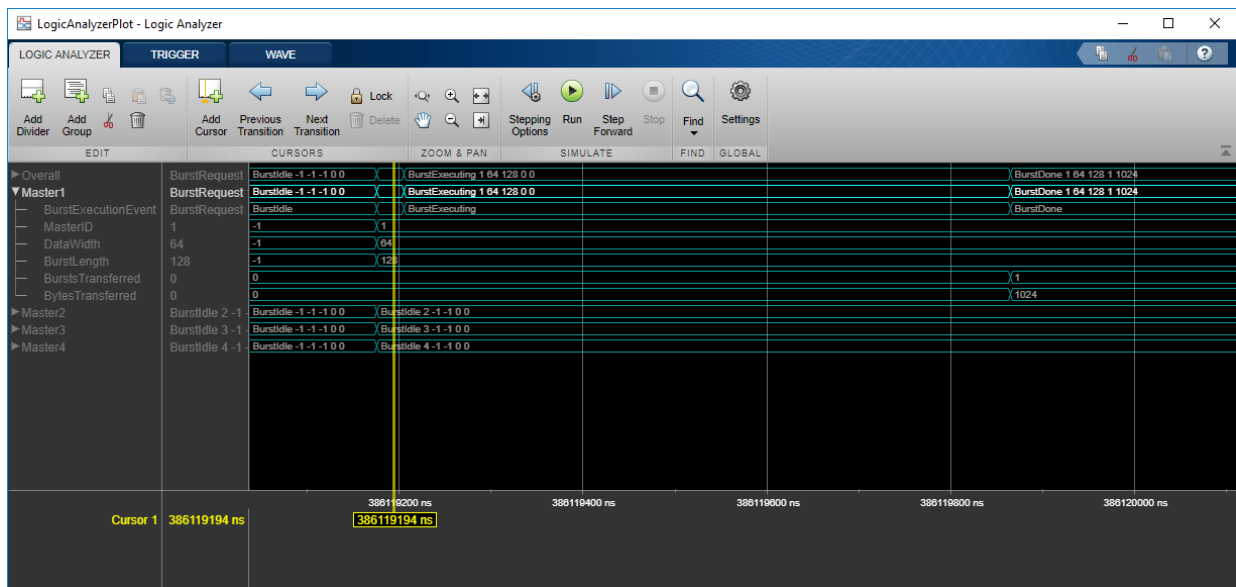
The waveforms show the event type (`BurstIdle`, `BurstRequest`, `BurstExecuting`, or `BurstDone`) and these parameters of the burst transaction:

- `MasterID` –– ID number of the memory master that made the request

- `DataWidth` –– Data width in bits

- `BurstLength` –– Number of data words in the burst request

- `BurstsTransferred` –– Number of bursts in this request (valid only with `BurstDone` event)

- `BytesTransferred` –– Number of bytes in this request (valid only with `BurstDone` event)

You can compare these waveforms with the waveforms captured from your Memory Controller block in simulation.

Configuring and Querying the AXI Interconnect Monitor

The AXI interconnect monitor (AIM) is an IP core that collects performance metrics for an AXI-based FPGA design. Create an `socIPCore` object to set up and configure the AIM IP, and use the `socMemoryProfiler` object to retrieve and display the data.

For an example of how to configure and query the AIM IP in your design using MATLAB as AXI Master, see "Analyze Memory Bandwidth Using Traffic Generators" on page 5-51. Specifically, review the `soc_memory_traffic_generator_axi_master.m` script that configures and monitors the design on the device.

Select Memory Monitor Mode

The performance monitor can collect two types of data. Choose *Profile* mode to collect average transaction latency, and counts of bytes and bursts. In this mode, you can launch a performance plot tool, and then configure the tool to plot bandwidth, burst count, and transaction latency. Choose *Trace* mode to collect detailed memory transaction event data and view the data as waveforms.

```
perfMonMode = 'Profile'; % or 'Trace'
```

Configure the AXI Interconnect Monitor

To obtain diagnostic performance metrics from your generated FPGA design, you must set up a JTAG connection to the device from MATLAB.Load a `.mat` file that contains structures derived from the board configuration parameters. This file was generated by the **SoC Builder** tool. These structures describe the memory interconnect and masters configuration such as buffer sizes and addresses. Use the `socHardwareBoard` object to set up the JTAG connection.

```
load('soc_memory_traffic_generator_zc706_aximaster.mat');
hwObj = socHardwareBoard('Xilinx Zynq ZC706 evaluation kit','Connect',false);
AXIMasterObj = socAXIMaster(hwObj);
```

Configure the AIM. The `socIPCore` object provides a function that performs this initialization. Next, set up a `socMemoryProfiler` object to gather the metrics.

```
apmCoreObj = socIPCore(AXIMasterObj,perf_mon,'PerformanceMonitor','Mode',perfMonMode);
initialize(apmCoreObj);
profilerObj = socMemoryProfiler(hwObj,apmCoreObj);
```

4-13

Retrieve Diagnostic Data

Retrieve performance metrics or signal data from a design running on the FPGA, use the `socMemoryProfiler` object functions.

For `'Profile'` mode, call the `collectMemoryStatistics` function in a loop.

```
NumRuns = 100;
for n = 1:NumRuns
    collectMemoryStatistics(profilerObj);
end
```

JTAG design setup time is long relative to FPGA transaction times, and if you have a small number of transactions in your design, they might have already completed by the time you query the monitor. In this case, the bandwidth plot shows only one sample, and the throughput calculation is not accurate. If this situation occurs, increase the total number of transactions the design executes.

For `'Trace'` mode, call the `collectMemoryStatistics` function once. This function stops the IP from writing transactions into the FIFO in the AXI interconnect monitor IP, although the transactions continue on the interconnect. Set the size of the transaction FIFO, **Trace capture depth**, in the configuration parameters of the model, under **Hardware Implementation** > **Target hardware resources** > **FPGA design (debug)**.

```
collectMemoryStatistics(profilerObj);
```

Visualizing Performance Metrics

Visualize the performance data, by using the `plotMemoryStatistics` function. In `'Profile'` mode, this function launches a performance plot tool, and you can configure the tool to plot bandwidth, burst count, and average transaction latency. In `'Trace'` mode, this function opens the **Logic Analyzer** tool to view burst transaction event data.

```
plotMemoryStatistics(profilerObj);
```

See Also

Memory Controller | collectMemoryStatistics | plotMemoryStatistics | socMemoryProfiler

More About

- "Simulation Diagnostics" on page 2-36

See Also

Related Examples

- "Analyze Memory Bandwidth Using Traffic Generators" on page 5-51

5

Examples

Random Access of External Memory

The example shows how to model external memory accesses from FPGA for rotating an ASCII art image. Many applications require FPGA to access memory in random fashion as per the requirements of algorithm. You will learn how to design memory address generation along with other AXI4 master signals to read and write specific regions of memory using SoC Blockset. You will simulate, implement and verify your design on hardware.

Supported hardware platforms:

- Artix® 7 35T Arty FPGA evaluation kit
- Xilinx® Kintex® 7 KC705 development board
- Xilinx Zynq® ZC706 evaluation kit
- Xilinx Zynq UltraScale™+ MPSoC ZCU102 Evaluation Kit
- Altera® Cyclone® V SoC development kit
- Altera Arria® 10 SoC development kit

Design Task

The ASCII art image is encoded as 24-by-64 matrix of uint8 characters. The design task is to rotate the image by modeling AXI4 Master interfaces in FPGA logic for external memory access. By simulating the design with external memory model and the AXI4 protocol, you verify the behavior at application design time. This saves time otherwise spent in debugging the design on hardware during the implementation phase.

The overall dataflow is as described in figure below. The image is stored in the external memory at the memory region from address 0x00000000 to 0x000017FF. FPGA algorithm reads the image from this region and rotates it by writing it in the reverse order into the memory region from 0x00001800. Finally, the data is read back from the memory.
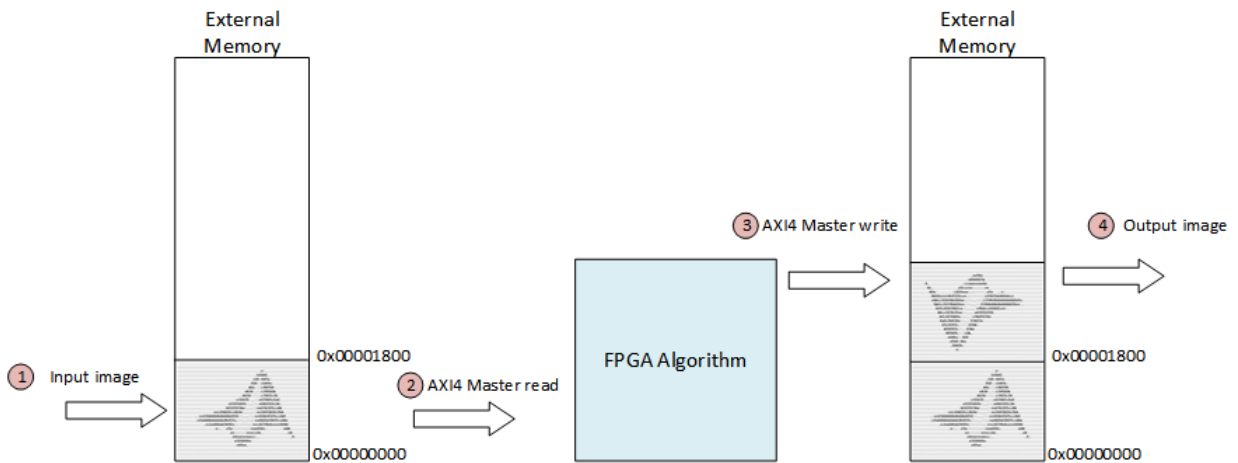
Model Structure

The models are structured using Model references. Top model 'soc_image_rotation' includes the FPGA model 'soc_image_rotation_fpga' using Model block as *model reference*.

Random Access of External Memory

Copyright 2019 The MathWorks, Inc.
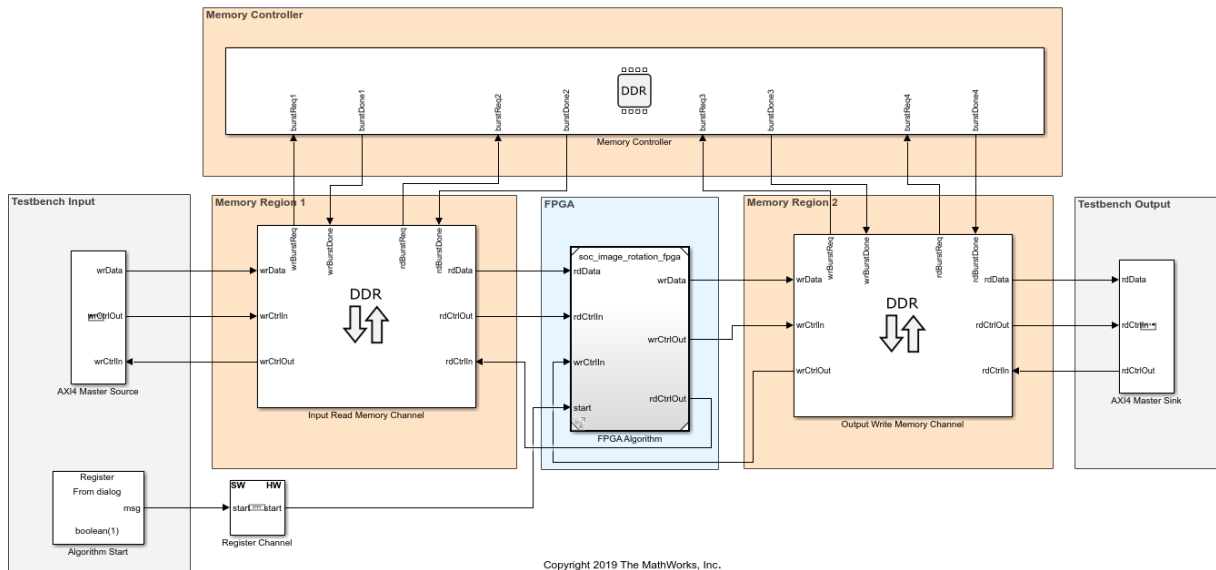
The top model covers the following areas:

- **Testbench Input**: It models the stimuli to set up the design for simulation. The `AXI4 Master Source` block initializes the input image data to the external memory. The `Algorithm Start` block sends a Start signal to the FPGA algorithm via Register Channel block. Open preload function soc_image_rotation_init.m to see how model parameters and input data are initialized.

- **Testbench Output**: The `AXI4 Master Sink` block models the reading of the output image data from the external memory. The output data is saved in the variable AXI4MasterSinkContent in the workspace. Open stop function soc_image_rotation_post.m to see how input data and output data are plotted.

- **Memory**: Memory system is modeled using one Memory Controller and two Memory Channel blocks. `Input Read Memory Channel` block models memory region 1 where input image is stored and `Output Write Memory Channel` block models memory region 2 where the rotated image is stored.

- **FPGA**: This area instantiates the FPGA model reference which models the logic for AXI4 Master interfaces and data rotation.

FPGA model implements the algorithm in three subsystems, AXI4MasterRead, ReverseDataOrder and AXI4MasterWrite. Open FPGA subsystem for image rotation:
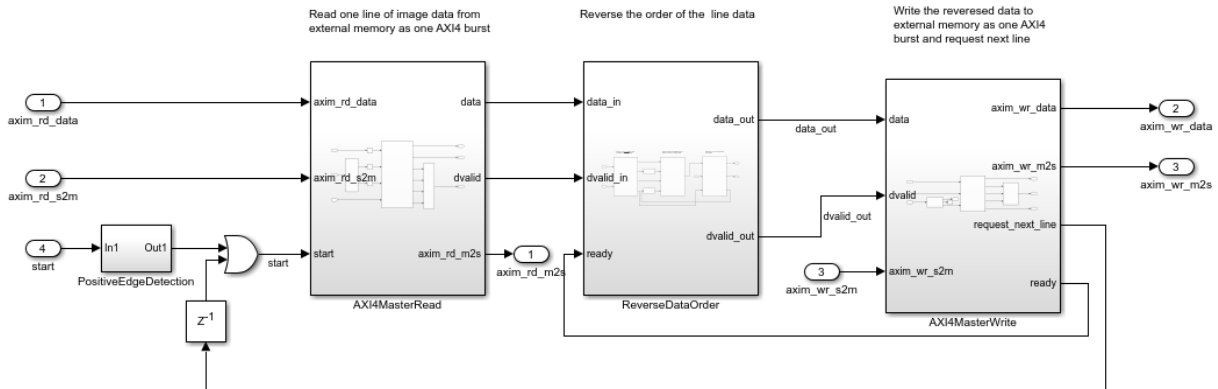
As the positive edge of start signal is detected, AXIMasterRead reads one line of image data and deliver it to ReverseDataOrder for reverses the order of data. The reversed data is then written to external memory by AXIMasterWrite subsystem. Once the data for one line is written, it sends a signal request_next_line to trigger reading of next line by AXIMasterRead. This cycle continues until all lines of the image are processed.

Open AXI4MasterReadController and AXI4MasterWriteController blocks to inspect the MATLAB® code for AXI4 Master interfaces. These blocks design the addressing logic for read and write operations as per AXI4 protocol. SoC Blockset supports AXI4 Master protocol and for timing diagrams of AXI4 signals, please refer to Model Design for AXI4 Master Interface Generation.

Simulation

Run the model and open the Logic analyzer from the FPGA model. Notice the following key points:

- One line of data is written/read by masters in one burst. Since each line is 64 characters long; the burst length is 64 (0x400). Note this value on signals rd_len and wr_len.

- Each character has 4 bytes as it is extended to unit32 data type, which makes the length of line 64x4 = 256 (0x100) bytes. Therefore, addresses increment/decrement by 0x100. Note this on rd_addr and wr_addr signals.

- One read burst is followed by one write burst. Observe how `rd_dvalid` and `wr_dvalid` toggle alternatively.
- `request_next_line` asserts after each write burst, which trigger the next read burst.
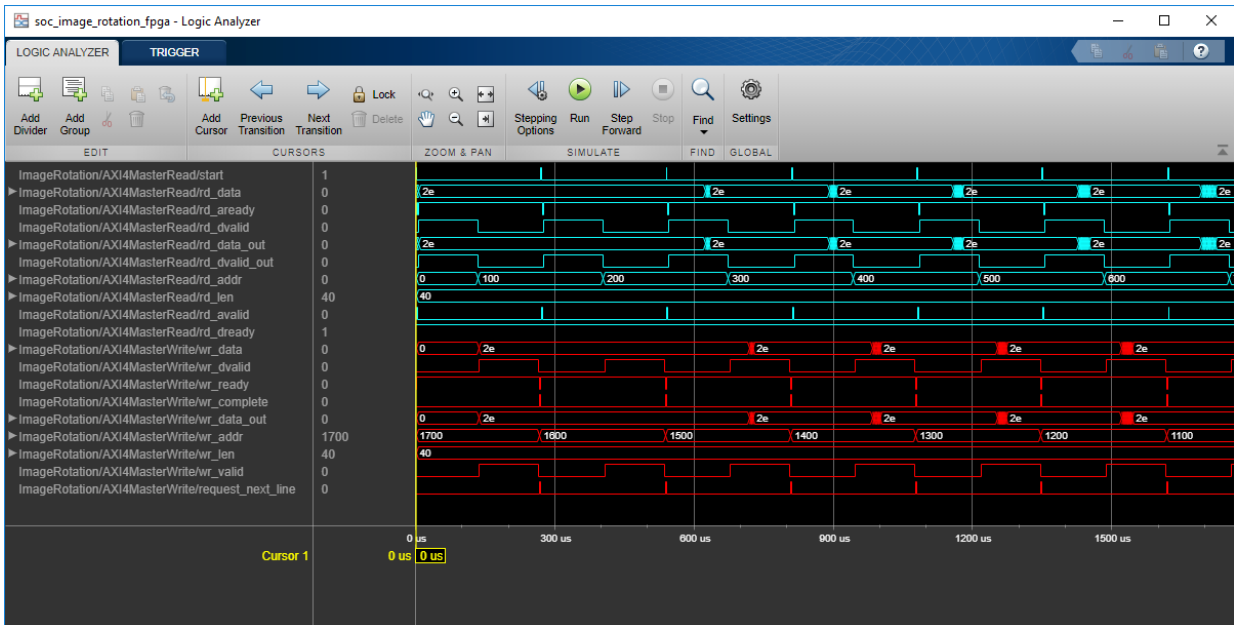
The input and output images are plotted at the end of simulation:

Implementation

Following products are required for this section:

- HDL Coder™
- SoC Blockset Support Package for Xilinx Devices, or
- SoC Blockset Support Package for Intel® Devices

To implement the model on a supported FPGA board, use the SoC Builder application. Make sure you have installed required products and FPGA vendor software before implementation.

Open
<docid:soc_ref#mw_c7ffe7b7-da90-4248-9010-19b4413be7ab SoC Builder> from
the Tools menu and follow these steps:

- Select 'Build Model' on 'Setup' screen. Click 'Next'.

- Click 'View/Edit Memory Map' to view the memory map on 'Review Memory Map' screen. Notice that the base address 0x00000000 is assigned to `Input Read Memory Channel` block, and base address 0x00001800 is assigned to `Output Write Memory Channel` block. The AXI4 address is the sum of base address and address from FPGA algorithm. For example, `wr_addr` from FPGA algorithm starts with 0x1700. The output data will be written to the external memory from address 0x00001800 + 0x1700 = 0x00002F00. Refer to Model Design for AXI4 Master Interface Generation for more information about base address register calculation. Click 'Next'.

- Specify project folder on 'Select Project Folder' screen. Click 'Next'.

- Select 'Build, load and run' on 'Select Build Action' screen. Click 'Next'.

- Click 'Validate' to check the compatibility of model for implementation on 'Validate Model' screen. Click 'Next'.

- Click 'Build' to begin building of the model on 'Build Model' screen. An external shell will open when FPGA synthesis begins. Click 'Next' to 'Load Bitstream' screen.

The FPGA synthesis may take more than 30 minutes to complete. To save time, you may want to use the provided pre-generated bitstream by following these steps:

- Close the external shell to terminate synthesis.

- Copy pre-generated bitstream to your project folder by running the command below and then,

- Click 'Load' button to load pre-generated bitstream.

```
>> copyfile(fullfile(matlabroot,'toolbox','soc','socexamples','bitstreams','soc_image_
```

To run this example, copy the example test bench to your project folder.

```
>> copyfile(fullfile(matlabroot,'toolbox','soc','socexamples','soc_image_rotation_axima
```

Enter the following command to run the test bench:

```
>> soc_image_rotation_aximaster
```

The test bench performs the following operations:

- Initializes image rotation IP
- Writes input image data to external memory
- Starts the image rotation operation
- Reads back and display output image data from external memory

If your FPGA board is not Xilinx Zynq ZC706 evaluation kit you need to do the following settings in the configuration parameters of the top model before launching the SoC Builder.

- Select the 'Hardware board' under 'Hardware Implementation' panel to match your board.
- Uncheck 'Include processing system' under 'Hardware Implementation -> Target hardware resources -> FPGA design (top-level)' panel.
- Set 'Interconnect data width (bits)' to '32' under 'Hardware Implementation -> Target hardware resources -> FPGA design (mem channels)' panel.

Available pre-generated bitstreams are:

- 'soc_image_rotation_fpga-zc706.bit'
- 'soc_image_rotation_fpga-arty.bit'
- 'soc_image_rotation_fpga-kc705.bit'
- 'soc_image_rotation_fpga-a10soc.sof'

Modify the copyfile command and example test bench to match your board and selected project folder as appropriate. Note that pre-generated bitstream may not work if you customized the memory map.

Enter the following command to close the top model and FPGA model:

Conclusion

This example shows modeling of AXI4 Master interfaces for accessing external memory in random fashion using SoC Blockset by rotating an ASCII art image. You can use this as a guide to design your own algorithm to access memory directly using AXI4 Master protocol.

Packet-Based ADS-B Transceiver

Packet-based systems are common in wireless communications. Data is received over the air and is decoded as discrete packet data on a compute device. For given system requirements, it is difficult to design a system and implement directly on SoC as it often involves long iterations of debugging and integration on hardware since hardware effects are difficult to account for at design time. In this example, you will design packet-based airplane tracking application based on Automatic Dependent Surveillance Broadcast (ADS-B) standard, partitioned between FPGA and embedded processor. Unlike traditional methods, you will simulate the application design with memory interface before implementation on hardware using SoC Blockset to shorten development time. You will then validate the design on hardware by automatically generated code from the model.

Supported Hardware Platforms:

- Xilinx® Zynq® ZC706 evaluation kit + Analog Devices® FMCOMMS2/3/4 card.
- ZedBoard™ + Analog Devices FMCOMMS2/3/4 card.

Design Task and System Requirements

As per ADS-B standard a message packet contains a total of 120 bits which has an 8 bit preamble and 112 bits of information about the aircraft including its position and velocity. For an introduction to the Mode-S signaling scheme and ADS-B technology for tracking aircraft, refer to the 'Airplane Tracking Using MATLAB®' example in Communications Toolbox.

Our task is to design a system to receive ADS-B messages off the air and decode with following performance requirements:

- Latency: 0.5 seconds
- Drop sample rate: < 1 in 105 messages
- Throughput: 0.125 MBps (for capacity of maximum 300 aircrafts)

Design Using SoC Blockset

Design Parameters : Data is transferred from FPGA to processor across shared memory as a frame of samples. There are two key design parameters, **Frame Size** and **Number of Buffers** which affect the above performance requirements.

- **Frame Size** : Frame Size is the number of samples in a frame.

- **Number of Buffers** : Number of frame buffers in memory channel. Data is continuously written into memory by FPGA algorithm as frame buffers which are then read by processor to execute its identification algorithm task.

Select the design parameters to satisfy the system requirements as follows:

Design to Meet Latency Requirement : Latency is the time period between when the data is received by the FPGA logic and the data is ready to be processed by the processor. It comprises of two parts, latency through the FPGA logic and the latency for the processor to be available to process data.

Latency through the FPGA logic is the time required for data processing through the FPGA. This is typically on the order of a number of clock cycles with the clock running in MHz range. Latency for the processor to be available to process data, is determined by the time it takes for samples to transfer from FPGA to processor through FIFO and memory frame buffers. If we size FPGA FIFO equivalent to one frame buffer, then the maximum latency can be written as follows:

$$MaxLatency = (NumberOfBuffers + 1) * (TimeToGatherAFrame)$$

As the Time to gather a frame is directly proportional to Frame Size, therefore, the maximum latency in the data transfer is directly proportional to Frame Size and Number of buffers.

Time to gather a frame is a constant for continuously streaming applications and is equal to Frame Size times the FPGA output sample time. However, for asynchronous packet-based systems, this time also depends on the frequency of arrival of packets. If you choose a Frame size larger than the packet size, then you may have to wait for an indeterminate time for arrival of all the packets required to make a frame. If you choose the packet size smaller than packet size, then it will adversely affect the throughput. Therefore, for asynchronous packet based systems, Frame Size equal to packet size is a reasonable choice. This allows each packet to transfer to processor as soon as the FPGA processing is completed, thereby reducing the latency.

For this example, the decoded packet length is 112 bits, packed into four 32-bit samples. So, the frame size is 4.

Design to Meet Throughput Requirement: Throughput is the amount of data produced as output per unit of time. This is a function of the data processing in FPGA and the data transfer & processing by processor. For FPGA logic, the data is processed at clock frequencies of the order of MHz and an output is produced every few clock cycles.

For data transfer and processing by processor, it depends on Frame Size. A typical tradeoff is larger Frame size results in higher throughput but it increases the latency. Conversely, a smaller frame size results in lower latency but it decreases the throughput.

Design to Meet Drop Samples Requirement: An application may tolerate occasional drop data caused by the variations in task execution durations. Frame buffers in a memory channel hold data when it can't be immediately processed by the processor. Therefore, increasing the number of Frame buffers reduces the sample drop-outs but it adversely affects the latency as explained earlier.

Choose the Number of Buffers value such that you are able to meet the Drop samples requirement without affecting the maximum latency requirement.

For this example, the mean task duration, as measured on ZC706 is 114us. Each packet duration is 120us. Even if the packets arrive back to back, they can be processed with minimal number of frame buffers since on average the task is processed before the new packet arrives. So, set the number of frame buffers to the minimum possible, 3.

Create an SoC Model: Use the "SDR Template" on page 1-51 for creating an SoC model for wireless communications applications.

Packet-Based ADS-B Transceiver
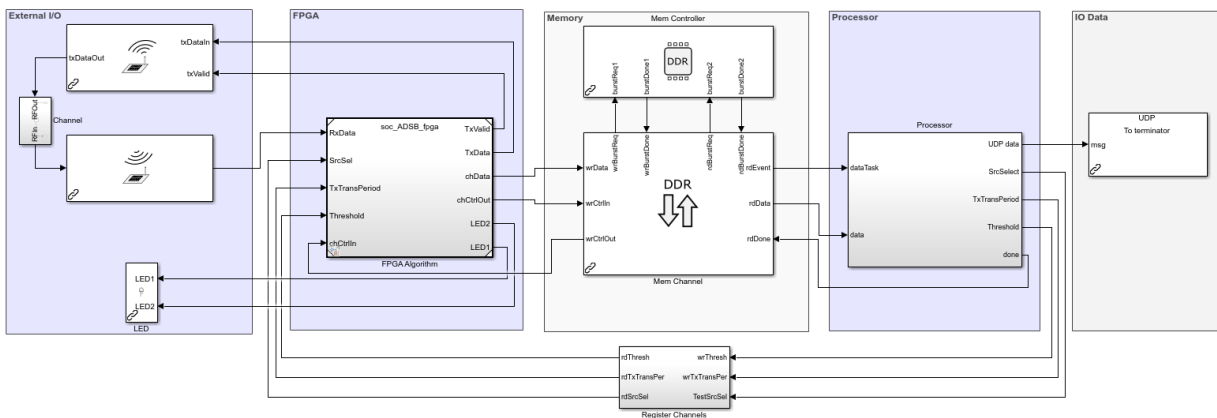
Copyright 2019 The MathWorks, Inc.

The top model is depicted with bounding boxes that segment the model as follows:

- **External I/O** : This part of the model contains the AD9361 RF Input and Output blocks which are connected to each other using a simplified channel model. In addition this region has LED blocks that connect the FPGA logic.
- **FPGA**: The FPGA section of the model contains the FPGA algorithms which are designed in a separate model and instantiated here using model reference.
- **Memory**: This section models the memory channel between FPGA and processor. It simulates the latencies in the HW/SW connection.
- **Register Channel**: This section models three FPGA registers that are configured by the processor.
- **Processor**: This section contains the Task Manager that is connected to processor model. The Task Manager controls the scheduling of processor tasks. The processor algorithm and initialization tasks are modeled in a separate model and is instantiated here using model references.

FPGA model contains *the ADS-B Transmitter Algorithm* that transmits test ADS-B packets at a variable rate and the *ADS-B Receiver Algorithm* that decodes received ADS-B messages.
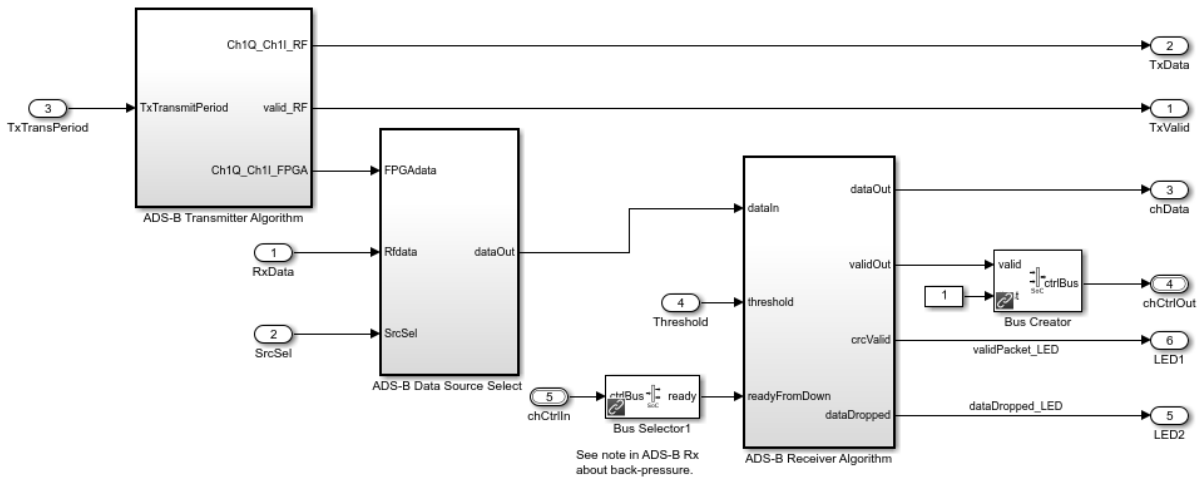
The processor model contains *Processor Algorithm* that unpacks the received ADS-B packets into information bits and sends them via UDP Send block to another system for reporting the aircraft information. The processor algorithm task is denoted as *dataTask* in

the Task Manager block and is specified as event-driven. The Task Manager schedules data asynchronously by means of a buffer ready event *rdEvent* in the memory channel.
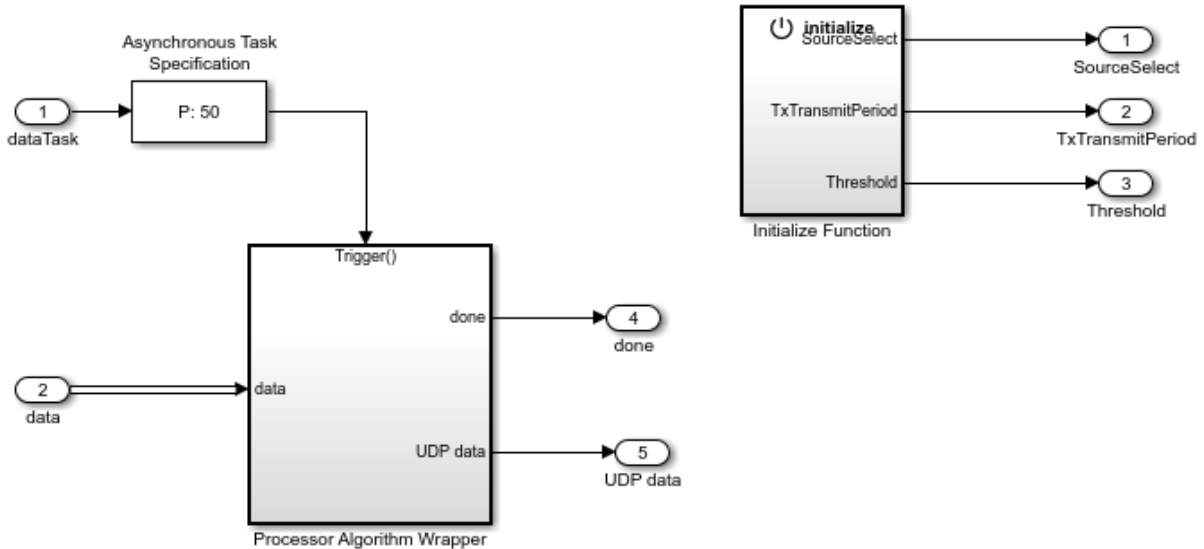
The *Initialize Function* subsystem initializes appropriate hardware configuration registers. The AD9361 blocks set the center frequency, gain mode, and baseband sample rate of the attached FMC RF board. The other blocks model three memory mapped configurations of the ADS-B packet detector datapath. These include selection of input to receiver algorithm, transmit period of test packets from FPGA and threshold value for detection algorithm.

The model *soc_ADSB_UDP_HostPrintout* is a host UDP-based receive model that decodes ADS-B messages. Run this model in parallel to the ADSB simulation or deployment model to display the decoded ADS-B messages and also optionally map the aircraft location.

Host Model for Receiving ADS-B Messages
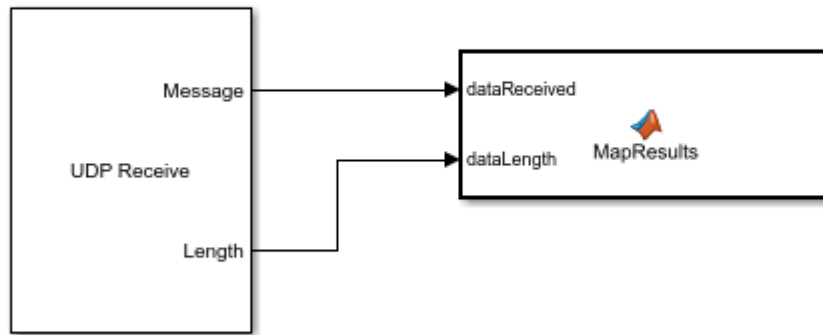
Copyright 2019 The MathWorks, Inc.

Simulate

Run the model to visualize data transfer between the FPGA and the processor. The time period between the arrival of packets is a function of number of aircrafts. Given system requirement of detecting 300 aircrafts, there will be on average 300*6.2 = 1860 messages per second (or a message every 1/1860 = 0.54 ms). You can set the number of aircrafts using the variable *NumAircraft* which in turn sets the period in the *Initialize Function* subsystem. The default setting is 300 to match the allowable system capacity.

Open the Logic Analyzer window to see the waveforms, and notice that the memory transfers are taking place in buffers of 4 samples, or 16 bytes.

To view the external memory bandwidth usage, open the *Mem Controller* block, select the *Performance* tab and click *Launch performance app* . Select all the masters and click *Create Plot*. The plot shows the bandwidth of 0.125 MBps. Since 4 bytes of data is transferred every 32us, the expected bandwidth is 4/32e-6 = 0.125 MBps.
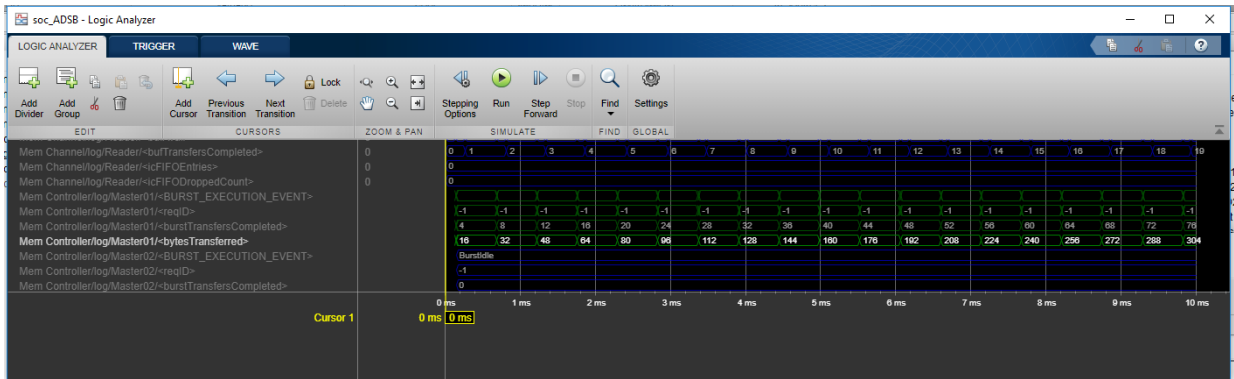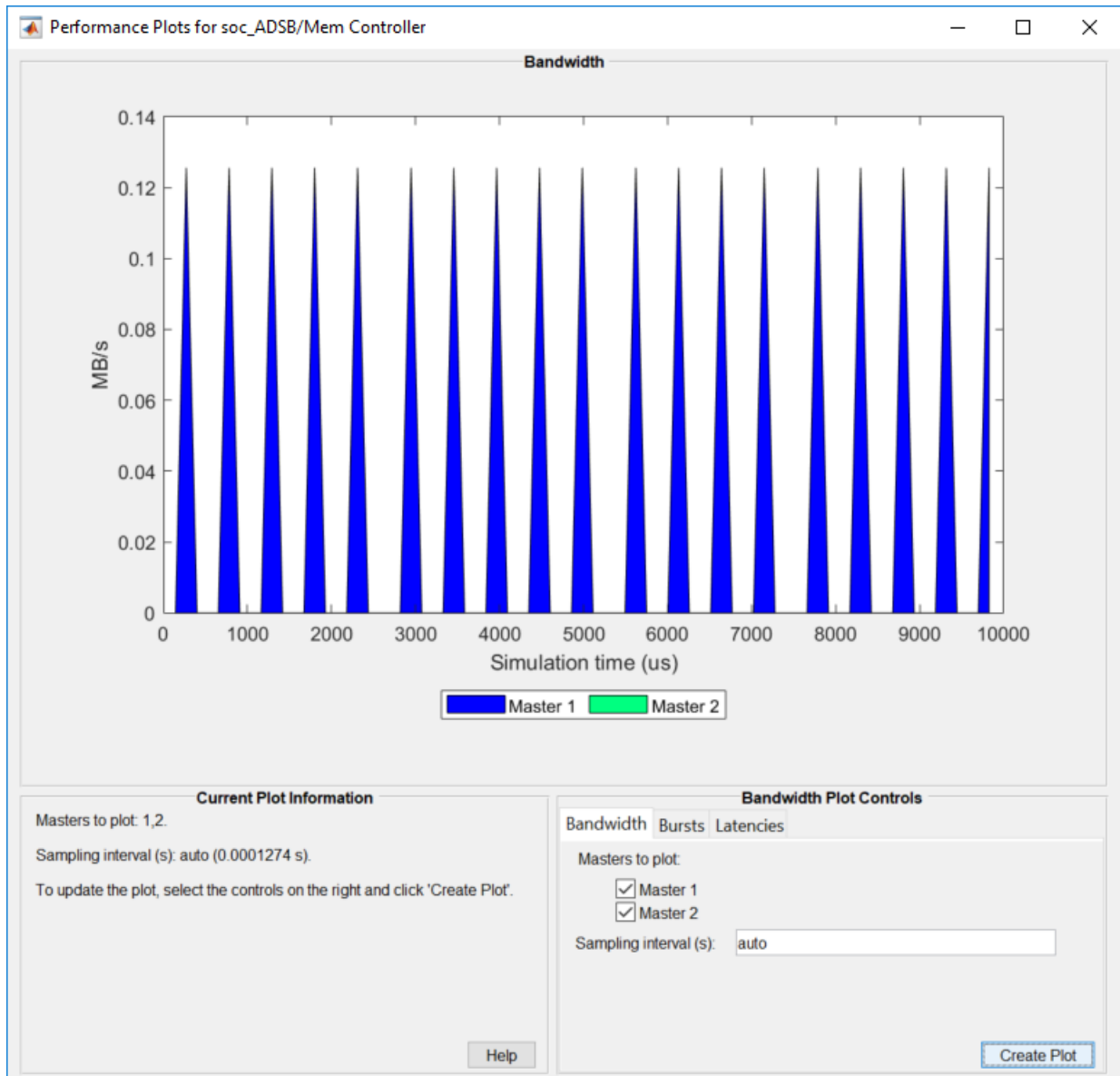
Using the Simulation Data Inspector, you can visualize the task execution schedule. The data task is driven by the event from FPGA notifying the processor that a packet has been decoded by the FPGA, written to external memory, and read by the DMA driver.
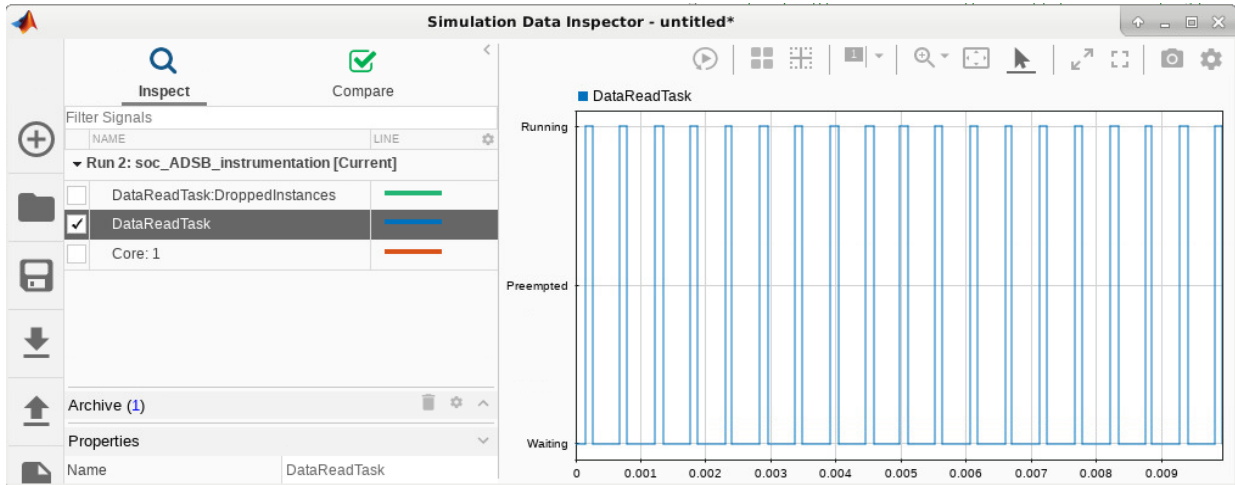
To see the decoded messages, run the companion UDP receive model. This model will display the aircraft tracking information on a GUI.

	Current	Aircraft ID	Flight ID	Latitude(deg)	Longitude(deg)	Altitude(ft)	Speed(knots)	Heading(°)	Vertical Rate(ft/min)	Time
1	✓	3C56EA	EWG8J	55.7467	-4.1555	23050	365	116 (SE)	2560	13:21:24
2		C025A8	WJA2	55.7996	-4.1275	32000	492	317 (NW)	64	13:21:14
3										
4										
5										
6										
7										
8										
9										
10										
11										
12										
13										
14										
15										

Hardware Requirements Analysis

As discussed earlier, since mean task duration of 114us is less than the packet duration of 120us, the messages are not dropped on average, during the transfer to the processor. This is confirmed by looking at the number of dropped samples at FIFO using signal *icFIFODroppedCount* in the Simulation Data Inspector.
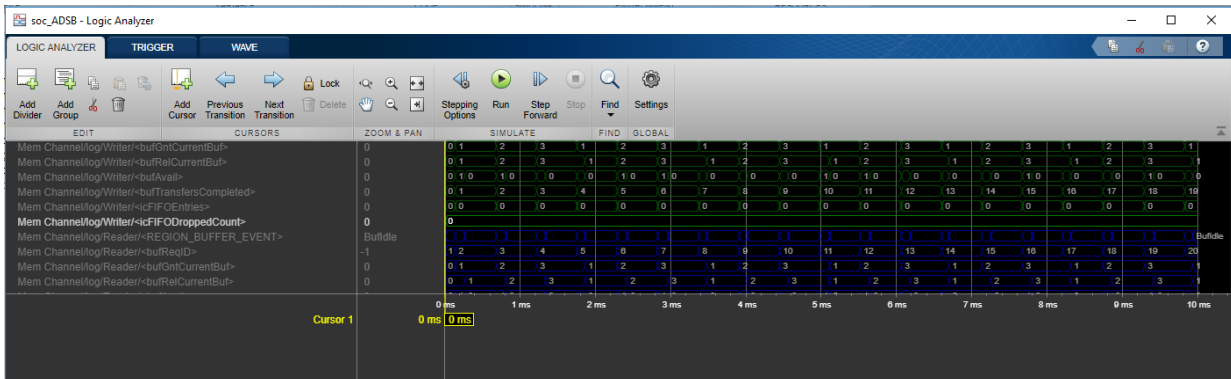
The SoC model can be used to explore the design space. Consider the worst-case scenario when the plane messages are received densely and there is more computation load on the processor. You can modify the model settings and simulate and determine whether packets are dropped in this more aggressive scenario.

Set the *NumAirCraft* to 990 (a new message every 163us) to simulate back to back arrival of plane messages. Modify the task specification on the Task Manager block to simulate more computation load on processor. On the Simulation tab, choose the second distribution by setting the Percent value to 100% on second row and 0% on the first row. This assigns a mean task duration of 163us, which will result in some task executions taking longer than allowed. Set the simulation time to 0.1ms and simulate. For 990 planes, the messages arrival rate is 990*6.2 = 6138 messages per second. The drop packet requirement is therefore, 6138/105 = 58 messages per second or 5.8 messages in 0.1 sec. Upon simulation notice in the Logic Analyzer that this requirement is violated as 18 messages have been dropped.

Implement and Run on Hardware

Following products are required for this section:

- HDL Coder™
- Embedded Coder®
- "SoC Blockset Supported Hardware" on page 6-2

To implement the model on a supported SoC board use the SoC Builder tool. By default, the model will be implemented on **Xilinx® Zynq® ZC706 evaluation kit** as it is configured with that board. Open SoC Builder from the Tools menu and follow these steps:

5-21

- Select 'Build Model' on 'Setup' screen. Click 'Next'.
- Click 'View/Edit Memory Map' to view the memory map on 'Review Memory Map' screen. Click 'Next'.
- Specify project folder on 'Select Project Folder' screen. Click 'Next'.
- Select 'Build, load and run' on 'Select Build Action' screen. Click 'Next'.
- Click 'Validate' to check the compatibility of model for implementation on 'Validate Model' screen. Click 'Next'.
- Click 'Build' to begin building of the model on 'Build Model' screen. An external shell will open when FPGA synthesis begins. Click 'Next'.
- Click 'Test Connection' on 'Connect Hardware' screen to test the connectivity of host computer with SoC board. Click 'Next' to go to 'Run Application' screen.

The FPGA synthesis may take more than 30 minutes to complete. To save time, you may want to use the provided pre-generated bitstream by following these steps:

- Close the external shell to terminate synthesis.
- Copy pre-generated bitstream to your project folder by running the command below and then,
- Click 'Load and Run' button to load pre-generated bitstream and run the model on SoC board

```
>> copyfile(fullfile(matlabroot,'toolbox','soc','socexamples','bitstreams','soc_ADSB_f
```

Implementation on ZedBoard: To implement the model on ZedBoard, you must first configure the model to ZedBoard and set the following example parameters. Open **Model Configuration Parameters**, navigate to **Hardware Implementation** tab and perform the following:

- Select **ZedBoard** from the drop-down list under 'Hardware board' on both top and processor model.
- Navigate to **Target hardware resources > FPGA design (top level)** tab, enable **Include MATLAB as AXI Master IP for host-based interaction** and set **IP core clock frequency (MHz)** to 4 MHz.
- Navigate to **Target hardware resources > FPGA design (debug)** tab and enable **Include AXI Interconnect monitor**.
- Navigate to **Device details** and select **Suport long long** .

Next, open SoC Builder and follow the steps as previously stated for Xilinx® Zynq® ZC706 above. Modify the *copyfile* command to match Zedboard bitstream 'soc_ADSB_fpga-zedboard.bit'.

Profiling Results

To enable processor task profiling, open configuration parameters and navigate to **Hardware Implementation > Hardware Board settings > Task Profiling on processor** and select 'Show on SDI' and 'Save to file'. Set the Simulation stop time to 10 seconds and run the model in external mode. After simulation is completed, open Simulation Data Inspector (SDI) and navigate to the latest run and add signal *DataReadTask* to the plot. Observe that the simulation model accurately predicted how the application would perform on hardware.
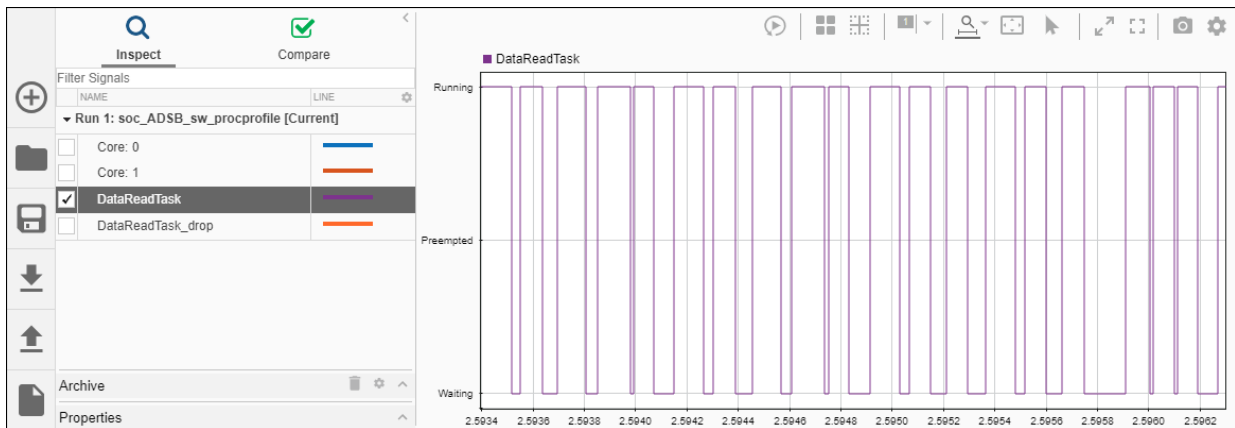
Summary

This example showed how SoC Blockset is used to design packet-based ADS-B standard to meet system requirements. By simulating the design with memory channel as interface between the FPGA and the Processor you validated that the system requirements of throughput and drop packets are met at the design time. You implemented the design on SoC device from the model and verified the results on hardware. Although ADS-B is not a computationally intensive standard, it is useful to demonstrate the design process for packet-based systems intended for implementation on a SoC device. You can follow the same design procedure for even more computationally intensive requirements for this applicaiton or another packet-based application.

Histogram Equalization Using Video Frame Buffer

Video processing applications often store a full frame of video data to process the frame and modify the next frame. In such designs video frames are stored in external memory while FPGA resources are used to process same data. This example shows how to design a video application with HDMI input and output performing histogram equalization using external memory for video frame buffering.

Supported hardware platform

• Xilinx® Zynq® ZC706 evaluation kit + FMC-HDMI-CAM mezzanine card

Design Task and System Requirements

Consider an application involving continuous streaming of video data through the FPGA. The FPGA calculates the histogram of the incoming video stream, in the 'FPGA' subsystem, while streaming the same video stream to external memory for storage. Once the histogram has been calculated and accumulated across the entire video frame, a synchronization signal is toggled to trigger the read back of the stored frame from external memory. The accumulated histogram vector is then applied to the video stream read back from external memory to perform the equalization algorithm. The external memory frame buffer is modeled using the 'Memory Channel' block in `AXI4-Stream Video Frame Buffer` mode.

The 'HDMI Input' block reads a video file and provides video data and control signals to downstream FPGA processing blocks. Video data is in YCbCr 4:2:2 format, and the control signals are in the `pixel control bus` format. The 'HDMI Output' block reads video data and control signals, in the same format as output by the 'HDMI Input' block, and provides a visual output using the Video Display block.

The Push Button block enables bypassing of the histogram equalization algorithm, routing the unprocessed output from the external memory frame buffer to the output.

Histogram Equalization Using Video Frame Buffer
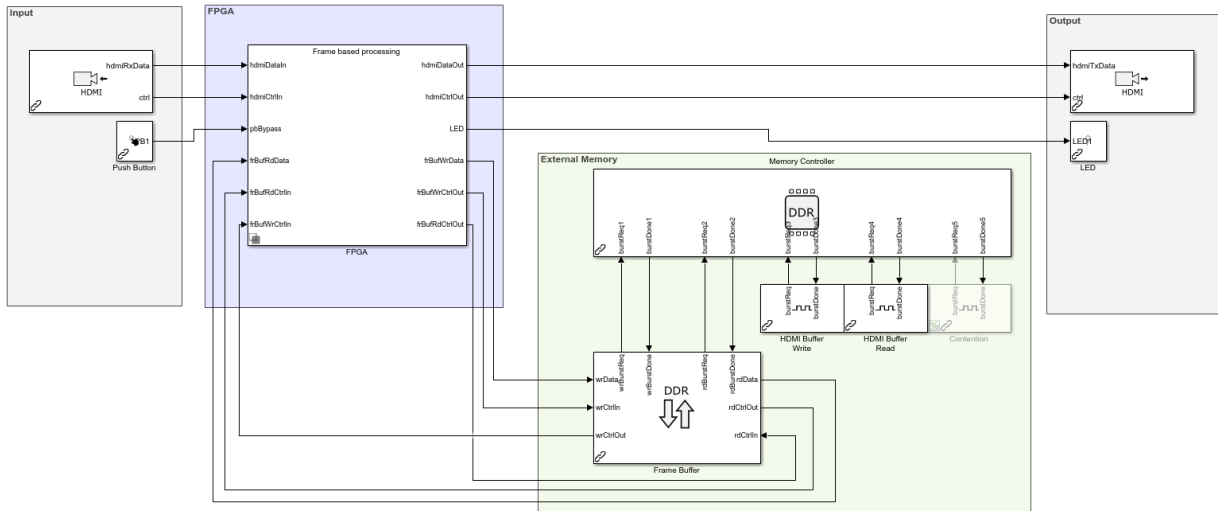
Copyright 2019 The MathWorks, Inc.

There are a number of requirements to consider when designing an application that interfaces with external memory:

- **Throughput**: What is the rate that you need to transfer data to/from memory to satisfy the requirements of your algorithm? Specifically for vision applications, what is the frame-size and frame-rate that you must be able to maintain?

- **Latency**: What is the maximum amount of time that your algorithm can tolerate between requesting and receiving data? For vision applications, do you need a continuous stream of data, without gaps? Are you able to buffer samples internal to your algorithm in order to prevent data loss when access to the memory is blocked?

For this histogram equalization example, we have defined the following requirements:

- Throughput must be sufficient to maintain a 1920x1080p video stream at 60 frames-per-second.

- Latency must be sufficiently low so as not to drop frames.

With the above throughput requirement, we can calculate the value that is required for the frame buffer:

$$1920 \times 1080 \times 60 = 124.416 \, \text{Msps}$$

As the video format is YCbCr 4:2:2, we require 2 bytes-per-pixel (BPP), this equates to a throughput requirement of

$$2 \times 124.416 = 248.832 \, \text{MB/s}$$

Because the algorithm must both write and read the video data to/from the external memory, this throughput requirement must be doubled, for a total throughput requirement of

$$2 \times 248.832 = 497.664 \, \text{MB/s}$$

Design Using SoC Blockset

In general, your algorithm will be a part of a larger SoC application. In such applications, it is likely that there will be other algorithms also requiring access to external memory. In this scenario, you must consider the impact of other algorithm's memory accesses on the performance and requirements of your algorithm. Assuming that your algorithm shares the memory channel with other components, you should consider the following:

- What is the total available memory bandwidth in the SoC system?
- How will your algorithm adapt to shared memory bandwidth?
- Can your algorithm tolerate an increased read/write latency?

By appropriate modeling of additional memory consumers in the overall application, you can systematically design your algorithm to meet your requirements in situations where access to the memory is not exclusive to your algorithm.

To avoid modeling of all memory readers and writers in the overall system, you can use 'Memory Traffic Generator' blocks to consume read/write bandwidth in your system by creating access requests. In this way, you can simulate additional memory accesses within your system without explicit modeling.

Modeling Additional Memory Consumers

When implemented on hardware, the HDMI output requires an additional frame buffer for synchronization of the video stream data between clock-domains, and introduces an additional memory consumer in the overall system. You can model this using `Memory Traffic Generator` blocks to simulate the additional memory consumption. As we are modeling both read and write transactions, we will use two `Memory Traffic Generator` blocks - one each for read and write.

Based on the throughput calculation for our 1080p video stream, we know that the additional frame buffer will require $497.664\,\mathrm{MB/s}$ of bandwidth for simultaneous read and write access.

The write transactions are modeled by `HDMI Buffer Write` and the read transactions are modeled by `HDMI Buffer Read`. The block mask for both are shown below.
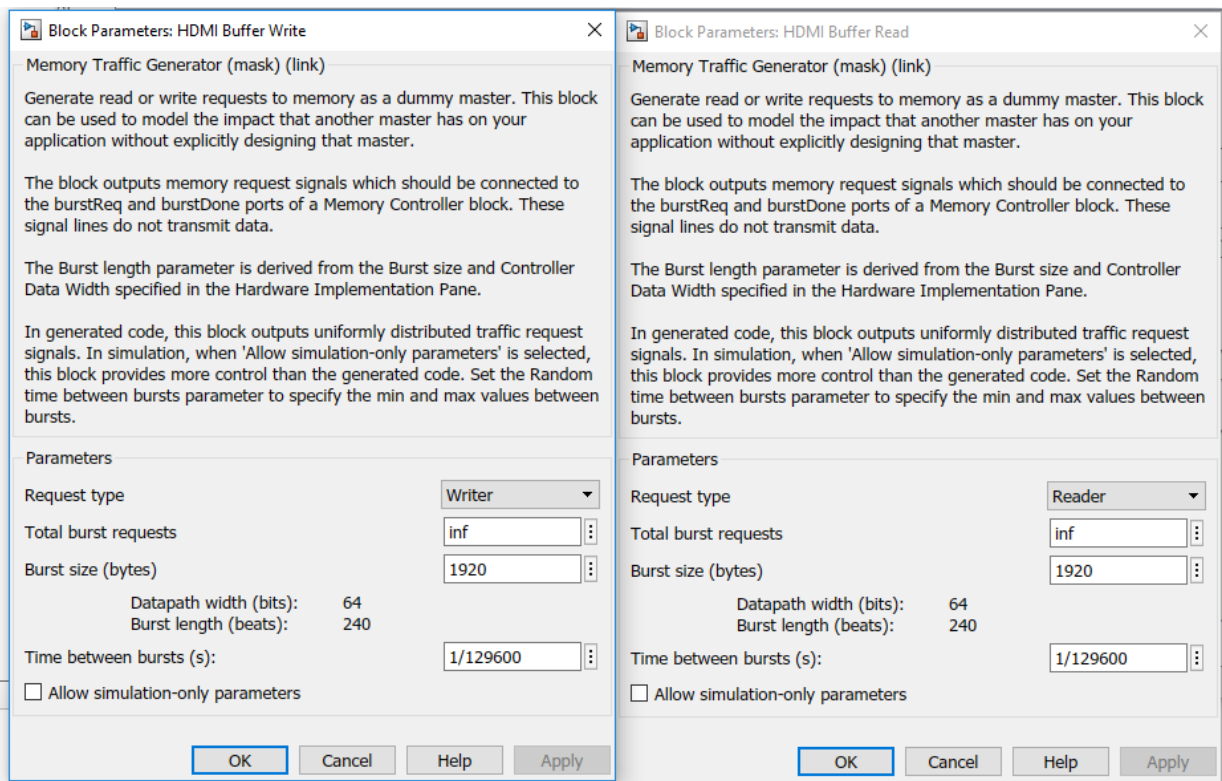
The total burst requests are configured as *inf*, as we want to simulate a continuous stream of data to/from the memory. This will ensure that the traffic generator block will continue to issue transaction requests for the entirety of the simulation.

The burst size is specified as *192*, which is the 1/10th of pixels per line. As the burst size is specified in bytes, this is equivalent to one tenth of a single line of a single component of the output video stream, i.e. a single line of the Y-component of the YCbCr 4:2:2 video stream.

The time between burst is specified as *1/1296000*. This can be expanded as

$$\frac{192}{1080 \times 1920 \times 60 \times 2}$$

where,

192 is the number of bytes per burst,

1080 is the number of lines in the video stream,

1920 is the number of pixels per line in the video stream,

60 is the number of frames-per-second and,

2 is the number of components in our video stream.

Putting the above parameters together, we can calculate our requested throughput as follows:

$$192 \times 1296000 = 248.832\,\mathrm{MB/s}$$

And, as we have two traffic generators to simulate both read and write transactions, the total bandwidth consumption will be $497.664\,\mathrm{MB/s}$

Simulating the system with the above settings results in the following Memory Bandwidth Usage plot.
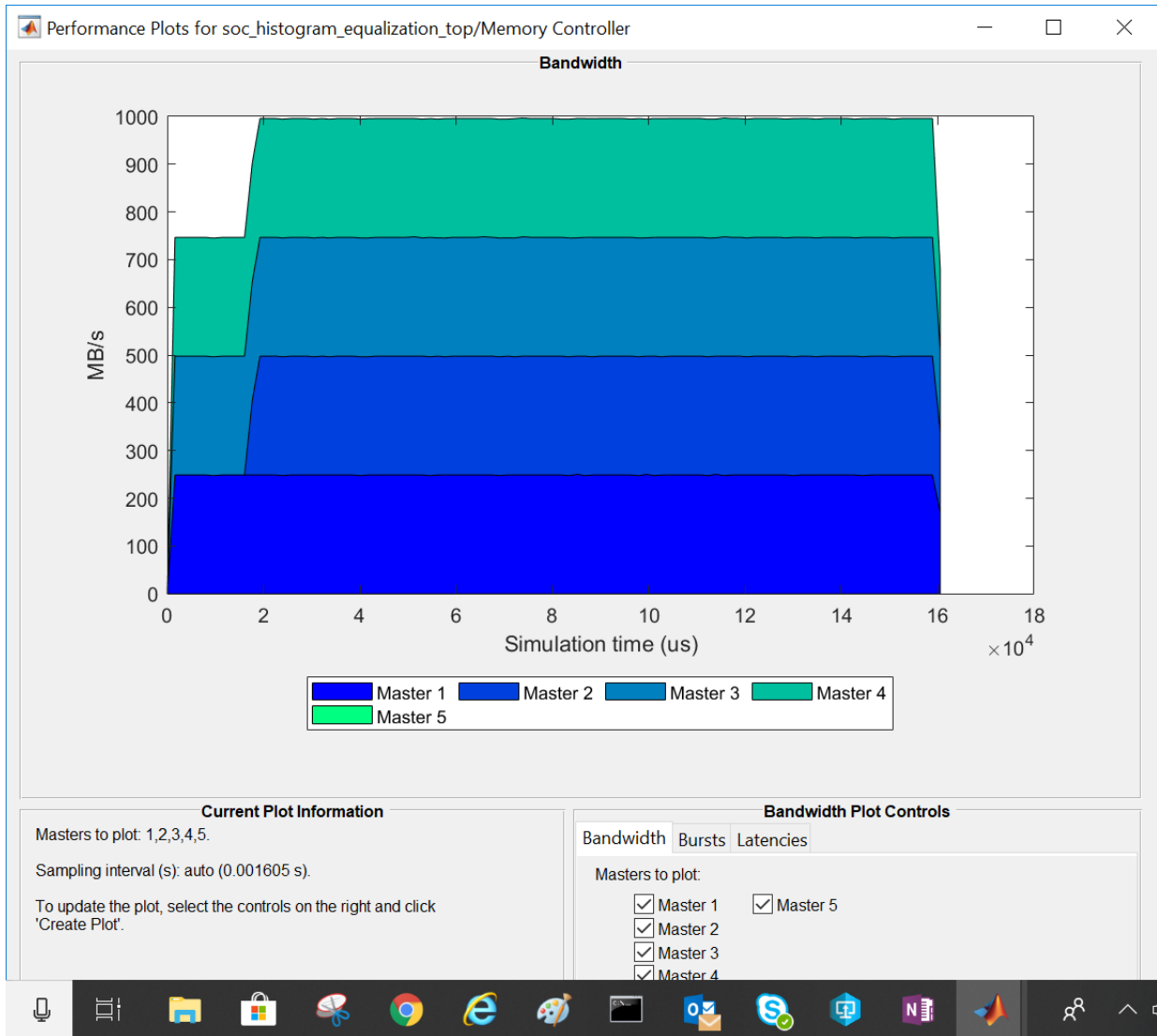
Here, the memory masters are as follows:

1   Master 1: Frame Buffer write

2   Master 2: Frame Buffer read

3 Master 3: HDMI Buffer Write (Memory Traffic Generator)

4 Master 4: HDMI Buffer Read (Memory Traffic Generator

5 Master 5: Contention (Memory Traffic Generator) (commented out)

You can see that all 4 active masters are consuming 248.8 MB/s of memory bandwidth.

More Memory Consumers: Consider that your algorithm is part of a larger system, and a secondary algorithm is being developed by a colleague or third-party. In this scenario, the secondary algorithm will be developed separately for the interest of time and division of work. Rather than combine the two algorithms into a single simulation, you can model the memory access of the secondary algorithm using a Memory Traffic Generator, and simulate the impact, if any, that it will have on your algorithm.

For example, assume that you are provided with the following memory requirements for the secondary algorithm:

• Throughput: 650 MB/s

Given that we know that at any one time the primary algorithm, plus the HDMI output frame buffer, is consuming ~995 MB/s of the memory bandwidth, and our total available memory bandwidth is 1600 MB/s, we know that with the total bandwidth requirement for our system exceeds the total available bandwidth by ~50 MB/s.

To enable the modeling of the secondary algorithm memory access, uncomment the `Contention` Memory Traffic Generator block. The block mask settings are shown below.

Block Parameters: Contention   ✕

Memory Traffic Generator (mask) (link)

Generate read or write requests to memory as a dummy master. This block can be used to model the impact that another master has on your application without explicitly designing that master.

The block outputs memory request signals which should be connected to the burstReq and burstDone ports of a Memory Controller block. These signal lines do not transmit data.

The Burst length parameter is derived from the Burst size and Controller Data Width specified in the Hardware Implementation Pane.

In generated code, this block outputs uniformly distributed traffic request signals. In simulation, when 'Allow simulation-only parameters' is selected, this block provides more control than the generated code. Set the Random time between bursts parameter to specify the min and max values between bursts.

Parameters

Request type	Reader ▼
Total burst requests	50
Burst size (bytes)	2048

Datapath width (bits):   64
Burst length (beats):   256

☑ Allow simulation-only parameters

Simulation-only parameters

First burst time:	0.03
Random time between bursts (s):	[1/332800 1/332800]

☐ Wait for burst done

OK   Cancel   Help   Apply

5-31

Simulating the system with the secondary algorithm's memory accesses, results in the following Memory Bandwidth Usage plot.
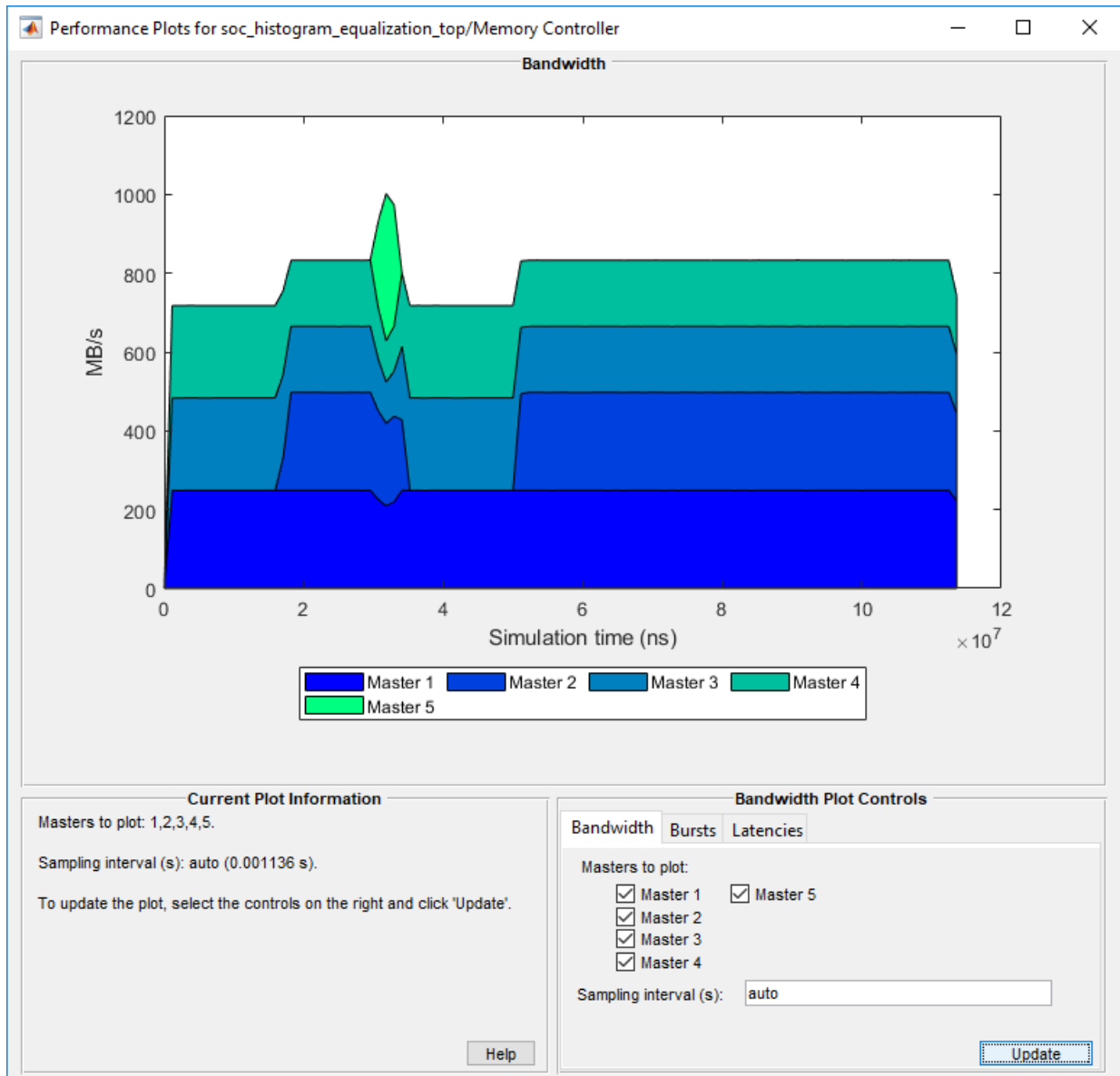
As you can see, the combined required memory bandwidth exceeds the available bandwidth at around 0.03s - when the secondary algorithm begins memory access requests, resulting in the other masters not achieving their required throughput. Looking at the logic analyzer waveform, we can see this manifested as dropped buffers for the Frame Buffer write master, meaning that the input video frame will not be written to memory.
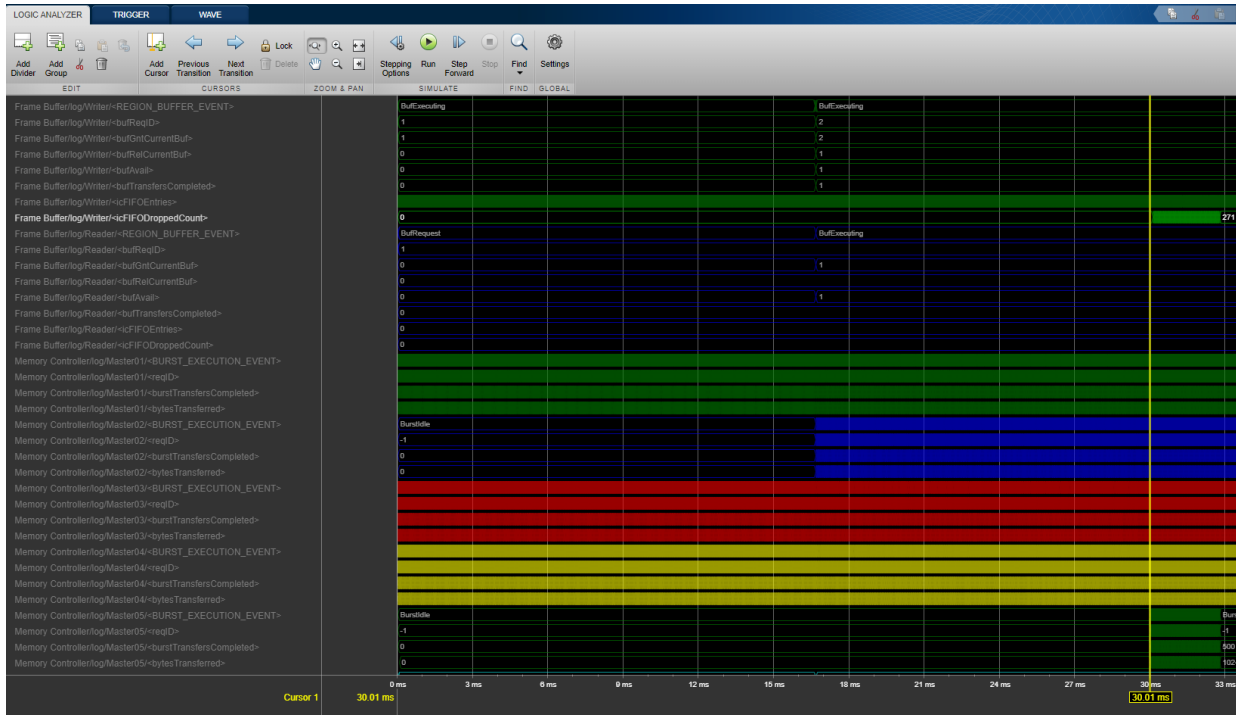
Implement and Run on Hardware

Following products are required for this section:

- HDL Coder™
- "SoC Blockset Supported Hardware" on page 6-2

To implement the model on a supported SoC board use the SoC Builder application. Open the mask of 'FPGA' subsystem and set model variant to 'Pixel based processing'.

Comment out 'HDMI Buffer Write', 'HDMI Buffer Read' and 'Contention' blocks.

Open SoC Builder from the Tools menu and follow these steps:

- Select 'Build Model' on 'Setup' screen. Click 'Next'.
- Click 'View/Edit Memory Map' to view the memory map on 'Review Memory Map' screen. Click 'Next'.
- Specify project folder on 'Select Project Folder' screen. Click 'Next'.
- Select 'Build, load and run' on 'Select Build Action' screen. Click 'Next'.
- Click 'Validate' to check the compatibility of model for implementation on 'Validate Model' screen. Click 'Next'.
- Click 'Build' to begin building of the model on 'Build Model' screen. An external shell will open when FPGA synthesis begins. Click 'Next'.
- Click 'Next' to 'Load Bitstream' screen.

The FPGA synthesis may take more than 30 minutes to complete. To save time, you may want to use the provided pre-generated bitstream by following these steps:

- Close the external shell to terminate synthesis.
- Copy pre-generated bitstream to your project folder by running the command below and then,
- Click 'Load and Run' button to load pre-generated bitstream and run the model on SoC board

```
>>copyfile(fullfile(matlabroot,'toolbox','soc','socexamples','bitstreams','soc_histogra
```

To run the model, execute the following aximaster test bench for soc_histogram_equalization_top_aximaster.

The following figure shows the Memory Bandwidth usage when the application is deployed on hardware.
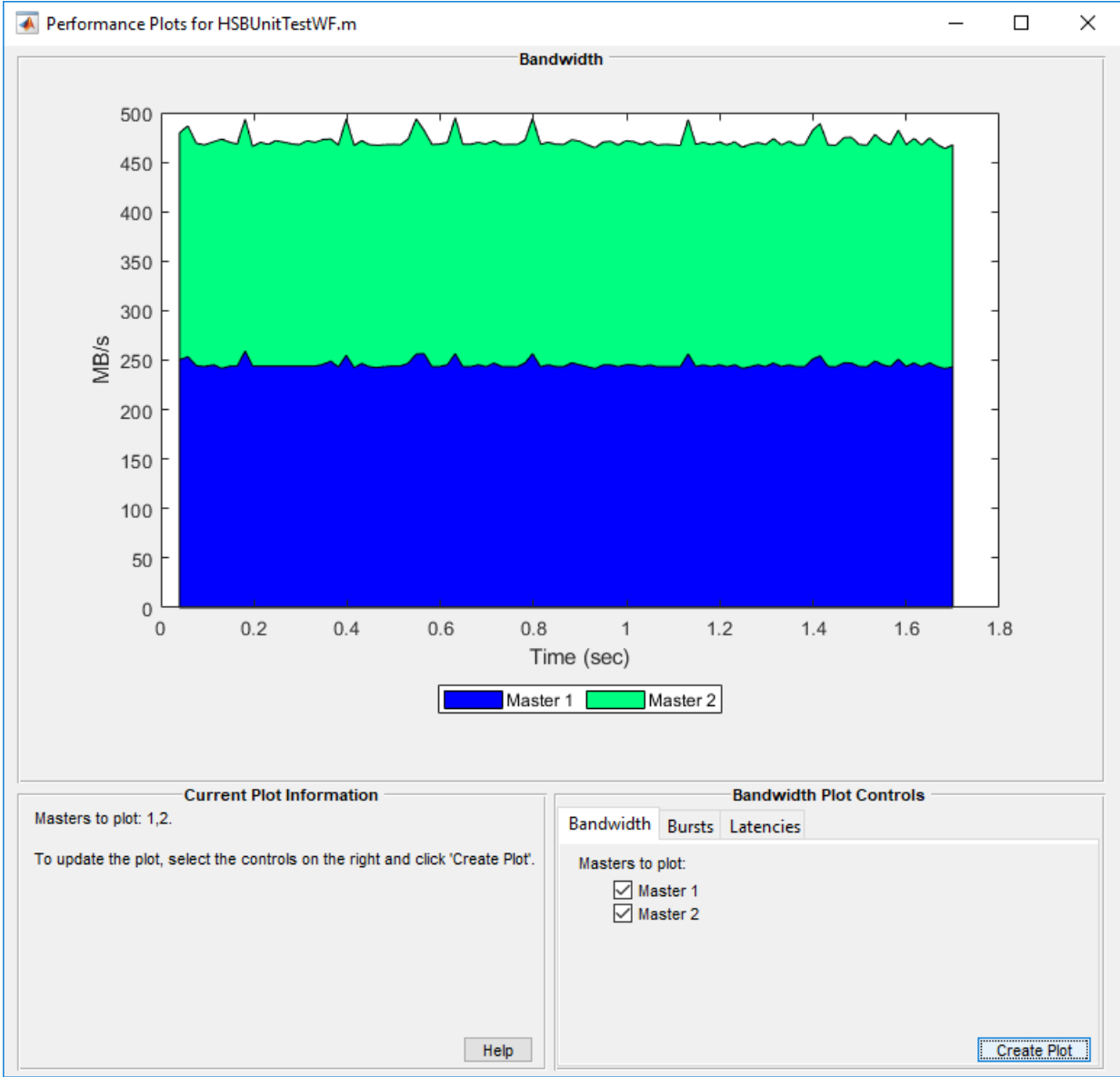
Summary

You designed a video application with real time HDMI I/O and frame buffering in external memory. You explored effects of other consumers of memory on overall bandwidth. You used SoC Builder to implement the model on hardware and verify the design.

Streaming Data from Hardware to Software

This example presents a systematic approach to design the data-path between hardware logic (FPGA) and embedded processor using SoC Blockset. Applications are often partitioned between hardware logic and embedded processor on a system-on-chip (SoC) device to meet throughput, latency and processing requirements. You will design and simulate the entire application comprising of FPGA & processor algorithms, memory interface and task scheduling to meet the system requirements. You will then validate the design on hardware by generating code from the model and implementing on a SoC device.

Supported hardware platforms:

- Xilinx® Zynq® ZC706 evaluation kit
- Xilinx Zynq UltraScale™+ MPSoC ZCU102 Evaluation Kit
- ZedBoard™ Zynq-7000 Development Board
- Altera® Cyclone® V SoC development kit
- Altera Arria® 10 SoC development kit

Design Task and System Requirements

Consider an application that continuously process data on the FPGA and the embedded processor. In this example, the FPGA algorithm filters the input signal and streams the resulting data to the processor. In the implementation model soc_hwsw_stream_implementation, the *Buffer* block represents the transfer of data from FPGA to processor. The processor operates on the buffered data and classifies the data as either high or low frequency in the *Processor Algorithm* subsystem. FPGA generates a test data of either low or high frequency sinusoid based on the DIP switch setting in *Test Data Source* subsystem.

External Input — DIP Switch — Low Freq Signal — true / false — High Freq Signal — SourceSelector — FPGA — DataOut — Buffer — DataIn — Processor — Processor Algorithm — DataOut — Low Freq — High Freq — Testbench Output — Vector Scope — Detected Signal (LEDs) — Low Freq — High Freq
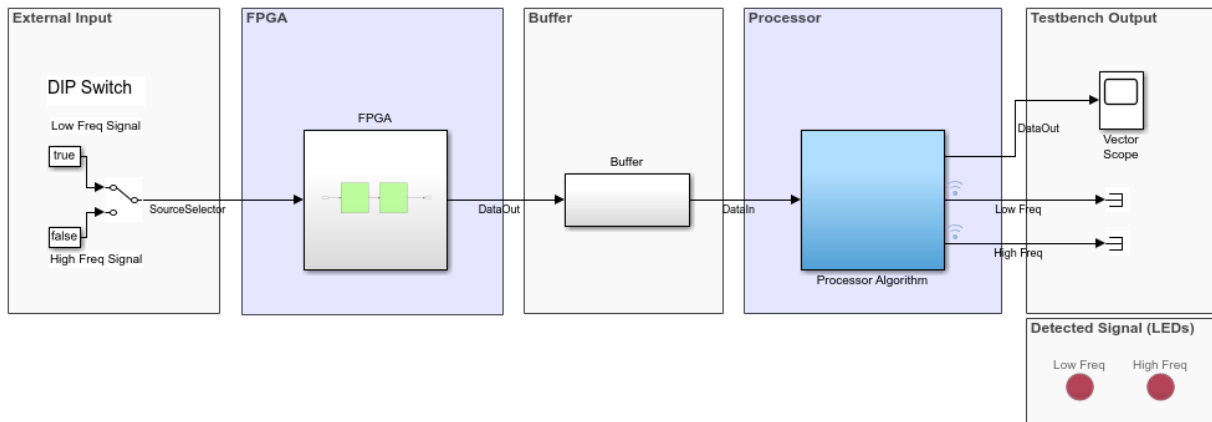
Copyright 2019 The MathWorks, Inc.

The application has following performance requirements:

- Throughput: 10e6 samples per second
- Maximum latency: 100ms
- Samples dropped: < 1 in 10000

Challenges in Designing Datapath

The FPGA processes data sample by sample while the processor operates on a frame of data at a time. The data is transferred asynchronously between FPGA and processor, and the duration of software task can vary for each execution. Therefore, a queue is needed to hold the data between FPGA and processor to prevent data loss. This queue is implemented in two stages, one as a FIFO of bursts of data samples in FPGA memory and other as a series of frame buffers in external memory. You will need to set three parameters related to the queue: frame size (number of samples in a frame of data), number of frame buffers and FIFO size (number of bursts of samples in FIFO).
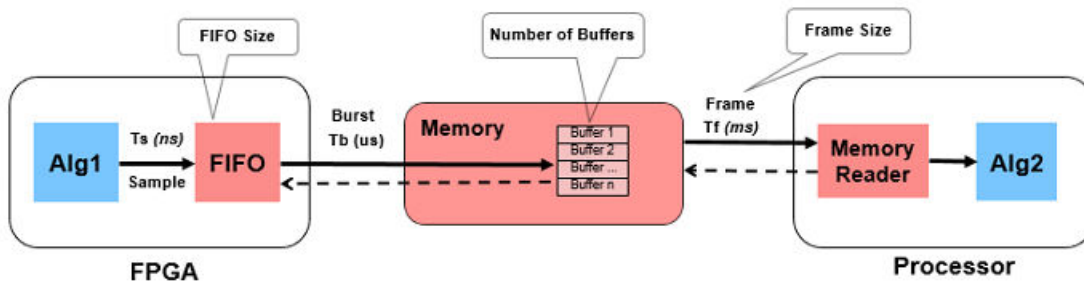
Figure 1

These design parameters affect performance and resource utilization. Increasing the frame size allows more time for software task execution and to meet throughput requirements at the cost of increasing latency. Typically, you set these parameters only when you are ready to implement on hardware, which presents the following challenges:

- It is difficult to debug issues like dropping of samples in hardware due to lack of visibility.
- It is difficult to design your application efficiently without first evaluating the effects of hardware interfaces. It can take many design-implementation iterations as you can assess performance only via implementation on hardware.
- It is difficult to optimize design since performance and cause-effect relationships are difficult to determine through implementation.

Ideally you want to account for these hardware effects while you are developing the application at design time, before implementing and running on hardware. One way to satisfy these requirements is to simulate the hardware effects, at design time. If you can simulate the variation in task durations, utilization of memory buffers/FIFOs and external memory transfer latencies, you can evaluate their effects on application design and implement the proven design on hardware. SoC Blockset allows you to simulate these effects so you can evaluate the performance of the deployed application before running on hardware.

Design Using SoC Blockset

Create an SoC model soc_hwsw_stream_top from the implementation model soc_hwsw_stream_implementation using the "Stream from FPGA to Processor Template" on page 1-47. The top model includes FPGA model soc_hwsw_stream_fpga and processor

model soc_hwsw_stream_proc instantiated as model references. The top model also includes Memory Channel and Memory Controller blocks which model shared external memory between FPGA and processor. These were earlier modeled using buffer block in the implementation model. To improve simulation performance, FPGA algorithm is also modeled for Frame-based processing soc_hwsw_stream_fpga_frame and is included as model variant subsystem at the top level. You can select the model to run in Frame-based or Sample-based processing by selecting from the mask of FPGA subsystem.

Streaming Data from Hardware to Software
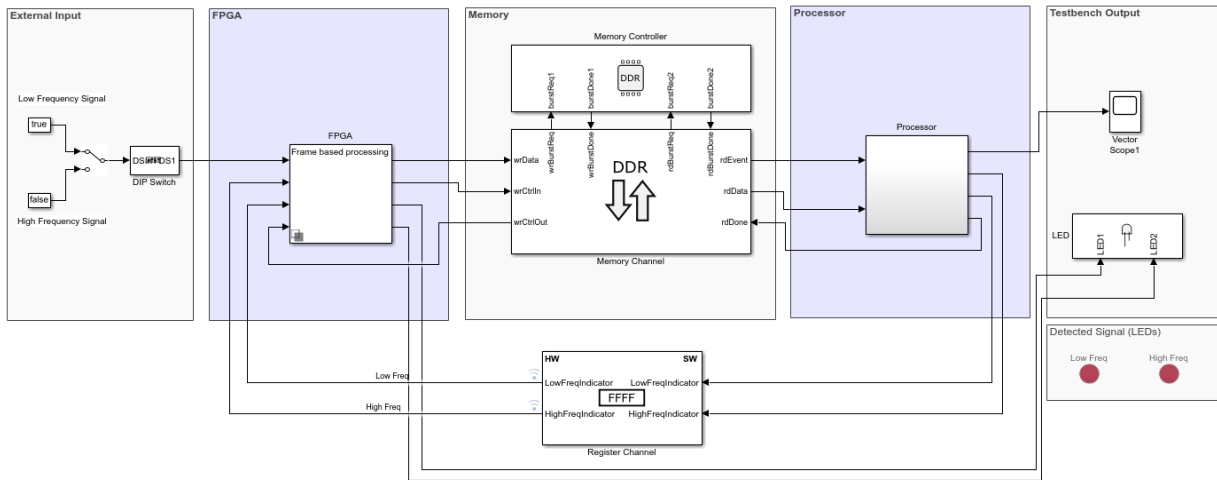
Copyright 2019 The MathWorks, Inc.

Design to Meet Latency Requirement : Latency in the datapath from FPGA to processor comprises of the latency through the FPGA logic and the time for data transfer from FPGA to processor through memory channel. In this example, the FPGA clock is 10MHz and the latency is on the order of nanoseconds. This is negligible in comparison with latency within the memory channel, which is on the order of milliseconds. Therefore, let us focus on designing for latency for data transfer in the following manner.

Begin with a few potential frame sizes and calculate Frame period for each frame size in Table -1. Frame period is the time between two consecutive frames from FPGA to processor. For this example, FPGA output sample time is 10e-6 as a valid data is output every 100 clock cycles from the FPGA.

$$FramePeriod = Framesize * FPGAOutputSampleTime$$

Latency of the memory channel is due to time elapsed by samples in the queue of frame buffers and FPGA FIFO. Let us size FPGA FIFO equivalent to one frame buffer. To stay within the maximum latency requirement, calculate the number of frame buffers for each frame size as per:

$$(NumFrameBuffers + 1) * FramePeriod <= MaxLatency$$

Maximum latency allowed for this example is 100 ms. Since the number of buffers account for maximum latency requirement, for all the cases in Table -1, latency requirement is met. A minimum of 3 frame buffers is needed in external memory for data transfer. While one of the frame buffers is written by FPGA, the other frame buffer is read by processor. Therefore, Case #8-10 from the table below are rejected as they violate the minimum buffer requirement.

#	Frame Size	Frame period (ms)	Number of buffers	Meets or Violates requirements
1	5	0.05	1999	
2	100	1	99	
3	800	8	11	
4	1000	10	9	
5	1600	16	5	
6	2000	20	4	
7	2400	24	3	
8	8000	80	<1	Violates min buffers req
9	18000	180	<1	Violates min buffers req
10	30000	300	<1	Violates min buffers req

Table -1

To visualize the latency, simulate the model and open *Memory Channel* block, go to *Performance* tab and click on *View performance plots* . Select all the latency options under *Plot Controls* and click *Create Plot* . As captured in Figure - 2, you will notice that the composite latency meets the < 100 ms requirement.
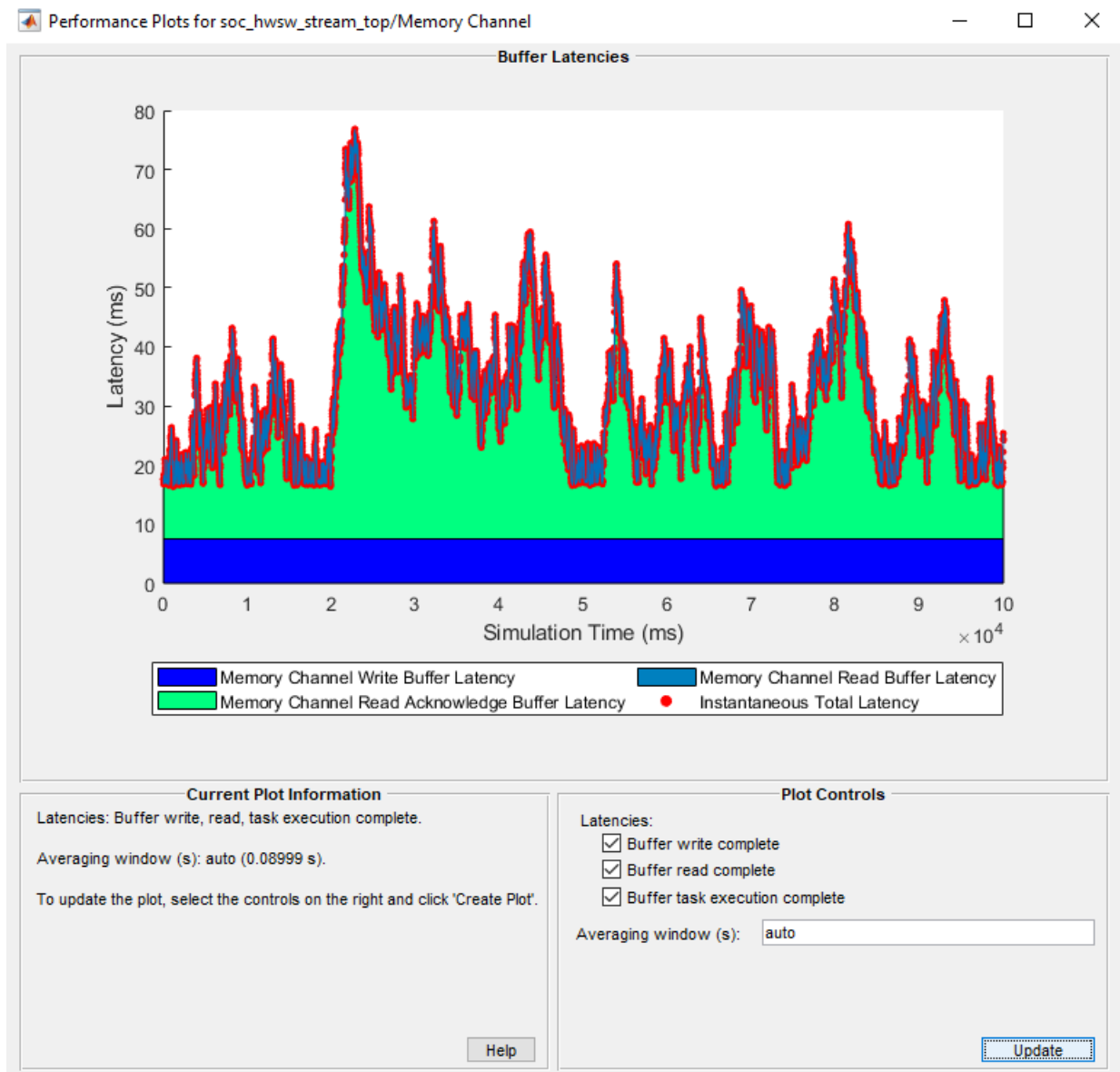
Figure - 2

Design to Meet Throughput Requirement : On average, the software tasks processing must complete within a frame period, as otherwise, task will overrun leading to dropping of data and violate the throughput requirement. i.e.

$$FramePeriod > MeanTaskDuration$$

There are various ways of obtaining mean tasks durations corresponding to frame sizes for your algorithm, which are covered in "Task Execution" on page 5-74 Example. Mean task durations for various frame sizes are captured in Table 2.

#	Frame Size	Frame period (ms)	Number of buffers	Mean Task Duration (ms)	Meets or Violates requirements
1	5	0.05	1999	0.059	Violates throughput
2	100	1	99	1.06	Violates throughput
3	800	8	11	7.858	
4	1000	10	9	9.61	
5	1600	16	5	15.3	
6	2000	20	4	19.067	
7	2400	24	3	22.812	
8	8000	80	<1	76.56	Violates min buffers req
9	18000	180	<1	175.23	Violates min buffers req
10	30000	300	<1	289.52	Violates min buffers req

Table -2

To simulate the model with the parameters corresponding to rows (#2-#7) in the table use the function `set_hwsw_stream_set_parameters` function with row # as an argument. Set the model parameters for row # 2 as below:

```
>> soc_hwsw_stream_set_parameters(2); % row # 2
```

Since the Mean Task Duration of 1.06 ms is less than Frame Period of 1.0 ms, the data is dropped in the memory channel. Open Logic Analyzer and notice that signal *icFIFODroppedCount* is increasing throughout the simulation as captured in Figure 3, indicating accumulating amount of dropped data.
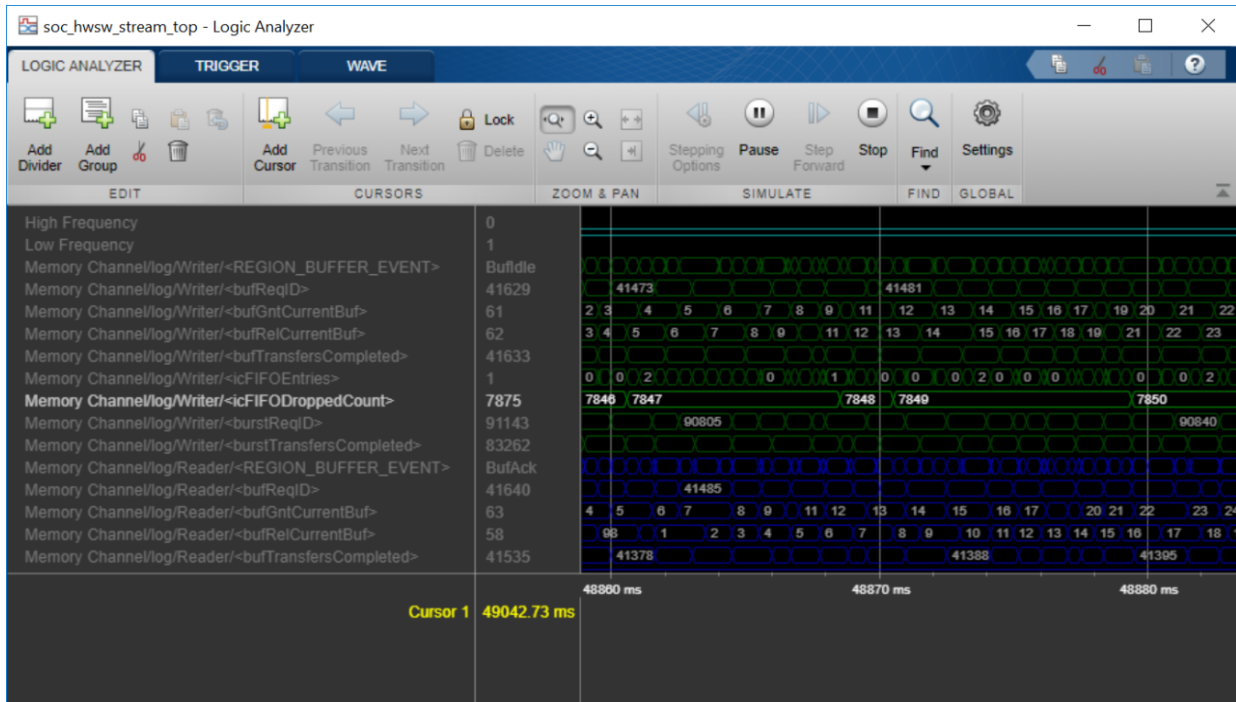
Figure - 3

Since data is dropped during transfer from FPGA to processor through memory, this is reflected as a drop in throughput. Open *Memory controller* block, go to *Performance* tab and click on *Plot data throughput* button under Performance tab to see the memory throughput plot as in figure 4. Note that the throughput is less than the required 0.4 MBps. Since the FPGA output data sample time is 10e-6 and each sample is 4 bytes wide, the required streaming throughput for the system is 4 bytes/10e-6 = 400 KBps.
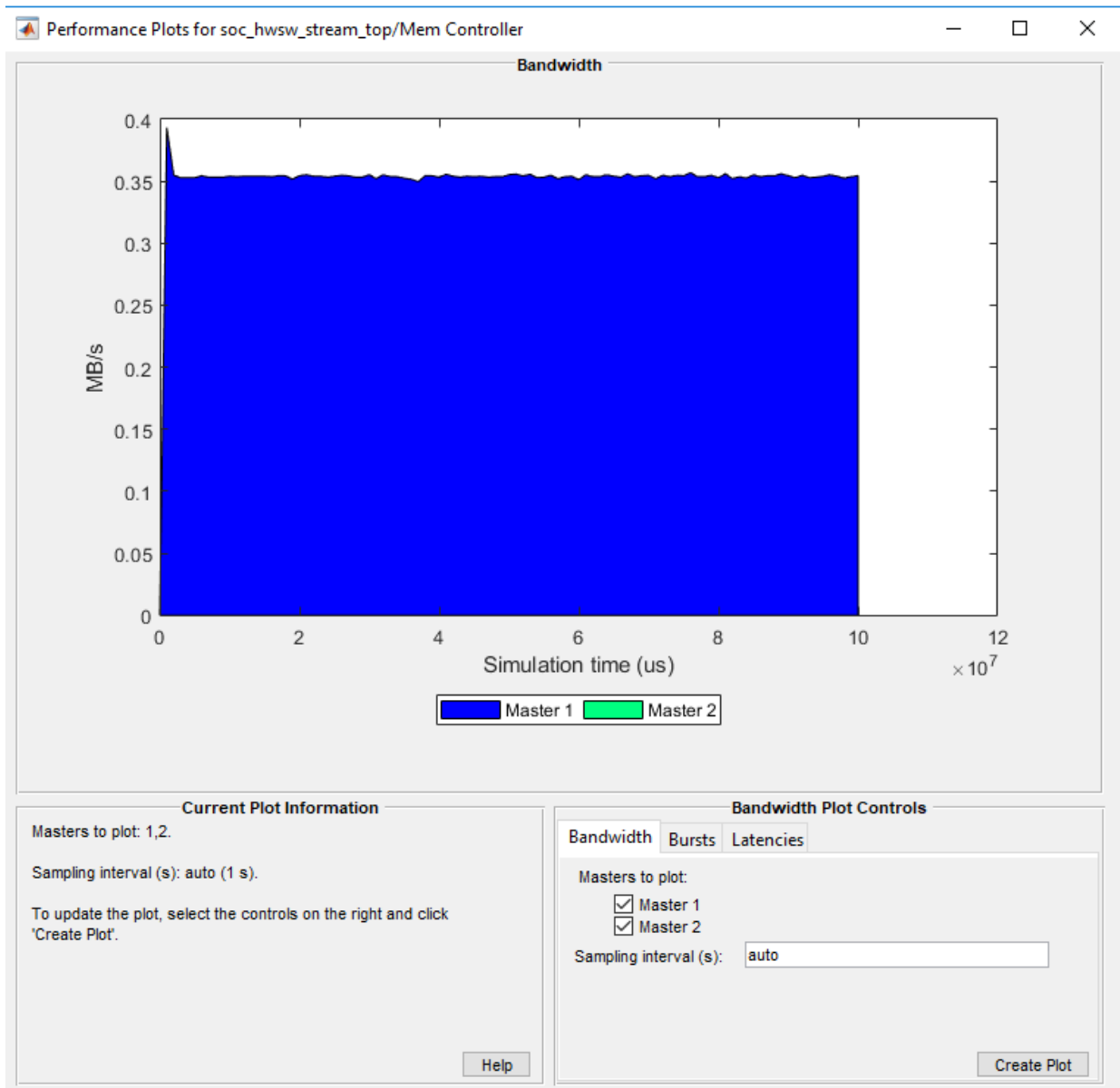
Figure - 4

Design to Meet Drop Samples Requirement : Since the task durations can vary for many reasons like different code execution paths and variation in OS switching time, it is possible that data is dropped in the memory channel. Specify the mean task execution duration and statistical distribution for task durations in the mask of Task Manager block. Size the FIFO equivalent to one frame buffer. Set the FIFO burst size to 16 Bytes and calculate the FIFO depth:

$$FIFO_{depth} = FrameSize/FIFOBurstSize$$

Now, simulate the model for 100 sec (10e6 samples at 10e-6 samples per second) for cases # 3-7. Open the Logic analyzer and note the number of samples dropped on signal *icFIFODroppedCount*.

```
>> soc_hwsw_stream_set_parameters(3); % set the model parameters for #3
```

Open Simulation Data Inspector and add signals from memory channel as shown in Figure 5 below. Note that as buffers usage (signal *buffAvail*) increase to the maximum 11, the FIFO usage (signal *isFIFOEntries*) begin to increase. When FIFO is completely used, the samples get dropped (signal *isFIFODroppedCount*)
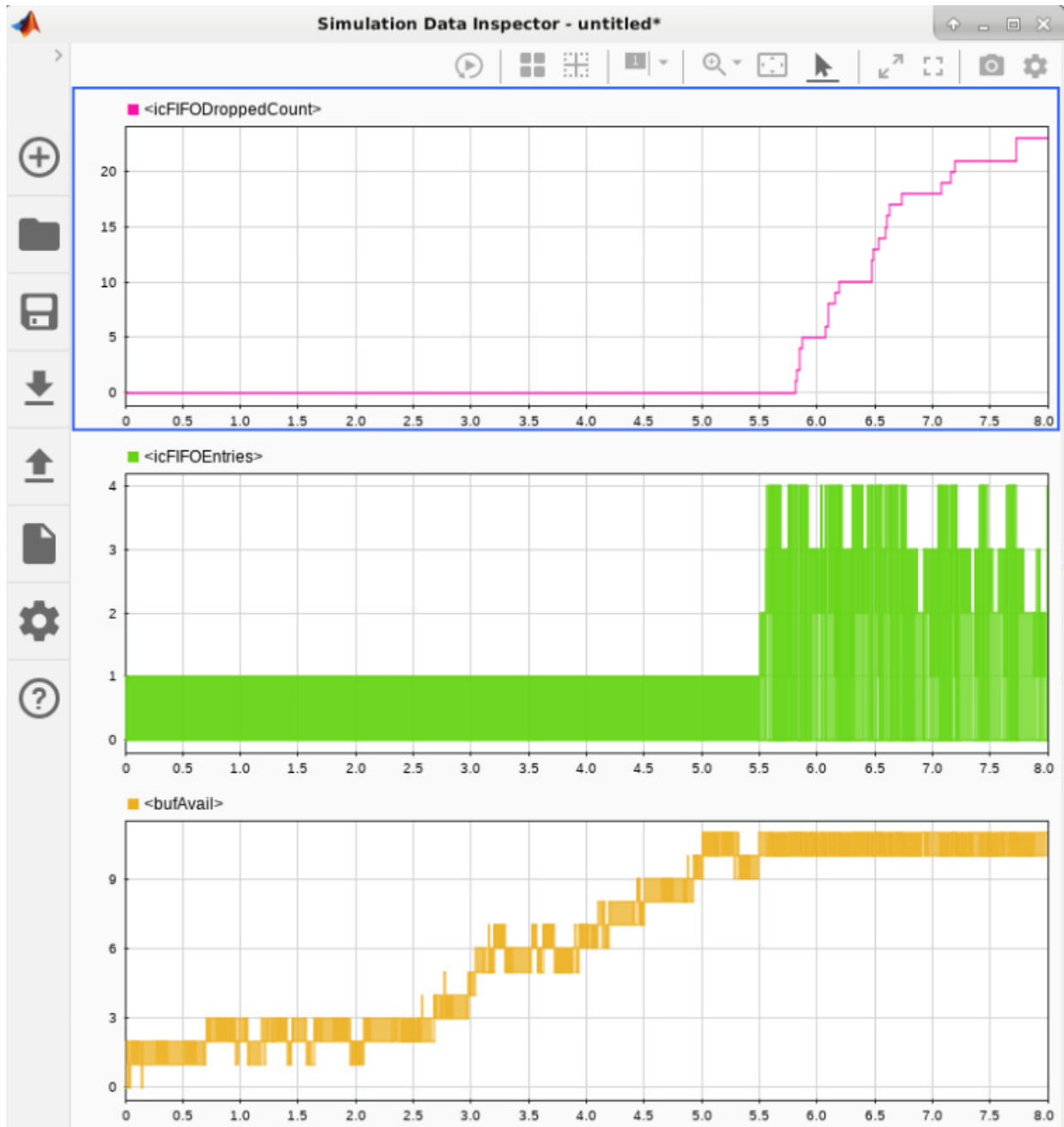
Figure - 5

The results of simulation for all the cases #3-7 and resultant sample dropped per 10000 are tabulated in Table 3.

#	Frame Size	Frame period (ms)	Number of buffers	Mean Task Duration (ms)	Avg Samples dropped per 10000	Meets or Violates requirements
1	5	0.05	1999	0.059		Violates throughput
2	100	1	99	1.06		Violates throughput
3	800	8	11	7.858	172.6	Violates drop samples
4	1000	10	9	9.61	0	Meets all requirements
5	1600	16	5	15.3	1	Meets all requirements
6	2000	20	4	19.067	2.25	Violates drop samples
7	2400	24	3	22.812	3.9	Violates drop samples
8	8000	80	<1	76.56		Violates min buffers req
9	18000	180	<1	175.23		Violates min buffers req
10	30000	300	<1	289.52		Violates min buffers req

Table - 3

The highlighted entries (rows #4 and #5) are valid design choices since they meet throughput, latency and drop samples requirement.

Implement and Run on Hardware

Following products are required for this section:

- HDL Coder™
- Embedded Coder®
- "SoC Blockset Supported Hardware" on page 6-2, or
- "SoC Blockset Supported Hardware" on page 6-2

To implement the model on a supported SoC board use the SoC Builder tool. Open the mask of 'FPGA' subsystem and select model variant to 'Sample based processing'. By default, the model will be implemented on **Xilinx® Zynq® ZC706 evaluation kit** as it is configured with that board. Open SoC Builder from the Tools menu and follow these steps:

- Select 'Build Model' on 'Setup' screen. Click 'Next'.
- Click 'View/Edit Memory Map' to view the memory map on 'Review Memory Map' screen. Click 'Next'.
- Specify project folder on 'Select Project Folder' screen. Click 'Next'.
- Select 'Build, load and run' on 'Select Build Action' screen. Click 'Next'.
- Click 'Validate' to check the compatibility of model for implementation on 'Validate Model' screen. Click 'Next'.
- Click 'Build' to begin building of the model on 'Build Model' screen. An external shell will open when FPGA synthesis begins. Click 'Next'.
- Click 'Test Connection' on 'Connect Hardware' screen to test the connectivity of host computer with SoC board. Click 'Next' to go to 'Run Application' screen.

The FPGA synthesis may take more than 30 minutes to complete. To save time, you may want to use the provided pre-generated bitstream by following these steps:

- Close the external shell to terminate synthesis.
- Copy pre-generated bitstream to your project folder by running the command below and then,
- Click 'Load and Run' button to load pre-generated bitstream and run the model on SoC board

```
>> copyfile(fullfile(matlabroot,'toolbox','soc','socexamples','bitstreams','soc_hwsw_st
```

While the application is running on hardware, toggle the DIP switch on your board to change the test data from 'low' to 'high' frequency and notice the blinking of corresponding LED on the board. You can also read the samples dropped count in the model running on external mode. Thus, you verify that your implementation from SoC Blockset model matches the simulation and meets the requirements.

Implementation on other boards: To implement the model on a supported board other than Xilinx® Zynq® ZC706 evaluation kit board, you must first configure the model to the supported board and set the example parameters as below. Open **Model Configuration Parameters**, navigate to **Hardware Implementation** tab and perform the following settings:

- Select your board from the drop-down list under **Hardware board** on both top and processor model.

- Navigate to **Target hardware resources > FPGA design (top level)** tab and enable **Include MATLAB as AXI Master IP for host-based interaction** and set **IP core clock frequency (MHz)** to 10 MHz.
- Navigate to **Target hardware resources > FPGA design (debug)** tab and enable **Include AXI Interconnect monitor**.

Next, open SoC Builder and follow the steps as previously stated for Xilinx® Zynq® ZC706 above. Modify the *copyfile* command to match the bitstream corresponding to your board. Available pre-generated bitstreams are:

- 'soc_hwsw_stream_top_fpga-zc706.bit'
- 'soc_hwsw_stream_top_fpga-zedboard.bit'
- 'soc_hwsw_stream_top_fpga-zcu102.bit'
- 'soc_hwsw_stream_top_fpga-c5soc.sof'
- 'soc_hwsw_stream_top_fpga-a10soc.sof'

In summary, this example showed you a systematic approach to design the datapath between hardware logic and embedded processor using SoC Blockset. You chose design parameters of frame size, number of frame buffers and FIFO size to meet the system performance requirements of throughput, latency and drop samples. By simulating and visualizing the effects of these parameters on the complete model containing hardware logic, processor algorithms, external memory and processor task durations, you uncovered issues like loss of throughput, latency and dropping of samples before implementing on hardware. This workflow ensures that the design works on hardware before implementation and avoids long design-implementation iterations.

Analyze Memory Bandwidth Using Traffic Generators

This example demonstrates how to analyze memory bandwidth for an SoC application. In memory intensive hardware designs, you may have multiple masters accessing a common DDR memory. In such cases, it is important to analyze dynamic requirement of all memory masters to guide algorithm design and hardware board requirement for deployment. You will simulate the memory traffic using Memory traffic generators and analyze the bandwidth usage and verify it on hardware.

Supported hardware platforms

- Xilinx Zynq® ZC706 evaluation kit
- Xilinx® Kintex® 7 KC705 development board

Design Task

Consider an application performing HD video processing in FPGA on real time input and output. This application requires four memory consumers vying for DDR access simultaneously. Memory master 1 writes incoming video frames to memory and Memory master 4 reads video frames out of memory and connect to output display. Memory master 2 reads the data from memory for processing in FPGA and memory master 3 writes the data back to memory.
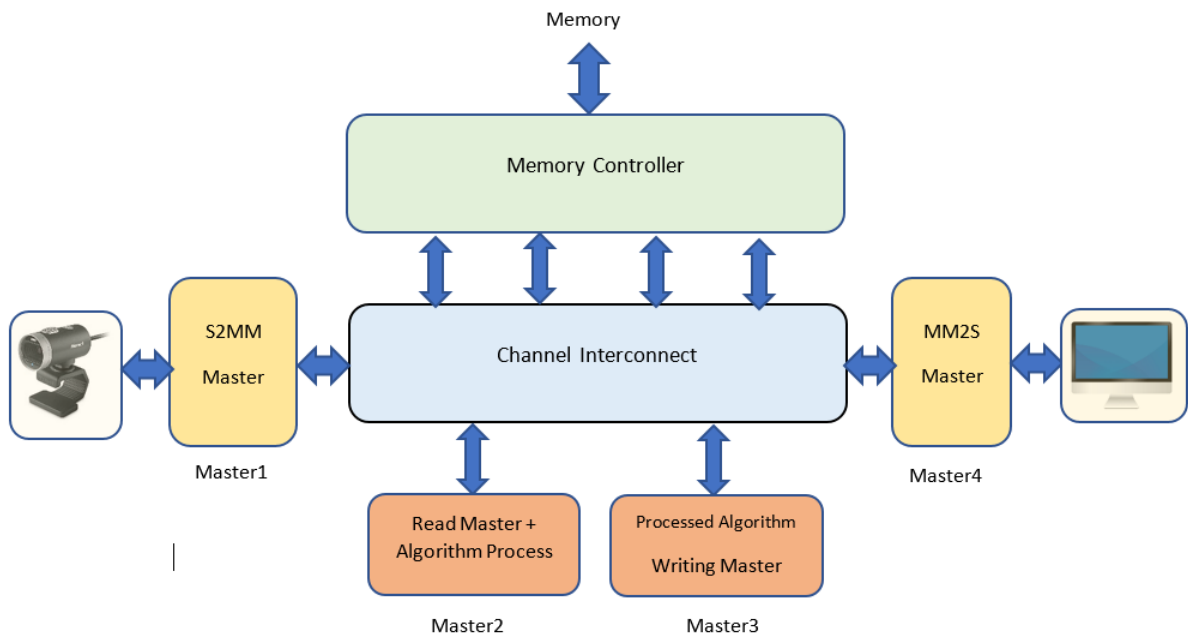
Each master operates on HD video with following characteristics:

- Frame size: 1920x1080p
- Pixel size: 2 Bytes (YCBCr format)
- Frame period: 1/60 = 16.67ms (for 60 FPS)
- Frame data: 1920x1080x2 = 4.1472MB

Assume the memory controller characteristics are as follows:

- Clock Frequency: 200 MHz
- Data width: 32 bits
- Burst Transaction length: 128

Design Using SoC Blockset

Create a model using Memory Controller and Memory Traffic Generator blocks to model four memory masters.

Memory Controller: Set the memory controller parameters in *Configuration Parameters > Hardware Implementation > Target Hardware Resources* . Under *FPGA Design (Mem Controllers*) tab, set the clock frequency to 200 MHz and data width to 32. Under *FPGA Design (debug)* tab, select 'Include AXI Interconnect Monitor'.

Analyze Memory Bandwidth for SoC Design Using Traffic Generators
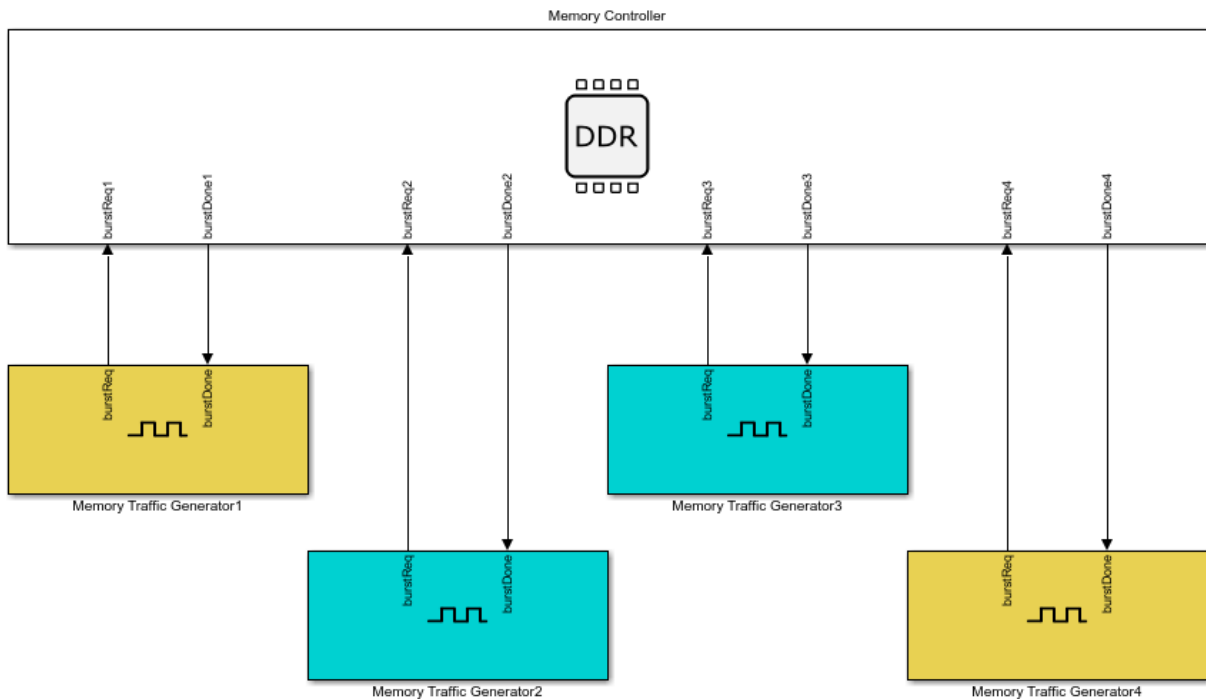
Copyright 2019 The MathWorks, Inc.

Memory Traffic Generators 1 & 4: Memory traffic characteristics for Master 1 and 4 are same as they represent streaming of video frames to and from memory. Set the memory traffic characteristics for masters 1 and 4 as follows:

- Burst size (in bytes): Burst transaction length * (Data width/8) = 128* 32/8 = 512

- Total burst requests: 4 frames data for simulation = 4 * Traffic data/Burst size = 4*8100 = 32400

- Burst inter access time: Frame Period/Number of Burst requests = 16.67e-3/8100 = 20.58e-7 sec

- Set the First burst time & Random time between the burst = 20.58e-7 as a constant data traffic since the data is continuously received at a constant rate.

- Memory bandwidth: Frame Data/ Frame Period = 4.1476/16.67e-3 = 248.8MBps

Update the Memory Traffic Generator1 and Memory Traffic Generator4 block mask with above values. Set the request type for Memory Traffic Generator1 with writer and Memory Traffic Generator4 with reader. Clear the 'Wait for burst done' option in both the blocks as these masters represent the masters with continuous traffic viz. HDMI Camera and display.

Block Parameters: Memory Traffic Generator1 ✕

Memory Traffic Generator (mask) (link)

Generate read or write requests to memory as a dummy master. This block
can be used to model the impact that another master has on your
application without explicitly designing that master.

The block outputs memory request signals which should be connected to
the burstReq and burstDone ports of a Memory Controller block. These
signal lines do not transmit data.

The Burst length parameter is derived from the Burst size and Controller
Data Width specified in the Hardware Implementation Pane.

In generated code, this block outputs uniformly distributed traffic request
signals. In simulation, when 'Allow simulation-only parameters' is selected,
this block provides more control than the generated code. Set the Random
time between bursts parameter to specify the min and max values between
bursts.

Parameters

Request type	Writer ▾
Total burst requests	32400
Burst size (bytes)	512

Datapath width (bits):   32
Burst length (beats):   128

☑ Allow simulation-only parameters

Simulation-only parameters

First burst time:	20.58e-7
Random time between bursts (s):	[20.58e-7 20.58e-7]

☐ Wait for burst done

OK   Cancel   Help   Apply

5-55

Memory Traffic Generators 2 & 3: Memory Traffic Generator 2 represent reader for FPGA Algorithm and Memory Traffic Generator 3 represent writer from FPGA Algorithm.

- Burst size (in bytes): Burst transaction length * (Data width/8) = 128* 32/8 = 512
- Total burst requests: 4 * Traffic data/Burst size = 4*8100 = 32400(4 frames data for simulation)
- Burst inter access time: (Burst Length + 10)/Clock period = 6.9e-7(0.69us)
- Allowing for some randomness in times for read and write request of data due to variation in demands of algorithm, select the burst times as follows:
- First burst time: 7.2e-7
- Random time between the bursts: [7.2e-7 7.4e-7]

Block Parameters: Memory Traffic Generator2   ✕

Memory Traffic Generator (mask) (link)

Generate read or write requests to memory as a dummy master. This block can be used to model the impact that another master has on your application without explicitly designing that master.

The block outputs memory request signals which should be connected to the burstReq and burstDone ports of a Memory Controller block. These signal lines do not transmit data.

The Burst length parameter is derived from the Burst size and Controller Data Width specified in the Hardware Implementation Pane.

In generated code, this block outputs uniformly distributed traffic request signals. In simulation, when 'Allow simulation-only parameters' is selected, this block provides more control than the generated code. Set the Random time between bursts parameter to specify the min and max values between bursts.

Parameters

Request type	Reader ▼
Total burst requests	32400
Burst size (bytes)	512

Datapath width (bits):   32
Burst length (beats):   128

☑ Allow simulation-only parameters

Simulation-only parameters

First burst time:	7.2e-7
Random time between bursts (s):	[7.2e-7 7.4e-7]

☑ Wait for burst done

OK   Cancel   Help   Apply

5-57

Simulate

Run the model and after simulation is complete, open the Memory Controller block and click on *Launch Performance App* under *Performance* tab. Select all the masters under Bandwidth tab and click Create Plot. You will notice that all masters roughly achieved a bandwidth of 190 MBps and did not meet the required 248 MBps. It is also observed by the warnings in the diagnostic viewer.
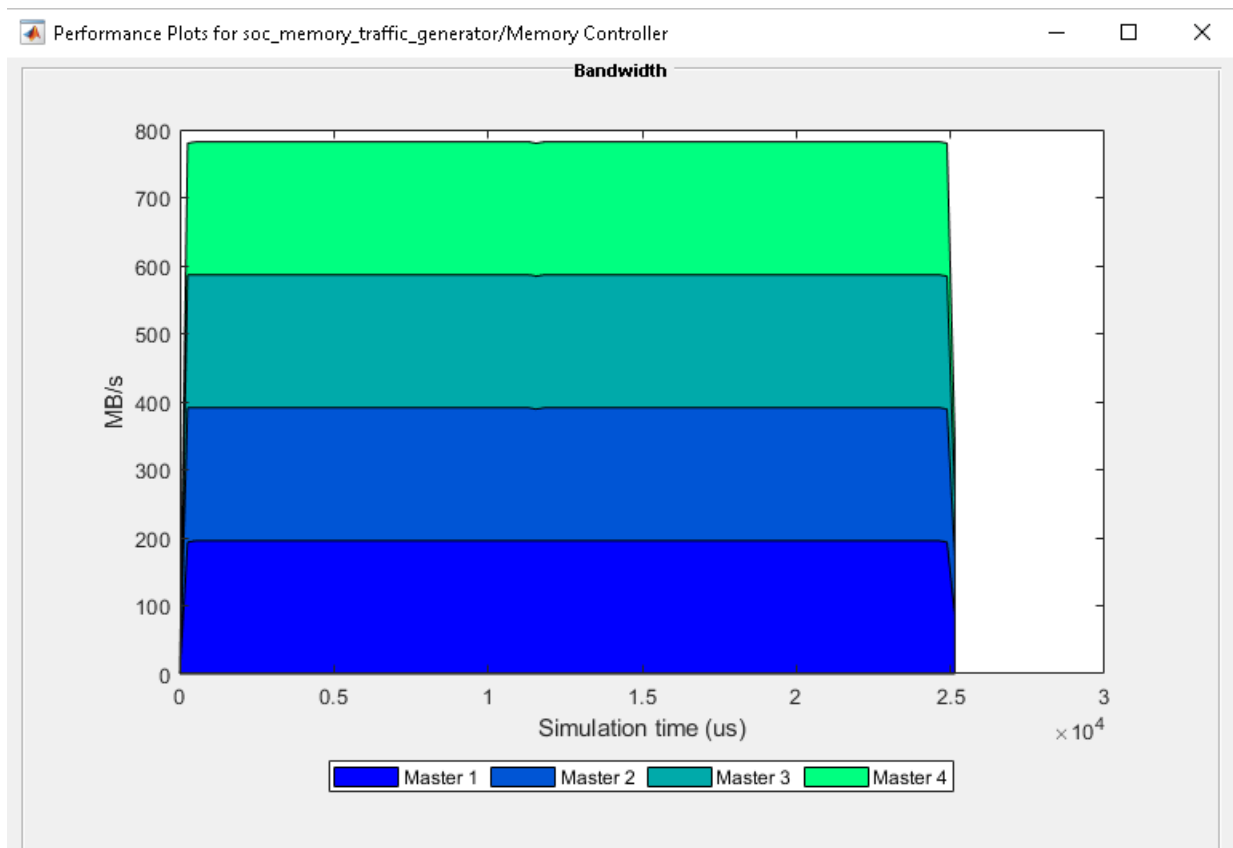
To meet the required bandwidth, modify the data width of controller from 32 to 64 in configuration parameter settings under Target Hardware Resources. This will require changing the Memory Traffic Generator settings accordingly as follows:

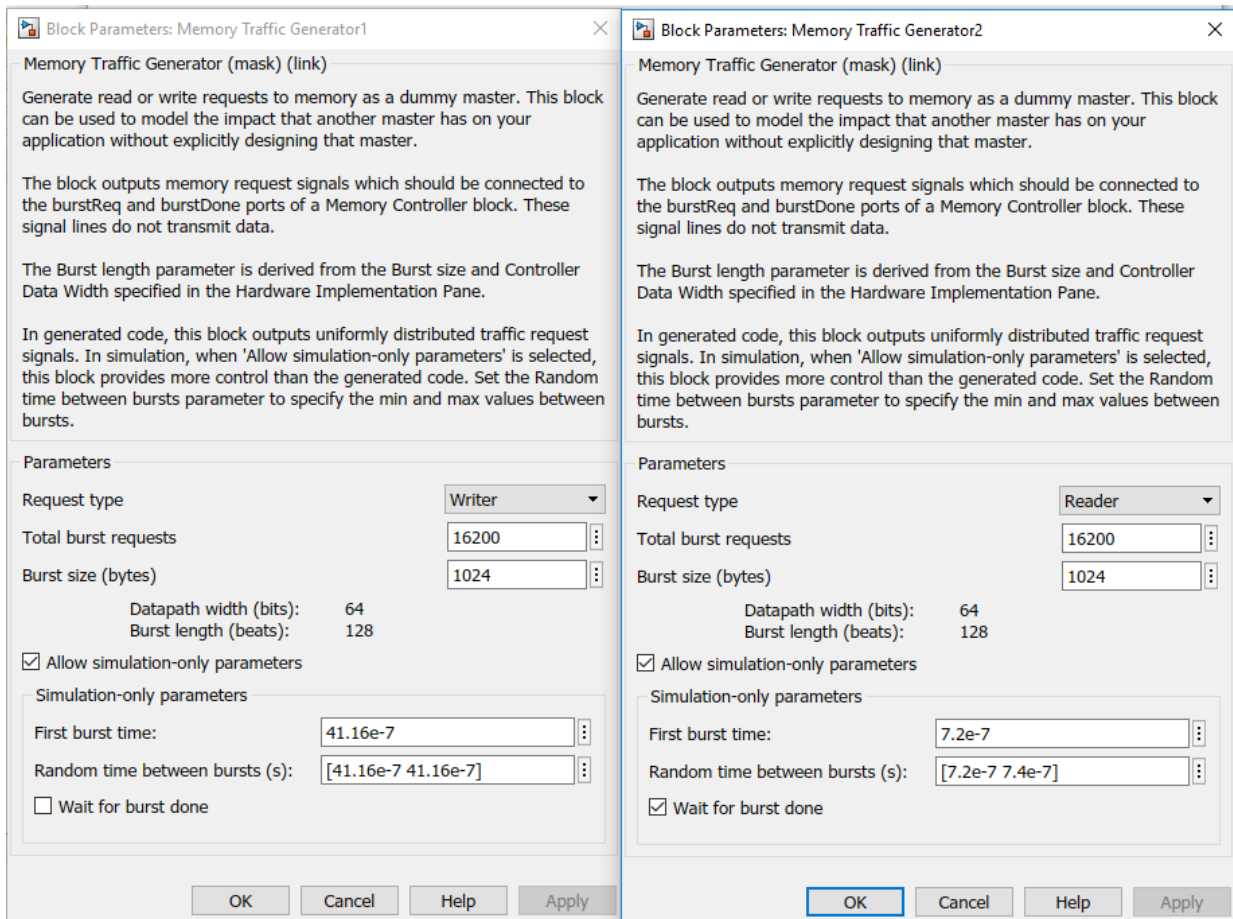- Burst size (in bytes): Burst transaction length * (Data width/8) = 128* 64/8 = 1024

- Total burst requests: 4 * Traffic data/Burst size = 4*4050 = 16200 (4 frames of data for simulation)
- Burst inter acces time for Memory Traffic Generators 1 & 4: Frame Period/Number of Burst requests = 16.67e-3/4050 = 41.16e-7 sec
- No change in Burst Times for Memory Traffic Generator 2 and 3 since they are determined based on algorithm needs.

The new parameter settings are as follows:

Block Parameters: Memory Traffic Generator1	✕

Memory Traffic Generator (mask) (link)

Generate read or write requests to memory as a dummy master. This block can be used to model the impact that another master has on your application without explicitly designing that master.

The block outputs memory request signals which should be connected to the burstReq and burstDone ports of a Memory Controller block. These signal lines do not transmit data.

The Burst length parameter is derived from the Burst size and Controller Data Width specified in the Hardware Implementation Pane.

In generated code, this block outputs uniformly distributed traffic request signals. In simulation, when 'Allow simulation-only parameters' is selected, this block provides more control than the generated code. Set the Random time between bursts parameter to specify the min and max values between bursts.

Parameters

Request type	Writer
Total burst requests	16200
Burst size (bytes)	1024

Datapath width (bits): 64
Burst length (beats): 128

☑ Allow simulation-only parameters

Simulation-only parameters

First burst time:	41.16e-7
Random time between bursts (s):	[41.16e-7 41.16e-7]

☐ Wait for burst done

OK   Cancel   Help   Apply

Block Parameters: Memory Traffic Generator2	✕

Memory Traffic Generator (mask) (link)

Generate read or write requests to memory as a dummy master. This block can be used to model the impact that another master has on your application without explicitly designing that master.

The block outputs memory request signals which should be connected to the burstReq and burstDone ports of a Memory Controller block. These signal lines do not transmit data.

The Burst length parameter is derived from the Burst size and Controller Data Width specified in the Hardware Implementation Pane.

In generated code, this block outputs uniformly distributed traffic request signals. In simulation, when 'Allow simulation-only parameters' is selected, this block provides more control than the generated code. Set the Random time between bursts parameter to specify the min and max values between bursts.

Parameters

Request type	Reader
Total burst requests	16200
Burst size (bytes)	1024

Datapath width (bits): 64
Burst length (beats): 128

☑ Allow simulation-only parameters

Simulation-only parameters

First burst time:	7.2e-7
Random time between bursts (s):	[7.2e-7 7.4e-7]

☑ Wait for burst done

OK   Cancel   Help   Apply

Simulate the model and open the Bandwidth plot from Memory Controller as in previous case. Notice that Memory bandwidth achieved by Memory Traffic Generator 1 and 4 is 248 MBps. The memory bandwidth from Generator 2 and 3 is around 500 MBps. This meets the design requirement as all the masters are able to meet the real-time requirement of 248 MHz. Observe that there are no warnings on the diagnostic viewer as burst requests are not dropped.
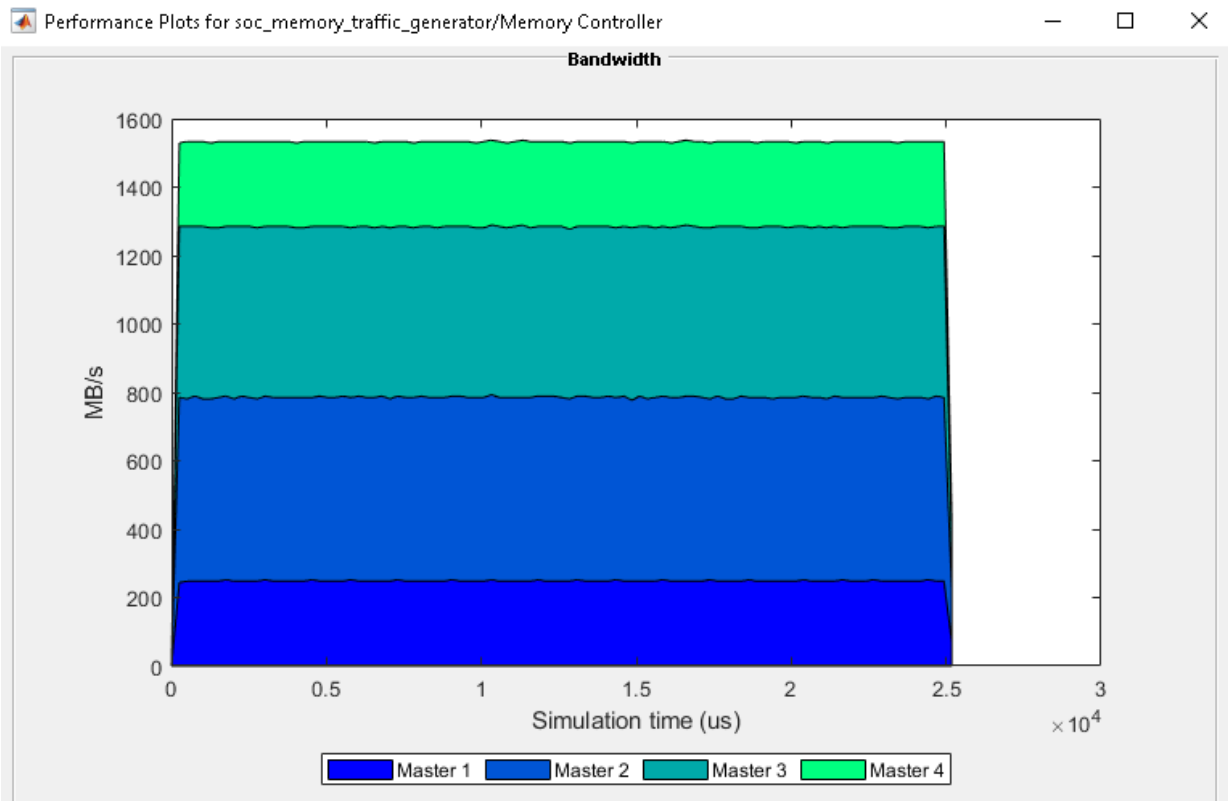
Implement and Run on Hardware

"SoC Blockset Supported Hardware" on page 6-2 is required for this section.

To implement the model on a supported FPGA board, use the SoC Builder application. By default, the model will be implemented on **Xilinx® Zynq® ZC706 evaluation kit** as it is configured with that board.

AXI Traffic Generator(ATG), the hardware IP Core for Memory Traffic Generator block do not support random burst inter access times and it differentiates Reader and Writer masters in arbitration policy unlike the Memory Traffic Generator block for simulation. Therefore before implementing on hardware, modify the Memory block settings as follows:

- Make all the Memory Traffic Generators as 'Writers'
- For Memory Traffic Generator 2 and 3, set [7.2e-7 7.2 e-7] for Random time between burst to make it fixed inter burst time of 7.2e-7

Open SoC Builder from the Tools menu and follow these steps:

- Select 'Build Model' on 'Setup' screen. Click 'Next'.
- Click 'View/Edit Memory Map' to view the memory map on 'Review Memory Map' screen. Click 'Next'.
- Specify project folder on 'Select Project Folder' screen. Click 'Next'.
- Select 'Build, load and run' on 'Select Build Action' screen. Click 'Next'.
- Click 'Validate' to check the compatibility of model for implementation on 'Validate Model' screen. Click 'Next'.
- Click 'Build' to begin building of the model on 'Build Model' screen. An external shell will open when FPGA synthesis begins. Click 'Next' to 'Load Bitstream' screen.

The FPGA synthesis may take more than 30 minutes to complete. To save time, you may want to use the provided pre-generated bitstream by following these steps:

- Close the external shell to terminate synthesis.
- Copy pre-generated bitstream to your project folder by running the command below and then,
- Click 'Load' button to load pre-generated bitstream.

```
>> copyfile(fullfile(matlabroot,'toolbox','soc','socexamples','bitstreams','soc_memory_
```

To run this example, copy the example test bench to your project folder.

```
>> copyfile(fullfile(matlabroot,'toolbox','soc','socexamples','soc_memory_traffic_gener
```

The testbench configures the generated hardware ATG IP cores for Memory Traffic Generators. To run on hardware, we will increase the number of burst requests by 100 times since we are using *MATLAB® as AXI Master* IP to get the samples back to MATLAB® which involves substantial delay in accessing hardware. Load

soc_memory_traffic_generator_zc706_aximaster.mat file and increase the number of burst requests for all the masters in ATG configuration to 100 times. Save the mat file requests in ATG configuration.

Enter the following command to run the test bench soc_memory_traffic_generator_aximaster.

```
>> soc_memory_traffic_generator_aximaster
```

The following output is produced showing the memory traffic after running the test bench. All masters passing the bandwidth requirements.
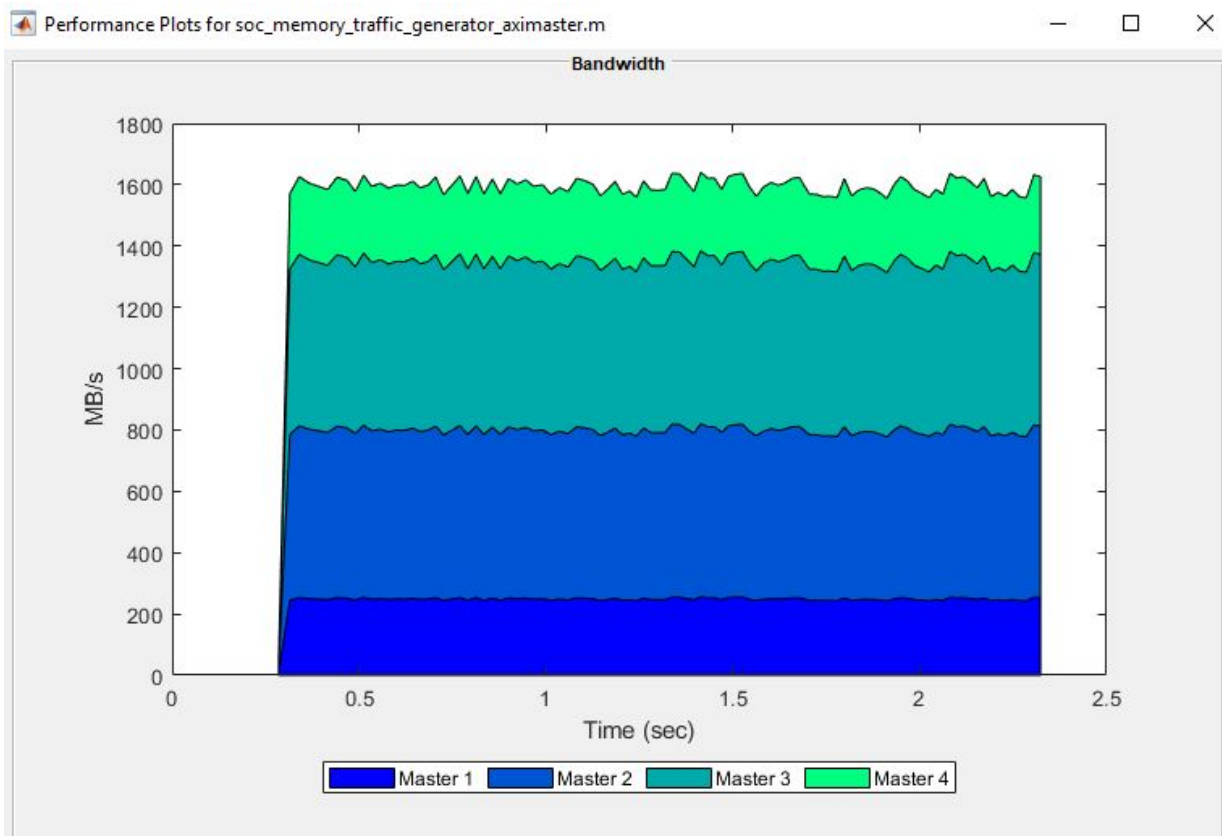
Implementation on Xilinx® Kintex® 7 KC705 development board: To implement the model on KC705 development board, you must first configure the model to Xilinx®

Kintex® 7 KC705 development board and set the following example parameters. Open **Model Configuration Parameters**, navigate to **Hardware Implementation** tab and perform the following:

- Select **Xilinx® Kintex® 7 KC705 development board** from the drop-down list under **Hardware board**
- Navigate to **Target hardware resources > FPGA design (top level)** tab and enable **Include MATLAB as AXI Master IP for host-based interaction** and enable **Include processing system**.
- Navigate to **Target hardware resources > FPGA design (mem controller)** tab and set 'Controller data width (bits)' to 64.

Next, open SoC Builder and follow the steps as previously stated for Xilinx® Zynq® ZC706 above. Modify the copyfile command to match Kintex® 7 KC705 development board bitstream 'soc_memory_traffic_generator_fpga-kc705.bit'.

In summary, you simulated the memory traffic for a prospective design before designing the algorithms. You analyzed memory bandwidth and modified memory parameters to meet the design requirement. You verified the results on hardware.

Record I/O Data from SoC Device

This example shows you how to record real-world data from hardware for use in simulation.

Supported hardware platforms:

- Xilinx® Zynq® ZC706 evaluation kit
- Xilinx Zynq UltraScale™+ MPSoC ZCU102 Evaluation Kit
- ZedBoard™ Zynq-7000 Development Board
- Altera® Cyclone® V SoC development kit
- Altera Arria® 10 SoC development kit

In many situations you may want to verify your algorithm against real-world data. This example, using the Streaming Data from Hardware to Software model, shows how to record signals from the AXI4 interface on a SoC device. This workflow allows you to focus on the processor side of the algorithm by substituting a pre-recorded data stream in place of the Simulink® FPGA design.

We recommend completing "Streaming Data from Hardware to Software" on page 5-37 example.

Streaming Data from Hardware to Software

Copyright 2019 The MathWorks, Inc.
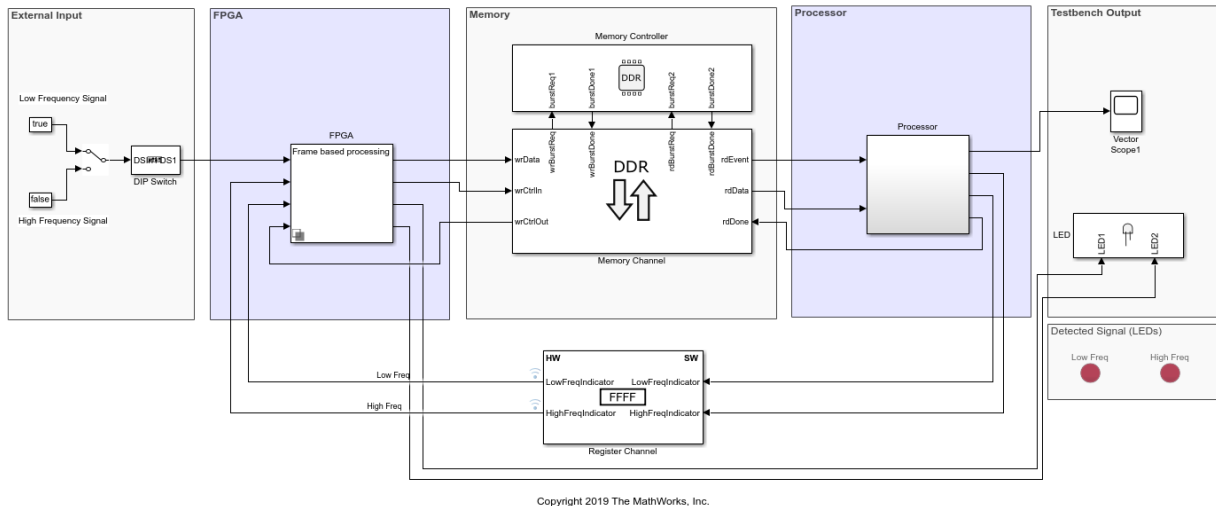
Record Data from FPGA

In this section, you will record data generated by the FPGA subsystem in the Streaming Data from Hardware to Software model. In this model, the FPGA subsystem generates a sinusoidal signal with frequency 1kHz or 10kHz, controlled via a DIP switch (DS1). The FPGA algorithm filters the signal and sends it to the processor through AXI4 Stream Memory Channel.
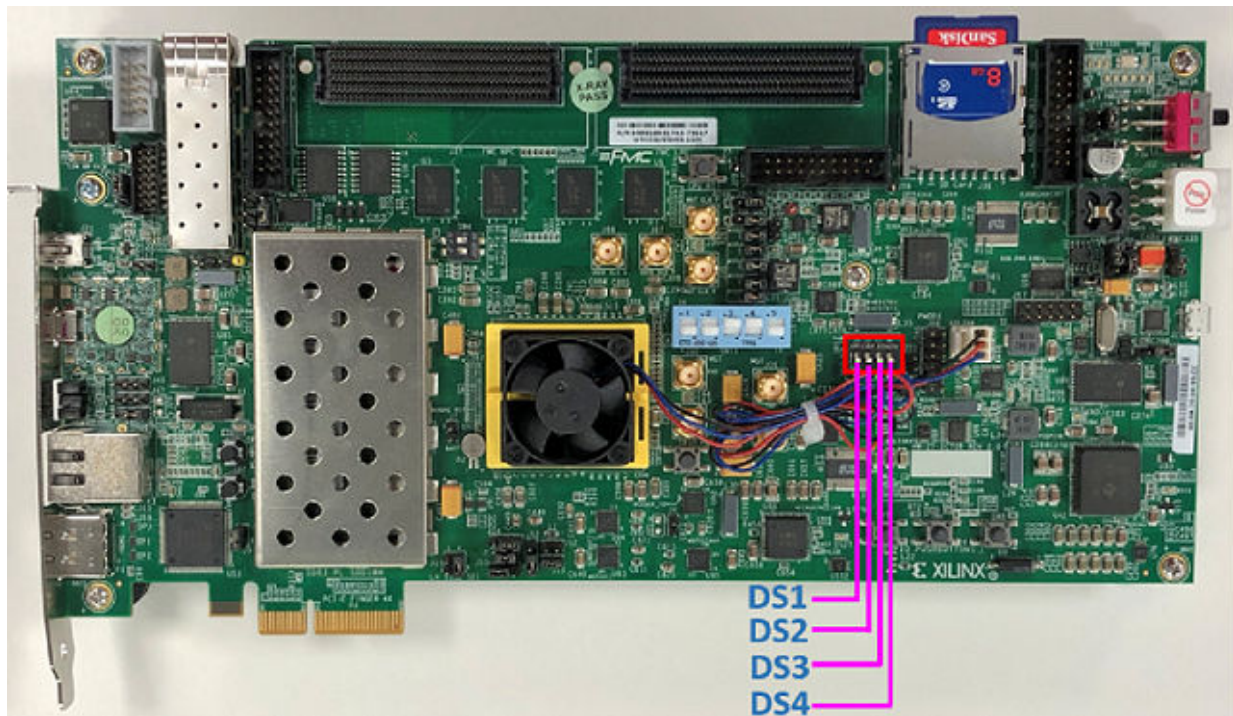
Follow the steps below to record data from FPGA:

1. Create a hardware communication object executing the following on the MATLAB® command prompt.

```
hw = socHardwareBoard('Xilinx Zynq ZC706 evaluation kit','hostname','10.10.10.15','user
```

Enter the appropriate hardware board name, IP address and the user credentials in the command above. The hardware object hw, is a communication gateway that provides control commands and I/O exchange.

2. Open Streaming Data from Hardware to Software model. Load the provided pre-generated FPGA bitstream for this model to hardware.

```
socLoadExampleBitstream(hw,'soc_hwsw_stream_top')
```

3. Create a data recorder for your hardware board.

```
dr = soc.recorder(hw);
```

4. Create an **AXI Stream Read** input source object and configure the source properties.

```
src = soc.iosource(hw,'AXI Stream Read');
src.devName = 'mwfpga_algorithm_wrapper_ip0:s2mm0';
samplingFrequency = 1e5;
src.dataTypeStr = 'uint32';
src.SamplesPerFrame = 1000;
src.SampleTime = src.SamplesPerFrame/samplingFrequency;
```

The `samplingFrequency` represents the sine wave sampling rate in the Streaming Data from Hardware to Software model.

5. Add the **AXI Stream Read** source to the data recording session.

```
addSource(dr,src,'AXI4 stream interface')
```

6. Initialize the I/O sources on the hardware board for recording.

```
setup(dr)
```

7. Use the `record` function to record 10 seconds of data.

```
record(dr, 10)
while isRecording(dr)
    pause (0.1);
end
```

During the recording, toggle the DIP switch (DS1) to change the frequency of signal generated by the FPGA.

8. Save the recorded data to a file:

```
save(dr,'sine_wave_data')
```

Record RF Signals

In this section, you will capture RF signals from an AD - FMCOMMS2/3/4 RF card connected to the FPGA. The data will be streamed from the RF card to the processor using AXI4 stream interface.

Following products are required for this section:

• SoC Blockset Support Package for Xilinx® Devices

Supported hardware platforms for this section are:

- Xilinx® Zynq® ZC706 evaluation kit
- ZedBoard™ Zynq-7000 Development Board

To configure RF card refer to "Manual Host-Radio Hardware Setup" (Communications Toolbox Support Package for Xilinx Zynq-Based Radio)

RF Capture
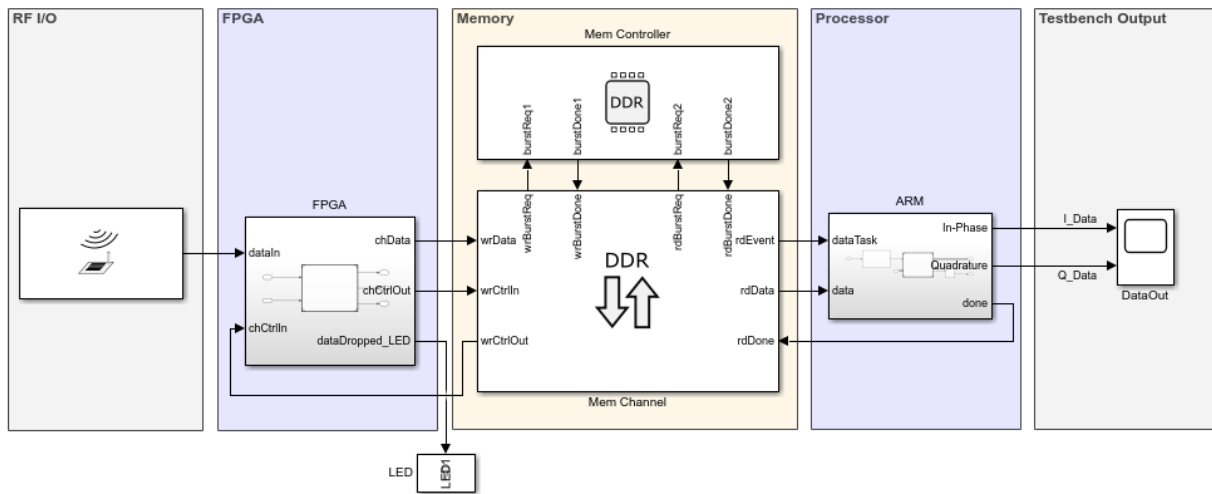
Copyright 2019 The MathWorks, Inc.

1. Open RF Capture model. Load the provided pre-generated FPGA bitstream for this model to hardware.

```
socLoadExampleBitstream(hw,'soc_rfcapture')
```

2. Configure radio card.

```
rf = rfcard(hw);
rf.CenterFrequency = 1090e6;
rf.GainSource = 'AGC Fast Attack';
rf.BasebandSampleRate = 4e6;
rf.ShowAdvancedProperties = true;
rf.ShowInternalProperties = true;
rf.BISTToneMode = 'Tone Inject Rx';
rf();
```

3. Setup data recorder.

```
dr = soc.recorder(hw);
src = soc.iosource(hw,'AXI Stream Read');
src.devName = 'mwfpga_data_capture_ip0:s2mm0';
src.dataTypeStr = 'uint32';
src.SamplesPerFrame = 4000;
src.SampleTime = src.SamplesPerFrame/rf.BasebandSampleRate;
addSource(dr,src,'AXI4 stream interface');
```

4. Record radio signals.

```
setup(dr)
system(hw,'devmem 0x40010100 32 1');
record(dr,1)
while isRecording(dr)
     pause (0.1);
end
save(dr,'zynq_rf_data')
```

5. To playback the recorded RF data, open RF Playback model. Enter the dataset name and the source name on the **IO Data Source** block and simulate the model.
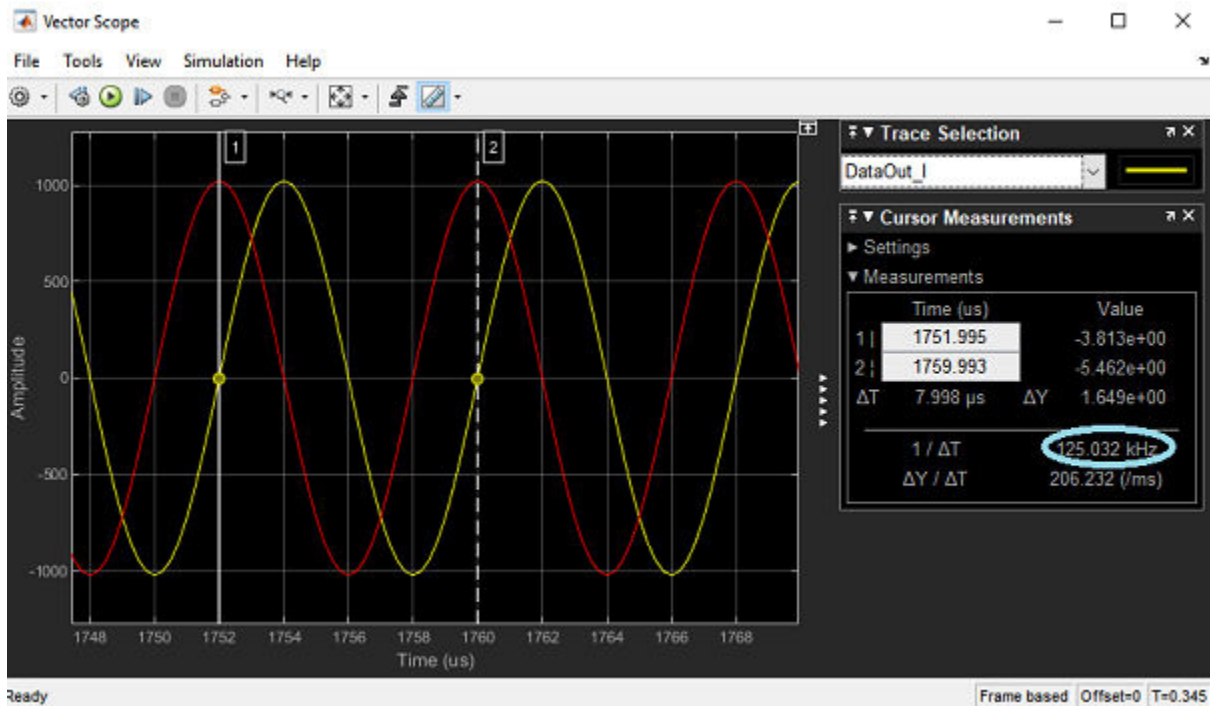
A pre-recorded dataset file **zynq_rf_data.tgz** is available at `matlabroot\toolbox\soc\socexamples`.

See Also

"Simulate with I/O Data Recorded from SoC Device" on page 5-71

Simulate with I/O Data Recorded from SoC Device

This example shows you how to use recorded real-world data in simulation.

Supported hardware platforms:

- Xilinx® Zynq® ZC706 evaluation kit
- Xilinx Zynq UltraScale™+ MPSoC ZCU102 Evaluation Kit
- Altera® Cyclone® V SoC development kit
- Altera Arria® 10 SoC development kit

In many situations you may want to verify your algorithm against real-world data. This example shows how to use the recorded data signal in a simulation of the generated processor system model of the complete SoC application.

We recommend completing "Streaming Data from Hardware to Software" on page 5-37 example.

Use Recorded Data in Simulation

In this section, you will simulate the processor subsystem of the SoC application model with recorded data as input. The processor subsystem of the SoC application uses AXI4 protocol to stream data from external memory and determine if the signal contained in the data is either high or low frequency. An **IO Data Source** block replaces the external memory and the FPGA subsystem of the model with playback of the AXI4 stream data. You will use data recorded in "Record I/O Data from SoC Device" on page 5-64 example.

Signal Detection
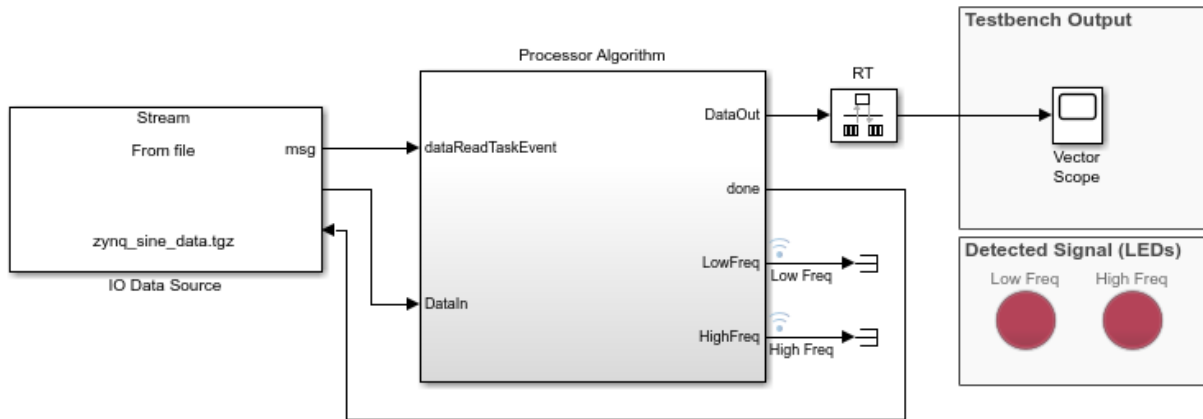
Copyright 2019 The MathWorks, Inc.

1. Open Signal Detection model.

2. Open **IO Data Source** block mask.

3. Click **Browse...** and select the **matlab\toolbox\soc\socexamples\zynq_sine_data.tgz** file containing recorded data.

4. Click **Select...** and choose the data source within the data file to playback. Click **OK** to close the block mask dialog.

5. Run the Simulink® model and open **Vector Scope** to observe the recorded data.

6. To access the recorded data in MATLAB®, use `socFileReader`.

```
h = socFileReader('zynq_sine_data.tgz');
data = getData(h,'AXI4 stream interface');
```

The returned data is a time series object. Plot the data in MATLAB.

```
plot(data);
```

See Also

"Record I/O Data from SoC Device" on page 5-64

Task Execution

This example shows how to simulate task execution and how to generate code and run it on an SoC hardware board.

Application development often includes simulating an algorithm to ensure the correct behavior. Such simulations usually ignore the real-time aspects of an embedded system environment. This may allow certain timing problems to remain undiscovered until the application runs on hardware.

The timing problems often lead to incorrect application behavior. SoC Blockset helps you detect these problems in simulation rather than on hardware. This can help you avoid costly debugging on hardware.

Timing problems are more likely to occur as applications become more complex. For example, rate overruns and undesired rate preemption are more frequent in applications with multiple tasks due to resource constraints and task dependencies. Simulating multitasking applications with SoC Blockset will help you in detecting these problems early.

In this example, task execution is simulated using SoC Blockset. You will learn about different techniques for simulating task duration and when to use them. You will also learn how to verify the timing specifications on hardware.

Supported hardware platforms:

- Xilinx® Zynq® ZC706 evaluation kit
- Xilinx Zynq UltraScale™+ MPSoC ZCU102 Evaluation Kit
- ZedBoard™ Zynq-7000 Development Board
- Altera® Cyclone® V SoC development kit
- Altera Arria® 10 SoC development kit
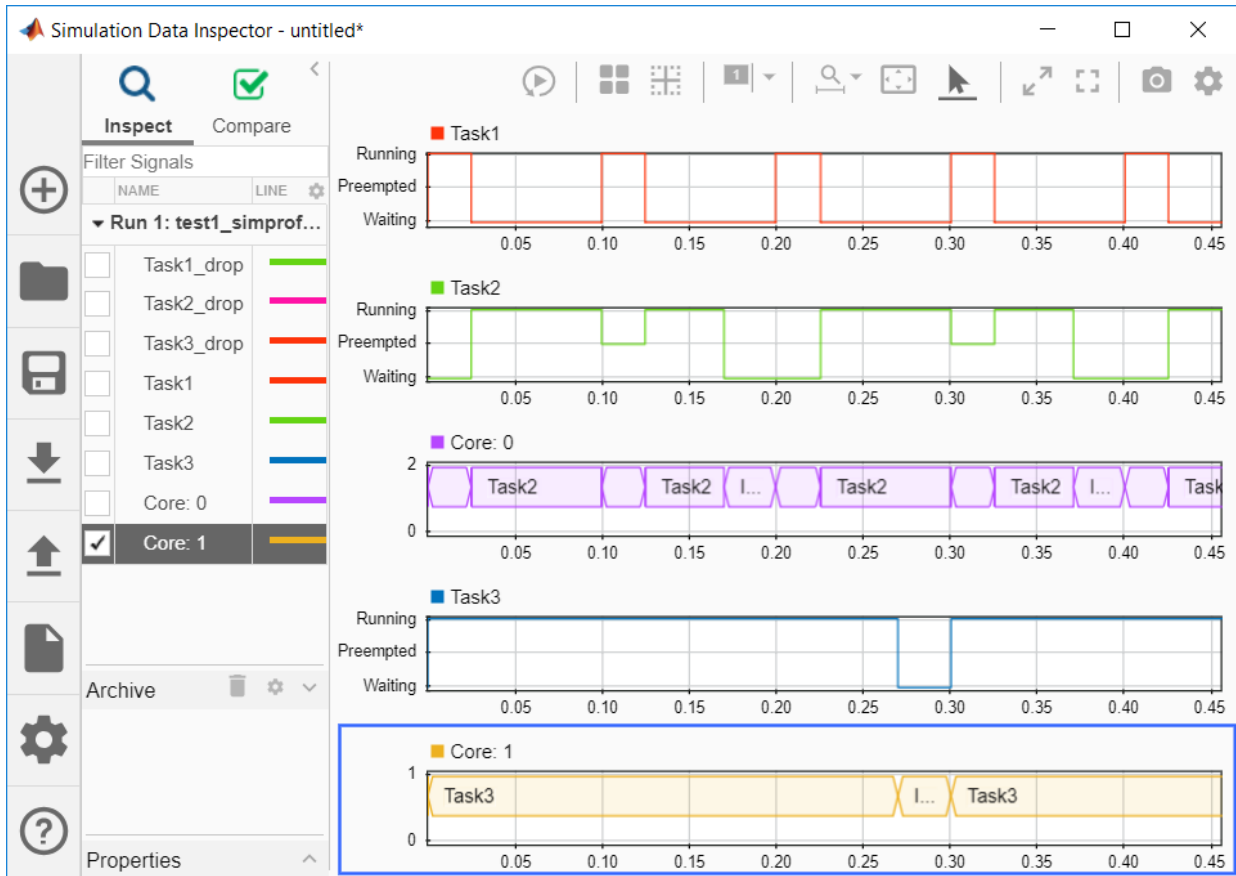
Task Execution
Case 1
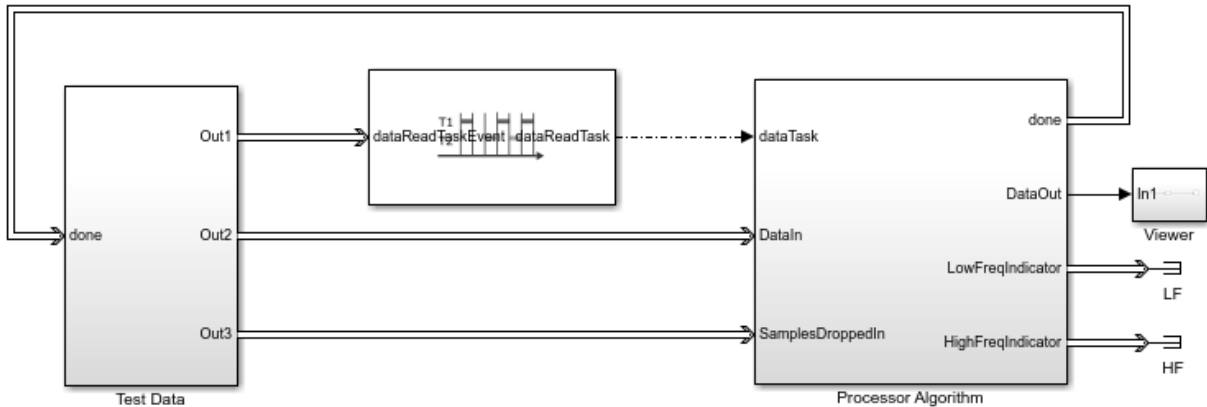
Copyright 2019 The MathWorks, Inc.

Introduction

SoC Blockset simulates the execution of software tasks as they would execute on an SoC processor. The simulation honors the parameters of the task, such as period, priority and processor core. SoC Blockset simulates task preemption, task overruns, and concurrent task execution.

The following diagram illustrates the above-mentioned task execution simulation aspects. In the first two subplots, you can observe that Task1 executes every 0.1 s and, since they both share Core 0, Task1 preempts Task2 that executes every 0.2 s. In the third subplot, you can observe that Core 0 still has some idle time. The last two subplots show Task3 running every 0.3 s on Core 1.

To learn more about simulating task execution, see "What is Task Execution?" on page 1-2

The **Task Manager** block allows you to configure execution of the tasks in your model. In the block dialog, you define how many tasks you need in your system using **Add** and **Delete** buttons. On the **Main** tab of the dialog, you set the main task properties, while on the **Simulation** tab you set the simulation task properties.

The following figure illustrates the **Main** tab of the **Task Manager** block.
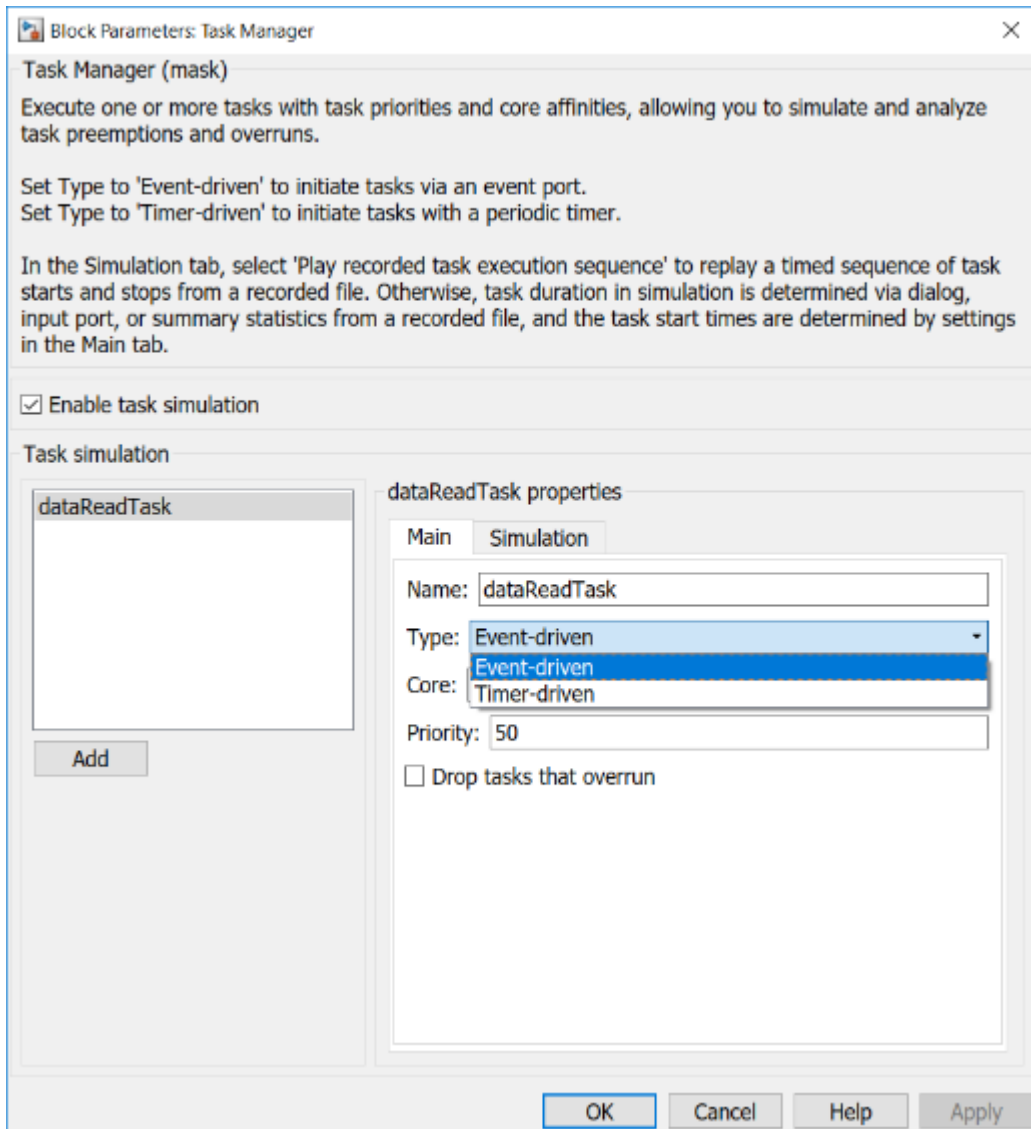
A task has a name so that it can be identified in the model and the various associated plots. Port labels on the **Task Manager** block use the task names for easy identification.

A task can be of two types. An event-driven task executes when triggered by an event. An event line from an IO data source block connected to the **Task Manager** block triggers the task. A timer-driven task executes with a defined period as defined in the **Main** tab of the **Task Manager**.

You define the priority of event-driven tasks in the **Main** tab of the **Task Manager**. Timer-driven task priority is assigned automatically.

In the **Task Manager** dialog you may also set the processor core on which to execute a task so that, if your hardware board has multiple cores, you may set the tasks to execute concurrently.

The **Task Manager** block also allows you to configure how task overruns are handled. For example, you may decide to drop an instance of a task if the previous task instance has not started or completed. Or, you may decide to try to catch up with the task schedule despite overruns.

To simulate real-time task effects, such as preemption and overruns, SoC Blockset requires you to provide the duration of each task. The duration is defined as the time elapsed between the task start and the task end. Ideally, you will measure the task duration on your hardware board. If that is not possible, look up the task duration in the data sheets provided by the task algorithm developers. As a last resort, you should set the duration relative to the task period or the shortest recurrence interval for aperiodic tasks.

SoC Blockset has several choices for setting the task duration. As the task duration is applied only to simulation, these choices are found in the **Simulation** tab of the **Task Manager** dialog.

The following figure illustrates the **Simulation** tab of the **Task Manager** dialog.
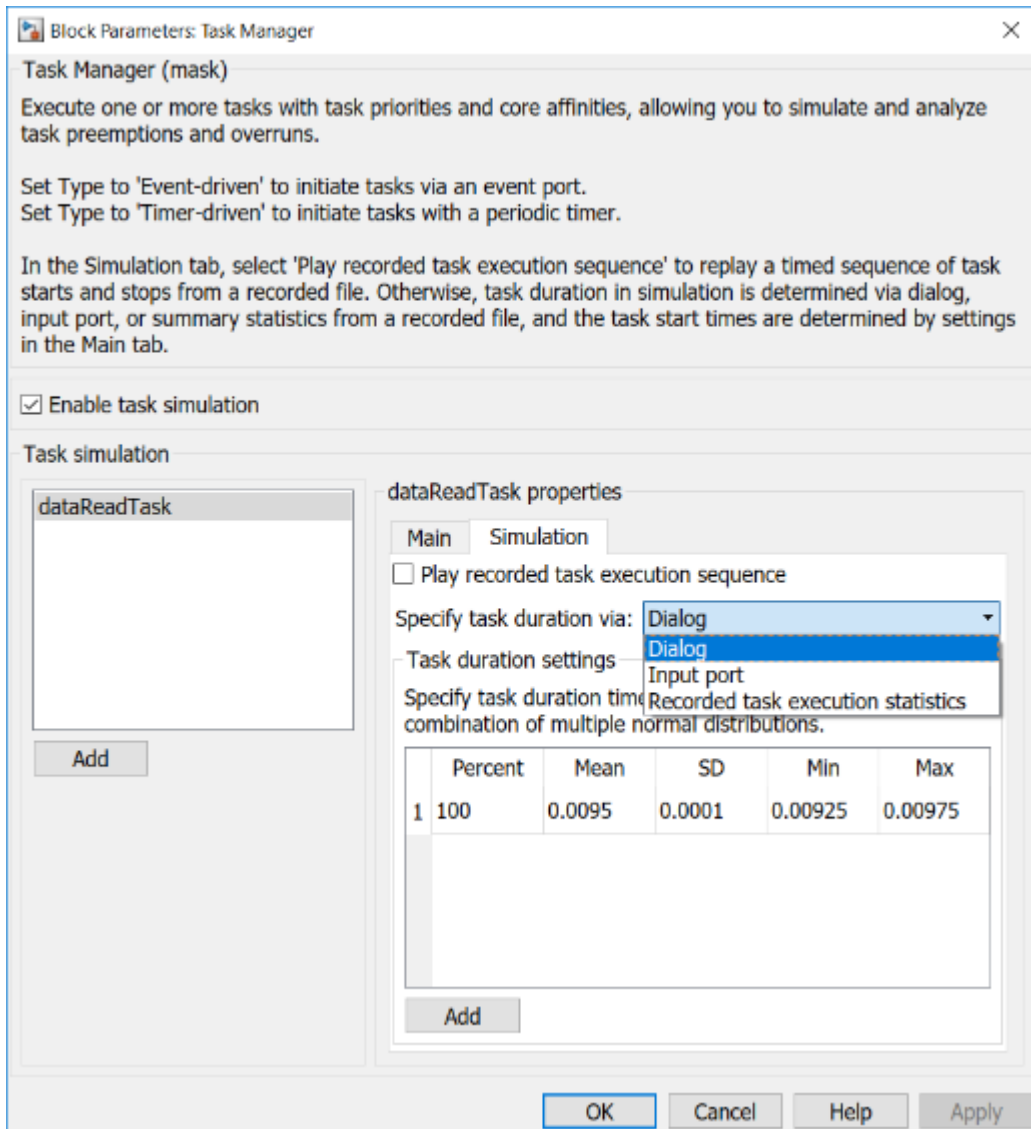
The most commonly used options are:

- **Dialog** - Allows you to specify task duration via a normal distribution, or a combination of multiple normal distributions, using the mean and the standard deviation parameters.

- **Input port** - Allows you to specify task duration on an instance basis. For example, you may create a model that calculates task duration and connect it to the **Task Manager** input port.

The following flowchart will guide you in selecting the most appropriate option.
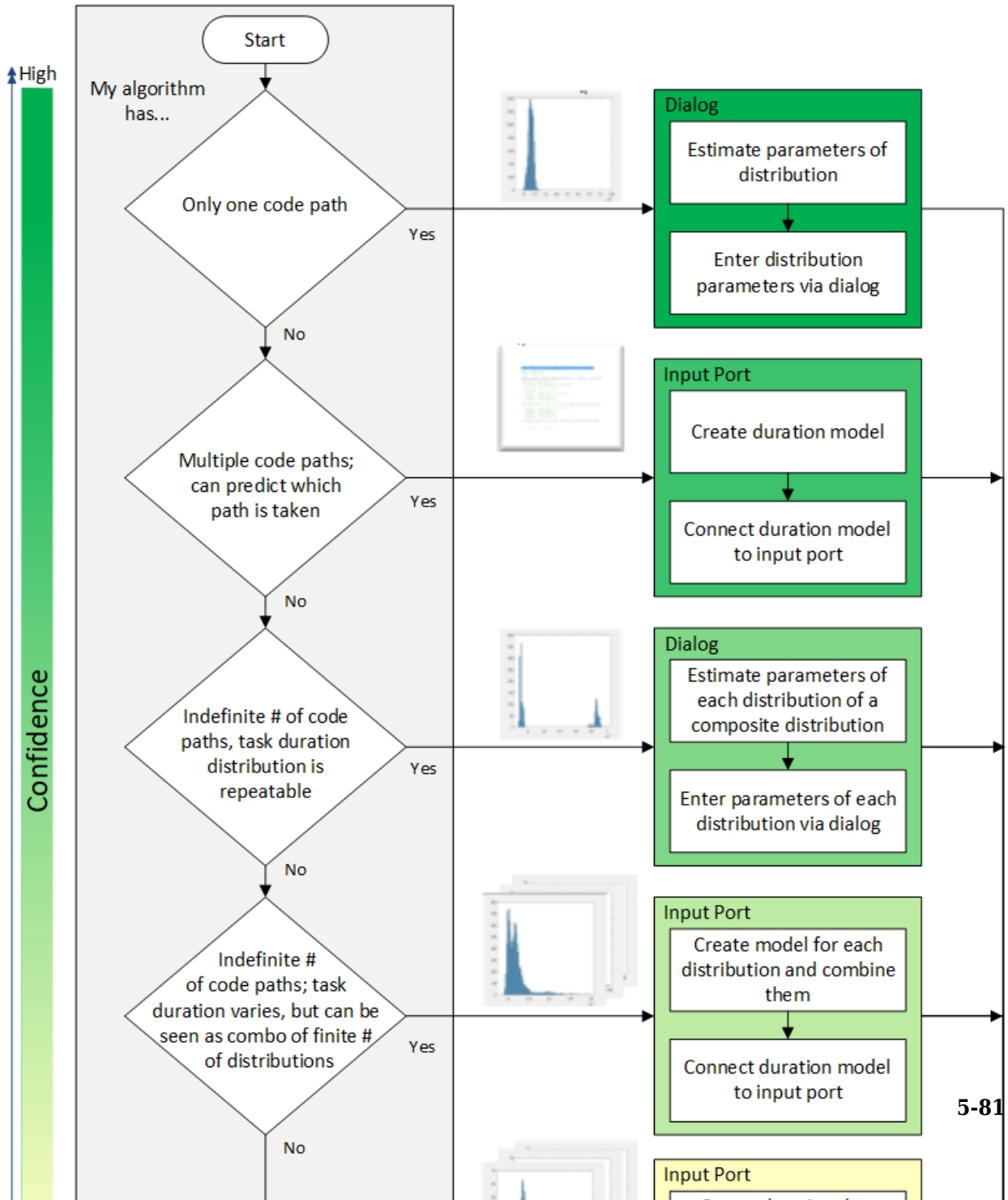
If the duration times for your task have different distributions and causes, select the most fitting options using the flowchart as general guidance.

You can configure additional simulation and execution parameters for SoC Blockset in the model configuration dialog. Task profiling, in simulation and on processor, allows you to profile task execution, stream results to Simulation Data Inspector and save them into a file.

You can also set the kernel latency value to affect task execution in simulation. This value varies a lot but is typically much smaller than task duration. Therefore, we recommend you leave the value set to 0 s unless you can deterministically find the appropriate value for your hardware board.

The following figure shows SoC parameters related to task execution in the model configuration parameters dialog. Note that the **Task profiling on processor** panel shows only if you install all required products and hardware support packages.
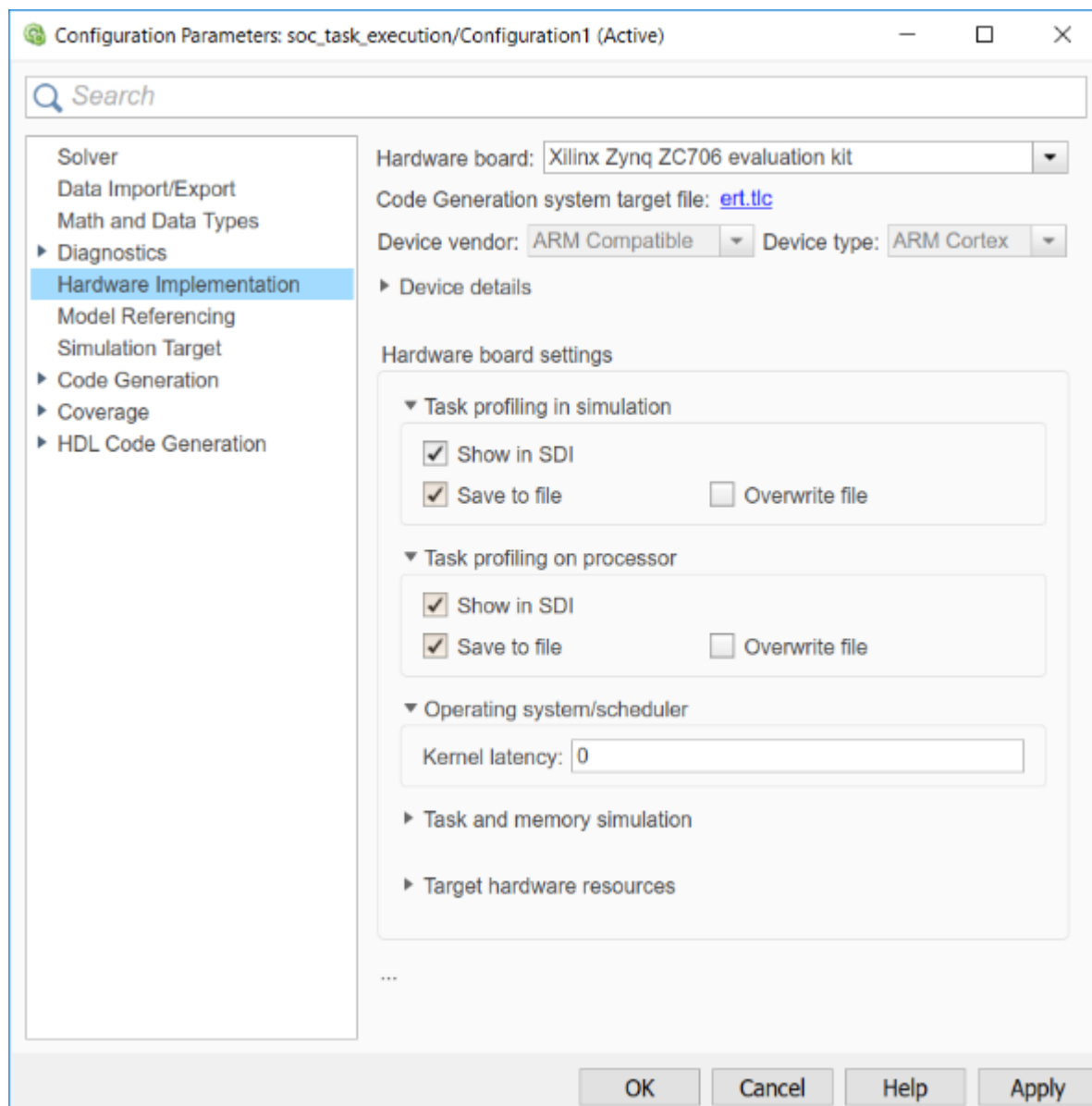
The remaining steps of this example will illustrate some of the options shown in the above flowchart.

Case 1 - Simulating an Algorithm with Single Code Path

This case requires you to simulate a DSP algorithm that processes a frame of data. The following product is required for that:

• DSP System Toolbox

If you do not have this product, proceed to the next case after reviewing the description of this case.

In this case, you will learn how to model the task duration when the task algorithm has a single code path.

Assume that you are tasked with developing an application that processes RF (radio frequency) data on an SoC board. After being preprocessed in the FPGA core, the data is streamed to the processor core using the AXI4 protocol. The algorithm running on the processor core should determine whether the data contains a high-frequency or a low-frequency signal. To that end, a low-pass and a high-pass filter are applied to the data. The resulting signals are then compared to a selected threshold. Based on this description, this task has a single code path, with no major code branches. The source code for the task function might have the following form.

```
double dataReadTask(double in[])
{
    /* Frame size is always 1000 */
    int signalType; /* 0 - LP, 1 - HP */
    double out1[1000], out2[1000];
    filterLP(in, out1, 1000);
    filterHP(in, out2, 1000);
    signalType = thresholding(out1, out2, 1000);
}
```

1. Open the model. Note the **Test Data** subsystem. The **RF Data Source** block in the subsystem represents the external memory and the FPGA core. The **RF Data Source** block has two output ports, **Stream Data** and **event**. They output the RF data and a notification when new data frame is available, respectively.

2. Note that the **RF Data Source** block generates frames of 1000 samples every 0.01 s. The frames are samples of a 1 kHz sine waveform.

3. Click the **Task Manager** block. Observe that it sets an event-driven task **dataReadTask**. The task is triggered by the arrival of a new data frame.

4. Click the **Simulation** tab in the **Task Manager** dialog to define the task duration for simulation.

Since the algorithm consists of two filters executing without conditions, the application has a single code path. Therefore, you follow the first left branch in the flowchart shown in the introduction and you expect that the algorithm execution times have a normal distribution.

Based on the information given by the algorithm developer, you determine that the mean execution time is 0.0095 s and that the standard deviation is 0.0001 s. To represent the real-time limits, you also decide to set the min and the max execution times to 0.00925 s and 0.00975 s, respectively.

Set the duration parameters in the **Task Manager** dialog in the **Simulation** tab as described above.

5. In the model, click **Run** to start the simulation. Wait until the simulation completes.

6. From the model toolbar, open the **Simulation Data Inspector** and inspect the **dataReadTask**. Zoom in to inspect the task execution times more closely.
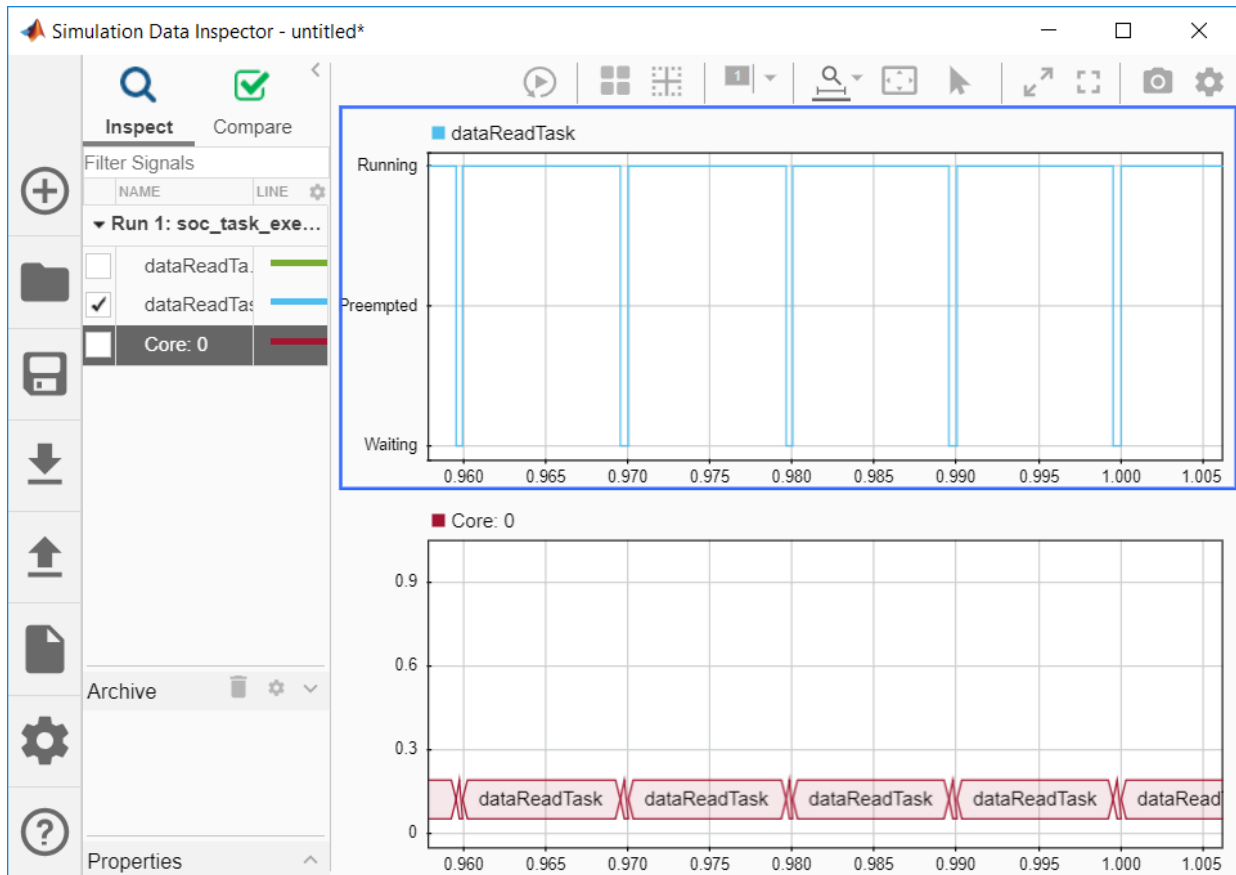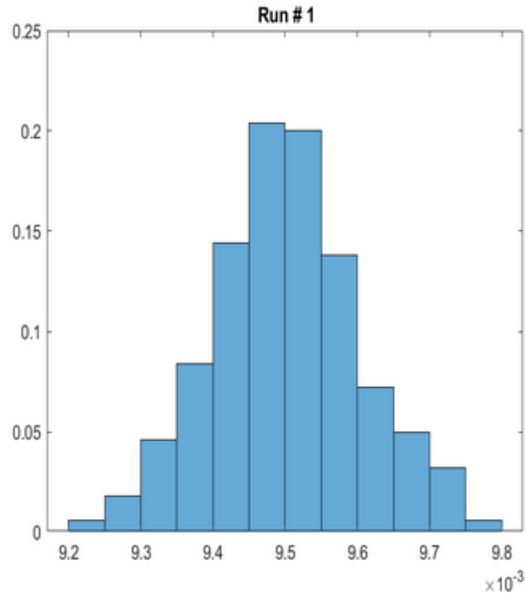
7. Run the following command to perform the statistical analysis of the task execution times. Observe the **Simulation Data Inspector** run numbers. Modify the command if your run numbers are different.

```
socTaskTimes('soc_task_execution', 'Run 1: soc_task_execution_simprofile')
```

Task: dataReadTask

Histogram of the execution times

Statistics of the execution times

	Mean	SD	Min	Max
Run # 1	0.0095026	0.00010217	0.00925	0.00975

Observe that the task durations vary. As expected, the histogram of the task duration times indicates that the algorithm has one code path. The duration values are clustered around the mean value of 0.0095 s.

8. Close the model without making any changes.

Case 2 - Simulating an Algorithm with Two Code Paths

In this case, you will learn how to model the task duration when the task algorithm has two code paths and it can be predicted which path will be taken.

Assume that you are developing a video surveillance application. The task is to constantly process video data to determine if there was intrusion in the system. The algorithm calculates the amount of scene change between consecutive video data frames. If the scene change exceeds the selected threshold, such frames are recorded as they may be used as evidence of potential intrusion. Thus, this algorithm has two code paths. The source code of this algorithm may be represented in the following form.

```
void VideoTask(single in[], in length, double threshold)
{
    double energy;
    energy = calcSceneChange(in, length);
    if (energy > threshold)
        recordFrame(in, length);
    }
}
```


Task Execution
Case 2
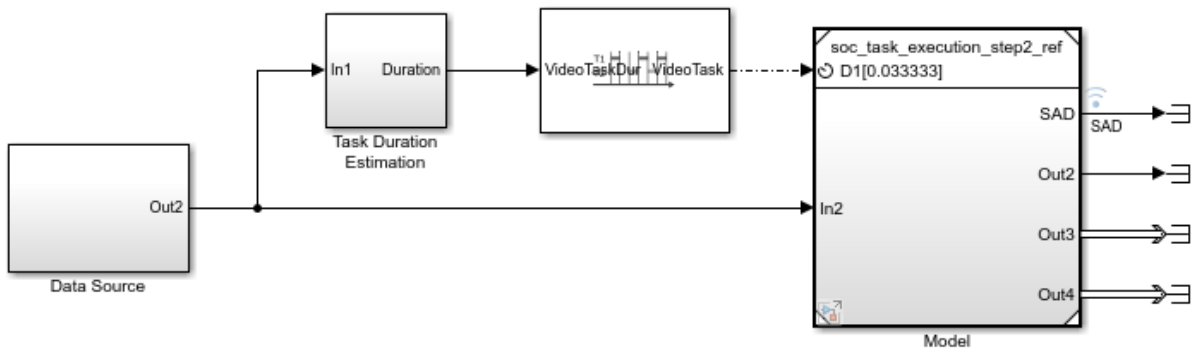
Copyright 2019 The MathWorks, Inc.

**1.** Open the model. Note the **Data Source** block that outputs the frames of video data.

2. Click the Model block and observe that the algorithm calculates motion energy between consecutive frames of data. If the calculated motion energy exceeds the threshold, the **Main Algorithm** is executed.

3. Click the **Task Manager** block. Observe that it sets a timer-driven task **VideoTask**. This task runs every 0.33333 s, which is the video frame rate.

4. Click the **Simulation** tab in **Task Manager** dialog to define the task duration for simulation.

Since the algorithm has two code paths and it can be predicted which code path will be taken, follow the second left branch in the flowchart.

Model task duration to depend on motion energy. Depending on whether the motion energy threshold is exceeded or not, you will assign the task duration with the mean of 75% or 50% of the frame rate, respectively.

Click the **Task Duration Estimation** subsystem to understand how to model task duration.

5. In the model, click **Run** to start the simulation. Wait until the simulation completes.

6. From the model toolbar, open the **Simulation Data Inspector** and inspect **VideoTask**. Zoom in to inspect the task execution times more closely.
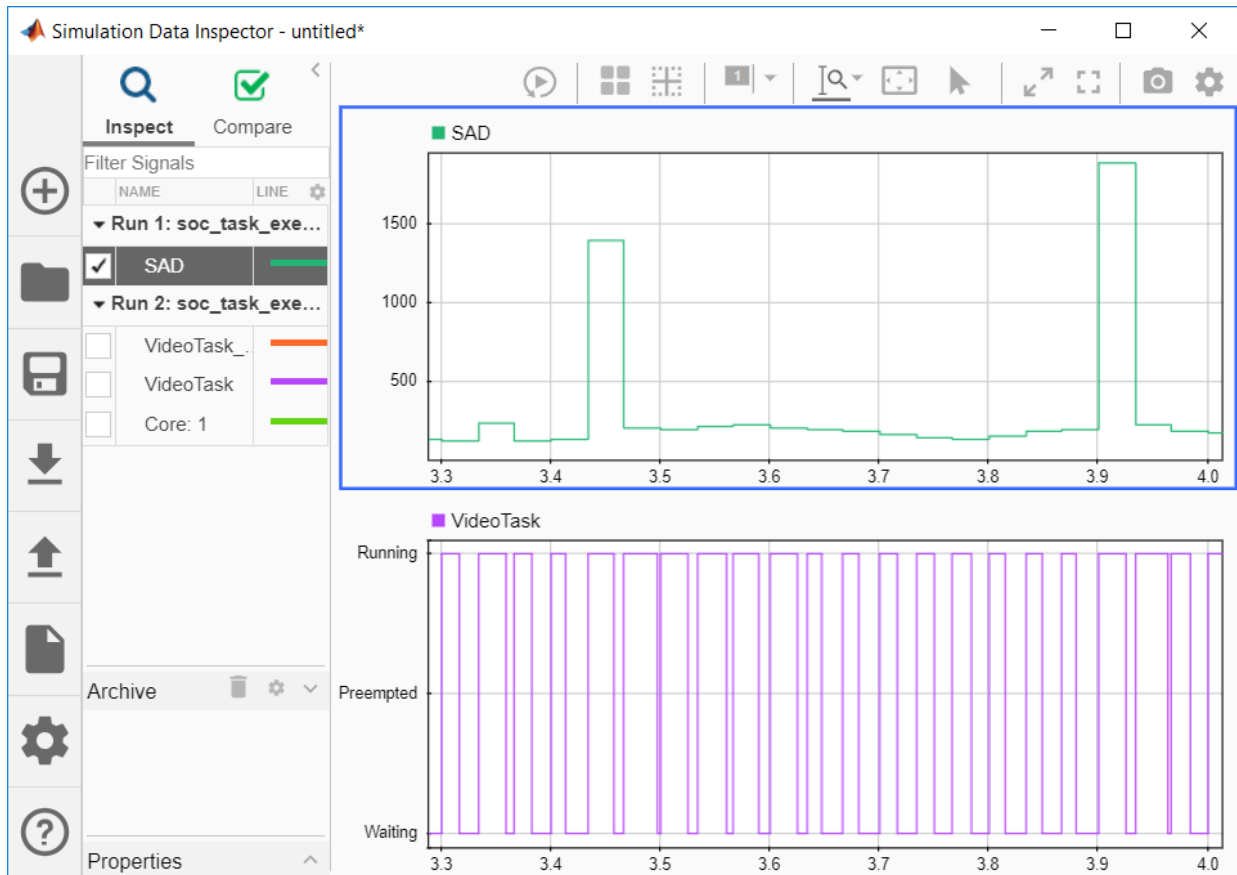
7. Run the following command to perform the statistical analysis of the task execution times. Observe the **Simulation Data Inspector** run numbers. Modify the command if your run numbers are different.

```
socTaskTimes('soc_task_execution_step2', 'Run 3: soc_task_execution_step2_simprofile
```

Histogram
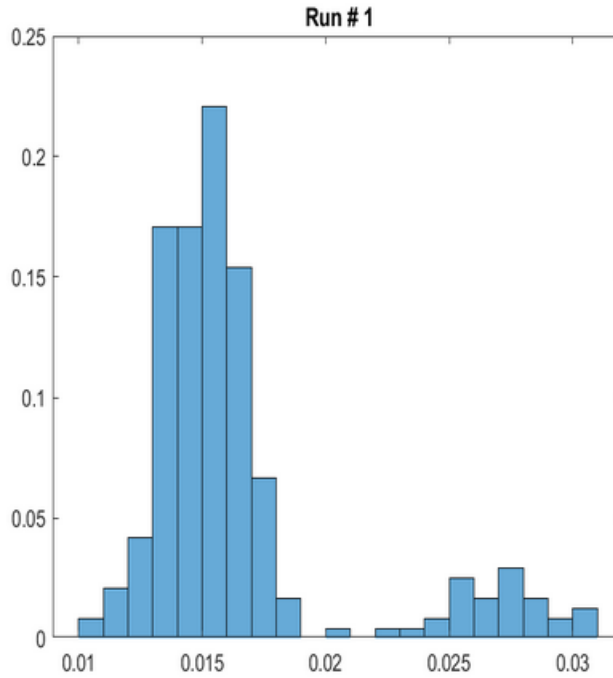
Run # 1

Statistics

	Mean	SD	Min	Max
Run # 1	0.016538	0.0043039	0.01069	0.030948

Observe that the task durations vary. As expected, the histogram of the task duration times indicates that the algorithm has two code paths.

8. Close the model without making any changes.

Case 3 - Simulating an Algorithm with Indeterminate Number of Code Paths

This case requires you to generate and run code on a hardware board. The following products are required for that:

- Embedded Coder
- SoC Blockset Support Package for Xilinx Devices, or
- SoC Blockset Support Package for Intel Devices

In this case, you will learn how to model the task duration when the task algorithm has an indeterminate number of code paths, but the code paths are repeatable for the given set of data.

In this case, assume that you are developing a complex application that processes data on an SoC board. Due to the complexity of the processing, the algorithm has an indeterminate number of code paths. As a result, it is not possible to predict which code path will be taken. However, it is known that the distribution of task durations is repeatable in multiple experiments. The source code for such an algorithm might have the following form.

```
int myTask(int arr[], int length)
{
    int i = 0;
    int sum = 0;
    while (i < length) {
        if (arr[i] > 0)
            sum = sum + arr[i]
        i++;
    }
}
```

Task Execution
Case 3
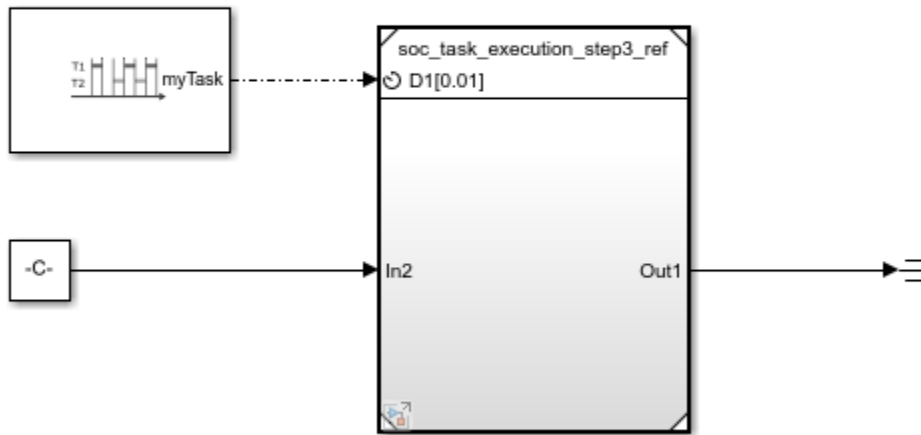
Copyright 2019 The MathWorks, Inc.

1. Open the model.

2. The model is set for **Xilinx Zynq ZC706 evaluation kit board**. To use a different board, go to the model Configuration Parameters dialog and select one of the supported boards listed in the **Hardware Implementation** page. Do the same for the top model and the referenced model.

3. Click the **Task Manager** block and select the task **myTask**. Click the **Simulation** tab. Observe that we define the probability distribution as a combination of two normal distributions.

4. Click **Run** to start the simulation. The task execution data will be streamed to the **Simulation Data Inspector**.

5. Next, in the model toolbar change the simulation mode to **External**. The model is already set to profile task execution as it runs on hardware.

6. Click **Run**. After the code is generated and built, it will start executing on your hardware. The profiling data will be streamed to **Simulation Data Inspector** in real-time.

7. Run the following commands to perform the statistical analysis of the task execution times obtained in simulation and on hardware. Observe the **Simulation Data Inspector** run numbers. Modify the commands if your run numbers are different.

```
socTaskTimes('soc_task_execution_step3', 'Run 4: soc_task_execution_step3_simprofile
socTaskTimes('soc_task_execution_step3', 'Run 5: soc_task_execution_step3_procprofile
```

Histogram
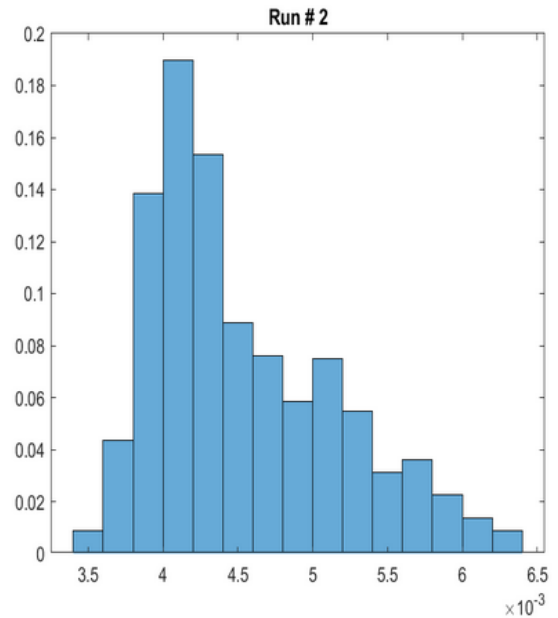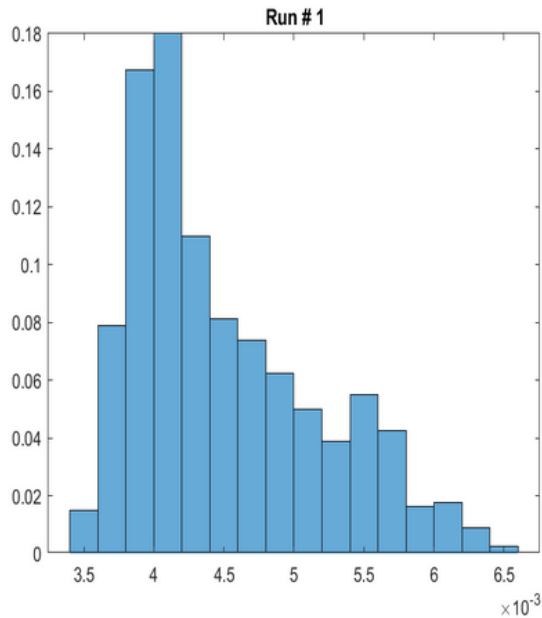
Statistics

	Mean	SD	Min	Max
Run # 1	0.0045092	0.0006698	0.0035	0.0065432
Run # 2	0.0045384	0.00062219	0.00354	0.0063962

Notice that the task durations obtained in simulation match the results obtained on hardware.

8. Close this model without making any changes.

Summary

This example showed you how to simulate task execution in a multitasking operating system, how to generate code and run it on a hardware board, and how to collect the real-time task execution data.

In this example, we used simple applications, each with one task. In a typical application, however, multiple tasks must be performed. Embedded applications must run each task per defined schedule. To allow for using the processor most efficiently and to react quickly to external events, a priority-based preemptive scheduling algorithm is used.

With priority-based preemptive scheduling, when a task gets preempted, a task switch occurs. The data used by the task (task context) is saved so that it can be restored when the task resumes executing. In this example, the task switching times are dwarfed by the task duration and are not simulated. In applications with much shorter task duration, you may need to consider them.

If a hardware board has multiple processor cores, embedded applications typically attempt to use all cores for the most efficient implementation. SoC Blockset uses a priority-based preemptive scheduling algorithm even when the processor has multiple cores. SoC Blockset honors assignment of tasks per core in both simulation and generated code.

Next, we recommend completing "Streaming Data from Hardware to Software" on page 5-37 example that illustrates a systematic approach to designing a complex SoC application using SoC Blockset.
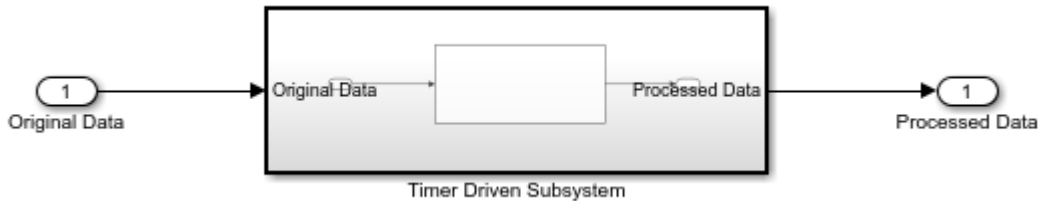
Timer-Driven Task

This example shows how to use the Task Manager block in a simple system where a timer-driven task samples and modifies data generated from a random number source.

Task Manager and Software Application Model

The following model simulates a software application running on an ARM processor. A Task Manager block schedules the execution of the `Timer Driven Subsystem`, inside the `Software Application Model Reference` block. A Random Number block simulates an data source that the timer driven task samples.
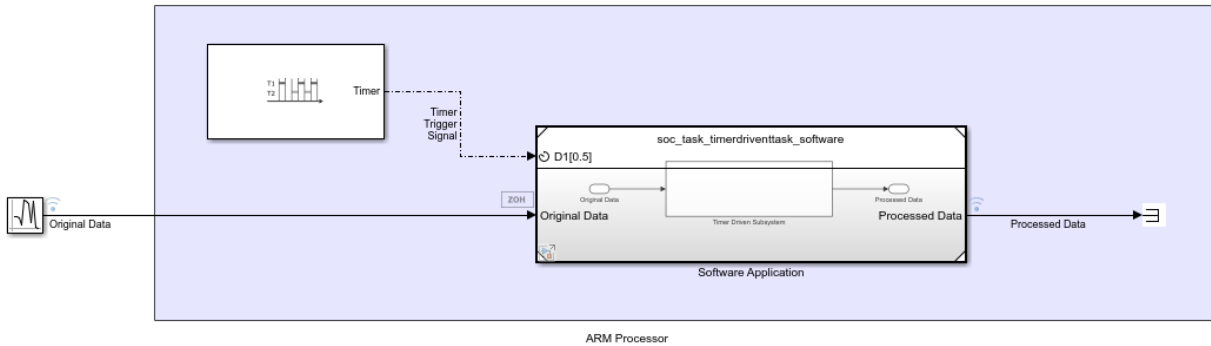

The following model shows the Software Application model. This model contains the Timer-Driven Subsystem that executes based on the Timer Task events from the Task Manager block in the top-level model.


The Timer-Driven Task Subsystem, a Subsystem, samples a data value every 0.5 seconds from the Random Number block and applies the Algorithm. In this model, the algorithm outputs the negative scalar value of the sampled data value. The following model shows

the Algorithm subsystem contained in the Timer-Driven Subsystem. The Inport block defines the 0.5 second sampling time for the Timer Driven Subsystem visible on the Software Application model when the Schedule rates parameter is enabled.
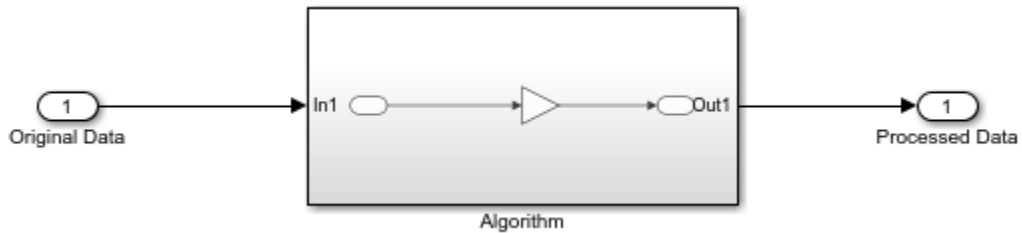
Asynchronous Simulation and Results

Click the Run button to build and run the model. When the model finishes running, open the SDI display to see the results of the simulation. Select the Timer_Task, original data, and processed data signals to see the effect of asynchronous task execution.
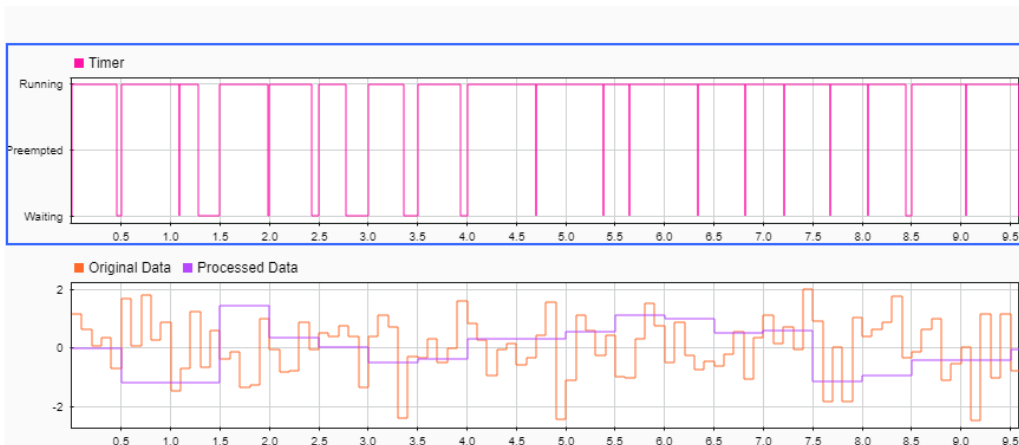
As shown in the SDI display, the running time of the Timer_Task varies at each instance. In some cases, the duration of the previous task execution delays the start of the next task execution. Additionally, the processed data from the task outputs at a the same time as

the completion of the task execution, resulting in observed delay in the processed Data compared to the original data. As a result, despite the specified time step of 0.5 seconds, the start of execution now behaves as if the subsystem were executed on an SoC device processor with the associated real world processing limitations.

Event-Driven Task

This example shows how to use the task manager block to a simple system where data from UDP source gets processed asynchronously each time a data packet arrives. The task manager block

Task Manager and Software Application Model

The following model simulates a software application running on an ARM processor. A Task Manager block schedules the execution of the Asynchronous Subsystem, inside the Software Application Model Reference block. An IO Data Source block simulates the network transmission of UDP packets.
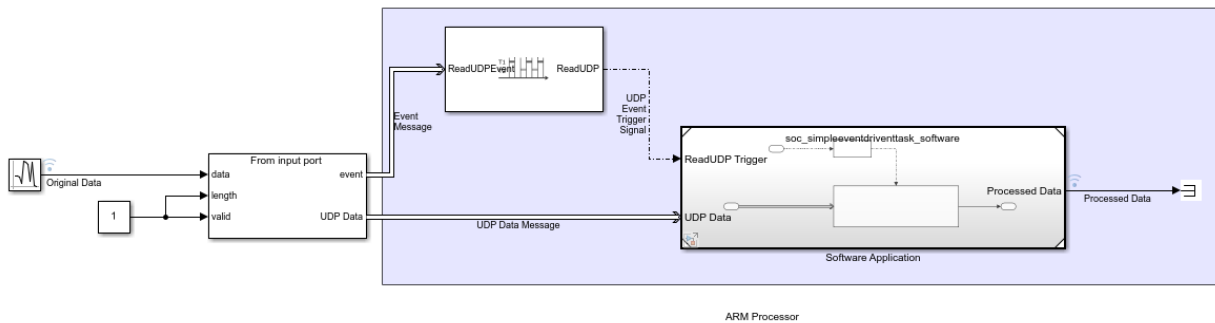
The Software Application contains the Asynchronous Task Subsystem, a Function-Call Subsystem, that executes each time an event trigger occurs. An Asynchronous Task Specification block specifies the priority of the UDP Task to match the priority set in the Task Manager block. A Rate Adaptor block allows sampling of the output signal of the Asynchronous Task Subsystem at the time step of the Simulink(c) model.
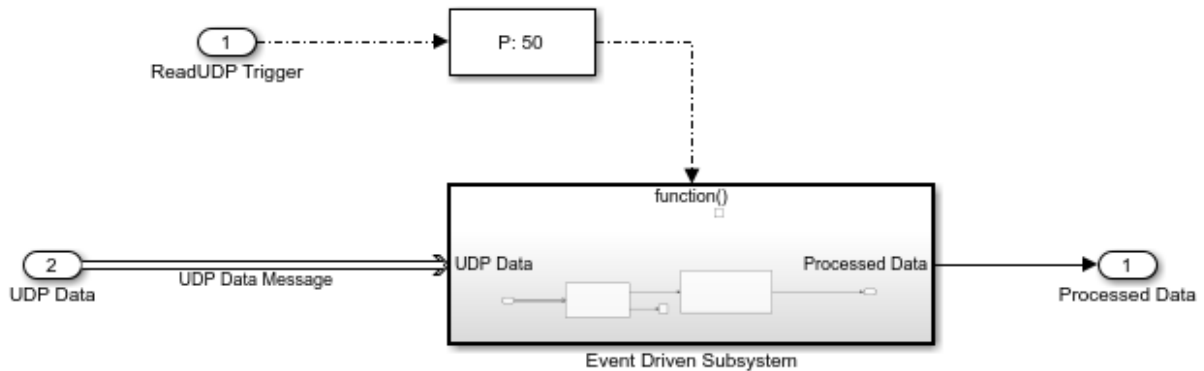
The Asynchronous Task Subsystem, a Function-Call Subsystem, reads a data value from a UDP Read block and applies the Algorithm each time a new UDP data value arrives. In this model, the algorithm outputs the negative scalar value received from the UDP Read block. The following model shows the UDP block and Algorithm subsystem contained in the function-call subsystem.
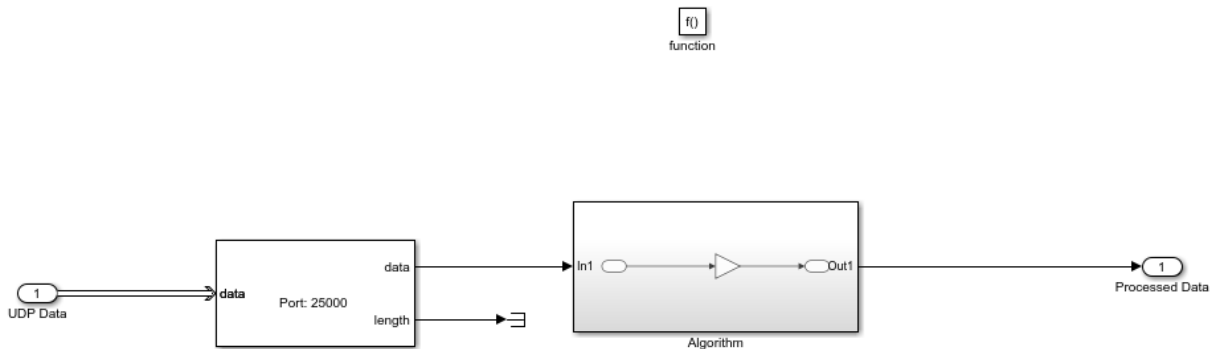
Asynchronous Simulation and Results

Click the Run button to build and run the model. When the model finishes running, open the SDI display to see the results of the simulation. Select the ReadUDP, original data, and processed data signals to see the effect of asynchronous task execution.
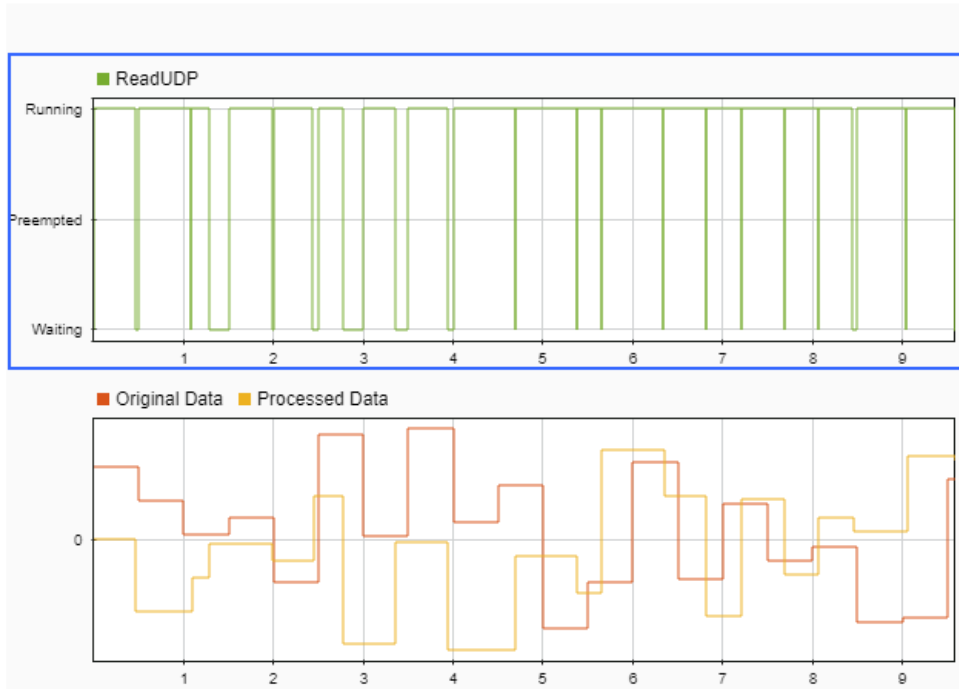
As shown in the SDI display, the Running time of the ReadUDP varies at each instance of receiving a UDP data packet. In some cases, the previous task execution delays the start of the next task execution. While, in this example, the UDP packets arrive at a fixed rate relative to the Simulink sample time, the start of the task execution is not directly dependent on the sample time. The processed data from the task outputs at a the completion of the task execution, resulting in observed delay in the processed Data compared to the original data.

Supported Hardware

SoC Blockset Supported Hardware

Get Support Package Now

With the SoC Blockset, together with Embedded Coder® or HDL Coder, you can generate reference designs for Xilinx and Intel FPGA devices and SoC platforms, including Zynq, UltraScale™, and SoC FPGA. These reference designs can be used with Xilinx or Intel design tools.

SoC Blockset provides support packages that help to automate integration, execution, and verification of reference designs for the SoC devices in this table.

Support Package	Vendor	Earliest Release Available	Last Release Available
SoC Blockset Support Package for Xilinx Devices	Xilinx	R2019a	Current
"SoC Blockset Support Package for Intel Devices"	Intel	R2019a	Current

For a complete list of hardware support packages, see Hardware Support.