

VisualApplets

Tutorial

Table of contents

1. VisualApplets Fundamentals	5
1.1 Introduction and Purpose of This Document	5
1.2 Basic Design Concepts	5
1.3 FPGA Technology	7
1.4 Pipeline Execution in Hardware	7
1.5 Related Topics	9
2. From Empty Design to Running Applet	10
2.1 Workflow Overview	10
2.2 Starting a New Project	11
2.3 Operators, Modules, Links, and the Pipeline	11
2.4 Setting Parameters at Design Time	12
2.5 Design Rules Check (DRC)	12
2.6 Simulation	15
2.7 Building the Hardware Applet	16
2.8 Flashing, Runtime Loading, and Host Software	17
2.9 Design Time and Runtime	17
2.10 Related Topics	17
3. Pipeline Concept	19
3.1 Modules, Links, and Pipeline Structure	19
3.2 Pipeline Inputs and Outputs	20
3.3 Unbuffered Pipeline Flow	21
3.4 Concurrent Processing in Hardware	21
3.5 Throughput and Bottlenecks	22
3.6 Link Properties and Image Formats	25
3.7 Flow Control, Blocking, and Backpressure	25
3.8 Related Topics	26
4. Operator Types and Synchronization	27
4.1 Recognizing Operator Types	27
4.2 O-Type Operators	28
4.3 M-Type Operators	28
4.4 P-Type Operators	28
4.5 Synchronous and Asynchronous Inputs	29
4.6 Synchronization Rules	30
4.7 Merging Independent Sources	31
4.8 Synchronizing with the SYNC Operator	31

4.9 Synchronizing with the InsertLine and InsertImage Operators	35
4.10 Synchronizing with the SourceSelector Operator	36
4.11 Avoiding Synchronization	36
4.12 Troubleshooting: Deadlock	37
4.13 Verifying Your Design with DRC	38
4.14 Related Topics	38
5. Stoppable and Non-Stoppable Data Streams	39
5.1 Stoppable and Non-Stoppable Streams	39
5.2 Non-Stoppable Sources in a VisualApplets Design	39
5.3 Connecting Non-Stoppable Sources	39
5.4 Verifying Your Design with DRC	43
5.5 Typical Pipeline Examples	43
5.6 Related Topics	45
6. Parallelism	46
6.1 Parallelism and Bandwidth	46
6.2 Calculating the Required Parallelism	47
6.3 Changing the Parallelism in a Design	49
6.4 Parallelism Constraints	50
6.5 Parallelism with Memory Operators	51
6.6 Troubleshooting Bottlenecks	51
6.7 Related Topics	52
7. Latency	53
7.1 Latency vs. Bandwidth	53
7.2 Latency Definition	53
7.3 Example Design	54
7.4 Pixel Flow in the Example Design	54
7.5 Calculating Latency	55
7.6 Parallelism and Latency	56
7.7 Finding Latency Values for Operators	56
7.8 Related Topics	56
8. Parameterization	58
8.1 Link Properties and Module Parameters	58
8.2 Link Properties	59
8.3 Propagation of Link Properties	61
8.4 Module Parameters	62
8.5 Verifying Your Design with DRC	66
8.6 Troubleshooting Link and Parameter Conflicts	67
8.7 Related Topics	71

9. Bit Arithmetic	72
9.1 Link Bit Width Limits and Operator Constraints	72
9.2 Bit Widths for Arithmetic Operations	72
9.3 Handling Arithmetic Overflow with Clip and Cast Operators	74
9.4 Signed and Unsigned Values	76
9.5 Fixed-Point Arithmetic	78
9.6 Extracting Bit Fields with SelectBitField	83
9.7 Widening Bit Width for Downstream Software	83
9.8 Verifying Your Design with DRC	86
9.9 Related Topics	87
10. Memory Elements	88
10.1 FPGA Memory vs. Frame Grabber DRAM	88
10.2 Choosing a Memory Operator	89
10.3 ImageFifo	89
10.4 ImageBuffer and LineBuffer	89
10.5 Full Frames, Random Access, Multi-ROI with the FrameBuffer Operators	91
10.6 Sharing DRAM Bandwidth	92
10.7 Recognizing Memory-Related Errors	93
10.8 Typical Pipeline Examples	93
10.9 Related Topics	95
11. Working with Kernels	96
11.1 Kernel Structure	96
11.2 Kernel Columns and Kernel Rows	96
11.3 Kernels and Parallelism	97
11.4 Building Kernels	98
11.5 Processing Kernels	101
11.6 Example Design	106
11.7 Related Topics	106

1. VisualApplets Fundamentals

1.1 Introduction and Purpose of This Document

This topic explains basic concepts you need to know for working with VisualApplets. After you have read this topic, you understand the core ideas behind VisualApplets designs: the image-processing pipeline, how you combine and parameterize operators, and how those concepts relate to the FPGA on the frame grabber.

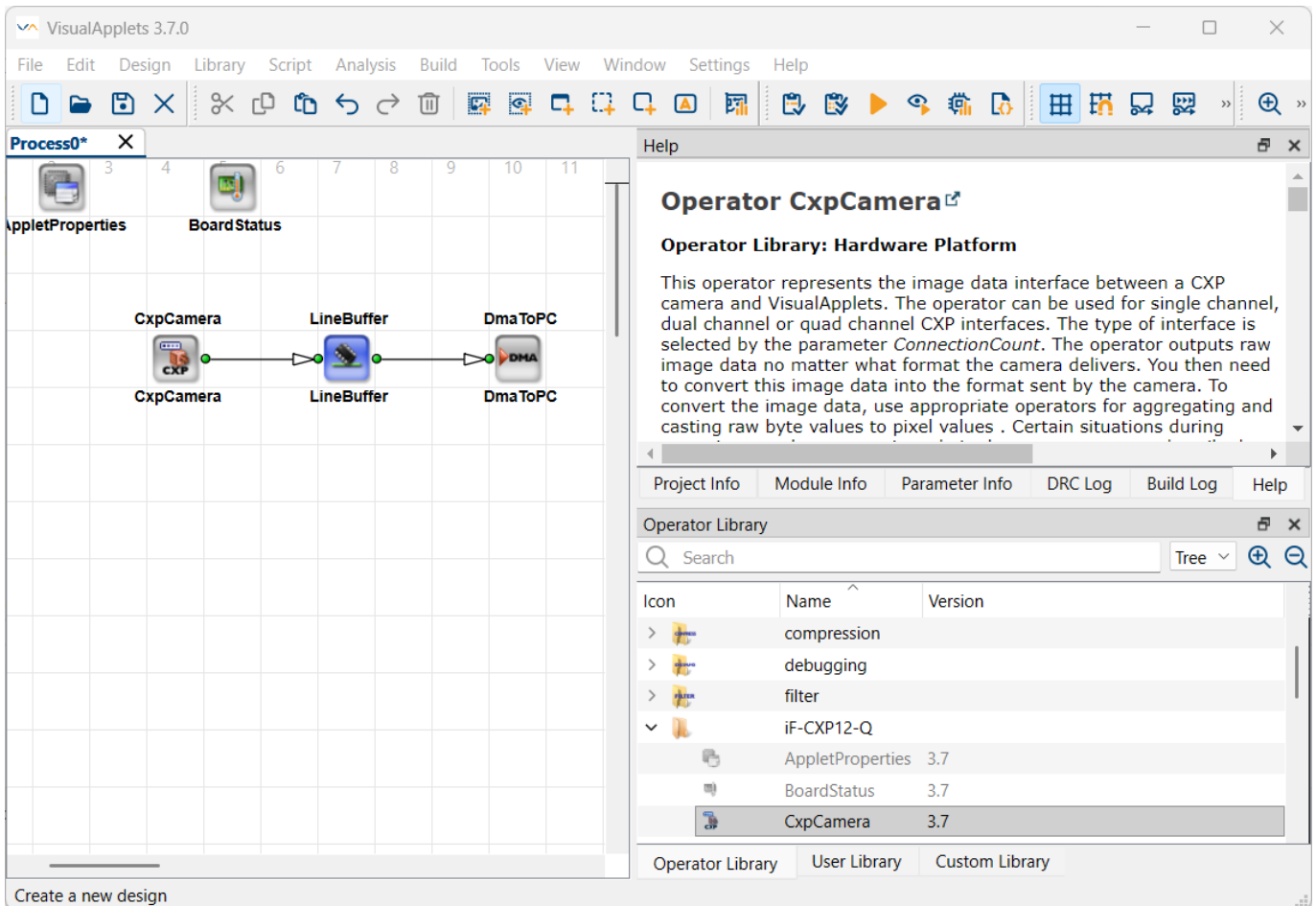
Key Terms

Throughout this tutorial, boxes labeled **Key Terms** explain common terms for readers who do not know them yet.

VisualApplets is the design tool on the PC. The **FPGA** is the chip on the **frame grabber**. You draw the pipeline in VisualApplets. The build configures that pipeline as hardware on the FPGA. Camera data is processed on the frame grabber, not in the VisualApplets editor.

1.2 Basic Design Concepts

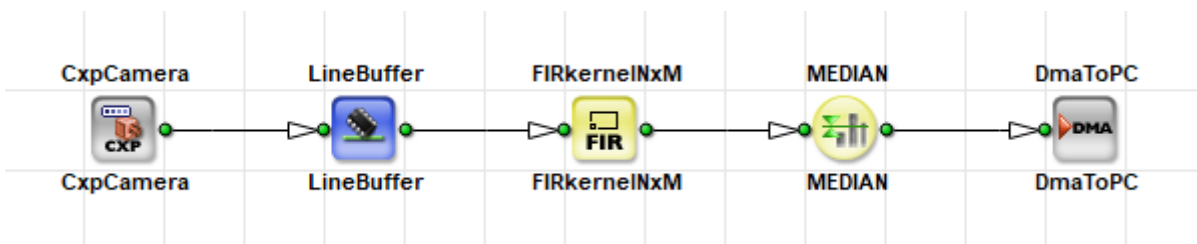
VisualApplets lets you create image-processing functionality on FPGA-based programmable hardware without writing code in a hardware description language like VHDL or Verilog. You build a design in a graphical way from library operators and connect them so that image data flows along defined paths.



Additionally, you can create and edit designs using the Python and Tcl scripting languages.

On a programmable **frame grabber**, the **camera** or another image source delivers a continuous pixel stream into the acquisition path, and the **FPGA** on that board processes that stream. What you build is not a program the FPGA fetches and executes like a CPU. Instead, the build **configures a dedicated digital circuit** on the FPGA whose structure matches your pipeline.

You create and edit the **VisualApplets design** on the PC (operators, modules, and links in a *.vad file). The **build** step then translates that pipeline into hardware logic.



In such a VisualApplets design, you instantiate operators from the operator libraries; each instance of an operator in a design is called a *module*. Modules are connected by links, and the order of processing is determined by how you arrange and connect those modules. Together, the modules and links form an image-processing pipeline that defines how pixel data move through acquisition, processing, and output stages.

When this tutorial says that a module processes, buffers, blocks, or transfers data at runtime, it refers to the FPGA logic generated from that module. The graphical module in VisualApplets describes that hardware behavior; it does not itself process runtime camera data.

You save the editable design as a ***.vad** file. When you build the design for a specific target frame grabber, VisualApplets produces a binary hardware applet file with extension ***.hap**. You load this file onto the frame grabber. Don't treat the ***.hap** file as an editable design source. You always change the ***.vad** design and build again. At runtime, you can still change **dynamic write parameters** that your application or runtime tools write to the device.

The practical goal is real-time image processing in hardware. While you edit and validate the pipeline as a ***.vad** project on the **PC**, only the built ***.hap** carries that processing logic and data-flow structure onto the FPGA as runnable hardware.

Key Terms

Build means turning your graphical design in VisualApplets into a configuration that the frame grabber FPGA can run.

A **hardware applet**, with file extension ***.hap**, is that runnable result.

A **frame grabber** is an expansion board in the computer that receives the camera data, hosts the FPGA, and runs the applet you built. VisualApplets does not contain the FPGA.

1.3 FPGA Technology

A **field-programmable gate array** (FPGA) is a chip that contains a large number of **programmable logic elements**. Those elements and the on-chip routing can be configured to realize **complex digital circuits** in hardware that perform many operations in parallel.

When you build a VisualApplets design for a target frame grabber, VisualApplets **automatically maps** your operators and links onto FPGA logic. You do not implement that mapping yourself. VisualApplets handles the vendor-specific FPGA toolchain (typically **Xilinx Vivado**) for placement, routing, and bitstream generation. See [Basic Design Concepts](#) for how the ***.vad** project on the PC relates to the ***.hap** that configures the FPGA.

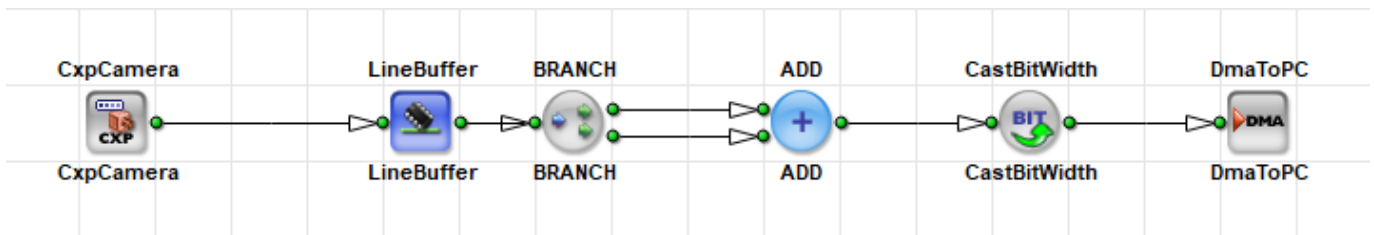
Besides **programmable logic elements**, which you typically use to build combinational circuits, an FPGA provides **flip-flops** for clocked storage, **block RAM** for line buffers and other on-chip memory, and **DSP or arithmetic slices** for fast multiply-add and similar functions. As part of the same automated build flow, VisualApplets maps operators and links onto these resources. Each operator in your design corresponds to concrete hardware such as logic, memory, arithmetic, or a mix.

The **FPGA resources** your design needs depend on how many modules you use, which operators you choose, and how you set parallelism, buffer sizes, and other parameters. Heavier use of logic, memory, or parallel data paths increases consumption. If the design exceeds what the device can implement, the build fails or you may not achieve timing closure. VisualApplets estimates and reports FPGA resource consumption when you run **Design Rules Check Level 2** (DRC Level 2). See [FPGA Resource Estimation](#) in the User Manual and [Design Rules Check](#) in the User Manual. For further explanation of how consumption builds up, which rules apply, and how you trade off design choices, see also the [Parallelism](#) Tutorial topic and the [Memory Elements](#) Tutorial topic.

1.4 Pipeline Execution in Hardware

On the frame grabber, the built hardware applet implements your design as a pipelined hardware path: pixel data move through linked operator stages in a continuous flow.

The pipeline you draw in VisualApplets is a model of the hardware path on the FPGA: The FPGA handles data stage by stage and passes it downstream in a continuous flow. The design doesn't assume that the pipeline stores an entire image at every stage along the path. The generated FPGA pipeline transfers data from stage to stage on clock cycles according to this pipeline model, as described in more detail in the [Pipeline Concept](#) Tutorial topic.



Compared with a typical **CPU** workflow, which often moves large blocks of data through memory and executes instructions largely in sequence, an FPGA can **process and transfer data pixelwise in parallel** along the pipeline. Compared with a **GPU**, which runs many threads over data under program control, an FPGA pipeline is closer to a fixed conveyor. Data moves through processing stages that you wired in the design. This is why documentation uses the **pipeline** metaphor so often.

i Key Terms

- Think of an **FPGA** as a chip on the frame grabber where many small operations run **at the same time** along a path you designed in VisualApplets. The build turns that design into dedicated hardware logic on the FPGA. Pixel data then flow through those hardware stages like items on a conveyor, which is the **pipeline** idea.
- A **CPU** usually works through steps one after another and frequently loads and stores whole chunks of data.
- A **GPU** runs the same kind of operation on many pixels or threads in parallel under a program.

1.4.1 Clock Cycles and Timing

FPGA designs rely on a clock signal that tells each pipeline stage when to capture, transfer, and process data. VisualApplets and this tutorial call that clock the **Base Design Clock**. It is the clock of the generated FPGA pipeline. It is not the same as the camera pixel clock. A *clock cycle* is one tick of the Base Design Clock. On each rising clock edge, flip-flops in the pipeline store their input data and pass it to the next stage.

Clock cycles therefore:

- **Define the timing** of all elements such as registers in the pipeline.
- **Ensure stable data flow**, because values only move to the next stage at predictable clock edges.
- **Determine the maximum processing speed**, since the Base Design Clock frequency must be slow enough for all combinational logic between registers to settle correctly before the next cycle.

In other words, the clock rhythm keeps the entire hardware pipeline aligned. Individual operators are pipelined as well: the build implements each operator with internal register stages so the FPGA stage can accept new input on every clock cycle, even when a single input needs several cycles before its result appears at the output. After the pipeline has filled, throughput at the output can still be one result per cycle while the latency from a given input to its result may be greater than one cycle. That is why pipelining and resource mapping during the build directly influence the achievable Base Design Clock frequency and throughput.

i Info

You find the Base Design Clock frequency and other hardware resource information about your frame grabber in [Device Resources](#).

1.5 Related Topics

- [From Empty Design to Running Applet](#) Tutorial topic — design, DRC, simulation, and build
- [Pipeline Concept](#) Tutorial topic — how data moves through the pipeline
- [Parallelism](#) Tutorial topic — bandwidth, clock, and FPGA resources
- [Memory Elements](#) Tutorial topic — FPGA memory and frame grabber DRAM
- [Device Resources](#) — Base Design Clock and platform resources
- [Design Rules Check](#) in the User Manual — DRC Level 1 and Level 2

2. From Empty Design to Running Applet

This topic walks you through the end-to-end workflow from an empty VisualApplets design to a runnable hardware applet. After you have read it, you know how design, simulation, design rule checks, build, and deployment on a frame grabber fit together when you use hardware. You also know where you set parameters at design time and where you set them at runtime.

Key Terms

You can follow this learning path without frame grabber hardware on your desk. You run design, parameter setup, design rule checks, and simulation in VisualApplets on the PC. **Build** needs the FPGA toolchain (typically Xilinx Vivado) on the PC; it does not need the board. **Flash/load** and **runtime** verification need the frame grabber and a runtime software, e.g. the [Basler Framegrabber SDK](#). For full integration tests you also need the frame grabber and a camera or another image source.

For a step-by-step walkthrough of the same workflow, see [Hardware Applet: From Idea to Application](#) in the User Manual.

For a short introduction to designing, building, and using an applet in microDisplay X, see [Getting Started](#) in the User Manual.

2.1 Workflow Overview

The overall workflow from a VisualApplets design to a running applet is as follows.

In VisualApplets on the PC:

1. Design the applet. Place **operators** as **modules** in the design panel and add **links** between them. Set module and link properties.
2. Run **DRC** Level 1 and 2 to check formal correctness and estimate FPGA resources before you build. DRC Levels 1 and 2 also run automatically when you start simulation or build.
3. Simulate the design with test images. Simulation starts only if DRC Level 1 reports no errors.
4. Build the hardware applet and produce the ***.hap** file. The build uses the FPGA toolchain on the PC. It does not require the frame grabber to be connected.

On the frame grabber and host:

1. For microEnable 5 family platforms, flash the applet to the board. For microEnable 6 / imaFlex boards, load the applet to the hardware.
2. Use the applet on the hardware with microDisplay X, the Framegrabber SDK, or pylon.

This tutorial refers mainly to **microEnable 5** (mE5) and **imaFlex** frame grabbers (CXP-12 Quad/Penta, microEnable 6 family). The operator libraries differ by platform. For example, mE5 designs use `ImageBuffer` for DRAM line buffering, while imaFlex designs use `LineBuffer`.

Key Terms

A **hardware applet** is the ***.hap** file that the build produces and the frame grabber runs. **Design time** is when you edit the ***.vad** project in VisualApplets on the PC. **Runtime** is when host software loads the applet to the frame grabber FPGA and acquires images. Simulation in VisualApplets is not runtime.

2.2 Starting a New Project

You carry out the design phase entirely in VisualApplets. Start the application from **VisualApplets.exe** in the **bin** folder of the installation directory, from the Windows Start menu, or from a desktop shortcut if you created one.

When no project is loaded, the main window shows an empty workspace. To start a project, click **File** and **New**, or press **Ctrl + n**. In the **New Project** dialog, you set the project name, the target hardware platform such as microEnable 5 marathon VCL, and the target runtime such as Win 64.

Info

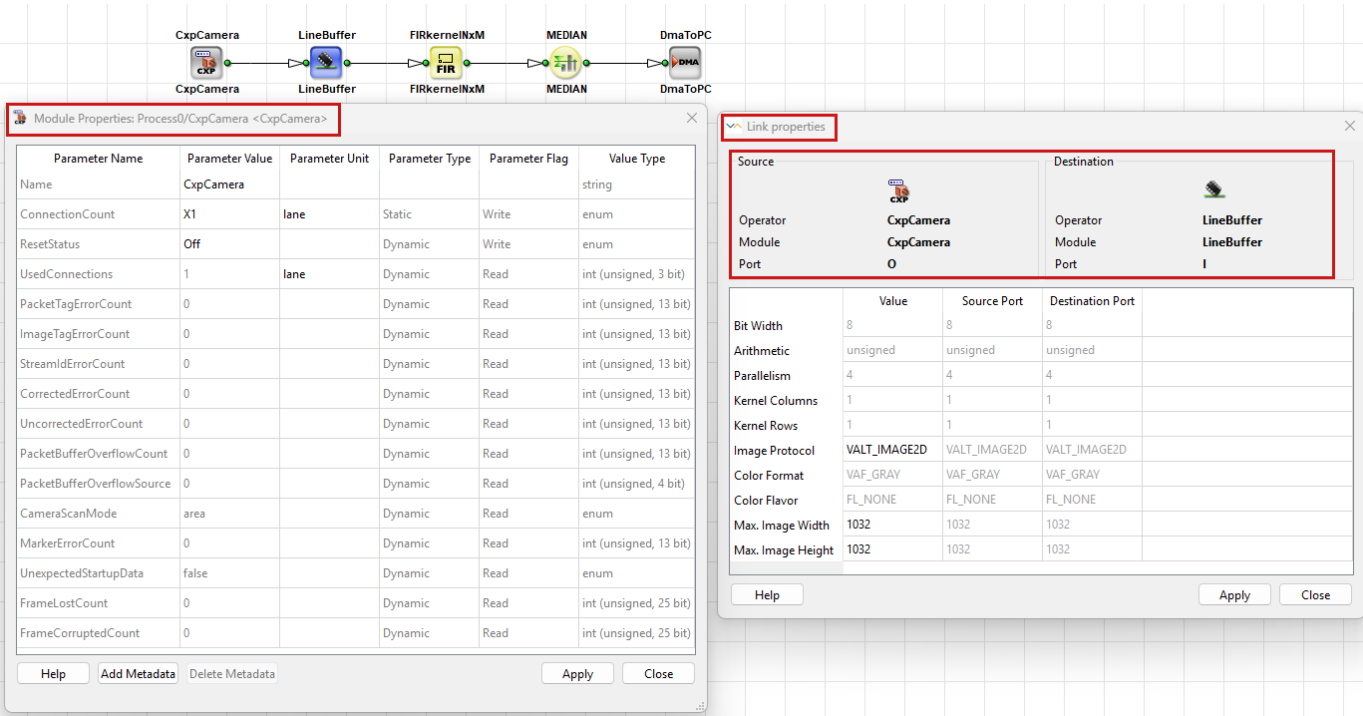
You can change these project settings later, but the platform determines which operator libraries and build options apply, so switching platforms may require you to replace or reparameterize some modules.

After you confirm with **OK**, VisualApplets opens a blank design window. A new design always contains two fixed modules called **AppletProperties** and **BoardStatus**. **AppletProperties** is used to set general properties of the design, such as author and version; **AppletProperties** also provides access to interface properties for certain hardware platforms. **BoardStatus** is used at runtime to provide properties such as PCIe interface status and temperature. The **Project Info** tab summarizes the current project name, hardware target, and platform information.

2.3 Operators, Modules, Links, and the Pipeline

You design the image-processing pipeline in VisualApplets from **operators** that live in the **Operator Library** in the design window. Drag operators into the diagram to place them. Each placed instance is a **module**. Modules expose **input** and **output ports**. Connect an output port of one module to an input port of another to create a **link**. The diagram shows each link as an arrow. Together, modules and links describe the image-processing **pipeline** that the build implements on the FPGA. Processing order follows the connection order.

Operators and links both have **properties** such as dimensions, bit width, protocol, parallelism, and other settings. **Links** carry their own properties along the connection. Which link fields you may edit depends on the **upstream operator**. Non-editable link fields receive their values either from the upstream operator or from the link's position in the chain. Additionally, rules also automatically propagate properties through the pipeline. For example, you set some properties only on the camera output link. They propagate downstream until an operator changes them. Double-click a link or module to open and change its properties. The in-product help and the [Operator Reference](#) document operator-specific behavior.



Basler recommends that you save the design regularly, rename modules when that improves readability, and use **HierarchicalBox** elements to structure larger designs. You can start from example designs under **Examples** in the installation directory. See [Processing Examples](#) in the User Manual for pointers.

2.4 Setting Parameters at Design Time

Link properties describe how image data travels along a connection. They cover settings such as maximum image size, image protocol, bit width, and parallelism. Higher maximum dimensions and higher parallelism typically use more FPGA resources, so Basler recommends that you set these values only as high as your application needs. You edit link properties only in VisualApplets, not at runtime. For a detailed description of link properties, see [Parameterization](#) Tutorial topic.

Operator properties are either **static** or **dynamic**. You change static parameters only in VisualApplets during design time. You can set dynamic parameters in VisualApplets as defaults, and you can change them later from host software while the applet runs on the frame grabber.

2.5 Design Rules Check (DRC)

After you create the design and set parameters, Basler recommends that you run **Design Rules Check Level 1** and **Design Rules Check Level 2**. Select **Analysis** → **Design Rules Check Level 1**, or **Design Rules Check Level 2**, or the matching icons on the build toolbar.

- Starting **Level 2** automatically runs **Level 1** first.
- **Level 1** also runs automatically before **simulation** and **build**. See [Design Rules Check](#) in the User Manual for details.

DRC Level 1 reports formal design problems in the **DRC log** and highlights affected locations in the design pane. Typical issues include mismatching bit widths on links, and ROI or image size inconsistencies. You must clear **Level 1 errors** before you build the hardware applet.

DRC Level 2 runs further checks and **estimates FPGA resource** usage. Treat this estimate as a guideline, not an exact prediction: For some operators, later logic optimization during build can change the final resource usage, so even an

estimation above 100% can still result in a successful build in some cases. If any resource is far above 100%, simplify the design, reduce parallelism or image dimensions, or choose other operators, then run the checks again.

The figure below shows a typical **DRC log** after Design Rules Check Level 1 and 2 without errors: formal status plus estimated use of LUTs, flip-flops, block RAM, and arithmetic resources.

DRC Log

0 Errors
0 Warnings

10:00:23 -- **Design Rules Check Level 1 successfully performed.**

Check Build Preconditions

Creation date: Monday, 11 April 2022 10:00:23

10:00:23 -- **All checks successfully performed.**

Netlist Generation

Creation date: Monday, 11 April 2022 10:00:23

Netlist generation started...

Info: Building design for FPGA clock frequency 125 MHz

FPGA resource estimation:

Resource	Count	Fill Level
LookupTables	37132	~ 36%
Flip Flops	44960	~ 22%
Block RAM	103	~ 15%
Embedded ALU	14	~ 2%

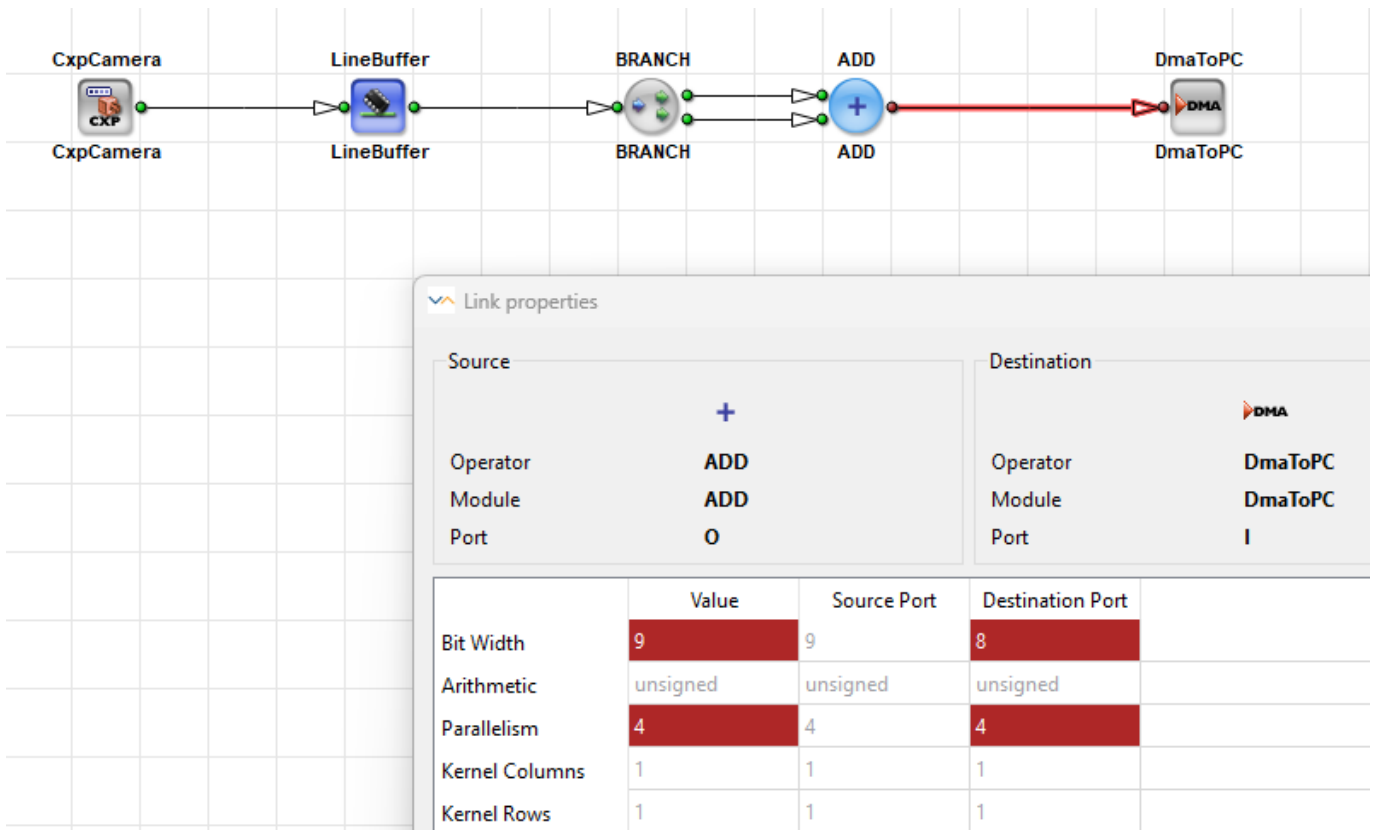
10:00:26 -- **Netlist successfully generated in 00:00:03.**

To see which module uses how many resources, open **Analysis → View FPGA Resources**. The **FPGA Resource Estimation** dialog displays both an estimated breakdown for the **entire design** and for **each module**. You can export the table as CSV. To see only the FPGA resources of a selected module, open the context menu of this module and select **FPGA Resources**. See also [FPGA Resource Estimation](#) in the User Manual.

If the DRC log reports problems, use the DRC messages together with highlighted modules or links to fix the errors. For more detail, see [Design Rules Check](#) in the User Manual.

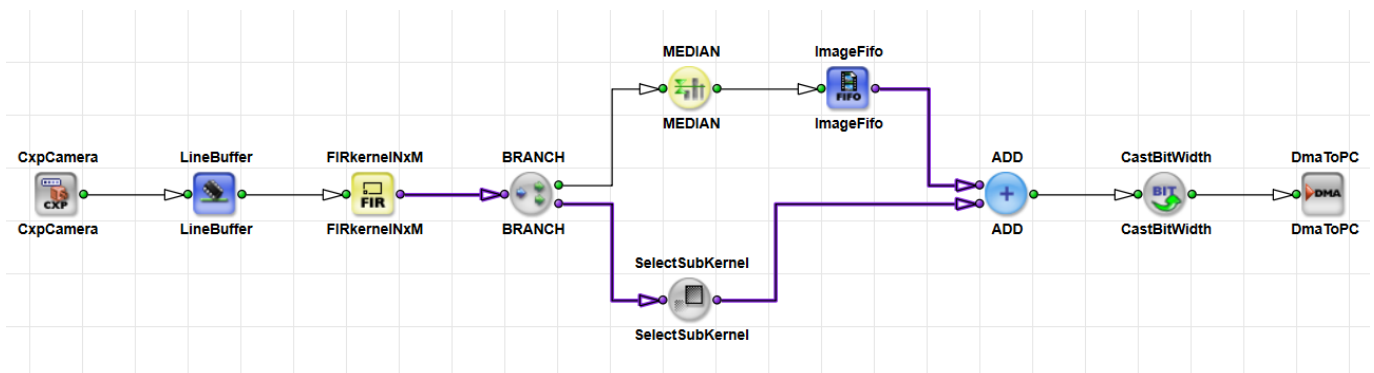
2.5.1 Link Colors

In the design pane, **red** link highlighting marks **formal design errors** on the image path. In **Link Properties** or **Module Properties** dialogs, red-marked links can also provide tooltip messages with details about the specific error type.



Purple highlighting marks **violations of synchronization rules** between branches, such as O-type network rules. You must fix red errors and purple synchronization violations before you deploy the applet to hardware. For processing with signals (i.e., links using `Image Protocol1 = VALT_SIGNAL`), purple links are permitted.

These colors are not a traffic-light status for the whole design. **Green** links, shown after you apply a property change, mean that VisualApplets has propagated and validated the updated link properties. Green does not by itself mean that the design is ready to build. See [Propagation of Link Properties](#) Tutorial topic.



For more details on link rules, see [Rules of Links](#) in the User Manual and [Operator Types and Synchronization](#) Tutorial topic.

2.5.2 Simulation vs. DRC

You can't start simulation if DRC Level 1 reports an **error**. **Warnings**, including synchronization warnings, don't block simulation. **Red** states indicate errors and stop simulation. **Purple** states mark synchronization problems. You may still simulate in that case, but you should fix them before you rely on the applet on hardware.

Even when DRC Level 1 and 2 complete without errors, **simulation** can still fail. For example, simulation fails when the loaded image width isn't an integer multiple of the link **parallelism**, or when the image exceeds maximum image dimensions set on the links. To fix these issues, align the simulation image size with those constraints or adjust the link properties.

The following image shows an **error in DRC Level 1**:

Design Rules Check Level 1

Program version: VisualApplets 3.6.1.74147-44a18e772
 Design name:
 Creation date: Friday, 24 April 2026 10:25:25

Error: 1040 Module (+) ADD (ADD) connection: 0 format error. Link parameter value not accepted at destination.
 Warning: 1310 Module (+) ADD (ADD) node: 1001 is not properly synchronized.

1 Error
 1 Warning

10:25:25 -- Design Rules Check Level 1 failed!

Source		Destination	
Operator	+	Operator	DmaToPC
Module	ADD	Module	DmaToPC
Port	O	Port	I
Bit Width	9	8	
Arithmetic	unsigned	unsigned	
Parallelism	4	4	
Kernel Columns	1	1	
Kernel Rows	1	1	
Image Protocol	VALE_IMAGE2D	VALE_IMAGE2D	
Color Format	VAF_GRAY	VAF_GRAY	
Color Flavor	FL_NONE	FL_NONE	
Max. Image Width	1032	1032	
Max. Image Height	1032	1032	

The following image shows an **error in DRC Level 2**:

Design Rules Check Level 2

Program version: VisualApplets 3.6.1.74147-44a18e772
 Design name:
 Creation date: Friday, 24 April 2026 10:23:58

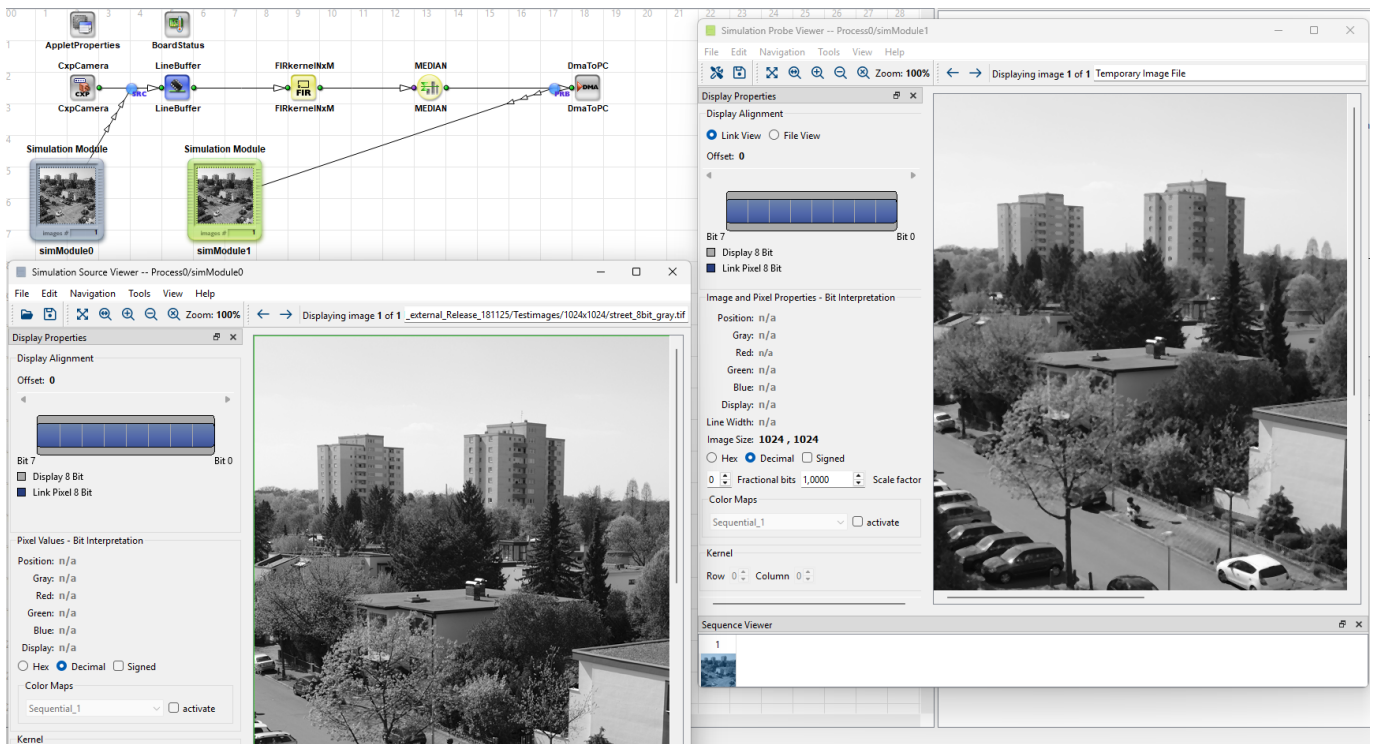
00:00:08 -- Error: -150 Checking output latency: Error at module (+) LineBuffer (LineBuffer). Module cannot buffer data stream / infinite source.
 Warning: 1310 Module (+) ADD (ADD) node: 1001 is not properly synchronized.

1 Error
 1 Warning

10:23:58 -- Design Rules Check Level 2 failed!

2.6 Simulation

During design, **simulation** feeds test images through the pipeline and lets you inspect intermediate results without hardware. Add **simulation sources** to inject images and **simulation probes** to view data on specific links. VisualApplets ships test images under **%VASINSTALLDIR%/testimages**. You can also load your own images into simulation sources. **TIFF** is the preferred format, but VisualApplets supports loading images in many other common formats as well.



When you start simulation from **Analysis** → **Start Simulation** or from the toolbar control, VisualApplets automatically runs **DRC Level 1**. The **Simulation** window lists active sources and probes and shows the DRC Level 1 outcome in the lower pane. You don't need **DRC Level 2** to simulate, but Basler recommends that you run DRC Level 2 before you build the applet so you get the FPGA resource estimate. See [Simulation](#) in the User Manual for all further simulation options, modes, and windows.

You can run **multiple simulation cycles** in one simulation session: In the **Simulation** window, set **Processing cycles** and click **Start**. If the simulation source contains only one image, VisualApplets **reuses** that image for every cycle. If the source contains several images, each new cycle uses the next image in order. After the last image, the sequence starts again from the first.

For the pixel values on image data paths, simulation usually matches hardware for the data under test. In a **Simulation Probe**, you inspect pixels in detail by zooming. Tools such as image histograms can help you with interpreting the pixel data.

Simulation does not show all hardware effects. Deadlocks, undersized buffers, shared DRAM bandwidth limits, and some overflow cases often appear only on the frame grabber. A successful simulation is not a substitute for a runtime test on hardware.

2.7 Building the Hardware Applet

When you **build** your design, VisualApplets translates the design into an FPGA bitstream and packages it together with content for the runtime software interface into a file called *hardware applet*. The build needs a supported AMD/Xilinx toolchain installed and configured on the build computer. Use Vivado according to your platform and VisualApplets version. See [Installing VisualApplets](#) for an overview of which toolchain versions match which frame grabber platform. For detailed instructions on building your hardware applet, see [Build](#) in the User Manual.

Build duration depends on FPGA logic usage, the hardware target platform, and the complexity of the design. Simple examples may finish in minutes, while large designs can take hours. When the build succeeds, the output ***.hap** file typically uses the same base name as the design file. For example, **Sobel_Filter.vad** yields **Sobel_Filter.hap**.

2.8 Flashing, Runtime Loading, and Host Software

For microEnable 5 family boards you must **flash** the new applet with **microDiagnostics** so the firmware partition on the board contains your ***.hap** file. For microEnable 6 boards, you load the applet to your hardware. For instructions on how to flash or load your applet to your frame grabber with the **microDiagnostics** tool of the Framegrabber SDK, see [Managing Applets in microDiagnostics](#).

To work with the built hardware applet, open the applet in **microDisplay X**. The parameter tree in microDisplay X reflects module names from the design, so choose clear module names in VisualApplets. **Dynamic** parameters appear there. You can change some parameters during acquisition, but others must be set before acquisition starts. DMA dimensions are a typical case. You can change other parameters during acquisition, such as a threshold value.

For microDisplay X features and acquisition behavior, see [microDisplay X](#) in the Framegrabber SDK documentation.

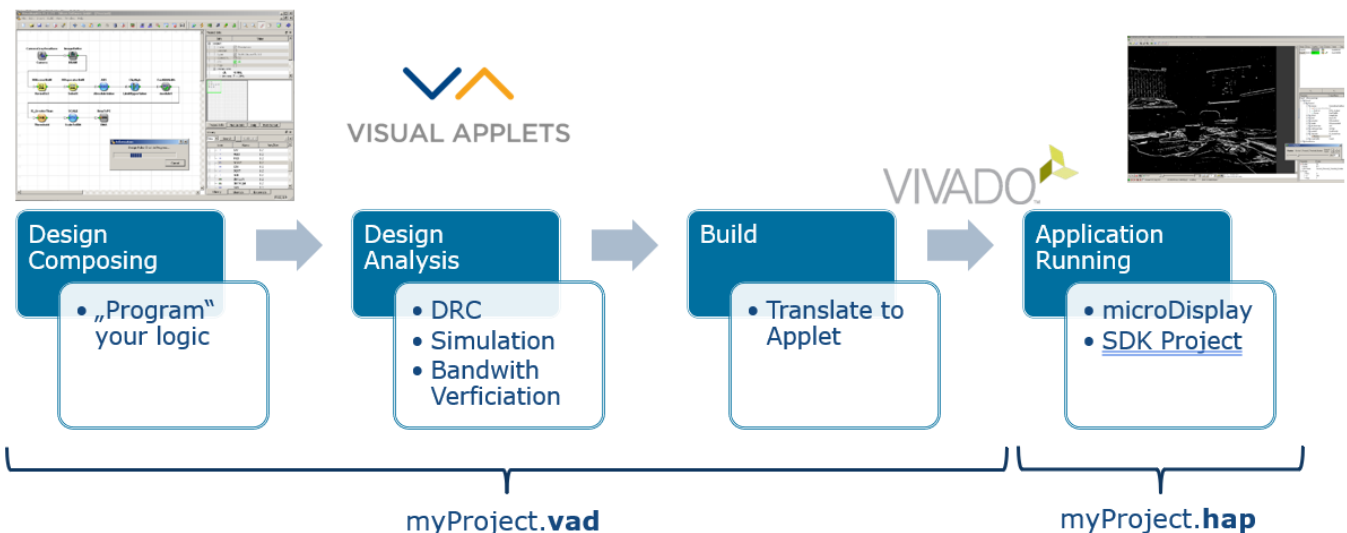
Alternatively, integrate acquisition and parameter control in your own application with the **Framegrabber API**. VisualApplets can generate example C++ projects with **Build** and **Generate SDK Example**. Those projects contain initialization, memory allocation, **Fg_setParameterWithType** / **Fg_getParameterWithType** calls, acquisition start and stop, and cleanup. See [Framegrabber SDK](#) in the User Manual for how to generate an SDK Example.

Treat the generated code as a starting point. Find the API reference in the Framegrabber SDK documentation, for example [basler_fg.h](#) and the [Framegrabber API](#) introduction.

2.9 Design Time and Runtime

At **design time**, you edit the graphical pipeline, set static and dynamic parameters, define link properties, run **DRC** Level 1 and 2, and **simulate**. Every time you start simulation, VisualApplets runs DRC Level 1 automatically. You also generate the build output as a ***.hap** file.

At **runtime**, host software loads that applet, configures **dynamic** parameters, and starts acquisition. The operator names and hierarchy defined in the VisualApplets design are also used in the parameter tree and API. Consistent naming and documentation help integrators and simplify field tuning.



2.10 Related Topics

- [VisualApplets Fundamentals](#) Tutorial topic — design, build, and hardware applet
- [Parameterization](#) Tutorial topic — link properties and module parameters at design time

- [Pipeline Concept](#) Tutorial topic — modules, links, and pipeline structure
- [Operator Types and Synchronization](#) Tutorial topic — operator types and link colors
- [Design Rules Check](#) in the User Manual — DRC Level 1 and Level 2
- [Simulation](#) in the User Manual — simulation sources, probes, and modes
- [Build](#) in the User Manual — generating the hardware applet
- [Installing VisualApplets](#) — Xilinx toolchain versions

3. Pipeline Concept

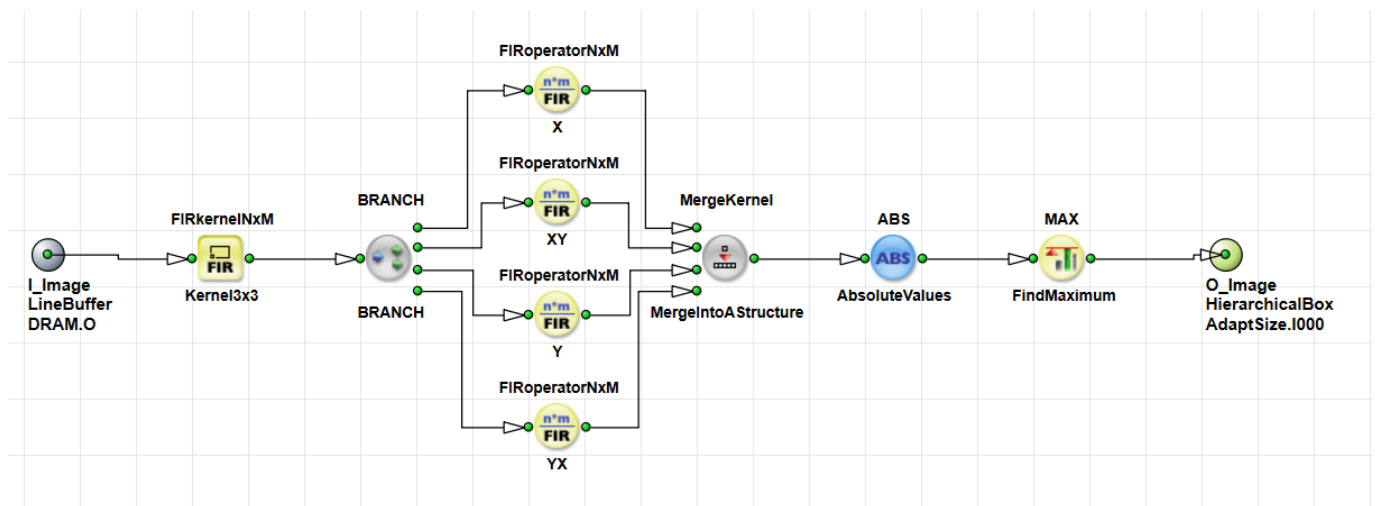
This topic explains how image data moves through the FPGA hardware generated from a VisualApplets design. After you have read this topic, you understand how modules and links in the design form the pipeline structure, how data enter and leave the pipeline, why most stages forward data immediately, how concurrent FPGA processing affects throughput, and which link properties limit the data rate.

For the formal data flow model, see [Data Flow](#) in the User Manual. For how you build and verify a design end to end, see [From Empty Design to Running Applet](#) Tutorial topic.

3.1 Modules, Links, and Pipeline Structure

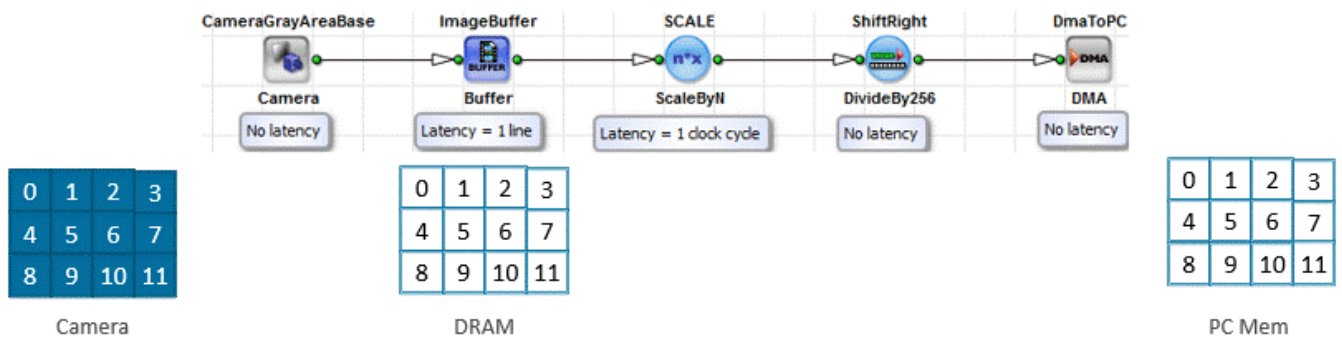
You build a VisualApplets design from **operators** in the operator libraries. Each placed operator instance is a **module**. Modules expose **input** and **output** ports. You connect ports with **links**. The diagram shows each link as an arrow.

Together, modules and links in the VisualApplets design form the image-processing **pipeline**. The build implements that graph as hardware stages on the frame grabber FPGA. Processing order follows the connection order. Branches and merges define parallel paths and reunion points in the graph.



Each module starts processing as soon as input data is available. It forwards results on its output links when the calculation is complete and the output isn't blocked. At runtime, this behavior is implemented by the FPGA logic generated from the module, not by the graphical symbol in VisualApplets. The pipeline structure therefore defines where data splits, where it reunites, and which operators run in series or in parallel.

Pixels don't enter the frame grabber as a complete image in one step. The camera transfers them **one after another** into the acquisition path — or an applet-internal image source generates them **one after another**. Those pixels then pass through the pipeline stages in the same sequential order:



[Double-click here for the animated version.](#)

Key Terms

Think of pixels like cars on a road: they follow each other along the path. At any moment, different cars (pixels) are at different positions on the road (pipeline stages). The road itself is the pipeline structure you wired in the VisualApplets design. After the build, that road is hardware on the frame grabber FPGA, not a path inside the VisualApplets editor.

In this analogy, **parallelism** is the number of lanes on the road: with more lanes, more cars (pixels) can move forward at the same time.

3.2 Pipeline Inputs and Outputs

Every pipeline has at least one **source** that feeds data into the design and one or more **sinks** that leave the FPGA processing path. Sources and sinks are usually M-type operators. Which operators are available depends on your target hardware platform.

3.2.1 Inputs

Data can enter the pipeline in these ways:

- **From a camera or other image source:** Platform-specific camera operators deliver the continuous pixel stream from a connected camera into the acquisition path.
- **Generated inside the applet:** Operators such as `CreateBlankImage` or `CoefficientBuffer` create or play back image data without a live camera. You use these sources for test patterns, reference data uploaded from the host, or designs that don't acquire from a camera.

Signal paths can also start from digital inputs (for example via `GPI`) or from applet-internal signal generators. For how camera sources differ from stoppable applet-internal sources under flow control, see [Stoppable and Non-Stoppable Data Streams](#) Tutorial topic.

3.2.2 Outputs

Processed results can leave the pipeline in these ways:

- **DMA to the host:** The `DmaToPC` operator transfers image data to host PC memory by direct memory access. This is the usual path for acquiring processed images in host software.
- **Signal via GPO:** The `GPO` operator drives a digital output pin on the frame grabber. Use this path for trigger, status, or other hardware control signals outside the host PC.
- **Event to the host:** The `EventToHost` and `EventDataToHost` operators send software events to the host. Host applications receive them through the Framegrabber API and can react to conditions in the applet. `EventToHost` raises an event on a rising edge at a signal input and carries no payload. `EventDataToHost` raises an event together with a configurable data payload.
- **Register information only:** Some results stay on the frame grabber as readable values. Operators such as `GetStatus` store values that host software reads as dynamic parameters. No image stream or DMA transfer is required when the application only needs status or measurement numbers.

A design can combine several of these outputs. For example, one branch can send images through `DmaToPC` while another branch asserts a `GPO` pin or raises a host event when a condition occurs.

For operator details, see [DmaToPC](#) in the Operator Reference, [GPO](#) in the Operator Reference, [EventToHost](#) in the Operator Reference, [EventDataToHost](#) in the Operator Reference, and [GetStatus](#) in the Operator Reference.

3.3 Unbuffered Pipeline Flow

Most modules don't store a full image internally. They calculate output values and forward them immediately when the information is available.

Not all operators output a result as soon as data arrives at the input. **M-type** operators in particular can delay the output. These aren't only buffer operators such as `LineBuffer` or `ImageFifo` — many filter and synchronization operators are M-type as well. How large the latency is depends on the operator. For details, see the [Operator Types and Synchronization](#) Tutorial topic and the [Latency](#) Tutorial topic.

Buffer operators store data until downstream modules can accept it.

This **unbuffered** behavior keeps **latency** low. A pixel can reach the output while other pixels of the same frame are still at the camera input. Acquisition, processing, and DMA can overlap in time.

Key Terms

Think of the generated FPGA pipeline like a system of pipes with branches and valves. Water flows through the pipes as soon as space is available. A buffer operator is like a small reservoir that fills when the drain can't keep up.

For how latency builds up along a path, and for the difference between bandwidth and latency, see [Latency](#) Tutorial topic.

3.4 Concurrent Processing in Hardware

On the frame grabber FPGA, all pipeline stages run **at the same time**. This differs from a typical CPU program that executes instructions largely one after another.

While one FPGA module processes a pixel, other modules process different pixels farther along the path. After the pipeline has filled, multiple modules work on the same image concurrently. The FPGA implements your VisualApplets design as dedicated hardware whose structure matches the pipeline graph.

Key Terms

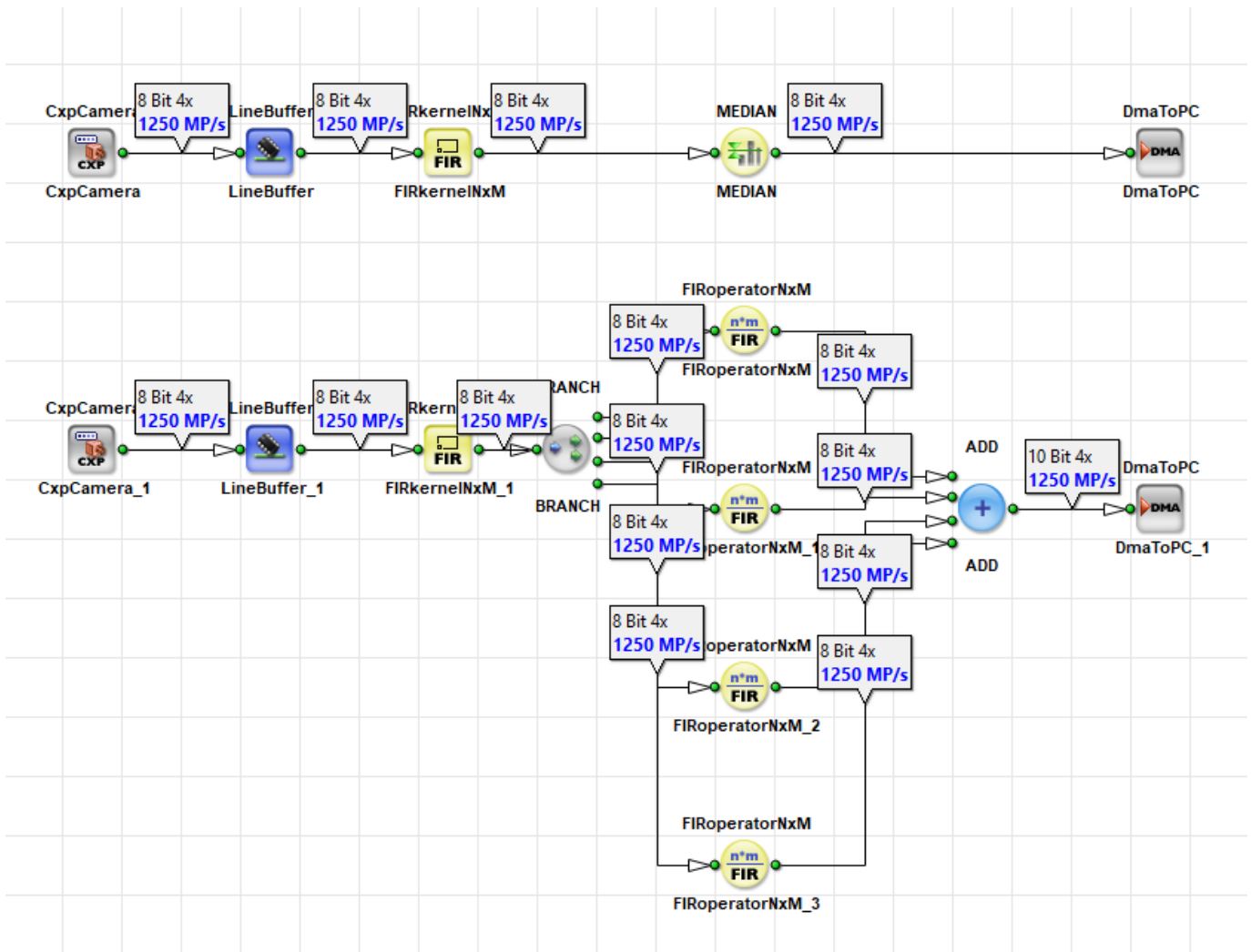
In the FPGA hardware generated from a VisualApplets design, the processing path is implemented as dedicated hardware logic on the frame grabber. Data moves through those hardware stages like items on a conveyor. A CPU usually loads whole chunks of data and works through steps in sequence.

For details about how the built applet relates to the editable ***.vad** design, see [VisualApplets Fundamentals](#) Tutorial topic.

3.5 Throughput and Bottlenecks

Throughput is the sustained data rate through the pipeline. In a VisualApplets design it is closely related to **bandwidth** on the links of the generated FPGA hardware.

The number of operators in a design doesn't reduce the throughput. All generated FPGA stages can run in parallel. The number of parallel branches doesn't reduce throughput either. Branches add hardware resources, but they don't slow the data rate on an individual path by themselves. Two acquisition paths can use a **different number of operators** and still deliver the **same throughput**, as you can see in the following two designs:



Throughput depends on these factors:

- **Link capacity:** How much data a connection between modules can carry. Capacity is set by the **parameterization of modules and links**, especially **parallelism**, **bit width**, and **maximum image dimensions**, together with the Base Design Clock. See the [Parameterization](#) Tutorial topic for details about link and module properties.
- **Downstream readiness:** Whether modules can accept data **continuously**. If a module can't take data permanently, for example because a buffer is full or a successor blocks, flow control stops the stream, which leads to **blocking** and/or **backpressure**.
- **Operator limits:** Some operators can't process the full bandwidth that their input link provides. The [Operator Reference](#) documents these limits for each affected operator.

The **bandwidth** of a link as a pixel rate (pixels per second) follows this formula:

Bandwidth = parallelism × Base Design Clock

This pixel-rate formula does not include bit width or kernel size. A wider bit width or a larger kernel increases the **amount of data per clock** (bits or values), which matters for DRAM access and FPGA resources, but it does not change this pixel-rate number. For DRAM data width, see the [Memory Elements](#) Tutorial topic. For kernels versus pixel-rate bandwidth, see the [Working with Kernels](#) Tutorial topic.

The Base Design Clock depends on your frame grabber platform. Look up the value in [Device Resources](#).

Maximum image dimensions also don't appear in this pixel-rate formula, but they still shape the design: they constrain which image sizes the pipeline can handle. Together with parallelism and bit width, these link properties must match your camera or line scan source.

The **bottleneck** of a path is the **lowest** bandwidth along that path. The overall throughput of the design is always limited by this narrowest point. One narrow link or one bandwidth-limited operator limits the throughput for everything upstream of it on the same path. Throughput is therefore set by the narrowest bandwidth on the path and by whether downstream modules accept data continuously.

Design Guideline

Configure parallelism and link properties only as high as needed to support the rate required by your camera or line scan source. Increasing these values further consumes additional FPGA resources and does not improve throughput if another component is already the bottleneck.

Some operators change the parallelism between their input and output. Others reduce the number of pixels, for example by cropping an image. As a result, the required output bandwidth can differ from the input bandwidth. Where possible, operators automatically adapt the parallelism. If adaptation is not possible, VisualApplets reports a **Design Rules Check** error.

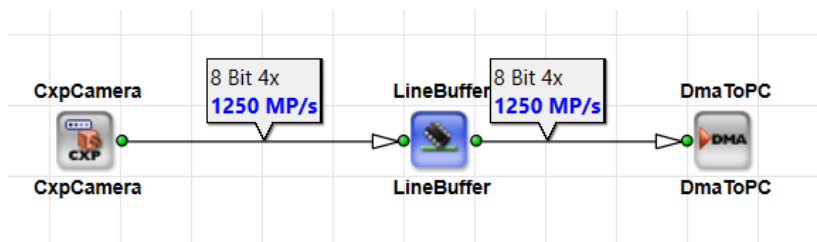
For information about calculating and adjusting parallelism, including examples of how to calculate the required bandwidth for your design and how to troubleshoot bandwidth bottlenecks when operators merge streams, see the [Parallelism](#) Tutorial topic and [Troubleshooting Bottlenecks](#) Tutorial topic.

3.5.1 Visualizing Throughput in the Design

You can make the parallelism and the data throughput visible in your design by using the **Link Info** and the **Throughput** flags. Activate these flags with the following buttons from the toolbar:



As a result, the **Link Info** and **Throughput** flags are displayed in your design:



Compare the values along a path to find the narrowest link. That link is the bottleneck for the path unless an operator further downstream can't keep up.

Info

The **Throughput** flags don't show all performance limitations. For example, when multiple DRAM operators share the same memory interface, the resulting DRAM bandwidth limit from **Shared Memory** isn't reflected in the throughput display. Other **DRAM inefficiencies**, as well as blocking and potential **deadlocks**, are also not shown. For guidance on Shared Memory usage and DRAM bandwidth planning, see the [Memory Elements](#) Tutorial topic. For details on deadlocks, see the [Operator Types and Synchronization](#) Tutorial topic.

3.6 Link Properties and Image Formats

Throughput and data format depend on **link properties** and **module properties**. Besides the capacity-related properties covered above (**parallelism**, **bit width**, and **maximum image dimensions**), other typical properties include image protocol and color format.

Link properties **propagate** through the pipeline. You set many properties on the camera output link or on an early processing stage. They apply to all downstream links until an operator changes them. Module parameters must match the link properties on the connected links. Changes on an upstream link therefore affect all following operators until another operator resets a property.

For the full list of link properties and propagation rules, see the [Parameterization](#) Tutorial topic and [Rules of Links](#) in the User Manual.

The **image protocol** defines the structure of the data stream on a link. VisualApplets supports different data types / stream formats:

Protocol	Typical use
<code>VALT_IMAGE2D</code>	Two-dimensional images with end-of-line and end-of-frame markers, for example from area scan cameras. Lines within a frame can have different lengths.
<code>VALT_LINE1D</code>	Endless or continuous line data, for example from line scan cameras.
<code>VALT_PIXEL0D</code>	Unstructured continuous pixel streams without image dimension information.
<code>VALT_SIGNAL</code>	Single-bit signals valid at every clock cycle, for example in trigger logic.

Pixels move through the pipeline in scan order: left to right within a line, then top to bottom within a frame. Some camera sources use a different order. Use the appropriate operators to correct the order when your source doesn't match this convention.

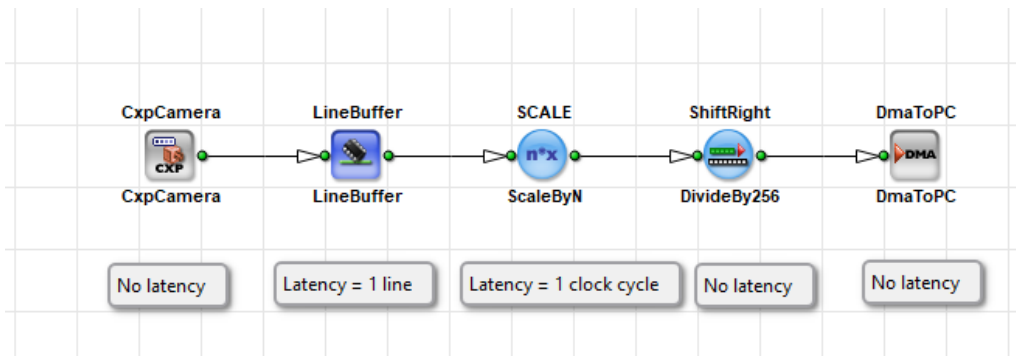
For more information about image protocols and pixel order, see [Image Protocols, Image Dimensions and Data Structure](#) in the User Manual.

Whether a source can pause under flow control affects how you wire the pipeline. Camera operators are **non-stoppable** sources. You often place a buffer operator directly after them. For details, see [Flow Control, Blocking, and Backpressure](#) below and the [Stoppable and Non-Stoppable Data Streams](#) Tutorial topic.

3.7 Flow Control, Blocking, and Backpressure

The generated FPGA pipeline applies **flow control** between stages. When a stage isn't ready to accept more data, it can **block its input**: it stops requesting data from the upstream stage, and the upstream stage stops outputting. That blocking propagates backward through the pipeline as **backpressure**. Note that only M-type operators can block the input. The graphical modules in VisualApplets describe this hardware behavior; they do not apply flow control themselves.

Suppose `DmaToPC` can't transfer data to the host computer fast enough. It blocks its input from time to time. The blocking signal propagates upstream to a buffer operator. The buffer stops forwarding data while blocking is active. It stores incoming pixels and its fill level rises. When the camera stops sending data, the buffer drains again.



Buffer operators decouple stages that process at different rates. Without a buffer after a non-stoppable camera source, blocking by downstream modules can't slow down the camera stream. As a result, the design may fail Design Rules Check or deadlock on hardware.

For operator types and how blocking behaves on O-type, M-type, and P-type modules, see the [Operator Types and Synchronization](#) Tutorial topic.

i Key Terms

Blocking means a generated FPGA stage can't accept further input data, for example because the successor doesn't accept data or because processing inside the stage takes more time. The module icon in VisualApplets represents that stage; the icon itself does not wait or block.

Backpressure is the mechanism behind that: the "pressure" acts backward. If one stage doesn't take data fast enough, upstream stages must wait and the flow is blocked. In practice: A downstream stage can't accept data (for example, because a buffer is full), hence the upstream stage waits.

Bandwidth in this topic is the pixel rate of a connection: how many pixels per second fit through the link. Bit width and kernel size change how much data each of those pixels carries, which is a different quantity. See [Parallelism](#) Tutorial topic.

3.8 Related Topics

- [Data Flow](#) in the User Manual — formal pipeline data-flow model
- [VisualApplets Fundamentals](#) Tutorial topic — design versus hardware applet
- [From Empty Design to Running Applet](#) Tutorial topic — design, DRC, simulation, and build
- [Parallelism](#) Tutorial topic — bandwidth and parallelism
- [Latency](#) Tutorial topic — delay along the pipeline
- [Parameterization](#) Tutorial topic — link properties and image formats
- [Stoppable and Non-Stoppable Data Streams](#) Tutorial topic — flow control and camera sources
- [Operator Types and Synchronization](#) Tutorial topic — operator types and blocking
- [Operator Reference](#)

4. Operator Types and Synchronization

This topic explains how VisualApplets classifies operators and how you synchronize data streams when branches merge. After you have read this topic, you understand O-type, M-type, and P-type behavior, the O-type network rule, synchronous and asynchronous inputs, and when you need operators such as `SYNC`, `InsertLine`, `InsertImage`, or `SourceSelector` to synchronize data streams.

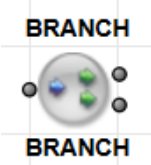
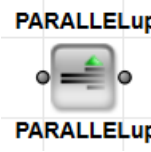
Key Terms

VisualApplets marks each operator with a shape that shows its type. **Round** icons are **O-type** operators. **Square** icons are **M-type** or **P-type** operators. M-type and P-type both use square icons, so you cannot tell them apart by shape alone. Check the in-product help or the [Operator Reference](#) for the type. The main difference: M-type operators can **block** their input; P-type operators do not actively block.

Blocking means a generated FPGA stage can pause its input when it isn't ready to accept more data. You can't connect two independent camera streams with a normal operator, because their timing differs. Use a synchronization operator with a buffer to synchronize the data streams.

4.1 Recognizing Operator Types

VisualApplets groups operators into three types. You recognize the type by the module icon shape in the design pane. The in-product help and the [Operator Reference](#) document the type of each operator.

Type	Shape	Main behavior
O-type	Round: 	• Simple processing • Pixel count unchanged • Chain in series or parallel
M-type	Square: 	• Complex processing • Can change pixel count and dimensions • Can block the input
P-type	Square: 	• Similar to M-type but doesn't actively block the input

Every source operator, such as a camera module, is an M-type operator. Every final destination, such as `DmaToPC`, is an M-type operator.

4.2 O-Type Operators

O-type operators perform simple processing steps. They follow these rules:

- The number of pixels on the output equals the number of pixels on the input.
- There is a delay of a constant number of clock cycles between input and output. This timing delay must be compensated for during timing synchronization.
- You can assemble them in series or in parallel without restrictions. Splitting and merging O-type pipelines doesn't require any additional synchronization.
- They can't block the data stream themselves, but they propagate blocking conditions from downstream modules.

Info

Basler recommends that you prefer O-type operators where they fit your algorithm. They typically use fewer FPGA resources than M-type operators.

4.3 M-Type Operators

M-type operators perform more complex processing. They follow these rules:

- They can change the number of pixels between input and output.
- They can delay the output.
- They can block the input when they aren't ready to accept more data.
- You must follow the link rules when you connect them.
- Merging data streams originating from different M-type sources is only possible using dedicated operators with asynchronous inputs (e.g., `SYNC`, or `SourceSelector`).

Camera operators and buffer operators are always M-type operators. Many filter and synchronization operators are M-type as well.

4.4 P-Type Operators

P-type operators are similar to M-type operators. The main difference is that they **don't actively block the input**. They also don't reduce bandwidth on the link and don't generate more data than they receive. They can be used with non-stoppable sources.

Key Terms

A **non-stoppable** stream runs continuously. A non-stoppable source cannot be paused by the flow-control logic of the generated FPGA design. A typical example is a **camera** that keeps sending pixels. As a contrast, a **stopable** stream can pause when the next operator isn't ready. A **buffer** between camera and downstream processing often turns a non-stoppable stream into a stopable one.

For details on stopable and non-stoppable data streams, see [Stoppable and Non-Stoppable Data Streams](#) Tutorial topic.

Unless an exception is specified in the [Operator Reference](#), the same link rules apply to both P-type and M-type operators. However, unlike M-type operators, P-type operators can be used with non-stoppable sources.

4.5 Synchronous and Asynchronous Inputs

M-type operators with multiple inputs accept **synchronous** inputs, **asynchronous** inputs, or both. Some operators provide groups of synchronous inputs which may be asynchronous to each other. The in-product help and the [Operator Reference](#) document which input type each port requires.

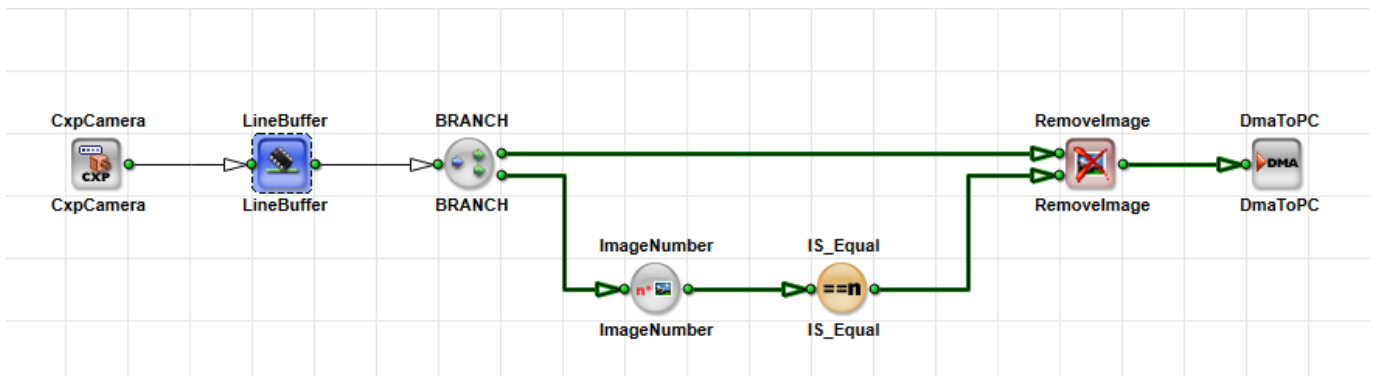
4.5.1 Synchronous Inputs

Many operators require synchronous inputs on all or on a subset of their input ports. Synchronous means that the data streams on these inputs are aligned: The protocol, the number of ingoing pixels, and the timing of the data must all match exactly.

Synchronous behavior is guaranteed when all inputs originate from a common M-type source that is routed exclusively through a network of O-type operators. These inputs may be connected through any number of O-type operators, as long as no other source (i.e. M-type operator) is introduced.

All inputs that belong to the same synchronous group always share a single M-type source.

A typical example is `RemoveImage` with two image inputs from the same `LineBuffer` :



4.5.2 Asynchronous Inputs

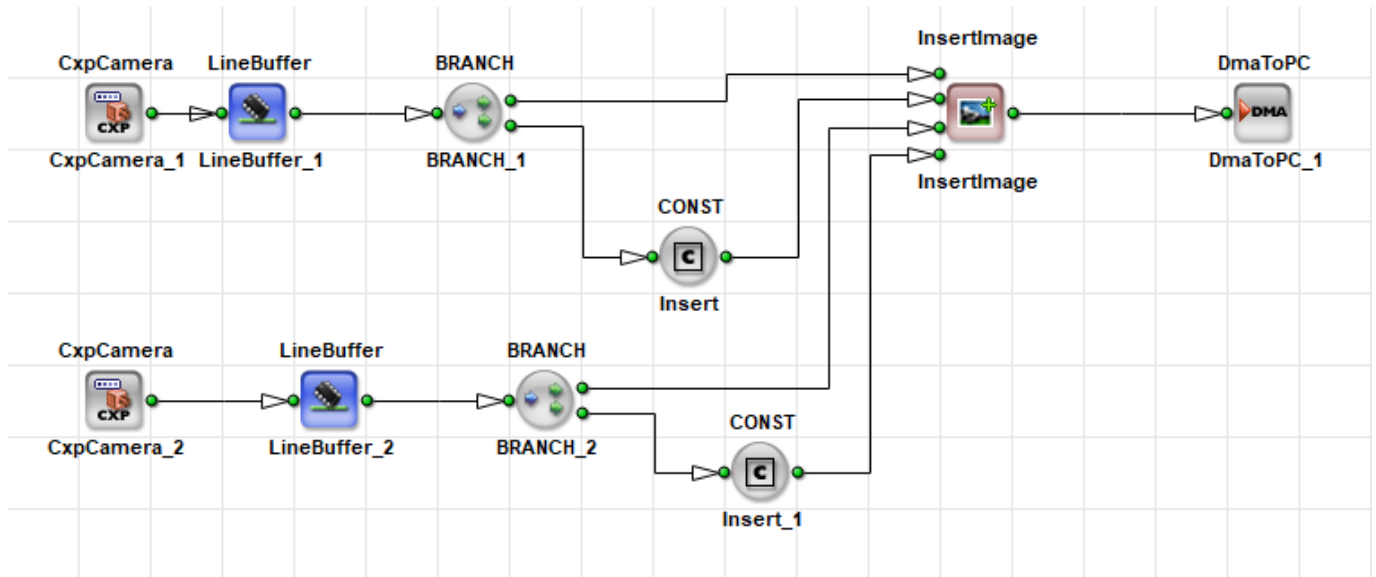
Only a small number of operators support asynchronous inputs. Asynchronous means that the data streams on these inputs are not aligned: The timing, protocol, and number of incoming pixels may differ. The specific properties that may differ between asynchronous inputs depend on the operator. Asynchronous inputs may also be connected through O-type operator networks from their respective sources. Use asynchronous inputs only with operators that explicitly support them. These operators convert asynchronous data streams into synchronous data streams.

Operators that support asynchronous image inputs include:

- `SYNC`
- `InsertLine`
- `InsertImage`
- `SourceSelector`.

`FrameBufferRandomRead` and `FrameBufferMultiRoiDyn` also accept asynchronous image links, but their coordinate input must still be synchronous.

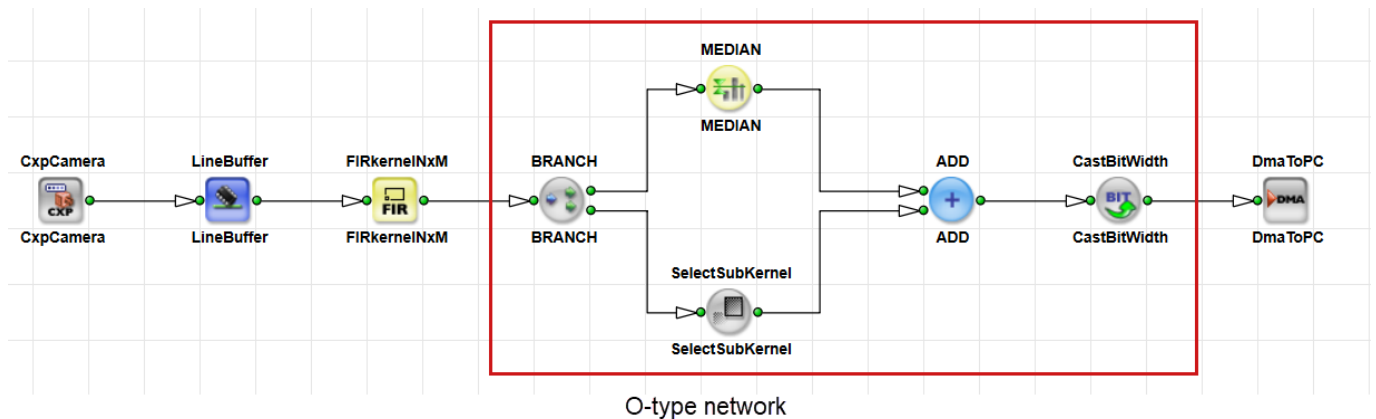
You can't merge two independent M-type sources with a normal multi-input operator. For this, you need an operator that supports asynchronous inputs:



4.6 Synchronization Rules

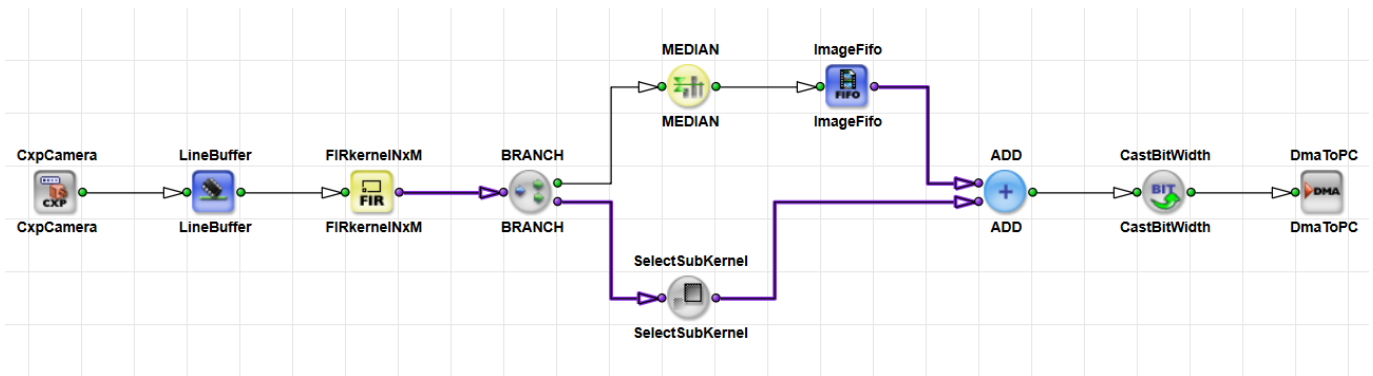
In an O-type network, all O-type modules, regardless of network complexity, must be sourced from the same M-type or P-type module. Links from this source can pass through other O-type modules or O-type subnetworks.

Splitting and merging O-type pipelines is straightforward and doesn't require additional synchronization.



When an M-type or P-type operator is introduced into the network, it becomes a new data source. If branches that originate from different sources are merged downstream, the merge requires synchronization, i.e., you must add an operator with asynchronous inputs, such as `SYNC`, `InsertLine`, `InsertImage`, or `SourceSelector`. These branches can't be connected to standard multi-input operators.

VisualApplets highlights **synchronization violations** with **purple links** in the design pane:



This **O-type network rule** ensures synchronous inputs: All branches that belong to the same synchronous input group must originate from a single M-type or P-type source and may only pass through O-type operators.

The O-type network rule doesn't apply to signal links that use `Image Protocol = VALT_SIGNAL`. For more information, see [Differing Rules for Signal Links](#) in the User Manual.

4.7 Merging Independent Sources

When two or more independent sources feed one processing path, you must synchronize **timing** and **image dimensions**.

- **Timing synchronization** aligns when pixels arrive on each branch.
- **Image dimension synchronization** aligns width, height, and image protocol.

Synchronization is typically done using the operator `SYNC`. For certain situations there are a few other options for synchronization.

Incorrect timing synchronization can cause deadlocks, insufficient bandwidth, or data loss.

Info

To avoid deadlocks, place a memory operator, such as `ImageFifo` or `ImageBuffer`, before the `SYNC` operator when sources run at different speeds or when one branch adds delay. Memory elements must also be sized appropriately: You need to know the expected delay in order to choose a sufficient buffer size.

Deadlocks can't reliably be detected with simulation in VisualApplets, but appear only in hardware. To detect them, watch the `FillLevel` parameter of the `ImageFifo` operator or other buffer operators **on the frame grabber**, for example in **microDisplay X**. A FIFO that stays at 100% often indicates the source of a deadlock. You cannot observe that fill level in the VisualApplets design pane.

For more details on deadlocks, see [Troubleshooting: Deadlock](#)

When you merge asynchronous streams, ensure that line rate, frame rate, and pixel rate match. If one camera runs faster than another, a buffer fills until it overflows and data is lost. Use a trigger system when you need long acquisitions from multiple cameras at the same rate.

4.8 Synchronizing with the SYNC Operator

The `SYNC` operator synchronizes timing and image dimensions on all input links. The number of inputs is fixed when you instantiate the operator. All outputs are synchronous. Downstream O-type operators can then process the merged streams.

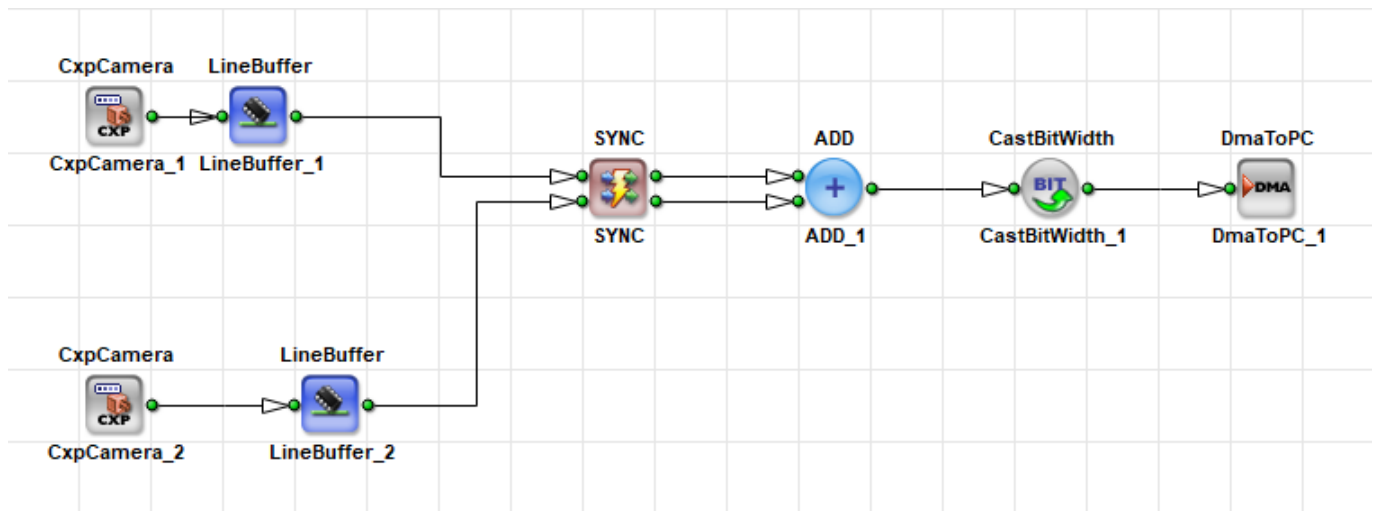
The `SYNC` operator acts like a valve on all inputs. It waits until a valid pixel is present on every input. Then it forwards pixels on all outputs at the same time.

When input images differ in size, `SYNC` aligns them. Depending on the parameterization, it expands the smaller image or crops the larger one. See the operator description [SYNC](#) in the Operator Reference and section [Using the SyncToMin and SyncToMax Parameter Values](#) for **SyncToMin** and **SyncToMax** behavior.

Info

Use the `SYNC` operator near the camera in a design only when the cameras are triggered synchronously. Otherwise the buffers may fill up quickly.

A typical two-camera path looks like this:



The buffers absorb timing skew between the cameras. Without buffers and without `SYNC`, the design risks deadlock or DRC warnings.

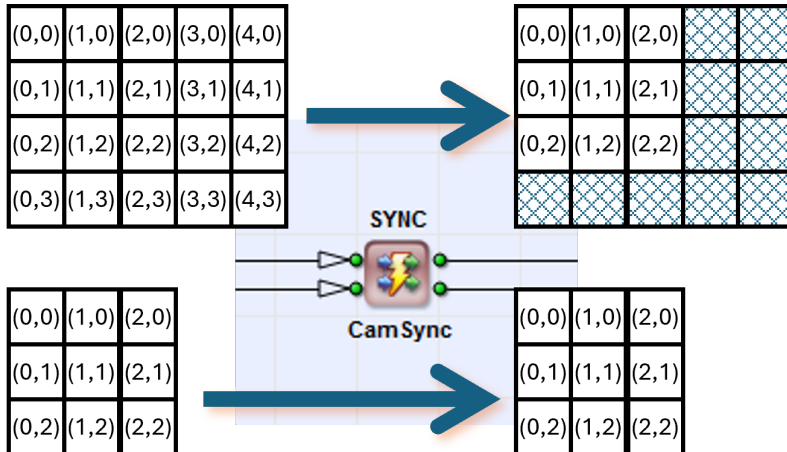
For a step-by-step overlay and stitching walkthrough, see [Synchronization of Asynchronous Image Pipelines](#) in the User Manual.

4.8.1 Using the SyncToMin and SyncToMax Parameter Values

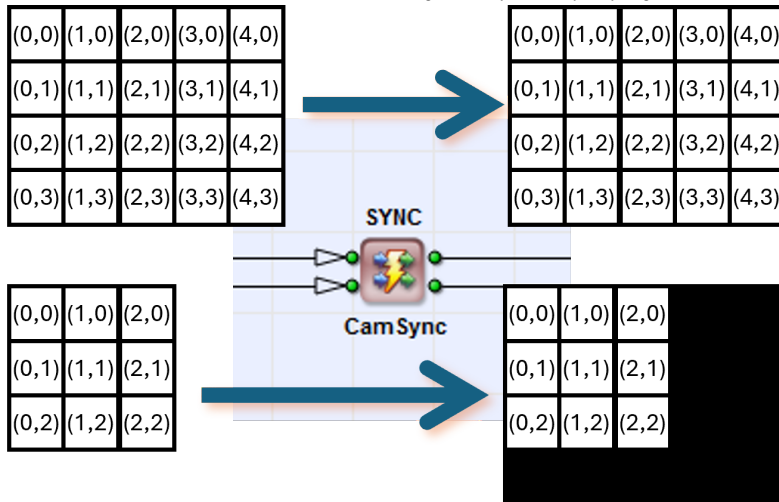
If sources contain **images of different sizes**, use the `SyncMode` parameter of the `SYNC` operator to align their dimensions. You can choose between the values `SyncToMin` and `SyncToMax`.

The synchronization ensures that all output images have the same size. The **SYNC** operator supports two modes:

- **SyncToMin**: Aligns all images to the smallest image size by cropping larger images.

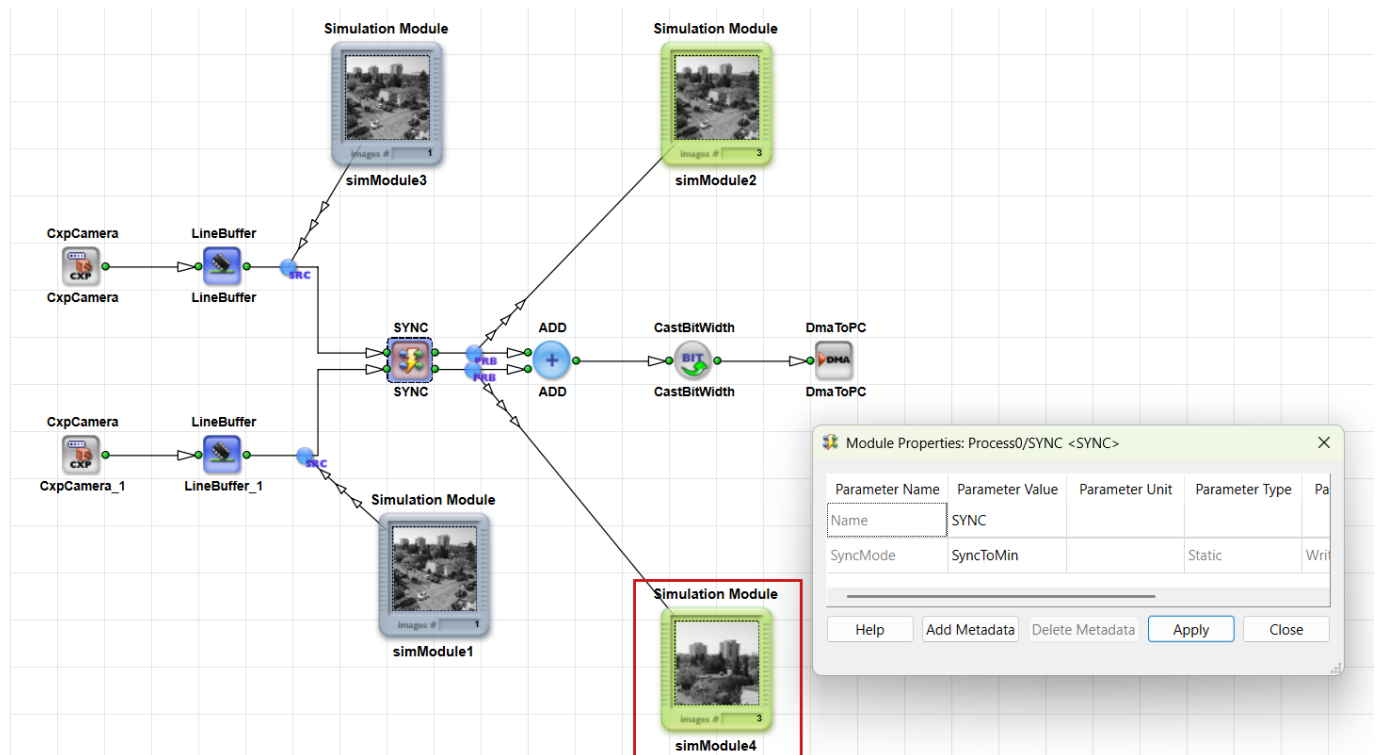


- **SyncToMax**: Aligns all images to the largest image size by expanding smaller images. In **SyncToMax** mode, any additional areas are filled with dummy zero pixels (displayed as black).

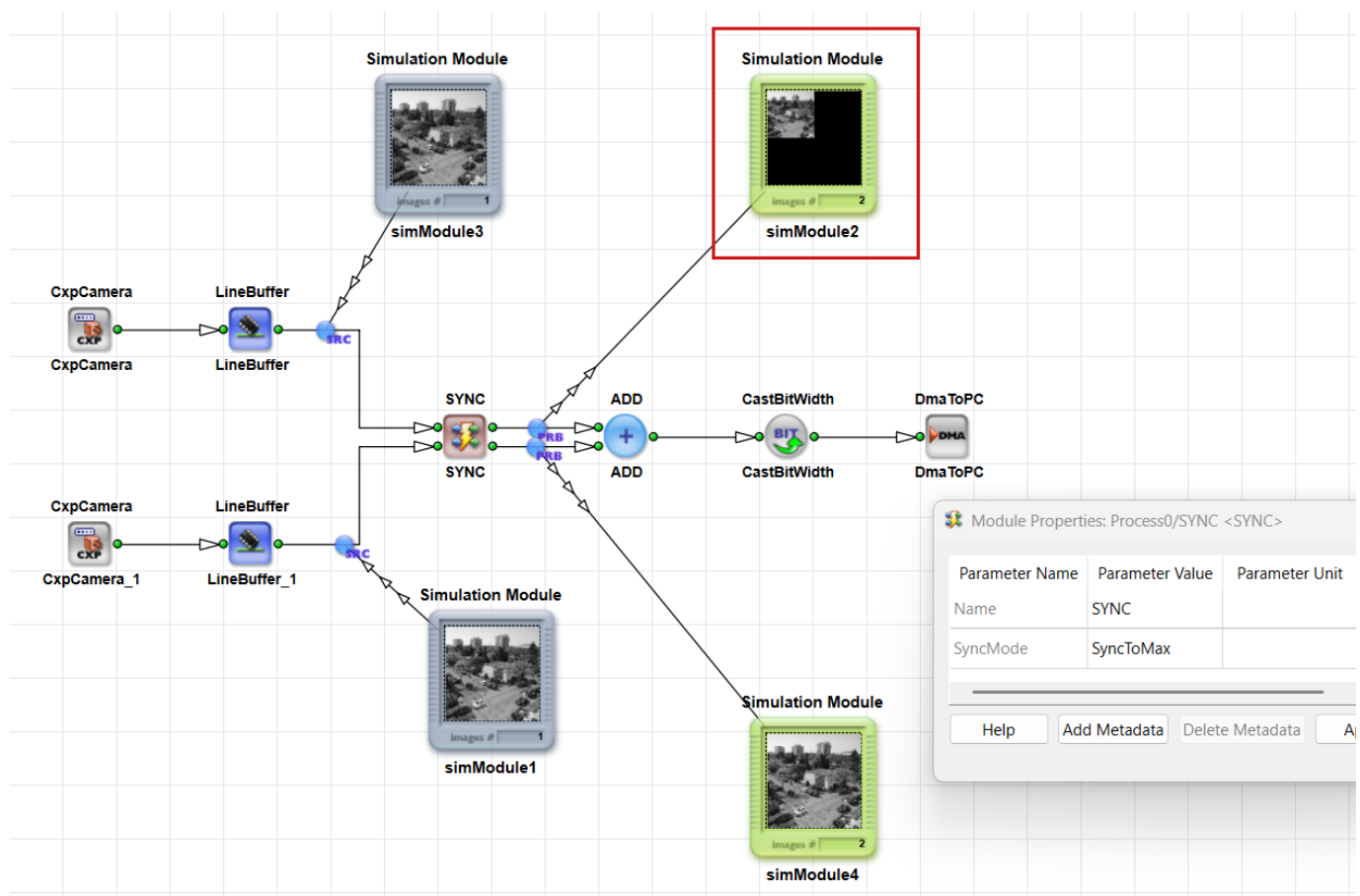


In the simulation, this looks as follows:

SyncToMin: The larger image is cropped.



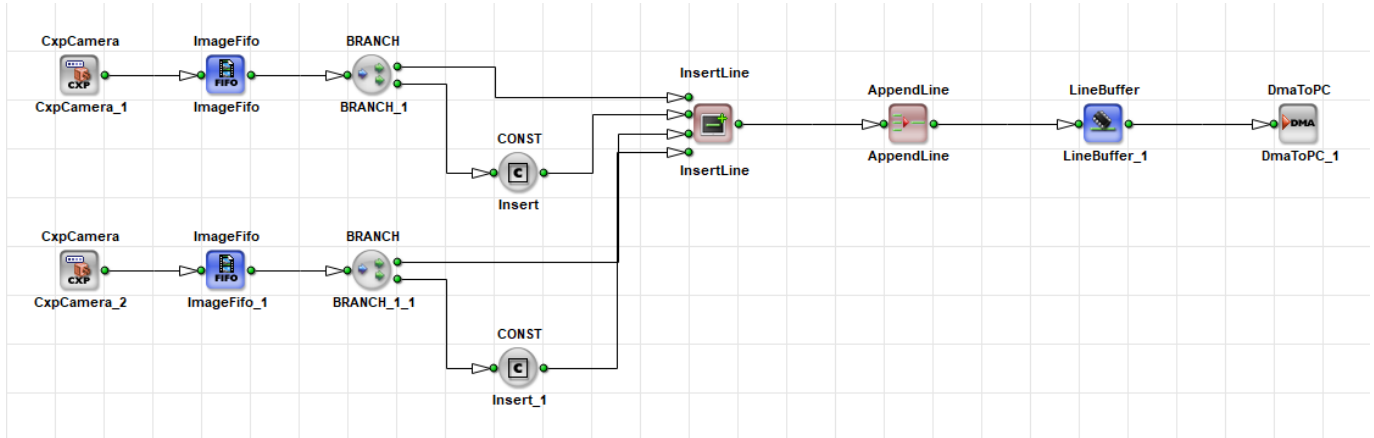
SyncToMax: The smaller image is filled with dummy pixels.



4.9 Synchronizing with the InsertLine and InsertImage Operators

InsertLine assembles an image by inserting lines from several sources. Use it for line-based stitching when both sources deliver lines in sequence.

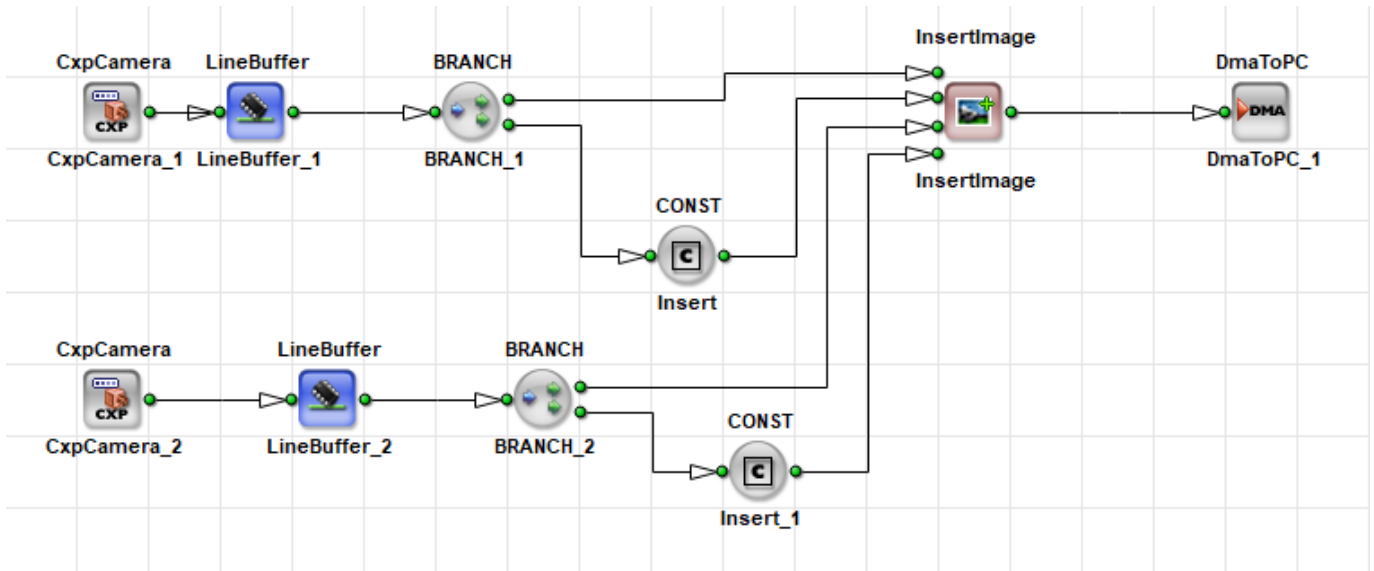
When the cameras are triggered synchronously, you can use a smaller **ImageFifo** operator instead of a **LineBuffer** operator. This reduces resource usage.



Info

InsertLine and **InsertImage** are often used together with the **CONST** operator. In this context, **CONST** specifies whether the image or line is included in the output stream. The data is always processed, but **CONST** determines whether it appears in the output stream or is discarded internally (i.e., sent to an internal trash path).

InsertImage assembles a sequence of images by inserting images from several sources. Use it to multiplex images from two cameras into one output sequence.



Both operators accept asynchronous inputs. If these operators are used with non-stoppable sources, place buffers with sufficient size before them to avoid a deadlock. See also [InsertLine](#) in the Operator Reference and [InsertImage](#) in the Operator Reference.

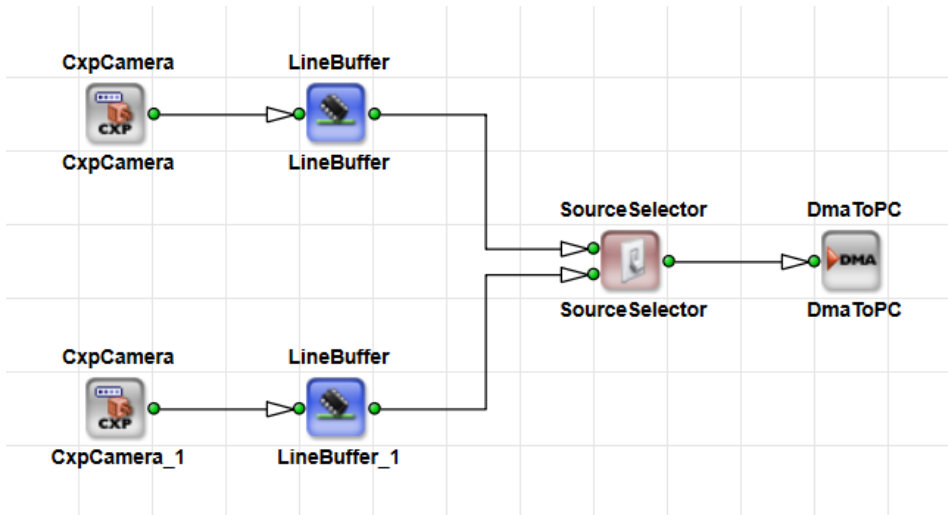
4.10 Synchronizing with the SourceSelector Operator

`SourceSelector` selects between multiple image sources, for example two cameras. It switches between asynchronous sources while maintaining the integrity of the incoming images.

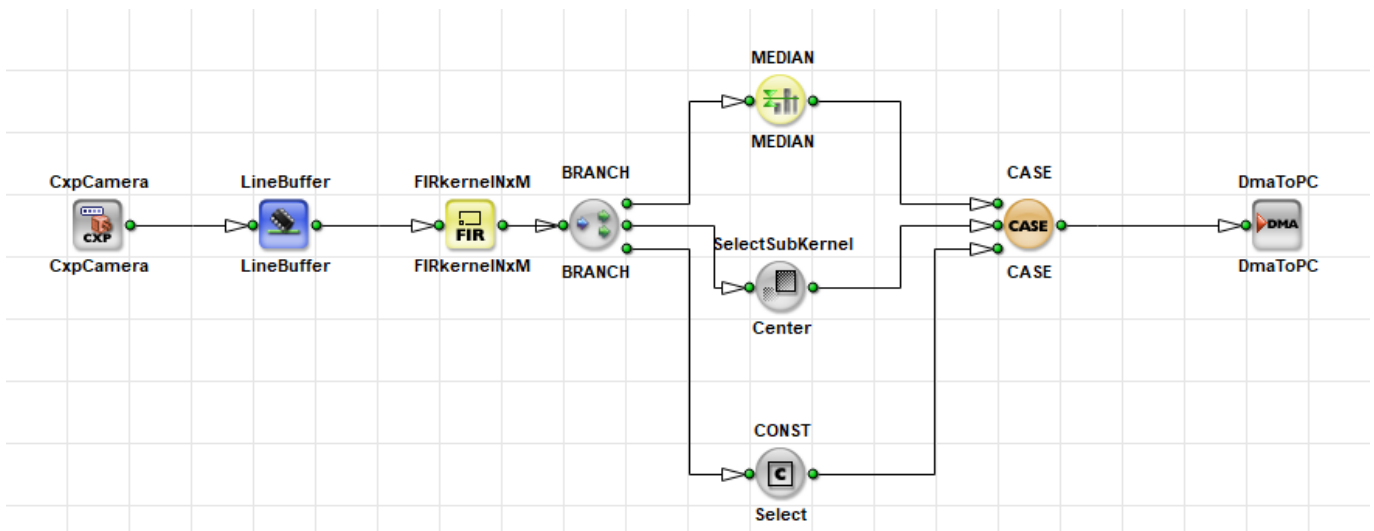
Only the selected input is forwarded to the output; all other inputs are discarded. If a switch occurs while an image is still being transferred on an input, the transfer is completed before the new source becomes active.

Use the `SelectSource` parameter to control the active source at runtime.

The operator finishes the current frame before it switches to another input. Set `InfiniteSource` to `ENABLED` when a camera feeds the input. Set it to `DISABLED` when the input comes from a buffered, stoppable stream.



Basler recommends that you use `IF` or `CASE` instead of `SourceSelector` when all inputs are sourced from the same M-type module. Those operators use fewer resources. See for example the following design with a `CASE` operator:

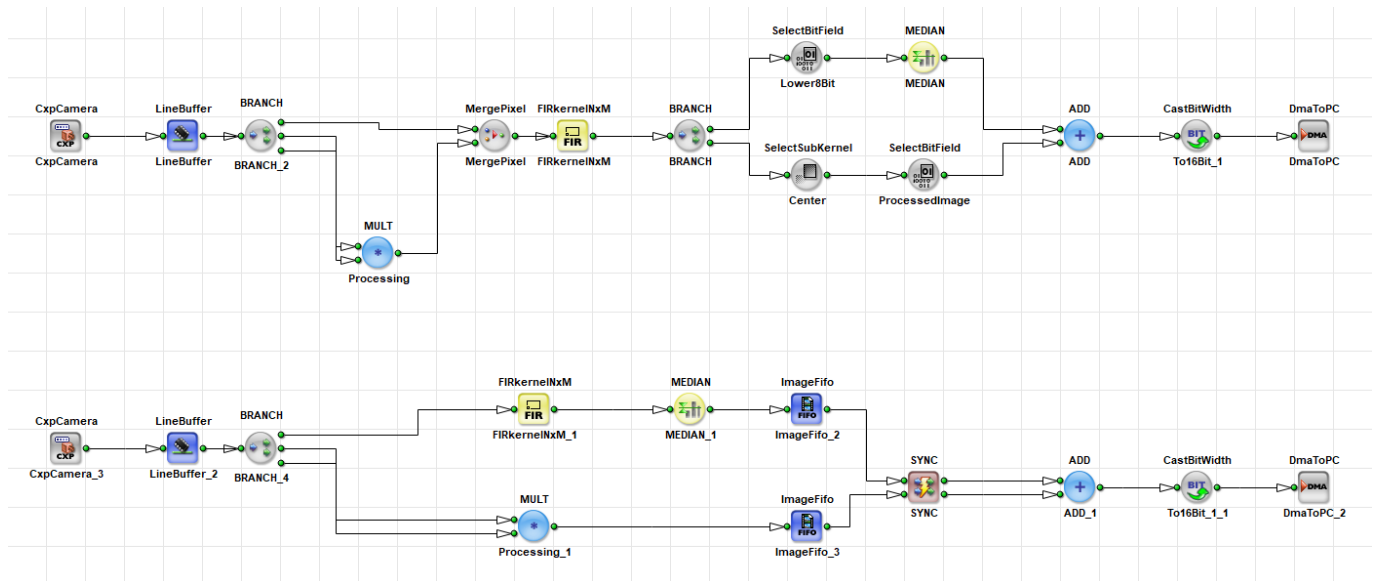


See also [SourceSelector](#) in the Operator Reference.

4.11 Avoiding Synchronization

As operators for synchronization and buffering consume a lot of FPGA resources, avoid the need for synchronization whenever possible. In many cases this can be done by applying alternative design techniques. Such techniques may

include merging data before required M-type operators and splitting the data afterwards. The following screenshot shows two implementations of similar workflows, once using `MergePixel` and `ADD`, and once using `SYNC`:

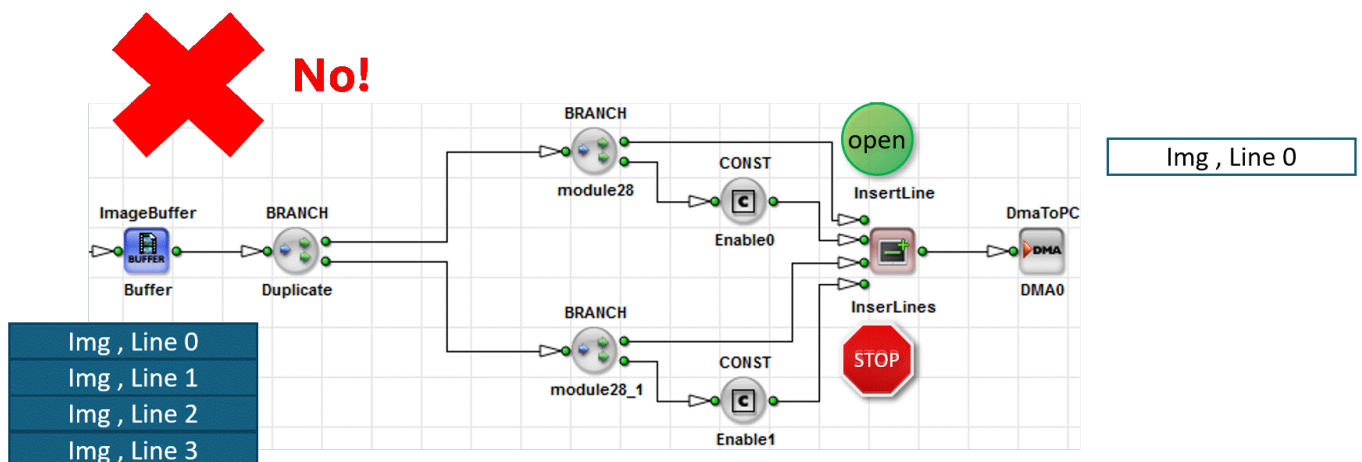


4.12 Troubleshooting: Deadlock

One of the most common and most difficult synchronization errors is a **deadlock**.

A deadlock occurs when two sources are synchronized directly, for example, by using a `SYNC` operator, without any buffering. In this situation, each source waits for the other: A waits for B, and B waits for A. As a result, no data is transferred.

The diagram below shows a branch module on the lower path that receives a stop signal and therefore no longer accepts data at its input. As a result, the upper path also stops receiving data, and the entire pipeline halts:



[Double-click here for the animated version.](#){large-image}

Deadlocks are often not visible in simulation, where the system may appear to run normally. On hardware, however, they typically result in no output. In such cases, the hardware applet freezes, stops transferring data to the PC, and eventually causes a DMA timeout.

Info

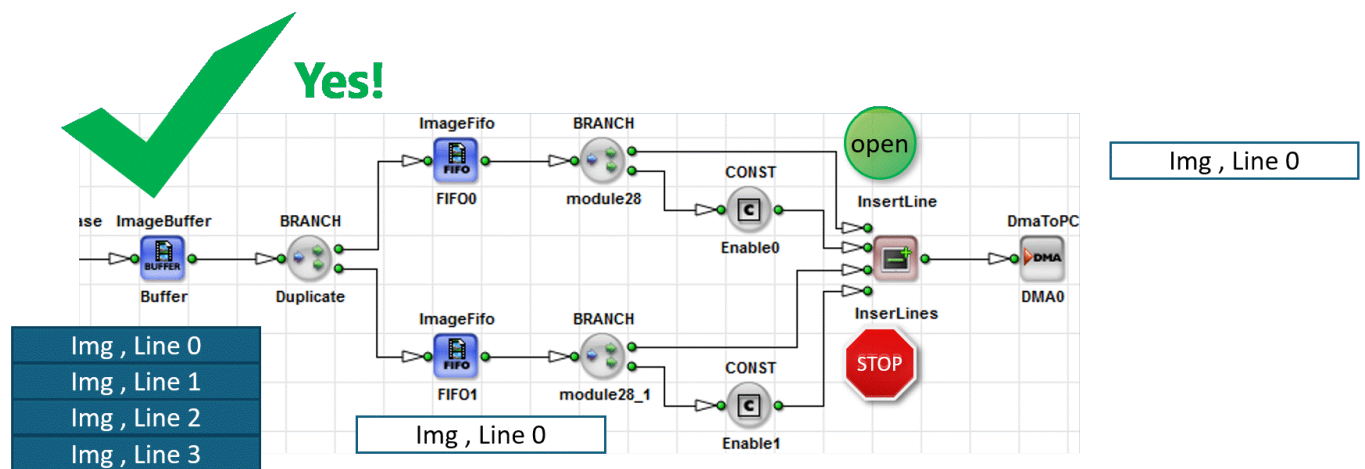
Deadlocks are difficult to detect because they are often not visible in simulation. To identify a possible deadlock, monitor the FIFO fill levels in hardware using runtime tools such as microDisplay X. A FIFO that remains at **100%** capacity indicates a possible deadlock or a bottleneck.

Typical causes for a deadlock:

- Missing buffers
- Incorrect synchronization

How to fix a deadlock:

To fix a deadlock, insert FIFO-based buffering, for example, `ImageBuffer` or `ImageFifo`, before the `SYNC` operator. This allows one source to run slightly ahead and prevents the mutual wait condition:



[Double-click here for the animated version.](#){large-image}

4.13 Verifying Your Design with DRC

Run **Analysis** → **Design Rules Check Level 1** after you wire branches that merge. Level 1 reports synchronization violations and marks affected links in purple.

Design Rules Check Level 2 runs extended checks and estimates FPGA resources. Fix formal errors from Level 1 before you build.

Read the **DRC log** together with highlighted modules and links. For the full workflow, see [From Empty Design to Running Applet](#) Tutorial topic.

4.14 Related Topics

- [Rules of Links](#) in the User Manual — O-type network rule and signal links
- [Stoppable and Non-Stoppable Data Streams](#) Tutorial topic — infinite sources and flow control
- [Memory Elements](#) Tutorial topic — buffers before `SYNC`
- [Pipeline Concept](#) Tutorial topic — blocking and backpressure
- [Synchronization of Asynchronous Image Pipelines](#) in Basics and Examples
- [From Empty Design to Running Applet](#) Tutorial topic — DRC, simulation, and build
- [Parameterization](#) Tutorial topic — link properties at merge points

5. Stoppable and Non-Stoppable Data Streams

This topic explains how image data streams behave when the generated FPGA pipeline applies **flow control**. After you have read this topic, you understand the difference between **stoppable** and **non-stoppable** streams, how **infinite sources** such as camera operators fit into a design, and how you use buffer operators, the `InfiniteSource` parameter, and the `Overflow` operator to avoid connection errors and deadlocks.

Key Terms

A **non-stoppable** stream runs continuously. A non-stoppable source can't be paused by the flow-control logic of the generated FPGA design. A typical example is a **camera** that keeps sending pixels. As a contrast, a **stoppable** stream can pause when the next operator isn't ready. A **buffer** between camera and downstream processing often turns a non-stoppable stream into a stoppable one.

Therefore, after a camera, place a buffer. For absorbing camera bursts, prefer a DRAM operator: `LineBuffer` on `imaFlex`, or `ImageBuffer` on `microEnable 5`. Use `ImageFifo` (FPGA-internal memory) only for small decoupling, for example a short delay before `SYNC`. If you use `ImageFifo` on a non-stoppable stream, combine it with the `Overflow` operator, or use `ImageFlowControl` instead. See [Memory Elements](#) Tutorial topic.

For the formal rules behind link connections, see [Infinite Sources / Connecting Cameras](#) in the User Manual. For operator shapes and how modules pause or hold back data flow, see [Operator Types and Synchronization](#) Tutorial topic.

5.1 Stoppable and Non-Stoppable Streams

The generated FPGA pipeline controls data flow between modules. An operator can **pause its input** when it is not ready to accept more data. Downstream operators can therefore slow the stream for a limited time. VisualApplets does not apply that flow control in the editor; the built applet on the FPGA does.

A stream is **stoppable** when flow control can hold it back without breaking the design. A stream is **non-stoppable** when the source can't be stopped by flow control. The source keeps delivering data even if the next module is busy.

Non-stoppable streams are also called **infinite sources**. The name refers to flow control, not to unlimited image size. The stream has no internal buffer that lets the generated FPGA design pause the source.

5.2 Non-Stoppable Sources in a VisualApplets Design

Camera operators are the most common non-stoppable sources. They receive pixels from the camera and forward them on the output link. They have no buffer that can absorb backpressure, and they can't command the camera to stop the transfer.

In contrast, stoppable sources include operators such as `CreateBlankImage` and `CoefficientBuffer`.

5.3 Connecting Non-Stoppable Sources

All **O-type** and **P-type** operators can sit downstream of a non-stoppable source. The infinite-source property propagates through these operators in the design until a module converts the stream into a **controllable** stream.

Only some **M-type** operators accept a non-stoppable source directly. These are mainly **memory operators** such as `LineBuffer`, `ImageBuffer`, `ImageFifo`, and `FrameBuffer`. Whether a specific M-type operator accepts an infinite source is documented in the operator help and in the [Operator Reference](#).

If you connect a non-stoppable source to an M-type operator that doesn't accept infinite sources, **Design Rules Check Level 2** reports an **infinite source** error. Fix the error before you build the applet.

See [Operator Types and Synchronization](#) Tutorial topic for M-type, O-type, and P-type behavior and for flow-control rules.

You have two options to control an infinite source:

- With a buffer operator such as `LineBuffer`, `ImageBuffer`, or `ImageFifo`
- With the `overflow` operator.

For a camera path, Basler recommends a DRAM buffer (`LineBuffer` or `ImageBuffer`) as the default. Use `ImageFifo` only for small decoupling tasks. See the Key Terms box at the start of this topic and [Memory Elements](#) Tutorial topic.

Info

The `overflow` operator in the **Synchronization** library cuts frames that can't be processed. This leads to different frame lengths, and not all operators downstream can handle this. Do not confuse this operator with **arithmetic overflow** (a result that no longer fits the bit width; see [Bit Arithmetic](#) Tutorial topic) or with the **overflow status parameter** on buffer operators (the buffer is full; see [Memory Elements](#) Tutorial topic).

The `overflow` operator doesn't buffer data itself. Therefore Basler recommends to use the `overflow` operator always in combination with a buffer operator.

5.3.1 Using Buffer Operators

Buffer operators turn a non-stoppable stream into a controllable stream. They accept data on the input even when downstream modules aren't ready to take more data. They store data internally and forward it when downstream modules are ready.

Typical decoupling operators include **LineBuffer**, **ImageBuffer**, **ImageFifo**, and other memory operators that support infinite sources. Place a DRAM buffer (`LineBuffer` or `ImageBuffer`) directly after the non-stoppable source when you need to absorb camera bursts and when downstream processing or **DmaToPC** must apply flow control. Use `ImageFifo` only when the required buffer is small.

Info

The `ImageFifo` operator buffers only small portions of an image. As a result, there is a significant risk of FIFO overflow, which is not internally detected or handled.

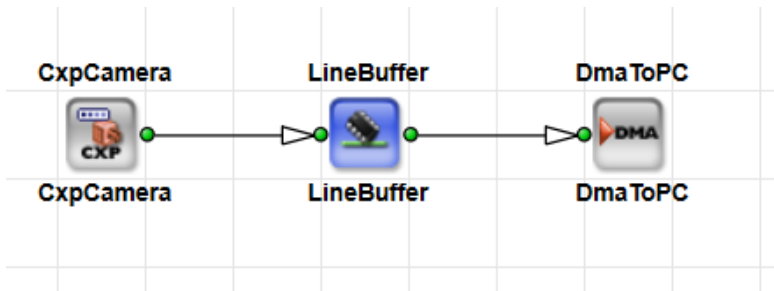
If you want to use an `ImageFifo` operator to buffer a non-stoppable stream, Basler recommends to use the `ImageFifo` in combination with a preceding `overflow` operator. This ensures proper handling of incoming data and prevents uncontrolled overflow conditions.

As an alternative to combining the `overflow` operator with a memory operator, you can use the `ImageFlowControl` operator for safer and more robust handling. It combines the buffering functionality of the `ImageFifo` with integrated overflow handling.

A typical acquisition path looks like this:

Camera → **LineBuffer** or **ImageBuffer** (DRAM) → further processing → **DmaToPC**

For small decoupling only, you can use **ImageFifo** instead of the DRAM buffer, preferably together with **Overflow**, or use **ImageFlowControl**.



Without the buffer, **DmaToPC** or another operator that pauses its input can't throttle the camera stream. The design fails DRC or risks deadlock on hardware.

When using a buffer operator to control infinite sources, you must set the `InfiniteSource` parameter of the buffer operator to `ENABLED`.

Process0* X

Module Properties: Process0/LineBuffer <LineBuffer>

Parameter Name	Parameter Value	Parameter Unit	Parameter Type
Name	LineBuffer		
RamDataWidth	384		Static
XOffset	0	pixel	Dynamic
XLength	1024	pixel	Dynamic
YOffset	0	pixel	Dynamic
YLength	1024	pixel	Dynamic
ParallelismConverter	No		Static
InfiniteSource	ENABLED		Static
WritePriority	DISABLED		Static
FillLevel	0	%	Dynamic
MaxLineCount	1458888	line	Dynamic
LineCount	0	line	Dynamic

enum (ENABLED=0, DISABLED=1)

DRC Log

Netlist Generation

Creation date: Mittwoch, 3. Juni 2026 11:12:12

Netlist generation started...

Info: Building design for FPGA clock frequency 312.500 MHz

FPGA resource estimation:

Resource	Count	Fill Level
LookupTables	54177	~ 33%
Flip Flops	53183	~ 16%
Block RAM	125	~ 17%
Ultra RAM	0	~ 0%
Embedded ALU	47	~ 3%

11:12:16 -- Netlist successfully generated in 00:00:04.

Design Rules Check Level 2

Program version: VisualApplets 3.7.100.76452M-9d0b71df5

Design name:

Creation date: Mittwoch, 3. Juni 2026 11:12:16

0 Errors
0 Warnings

11:12:16 -- Design Rules Check Level 2 successfully performed.

If the `InfiniteSource` parameter of the buffer operator is set to `DISABLED`, DRC Level 2 will fail:

The screenshot displays the software interface with three main components:

- Process0* Window:** Shows a data flow diagram with three modules: CxpCamera, LineBuffer, and DmaToPC. The LineBuffer module is highlighted.
- Module Properties: Process0/LineBuffer <LineBuffer>:** A dialog box showing the configuration for the LineBuffer module. The **InfiniteSource** parameter is set to **DISABLED** and is highlighted with a red box. Other parameters include RamDataWidth (384), XOffset (0), XLength (1024), YOffset (0), YLength (1024), ParallelismConverter (No), WritePriority (DISABLED), FillLevel (0), MaxLineCount (145888), and LineCount (0).
- DRC Log Window:** Shows the results of the Design Rules Check Level 2. It indicates that the netlist was successfully generated. The check failed due to an error: "Error: -150 Checking output latency: Error at module {+} LineBuffer (LineBuffer) . Module cannot buffer data stream / infinite source." The results show 1 Error and 0 Warnings.

For buffer sizing, `FillLevel`, and RAM usage, see [Memory Elements](#) Tutorial topic.

5.3.2 Setting the InfiniteSource Parameter

Many buffer operators expose the parameter `InfiniteSource`. Set it to `ENABLED` when the operator input connects to a non-stoppable source. Set it to `DISABLED` when the input comes from a stoppable stream.

Follow these rules:

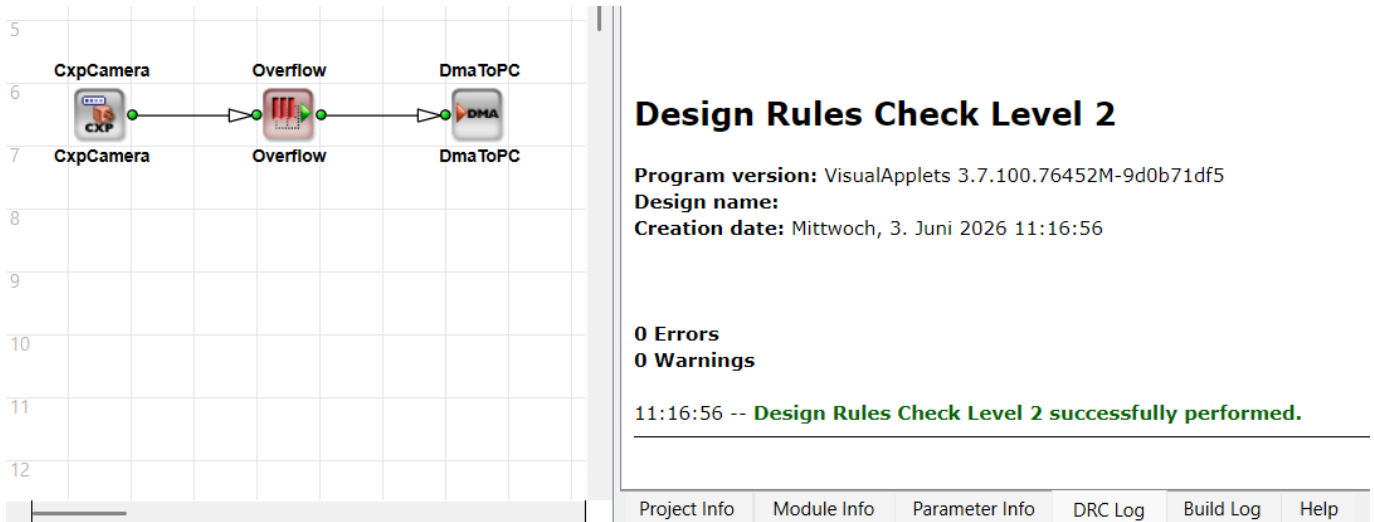
- Set `InfiniteSource` to `ENABLED` only when the module input actually connects to a non-stoppable source.
- Set `InfiniteSource` to `DISABLED` on buffer operators that receive data from a stoppable stream.
- If you are unsure, set `DISABLED` first. Run **Design Rules Check Level 2** and adjust modules that the check flags.

To change module properties, open **Module Properties** with a double-click on the module. Apply parameter changes with **Apply**. See [Parameterization](#) Tutorial topic for link and module properties.

5.3.3 Using the Overflow Operator When Downstream Processing Can Pause

The `Overflow` operator decouples a non-stoppable incoming 2D stream from processing that may block temporarily. It doesn't have an internal buffer on its own, therefore Basler recommends to use the `Overflow` operator always in combination with a buffer operator. When the buffer fills because output data is blocked, the `Overflow` operator cuts the current frame, marks it as corrupted, and can skip further frames until the output accepts data again.

Use `Overflow` when the source can't be stopped and downstream operators may pause the data flow for longer than a line or frame buffer can cover.



When you place the `Overflow` operator between a non-stoppable source and a memory operator such as `FrameBuffer`, set the `InfiniteSource` parameter on that memory operator to `DISABLED`. The `Overflow` operator already handles overflow from the non-stoppable stream. Enabling `InfiniteSource` on the memory module in that topology is wrong and triggers DRC errors.

The `Overflow` operator can produce non-rectangular frames when it cuts a frame. Verify that downstream operators tolerate shortened last lines. See the operator description [Overflow](#) in the Operator Reference.

Info

As an alternative to combining the `Overflow` operator with a memory operator, you can use the `ImageFlowControl` operator for safer and more robust handling. It combines the buffering functionality of the `ImageFifo` with integrated overflow handling.

5.4 Verifying Your Design with DRC

Run **Analysis** → **Design Rules Check Level 2** after you wire sources, buffers, and DMA paths. Level 1 also runs automatically before simulation and build.

DRC Level 2 checks extended rules and estimates FPGA resources. It reports infinite-source errors when a non-stoppable source feeds an M-type operator that doesn't accept infinite sources, or when `InfiniteSource` settings don't match the actual connection.

Read the **DRC log** and fix highlighted modules or links. For the full workflow, see [From Empty Design to Running Applet](#) Tutorial topic.

5.5 Typical Pipeline Examples

5.5.1 Camera to DMA Through a Buffer

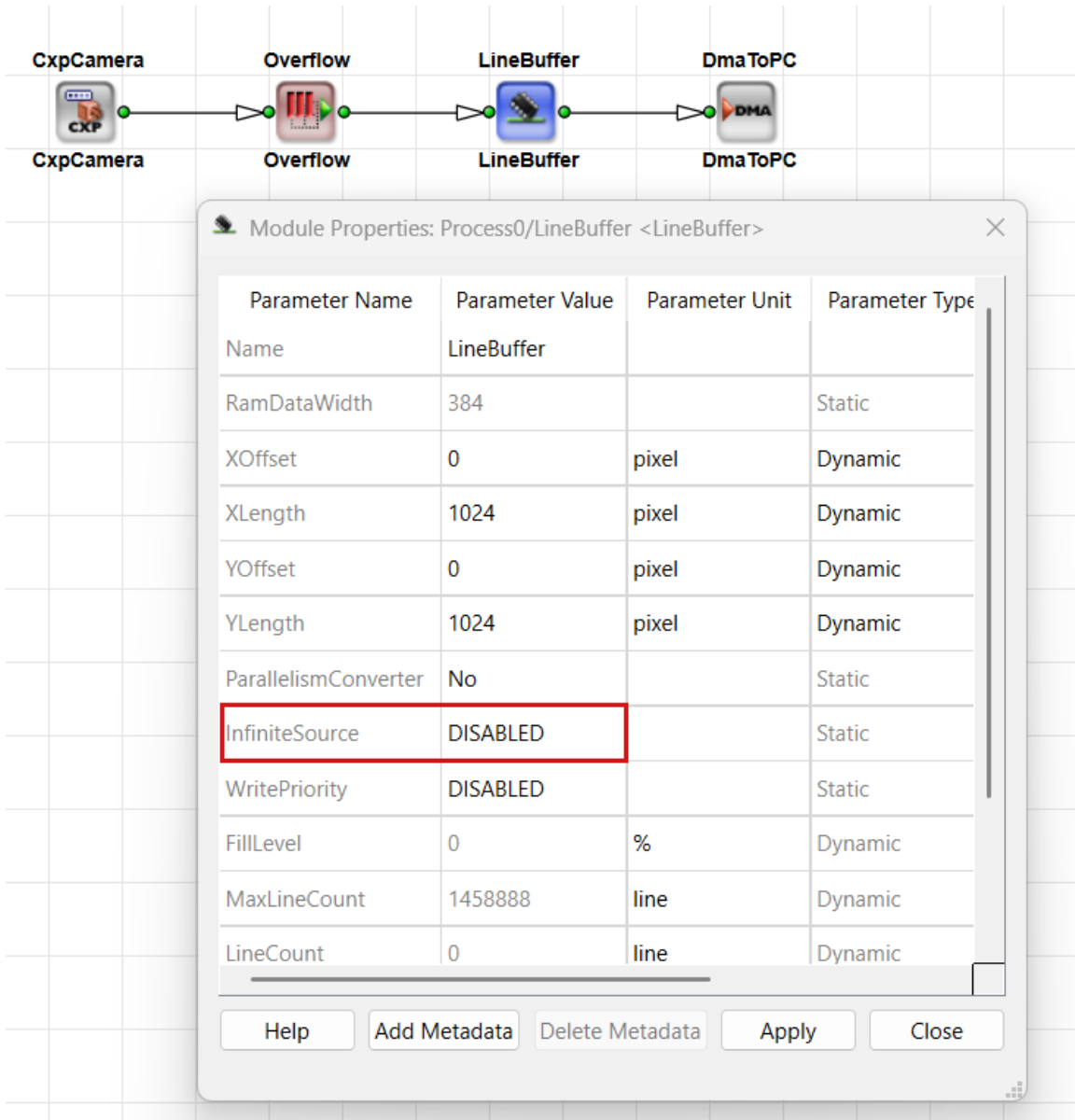
1. Place a **camera** operator as the source.
2. Insert `LineBuffer` or `ImageBuffer` on the image path (or `ImageFifo` only for small decoupling; see above).
3. Set `InfiniteSource` to `ENABLED` on the buffer.
4. Connect processing operators and `DmaToPC`.

5. Run DRC Level 1 and Level 2, then simulate or build.

Without step 2 and step 3, DRC Level 2 reports an infinite-source error on the illegal connection.

5.5.2 Non-Stoppable Source with Overflow and Buffer

1. Connect the non-stoppable source to `Overflow`.
2. Connect `Overflow` to `ImageFifo`, `LineBuffer`, or another buffer operator.
3. Set `InfiniteSource` to `DISABLED` on the buffer after `Overflow`.
4. Connect further processing and run DRC Level 2. Confirm that no infinite-source or `InfiniteSource` mismatch remains.



`Overflow` cuts or skips frames when its output is inhibited. The following buffer stores data for downstream processing that may pause. Without that buffer, `Overflow` alone cannot decouple a non-stoppable source from pausing downstream modules.

5.6 Related Topics

- [Rules of Links — Infinite Sources / Connecting Cameras](#) in the User Manual — formal connection rules
- [Operator Types and Synchronization](#) Tutorial topic — M-type operators and blocking
- [Memory Elements](#) Tutorial topic — buffer operators and `InfiniteSource`
- [Parameterization](#) Tutorial topic — `InfiniteSource` and related module parameters
- [Pipeline Concept](#) Tutorial topic — flow control, blocking, and backpressure
- [From Empty Design to Running Applet](#) Tutorial topic — DRC, simulation, and build

6. Parallelism

This topic explains what the **Parallelism** link property means in a VisualApplets design and how that setting maps to the generated FPGA pipeline. It also explains how you calculate the parallelism a design needs, and how you adapt parallelism with dedicated operators. After you have read this topic, you understand the relationship between parallelism, bandwidth, and FPGA resource usage, how image width constraints depend on parallelism, and how memory operators such as `FrameBufferRandomRead` limit parallelism.

Key Terms

The **Base Design Clock** is the clock of the generated FPGA pipeline on the frame grabber, for example 312.5 MHz. It is not the same as the camera pixel clock, and it is not a clock inside the VisualApplets editor. A **clock cycle** is one tick of the Base Design Clock. Per clock cycle, a certain number of pixels can pass through a link. That number is **parallelism**. Parallelism 4 means four pixels per clock cycle. A higher Base Design Clock frequency or higher parallelism increases the **pixel rate**, which this topic calls **bandwidth**.

6.1 Parallelism and Bandwidth

Parallelism is the number of pixels that flow **per clock cycle** through a link of the generated FPGA pipeline.

Parallelism affects the processing granularity inside modules. For example, the `LineBuffer` parameter `XLength` depends on parallelism.

The following rules apply:

- **Higher parallelism → higher pixel-rate bandwidth.**
- **Higher parallelism → higher FPGA resource usage.**

Bit width and kernel size are not part of this pixel-rate number. They increase the amount of data per clock, which matters for DRAM and resources. See [Working with Kernels](#) Tutorial topic and [Memory Elements](#) Tutorial topic.

Design Guideline

Keep parallelism no higher than necessary to save FPGA resources.

The bandwidth of a link as a pixel rate follows this formula:

Bandwidth = parallelism × Base Design Clock

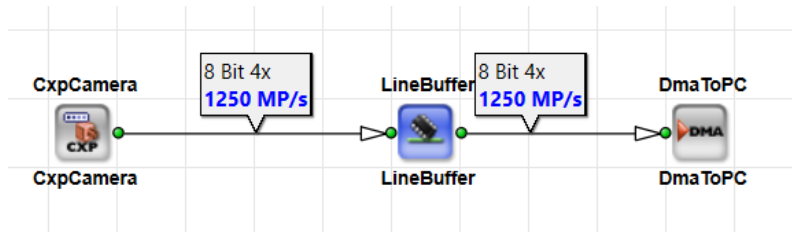
The Base Design Clock depends on the frame grabber you use. For Basler frame grabbers, look up the value in [Device Resources](#).

Key Terms

You can make the parallelism and the data throughput visible in your design by using the **Link Info** and the **Throughput** flags. Activate these flags with the following buttons from the toolbar:



As a result, the **Link Info** and **Throughput** flags are displayed in your design:



In this topic, we show all screenshots of the designs with the **Link Info** and **Throughput** flags enabled.

6.2 Calculating the Required Parallelism

To dimension a design, first calculate the required pixel rate, then derive the required parallelism from the Base Design Clock.

6.2.1 Calculating the Parallelism for Area Scan Designs

For designs with area scan cameras, first calculate the bandwidth in pixels per second:

$$\text{Bandwidth} = (\text{pixels per line}) \times (\text{lines per frame}) \times \text{frame rate}$$

Then use the bandwidth to calculate the required parallelism:

$$\text{Required parallelism} = \text{bandwidth} / \text{Base Design Clock}$$

Select the smallest supported parallelism that is at least as large as the calculated requirement. Rounding down would leave the design below the required bandwidth. For example, 6000 MP/s at a Base Design Clock of 312.5 MHz needs a parallelism of 19.2. Parallelism 19 would provide only 5937.5 MP/s, which is not enough. Use 20, or the next higher value that your platform allows.

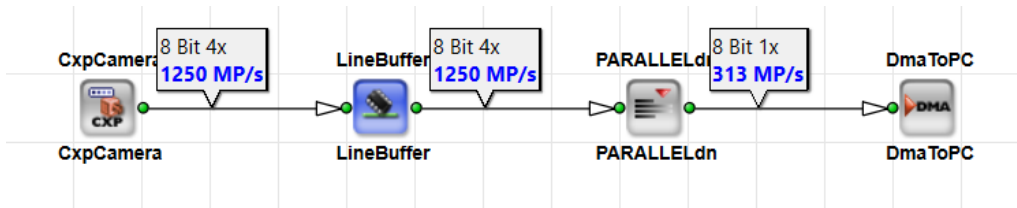
6.2.2 Area Scan Calculation Example

For a camera with resolution 2064 × 2064 pixels and 30 frames per second, the following bandwidth calculation applies:

$$\text{Bandwidth} \approx 2064 \times 2064 \times 30 \approx 127.9 \text{ MP/s}$$

With a Base Design Clock of 312.5 MHz, this leads to the following calculated parallelism:

$$\text{Required parallelism} \approx 127.9 / 312.5 \approx 0.41 \rightarrow \text{round up to 1}$$



The screenshot above shows the corresponding VisualApplets design implementation. The parallelism at the **camera interface** is set to 4. That value is not the processing parallelism from the bandwidth calculation. The **CxpCamera** operator sets interface parallelism automatically from the number of CXP connections defined by the **ConnectionCount** module parameter (in this case, x1). Interface parallelism and required processing parallelism are different numbers. Since the required processing parallelism is 1, the link parallelism is reduced to 1 at the output link of the **LineBuffer** operator.

At low frame rates, a parallelism value of 1 is sufficient. At higher frame rates or resolutions, the required parallelism increases. Calculate the value and then round it up to the next supported level, for example: 2, 4, or 8. If the platform requires an **even** parallelism, round up first, then select the next higher valid even value. For example, if the calculation results in 6.3, use 8, and not 7.

6.2.3 Calculating the Parallelism for Line Scan Designs

For designs with line scan cameras, first calculate the bandwidth in pixels per second:

$$\text{Bandwidth} = (\text{pixels per line}) \times \text{line rate}$$

Then use the bandwidth to calculate the required parallelism:

$$\text{Required parallelism} = \text{bandwidth} / \text{Base Design Clock}$$

Round the result up according to your platform rules and the allowed parallelism values.

6.2.4 Line Scan Calculation Example

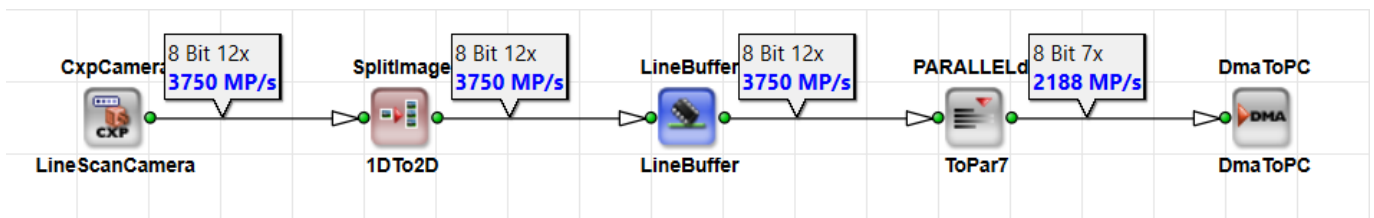
For a camera with resolution 16384 pixels and 120 kHz line rate, the following bandwidth calculation applies:

$$\text{Bandwidth} \approx 16384 \times 120 \text{ kHz} \approx 1966.08 \text{ MP/s}$$

With a Base Design Clock of 312.5 MHz, this leads to the following calculated parallelism:

$$\text{Required parallelism} \approx 1966.08 / 312.5 \approx 6.3 \rightarrow \text{round up to } 7$$

The calculation yields 6.3, so the next integer is 7. If the platform requires an **even** parallelism, select the next higher valid even value (8, not 7). The screenshot below uses parallelism 7, which is valid when even parallelism is not required.



The screenshot above shows the corresponding VisualApplets design implementation. The parallelism at the camera interface is set to 12. This value is automatically configured by the **CxpCamera** operator from the number of CXP connections (**ConnectionCount** parameter set to x2). That interface parallelism is again not the same as the required processing parallelism. The **SplitImage** operator performs a 1D-to-2D conversion. The **LineBuffer** operator then reduces the link parallelism to 7 to match the calculated processing requirement.

6.3 Changing the Parallelism in a Design

The operators below adapt the parallelism of a design to the **bandwidth requirements**:

6.3.1 Increasing or Decreasing the Parallelism

- `PARALLELDn` — reduces parallelism. See [PARALLELDn](#) in the Operator Reference.
- `PARALLELup` — increases parallelism. See [PARALLELup](#) in the Operator Reference.
- **LineBuffer (imaFlex only):** The `ParallelismConverter` parameter of the `LineBuffer` operator adjusts parallelism between input and output without additional parallelism operators. See [LineBuffer \(imaFlex\)](#) in the Operator Reference.

6.3.2 Splitting, Selecting, and Merging Streams

- `SplitParallel` — Splits one data stream with a higher parallelism into several streams with lower parallelism. Example: Parallelism 8 becomes two streams with parallelism 4 each. See also [SplitParallel](#) in the Operator Reference.
- `SelectFromParallel` — Selects individual components from a parallel stream, for example one data word out of several transferred per clock cycle. See also [SelectFromParallel](#) in the Operator Reference.
- `MergeParallel` — Merges parallel streams again after separate processing. See also [MergeParallel](#) in the Operator Reference.

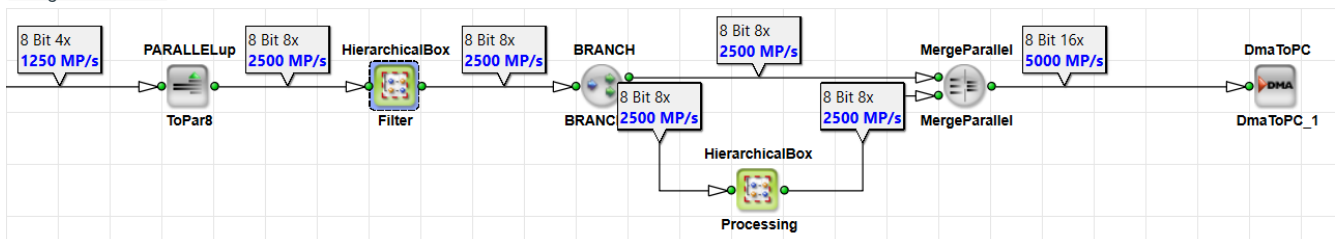
6.3.3 Other Operators

- `CastParallel` — Reinterprets bits on the link. Parallelism and bit width may change as long as **bit width × parallelism** stays equal at input and output. Example: 16 bit × PAR 2 becomes 8 bit × PAR 4. Image width changes accordingly. See also [CastParallel](#) in the Operator Reference.
- `ExpandToParallel` — Takes an input with parallelism 1 and produces the desired output parallelism. The input value is replicated to all parallel channels. See also [ExpandToParallel](#) in the Operator Reference.

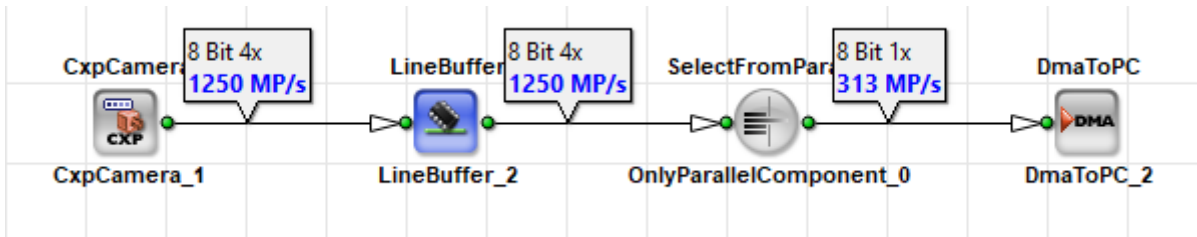
6.3.4 Use Cases

The following use cases appear frequently:

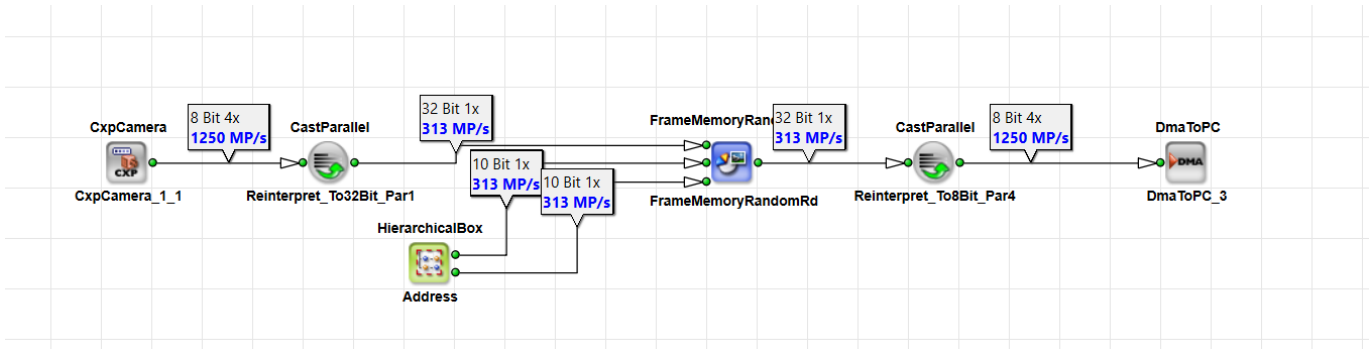
- Use `PARALLELup` to increase parallelism, process data in parallel branches, and merge the branches again with `MergeParallel`.



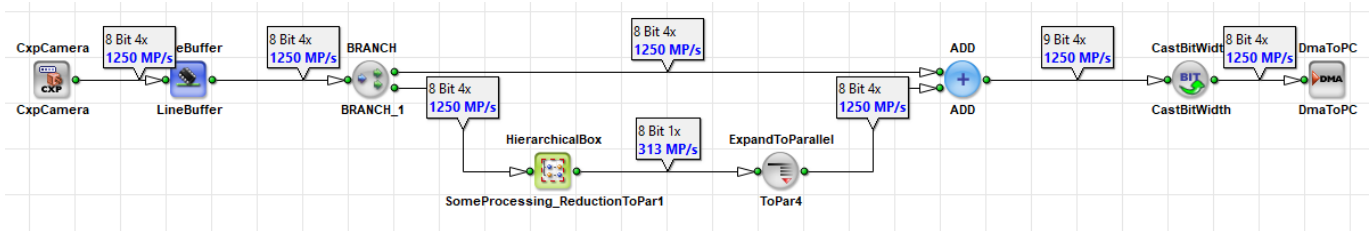
- For certain processing tasks, selecting just one component is mathematically sufficient and provides the required results while reducing the data amount. Use `SelectFromParallel` to forward only one channel from a parallelism 4 stream.



- After `FrameBufferRandomRead`, the stream runs at parallelism `1` with, for example, 64 bit per pixel. In order for downstream processing to run at higher parallelism, reinterpret the data stream as 8 bit × parallelism `8`. For this, use the `CastParallel` operator.



- Use `ExpandToParallel` to expand a control value or pixel stream with parallelism `1` to parallelism `4` when the same value must be available in all four parallel channels, for example for coefficients or masks in parallel branches.



6.4 Parallelism Constraints

The **maximum image width** in pixels must be **divisible by the parallelism**.

Examples:

- With parallelism `4`, allowed image widths include 1024 and 2048.
- With parallelism `12`, an image width of 16384 is **not** allowed because $16384 / 12$ is not an integer.

Workarounds:

1. **Dummy pixels** — widen the image, for example through simulation link properties or operators, until the image width divides evenly by the parallelism value.
2. **Crop** — trim the image to a valid image width.

Key Terms

The generated FPGA hardware processes pixels in packets per clock. A line of 16384 pixels doesn't pack cleanly into parallelism 12, because $16384 / 12$ is not an integer. VisualApplets therefore requires that the image width in the design is divisible by the parallelism.

6.5 Parallelism with Memory Operators

`FrameBufferRandomRead` allows only **parallelism 1**, because memory is accessed sequentially. To use higher parallelism in the design and to not influence the overall bandwidth, convert the parallel components to kernel components using the `CastKernel` operator or to higher bit components of the current pixel using the `CastParallel` operator.

See the [Working with Kernels](#) Tutorial topic for an example of how to do this.

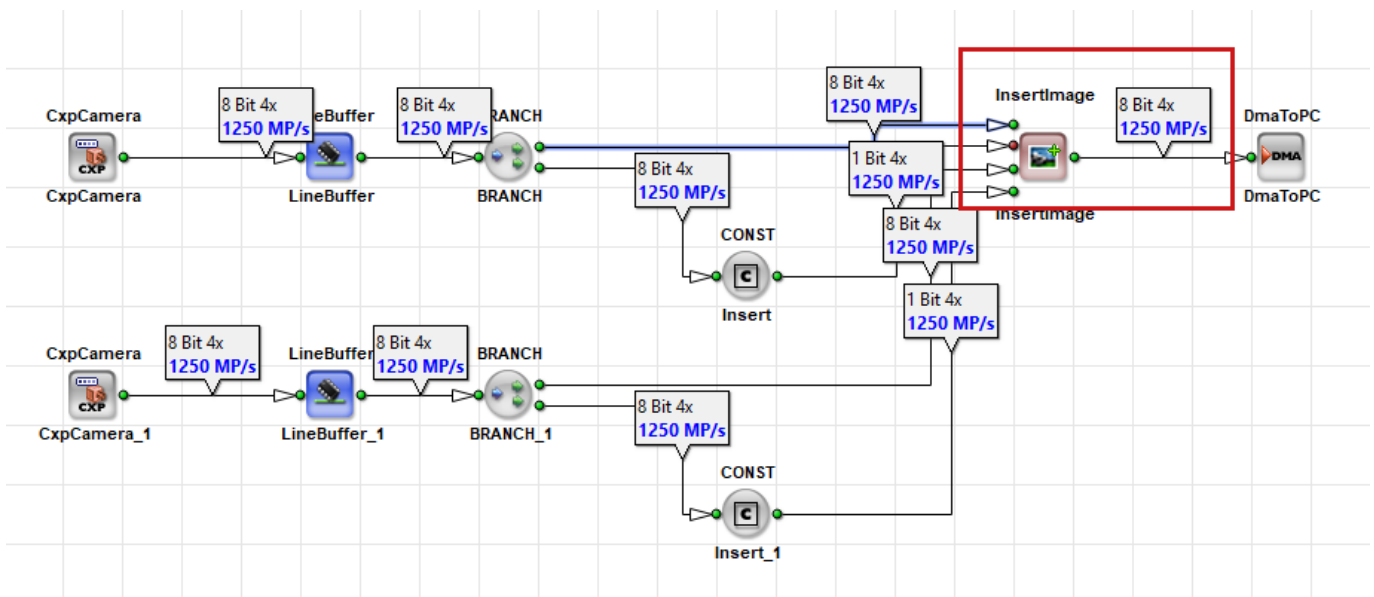
See [FrameBufferRandomRead](#) in the Operator Reference and [FrameBufferRandomRead \(imaFlex\)](#) in the Operator Reference.

6.6 Troubleshooting Bottlenecks

A typical bottleneck arises when [InsertLine](#) in the Operator Reference or [InsertImage](#) in the Operator Reference isn't combined with `PARALLELup`. If the operator order is unfavorable, throughput can drop.

Using insert operators increases the amount of data fed into a processing pipeline, which raises the overall bandwidth. Therefore, the parallelism after the `InsertLine` or `InsertImage` operator must be high enough to handle the resulting bandwidth.

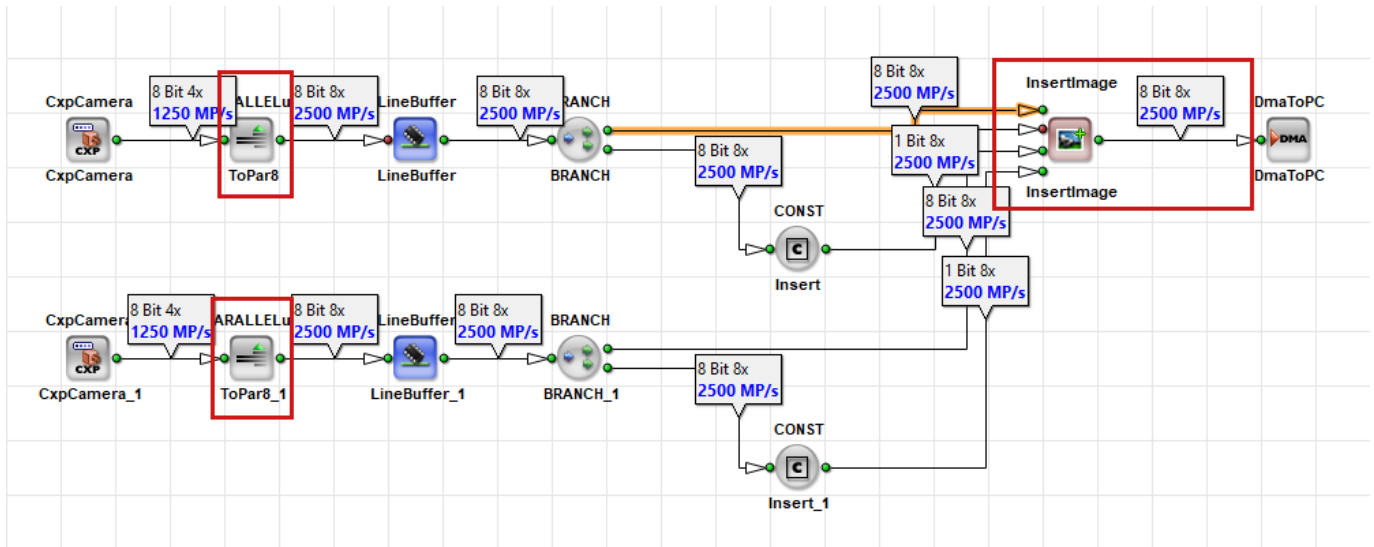
6.6.1 Example



The example in the image above shows that the `InsertImage` operator has two input links. Each input link processes images with a resolution of 5120×5120 pixels at a bandwidth of 1250 MPixel/s, using a parallelism of 4.

If the parallelism isn't increased before the `InsertImage` operator, the bandwidth per input remains 1250 MPixel/s. However, because there are two input links, the design must process a combined throughput of 2×1250 MPixel/s.

To avoid this bandwidth bottleneck, increase the parallelism using the `PARALLELup` operator before the `InsertImage` operator as shown below:



6.7 Related Topics

- [Pipeline Concept](#) Tutorial topic — throughput, bottlenecks, and link capacity
- [Parameterization](#) Tutorial topic — Parallelism link property
- [Memory Elements](#) Tutorial topic — RAM data width and LineBuffer parallelism conversion
- [Working with Kernels](#) Tutorial topic — kernel size versus parallelism
- [Latency](#) Tutorial topic — parallelism and latency
- [Bandwidth of an Applet](#) in the User Manual — formal bandwidth definition
- [Device Resources](#) — Base Design Clock frequencies
- [Bandwidth Bottlenecks](#) in the User Manual — operator ordering and bottlenecks

7. Latency

This topic explains what **latency** means in the FPGA pipeline generated from a VisualApplets design, how latency differs from **bandwidth**, and how large the latency is in a typical design. After you have read this topic, you understand why a pixel does not appear at the output immediately, how operator types and buffering affect delay, and where you find latency values for individual operators.

Key Terms

Latency is the time a single pixel needs to pass through the generated FPGA pipeline from camera (or other source) input to an output such as `DmaToPC`. That delay happens on the frame grabber, not in the VisualApplets editor. **Bandwidth** is how many pixels per second can flow through a link.

A design can have high bandwidth and low latency at the same time: many pixels move in parallel through the FPGA pipeline, but the first pixel of a frame still needs some time before it is handed to DMA.

7.1 Latency vs. Bandwidth

Bandwidth and **latency** describe different properties of the same pipeline:

	Bandwidth	Latency
Question	How much data flows per second?	How long does one pixel wait until it is processed?
Depends on	Parallelism and Base Design Clock	Operator type and algorithm
Formula (link)	Bandwidth = parallelism × Base Design Clock	Defined per operator; see below

A pipeline can transfer and process image data at the same time. For example, while pixel 1000 is already at the output, pixel 5000 may still be at the camera input. The overall data rate (bandwidth) stays high because multiple pixels run through the design in parallel. Latency only describes the delay of an individual pixel.

Design Guideline

In the FPGA pipeline generated from a VisualApplets design, latency is kept close to the theoretical minimum. Most operators do not store a full image. They forward results as soon as the information is available.

7.2 Latency Definition

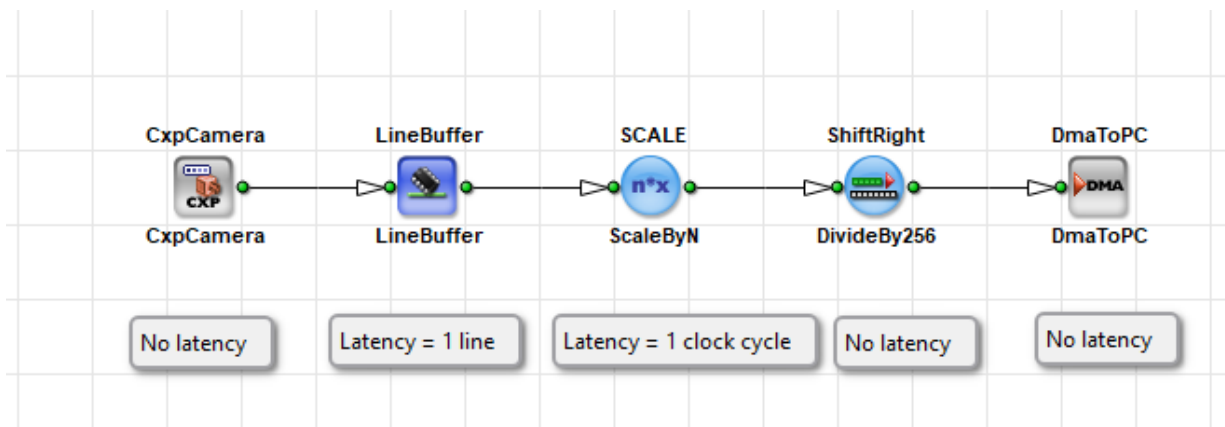
Latency is the delay of a pixel in a design — the time from when a pixel enters the processing chain until it leaves it (for example, until it reaches `DmaToPC`).

Important points:

- Latency is defined **individually by each operator** and mostly depends on its algorithmic implementation.
- Adding operators does **not** reduce **throughput**. All operators in the generated FPGA pipeline run in parallel: while one module processes a pixel, the other modules process other pixels. After the pipeline has filled, a longer chain can still accept a new pixel (or pixel packet) on every clock cycle.
- Adding operators **can** increase **latency**. Each operator may add delay on the path of a **single** pixel, and those delays add up along the chain. A few clock cycles from O-type operators are typically small compared with the line or frame delay of a buffer.
- Latency is **not the same** as blocking or buffer fill level. If downstream processing is slow, buffers can fill up and the delay grows further. This topic describes the **minimum latency** built into the operators themselves.

7.3 Example Design

The following design is used throughout this topic. It shows a simple acquisition path with one buffer and two processing steps:



The latency annotations in the design above apply to this specific example:

Operator	Latency in this example
CxpCamera	No additional algorithmic latency in this example
LineBuffer	1 line
ScaleByN	1 clock cycle
ShiftRight	No additional latency
DmaToPC	No additional algorithmic latency in this example

The total latency for a pixel on this path is dominated by the **one-line delay** of `LineBuffer`, plus the small processing delays of the other operators. "No additional algorithmic latency" for `CxpCamera` and `DmaToPC` means those operators add no extra delay of that kind in this example. It does not mean that the pixel appears immediately on the PC. Camera interface timing, PCIe transfer, host buffering, and display are outside this table.

7.4 Pixel Flow in the Example Design

In the example design, pixels are transferred from the camera through the camera interface into `LineBuffer`. The `LineBuffer` collects one full line before it forwards data to the output.

After a full line is stored in `LineBuffer`, the buffer outputs the pixels of that line. At the **same time**, the camera already transfers the **next line**. This is the pipelined behavior of the generated FPGA hardware: acquisition, buffering, processing, and DMA can overlap. VisualApplets describes that pipeline; it does not process the camera stream itself.

7.5 Calculating Latency

Latency depends on the **operator**. There is no single value for an entire design. You add up (or follow in time) the delays along the path of one pixel.

7.5.1 Line-Based Latency (Buffer Operators)

Buffer operators such as `ImageBuffer` and `LineBuffer` need at least **one full line** before they can output the first pixel of that line. For `LineBuffer` in the example above:

Latency = 1 line

Measured in time, the latency of one line is:

Line latency = (pixels per line) / (pixel rate of the link)

The pixel rate depends on parallelism and the Base Design Clock. A wider line or a slower link increases the time until the first pixel leaves the buffer.

7.5.2 Clock-Cycle Latency (Processing Operators)

Many processing operators delay a pixel by a **fixed number of clock cycles**. After the pipeline has filled, they can still accept and output one pixel (or one parallel packet of pixels) on every clock cycle. The delay of a single pixel is then only those internal register stages.

In the example design, `ScaleByN` adds:

Latency = 1 clock cycle

Measured in time, the latency of one clock cycle is:

Cycle latency = 1 / Base Design Clock frequency

For example, at a Base Design Clock of 312.5 MHz, one clock cycle is 3.2 ns.

Operators, such as `ShiftRight` in this example, add **no meaningful latency** in practice. They forward results as soon as the input value is available.

7.5.3 Operator Types and Latency

VisualApplets groups operators into types. The type helps you estimate the latency:

Type	Icon Shape	Typical Latency Behavior
O-type	Round	Very small, fixed delay — usually a constant number of clock cycles. Example: a delay of about 3.2 ns (one cycle at 312.5 MHz Base Design Clock).
M-type	Square	Variable delay. Can buffer lines or frames, change image dimensions, or block the input. Check the operator documentation for the exact latency.
P-type	Square	Similar to M-type, but does not actively block the input. Check the operator documentation for the exact latency.

`CxpCamera`, `LineBuffer`, and `DmaToPC` are M-type operators. `ScaleByN` and `ShiftRight` are O-type operators. Even though `DmaToPC` is M-type, it does not add a large algorithmic delay in the example — it hands data to DMA as soon as data arrives. The time until the image is visible in host software is a separate, later delay.

Key Terms

Think of O-type operators as short pipe sections with a fixed length. Think of M-type operators as sections that may contain a storage tank (buffer) or more complex machinery. The tank must fill before the first drop comes out — that is line-based latency. Those pipes and tanks are hardware stages on the FPGA, generated from the modules you placed in VisualApplets.

7.6 Parallelism and Latency

Parallelism does not remove latency. It increases **bandwidth**: More pixels per clock cycle move through the links.

Example: With parallelism 4, four pixels pass through a link in one clock cycle. The **first pixel** of a frame still needs the same operator delays, for example, one line in `ImageBuffer`. After the pipeline is full, four new pixels can leave the operator on every clock cycle.

- **High parallelism** → high throughput (bandwidth).
- **Buffer and algorithm** → latency of the first pixel.

Both properties are independent. VisualApplets designs aim for high bandwidth **and** minimum latency in the generated FPGA pipeline.

7.7 Finding Latency Values for Operators

To find the latency values for the operators you want to use in your design, use these sources:

1. **Operator Reference** — Each operator documents its latency (or processing delay) in the [Operator Reference](#).
2. **In-product help** — Select the operator in the design and open the help for parameter and timing details. This is the same content as in the Operator Reference, provided as in-tool help in VisualApplets.

For M-type operators especially, always check the documentation. Latency can depend on parameters (for example, kernel size in filter operators, or buffer depth in memory operators).

Most operators don't store a full image. They calculate output results as soon as the information is available and the output isn't blocked. That is why latency in the generated FPGA pipeline is kept close to the theoretical minimum for the chosen algorithm.

If the output is blocked, for example, because the PC can't accept data fast enough, the buffer fill level increases and the time until a pixel appears at the host increases. That additional delay is not a fixed operator latency — it comes from **flow control** and buffering under load.

7.8 Related Topics

- [Pipeline Concept](#) Tutorial topic — unbuffered flow and concurrent processing
- [Parallelism](#) Tutorial topic — parallelism and latency
- [Operator Types and Synchronization](#) Tutorial topic — operator types and latency
- [Memory Elements](#) Tutorial topic — line latency of buffer operators
- [Working with Kernels](#) Tutorial topic — kernel-related delay

- [Bandwidth of an Applet](#) in the User Manual — bandwidth versus latency
- [Operator Reference](#) — latency values of individual operators
- [Device Resources](#) — Base Design Clock frequencies

8. Parameterization

This topic explains which kind of parameters you can configure in a VisualApplets design, how VisualApplets distinguishes **link properties** from **module parameters**, how link properties propagate through a pipeline, and how you resolve conflicts between them. After you have read this topic, you understand which properties you set on links and which parameters you set on operators, how static and dynamic module parameters differ, and how you adapt a design to a specific camera format without violating the design rules.



Key Terms

A **link** is the connection between two modules. **Link properties** describe the data format on that connection, for example bits per pixel or maximum image size. You set them only in VisualApplets during design time. **Module parameters** are the settings on an operator itself, for example a threshold value or a camera format. If a value that you enter doesn't stick, link properties often limit what the module can accept.

Static parameters can be changed **only** in VisualApplets on the PC (design time). **Dynamic** parameters can also be changed at **runtime**, when host software talks to the applet on the frame grabber. Values you set for dynamic parameters in VisualApplets are the defaults for runtime. Simulation in VisualApplets is not runtime.

8.1 Link Properties and Module Parameters

VisualApplets uses two kinds of settings:

Link properties are set on **links** and define e.g. maximum image width and height, bit width, parallelism, image protocol, color format, and kernel size. Link properties constrain the operators that are connected to the link. If a module parameter conflicts with link properties, VisualApplets either adjusts the value directly or reports an error. Link properties are always static: You change them only in VisualApplets during the design phase. Link properties propagate through the pipeline and affect all downstream operators until an operator changes them.

Module parameters define **operator behavior**, for example an operation mode, an image position, or the size of a region of interest. They can be **static** or **dynamic**. You change static parameters only during design time. Dynamic parameters can also be changed at runtime from host software. Static parameters typically use fewer FPGA resources than dynamic parameters. Dynamic parameters are useful when you need to switch camera formats or processing settings while the applet runs. Module parameters are often limited by link properties. If you can't set a value and VisualApplets auto-corrects it, check the link properties on the connected links. Many operators also define link properties on their output, so a change on one side of the design can propagate to the other.

Module Properties and Link Properties of the CxpCamera Operator:

CxpCamera

LineBuffer

FIRkernelNxM

MEDIAN

DmaToPC

CxpCamera

LineBuffer

FIRkernelNxM

MEDIAN

DmaToPC

Module Properties: Process0/CxpCamera <CxpCamera>

Parameter Name	Parameter Value	Parameter Unit	Parameter Type	Parameter Flag	Value Type
Name	CxpCamera				string
ConnectionCount	X1	lane	Static	Write	enum
ResetStatus	Off		Dynamic	Write	enum
UsedConnections	1	lane	Dynamic	Read	int (unsigned, 3 bit)
PacketTagErrorCount	0		Dynamic	Read	int (unsigned, 13 bit)
ImageTagErrorCount	0		Dynamic	Read	int (unsigned, 13 bit)
StreamIdErrorCount	0		Dynamic	Read	int (unsigned, 13 bit)
CorrectedErrorCount	0		Dynamic	Read	int (unsigned, 13 bit)
UncorrectedErrorCount	0		Dynamic	Read	int (unsigned, 13 bit)
PacketBufferOverflowCount	0		Dynamic	Read	int (unsigned, 13 bit)
PacketBufferOverflowSource	0		Dynamic	Read	int (unsigned, 4 bit)
CameraScanMode	area		Dynamic	Read	enum
MarkerErrorCount	0		Dynamic	Read	int (unsigned, 13 bit)
UnexpectedStartupData	false		Dynamic	Read	enum
FrameLostCount	0		Dynamic	Read	int (unsigned, 25 bit)
FrameCorruptedCount	0		Dynamic	Read	int (unsigned, 25 bit)

HelpAdd MetadataDelete MetadataApplyClose

Link properties

Source

Operator
Module
Port

CxpCamera
CxpCamera
0

Destination

Operator
Module
Port

LineBuffer
LineBuffer
1

	Value	Source Port	Destination Port
Bit Width	8	8	8
Arithmetic	unsigned	unsigned	unsigned
Parallelism	4	4	4
Kernel Columns	1	1	1
Kernel Rows	1	1	1
Image Protocol	VALT_IMAGE2D	VALT_IMAGE2D	VALT_IMAGE2D
Color Format	VAF_GRAY	VAF_GRAY	VAF_GRAY
Color Flavor	FL_NONE	FL_NONE	FL_NONE
Max. Image Width	1032	1032	1032
Max. Image Height	1032	1032	1032

HelpApplyClose

For the formal rules behind links, see [Rules of Links](#) in the User Manual. For the editing workflow in the VisualApplets GUI, see [Parameterization](#) in the User Manual.

8.2 Link Properties

Each link between modules is defined by its link properties. All links have the same set of properties. Link properties are always static: You change them only in VisualApplets during the design phase. You open the link properties by double-clicking the link or by right-clicking the link and selecting **Properties**.

The following link properties define how image data is transported on a connection:

- **BitWidth**: The number of bits per pixel on the link. For color links, the bit width covers all color components together. Camera operators often set the initial bit width. For links with `ImageProtocol = VALT_SIGNAL`, the bit width is always 1 bit.
 - Some operators let you set the output `BitWidth` directly on the link, for example `CastBitWidth`, `CONST`, or `ModuloCount`. Others derive the output `BitWidth` automatically from input link properties or module parameters. For example, `ShiftLeft` adds the value of parameter `Shift` to the input bit width, and `MULT` uses the sum of the input bit widths. Operators such as `Coordinate_X` and `Coordinate_Y` calculate the output `BitWidth` from `MaxImageWidth` and `MaxImageHeight`. For operators with automatic output `BitWidth`, the [Operator Reference](#) shows **auto** in the **Supported Link Format** table, and you usually can't edit the value on the link. When an operator changes the output `BitWidth`, VisualApplets propagates the new value to downstream links until another operator changes it again.
- **Arithmetic**: Defines the arithmetic used for a link. The arithmetic can be either `Signed` or `Unsigned`.
 - If set to `Signed`, two's complement arithmetic is used. The sign bit is included in the configured bit width.
 - For the image protocol `VALT_SIGNAL`, the arithmetic is always `Unsigned`, regardless of the configured bit width.
- **Parallelism**: The number of pixels that are transferred in parallel between two modules within one Base Design Clock cycle. For more information, see [Parallelism](#) Tutorial topic.
- **KernelColumns** and **KernelRows**: The number of neighboring pixels per clock cycle that filter operators need. For the image protocol `VALT_SIGNAL`, both values are always 1. For more information, see [Working with Kernels](#) Tutorial topic.
- **ImageProtocol**: Defines the structure of the data stream on the link. The following image protocols are available:
 - `VALT_IMAGE2D`: Two-dimensional images with end-of-line and end-of-frame markers, for example from area scan cameras.
 - `VALT_LINE1D`: One-dimensional line data, for example from line scan cameras.
 - `VALT_PIXEL0D`: A continuous pixel stream without image dimension information.
 - `VALT_SIGNAL`: Single-bit signals that are valid at every clock cycle, for example in trigger logic.

The image protocol can change inside a design. For example, `SplitImage` can cut a 1D image stream into 2D images of a defined height. For more information, see [Image Protocols, Image Dimensions and Data Structure](#) in the User Manual.

- **ColorFormat** and **ColorFlavor**: These properties describe how pixel data is represented, for example gray, or RGB. Some operators support only certain color formats. On some platforms, CoaXPress camera operators accept gray formats only. To produce color output, use operators such as `MergeComponents`. Operators such as `MergeComponents`, `ParallelUp`, and `CastBitWidth` can change color-related link properties.
- **MaxImageWidth** and **MaxImageHeight**: The maximum image dimensions that the link can carry. Images on the link must not exceed these values, but they may be smaller. These properties are relevant for operators that buffer lines or full images, such as `ImageFifo`, `LineBuffer`, or `CoIMax`. Set them at least as large as the largest image in the processing path. Values that are too small cause Design Rules Check (DRC) errors. Larger values use more FPGA resources. Camera operators or operators such as `SetDimension` often set these properties. `MaxImageHeight` has no effect when `ImageProtocol` is set to `VALT_LINE1D`, `VALT_PIXEL0D`, or `VALT_SIGNAL`.

Info

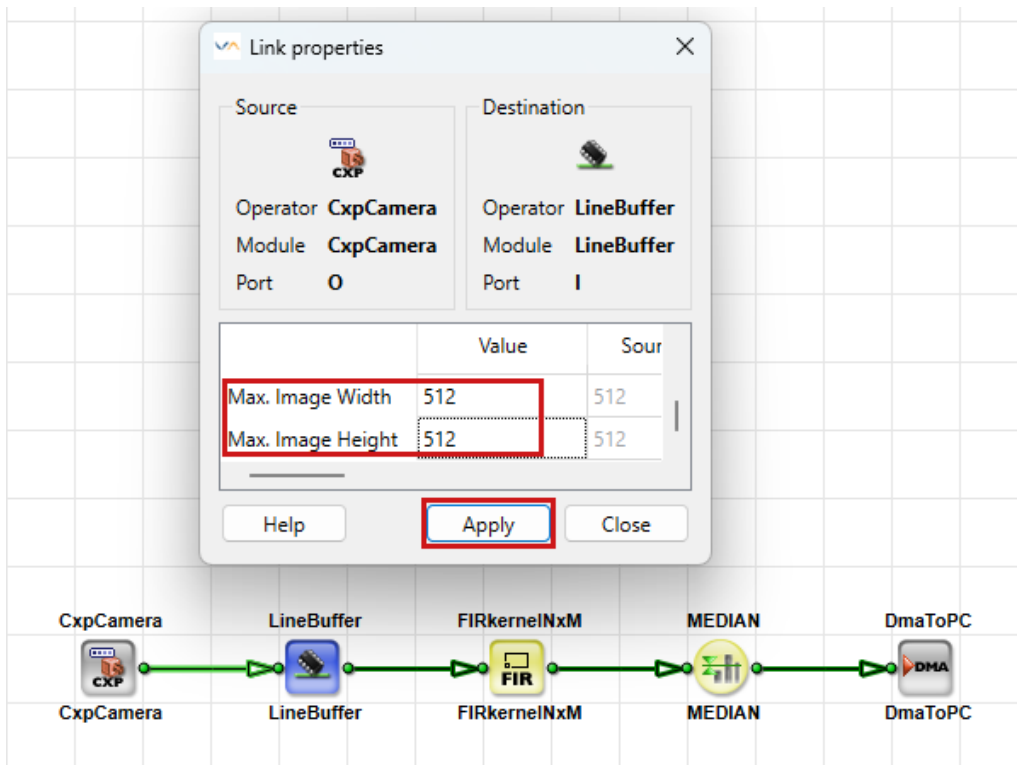
The higher the maximum image dimensions and the parallelism you choose, the more FPGA resources the design consumes. Basler recommends that you set these values only as high as your application requires.

Each operator defines which output link properties it allows and which values are fixed. Some properties on a link are disabled because the upstream operator doesn't permit editing them. The [Operator Reference](#) documents the allowed link formats for each operator.

8.3 Propagation of Link Properties

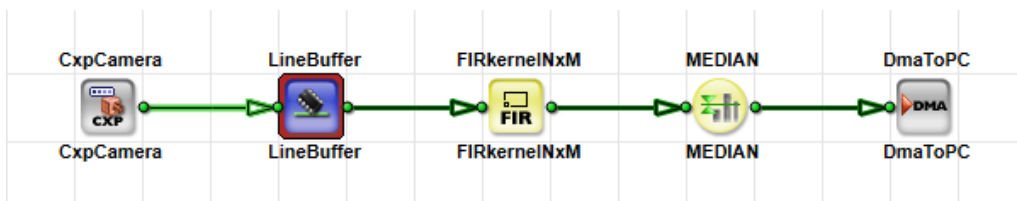
Link properties are defined by operators. You can change some properties directly on the output link of an operator. Others are fixed or are derived from module parameters or from input link properties. When you change a link property, VisualApplets propagates the change through successive modules and links.

After you change a value, click **Apply** in the properties dialog. If the change is accepted, affected links turn **green** in the design pane. Green means that VisualApplets has propagated and validated the updated link properties. It is not a traffic-light "all valid" mark for the whole design.



Link properties propagate through modules until an operator changes it. If the downstream link doesn't allow editing a property, the upstream operator or an intermediate operator that can change that property is the place to set it. For example, an `ImageBuffer` operator doesn't change the link bit width. You must set the bit width on the camera output link or insert an operator such as `ConvertPixelFormat` or `ShiftLeft` that changes the bit width.

A link property change can also conflict with a module parameter. In that case, VisualApplets marks the module and the affected parameters in **red**:



Run **Design Rules Check Level 1** and read the DRC log to locate the error. Typical causes include a region of interest that exceeds `MaxImageWidth` or `MaxImageHeight`, or a parameter value outside the range allowed by the link bit width:

DRC Log

Design Rules Check Level 1

Error: 1700 Module [+] `LineBuffer(LineBuffer)`. Invalid parameter value: `XOffset` is not accepted.

Error: 1701 Module [+] `LineBuffer(LineBuffer)`. Invalid parameter value: `XLength` is not accepted. Value out of range.

Error: 1700 Module [+] `LineBuffer(LineBuffer)`. Invalid parameter value: `YOffset` is not accepted.

Error: 1701 Module [+] `LineBuffer(LineBuffer)`. Invalid parameter value: `YLength` is not accepted. Value out of range.

4 Errors
0 Warnings

13:19:50 -- **Design Rules Check Level 1 failed!**

```

graph LR
    CxpCamera[CxpCamera] --> LineBuffer[LineBuffer]
    LineBuffer --> FIRkernelNxM[FIRkernelNxM]
    FIRkernelNxM --> MEDIAN[MEDIAN]
    MEDIAN --> DmaToPC[DmaToPC]
  
```

8.4 Module Parameters

Module parameters differ for each operator. They control how the operator processes data. Module parameters can be static or dynamic. You set static parameters only in VisualApplets during design time. Dynamic parameters can also be changed at runtime from host software through microDisplay X, the Framegrabber SDK, or pylon. In the **Module Properties** dialog, the **Parameter Type** column shows whether a parameter is static or dynamic:

Module Properties: Process0/LineBuffer <LineBuffer>

Parameter Name	Parameter Value	Parameter Unit	Parameter Type	Parameter Flag	Value Type
Name	LineBuffer				string
RamDataWidth	384		Static	Write	int (unsigned, 9 bit)
XOffset	0	pixel	Dynamic	Write	int (unsigned, 10 bit)
XLength	1024	pixel	Dynamic	Write	int (unsigned, 11 bit)
YOffset	0	pixel	Dynamic	Write	int (unsigned, 10 bit)
YLength	1024	pixel	Dynamic	Write	int (unsigned, 11 bit)
ParallelismConverter	No		Static	Write	enum
InfiniteSource	DISABLED		Static	Write	enum
WritePriority	DISABLED		Static	Write	enum
FillLevel	0	%	Dynamic	Read	int (unsigned, 25 bit)
MaxLineCount	1458888	line	Dynamic	Read	int (unsigned, 25 bit)
LineCount	0	line	Dynamic	Read	int (unsigned, 25 bit)
Overflow	0		Dynamic	Read	int (unsigned, 2 bit)
OverflowClearMode	AutoClear		Dynamic	Write	enum

Help Add Metadata Delete Metadata Apply Close

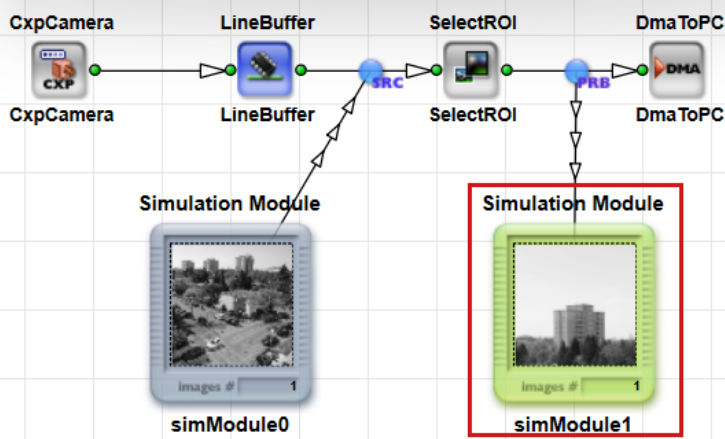
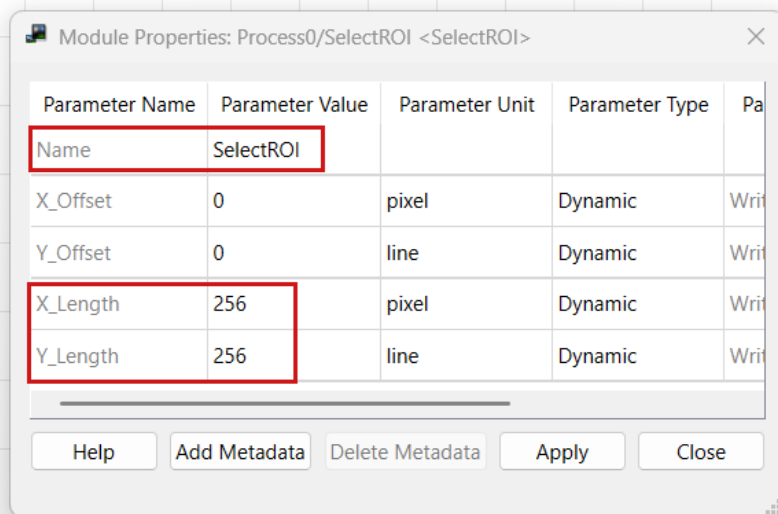
After you edit a parameter, click **Apply** in the **Module Properties** dialog. VisualApplets discards a new value if it isn't within the allowed range. Some parameters are disabled and can't be edited. Whether a parameter is enabled depends on other parameter settings, or on the input or output link format. For details about the module properties of a specific operator, check the [Operator Reference](#) for the operator you use.

For a walkthrough that sets link and module parameters in a sample design, see [Hardware Applet: From Idea to Application](#) in the User Manual.

8.4.1 Dimension Parameters

Many operators, such as `LineBuffer`, `FrameBufferMultiRoi`, and `SelectROI`, define a region of interest (ROI) through the parameters `XLength`, `YLength`, `XOffset`, and `YOffset`. The region of interest must fit within the maximum image dimensions on the connected links. If `XLength + XOffset` exceeds `MaxImageWidth`, or `YLength + YOffset` exceeds `MaxImageHeight`, DRC reports an error and the parameters are marked red.

In the following screenshot, the `SelectROI` operator defines an ROI through the `XLength` and `YLength` parameters:



8.4.2 Infinite Sources and the InfiniteSource Parameter

Operators such as camera modules can't be stopped by flow control. They behave as **infinite sources** and must connect only to M-type operators that accept infinite sources, or to O-type and P-type operators that propagate the infinite-source property downstream.

Buffer operators convert an infinite stream into a controllable, stoppable stream. When a buffer receives data from a non-stoppable source, set its `InfiniteSource` parameter to `ENABLED`. When the input is already a stoppable stream, set `InfiniteSource` to `DISABLED`.

Design Rules Check with InfiniteSource Set to DISABLED After an Infinite Source:

Module Properties: Process0/LineBuffer <LineBuffer>

Parameter Name	Parameter Value	Parameter Unit	Parameter Type
Name	LineBuffer		
RamDataWidth	384		Static
XOffset	0	pixel	Dynamic
XLength	1024	pixel	Dynamic
YOffset	0	pixel	Dynamic
YLength	1024	pixel	Dynamic
ParallelismConverter	No		Static
InfiniteSource	DISABLED		Static
WritePriority	DISABLED		Static

enum (ENABLED=0, DISABLED=1)

Help Add Metadata Delete Metadata Apply Close

CxpCamera LineBuffer ImageFifo DmaToPC
CxpCamera LineBuffer ImageFifo DmaToPC

DRC Log

Resource	Count	Usage
Embedded ALU	47	~ 3%

09:42:24 -- Netlist successfully generated in 00:00:05.

Design Rules Check Level 2

Program version: VisualApplets 3.7.100.76452M-9d0b71df5
Design name:
Creation date: Donnerstag, 2. Juli 2026 09:42:24

00:00:05 -- Error: -150 Checking output latency: Error at module [+] LineBuffer (LineBuffer). Module cannot buffer data stream / infinite source.

1 Error
0 Warnings

09:42:24 -- Design Rules Check Level 2 failed!

Design Rules Check with InfiniteSource Set to ENABLED After an Infinite Source:

Module Properties: Process0/LineBuffer <LineBuffer>

Parameter Name	Parameter Value	Parameter Unit	Parameter Type
Name	LineBuffer		
RamDataWidth	384		Static
XOffset	0	pixel	Dynamic
XLength	1024	pixel	Dynamic
YOffset	0	pixel	Dynamic
YLength	1024	pixel	Dynamic
ParallelismConverter	No		Static
InfiniteSource	ENABLED		Static
WritePriority	DISABLED		Static

enum (ENABLED=0, DISABLED=1)

Help Add Metadata Delete Metadata Apply Close

CxpCamera LineBuffer ImageFifo DmaToPC
CxpCamera LineBuffer ImageFifo DmaToPC

DRC Log

Resource	Count	Usage
Flip Flops	53458	~ 16%
Block RAM	126	~ 17%
Ultra RAM	0	~ 0%
Embedded ALU	47	~ 3%

09:42:58 -- Netlist successfully generated in 00:00:05.

Design Rules Check Level 2

Program version: VisualApplets 3.7.100.76452M-9d0b71df5
Design name:
Creation date: Donnerstag, 2. Juli 2026 09:42:58

0 Errors
0 Warnings

09:42:58 -- Design Rules Check Level 2 successfully performed.

For more detailed information about infinite sources, see [Stoppable and Non-Stoppable Data Streams](#) Tutorial topic.

The Overflow Operator

The `Overflow` operator in the **Synchronization** library handles overflow situations when a non-stoppable source keeps sending image data while downstream processing is blocked. It doesn't buffer data itself. Basler recommends that you use it together with a buffer operator. This operator is not the same as **arithmetic overflow** (see [Bit Arithmetic](#) Tutorial topic) or the `Overflow` **status parameter** on buffer operators (see [Memory Elements](#) Tutorial topic).

When you place `Overflow` between an infinite source and a memory operator such as `LineBuffer`, set `InfiniteSource` on the memory operator to `DISABLED`. The `Overflow` operator already handles overflow from the non-stoppable stream. Enabling `InfiniteSource` on the memory module in this topology is incorrect and triggers DRC errors.

The screenshot displays the **Module Properties: Process0/LineBuffer <LineBuffer>** dialog box, the **DRC Log** window, and a block diagram of the design topology.

Module Properties: Process0/LineBuffer <LineBuffer>

Parameter Name	Parameter Value	Parameter Unit	Parameter Type
Name	LineBuffer		
RamDataWidth	384		Static
XOffset	0	pixel	Dynamic
XLength	1024	pixel	Dynamic
YOffset	0	pixel	Dynamic
YLength	1024	pixel	Dynamic
ParallelismConverter	No		Static
InfiniteSource	DISABLED		Static
WritePriority	DISABLED		Static

enum (ENABLED=0, DISABLED=1)

DRC Log

Flip Flops	53509	~ 16%	
Block RAM	125	~ 17%	
Ultra RAM	0	~ 0%	
Embedded ALU	48	~ 3%	

12:26:35 -- **Netlist successfully generated in 00:00:05.**

Design Rules Check Level 2

Program version: VisualApplets 3.7.100.76452M-9d0b71df5
Design name:
Creation date: Donnerstag, 2. Juli 2026 12:26:35

0 Errors
0 Warnings

12:26:35 -- **Design Rules Check Level 2 successfully performed.**

Block Diagram: CxpCamera → Overflow → LineBuffer → DmaToPC

8.5 Verifying Your Design with DRC

After you change link properties or module parameters that affect image size, bit width, or regions of interest, run **Analysis → Design Rules Check Level 1**. Level 1 reports formal errors in the DRC log and highlights affected modules and links.

Design Rules Check Level 2 runs extended checks and estimates FPGA resources. Fix all Level 1 errors before you build. Level 2 also checks infinite-source connections and `InfiniteSource` settings.

Read the **DRC log** together with highlighted modules and links. For the full workflow, see [From Empty Design to Running Applet](#) Tutorial topic.

DRC Log

0 Errors
0 Warnings

10:00:23 -- **Design Rules Check Level 1 successfully performed.**

Check Build Preconditions

Creation date: Monday, 11 April 2022 10:00:23

10:00:23 -- **All checks successfully performed.**

Netlist Generation

Creation date: Monday, 11 April 2022 10:00:23

Netlist generation started...

Info: Building design for FPGA clock frequency 125 MHz

FPGA resource estimation:

Resource	Count	Fill Level
LookupTables	37132	~ 36%
Flip Flops	44960	~ 22%
Block RAM	103	~ 15%
Embedded ALU	14	~ 2%

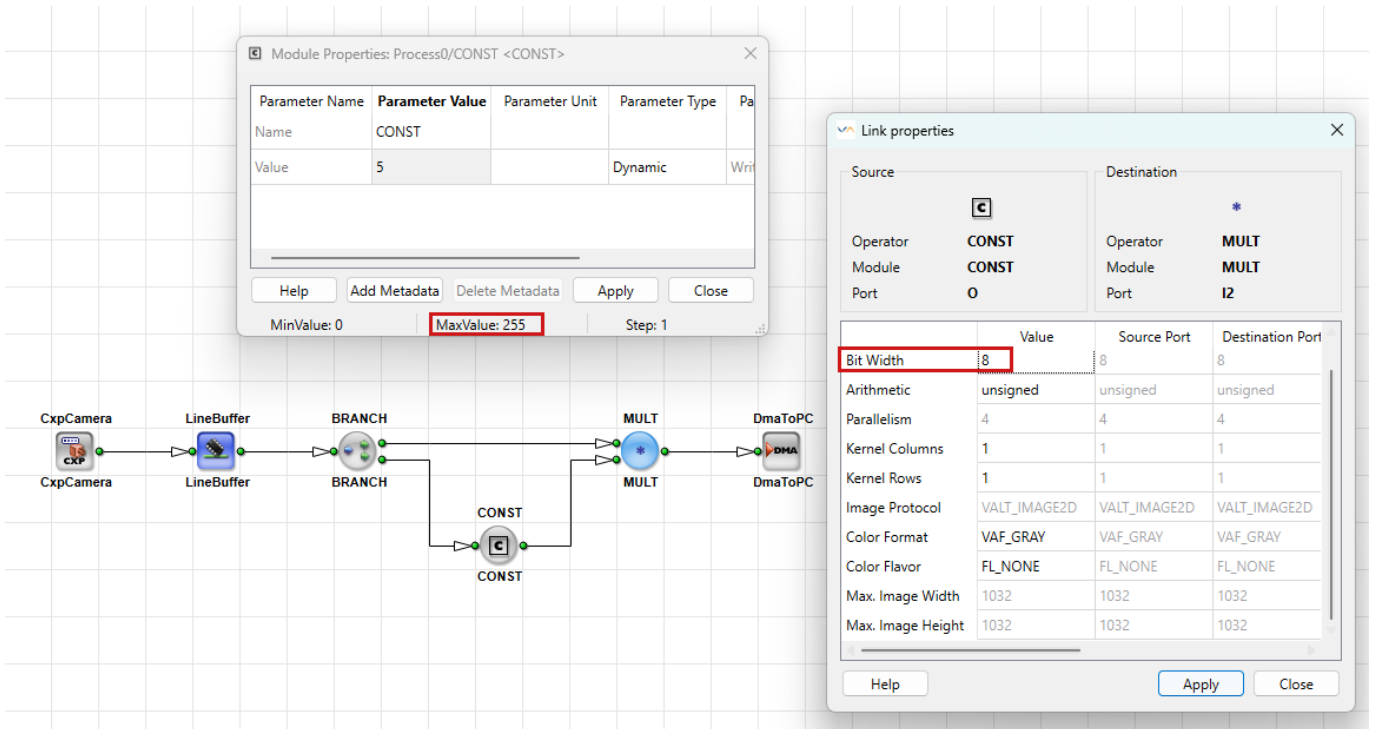
10:00:26 -- **Netlist successfully generated in 00:00:03.**

8.6 Troubleshooting Link and Parameter Conflicts

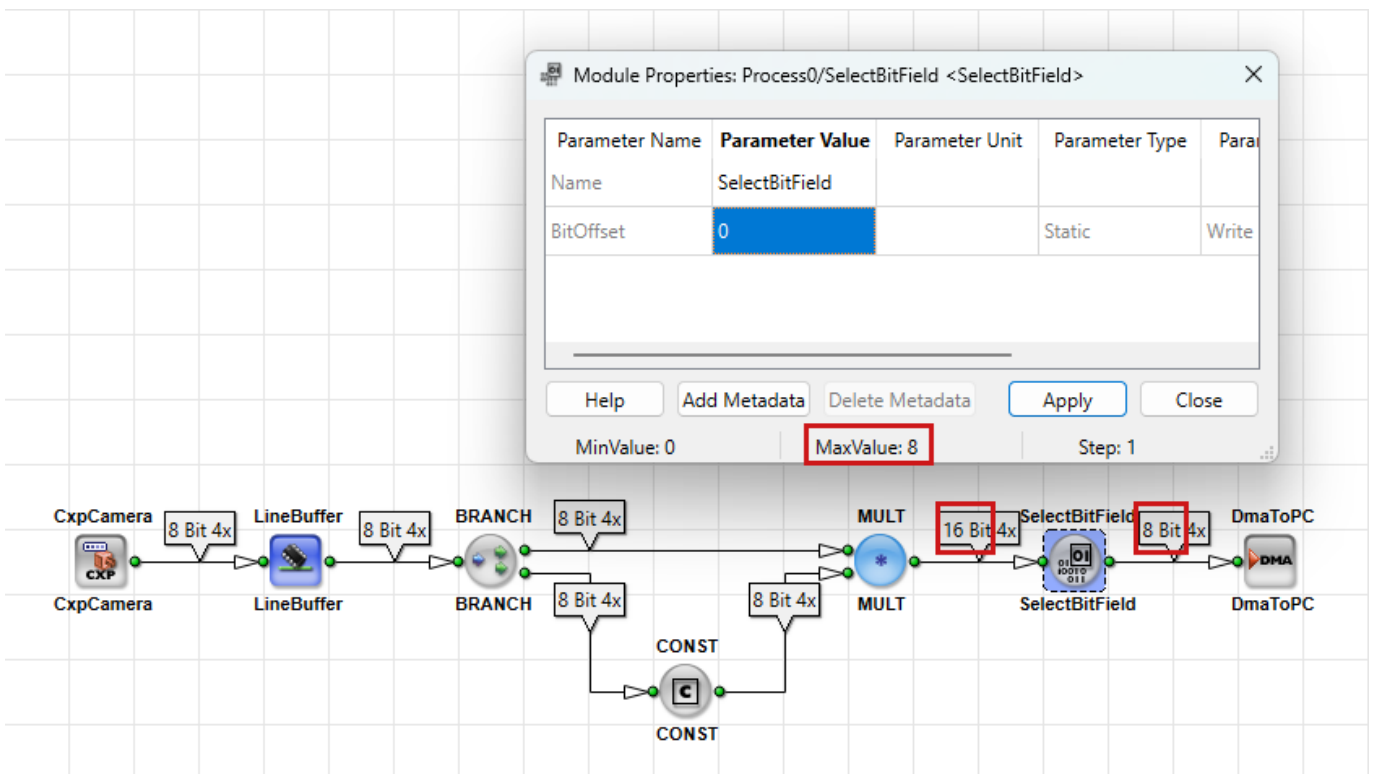
Module parameters and link properties aren't independent settings. They directly depend on each other and mutually limit their allowed value ranges. Link properties on the input and output connections define the data format that an operator must work with. Module parameters define how the operator processes data within that format. When you change a link property, VisualApplets propagates the change through successive modules and links. If a module parameter no longer fits the updated link properties, VisualApplets marks the module or the parameter in red. The same applies in the other direction: a module parameter can define or change output link properties, so a value that was valid before can become illegal after a link change. Whether a parameter is enabled, disabled, or limited to a specific range therefore depends on link properties, on other module parameters, or on both. For each operator, the [Operator Reference](#) documents these dependencies. For the propagation behavior and the three possible outcomes of a link change, see [Propagation and Dependencies of Operator Parameters and Link Properties](#) in the User Manual.

When a parameter value doesn't stick or a module turns red after a link change, resolve conflicts in this order: Check **link properties before module parameters**. Link properties define the allowed value range for module parameters. Module parameters must fit inside that range. Many operators also define or change output link properties, so a change on one side of the design can affect the other. Read the DRC log, click an error message to highlight the affected module or link, and consult the [Operator Reference](#) for the allowed link formats and parameter dependencies. Red highlighting on links or in the **Module Properties** dialog indicates a formal error that you must fix before simulation or build.

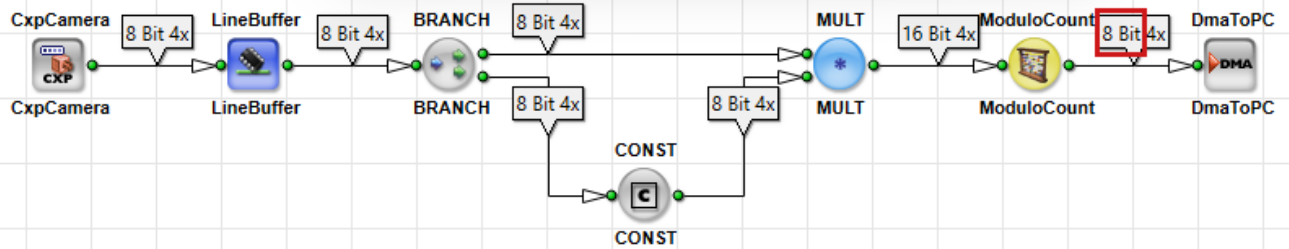
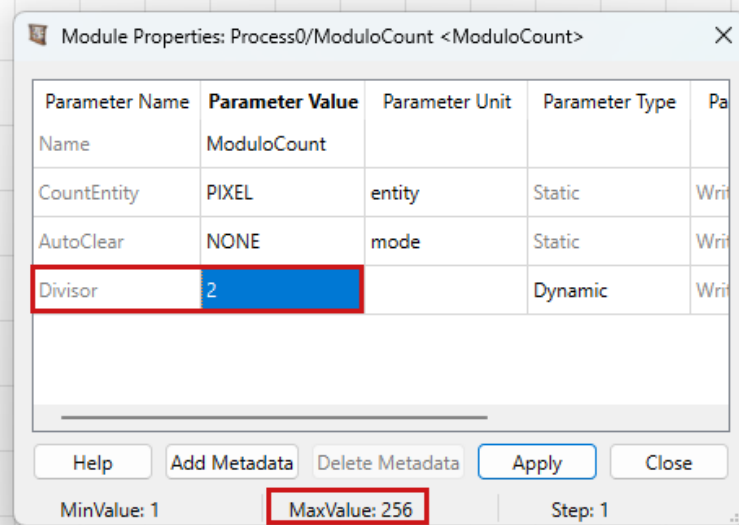
A typical conflict occurs when you try to set the `Value` parameter of a `CONST` operator e.g. to 500, but VisualApplets auto-corrects the entry. The outgoing link has a `Bit Width` of 8 bits, which allows only values from 0 to 255. To fix the conflict, increase the link `Bit Width` on the outgoing link, then set the module parameter again:



Similar dependencies appear with other operators. On **SelectBitField**, the valid range of the **BitOffset** parameter depends on the input and output link **Bit Width**s. For gray images, **BitOffset** must be in the range from 0 to input **BitWidth** minus output **BitWidth**:



On the `ModuloCount` operator, the `Divisor` parameter is limited by the output link `Bit Width` to values from 1 to 2^n , where n is the output `Bit Width`:



Region-of-interest parameters such as `XLength` and `YLength` must fit within `MaxImageWidth` and `MaxImageHeight` on the connected links. Conflicts can also involve parallelism, color format, and image protocol. For example, `XLength` must always be a multiple of the parallelism, which is indicated in the Module Properties dialog as **Step Size**:

The screenshot displays two dialog boxes and a block diagram. The **Link properties** dialog shows the connection between **CxpCamera** (Source) and **LineBuffer** (Destination). The **Module Properties: Process0/LineBuffer <LineBuffer>** dialog shows the parameters for the **LineBuffer** module. In both dialogs, `Parallelism` is set to 4, `Max. Image Width` and `Max. Image Height` are set to 1032, and `XLength` and `YLength` are set to 1024. The **Module Properties** dialog also shows `MinValue: 4`, `MaxValue: 1032`, and `Step: 4`. The block diagram below shows the data flow from **CxpCamera** through **LineBuffer**, a **BRANCH** node, a **CONST** node, a **MULT** node, a **ModuloCount** node, and finally a **DmaToPC** node.

Link properties

Source		Destination	
Operator	CxpCamera	Operator	LineBuffer
Module	CxpCamera	Module	LineBuffer
Port	0	Port	1

	Value	Source Port	Dest Port
Bit Width	8	8	8
Arithmetic	unsigned	unsigned	unsigned
Parallelism	4	4	4
Kernel Columns	1	1	1
Kernel Rows	1	1	1
Image Protocol	VALT_IMAGE2D	VALT_IMAGE2D	VALT_IMAGE2D
Color Format	VAF_GRAY	VAF_GRAY	VAF_GRAY
Color Flavor	FL_NONE	FL_NONE	FL_NONE
Max. Image Width	1032	1032	1032
Max. Image Height	1032	1032	1032

Module Properties: Process0/LineBuffer <LineBuffer>

Parameter Name	Parameter Value	Parameter Unit	Parameter Type
Name	LineBuffer		
RamDataWidth	384		Static
XOffset	0	pixel	Dynamic
XLength	1024	pixel	Dynamic
YOffset	0	pixel	Dynamic
YLength	1024	pixel	Dynamic
ParallelismConverter	No		Static

Buttons: Help, Add Metadata, Delete Metadata, Apply, Close

MinValue: 4, MaxValue: 1032, Step: 4

Block Diagram:

```

graph LR
    CxpCamera[CxpCamera] --> LineBuffer[LineBuffer]
    LineBuffer --> BRANCH1[BRANCH]
    BRANCH1 --> BRANCH2[BRANCH]
    BRANCH2 --> CONST1[CONST]
    CONST1 --> MULT1[MULT]
    MULT1 --> ModuloCount[ModuloCount]
    ModuloCount --> DmaToPC[DmaToPC]
  
```

Key Terms

Basler recommends the following practices when you work with parameters:

- Use dynamic parameters when you need runtime flexibility, for example for threshold values. Runtime means host software talking to the applet on the frame grabber, not simulation in VisualApplets.
- After each change to link properties or module parameters, check the propagated link properties on downstream links.
- Plan bit-width adjustments for downstream software, for example widening 12-bit camera data to 16 bits per pixel before the DMA.
- When a value can't be set, check link properties on the connected links before you change module parameters.
- Run DRC regularly to catch conflicts between links and modules early.

8.7 Related Topics

- [Parameterization](#) in the User Manual — editing link properties and module parameters in the GUI
- [Example: Applet Parameterization in Basics and Examples](#) in the User Manual — worked parameterization example
- [Rules of Links](#) in the User Manual — formal link rules
- [Parallelism](#) Tutorial topic — `Parallelism` link property
- [Working with Kernels](#) Tutorial topic — `KernelColumns` and `KernelRows`
- [Bit Arithmetic](#) Tutorial topic — `BitWidth` and arithmetic overflow
- [Stoppable and Non-Stoppable Data Streams](#) Tutorial topic — `InfiniteSource`
- [Operator Types and Synchronization](#) Tutorial topic — operator types and link formats
- [From Empty Design to Running Applet](#) Tutorial topic — setting parameters at design time

9. Bit Arithmetic

This topic explains how bit widths behave in the FPGA pipeline generated from a VisualApplets design. After you have read this topic, you understand how link bit width limits arithmetic results, how you handle overflow, and how you use cast, clip, and shift operators to control value ranges. You also know how signed and unsigned values differ, how fixed-point numbers work, and how you extract bit fields from wider data words.

Key Terms

Bit width is the number of bits per pixel or per value on a link. The bit width defines the **value range**. For example, an 8-bit unsigned link carries values from 0 to 255. **Arithmetic overflow** means that an arithmetic result no longer fits into the available bits. The result wraps or loses high-order bits.

For link properties and parameter dependencies, see [Parameterization](#) Tutorial topic. For the formal rules behind links, see [Rules of Links](#) in the User Manual.

9.1 Link Bit Width Limits and Operator Constraints

Each **link** between two modules (i.e. instantiated operators) carries a **Bit Width** property. This property defines how many bits the generated FPGA pipeline uses for each pixel or value on that connection. For color links, the bit width covers all color components together. On links with `ImageProtocol = VALT_SIGNAL`, the bit width is always 1 bit.

A link supports at most **64 bits** per data word. When an operator calculates a wider result, you must clip, cast, or shift the data before the link width exceeds this limit.

Not every operator accepts every bit width. Some operators allow only specific ranges such as 1 to 64 bits, 8 bits, 12 bits, or 16 bits. The **Supported Link Format** table in the [Operator Reference](#) documents the allowed values for each operator. You can also open operator help from the design pane with a right-click on a module and **Help**, or press **F1**. This is the same content as in the Operator Reference, provided as in-tool help in VisualApplets.

When a bit width conflict appears, check the output link of the upstream operator first. Adjust the **Bit Width** link property on an operator that lets you edit it, or insert an operator such as `CastBitWidth`, `ClipHigh`, `ClipLow`, `ShiftLeft`, or `ConvertPixelFormat` to change the bit width along the path.

Link properties propagate through the pipeline until an operator changes them. Set the width on the camera output link or insert a bit width operator on the processing path. See [Propagation of Link Properties](#) Tutorial topic.

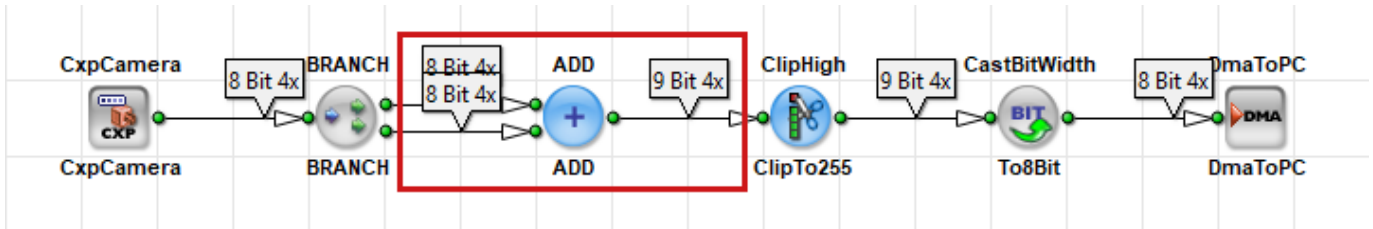
9.2 Bit Widths for Arithmetic Operations

Operators of the Arithmetic library derive the output **Bit Width** automatically. The **Supported Link Format** table in the [Operator Reference](#) marks these outputs as **auto**. You usually can't edit these output bit width link properties directly on the link. If you wish to change the bit width manually after the arithmetic operation, use additional operators such as `CastBitWidth` in combination with `ClipHigh` and/or `ClipLow`. The following arithmetic operators also change the bit width:

9.2.1 Addition and Subtraction

The **ADD** operator sums the pixel values on all input links. The **SUB** operator subtracts the pixel values on the second input from the first input.

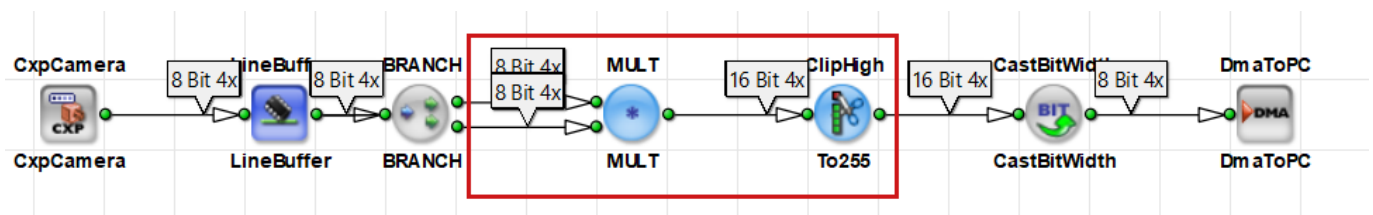
VisualApplets sets the output bit width from the input bit width. The result is one bit wider than that maximum. If you mix unsigned and signed inputs on **ADD**, the output may need one additional bit.



The output bit width must not exceed 64 bits. After **ADD** or **SUB**, check the propagated link width on downstream connections.

9.2.2 Multiplication

The **MULT** operator multiplies the pixel values at its two inputs. The maximum allowed input bit width at one link is 32 bit. The output bit width equals the sum of the two input bit widths. An 8-bit value multiplied by an 8-bit value produces a 16-bit result:



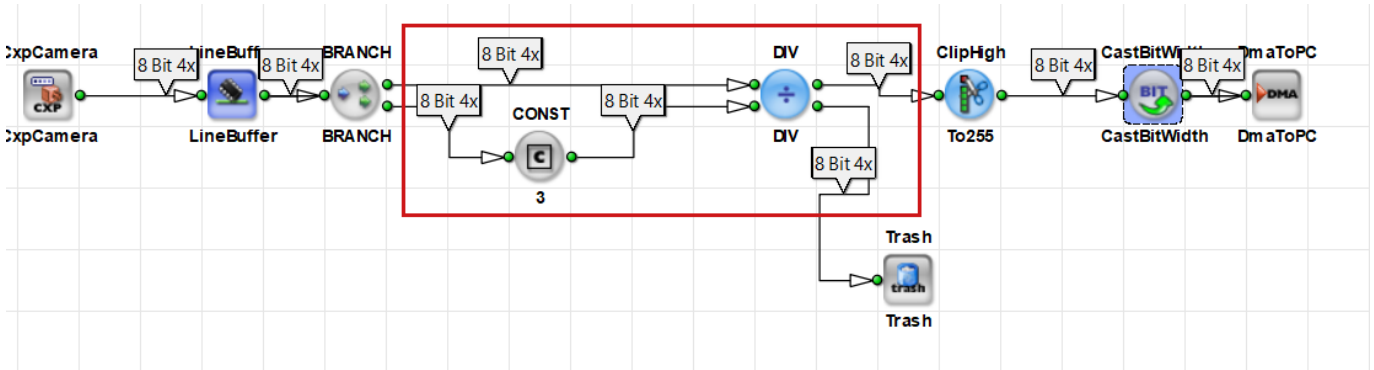
Info

MULT uses many FPGA resources. For scaling by a constant, Basler recommends **SCALE** or **ShiftLeft** instead.

9.2.3 Division

The **DIV** operator divides the value at input link **I1** by the value at input link **I2**. Output link **O** carries the integer quotient. Output link **R** carries the remainder. A division by zero is undefined at both outputs.

The bit width at input link **I2** must be less than or equal to the bit width at input link **I1**. For unsigned inputs, the output bit width at output link **O** equals the bit width at input link **I1**. For signed inputs, the output bit width at output link **O** equals the bit width at input bit width **I1** plus one.



DIV performs integer division. It doesn't preserve fractional digits on its own. For fixed-point division, use the **ShiftLeft** operator before you divide.

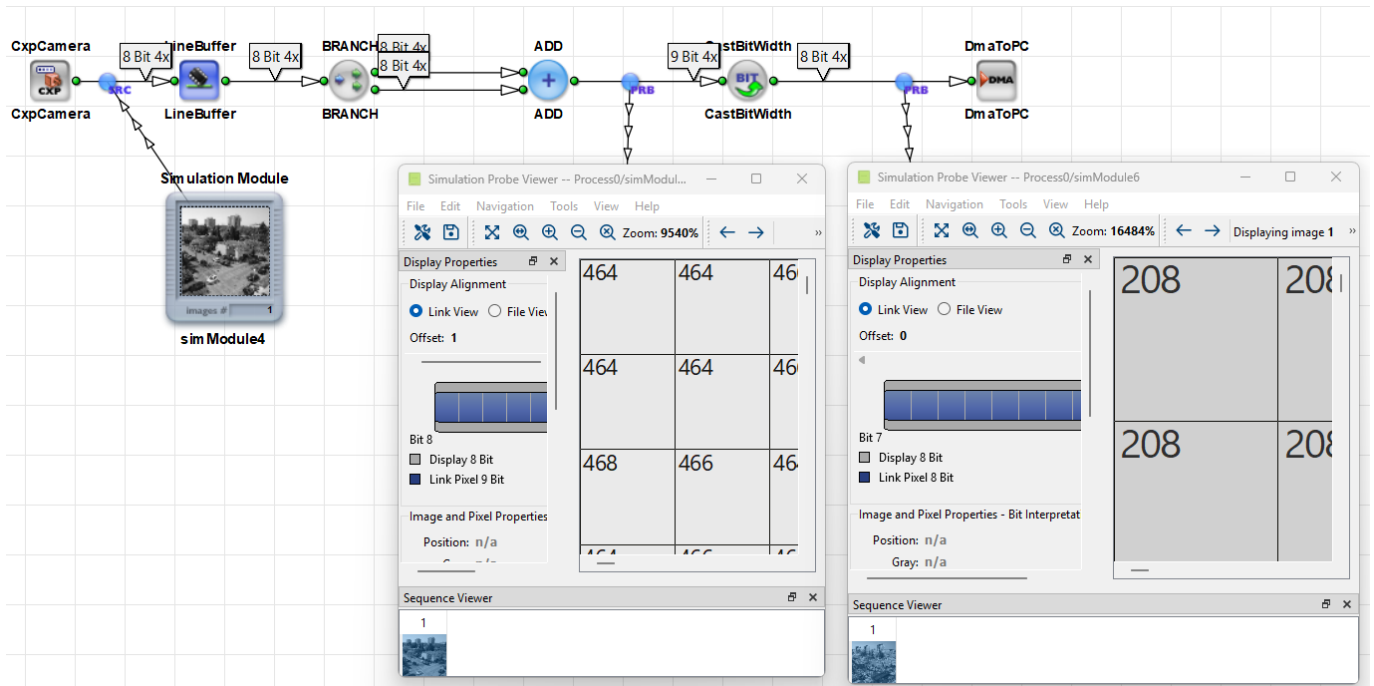
For detailed information about fractional values see the [Fixed-Point Arithmetic](#) section of this topic.

Info

DIV uses many FPGA resources. For division by a power of two, use **ShiftRight** instead.

9.3 Handling Arithmetic Overflow with Clip and Cast Operators

Arithmetic overflow occurs when a result exceeds the range that the current bit width can represent. On an 8-bit unsigned link, the range is 0 to 255. The sum $200 + 100$ equals 300. Without enough output bits, the stored value is wrong. The overflow is visible as incorrect pixel values:



Arithmetic overflow should not be confused with the overflow condition handled by the **overflow** operator in the **Synchronization** library: The **overflow** operator is used for flow control with non-stoppable sources. For further information about the **overflow** operator, see [Stoppable and Non-Stoppable Data Streams](#) Tutorial topic.

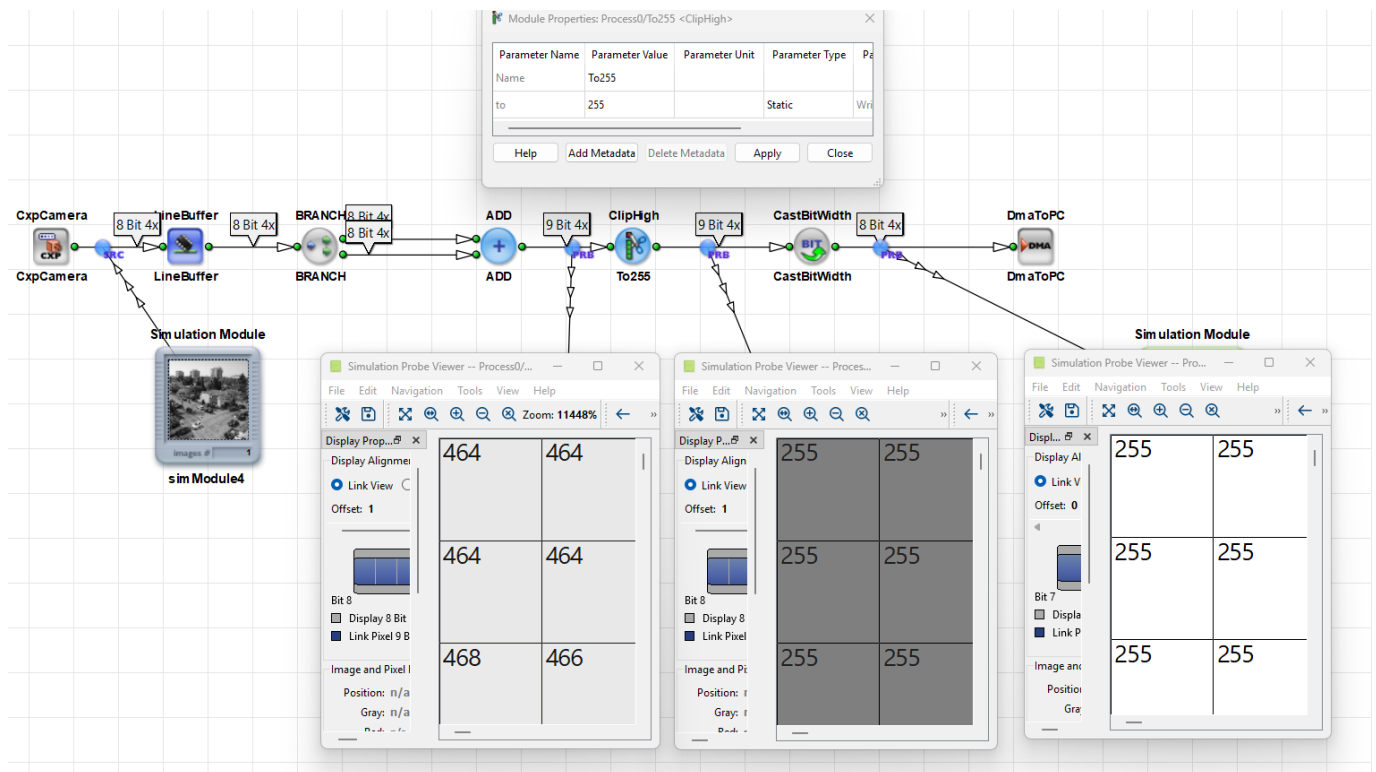
You can fix an arithmetic overflow with the following operators:

9.3.1 ClipHigh and ClipLow

The `ClipHigh` operator limits values to a configurable maximum. Any input value that exceeds the maximum is replaced by the maximum value. The `ClipLow` operator works similarly for a configurable minimum.

Use `ClipHigh` and `ClipLow` to keep values within a target range **before** reducing the bit width. Configure the clip limits using the module parameters. The valid range depends on the `Bit Width` input link property and on whether the input link uses signed or unsigned arithmetic.

The screenshot shows that `ClipHigh` sets all pixel values greater than 255 to 255:



9.3.2 CastBitWidth

The `CastBitWidth` operator changes the link bit width. You set the link property `Bit Width` on the output link of the operator.

When the output bit width is greater than the input bit width, the operator keeps the value and adds bits at the most significant positions. For signed values, `CastBitWidth` performs sign extension.

When the output bit width is less than the input bit width, `CastBitWidth` discards the most significant bits. As a result, values that cannot be represented within the output bit width are modified. Because this operation removes bits permanently, information required for later processing may be lost.

When reducing the link bit width, Basler recommends the following steps in this order:

1. Insert `ClipHigh` and `ClipLow` to limit the bit width values to the target range.
2. Insert `CastBitWidth` to reduce the bit width.

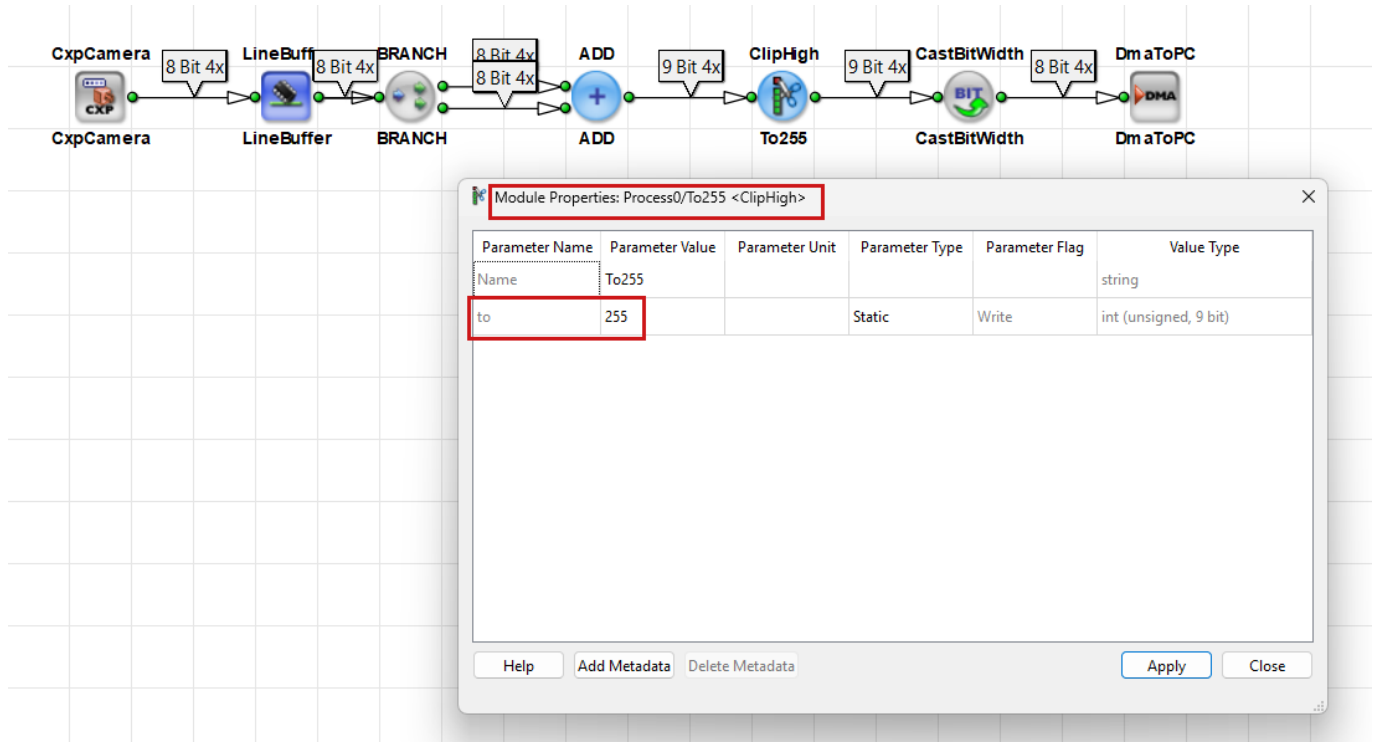
See also [Parameterization](#) Tutorial topic.

9.3.3 Example: Add Pixels and Narrow the Result

This example adds two 8-bit unsigned pixel streams and returns an 8-bit result for downstream processing:

1. Connect two 8-bit image paths to an `ADD` operator.
2. Check the output link. `ADD` produces a 9-bit result.
3. Insert `ClipHigh` after `ADD`. Set the maximum to 255.
4. Insert `CastBitWidth` after `ClipHigh`. Set the output `Bit Width` to 8.
5. Run **Design Rules Check Level 1** and confirm that all links are valid.

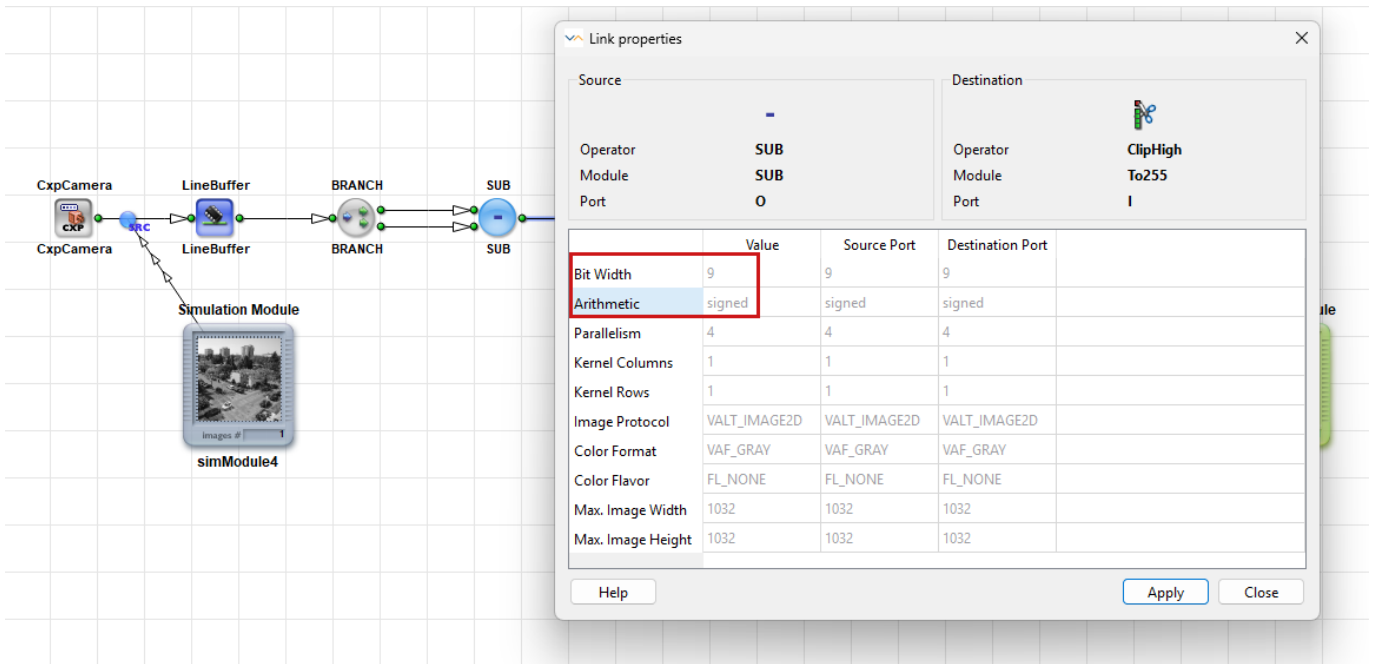
Without `ClipHigh`, `CastBitWidth` would only discard the ninth bit and therefore large sums produce wrong pixel values.



9.4 Signed and Unsigned Values

The link property `Arithmetic` defines whether the generated FPGA pipeline treats values as `Signed` or `Unsigned`. For `Signed`, the hardware uses two's complement representation. You set this property in VisualApplets during design time.

The sign bit is part of the configured bit width.

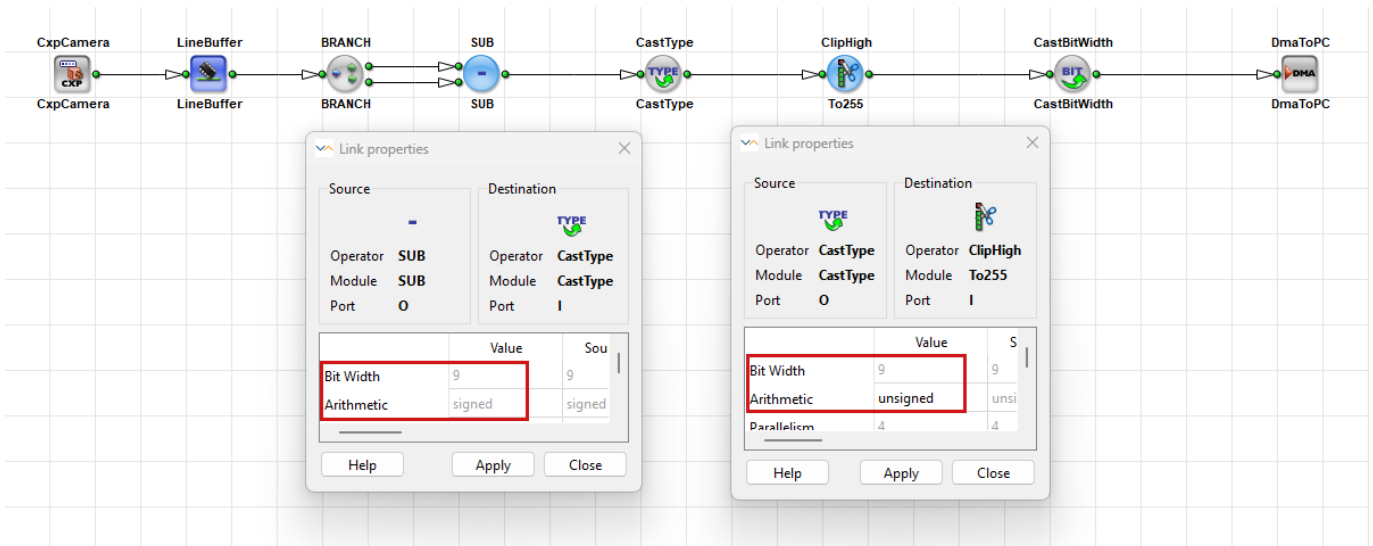


9.4.1 Changing the Arithmetic Link Property with the CastType Operator

The `CastType` operator changes the `Arithmetic` link property on the output link. The bit pattern stays the same. The generated FPGA logic doesn't change the stored bit pattern, so the operator uses no FPGA resources.

The same bit pattern can represent different decimal values depending on the arithmetic setting. The value `1010` on a 4-bit unsigned link equals 10. The same pattern on a 4-bit signed link equals -6.

Use the `CastType` operator when downstream operators require a different arithmetic type and the value range still fits.



9.4.2 Checking Whether a Value Fits to a Bit Range

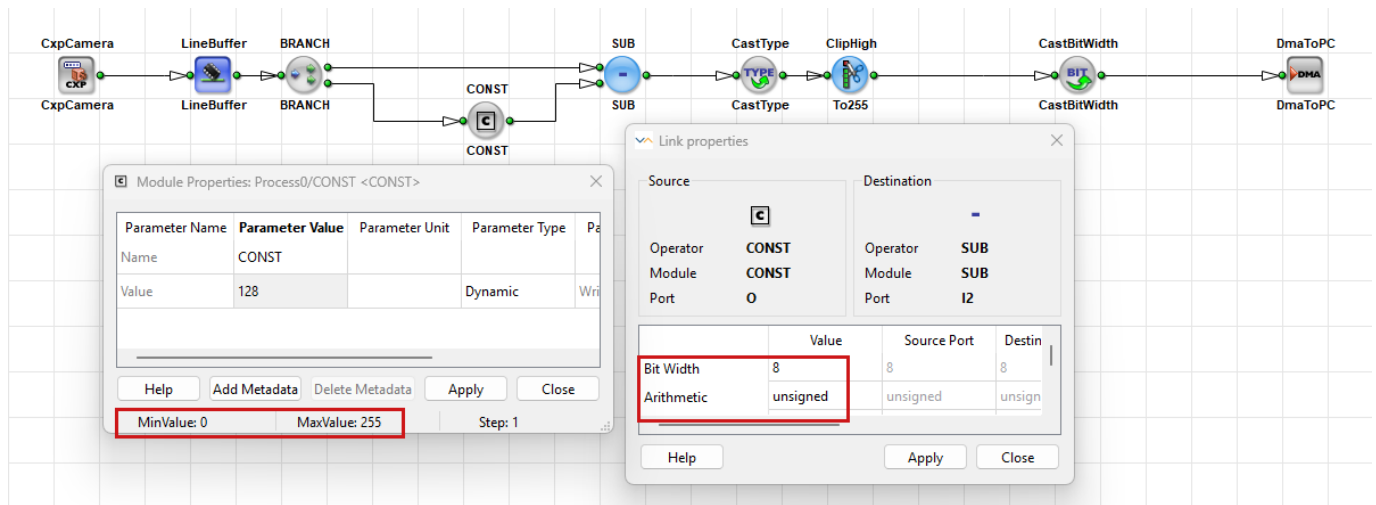
In VisualApplets, parameter values (for example, module parameters of the `CONST` operator) are entered and displayed as decimal numbers. However, the links connected to an operator can only store values within a specific bit range. Therefore, make sure that the parameter value fits into the bit width of the connected link.

Unsigned Links

An n-bit unsigned link can represent only positive values and zero. The valid range is:

$$0 \text{ to } 2^n - 1$$

For example, an 8-bit unsigned link can store values from 0 to 255:



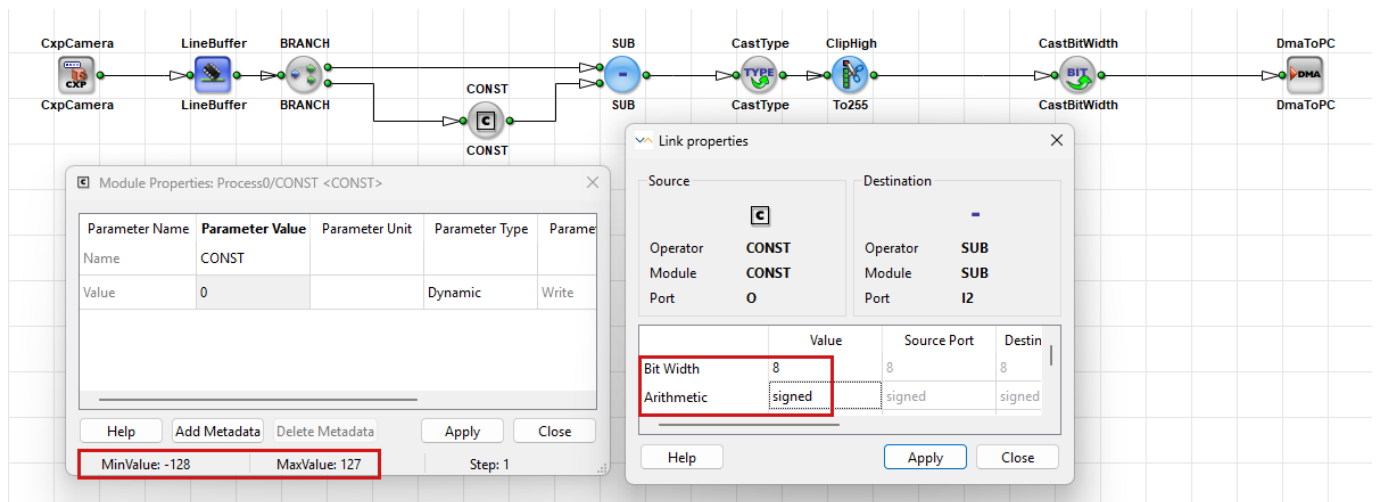
On links with `ImageProtocol = VALT_SIGNAL`, the arithmetic is always Unsigned.

Signed Links

An n-bit signed link can represent both positive and negative values. The valid range is:

$$-2^{(n-1)} \text{ to } 2^{(n-1)} - 1$$

For example, an 8-bit signed link can store values from -128 to 127.



The most significant bit is used as the sign bit and indicates whether the value is positive or negative.

9.5 Fixed-Point Arithmetic

VisualApplets is used to create image-processing designs that are implemented on an FPGA. Unlike software running on a PC, FPGA hardware usually doesn't work with floating-point numbers (such as 0.5 or 1.25). Instead, calculations are typically performed using integers.

To represent fractional values while still using integer arithmetic, VisualApplets uses fixed-point representation. This allows values such as scaling factors, gains, or subpixel coordinates to be stored and processed efficiently in hardware.

This section explains:

- Why fixed-point representation is needed.
- How fractional values are stored in integer bit patterns.
- How Q format defines the position of the decimal point.
- How bit width affects the available value range and precision.
- How operator selection influences the accuracy of calculations.

Every value transmitted on a link is stored as a sequence of bits. Hardware interprets these bits as an integer value.

For example, the bit pattern below represents the integer value 5:

```
00000101 = 5
```

However, image-processing applications often require fractional values such as: 0.5, 1.25, or 12.75.

Because the FPGA does not store floating-point numbers, these values are represented using a fixed-point format. In fixed-point representation, the bit pattern is still stored as an integer, but you define a fixed position for the decimal point.

The stored integer value is still 5, but it is interpreted as:

```
5 / 2^2 = 1.25
```

because two bits are reserved for the fractional part.

You can think of this as a **virtual decimal point** that is fixed at a predefined position. In binary arithmetic this split is a binary point, between the bit weights 2^0 and 2^{-1} . This tutorial keeps the familiar term **virtual decimal point**. The FPGA performs all calculations using integers, while you interpret the result according to the selected fixed-point format.

9.5.1 Q Format and Bit Allocation

Fixed-point numbers represent fractional values by assigning some bits to the integer part and some bits to the fractional part. The position of the virtual decimal point is described using **Q format**.

A Q format is written as:

```
Qm.n
```

where:

- **m** is the number of integer bits
- **n** is the number of fractional bits

For signed values, the sign bit is included in the integer part.

For example:

```
Q4.4
```

means:

- 4 bits for the integer part
- 4 bits for the fractional part
- 8 bits in total

The number of fractional bits determines the precision of the represented value.

The smallest representable step size is:

$$1 / 2^n$$

where n is the number of fractional bits.

Examples:

Format	Fractional Bits	Smallest Step Size
Q6.2	2	0.25
Q4.4	4	0.0625
Q2.6	6	0.015625

More fractional bits allow smaller differences between adjacent values and therefore higher precision.

The total bit width is fixed. As a result, bits used for the fractional part are no longer available for the integer part.

Therefore:

- More fractional bits provide higher precision but reduce the available integer range.
- Fewer fractional bits provide a larger integer range but lower precision.

When selecting a Q format, balance the required value range against the required precision.

In a fixed-point number, each bit represents a power of two.

Bits to the left of the virtual decimal point represent whole numbers:

$$\dots, 2^3, 2^2, 2^1, 2^0$$

Bits to the right represent fractions:

$$2^{-1}, 2^{-2}, 2^{-3}, \dots$$

For example, in a Q4.4 format:

Bit position:	7	6	5	4	3	2	1	0
Weight:	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}

The location of the 2^0 bit defines the position of the virtual decimal point. This position is not stored in the hardware; it is defined by the selected Q format and must be interpreted consistently throughout the design.

Example: Q4.4

In a Q4.4 format, 4 bits are used for the integer part and 4 bits for the fractional part.

The bit pattern

$$0\ 0\ 0\ 0\ .\ 1\ 0\ 0\ 0$$

contains a 1 in the first fractional position. This position represents the value 2^{-1} , which equals 0.5.

Therefore, the bit pattern represents:

$$0 * 2^0 + 1 * 2^{-1} + 0 * 2^{-2} + 0 * 2^{-3} + 0 * 2^{-4} = 0.5$$

Although the represented value is 0.5, the FPGA stores only the corresponding integer value:

Stored integer value:	8
Represented real value:	0.5

This works because Q4.4 uses 4 fractional bits. The stored integer value is interpreted as:

```
Real value = Stored integer / 2^4
           = 8 / 16
           = 0.5
```

Let `F` be the number of fractional bits.

To convert a real value into its fixed-point representation:

```
FixedPoint = round(RealValue * 2^F)
```

Example:

```
FixedPoint = round(0.5 * 2^4)
           = round(8)
           = 8
```

Resulting bit pattern:

```
0000.1000
```

To convert a fixed-point value back into a real value:

```
RealValue = FixedPoint / 2^F
```

Example:

```
RealValue = 8 / 2^4
           = 8 / 16
           = 0.5
```

9.5.2 Operators and Design Rules

Operators process integer bit patterns only. In fixed-point arithmetic, the position of the virtual decimal point is determined by the Q format and must be tracked throughout the design.

The following operators are commonly used to adjust scaling and maintain the intended Q format:

- The `ShiftLeft` and `ShiftRight` operators shift the input bit pattern left or right by a configurable number of bits. A left shift multiplies the value by 2^{Shift} ; a right shift divides it by 2^{Shift} . These operators are often used after arithmetic operations to restore the desired Q format.
- `SCALE` or `ShiftLeft`: Use these operators instead of `MULT` when multiplying a value by a constant power of two. They perform the same scaling operation, but typically require fewer FPGA resources.
- The `MULT` operator multiplies two input values. The output bit width equals the sum of the input bit widths. In fixed-point arithmetic, the number of fractional bits in the result is also the sum of the fractional bits of both operands.

Example

Multiplying a Q4.4 value by another Q4.4 value produces a Q8.8 result:

```
Q4.4 * Q4.4 -> Q8.8
```

The result provides higher precision, but it may require a subsequent `ShiftRight` operation or a bit-width reduction if the design expects a Q4.4 value.

Some operators assume a specific fixed-point format. For example, trigonometric operators such as `SIN`, `COS`, `TAN`, and their inverse functions expect a predefined number of fractional bits for their inputs and outputs.

Before connecting links to such operators, verify the required format in the [Arithmetics Library](#) in the Operator Reference.

Design Guideline

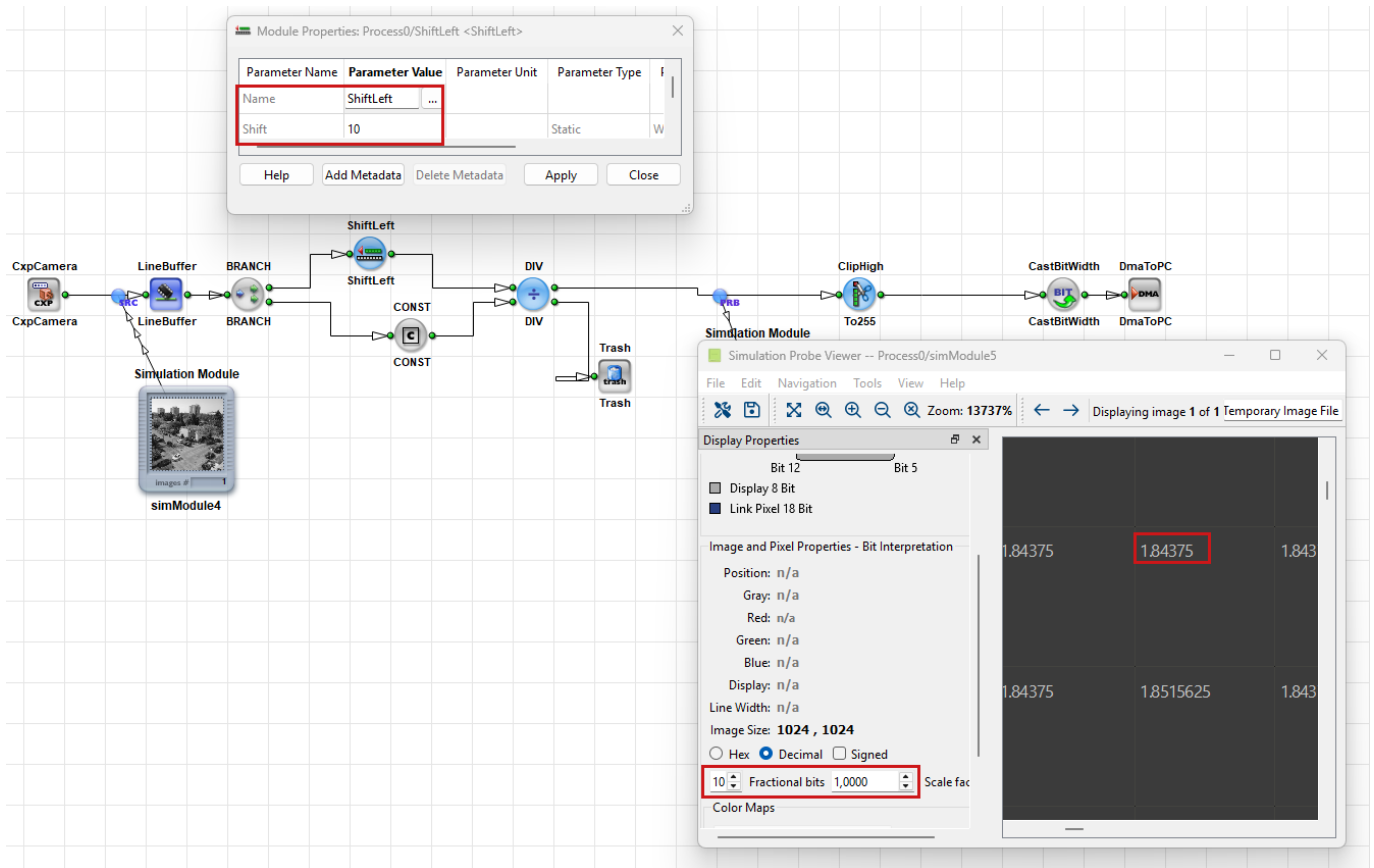
Document the Q format at each stage. After multiplication or division, use `ShiftRight` or bit-width adjustment if needed to restore the intended format.

9.5.3 Displaying Fixed-Point Values in Simulation

You can use the simulation function in VisualApplets to verify that fixed-point values are interpreted, scaled, and propagated through the design as intended before building the applet. In the **Simulation Probe Viewer**, link values remain integers, but you can display them as fixed-point by setting **fractional bits**, choosing signed/unsigned decimal, and optionally applying a **scaling factor**. This affects display only, not hardware. To check the fixed-point values via simulation in VisualApplets:

1. Add a **Simulation Source** to start of the processing pipeline.
2. Add a **Simulation Probe** to the operator's output link.
3. Start simulation via **Analysis** → **Start Simulation**.
4. Open the **Simulation Probe Viewer** for the probe you added in step 2.
5. In the **Simulation Probe Viewer**, select signed or unsigned decimal display.
6. Set the number of **fractional bits** to match your Q-format.
7. Optionally set a scaling factor for unit conversion, e.g. radians to degrees.

As a result, you see how the **Simulation Probes** take the fractional bits into account:



For further information about simulation in VisualApplets, see [Simulation](#) in the User Manual.

9.6 Extracting Bit Fields with SelectBitField

The `SelectBitField` operator extracts a contiguous range of bits from the input value and copies it to the output. Use this operator when you need only a specific portion of a wider data word.

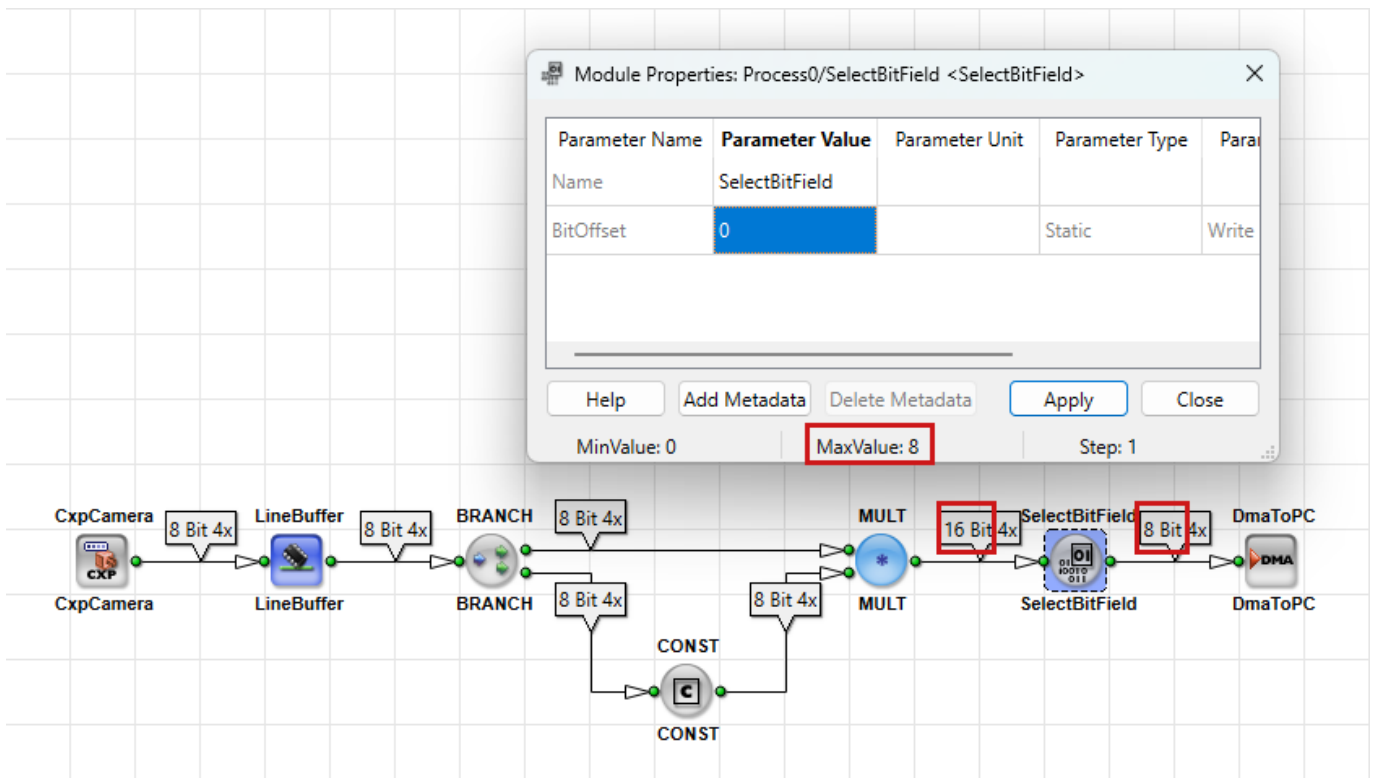
Configure the operator as follows:

- Set the output `Bit Width` on the output link. This determines how many bits are extracted.
- Set the `BitOffset` parameter. This specifies the position of the least significant bit (LSB) of the selected field to be copied to the output. The selected bit becomes bit 0 (the least significant bit) of the output value.

For gray images, `BitOffset` must be in the range:

$0 \dots \text{Input Bit Width} - \text{Output Bit Width}$

If the input uses signed arithmetic, the extracted bits are always output as an unsigned value.



9.7 Widening Bit Width for Downstream Software

Many cameras produce pixel data with bit widths such as 10 or 12 bits per pixel. However, image-processing software and image file formats often work with standard data widths such as 8, 16, or 32 bits.

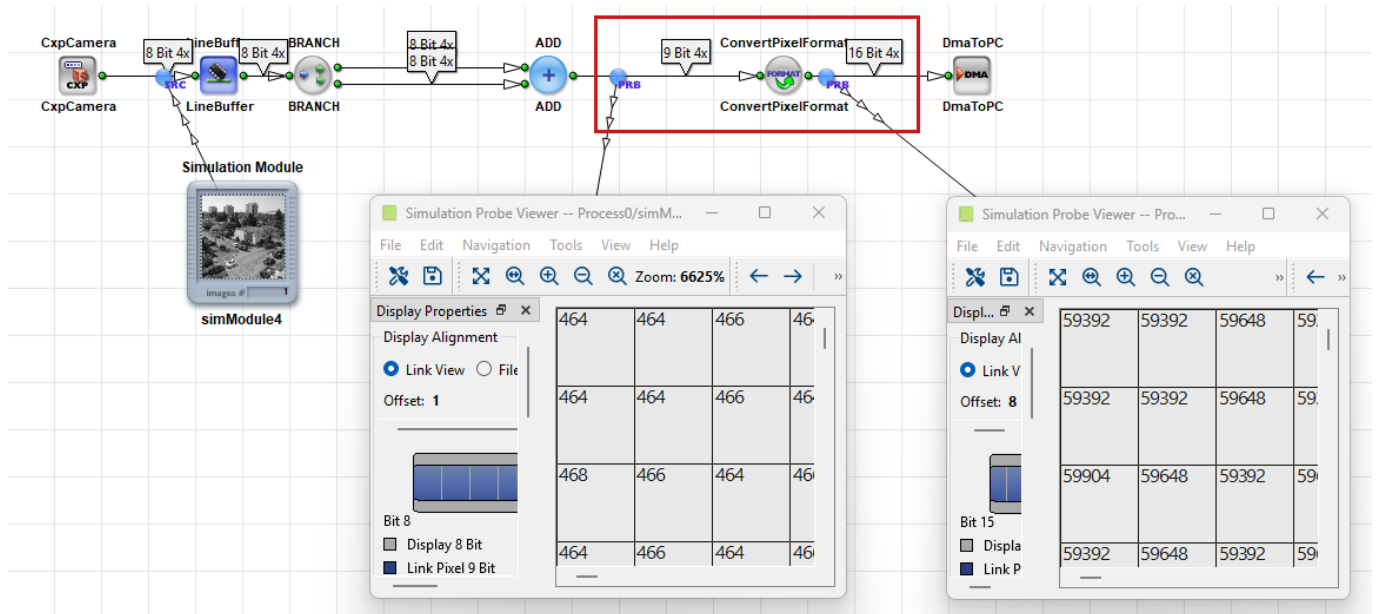
Therefore, it is common to convert camera data to 16-bit pixels before transferring it to the host PC. You can do this in the following ways:

9.7.1 Using ConvertPixelFormat

To widen the bit width, you can insert a `ConvertPixelFormat` operator between the image source or buffer and `DmaToPC`.

After inserting the `ConvertPixelFormat` operator, open the output link properties and set `Bit Width` to 16.

For example, when converting 9-bit pixel data to 16 bits, the operator shifts each pixel value 7 bits to the left and inserts zeros into the newly created least significant bit positions. This preserves the relative brightness range while adapting the data to a 16-bit format.



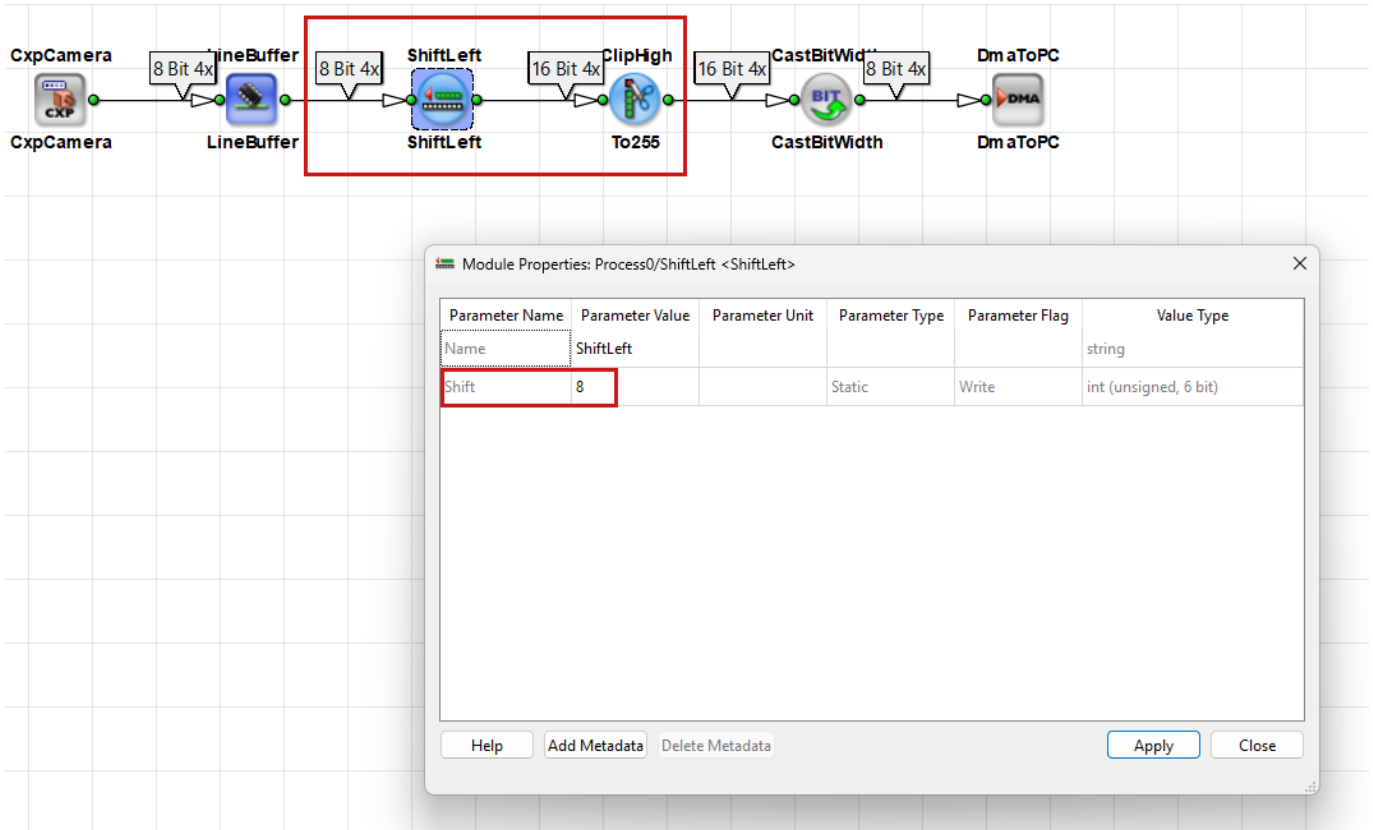
9.7.2 Using ShiftLeft

As an alternative, you can also insert a `ShiftLeft` operator and set the `Shift` parameter to 7.

A left shift by 7 bits produces the same scaling effect as converting a 9-bit value to a 16-bit value:

```
9-bit value << 7
```

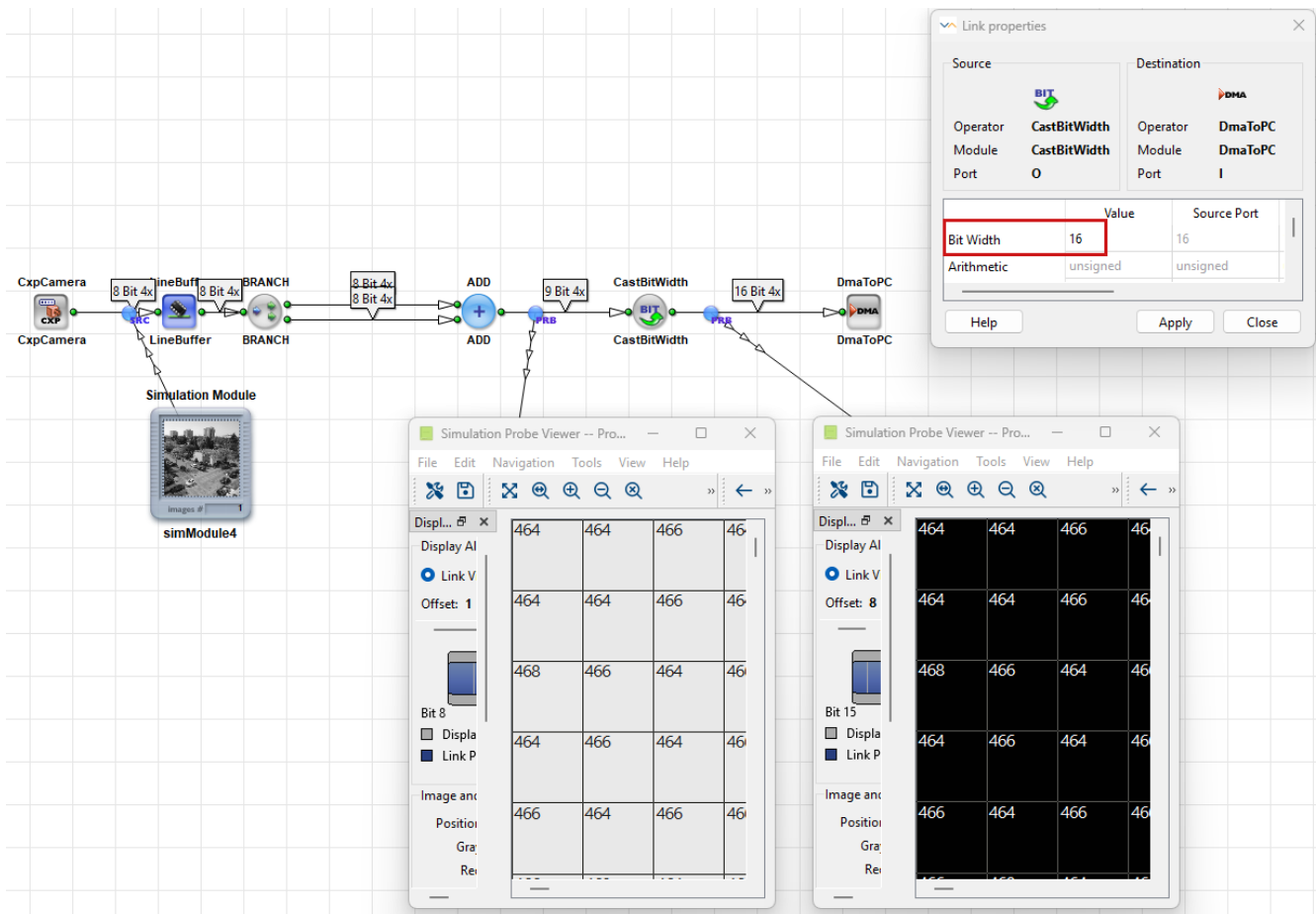
The output link then carries 16-bit pixel values.



9.7.3 Using CastBitWidth

If you want to increase the bit width without changing the numerical value, insert a **CastBitWidth** operator. Choose this option, if downstream software requires a wider bit width, but no scaling of the pixel values is desired.

To change the bit width with a **CastBitWidth** operator, change the bit width on the output link of the **CastBitWidth** operator:



Unlike `ConvertPixelFormat` or `ShiftLeft`, `CastBitWidth` doesn't shift the pixel data. Instead, it adds additional high-order bits while preserving the original value.

9.8 Verifying Your Design with DRC

After you change bit widths or insert arithmetic operators, run **Analysis → Design Rules Check Level 1**. Level 1 reports bit width mismatches and parameter conflicts in the **DRC log**. Affected modules and links are highlighted in the design pane.

Run **Design Rules Check Level 2** before you build. Level 2 estimates FPGA resource usage. Wide intermediate results and operators such as `MULT` and `DIV` increase resource consumption.

Read the DRC log together with highlighted modules and links. For the full workflow, see [From Empty Design to Running Applet](#) Tutorial topic.

Key Terms

Basler recommends these practices when you work with bit widths:

- Check the propagated `Bit Width` on downstream links after every arithmetic operator.
- Clip values before you cast to a narrower width.
- Plan bit width changes for host software early, for example 16 bits per pixel before the DMA.
- Use `ShiftLeft` and `ShiftRight` for power-of-two scaling instead of `MULT` and `DIV` when possible.
- Consult the [Operator Reference](#) when a parameter value doesn't stick.

9.9 Related Topics

- [Parameterization](#) Tutorial topic — `BitWidth` and arithmetic link properties
- [Rules of Links](#) in the User Manual — formal link rules
- [From Empty Design to Running Applet](#) Tutorial topic — DRC, simulation, and build
- [VisualApplets Fundamentals](#) Tutorial topic — pipeline and FPGA resources
- [Operator Reference](#) — supported link formats
- [CastBitWidth](#) in the Operator Reference
- [ClipHigh](#) in the Operator Reference
- [ClipLow](#) in the Operator Reference
- [ConvertPixelFormat](#) in the Operator Reference
- [SelectBitField](#) in the Operator Reference
- [ShiftLeft](#) in the Operator Reference
- [ShiftRight](#) in the Operator Reference

10. Memory Elements

This topic explains how **memory operators** in a VisualApplets design map to physical memory on the target hardware, and when you use which type. After you have read this topic, you understand the difference between **FPGA-internal memory** (for example block RAM) and **frame grabber DRAM**, how FIFO-style buffers differ from random-access frame buffers, how **shared DRAM bandwidth** limits designs with several RAM operators, and how you recognize memory-related problems in simulation and on hardware.

Key Terms

Memory operators in a VisualApplets design use one of two physical memory locations on the **target hardware**: **inside the FPGA** on the frame grabber (small, fast, local) and **on the frame grabber outside the FPGA** (DRAM — more capacity for line and frame buffers).

FillLevel tells you how full a buffer is (in percent) at runtime on the hardware, for example in microDisplay X. A buffer that stays near 100% often points to a bottleneck or deadlock. As a starting point, place a buffer after the camera.

For platform RAM sizes, data widths, and bandwidth values, see [Device Resources](#).

10.1 FPGA Memory vs. Frame Grabber DRAM

VisualApplets memory operators fall into two groups by **where** they store data on the frame grabber. A second distinction is **how** they access that data: FIFO-style buffers versus random-access or multi-ROI frame buffers. The [Overview of the Memory Library](#) in the Operator Reference shows where each memory operator stores its data. The **RAM** resource in VisualApplets is a device-resource token that maps to board DRAM. It is not memory inside the VisualApplets application.

Property	FPGA-internal memory	Frame grabber DRAM
Typical operators	<code>ImageFifo</code> ; for small random-access cases also <code>FrameMemoryRandomRd</code> / <code>LineMemoryRandomRd</code>	<code>ImageBuffer</code> , <code>LineBuffer</code> , <code>FrameBufferRandomRead</code> , <code>FrameBufferMultiRoiDyn</code>
Location	Inside the FPGA (BRAM, LUT RAM, or URAM on imaFlex CXP-12 Quad/Penta)	On the board, outside the FPGA
Capacity	Limited; full-resolution frames often exceed available FPGA memory	Higher; suitable for many lines or full frames
Latency	Low; no access to an external DRAM bus	Line-oriented FIFOs add at least one line
VisualApplets RAM resource	No — does not reserve a RAM device resource	Yes — one RAM resource per instance, mapped to board DRAM
Bandwidth	Local; operators don't compete for the frame grabber RAM interface	Shared among RAM operators on the same physical interface Bandwidth

Use the table in [Choosing a Memory Operator](#) to pick an operator, then read the matching operator section for parameters and platform details.

10.2 Choosing a Memory Operator

Use Case	Prefer
Small buffer, low latency, no DRAM bandwidth cost	<code>ImageFifo</code> (FPGA-internal)
Image FIFO on mE5, peak absorption, ROI	<code>ImageBuffer</code> (DRAM)
Line FIFO on imaFlex CXP-12 Quad/Penta	<code>LineBuffer</code> (DRAM)
Buffer before <code>SYNC</code> with small delay	<code>ImageFifo</code> (or larger DRAM buffer if delay is large)
Full-frame store with random pixel read	<code>FrameBufferRandomRead</code> (DRAM)
Multiple or dynamic ROIs from one buffer	<code>FrameBufferMultiRoi</code> / <code>FrameBufferMultiRoiDyn</code> (DRAM)

Design Guideline

Basler recommends that you use at least one DRAM operator in a design to compensate for data bursts from the camera. That DRAM buffer is the default camera decoupling path (`LineBuffer` or `ImageBuffer`). `ImageFifo` is not a substitute for that DRAM buffer when you need to absorb camera bursts.

10.3 ImageFifo

The `ImageFifo` operator queues image data as a **first-in, first-out** buffer in FPGA-internal memory. You size the buffer with the `EntitiesToStore` and `EntityType` parameters. `EntityType` has the values `PIXEL`, `LINE`, or `FRAME`. Data is forwarded immediately when the output is free. The buffer fills when downstream modules block.

Set the `InfiniteSource` parameter of the `ImageFifo` operator to `ENABLED` when the input comes from a non-stoppable source such as a camera.

Info

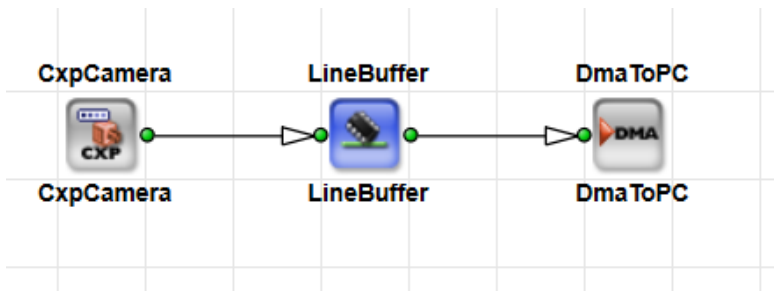
The `ImageFifo` operator buffers only a relatively small number of pixels. Therefore, there is a significant risk of a FIFO overflow, which isn't always handled internally in a robust way by the operator. For non-stoppable streams, Basler recommends combining `ImageFifo` with a preceding `Overflow` operator, or using the `ImageFlowControl` operator instead of the `ImageFifo` operator. The `ImageFlowControl` operator combines FIFO buffering with integrated overflow handling.

With the parameter `ImplementationType` you can select `AUTO` (default), `BRAM`, `LUTRAM`, or — on imaFlex CXP-12 Quad/Penta — `URAM`. Basler recommends to always use `AUTO` unless a specific FPGA resource type runs short.

For details about the operator, see [ImageFifo](#) in the Operator Reference.

10.4 ImageBuffer and LineBuffer

The `ImageBuffer` (microEnable 5) and the `LineBuffer` (imaFlex) operators work like a **line-oriented FIFO**: Completed lines in the selected ROI are forwarded as soon as downstream modules can accept them. Latency is at least **one line**. Latency grows if the buffer fills because the output is blocked. See also [Latency](#) Tutorial topic.



10.4.1 Platform Note: ImageBuffer vs. LineBuffer

Operator	Platforms
ImageBuffer	microEnable 5 (mE5)
LineBuffer	imaFlex CXP-12 Quad / Penta

imaFlex CXP-12 Quad and Penta **don't support** the `ImageBuffer` operator. When you convert a design from mE5 to imaFlex, VisualApplets **replaces** each `ImageBuffer` operator with a `LineBuffer` operator. The `LineBuffer` operator provides the same core buffering and ROI features for the imaFlex platforms.

10.4.2 Defining Regions of Interest (ROI) with ImageBuffer and LineBuffer

With the `ImageBuffer` and `LineBuffer` operators, you can also define regions of interest (ROIs). You do that by setting the `XOffset`, `XLength`, `YOffset`, and `YLength` parameters. Edge cases are documented in the Operator Reference: For example, if the input is smaller than the requested ROI, or if your design is in `Line1D` mode, or if you have undefined content for extra pixels.

For details about the operator, see [LineBuffer](#) in the Operator Reference and [ImageBuffer](#) in the Operator Reference.

10.4.3 Monitoring Buffer Usage and Overflow

The `ImageBuffer` and `LineBuffer` operators provide the following status parameters to monitor buffer usage and overflow:

Parameter	Meaning
FillLevel	Defines how full the buffer RAM is, in percent (on <code>ImageBuffer</code> in 25% steps).
LineCount	Indicates the current number of lines stored. Only available for the <code>LineBuffer</code> operator.
Overflow	Indicates that the buffer can't accept more data without exceeding capacity. This status parameter is not the same as the <code>Overflow</code> operator in the Synchronization library, and not the same as arithmetic overflow. See Stoppable and Non-Stoppable Data Streams Tutorial topic and Bit Arithmetic Tutorial topic.

If an overflow occurs, incoming data may be discarded and the effective image height might be reduced. Incomplete images can remain in the memory until the operator recovers. **Overflow** applies when the source is non-stoppable and `InfiniteSource` is `ENABLED`. Set `InfiniteSource` to `ENABLED` when the buffer sits directly after a camera or another infinite source. See also [Stoppable and Non-Stoppable Data Streams](#) Tutorial topic. For how to use these parameters when debugging, see [Recognizing Memory-Related Errors](#).

Process0*

Module Properties: Process0/LineBuffer <LineBuffer>

Parameter Name	Parameter Value	Parameter Unit	Parameter Type
Name	LineBuffer		
RamDataWidth	384		Static
XOffset	0	pixel	Dynamic
XLength	1024	pixel	Dynamic
YOffset	0	pixel	Dynamic
YLength	1024	pixel	Dynamic
ParallelismConverter	No		Static
InfiniteSource	ENABLED		Static
WritePriority	DISABLED		Static
FillLevel	0	%	Dynamic
MaxLineCount	145888	line	Dynamic
LineCount	0	line	Dynamic

enum (ENABLED=0, DISABLED=1)

DRC Log

Netlist Generation

Creation date: Mittwoch, 3. Juni 2026 11:12:12

Netlist generation started...

Info: Building design for FPGA clock frequency 312.500 MHz

FPGA resource estimation:

Resource	Count	Fill Level
LookupTables	54177	~ 33%
Flip Flops	53183	~ 16%
Block RAM	125	~ 17%
Ultra RAM	0	~ 0%
Embedded ALU	47	~ 3%

11:12:16 -- Netlist successfully generated in 00:00:04.

Design Rules Check Level 2

Program version: VisualApplets 3.7.100.76452M-9d0b71df5
Design name:
Creation date: Mittwoch, 3. Juni 2026 11:12:16

0 Errors
0 Warnings

11:12:16 -- Design Rules Check Level 2 successfully performed.

On the `LineBuffer` operator, Basler recommends enabling the `WritePriority` parameter when `InfiniteSource` is `ENABLED`. If `WritePriority` is `ENABLED`, write bursts from the camera are preferred over reads while the temporary write rate exceeds available DRAM bandwidth. The **average** bandwidth must still stay within the share available to that operator.

The `LineBuffer` operator can also convert parallelism between input and output with the module parameter `ParallelismConverter`, so you don't always need a separate `PARALLELup` / `PARALLELdn`. See also [Parallelism](#) Tutorial topic.

10.5 Full Frames, Random Access, Multi-ROI with the FrameBuffer Operators

The `FrameBufferRandomRead`, `FrameBufferMultiRoi`, and `FrameBufferMultiRoiDyn` operators store **full images or large regions** and read them in an order that is **not** a strict line FIFO. Use them when you need random pixel access or multiple ROIs from one buffer.

10.5.1 FrameBufferRandomRead

`FrameBufferRandomRead` writes the input image into DRAM, then reads pixels by **column and row addresses** on dedicated address ports. Use it for geometric transforms, remapping, and other random-access patterns.

For small frames or lines, prefer FPGA-internal operators such as `FrameMemoryRandomRd` or `LineMemoryRandomRd` instead of DRAM operators.

For details about the operator, see [FrameBufferRandomRead](#) in the Operator Reference.

10.5.2 FrameBufferMultiRoi and FrameBufferMultiRoiDyn

The `FrameBufferMultiRoi` and `FrameBufferMultiRoiDyn` operators store image data and output **multiple regions of interest**. `FrameBufferMultiRoiDyn` allows **dynamic** ROI size and position from control links.

10.6 Sharing DRAM Bandwidth

Each DRAM operator needs one VisualApplets resource of type **RAM**. That resource is a slot in the VisualApplets resource model. It maps to physical DRAM on the frame grabber. On imaFlex and microEnable 5 marathon platforms, all operators that reserve a **RAM** resource share the physical DRAM. Each operator gets a **dedicated, non-overlapping memory region**, but **size and bandwidth** are divided among active RAM operators on the same physical interface.

Consequences:

- Every **read** and every **write** consumes bandwidth on the shared interface.
- With several DRAM instances or high throughput, the total traffic can exceed the platform limit.
- Excess demand leads to blocking, rising `FillLevel1`, overflow, or a lower sustainable frame rate.

Look up **RAM size**, **RAM data width**, and **RAM bandwidth total (shared)** for your platform in [Device Resources](#).

Allocate resources in the **Resources** dialog when needed; see [Allocation of Device Resources](#) in Device Resources.

10.6.1 Bandwidth Calculation Notes

The [Shared Memory](#) Application Note explains how to use RAM size and RAM data width efficiently.

Key points:

1. **Read and write share the interface.** For a single DRAM operator, divide the platform maximum bandwidth by **2** when you estimate the usable image data rate (half for writing, half for reading), then apply efficiency factors. Documented bandwidth values are **theoretical maxima**; real efficiency is lower.
2. **Use as much of the RAM data width as practical.** The link's effective RAM data width is:

bit width × parallelism × kernel elements

That product is the effective width of each DRAM access. Do not confuse it with the pixel-rate bandwidth from **parallelism × Base Design Clock**. Maximum DRAM throughput is reached when the product matches the platform **RAM data width** without exceeding it, for example 256 bit, 384 bit, or 640 bit, depending on the board. Matching the product to the maximum RAM data width uses the full write and read bandwidth of the DRAM; if the value is much lower, DRAM access can become a bottleneck. A design with 8 bit × parallelism 4 uses only 32 bit of a 256-bit interface and wastes bandwidth headroom.

1. **Multiple DRAM operators share bandwidth.** On shared-memory platforms, the bandwidth available to one operator scales roughly with **1 / (number of RAM operators)** on that interface. Details differ by platform as documented in Device Resources and the Application Note: Shared Memory (see [Related Topics](#)).
2. **Balance operators across independent RAM banks** when the platform provides more than one physical interface, for example as the imaFlex 2 Dual 100 does. Bandwidth is shared only among operators on the **same** physical bank.



Design Guideline

Plan RAM usage early. Every additional DRAM operator reduces the bandwidth available to the others on the same physical memory interface. Before you add another DRAM buffer, calculate the combined read/write traffic and compare it with the shared bandwidth available for your platform (see Device Resources). Prefer `ImageFifo` (FPGA memory) for small decoupling tasks so you don't spend DRAM bandwidth unnecessarily.

10.7 Recognizing Memory-Related Errors

VisualApplets **can't always validate** that a design has enough memory of the right type and size for every runtime scenario. You must choose operators and buffer sizes that match your image dimensions, rates, and synchronization needs.

Simulation may run successfully even when buffers are undersized for hardware. Failures often appear only on the frame grabber: for example, no image is output in the SDK although simulation looked fine.

Common causes are `SYNC` **deadlocks** and **insufficient or wrong memory**.

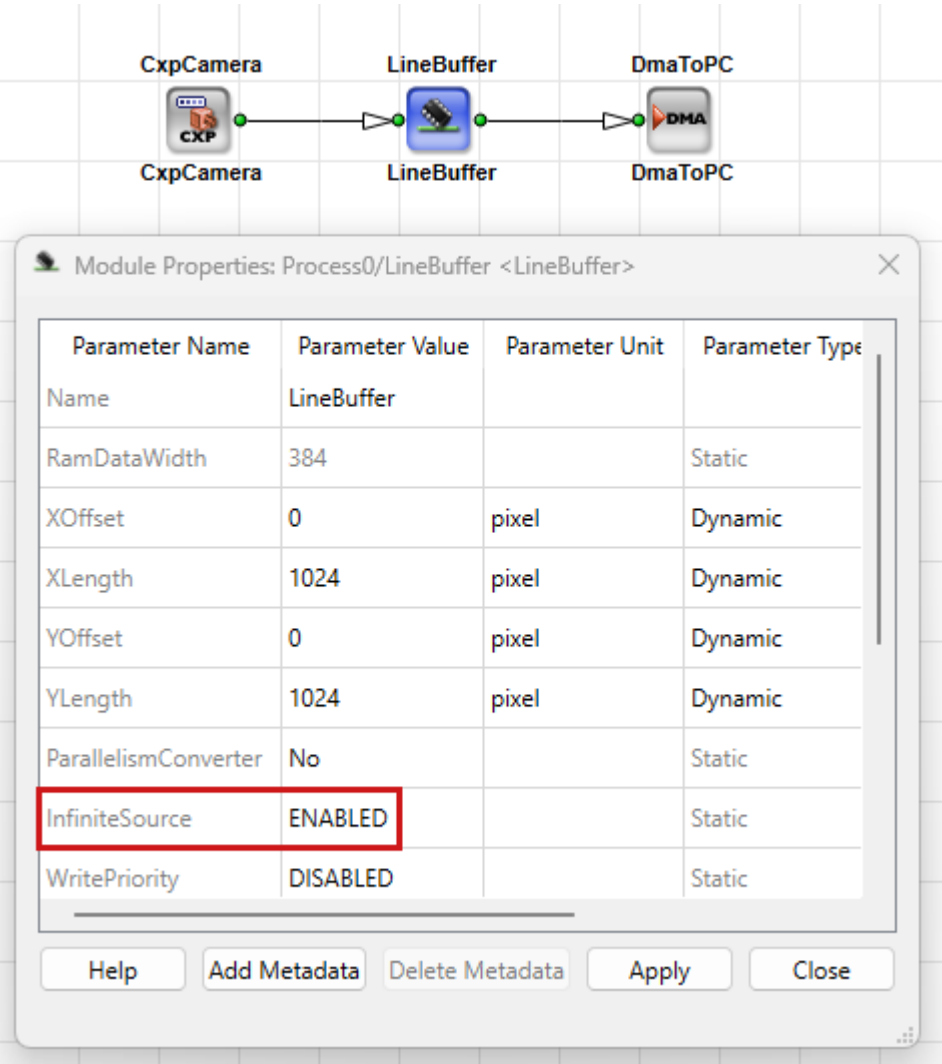
To avoid these issues, do these checks:

- Run **Design Rules Check Level 1 and Level 2** after wiring buffers and DMA paths.
- On hardware, watch `FillLevel` (and `Overflow` where available), for example in microDisplay X. Persistent 100% fill often indicates a deadlock or a bandwidth bottleneck.
- With several DRAM operators, open **Design** → **Device Resources** and verify RAM allocation and expected bandwidth share.
- For infinite sources, confirm that `InfiniteSource` is `ENABLED` on every buffer that accepts a non-stoppable stream.

10.8 Typical Pipeline Examples

10.8.1 Camera to DMA Through a DRAM Line Buffer

The smallest possible design has one camera operator, one DRAM buffer with `InfiniteSource` set to `ENABLED`, and one `DmaToPC` operator:



Use this buffer to absorb peaks and to feed line-based processing. Watch `FillLevel` on hardware; a value that stays near 100% often indicates a bottleneck or deadlock. See [Recognizing Memory-Related Errors](#).

10.8.2 Camera Through LineBuffer into a 3×3 Kernel

In the following design, the DRAM buffer supplies lines for a neighborhood operator that builds a kernel (for example 3×3). The buffer adds at least one line of latency — more if it fills because the filter path blocks. Account for the shared DRAM bandwidth of the `LineBuffer` operator as well. The Design Rules Check in the screenshot confirms that the link formats and buffer settings are consistent before you build.

The screenshot displays the VisualApplets software interface. The main workspace shows a data stream graph with the following components and connections:

- CxpCamera** (input) connects to **LineBuffer**.
- LineBuffer** connects to **FIRkernelNxM**.
- FIRkernelNxM** connects to **FIRoperatorNxM**.
- FIRoperatorNxM** connects to **DmaToPC**.

A "Module Properties" dialog box is open for the **LineBuffer** module, showing the following parameters:

Parameter Name	Parameter Value	Parameter Unit	Parameter Type	Parameter Flag	Value Type
YLength	1024	pixel	Dynamic	Write	int (unsigned, 11 bit)
ParallelismConverter	No		Static	Write	enum
InfiniteSource	ENABLED		Static	Write	enum

On the right side, a "Resource" table shows the following data:

Resource	Count	Fill Level
LookupTables	54947	~ 34%
Flip Flops	54104	~ 16%
Block RAM	127	~ 17%
Ultra RAM	0	~ 0%
Embedded ALU	47	~ 3%

Below the resource table, a status message reads: "10:45:02 -- Netlist successfully generated in 00:00:07."

The "Design Rules Check Level 2" section shows:

- Program version: VisualApplets 3.7.0.75995-d25941f83
- Design name:
- Creation date: Tuesday, 21 July 2026 10:45:02
- 0 Errors
- 0 Warnings

A final status message at the bottom right reads: "10:45:02 -- Design Rules Check Level 2 successfully performed."

10.9 Related Topics

- [Stoppable and Non-Stoppable Data Streams](#) Tutorial topic — `InfiniteSource`, overflow, camera decoupling
- [Parallelism](#) Tutorial topic — bandwidth, RAM data width, `LineBuffer` parallelism conversion
- [Latency](#) Tutorial topic — line latency of buffer operators
- [Operator Types and Synchronization](#) Tutorial topic — buffers before `SYNC`
- [Device Resources](#) — platform RAM and Shared Memory Concept
- [Shared Memory](#) Application Note — efficient DRAM usage and bandwidth examples
- [Memory Library](#) in the Operator Reference — operator overview by memory type

11. Working with Kernels

This topic explains what a **kernel** is and how you use kernels in VisualApplets for neighborhood operations such as noise reduction, blur, and edge detection.

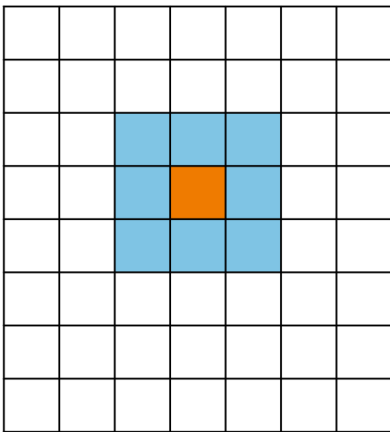
After you have read this topic, you understand how to build a kernel, process it with filter operators, and convert it back to single pixels before `DmaToPC`.

11.1 Kernel Structure

A kernel is a small window of neighboring pixel values around one position in an image. Many filters — noise reduction, blur, edge detection — compute a new output value from that neighborhood.

For a 3×3 kernel, the window contains nine values: the **current** pixel at that position plus eight neighbors. For odd kernel sizes, the current pixel sits in the **center** of the window.

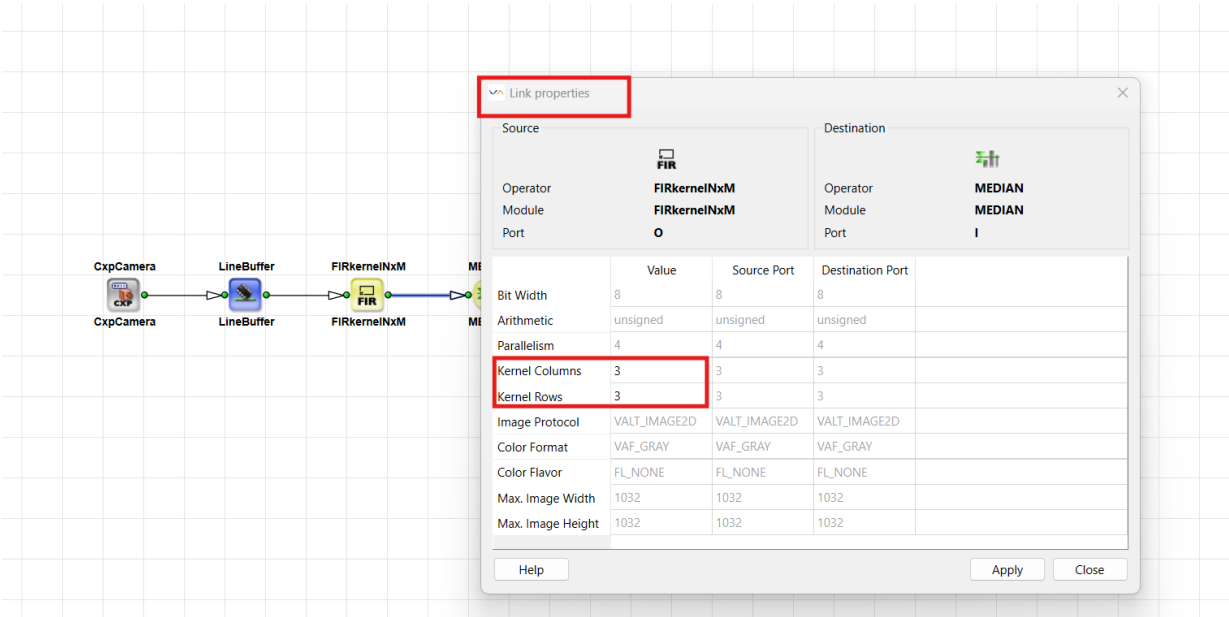
The kernel moves across the image **from left to right, row by row**. At each position, the filter uses the neighborhood around that pixel:



In VisualApplets, that neighborhood is defined by **link properties**. How you build the window, and where the current pixel sits inside it, depends on the operator. Those details follow below.

11.2 Kernel Columns and Kernel Rows

The kernel size is part of the **link properties**. `KernelColumns` is the width of the neighborhood window. `KernelRows` is its height. A 3×3 kernel therefore uses `KernelColumns = 3` and `KernelRows = 3`.



In VisualApplets, N is the number of kernel rows and M is the number of kernel columns. Kernel elements are addressed by index instead of coordinates. Indexing runs **row by row**, from top-left to bottom-right.

Always check the operator documentation for the exact mapping of kernel indices to image positions.

Not every operator accepts or forwards kernels larger than 1×1. Each operator documents the allowed values in its **Supported Link Format** table in the [Operator Reference](#). If you connect a kernel link to an operator that only allows `Kernel Columns = 1` and `Kernel Rows = 1`, VisualApplets reports a link format conflict.

For an overview of all link properties, see [Parameterization](#) Tutorial topic.

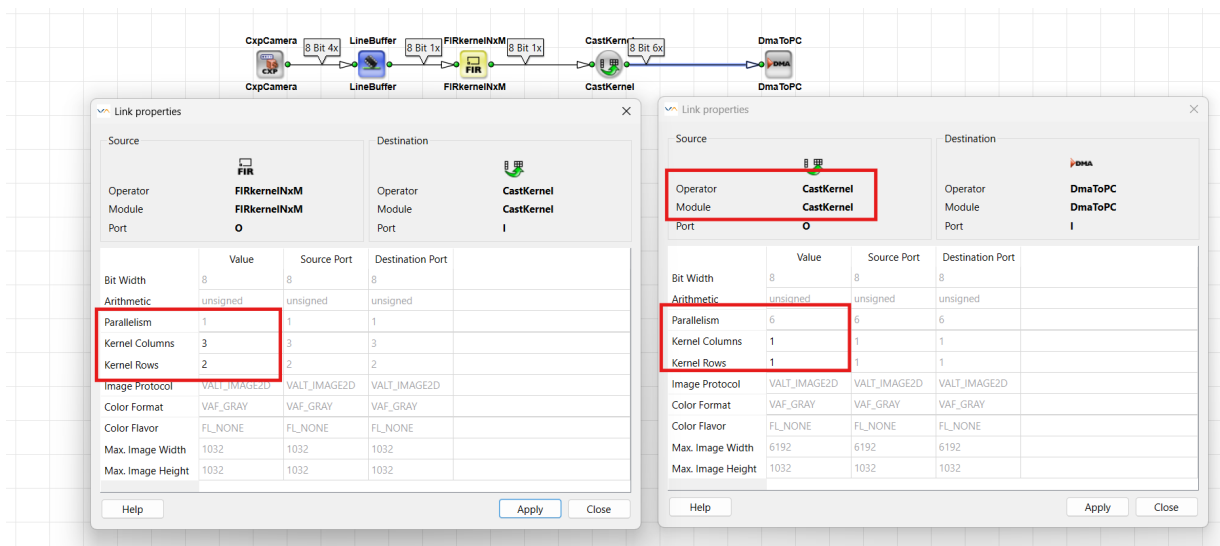
11.3 Kernels and Parallelism

Parallelism and **kernels** both increase the amount of data a link carries per clock cycle, but they describe different concepts:

	Parallelism	Kernel
Question	How many independent pixels pass per clock?	How many neighbor values belong to one position?
Typical use	Match camera / DMA bandwidth	Feed filters that need a neighborhood
Link properties	Parallelism	KernelColumns , KernelRows

You can keep **parallelism at 1** and still carry more data per clock by using a kernel: the link carries one neighborhood (for example 3×3 values) as one data word. The **amount of data per clock** rises because more values move per clock, not because more new image positions are processed per clock. The pixel-rate bandwidth formula in the [Parallelism](#) Tutorial topic does not multiply by the kernel size.

You can also combine both. The operator `CastKernel1` can re-interpret the product of parallelism and kernel size between input and output, as long as that product stays the same. For example, parallelism 1 with kernel 3×2 can become parallelism 6 with kernel 1×1.



For details, see [CastKernel](#) in the Operator Reference and [Parallelism](#) Tutorial topic.

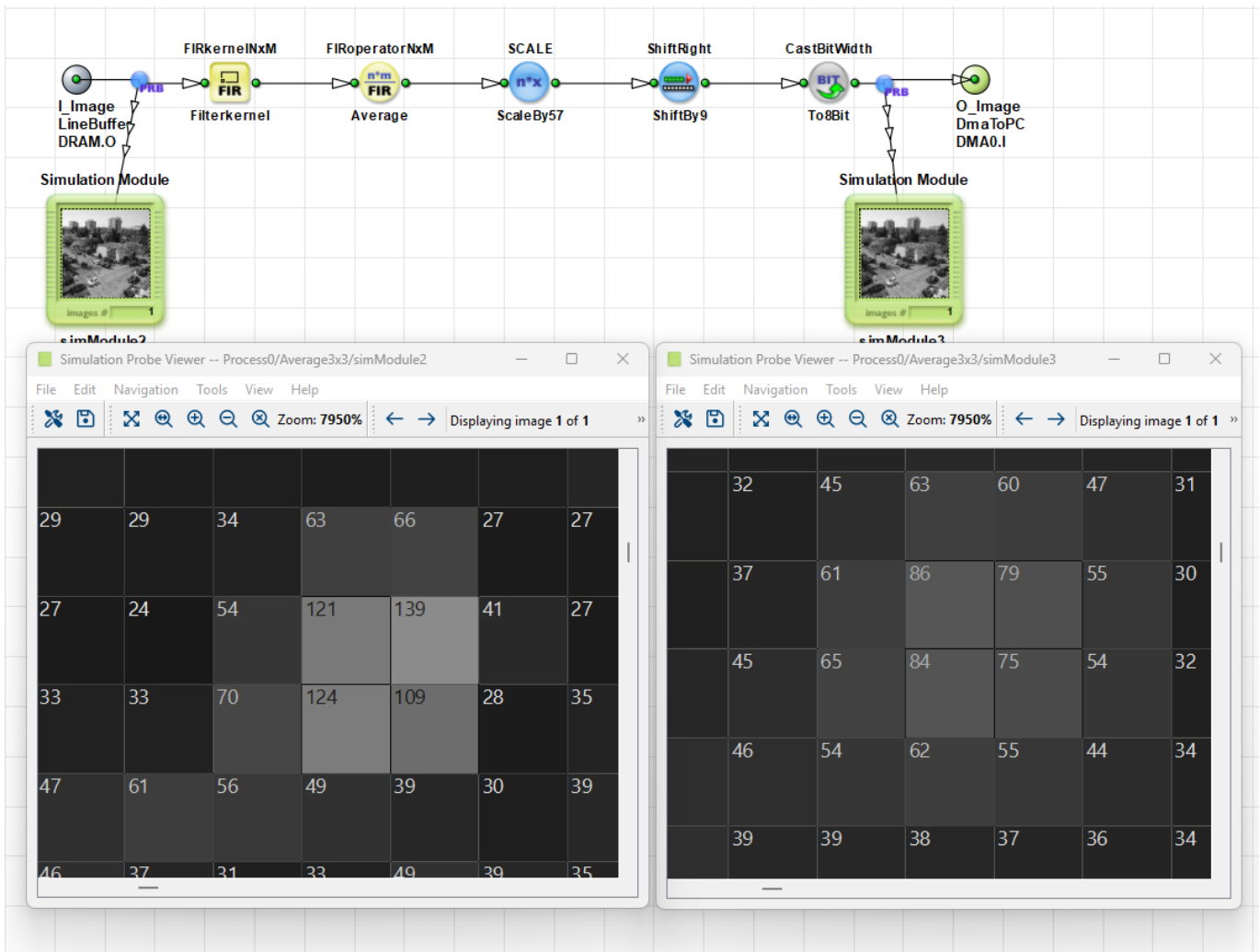
11.4 Building Kernels

To create neighborhood data from a normal pixel stream, use operators that set the kernel size on their **output** link. **FIRkernelNxM** builds a centered 2D kernel window. **LineNeighboursNx1** and **PixelNeighbours1xM** are a special case: they collect previous lines or previous pixels rather than a centered window. The operators differ in where the current pixel sits, in operator type, in FPGA cost and latency, and in edge handling (see below). Choose the operator that matches your use case: a centered 2D filter neighborhood, a comparison across previous lines or pixels, or a resource-efficient alternative to a full FIR-style window.

11.4.1 FIRkernelNxM

The operator **FIRkernelNxM** (library **Filter**) creates a rectangular kernel window around each pixel, including the neighbor pixels. Despite its name, it does **not** implement an FIR filter. It prepares the kernel data that filter operators need. For odd kernel sizes, the current pixel is in the **center** of the kernel. For kernels of even size, there is no exact center; the central position is displaced to the upper left.

Smoothing with a 3×3 Average Filter:



The design builds a centered 3×3 neighborhood with `FIRkernelNxM` (Filterkernel), averages it with `FIRoperatorNxM` (Average), then scales and casts the result back to 8-bit. The simulation probes compare pixel values before and after averaging: sharp local peaks become smoother, more uniform values.

You define the kernel size by editing the output link (`Kernel Columns` and `Kernel Rows`). The operator buffers the required pixels until the full neighborhood is available. At image borders, neighbors may be missing. Parameter `EdgeHandling` selects how missing values are filled (`EdgeMirrored` or `EdgeConstant`).

`FIRkernelNxM` is an **M-type** operator. Building a centered neighborhood such as 3×3 requires buffering and therefore uses more FPGA resources and adds more delay than the line- or pixel-neighbor operators below. Use `FIRkernelNxM` when a filter truly needs a **centered** 2D neighborhood — for example before `FIRoperatorNxM` for blur, averaging, or edge detection.

For details about the operator, see [FIRkernelNxM](#) in the Operator Reference.

11.4.2 LineNeighboursNx1 and PixelNeighbours1xM

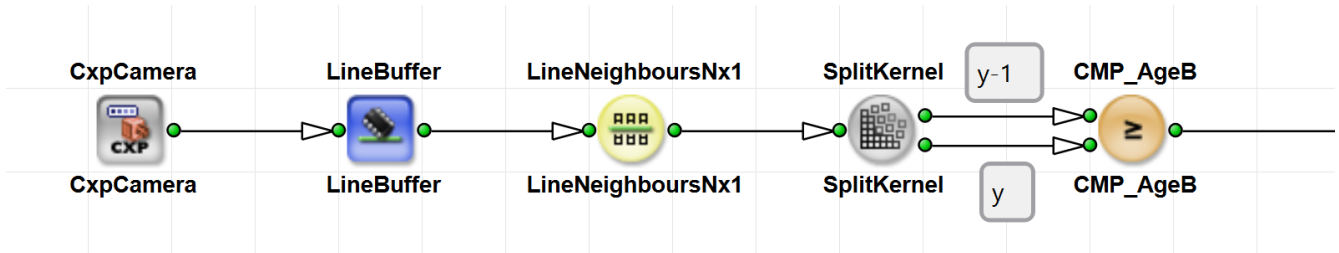
`LineNeighboursNx1` and `PixelNeighbours1xM` are a special case. They do not build a centered kernel in the usual sense.

`LineNeighboursNx1` creates a kernel of N rows and 1 column. The current pixel is at **kernel index 0**; the other elements are pixels from previous lines (above). `PixelNeighbours1xM` creates a kernel of 1 row and M columns. The current pixel is again at **kernel index 0**; the other elements are previous pixels on the same line (to the left).

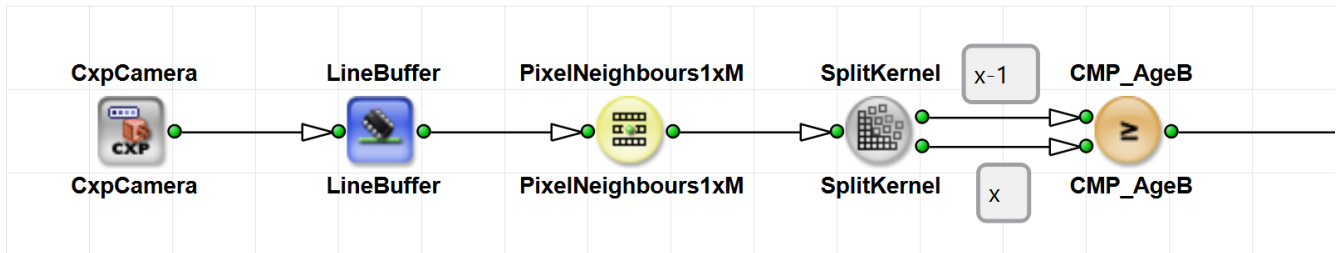
Unlike `FIRkernel1NxM`, these operators do **not** center the current pixel and do not provide "future" neighbors that have not been processed yet. They look only into the **past** (previous lines or previous pixels). They are **O-type** operators and do not add the line or pixel delay that comes from buffering a centered neighborhood. That makes them simpler and usually cheaper in FPGA resources than `FIRkernel1NxM`. You can still form a larger neighborhood — for example by combining both operators into a 3×3 window — but at the start of a line or image the past neighbors are not yet available, so those kernel elements stay empty until enough history has been collected. Missing pixels outside the image borders are set to parameter `Constant`; there is no mirrored edge handling.

Typical use cases:

- Compare or reduce values across **previous lines** with `LineNeighboursNx1` — for example find the minimum pixel value of the current pixel and the lines above it:

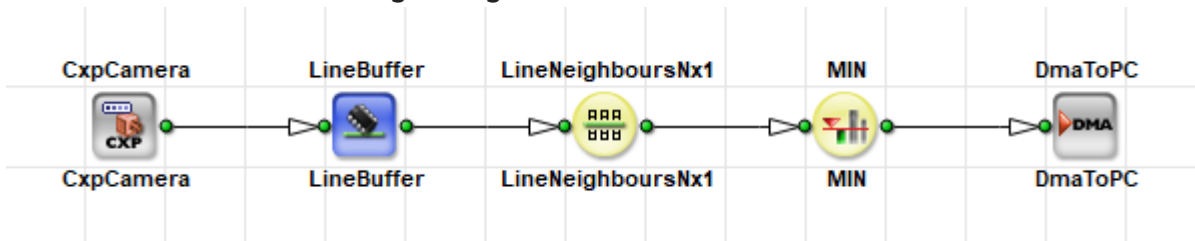


- Compare or reduce values across **previous pixels on the same line** with `PixelNeighbours1xM`.

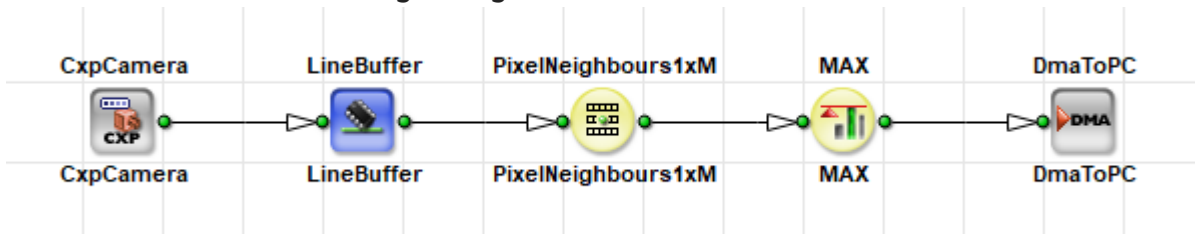


- Feed a reduction operator such as `MIN` or `MAX`: these operators expect a kernel on the input, select one value from the neighborhood (for example the smallest pixel), and output a 1×1 result. After that step the kernel is resolved again.

Minimum Pixel Value Across Neighboring Lines:



Maximum Pixel Value Across Neighboring Pixels:



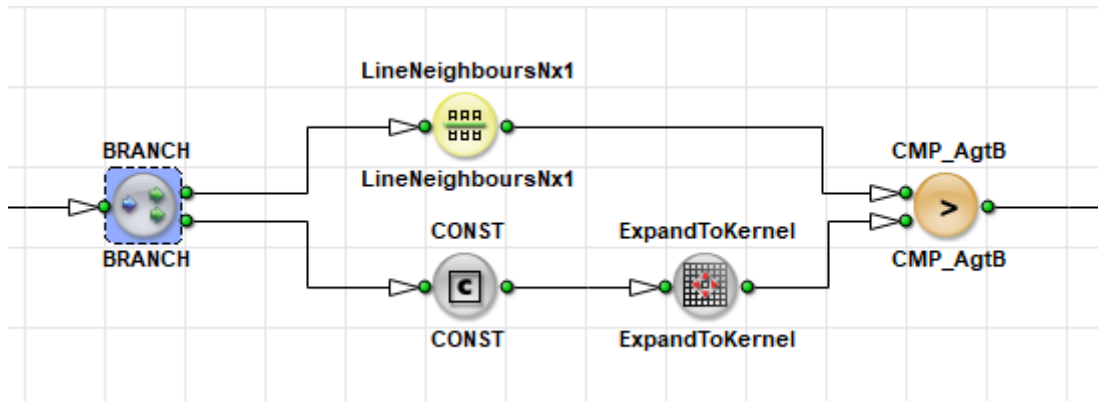
To build a two-dimensional kernel, you can combine `LineNeighboursNx1` followed by `PixelNeighbours1xM`.

For details about the operators, see [LineNeighboursNx1](#) in the Operator Reference and [PixelNeighbours1xM](#) in the Operator Reference.

11.4.3 ExpandToKernel

The operator `ExpandToKernel` expands a single pixel input to an arbitrary kernel size. It **replicates** the input value to every kernel component. You define the new kernel size on the output link.

Use `ExpandToKernel` when a downstream operator expects a kernel on the input, but you only have a **single** pixel value — for example a threshold, a mean, or a reference pixel. A common pattern is to expand that constant or reference so it matches an existing neighborhood from `FIRkernelNxM`, `LineNeighboursNx1`, or `PixelNeighbours1xM`. Then you can apply operators such as `CMP`, `SUB`, or `ADD` element-wise across the full kernel (for example compare every neighbor with the same threshold).



For details about the operator, see [ExpandToKernel](#) in the Operator Reference.

11.5 Processing Kernels

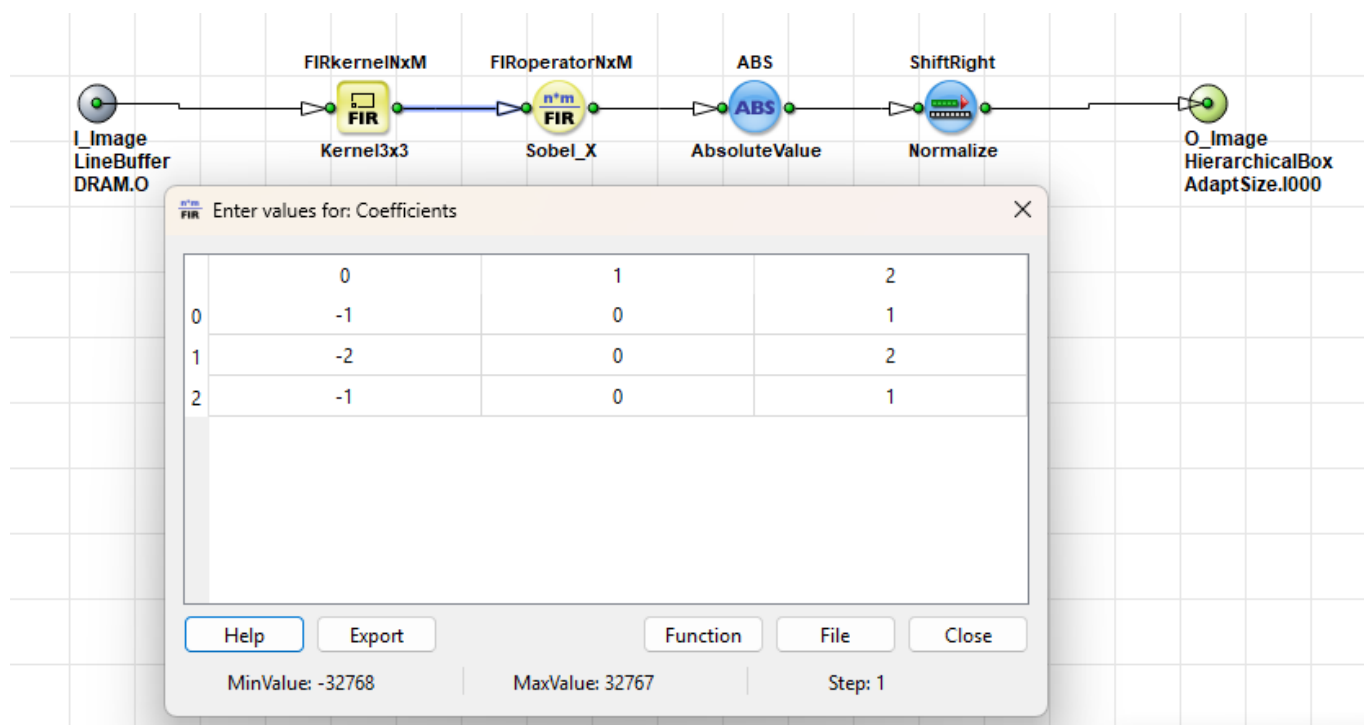
After you have built a kernel, you can filter it, extract a subset, merge or split kernel streams, or reorder kernel components. The operators in this section process an existing kernel. Before `DmaToPC`, you must reduce the kernel to 1×1 — see [Connecting to DMA](#).

11.5.1 FIRoperatorNxM

The operator `FIRoperatorNxM` multiplies each kernel element with a coefficient and sums the products. You can define these coefficients with the `Coefficients` parameter.

The output of the `FIRoperatorNxM` operator has the kernel size 1×1 , which means a single filtered pixel value per position. Use `FIRkernelNxM`, `LineNeighboursNx1`, or `PixelNeighbours1xM` to generate the required input kernel.

Use this operator for classical FIR-style filters (averaging, Gaussian, Sobel, and similar) when you already have a neighborhood on the link and need one filtered output pixel per position:



The screenshot shows the design example **Examples\Processing\Filter\EdgeDetection\SobelGradientX** delivered with VisualApplets.

For details about the operator, see [FIRoperatorNxM](#) in the Operator Reference.

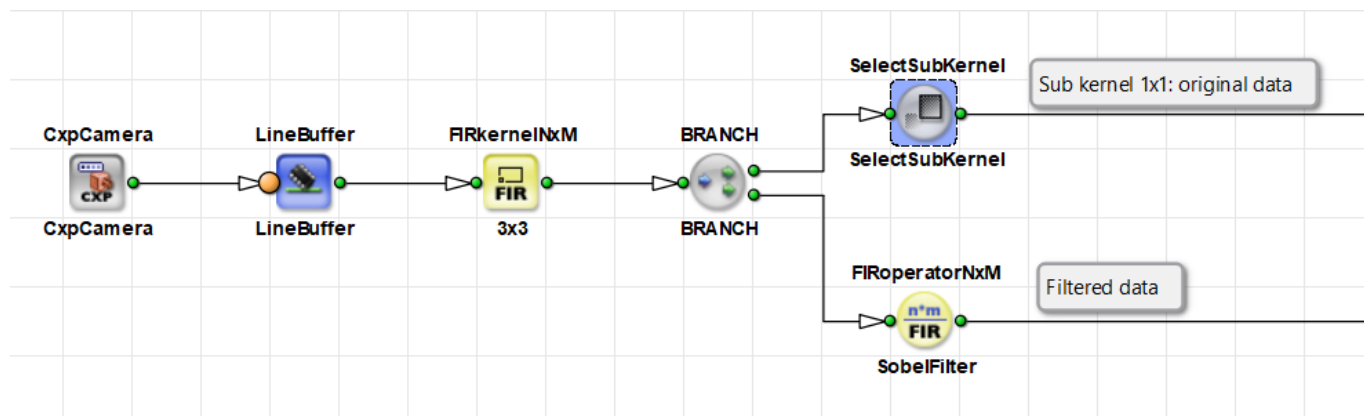
11.5.2 SelectSubKernel

The `SelectSubKernel` operator extracts a rectangular subset of the input kernel. Its parameters `FirstROW` and `FirstCOL` define the offset of the subset. The new kernel size is set on the output link.

With `SelectSubKernel` you extract the **central** element of a kernel — typically the original pixel at that image position — when you no longer need the full neighborhood.

A typical use case is to apply a filter and still keep the **original** image. After `FIRkernelNxM` has built the neighborhood, you branch the kernel link. One path computes the filtered result. The other path uses `SelectSubKernel` to extract a 1×1 sub-kernel — the original pixel at that image position. You then have both the filtered data and the unfiltered image.

Keeping Original Image Data Next to a Filter:



The design builds a 3×3 neighborhood with `FIRkernelNxM`, then branches. The upper path uses `SelectSubKernel` to extract the original pixel (sub-kernel 1×1). The lower path applies `FIRoperatorNxM` as a Sobel filter.

For details about the operator, see [SelectSubKernel](#) in the Operator Reference.

11.5.3 MergeKernel and SplitKernel

The operator `MergeKernel` merges several input links into one output kernel. For this, one kernel dimension (either rows or columns) must match on all inputs; the other dimension on the input may differ. The `Alignment` parameter with the values `horizontal` or `vertical` defines how the inputs are concatenated.

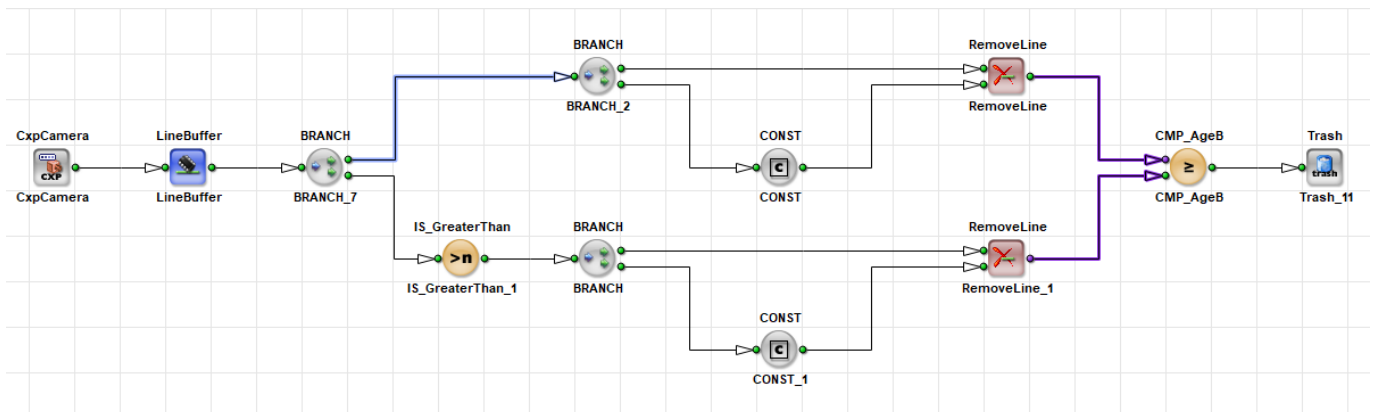
The opposite operator is `SplitKernel`. `SplitKernel` splits an $N \times M$ kernel into $N \times M$ separate output links, each with kernel size 1×1 . The number of outputs must equal the number of kernel components and is fixed when you instantiate the operator.

A typical use case is to combine **several variants of the same image** — for example different grayscale or processed versions — into one kernel so you can compare or process them together. Use `SplitKernel` when you need to process individual kernel components on separate paths again.

Together, `MergeKernel` and `SplitKernel` can also replace a design that would otherwise need a `SYNC` operator: `SYNC` usually requires buffers on its inputs and therefore costs FPGA resources. When the streams can be carried as kernel components instead, merging and splitting kernels is often the more resource-efficient approach.

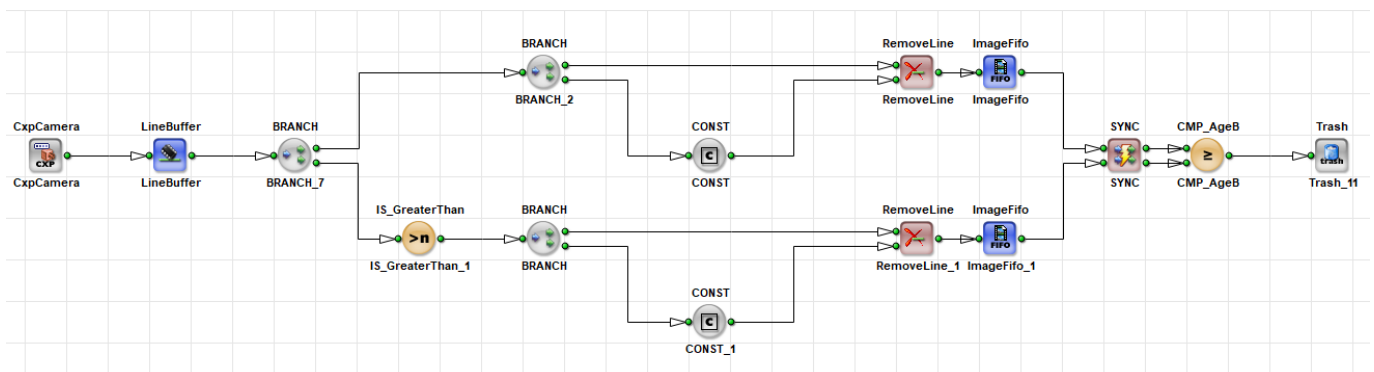
Use this pattern only for **several variants of the same image** that still share a common source, for example two different processings of one camera stream. Do **not** use `MergeKernel` to merge two independent cameras. Independent cameras remain asynchronous sources and still need `SYNC` (or another operator with asynchronous inputs) and buffers. See [Operator Types and Synchronization](#) Tutorial topic.

Sync Error When Merging Asynchronous Paths:



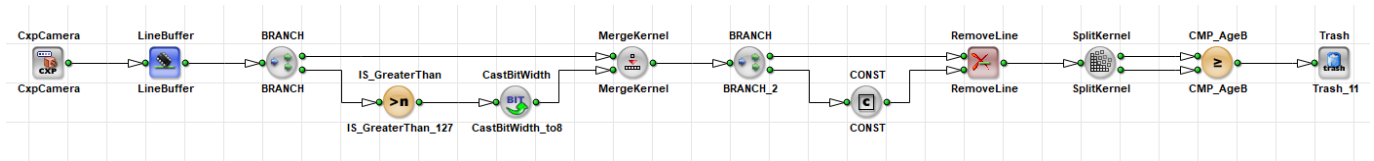
Here two branches process the same image differently and then feed a common `CMP` operator. Because an M-type operator on one path has made the branches asynchronous, VisualApplets reports a synchronization error (purple links) — you cannot connect them directly. This example is still one camera, not two independent cameras.

Sync Error Resolved with Buffers and SYNC:



One valid fix is to place an `ImageFifo` (or another buffer) on each path and re-synchronize with `SYNC` before `CMP`. This works, but the extra buffers cost FPGA resources.

Sync Error Resolved with MergeKernel and SplitKernel:



The resource-efficient alternative carries both variants as kernel components: `MergeKernel` combines the two paths into one kernel link, processing continues on that link, and `SplitKernel` separates the components again before `CMP`. No `SYNC` and no extra synchronization buffers are required.

For details about synchronization and synchronization errors, see the [Operator Types and Synchronization](#) Tutorial topic.

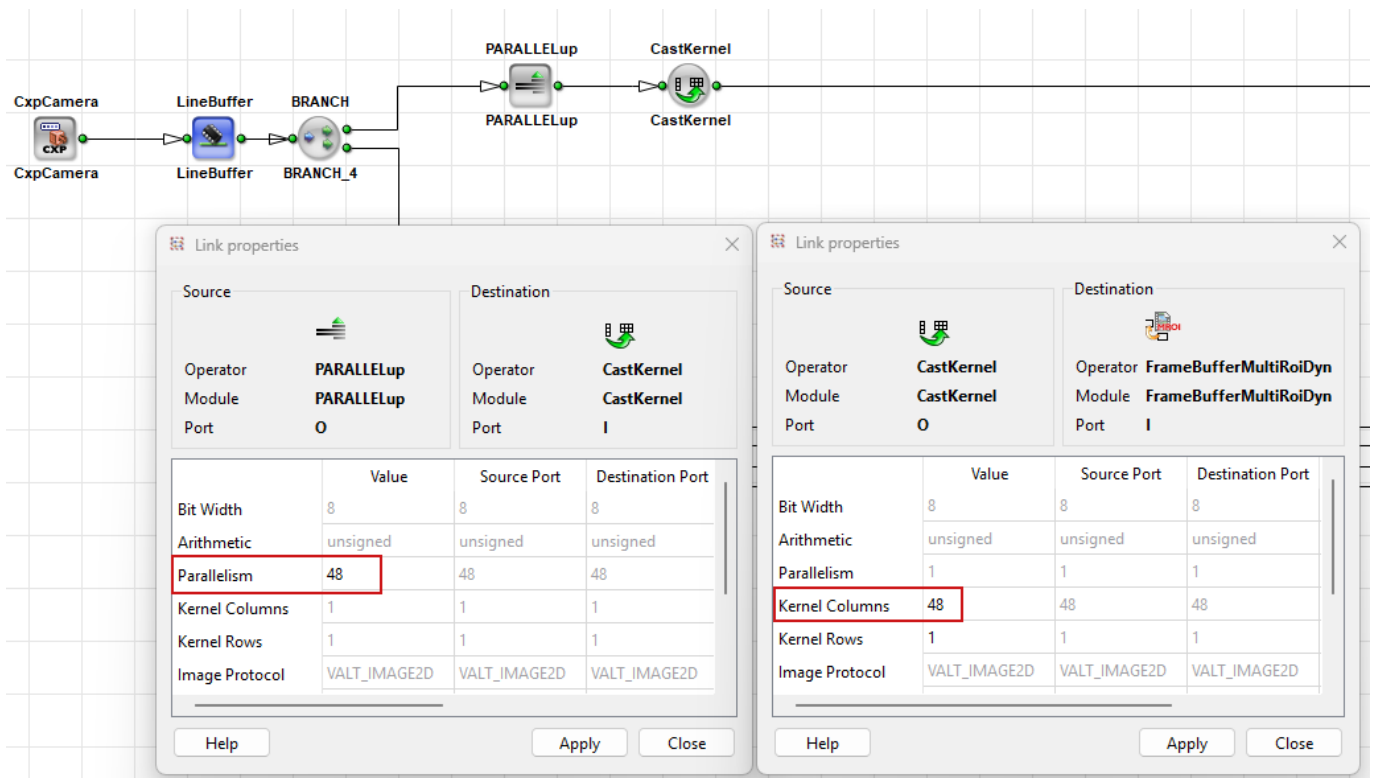
For details about the operators, see [MergeKernel](#) in the Operator Reference and [SplitKernel](#) in the Operator Reference.

11.5.4 CastKernel

`CastKernel` re-organizes parallelism and kernel size of the incoming data. You can change the kernel size on the output link. The product of kernel rows, kernel columns, and parallelism must be identical on input and output.

With `CastKernel` you can convert **parallelism into a kernel**. For example, parallelism 48 with kernel 1×1 becomes parallelism 1 with 48 kernel columns. The same 48 pixel values still move in one clock cycle, but they are now kernel components instead of parallel pixels. That is useful before memory operators that allow only a specific parallelism.

Converting Parallelism into a Kernel Before a Memory Operator:



In the screenshot, `PARALLELup` raises the parallelism to 48 (kernel 1×1). `CastKernel` re-interprets that as parallelism 1 with 48 kernel columns, so the following memory operator can accept the data.

For details about the operator, see [CastKernel](#) in the Operator Reference.

11.5.5 KernelRemap

`KernelRemap` remaps kernel components between input and output. The kernel dimensions stay the same. With the `SourceSelect` parameter, you can assign an input kernel index to each output kernel component.

With this operator, you can mirror or rotate the kernel content, map one input element to several outputs, or discard unused input elements.

Swapping Kernel Columns with KernelRemap:

The screenshot illustrates the configuration and results of the `KernelRemap` operator. At the top, a block diagram shows two simulation modules, `simModule8` and `simModule7`, connected to a `KernelRemap` operator. A dialog box titled "Enter values for: SourceSelect" is open, displaying a mapping table:

	0	1	2	3
0	3	2	1	0

The dialog also includes buttons for Help, Export, Function, File, and Close, along with MinValue: 0, MaxValue: 3, and Step: 1.

Below the diagram, two side-by-side Simulation Probe Viewer windows are shown. Both windows display a 2x2 grid of kernel components. The left window shows the kernel content before the remap, with the 'Column' dropdown set to 0. The right window shows the kernel content after the remap, with the 'Column' dropdown set to 3. The 'Display Properties' panel on the left of each window shows 'Image Size: 2, 2', 'Hex' selected, 'Signed' checked, 'Fractional bits: 1,0000', and 'Color Maps: Sequential_1'. The 'Sequence Viewer' at the bottom of each window shows a single data point.

The design in the screenshot shows how `KernelRemap` swaps kernel columns: column 0 and column 3 exchange places (and the inner columns follow). The mapping comes from the `SourceSelect` parameter, here `0, 1, 2, 3 → 3, 2, 1, 0`. The simulation probes show the kernel components before and after the remap.

For details about the operator, see [KernelRemap](#) in the Operator Reference.

11.5.6 Connecting to DMA

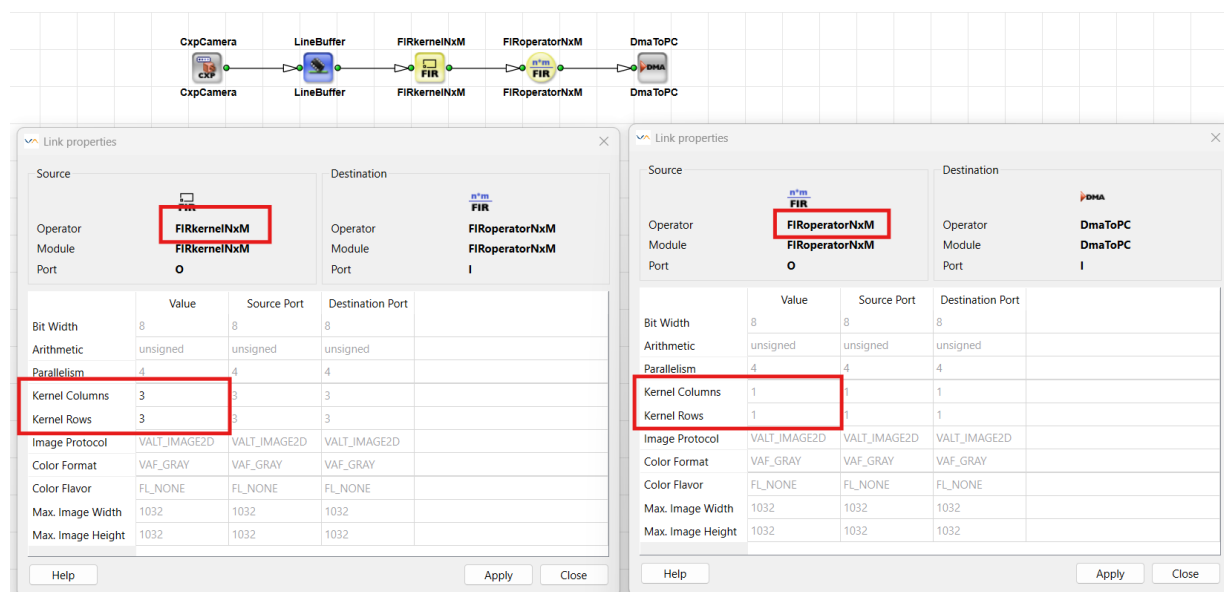
A raw kernel link must **not** go directly into `DmaToPC`. `DmaToPC` accepts only `Kernel Columns = 1` and `Kernel Rows = 1`.

Therefore, always place one of the following operators between the kernel output and `DmaToPC`. These operators convert the kernel to single pixels or to a smaller kernel you can then reduce further:

- `FIRoperatorNxM`
- `SelectSubKernel`
- `SplitKernel`
- `MIN` or `MAX`
- `Median`

11.6 Example Design

A typical filter path looks like this:



1. The camera delivers single pixels (kernel 1×1).
2. A buffer, for example `LineBuffer` or `ImageBuffer`, decouples the camera from processing and provides the line data that neighborhood operators need.
3. `FIRkernelNxM` builds a kernel out of the neighboring pixels, for example 3×3 or 5×5, on its output link.
4. `FIRoperatorNxM` applies filter coefficients and outputs one pixel per position.
5. `DmaToPC` transfers the single-pixel stream to the host.

Detailed filter examples, for example averaging, Gaussian, Sobel, adaptive threshold, and others, are listed in the **Examples of Use** sections of the `FIRkernelNxM` in the Operator Reference and `FIRoperatorNxM` in the Operator Reference.

Design Guideline

Always check the **Supported Link Format** of the following operator before you create a kernel larger than 1×1 on a link. Especially before `DmaToPC`, reduce the kernel to 1×1.

11.7 Related Topics

- [Parameterization](#) Tutorial topic — `KernelColumns` and `KernelRows` link properties

- [Parallelism](#) Tutorial topic — parallelism versus kernel data rate
- [Latency](#) Tutorial topic — latency of buffer and filter operators
- [FIRkernelNxM](#) in the Operator Reference
- [FIRoperatorNxM](#) in the Operator Reference
- [Operator Reference](#)
- [Parameterization](#) in the User Manual — editing link and module parameters in the GUI