CYPRESS SEMICONDUCTOR CORPORATION
Internal Correspondence

Date:	Saturday 2/2/2013	**WW: 1307**
To:	**PSoC Apps**	
Author:	Chris Keeser (KEES)	
Author File#:	KEES#196	
Subject:	**Datapath Super Primitive**	
Distribution:	PSoC Apps	

Summary:

Please welcome another four (four!) installments into the super primitive category, this time with the intent of getting you up and running with a datapath faster and easier than ever before. These four datapath super primitives eliminate 95% of the busywork associated with developing for a datapath and get you trying, testing and creating with PSoC's powerful UDB datapaths in no time at all.

These datapath super primitives allow you to start designing with an 8, 16, 24 or 32 bit datapath, preconfigured and bristling with features to help you try out new ideas and debug when things go wrong. These super primitives provide you with a readymade component symbol, configurable component parameter (A0, A1, D0, D1 initializations *right in the customizer*, FIFO configuration of status reporting and easy access to the FIFO single buffer mode as well as optional debugging hardware signals and reset). Premade API files simplify component development with a header file containing all the necessary datpath registers, pointers, masks, modes and shifts already defined for you, as well as a C file that provides a Start() API ready to go with all the features you enable in the customizer.

Excited? I'm just getting started. These super primitives also include a skeleton verilog file, with all the necessary code to get started already included such as standard inputs and outputs, component parameters, a state machine and a preconfigured datapath of your choosing (8, 16, 24, or 32 bit). The preconfigured datapaths are already chained for you so they "just work" and include shifting setups for left shifts and arithmetic right shifts. The datapaths are also moved into the merge region of the verilog file, so when you add inputs and outputs to your symbol, simply re-generate your verilog file, and the new inputs and outputs are added for you automatically, with nothing lost!

I can tell you are ready to start using these super primitives, but I am not done packing the awesomeness into these components. These super primitives also come packaged with a component debug capability file, giving you incredibly easy access to the working registers of the datapath (A0, A1, D0, D1, even the FIFOs in single buffer mode), simplifying debug and reducing the headaches associated with learning the datapaths. And to top it all off, you also get a DMA capability file, giving the DMA wizard easy access to all your important registers, simplifying the setup of getting data into and out of your datapath component.

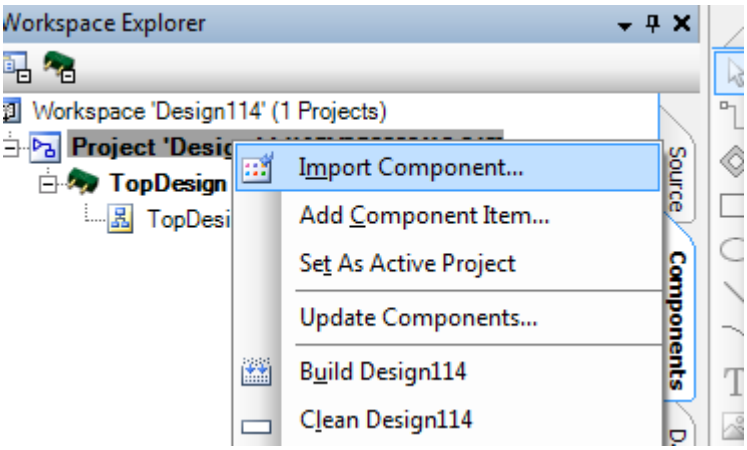
With these components, all you need to do to start using a datapath is import the desired super primitive, point the Datapath configuration tool at the premade verilog file, configure your datapath and write your verilog to control it and GO!

Attached File Summary:

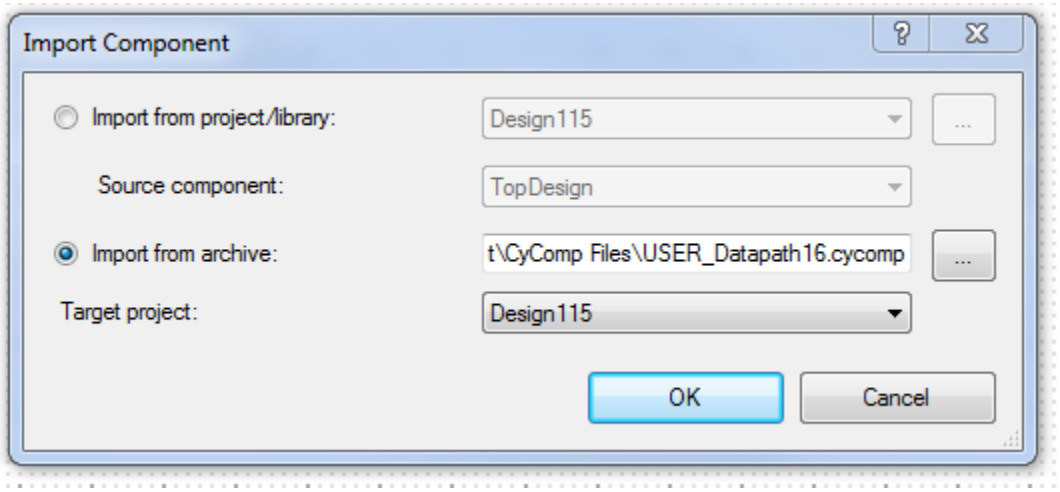
File # 2 is a zip file containing the 4 super primitive components. To use the file, download it and replace the .PDF extension with .ZIP. The four .cycomp files are inside this zip file.

Details:

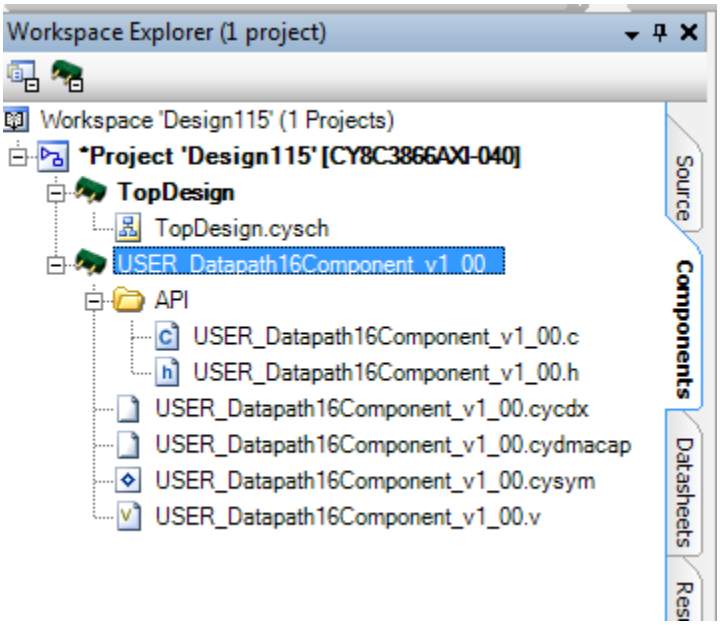
To get started with these new super primitives, open a project, navigate to the component tab in the workspace explorer, right click on you project and select "Import Component"



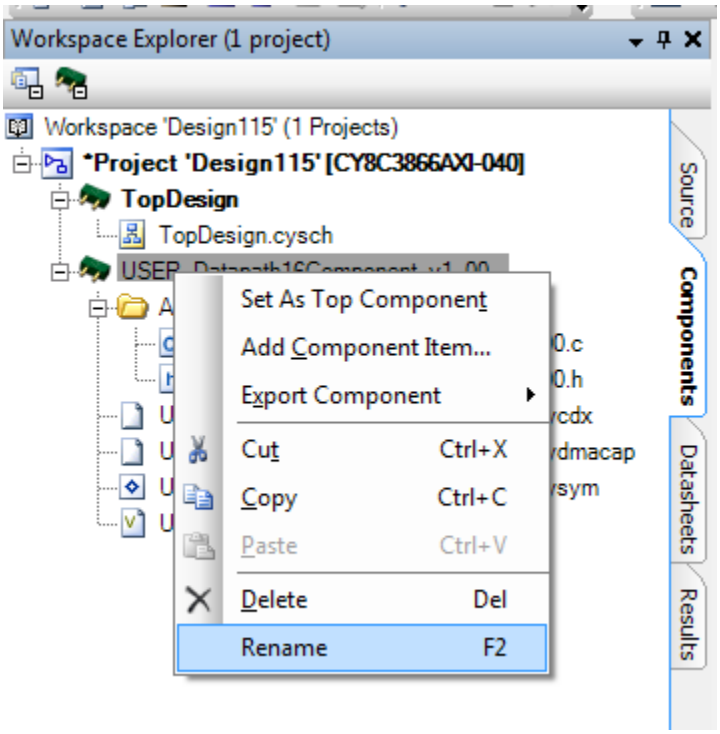
When the dialog box pops up, select "Import from archive" and navigate to the locations of the "USER_Datapath8/16/24/32.cycomp" file. Click OK. It is at this point you select the width of the datapath you want to use. The USER_Datapath8.cycomp is an 8 bit wide datapath, the USER_Datapath16.cycomp is a 16 bit wide datapath, the USER_Datapath24.cycomp is a 24 bit wide datapath and the USER_Datapath32.cycomp is a 32 bit wide datapath.



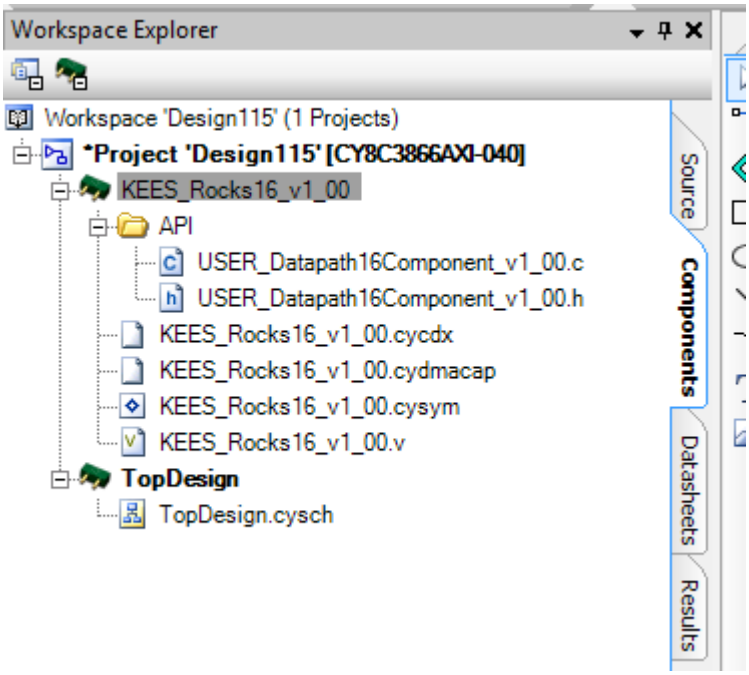
Your workspace explorer will now contain all the files for your component



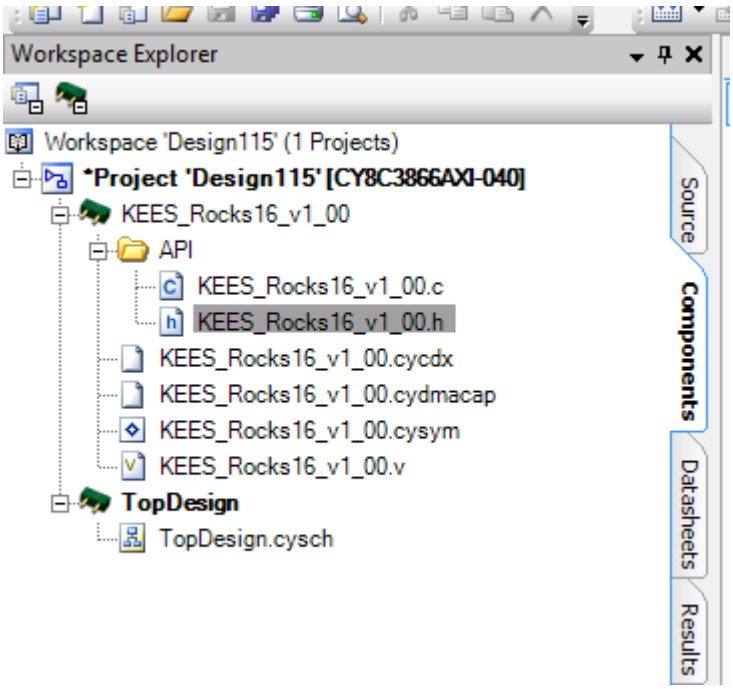
First thing you should do is rename the component by right clicking on the component and selecting “Rename.”



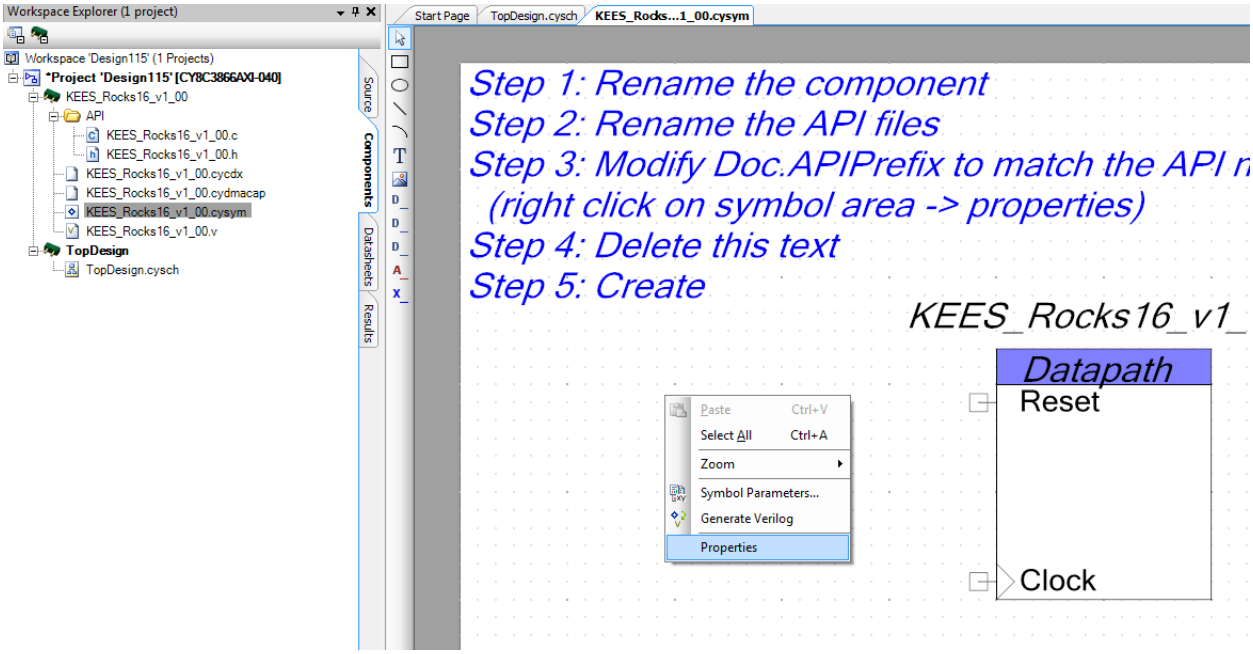
Rename it to something with your initials, or your name instead of USER and either leave the rest alone, or give a name that reflects your final purpose. Make sure to leave the _v1_00 appended to the end so that you can make new versions later if you choose.



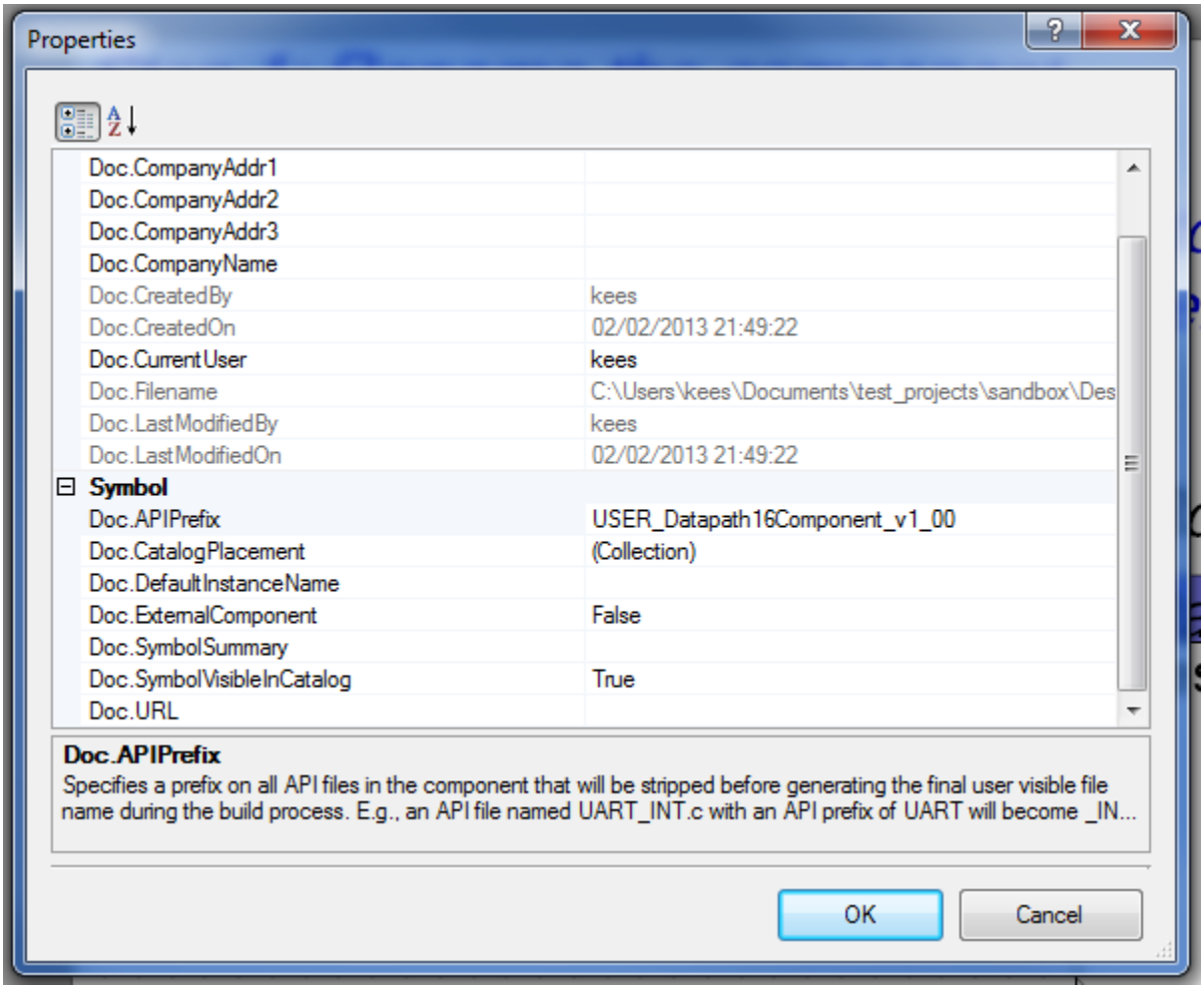
Notice how the API files did not change. You need to manually change them to match the new component name by right clicking on them and selecting “Rename”



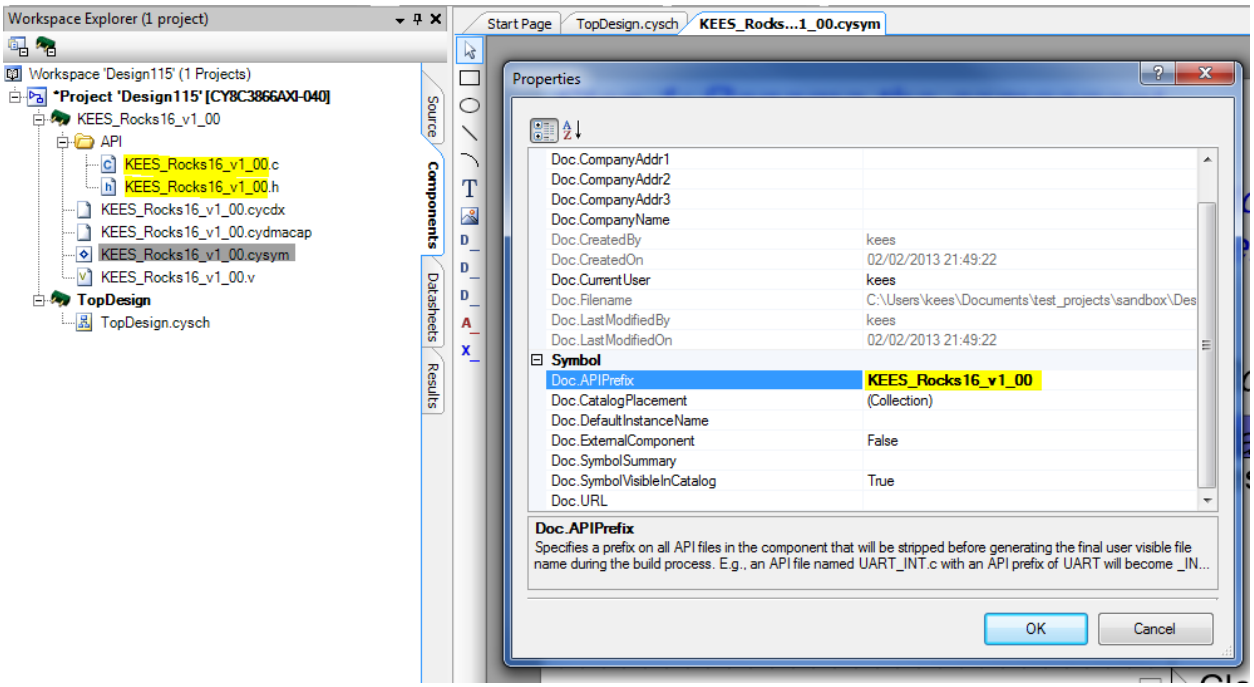
Now this step is important. Open the **.cysym** file, right click in the symbol area (not on the symbol itself) and select “Properties”



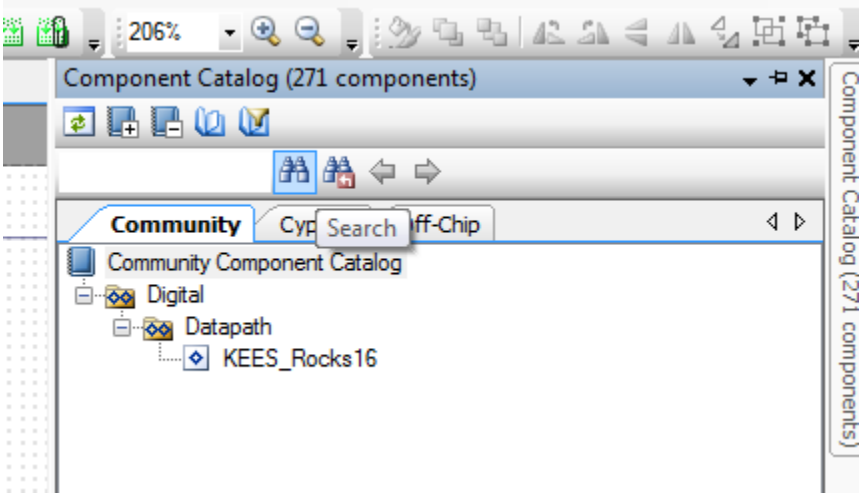
(The component symbols also include text to remind you of these steps. This text can be deleted at any time)



Change the Doc.APIPrefix to match the name of the renamed API files.



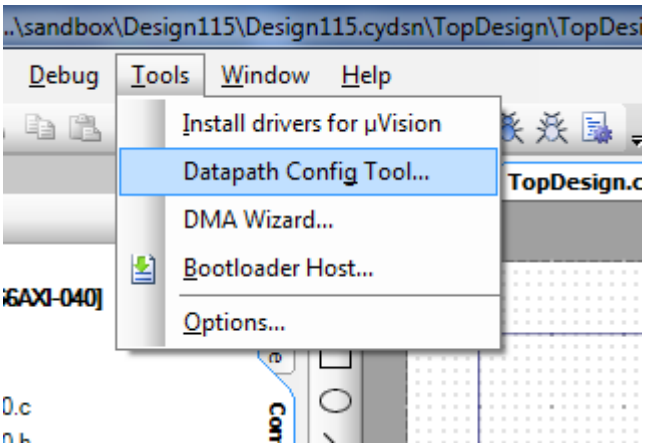
And that concludes all the component customization that you MUST do. Any further customization is optional. At this point, your component will be located in the “Community/Digital/Datapath/…” library tab.



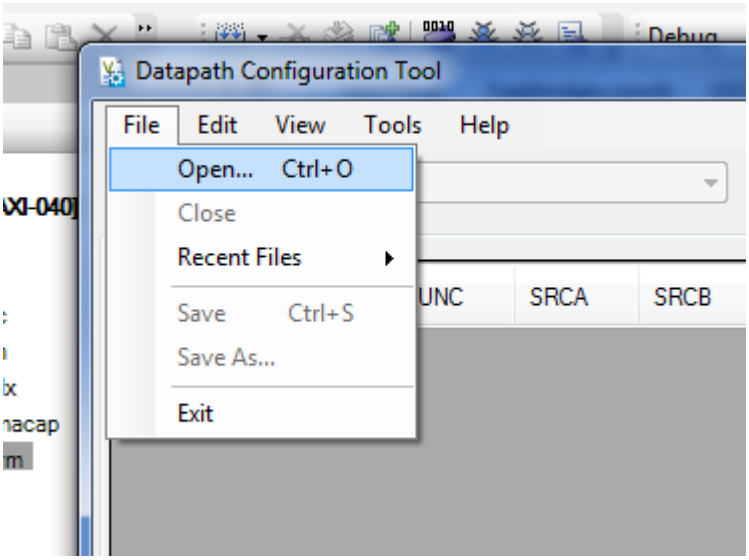
Getting started with you super primitive

Now that you have your component in your workspace, you can start to experiment with the datapath. Depending on your work flow with datapaths, you may dive right into the verilog file, or you may fire up the datapath configuration tool and set up your datapath instructions. For the purpose of this memo, I’m going to fire up the datapath configuration tool, and walk you through finalizing the setup of your new datapath component.

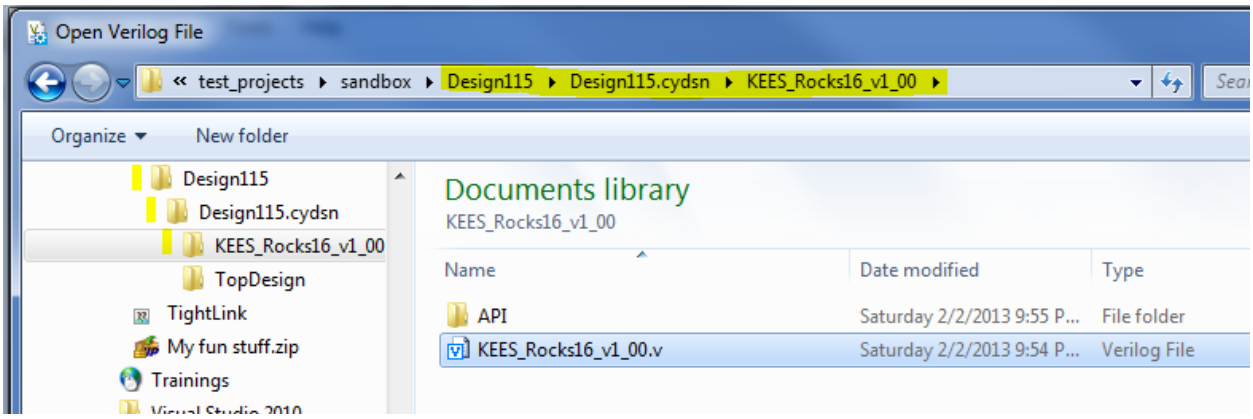
Go to Tools->Datapath configuration tool



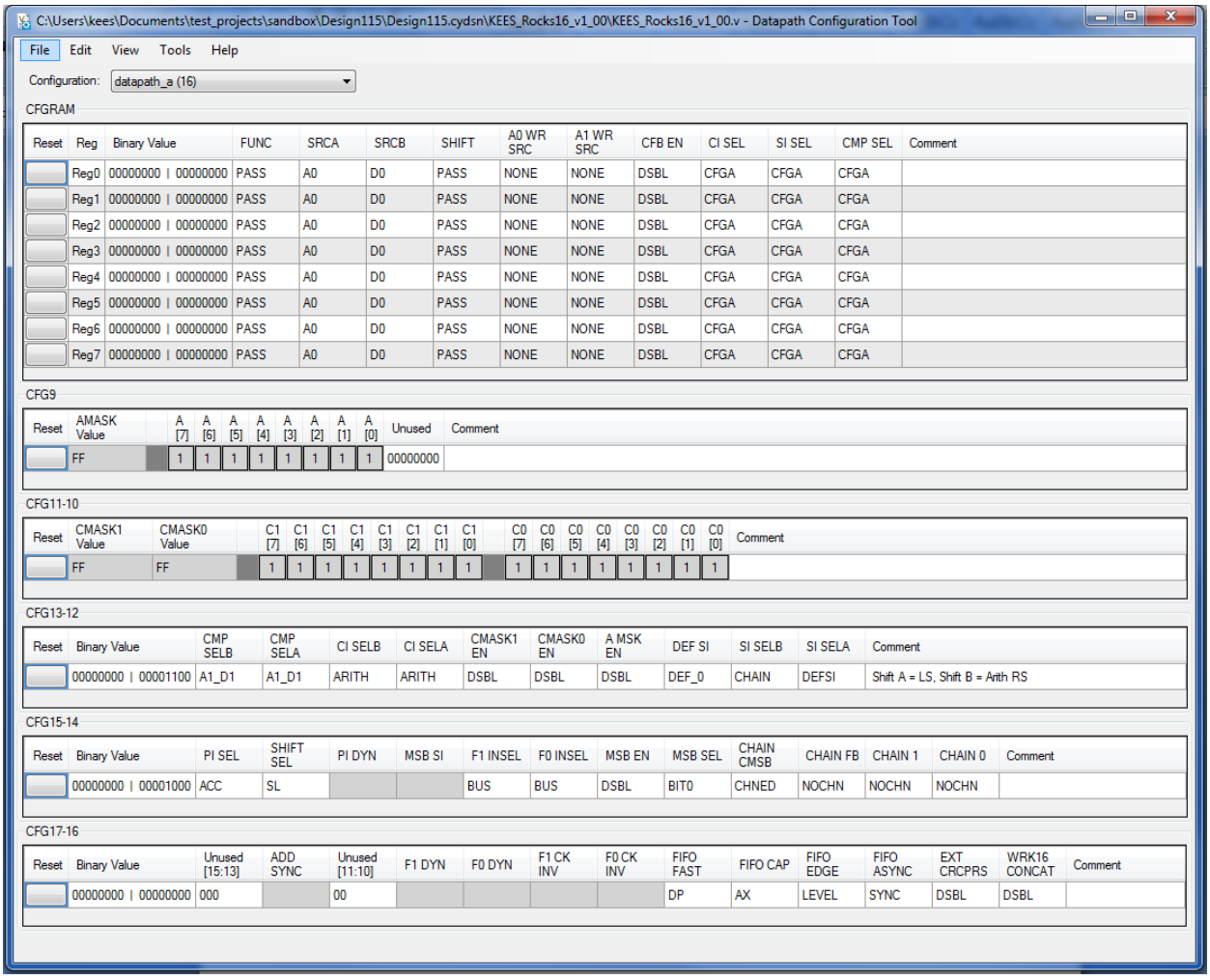
When the tool opens, select File->Open



Navigate to your project directory. In you project design folder, there will be another folder with your newly imported component name. Go into that folder, and select the verilog file.

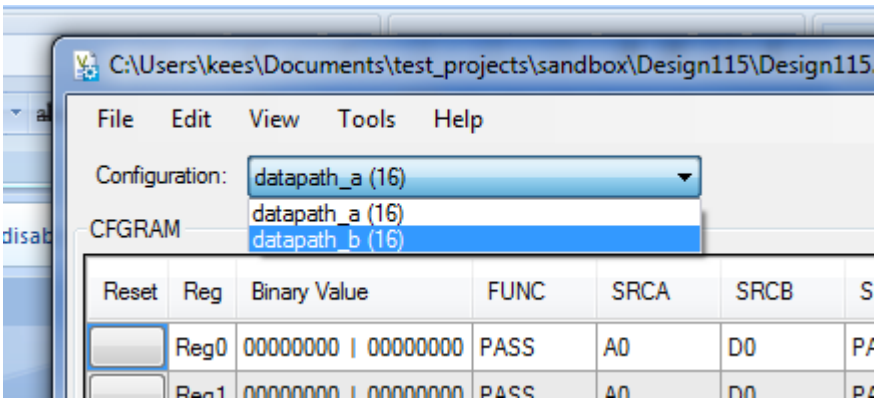


The Datapath configuration tool will update with the preconfigured datapath already in the file.



If you select a 16, 24, or 32 bit datapath, there will be a “datapath_a/b/c/d” representing the 8-15, 16-23, and 24-31st bit datapaths. In this example, I

imported a 16 bit datapath, so I have a datapath_a (LSB) and a datapth_B (MSB)



For reference, datapath_a is always the lowest byte, and the increasing letters correspond to the increasing bit order of the datapath.

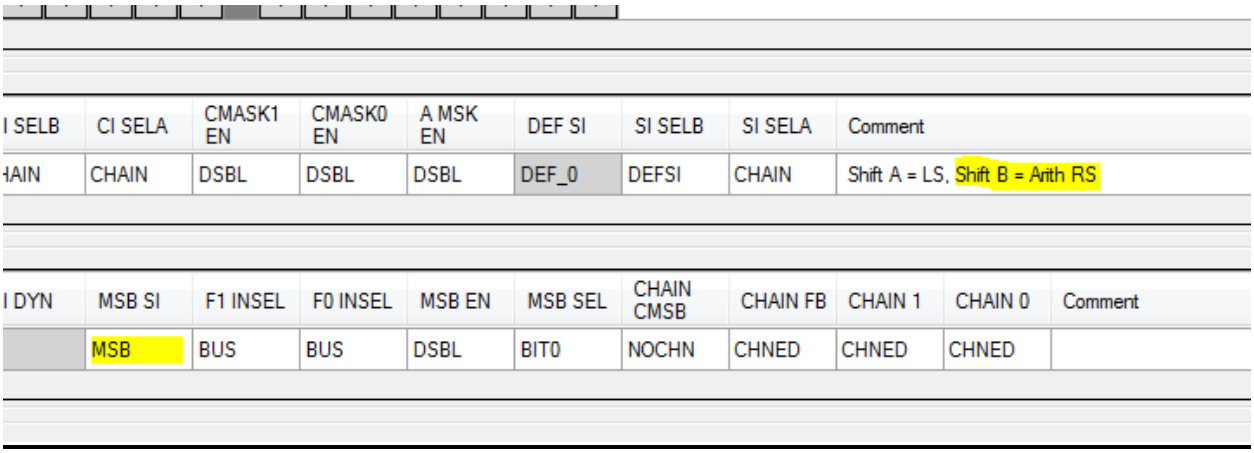
datapath_a: bits 0-7
datapath_b: bits 8-15
datapath_c: bits 16-23
datapath_d: bits 24-31

The datapaths have already been chained for you so that they “just work” for arithmetic, comparisons, and shifting. For a cheat sheet on how to chain datapaths, refer to KEES # 190.

A note on shifting: The preconfigured datapaths have been set up to perform Left Shifts (zero shifted in) when shift configuration A is selected, and ARITHmetic Right Shifts when shift configuration B is selected. I felt that these modes would be the most useful by default.

SI SELB	SI SELA	Comment
DEFSI	CHAIN	Shift A = LS, Shift B = Arith RS

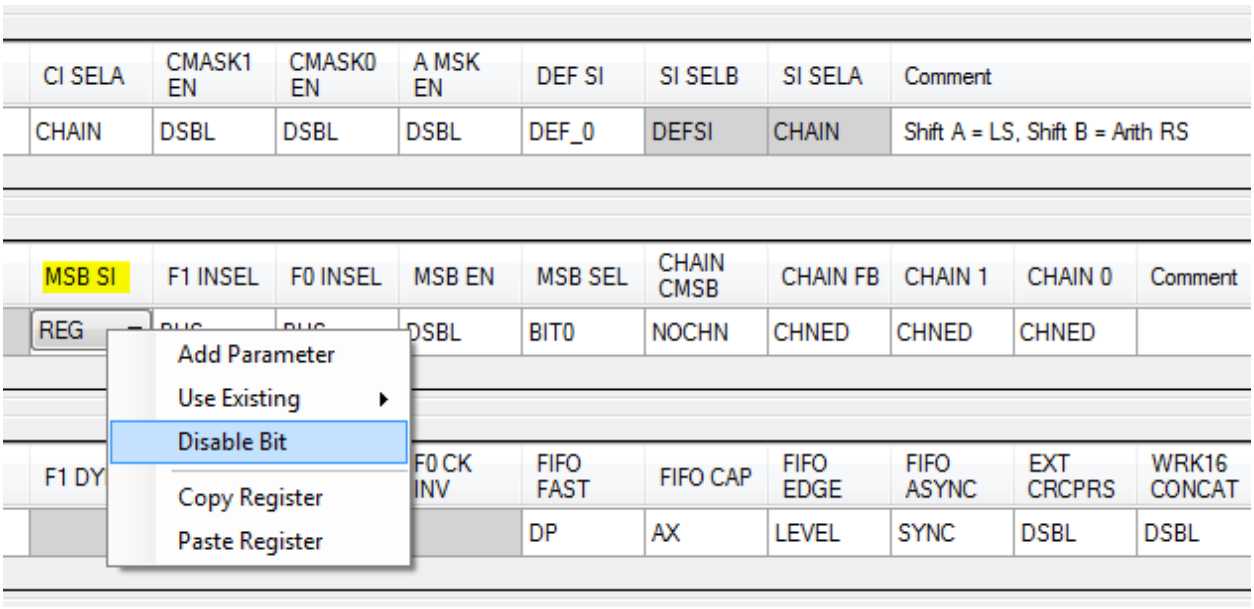
To change the behavior of the right shift from arithmetic to a normal right shift, shifting in zeros into the MSB, <u>make the following change on the most significant byte datapath</u>



I SELB	CI SELA	CMASK1 EN	CMASK0 EN	A MSK EN	DEF SI	SI SELB	SI SELA	Comment
HAIN	CHAIN	DSBL	DSBL	DSBL	DEF_0	DEFSI	CHAIN	Shift A = LS, Shift B = Arith RS

I DYN	MSB SI	F1 INSEL	F0 INSEL	MSB EN	MSB SEL	CHAIN CMSB	CHAIN FB	CHAIN 1	CHAIN 0	Comment
	MSB	BUS	BUS	DSBL	BIT0	NOCHN	CHNED	CHNED	CHNED	

Right click on the "MSB SI" bit field, and select "Disable Bit"



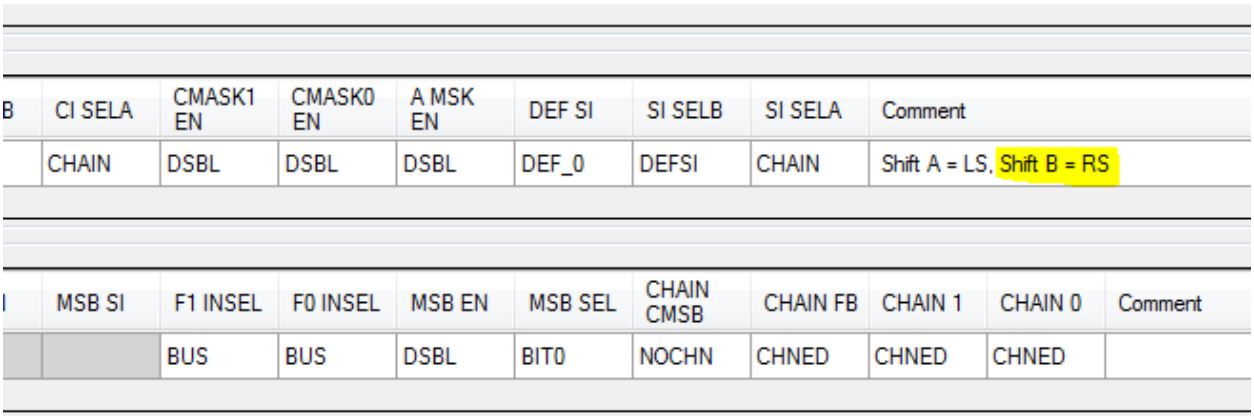
	CI SELA	CMASK1 EN	CMASK0 EN	A MSK EN	DEF SI	SI SELB	SI SELA	Comment
	CHAIN	DSBL	DSBL	DSBL	DEF_0	DEFSI	CHAIN	Shift A = LS, Shift B = Arith RS

	MSB SI	F1 INSEL	F0 INSEL	MSB EN	MSB SEL	CHAIN CMSB	CHAIN FB	CHAIN 1	CHAIN 0	Comment
	REG			DSBL	BIT0	NOCHN	CHNED	CHNED	CHNED	

Add Parameter
Use Existing
Disable Bit
Copy Register
Paste Register

	F1 DY	F0 CK INV	FIFO FAST	FIFO CAP	FIFO EDGE	FIFO ASYNC	EXT CRCPRS	WRK16 CONCAT
			DP	AX	LEVEL	SYNC	DSBL	DSBL

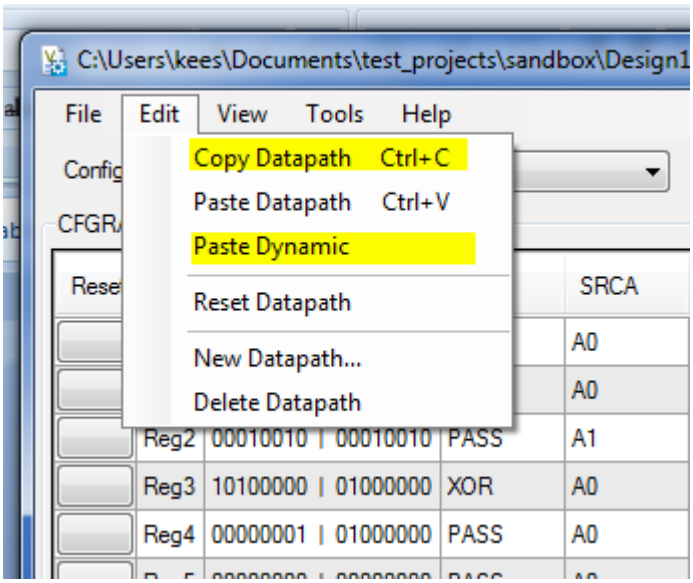
The datapath is now configured to do an non-arithmetic right shift in shift configuration B.



B	CI SELA	CMASK1 EN	CMASK0 EN	A MSK EN	DEF SI	SI SELB	SI SELA	Comment
	CHAIN	DSBL	DSBL	DSBL	DEF_0	DEFSI	CHAIN	Shift A = LS, Shift B = RS

	MSB SI	F1 INSEL	F0 INSEL	MSB EN	MSB SEL	CHAIN CMSB	CHAIN FB	CHAIN 1	CHAIN 0	Comment
		BUS	BUS	DSBL	BIT0	NOCHN	CHNED	CHNED	CHNED	

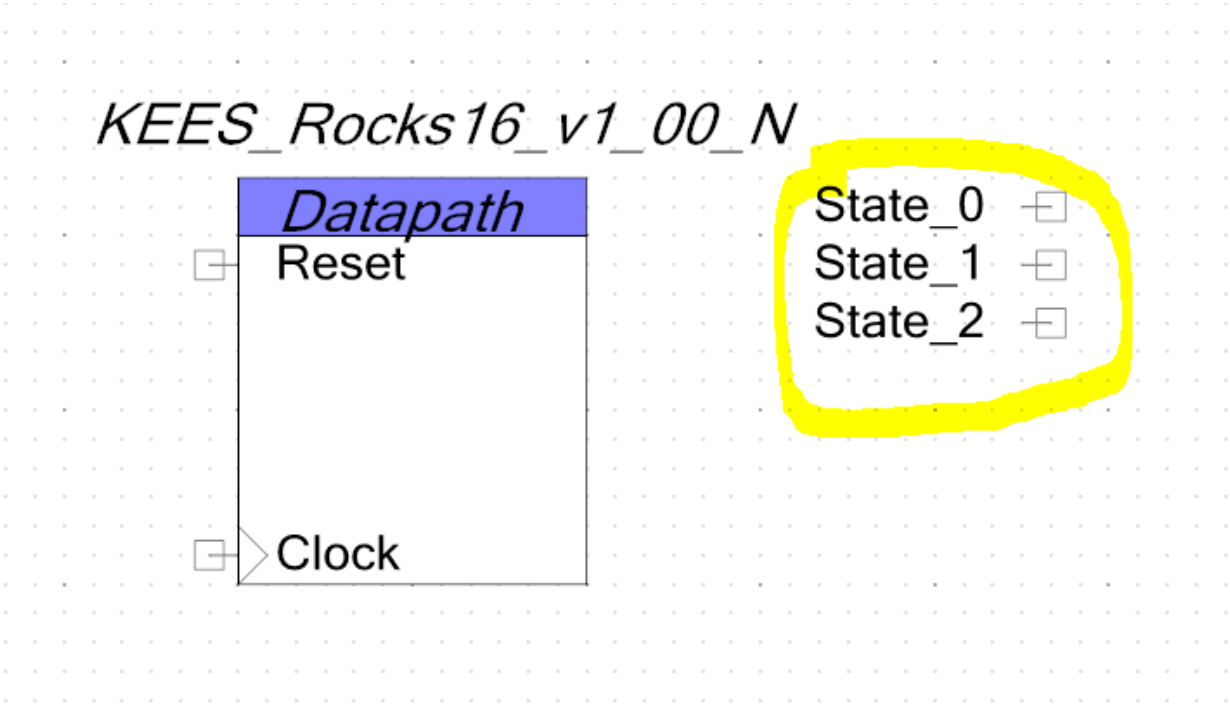
Now configure your datapath! Set up the instructions to do as you please (*make sure to configure all bytes of the datapaths the same way! You can use Edit->Copy Datapath, then move to another byte of the datapath and Edit->Paste Dynamic)*



Be wary of “Paste Datapath” since this will paste the entire datapath, messing up the chaining.

Save your datapath configuration and close the tool. Closing the tool is a hassle when you want to make quick edits to your datapath, but leaving the tool open can become problematic when you forget to save the verilog file or the datapath configuration and have 2 versions of the verilog file out of sync with each other. Take my advice, only edit one at a time, and close the datapath configuration tool when you are done using it. It will save you from many headaches.

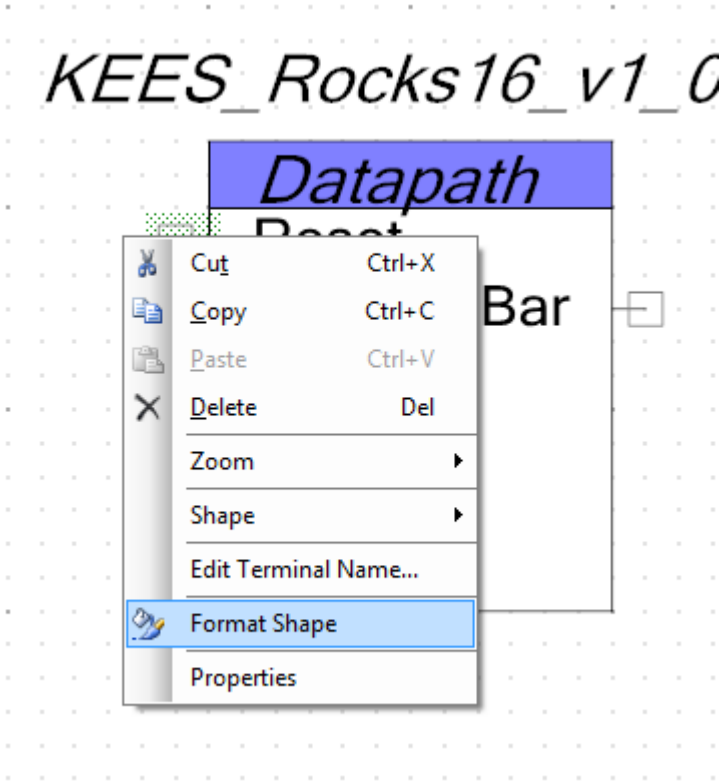
When you are done editing your datapath, you may want to add some input or outputs to your component. You can open the schematic symbol and add your inputs and outputs there. I am going to assume you know how to add inputs and outputs to the symbol so I will leave those instructions out. I will however point out the set of debugging pins included with the symbol.

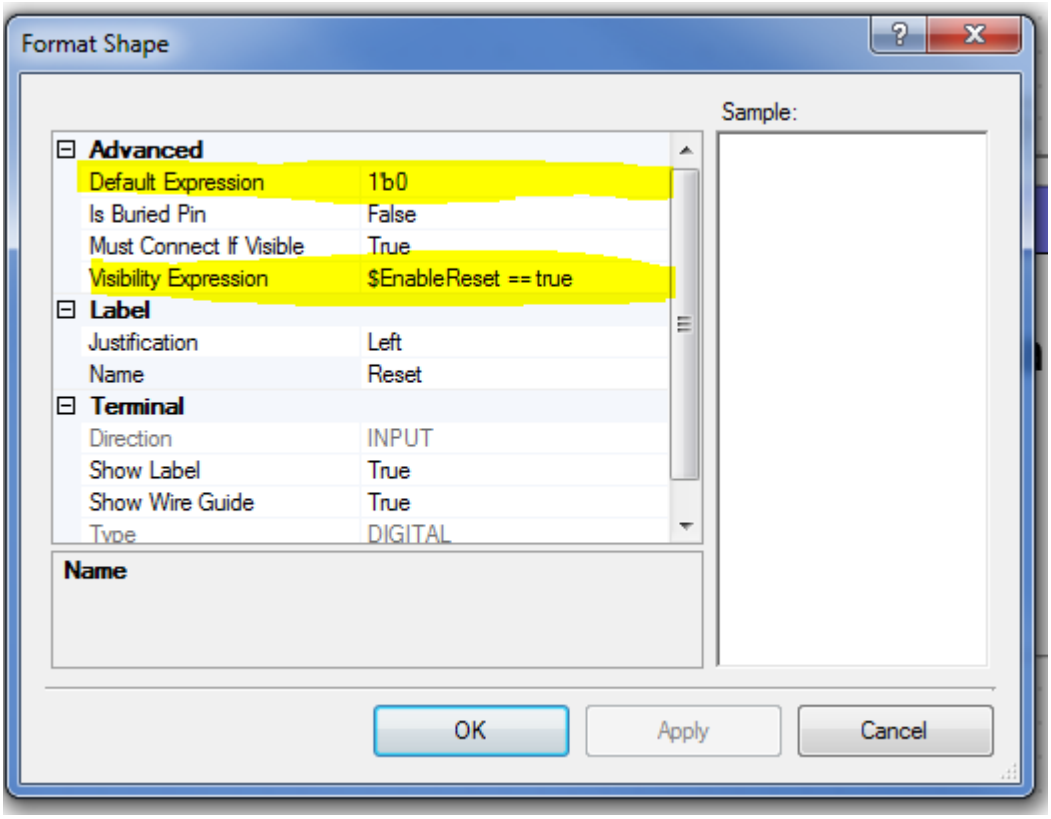


These pins are set to be invisible when the '$Debug' parameter is set to 'false'. I recommend adding useful signals to this set of debugging pins and setting them to hide when debugging is disabled for your component. When you don't need them, they are out of the way and not bothering anything. But when you do need them, they can provide critical insight into how your datapath is behaving. The easiest way to add debugging specific pins is to just copy one of the State_x pins. The shape formatting that hides the pin when debugging is disabled will be copied along with it. Just change the name to suit your needs.

A tip for adding digital inputs to your design that may be hidden based on a component parameter, make sure you set the default value along with the visibility expression.

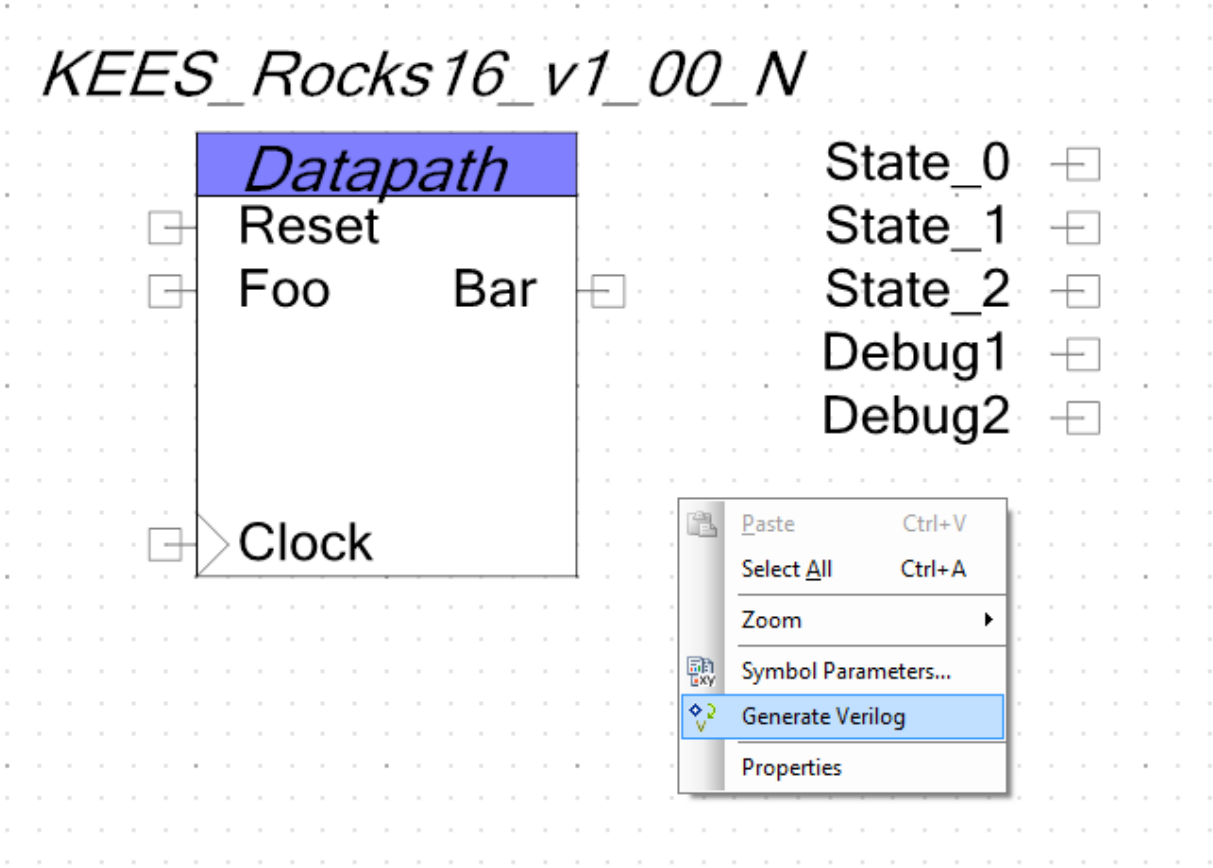
To access the visibility expression, right click on the pin and select "Format Shape"


KEES_Rocks16_v1_0
Datapath
Bar
Cut Ctrl+X
Copy Ctrl+C
Paste Ctrl+V
Delete Del
Zoom
Shape
Edit Terminal Name...
Format Shape
Properties


Format Shape
Sample:
Advanced
Default Expression 1'b0
Is Buried Pin False
Must Connect If Visible True
Visibility Expression $EnableReset == true
Label
Justification Left
Name Reset
Terminal
Direction INPUT
Show Label True
Show Wire Guide True
Type DIGITAL
Name
OK
Apply
Cancel

The default expression is the logic value that this input pin will be driven to when it's hidden.

After you have added whatever signals you desire, you can right click on the symbols empty space and select "Generate Verilog."



Click "Generate" on the dialog box that shows up, and then click "OK" when it warns that you are going to overwrite an existing file. All of the preconfigured verilog and datapath information is placed inside a merge region, meaning it will be untouched when the verilog is updated.

Before:

```
`include "cypress.v"
//`#end` -- edit above this line, do not edit this line
// Generated on 02/02/2013 at 22:49
// Component: KEES_Rocks16_v1_00
module KEES_Rocks16_v1_00 (
      output  State_0,
      output  State_1,
      output  State_2,
      input   Clock,
      input   Reset
);
```

```
        parameter A0_init_a = 0;
        parameter A0_init_b = 0;
…
```

After:

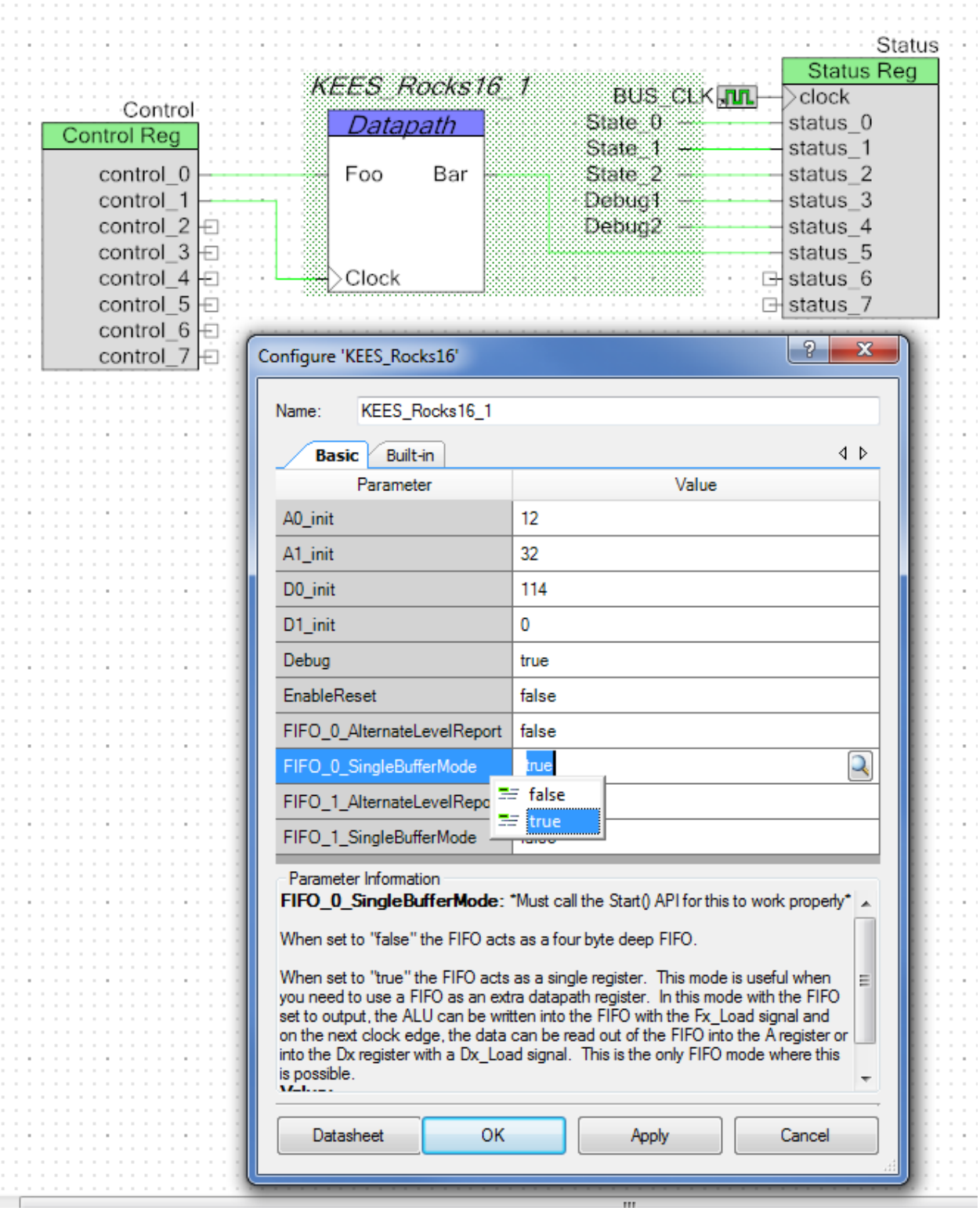
```
`include "cypress.v"
//`#end` -- edit above this line, do not edit this line
// Generated on 02/02/2013 at 22:51
// Component: KEES_Rocks16_v1_00
module KEES_Rocks16_v1_00 (
        output  Bar,
        output  Debug1,
        output  Debug2,
        output  State_0,
        output  State_1,
        output  State_2,
        input   Clock,
        input   Foo,
        input   Reset
);
        parameter A0_init_a = 0;
        parameter A0_init_b = 0;
…
```

You can use the same technique to add new parameters, or remove parameters you don't want.

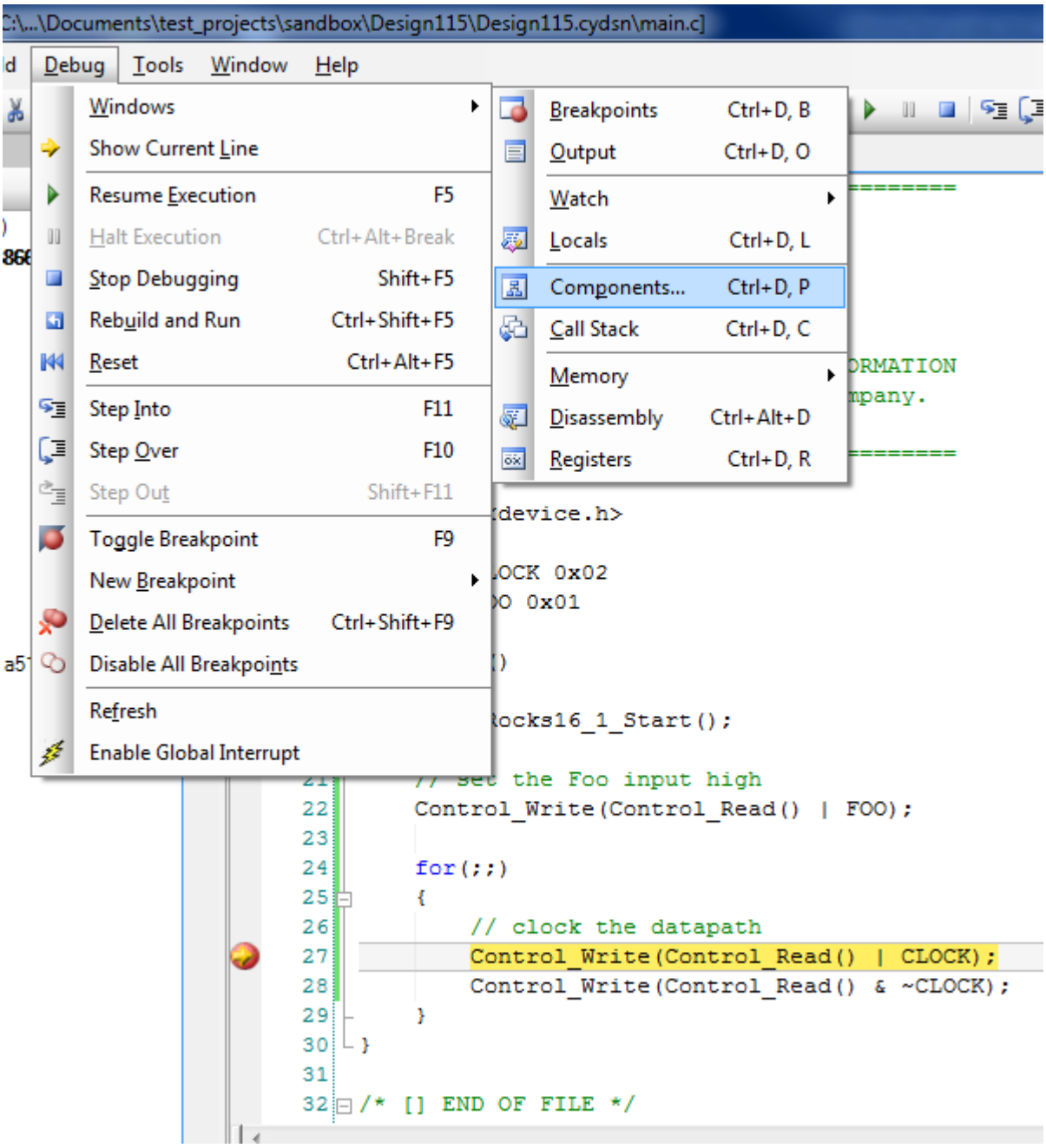
From here, take a quick look at the skeleton code provided in the verilog file. It has parameters passed in from the customizer to initialize the A0, A1, D0 and D1 registers, a 3 bit register called "State" that controls the state machine as well as provides the address for the datapath instruction. The skeleton state machine is set up with a synchronous reset (if used) as well as 8 empty states for you to write your verilog. Once your verilog is written, it's off to the debugger!

Component Debug features

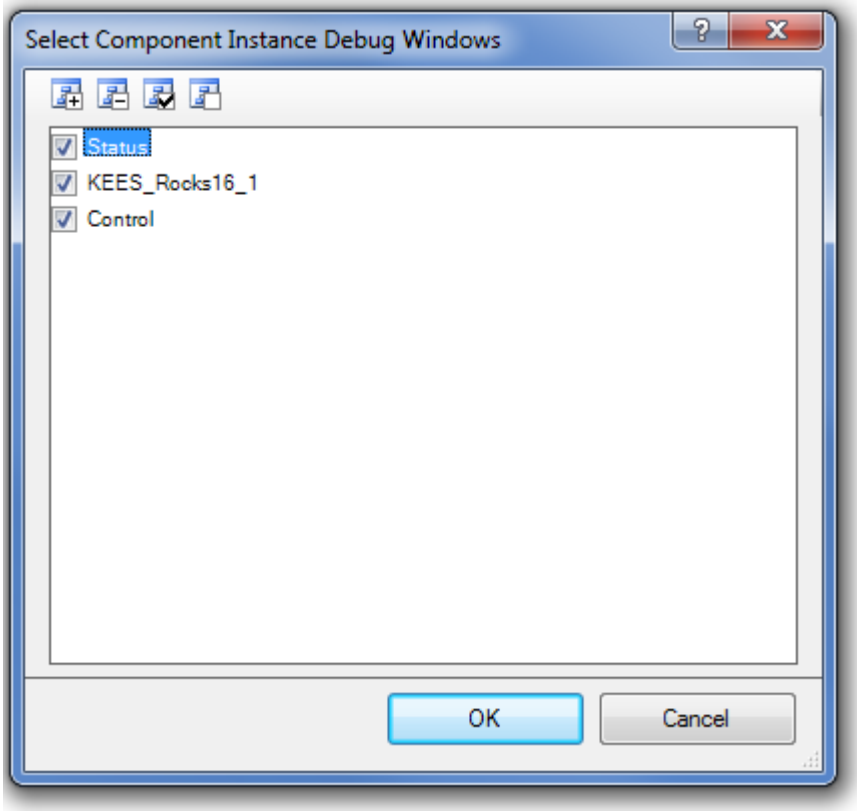
Add one of your components to your design schematic, and add a Start() API call in main. Hook up your necessary signals, configure the parameters offered in the customizer and start a debugging session.



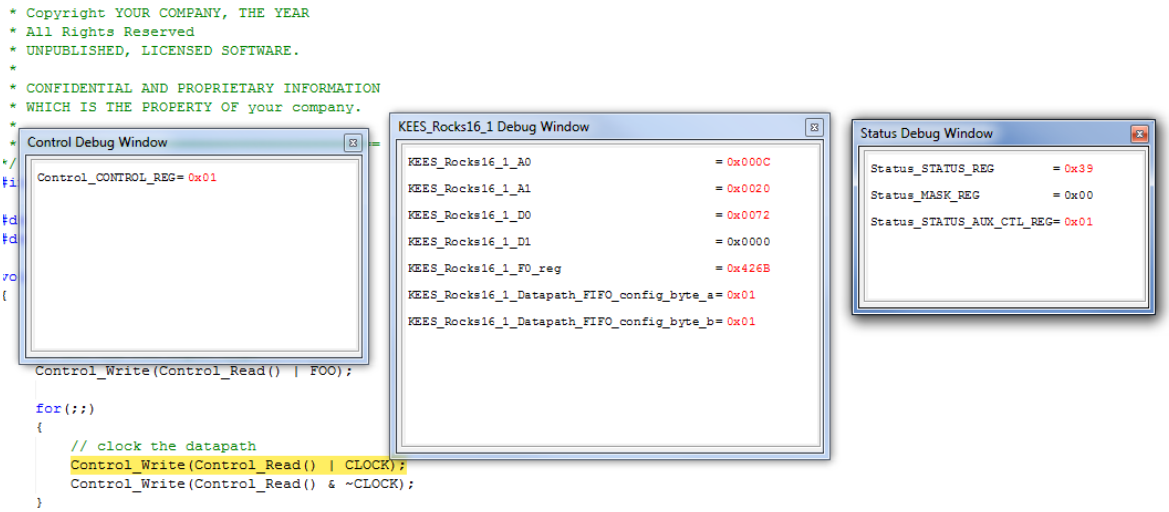
When the debugger starts, go to Debug->Windows->Components



This will bring up the component debug selection window. Check the box next to your component and click OK. I also have a control register and a status register attached to my component, allowing me to stimulate the inputs and check the outputs right in the debugger.



A new window in the debugger will appear filled with all of the registers associated with your datapath.



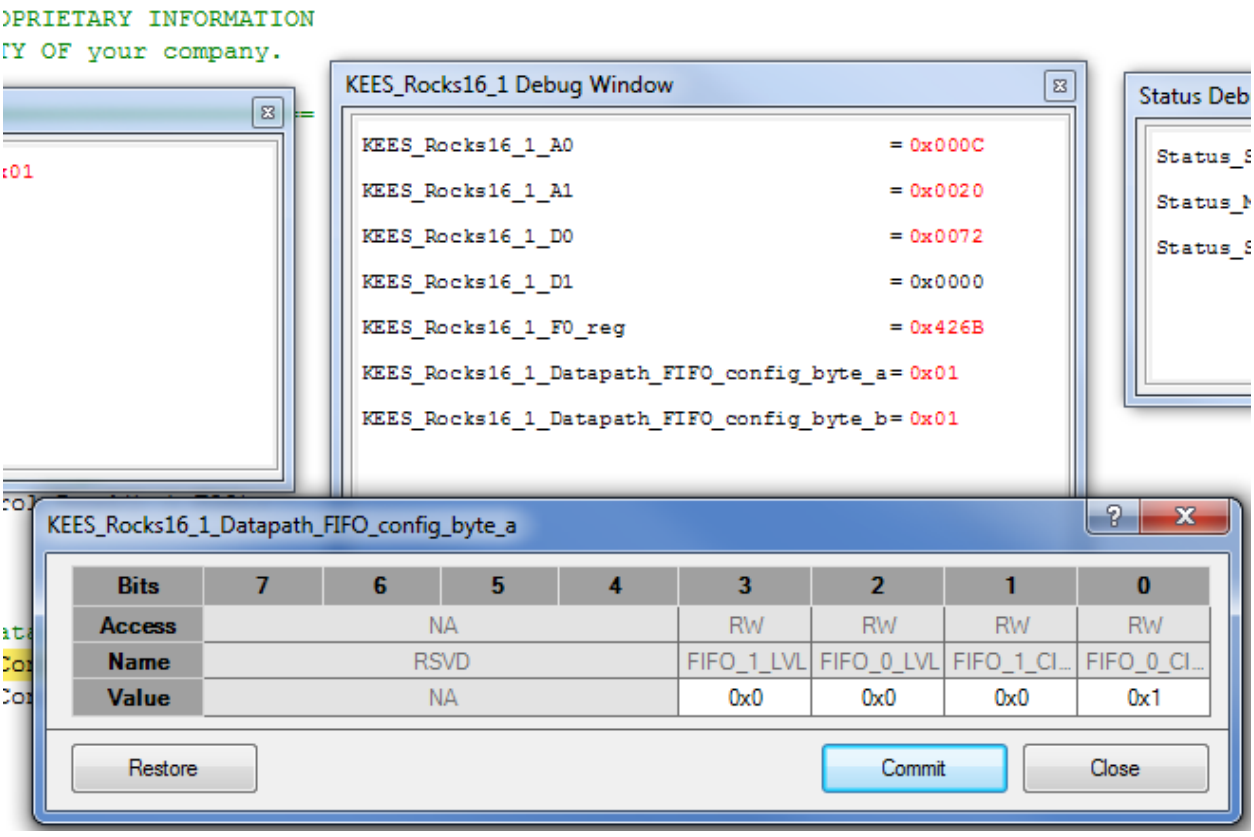
Note! There a few bugs in Creator 2.2 related to the component debugger.

1. ***When the window is displayed, all the values will be zero. You have to refresh the debugger to see the current contents of the memory. Select Debug->Refresh to update the window. Sadly, it seems that you need to refresh the window any time you wish to see a change.***

Hitting a breakpoint, stepping code or anything else that should cause it to update automatically does not seem to work.

2. ***The debugger will forget that you have a component debug window open if you stop and restart the debugger. For now, you have to re-enable the window every time you start a new debugging session.***
3. ***A radix change will not show up (left click on register, then right click and select "Radix") unless you refresh the debugger. You must also have a register selected when you right click, or you will get an unhandled exception. If this happens, click "continue" as it does not seem to cause the debugger to crash.***
4. ***When changing the radix for one register, all registers will change to the new radix.***
5. ***CDT 144250 was filed on all of these problems.***

Double clicking on the FIFO configuration registers will bring up detailed information on each bit field in that register, as well as the current configuration information.

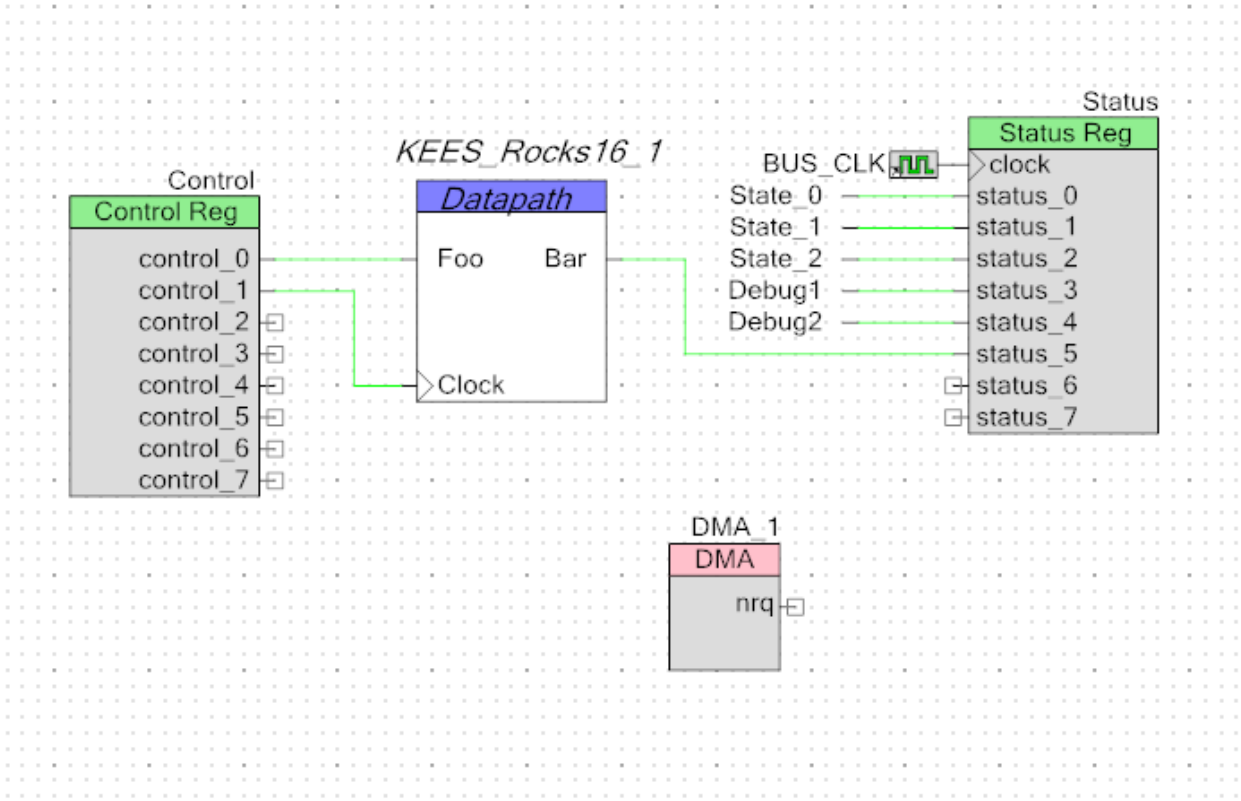


Hovering your cursor over a value will show all the options. You can modify the registers directly within these detailed views. Simply enter your new desired value and click Commit.

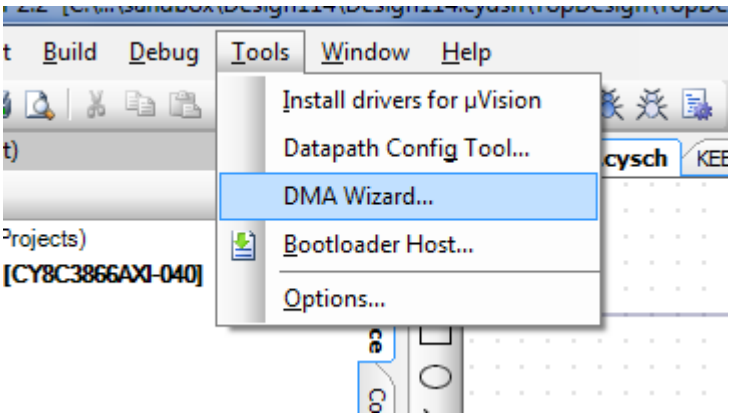
The contents of the A0, A1, D0 and D1 registers can be easily read and modified. Step through your datapath's operation! Debug your design and learn about the capabilities of the part!

DMA Wizard features

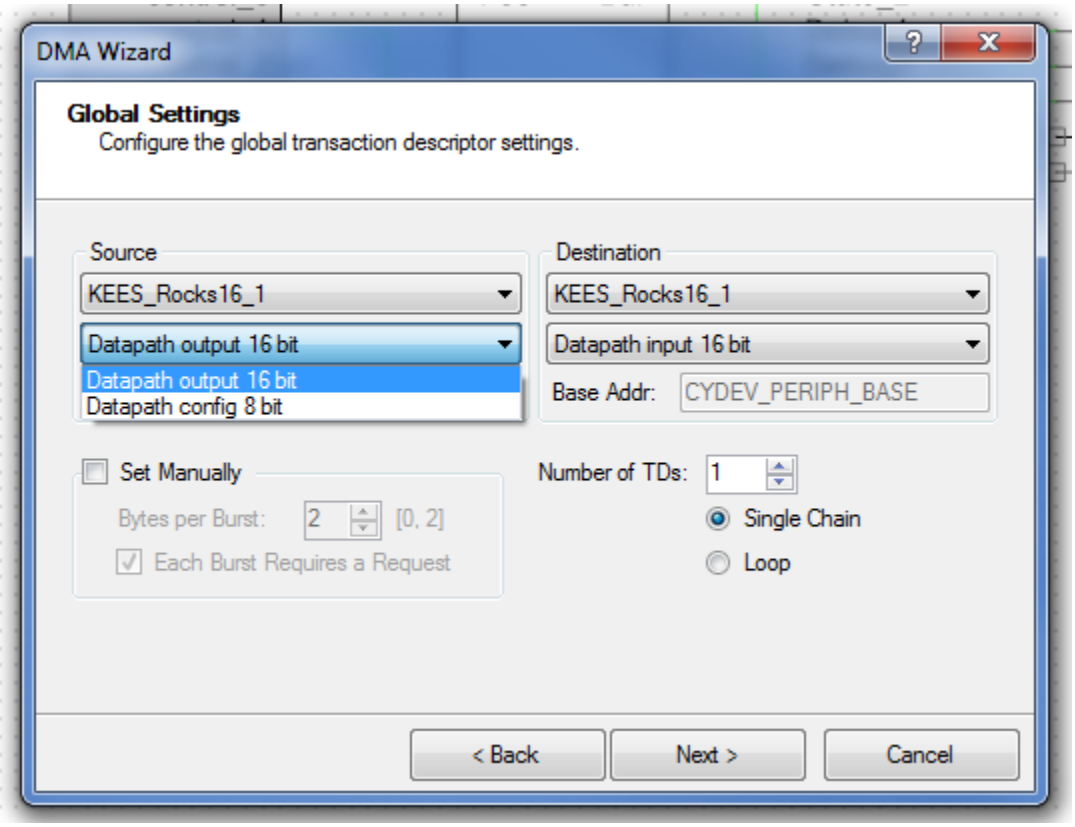
The component also comes with a DMA cpabability file, which allows the component to take advantage of the DMA wizard tool. To use the DMA capabilities, place a DMA component in the schematic.



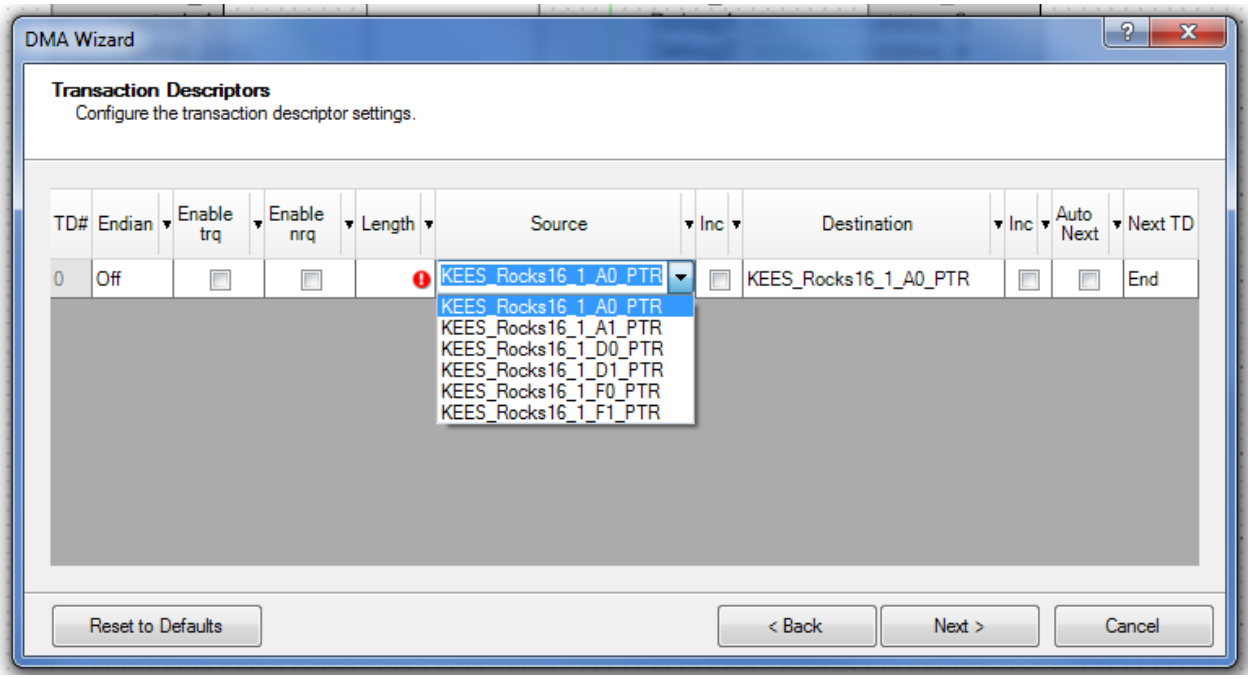
Open the DMA Wizard by selecting Tools->DMA Wizard



The DMA wizard will have register access to all the registers that the component has. There are multiple categories including input and output registers (A0, A1, D0, D1, F0, F1) as well as the FIFO configuration registers for each datapath.



The registers can be either sources or destinations. Under each category, there are pointers to all the associated registers in the selected category.



If you are curious about the DMA capability file, look at the **.cydmacap** file. For more information on how to modify this file to fit the final needs of your component, take a look at KEES # 189.

.c and .h API files

The included .c and .h API files provide all the register, masks, modes and shift definitions for configuring the component in any way you see fit. The .h file has all the REG and PTR definitions at the top, followed by sets of masks, modes and shifts for each register grouped by register and functionality.

The .c file includes a stub start function that configures the registers not configured by the datapath configuration tool. Using this component as a starting point, you can create ANY component you desire from the datapaths. Simply modify the component to suit your needs.

Hands down best reference for learning about and configuring the datapaths is the UDB BROS (Spec # 001-08005).

Happy Creating!