Peter the Great St. Petersburg Polytechnic University

Institute of Computer Science and Cybersecurity

Graduate School of Computer Technologies and Information Systems

Lecture: Nios II Custom Components and Instructions

Subject: Automation of discrete device design (in English)

Completed by student of group 5130901/10101 _________ Nepomnyaschiy M.T.
(signature)

Lecturer ____________________________ Antonov A.P.
(signature)

Saint Petersburg

2024

Table of contents

List of illustrations

Introduction

The presentation revolves around optimizing Nios II processor systems through custom components and instructions. It explores how these customizations enhance system performance, comparing the benefits of custom components versus custom instructions. Understanding these customization options is crucial for maximizing efficiency in embedded systems.

2. Custom Components

You can make custom components for Nios II processor systems. This is useful for systems where certain tasks are done a lot. Making these tasks into hardware components makes them faster than doing them with software. Also, it lets the processor do other things while the custom component works. Qsys has a tool called the Component Editor for making these custom components.

2.1. Custom Components

- Map into Nios II processor memory space
- Can be on-chip or off-chip
  - HDL code or an external component on your board
    - HDL can live inside Qsys system or out
- Conforms to a Qsys supported standard interfaces

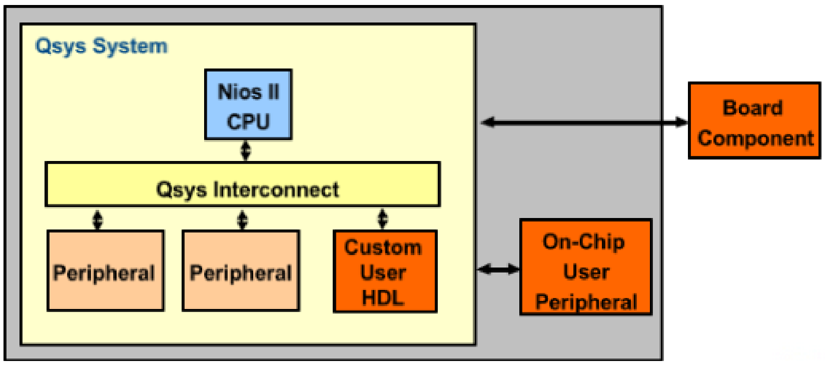
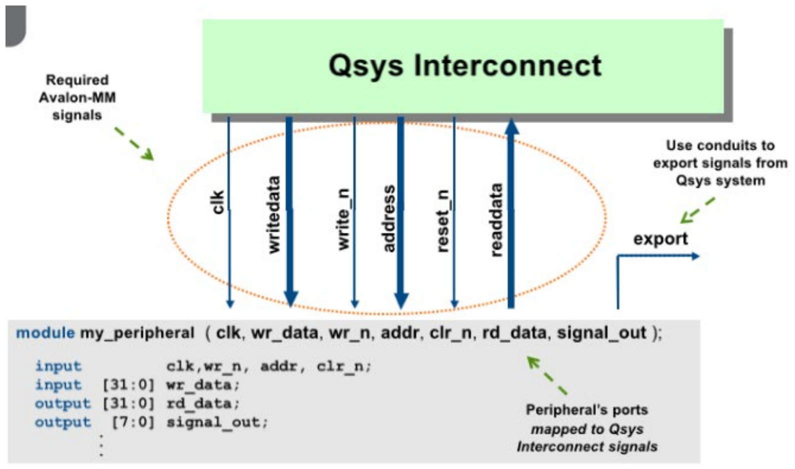


Figure 1 – Nios II Embedded Design Suite

To connect custom components to your Nios II system and Qsys Interconnect, you have a few options.

1. Custom user logic inside Qsys: You can integrate custom components directly into your Qsys system module.
2. Peripheral outside Qsys module on the same FPGA: Alternatively, the custom component can be outside the Qsys module but still on the same FPGA.
3. Off-chip connection: You can also connect custom components off-chip by routing signals to the FPGA pins and linking them to components on another board.

2.2. Example Signal Mapping


Qsys Interconnect
Required Avalon-MM signals
clk
writedata
write_n
address
reset_n
readdata
Use conduits to export signals from Qsys system
export
Peripheral's ports mapped to Qsys Interconnect signals

```
module my_peripheral  ( clk, wr_data, wr_n, addr, clr_n, rd_data, signal_out );

  input          clk,wr_n, addr, clr_n;
  input   [31:0] wr_data;
  output  [31:0] rd_data;
  output   [7:0] signal_out;
          :
```

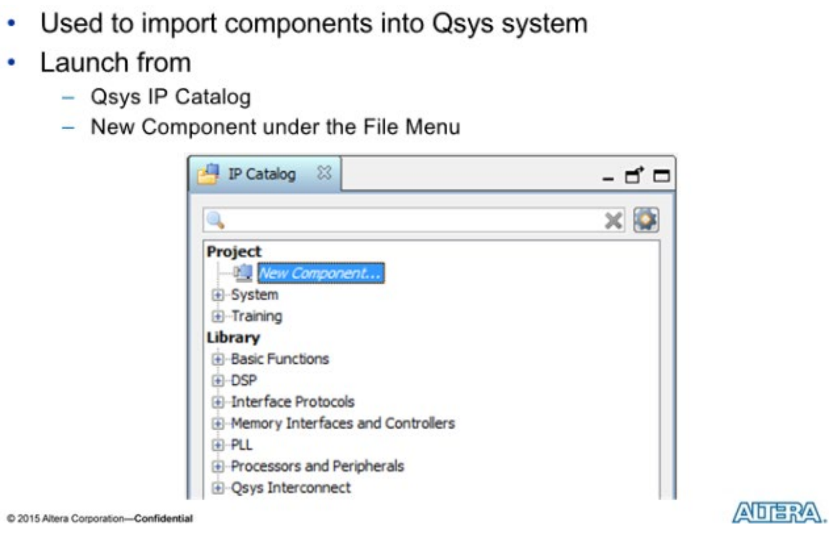
Figure 2 – Example Signal Mapping

Altera's tool flow makes it easy to connect custom components to the system interconnect.

Using the Qsys component editor, you can easily map the ports of your component to equivalent signal types in the System Interconnect. For example, if your component is a Verilog module named "my peripheral" and it's a MM Slave component, you can map its clk to the clk signal and wr_data to writedata, and so on, following the Avalon MM Standard Interface signal types in Qsys.

For signals that don't fit into the standard interface, like the signal_out output signal mentioned here, you can map them to the export signal of the conduit interface. This simplifies the integration process even for non-standard signals.

2.3. Component Editor



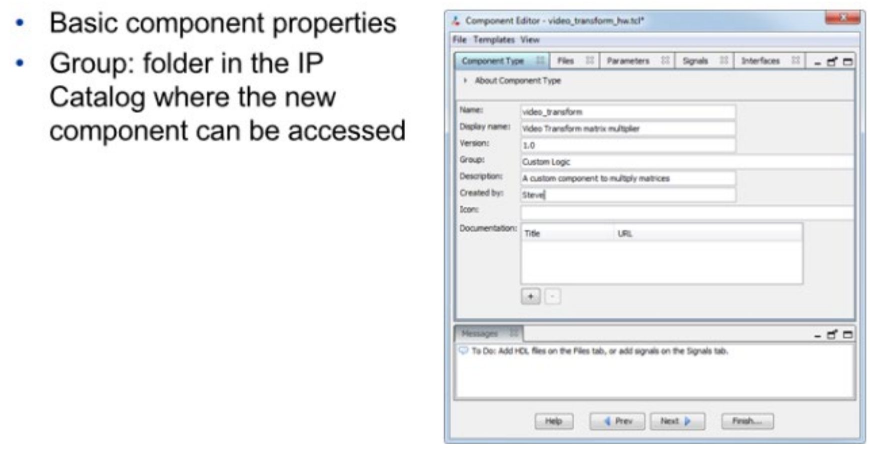
- Used to import components into Qsys system
- Launch from
  - Qsys IP Catalog
  - New Component under the File Menu

© 2015 Altera Corporation—Confidential

Figure 3 – Component Editor

The tool you use to map signals in your component to their corresponding Avalon interface and signal types is called the "Component Editor." You can access it from within Qsys, either directly from the IP Catalog or through the File menu. The Component Editor has several tabs for different functions or settings.

2.3.1. Component Editor (Component Type Tab)



- Basic component properties
- Group: folder in the IP Catalog where the new component can be accessed

Component Editor - video_transform_hw.tcl*

File Templates View

Component Type | Files | Parameters | Signals | Interfaces

About Component Type

Name:	video_transform
Display name:	Video Transform matrix multiplier
Version:	1.0
Group:	Custom Logic
Description:	A custom component to multiply matrices
Created by:	Steve
Icon:	
Documentation:	

Title	URL

Messages

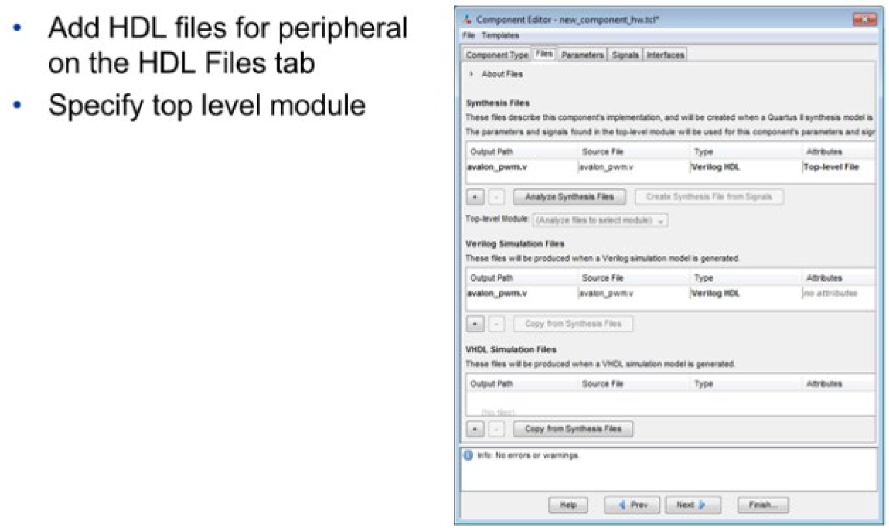
To Do: Add HDL files on the Files tab, or add signals on the Signals tab.

Help | Prev | Next | Finish...

Figure 4 – Component Type Tab

The Component Type tab is where you input basic information about the component, such as its name, description, and icon. You can also include links to documentation here. This information will be visible in the component's parameter editor. The Display Name determines how the component will appear in the Component Library, while the Group decides the folder it will be in. You can choose an existing folder or create a new one. By default, new components are placed in a folder called "Custom Logic."

2.3.2. Component Editor (Files Tab)



- Add HDL files for peripheral on the HDL Files tab
- Specify top level module

Component Editor - new_component_hw.tcl*

File Templates

Component Type | Files | Parameters | Signals | Interfaces

About Files

Synthesis Files

These files describe this component's implementation, and will be created when a Quartus II synthesis model is
The parameters and signals found in the top-level module will be used for this component's parameters and sig

Output Path	Source File	Type	Attributes
avalon_pwm.v	avalon_pwm.v	Verilog HDL	Top-level File

Analyze Synthesis Files | Create Synthesis File from Signals

Top-level Module: (Analyze files to select module)

Verilog Simulation Files

These files will be produced when a Verilog simulation model is generated.

Output Path	Source File	Type	Attributes
avalon_pwm.v	avalon_pwm.v	Verilog HDL	no attributes

Copy from Synthesis Files

VHDL Simulation Files

These files will be produced when a VHDL simulation model is generated.

Output Path	Source File	Type	Attributes
(No files)			

Copy from Synthesis Files

Info: No errors or warnings.

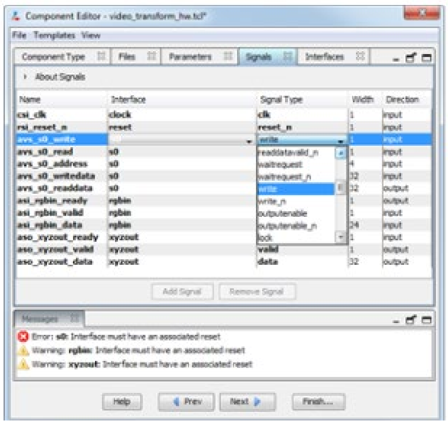
Help | Prev | Next | Finish...

Figure 5 – Files Tab

In the HDL Files tab of the component editor, you can add the HDL files that comprise your peripheral. You can also specify the top level of the component hierarchy if needed. Additionally, you can choose whether each file is used for simulation and/or synthesis. When you add HDL files, the component editor runs analysis and elaboration on them to catch errors and understand the structure of the code, including input/output signals and parameters associated with the component.

2.3.3. Component Editor (Signals Tab)

- Map component signals to Qsys Interface types and Signal Names on the Signals tab



Name	Interface	Signal Type	Width	Direction
csi_clk	clock	clk	1	input
rsi_reset_n	reset	reset_n	1	input
avs_s0_write		write	1	input
avs_s0_read	s0	readdatavalid_n	1	input
avs_s0_address	s0	waitrequest	4	input
avs_s0_writedata	s0	waitrequest_n	32	input
avs_s0_readdata	s0	write	32	output
asi_rgbin_ready	rgbin	write_n	1	output
asi_rgbin_valid	rgbin	outputenable	1	input
asi_rgbin_data	rgbin	outputenable_n	24	input
aso_xyzout_ready	xyzout	lock	1	input
aso_xyzout_valid	xyzout	valid	1	output
aso_xyzout_data	xyzout	data	32	output

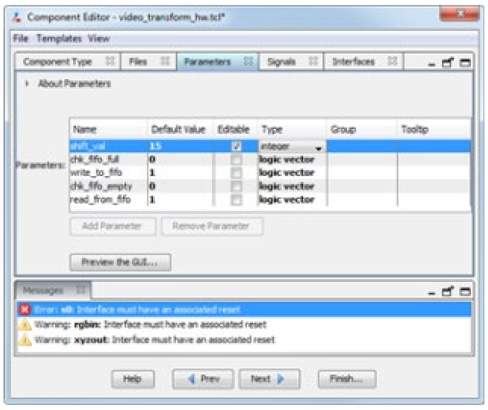
```
Error: s0: Interface must have an associated reset
Warning: rgbin: Interface must have an associated reset
Warning: xyzout: Interface must have an associated reset
```

Figure 6 – Signals Tab

In the Signals Tab of the Component Editor, you can map the signals in your component to the Avalon Standard Interface signal names. Additionally, you can create interfaces if needed. If you've named signals in your component's HDL file appropriately, the component editor will automatically recognize and map the signals, saving you from manually mapping them.

2.3.4. Component Editor (HDL Parameters)

- If component has HDL parameters, they show up on HDL Parameters tab
- Select which parameters can be edited at instantiation time in Parameters window



Name	Default Value	Editable	Type	Group	Tooltip
shift_val	15	✓	integer		
chk_fifo_full	0		logic vector		
write_to_fifo	1		logic vector		
chk_fifo_empty	0		logic vector		
read_from_fifo	1		logic vector		

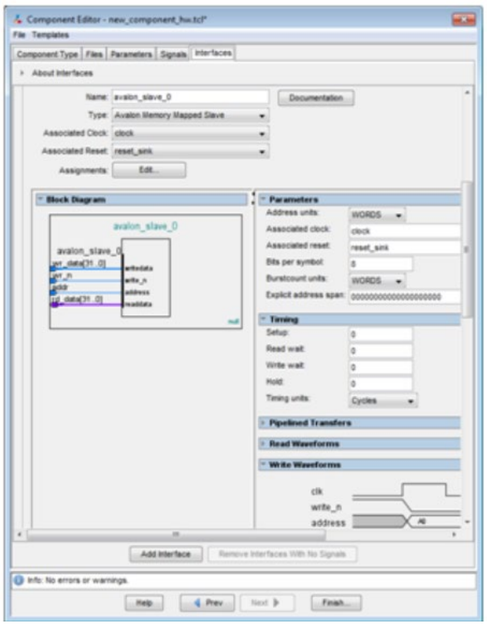
```
Error: s0: Interface must have an associated reset
Warning: rgbin: Interface must have an associated reset
Warning: xyzout: Interface must have an associated reset
```

Figure 7 – HDL Parameters

In the Signals Tab of the Component Editor, if your component's HDL code includes parameters or generics, they will appear here. This allows you to override their default settings for each instance in your system. Additionally, you can preview the graphical user interface (GUI) that appears when your IP is instantiated, along with the parameters that can be adjusted. This gives you control over customizing your component's behavior.

2.3.5. Component Editor (Interfaces Tab)

- Define properties for all component interfaces on the Interfaces tab
  - Timing
  - Associated Clocks and Resets
  - Etc.



Name: avalon_slave_0
Type: Avalon Memory Mapped Slave
Associated Clock: clock
Associated Reset: reset_sink

Parameters
Address units: WORDS
Associated clock: clock
Associated reset: reset_sink
Bits per symbol: 8
Burstcount units: WORDS
Explicit address span: 00000000000000000000

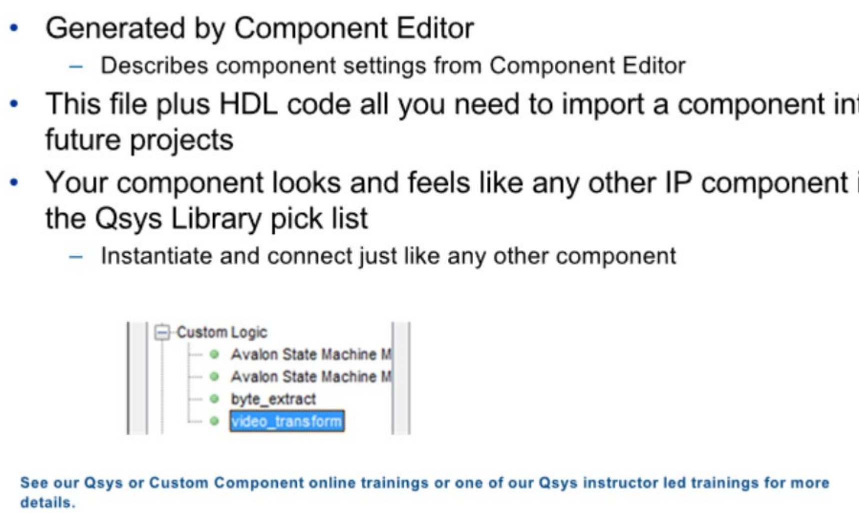
Timing
Setup: 0
Read wait: 0
Write wait: 0
Hold: 0
Timing units: Cycles

Info: No errors or warnings.

Figure 8 – Interfaces Tab

In the Interfaces Tab of the Component Editor, you can define parameters for your component's interfaces. This includes timing parameters, creating new interfaces if necessary, assigning clocks and resets, and removing unused interfaces identified in the Signals Tab. By navigating through all the animations, you'll be guided through the entire peripheral UI, enabling you to configure and customize your component comprehensively.

2.4. “hw.tci” File



- Generated by Component Editor
  - Describes component settings from Component Editor
- This file plus HDL code all you need to import a component into future projects
- Your component looks and feels like any other IP component in the Qsys Library pick list
  - Instantiate and connect just like any other component

Custom Logic
- Avalon State Machine M
- Avalon State Machine M
- byte_extract
- video_transform

See our Qsys or Custom Component online trainings or one of our Qsys instructor led trainings for more details.

Figure 9

The Component Editor generates an output file known as the hardware Tcl file. This file captures all the settings configured in the Component Editor GUI. Importing your component into another Qsys project is simple: ensure that Qsys can locate the hardware Tcl file, as well as the associated HDL and/or SDC files for your component. This streamlined process facilitates easy integration of custom components into different Qsys projects.

2.5. Custom Component Integration Into Qsys System

Custom Component Integration Into Qsys System

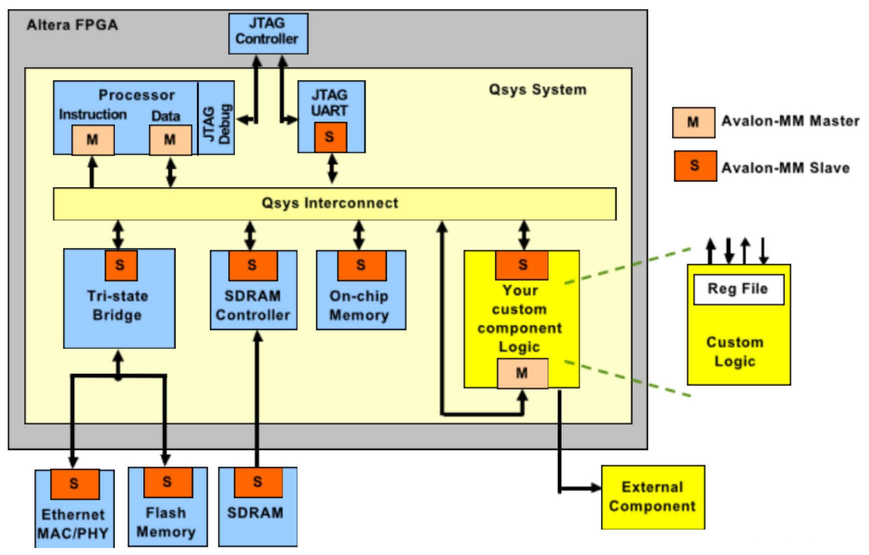


Figure 10 – Custom Component Integration Into Qsys System

In this example system, a custom component is integrated, featuring both master and slave ports that connect to the interconnect. This means the component can both initiate transactions (master) and respond to requests from other components (slave). This versatility allows the custom component to interact effectively within the system, enhancing its functionality and capabilities.

2.6. Edit Component Parameters

- Right Click then select Edit
  - Re-launches Component Editor
- Refresh system after editing component
  - Select Refresh System from the File menu

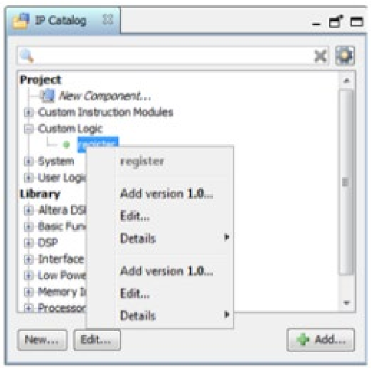


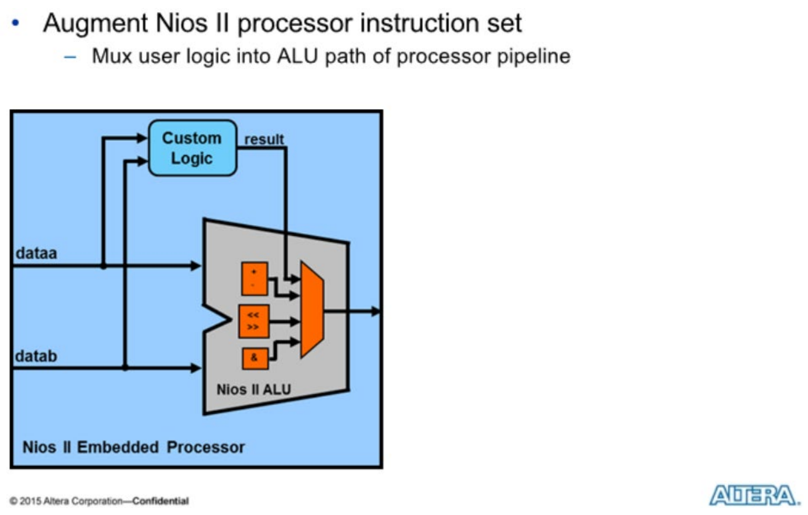
Figure 11 – Edit Component Parameters

If you need to modify the settings of your peripheral in the Component Editor, you can do so by right-clicking on the component in the library and selecting Edit. This action relaunches the Component Editor, enabling you to make changes and regenerate the hardware Tcl file. This "in-place" editing feature eliminates the need to remove and re-instantiate components for changes to take effect. However, remember to refresh the system after making changes so that Qsys re-reads all the_hw.tcl files in your project to ensure the updates are applied correctly.

3. Custom Instructions

With the Nios II processor, you can make software run faster by adding custom instructions. These instructions simplify complex tasks, like digital signal processing or packet processing, turning multiple steps into just one hardware-based step. This makes your programs more efficient, especially for tasks that need to be done quickly.

3.1. Custom Instructions

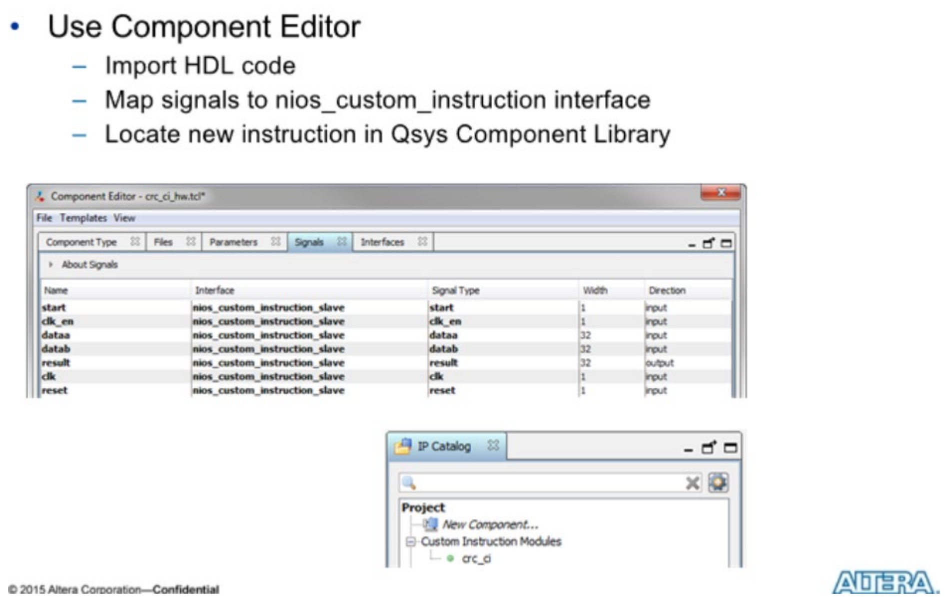


- Augment Nios II processor instruction set
  - Mux user logic into ALU path of processor pipeline

Figure 12 – Custom Instructions

Custom instructions in the Nios II processor are like personalized tools that sit alongside the main processing unit. They're designed to handle specific tasks efficiently. These instructions can be simple, just needing inputs and outputs, or more complex, requiring extra signals. Each processor can have up to 256 of these customized instructions. They can also communicate with the processor's memory using additional signals. If needed, they can even talk to other parts of the system through a special interface. This customization gives developers the flexibility to fine-tune the processor for their specific needs, making their applications run smoother.

3.2. Create Custom Instructions in Qsys



- Use Component Editor
  - Import HDL code
  - Map signals to nios_custom_instruction interface
  - Locate new instruction in Qsys Component Library

Component Editor - crc_ci_hw.tcl*

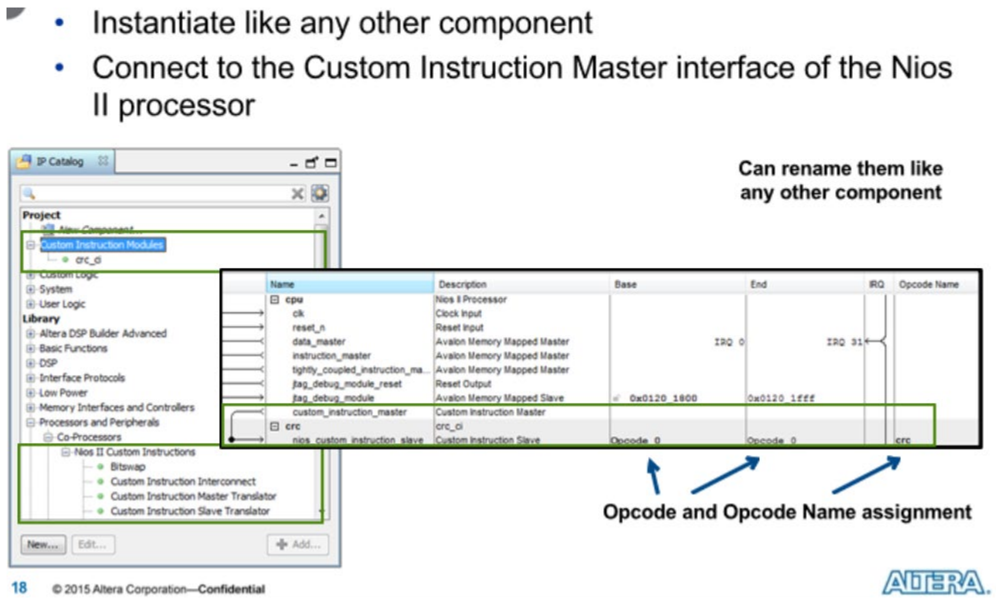
Name	Interface	Signal Type	Width	Direction
start	nios_custom_instruction_slave	start	1	input
clk_en	nios_custom_instruction_slave	clk_en	1	input
dataa	nios_custom_instruction_slave	dataa	32	input
datab	nios_custom_instruction_slave	datab	32	input
result	nios_custom_instruction_slave	result	32	output
clk	nios_custom_instruction_slave	clk	1	input
reset	nios_custom_instruction_slave	reset	1	input

© 2015 Altera Corporation—**Confidential**

Figure 13 – Create Custom Instructions

To import a custom instruction, you utilize the component editor similar to creating a custom component. Within the editor, select "nios_custom_instruction" as the interface type for your signals. Custom instructions can be developed using Verilog, VHDL, or EDIF formats. Once created, these instructions will appear in the component pick list under the Custom Instruction Modules folder. From there, you can effortlessly add them to your system and establish connections to the Nios II Processor.

3.3. Add Custom Instruction to System



- Instantiate like any other component
- Connect to the Custom Instruction Master interface of the Nios II processor

IP Catalog
Project
New Component...
Custom Instruction Modules
crc_ci
Custom Logic
System
User Logic
Library
Altera DSP Builder Advanced
Basic Functions
DSP
Interface Protocols
Low Power
Memory Interfaces and Controllers
Processors and Peripherals
Co-Processors
Nios II Custom Instructions
Bitswap
Custom Instruction Interconnect
Custom Instruction Master Translator
Custom Instruction Slave Translator
New...
Edit...
Add...

Name	Description	Base	End	IRQ	Opcode Name
cpu	Nios II Processor				
clk	Clock Input				
reset_n	Reset Input				
data_master	Avalon Memory Mapped Master	IRQ 0	IRQ 31		
instruction_master	Avalon Memory Mapped Master				
tightly_coupled_instruction_ma...	Avalon Memory Mapped Master				
jtag_debug_module_reset	Reset Output				
jtag_debug_module	Avalon Memory Mapped Slave	0x0120_1800	0x0120_1fff		
custom_instruction_master	Custom Instruction Master				
crc	crc_ci				
nios_custom_instruction_slave	Custom Instruction Slave	Opcode 0	Opcode 0		crc

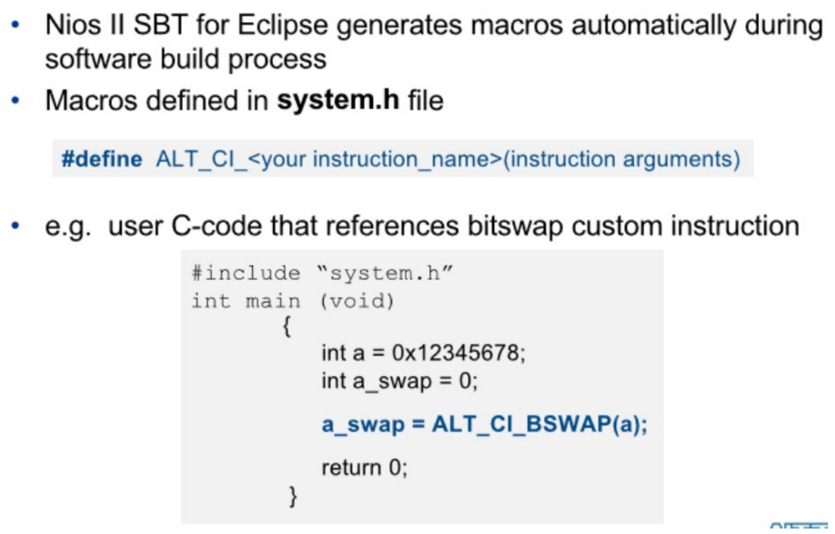
Can rename them like any other component
Opcode and Opcode Name assignment

18 © 2015 Altera Corporation—**Confidential**

Figure 14 – Add Custom Instruction to System

Once you've added the custom instruction from the IP Catalog, you can rename its instance just like any other component. Once integrated into the Qsys system, you can then connect it to the Custom Instruction Master Interface of the Nios II Processor. This interface allows the processor to communicate with and utilize the custom instruction effectively within the system.

3.4. Language Software Interface



- Nios II SBT for Eclipse generates macros automatically during software build process
- Macros defined in **system.h** file

```
#define ALT_CI_<your instruction_name>(instruction arguments)
```

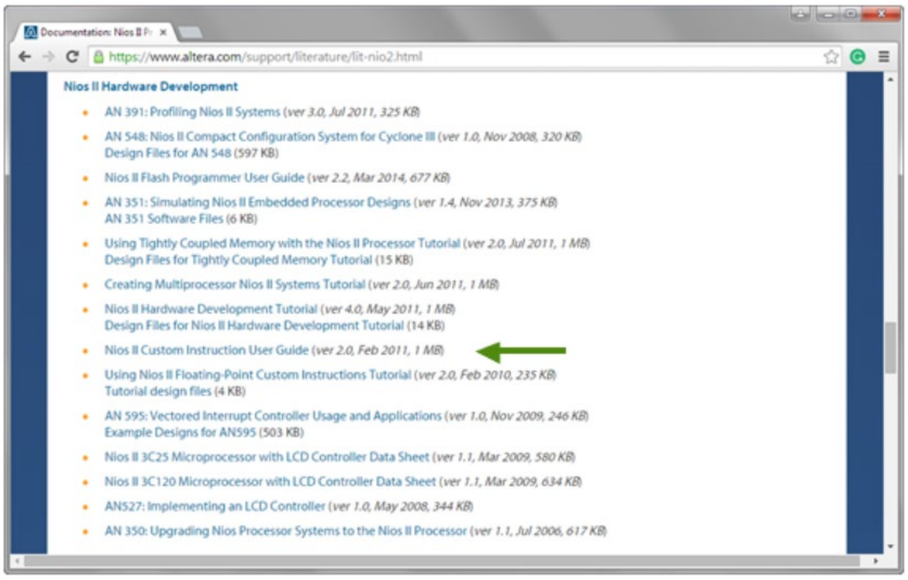
- e.g. user C-code that references bitswap custom instruction

```
#include "system.h"
int main (void)
      {
        int a = 0x12345678;
        int a_swap = 0;

        a_swap = ALT_CI_BSWAP(a);

        return 0;
      }
```

Figure 15 – Language Software Interface

Custom instructions offer a straightforward software interface. The macro to invoke your custom instruction is automatically defined in the system.h file of your BSP project, prefixed with ALT_CI. For instance, consider the example of performing a bit swap, a task that's challenging in software but simple in hardware. By employing the bit swap custom instruction, you can achieve around a 70x performance boost. In the provided code snippet, the macro is used to pass the integer "a" to the custom instruction, and the result is stored in "a_swap." This streamlined interface enhances performance significantly for tasks that benefit from hardware acceleration.

3.5. Nios II Software Instruction User Guide



Nios II Hardware Development

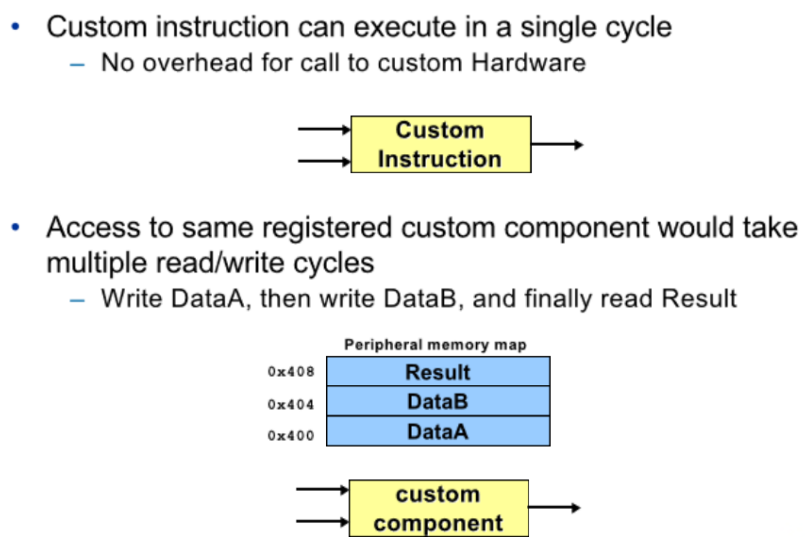
- AN 391: Profiling Nios II Systems (*ver 3.0, Jul 2011, 325 KB*)
- AN 548: Nios II Compact Configuration System for Cyclone III (*ver 1.0, Nov 2008, 320 KB*)
  Design Files for AN 548 (597 KB)
- Nios II Flash Programmer User Guide (*ver 2.2, Mar 2014, 677 KB*)
- AN 351: Simulating Nios II Embedded Processor Designs (*ver 1.4, Nov 2013, 375 KB*)
  AN 351 Software Files (6 KB)
- Using Tightly Coupled Memory with the Nios II Processor Tutorial (*ver 2.0, Jul 2011, 1 MB*)
  Design Files for Tightly Coupled Memory Tutorial (15 KB)
- Creating Multiprocessor Nios II Systems Tutorial (*ver 2.0, Jun 2011, 1 MB*)
- Nios II Hardware Development Tutorial (*ver 4.0, May 2011, 1 MB*)
  Design Files for Nios II Hardware Development Tutorial (14 KB)
- Nios II Custom Instruction User Guide (*ver 2.0, Feb 2011, 1 MB*)
- Using Nios II Floating-Point Custom Instructions Tutorial (*ver 2.0, Feb 2010, 235 KB*)
  Tutorial design files (4 KB)
- AN 595: Vectored Interrupt Controller Usage and Applications (*ver 1.0, Nov 2009, 246 KB*)
  Example Designs for AN595 (503 KB)
- Nios II 3C25 Microprocessor with LCD Controller Data Sheet (*ver 1.1, Mar 2009, 580 KB*)
- Nios II 3C120 Microprocessor with LCD Controller Data Sheet (*ver 1.1, Mar 2009, 634 KB*)
- AN527: Implementing an LCD Controller (*ver 1.0, May 2008, 344 KB*)
- AN 350: Upgrading Nios Processor Systems to the Nios II Processor (*ver 1.1, Jul 2006, 617 KB*)

https://www.altera.com/en_US/pdfs/literature/ug/ug_nios2_custom_instruction.pdf

Figure 16 - Nios II Software Instruction User Guide

For further details on custom instructions, refer to the Nios II Custom Instruction User Guide accessible at Altera.com. This resource provides comprehensive information and guidance on leveraging custom instructions to enhance the performance and functionality of Nios II processors.

4. Custom Instruction vs. Custom Component



- Custom instruction can execute in a single cycle
  - No overhead for call to custom Hardware
- Access to same registered custom component would take multiple read/write cycles
  - Write DataA, then write DataB, and finally read Result

Figure 17 – Custom Instruction vs. Custom Component

Deciding between custom components and custom instructions hinges on your specific application. Both offer significant performance enhancements over software-only operations.

Custom instructions excel in tasks that require minimal setup and can be executed in a single cycle without going through the System Interconnect. They are ideal for operations with up to two inputs and one output, such as Floating Point Multiply, where you can achieve up to a 30X performance improvement over optimized software.

For more complex algorithms like CRC calculations, custom components may be preferable. They offer better performance by incorporating additional capabilities like DMA to accelerate operations. Custom components are particularly beneficial for tasks requiring extensive computation or data manipulation beyond what custom instructions can provide.

Conclusion

Custom components and instructions offer powerful tools for enhancing the performance of Nios II processor systems. By tailoring the processor core to specific application needs, developers can achieve significant performance improvements. Whether utilizing custom components for complex algorithms or leveraging custom instructions for streamlined operations, the flexibility provided by these customization options empowers designers to optimize system efficiency. As embedded systems continue to evolve, understanding and harnessing the capabilities of custom components and instructions will remain essential for achieving optimal performance and functionality.