



ROCCC 2.0 User's Manual - Revision 0.7.6

June 4, 2013

Contents

1	Changes	6
1.1	Revision 0.7.6 Bug fixes	6
1.2	Revision 0.7.6 Added features	6
1.3	Revision 0.7.4 Bug fixes	6
1.4	Revision 0.7 Added Features	6
2	Installation	7
3	GUI	9
3.1	Installing The Plugin	9
3.2	Preparing the GUI for using ROCCC	9
3.3	GUI Menu Overview	10
3.3.1	ROCCC Menu	10
3.3.2	ROCCC Toolbar	12
3.3.3	ROCCC Context Menu	12
3.4	Loading the Example Files	13
3.5	IP Cores View	13
3.6	Creating a New ROCCC Project	14
3.7	Build to Hardware	14
3.8	Compiler Optimizations	18
3.8.1	System Specific Optimizations	19
3.8.2	Optimizations for both Systems and Modules	19
3.8.3	Low Level Optimizations	20
3.9	Add IPCores	20
3.10	Create New Module	21
3.11	Create New System	21
3.12	Import Module	22
3.13	Import System	22
3.14	Intrinsics Manager	23
3.15	Open "roccc-library.h"	23
3.16	Reset Compiler	23
3.17	Testbench Generation	23
3.18	Platform Generation	24
3.19	Updating	25
4	C Code Construction	26
4.1	General Code Guidelines	26
4.1.1	Limitations	26
4.2	Module Code	26
4.3	System Code	28
4.3.1	Windows and Generated Addresses	28
4.3.2	N-dimensional arrays	29
4.3.3	Feedback detection	29
4.3.4	Summation reduction	30
4.4	Instantiating Modules	30
4.4.1	Inlining Modules	30
4.5	Control Flow	31
4.6	Look Up Tables	31
4.7	Composed System Code	34
4.8	Legacy Code	35

4.8.1	Legacy Module Code	35
4.8.2	Legacy System Code	35
4.9	Hardware Specific Optimizations	36
4.9.1	Specifying Bit Width	36
4.9.2	Systolic Array Generation	37
4.9.3	Temporal Common Subexpression Elimination	37
4.9.4	Arithmetic Balancing	37
4.9.5	Copy Reduction	38
4.9.6	Fanout Tree Generation	38
4.9.7	Smart Buffers	39
5	Interfacing With Generated Hardware	40
5.1	Port Descriptions	40
5.1.1	Default Ports	40
5.1.2	Input And Output Ports	41
5.2	Interfacing Protocols	42
5.2.1	Input Registers	42
5.2.2	Input Streams	42
5.2.3	Output Scalars	44
5.2.4	Output Streams	44
5.2.5	Done	45
5.2.6	Stall	45
5.3	Memory Organization	46
5.3.1	Input Streams	46
5.3.2	Output Streams	46
5.3.3	Systolic Arrays	47
5.4	Pipelining	47
5.5	Fanout Reduction	47
5.6	Intrinsics	47
6	Generated Specific Hardware Connections	52
6.1	Basic Assumptions	52
6.2	Values created by optimizations	52
7	Examples Provided	55
7.1	Module Examples	55
7.2	System Examples	55

List of Figures

1	Copying the Plugins into Eclipse	9
2	Location of the ROCCC 2.0 Preferences	10
3	The ROCCC Preferences Page	10
4	ROCCC Toolbar	12
5	ROCCC Context Menu	12
6	Importing the Examples	13
7	The ROCCCExamples Project	14
8	IP Cores View	14
9	Creating a New Project	15
10	High-Level Optimizations Page	15
11	Low-Level Optimizations Page	16
12	Basic Control of the Pipelining Phase	16
13	Advanced Control of the Pipelining Phase	17
14	Stream Accessing Management Page	18
15	Successful compilation	18
16	VHDL Subdirectory Created	19
17	Add Component Wizard	20
18	New Module Wizard	21
19	Module Skeleton Code for MACC	21
20	New System Wizard	22
21	System Skeleton Code for WithinBounds	22
22	Intrinsics Manager	23
23	Testbench Generation	24
24	Dependent Files Window	24
25	(a) Module Code in C and (b) generated hardware	27
26	(a) Using a loop in module code and (b) resulting hardware	27
27	(a) System Code in C and (b) generated hardware	28
28	Accessing a 3x3 Window	29
29	A system with a three dimensional input and output stream	29
30	(a) System Code That Contains Feedback and (b) Generated Hardware	30
31	System Code That Results in a Summation Reduction	31
32	(a) Code That Instantiates a Module, (b) the Generated Hardware, and (c) Generated Hardware After Inlining	32
33	Boolean Select Control Flow. (a) In the original C, (b) in the intermediate representation, and (c) in the generated hardware datapath.	33
34	Predicated Control Flow (A) in the original C, (B) in the intermediate representation, and (C) in the generated hardware.	33
35	Reading From A Preinitialized LUT	34
36	Multiple Reads/Single Write LUT	34
37	Composite Systems	35
38	Legacy Module Code	36
39	Declaring And Using A Twelve Bit Integer Type	36
40	C Code To Generate A Systolic Array	37
41	Block Diagram Of Max Filter System	38
42	Block Diagram Of Max Filter System After TCSE	38
43	System Code That Accesses a 3x3 Window	39
44	3x3 Smart Buffer Sliding Along a 5x5 Memory	39
45	(a) System Code that reads from a FIFO and (b) Memory fetches when using a FIFO	39
46	Timing Diagram Of A System With Both Input Scalars And Input Streams	40
47	Block Diagram Of A Generated Module	41

48	Block Diagram Of A Generated System	42
49	Timing Diagram Of Module Use	43
50	Reading From A Stream	43
51	Reading From A Stream With Multiple Channels	44
52	Timing Diagram of Output Streams	45
53	Timing Diagram Of The End Of A System's Processing	46
54	C Code For MaxFilterSystem Which Uses A 3x3 Window	46
55	C Code That Writes To Three Locations In The Same Stream Each Loop Iteration	47
56	Basic Dataflow	48
57	Medium Dataflow	48
58	High fanout a) before registering and b) after registering	49
59	Generated Systolic Array Hardware	50
60	Theoretical Interface to a 32-bit Floating Point Divide IPCore	51
61	Wrapper for the Theoretical 32-bit Floating Point Divide	51
62	System Code Sections Translated Into Hardware	52
63	C Code That Infers Ports	53
64	Generated Ports	54

1 Changes

1.1 Revision 0.7.6 Bug fixes

- Fixed an issue where certain window sizes would cause problems in synthesis with Xilinx ISE Virtex 6 tools.
- Fixed an issue with output smart buffers for window sizes larger than the step sizes.
- Changed internal temporary variables to be unsigned when appropriate.

1.2 Revision 0.7.6 Added features

- Added debug variables.
- Added watchpoints to debug variables.

1.3 Revision 0.7.4 Bug fixes

- Fixed an issue with inlining where certain obscure code configurations would fail, such as when you had multiple inlined modules inside of another module that contained two or more loops and the innermost module had an if.then.else chain greater than 4 levels deep.
- Fixed an issue where the GUI might freeze when compiling if the compilation generated too many warnings or errors.
- We added support for the Pico M501 platform.

1.4 Revision 0.7 Added Features

- Look up tables can now be specified and provide run time random read and write access.
- Certain aspects of the calculation of generated addresses is now pipelined deeper, resulting in faster achievable clock speeds.
- Preliminary support for composed systems is now available.
- In composed systems, redundancy for component systems is supported.
- New intrinsics are supported for handling streams in composed systems, including stream voters and stream splitters.
- Port for Convey HC-1 now available
- Port for Pico M501 now available

2 Installation

Installation and execution of ROCCC has been tested on the following systems:

- 32-bit Ubuntu ??? Linux
- 64-bit Ubuntu ??? Linux
- 32-bit CentOS ??? Linux
- 64-bit CentOS ??? Linux
- 64-bit OpenSuse Linux
- Macintosh Snow Leopard
- Macintosh Lion
- Macintosh Mountain Lion

Other systems are not currently supported.

ROCCC is distributed as a binary package and requires Eclipse 3.5.1 or higher. When uncompressed, the ROCCC distribution folder should have the following directories:

- Documentation
The location of this user manual and any additional manuals.
- Examples
A directory to be imported into the Eclipse framework that contains all of the example code.
- GUI
The location of the Eclipse plugin .jar files.
- License
The default location where the ROCCC license file should be placed.
- LocalFiles
The location of one user's database and local information.
- ReferenceFiles
A directory containing the files necessary for PCore generation.
- SupplementaryVHDL A directory that contains VHDL that is necessary for some ROCCC generated code to run.
- bin
A directory that contains the precompiled binaries for a given platform.
- lib
A directory that contains the precompiled libraries for a given platform.
- scripts
A directory that contains a script for canceling compilation and a script for creating a subordinate installation.
- tmp
This directory is used for temporary storage when compiling with ROCCC.

There may be additional directories depending on if the installation contains a port of ROCCC to a specific machine.

Included in the GUI directory of the distribution are Eclipse plugins that control access to all of the ROCCC functionality. The user is responsible for moving the files into the appropriate eclipse plugin directory on the target system and removing any previously installed ROCCC plugins that may exist.

If you experience any failures in the installation procedure, consult the troubleshooting section at the end of this document.

3 GUI

The ROCCC GUI is a plugin designed for the Eclipse IDE that works on both Linux and Mac systems. The user must have at least Eclipse version 3.5.1 installed. ROCCC currently supports the C++ and Java versions of Eclipse. Eclipse can be downloaded for free at www.eclipse.org.

The ROCCC GUI plugin is continually evolving and may function slightly differently in future releases.

3.1 Installing The Plugin

Once you have downloaded and uncompressed Eclipse, open the resulting uncompressed eclipse folder. Inside of there, you should see a folder named plugins. This is where we need to copy the ROCCC GUI plugins into as shown in Figure 1. Any previous versions of the ROCCC plugins must also be removed from this directory. The ROCCC plugins are located inside the GUI folder of the uncompressed ROCCC distribution folder.

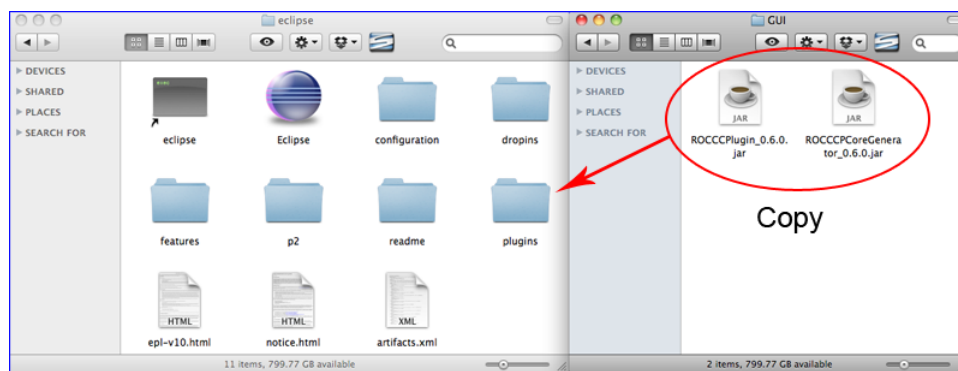


Figure 1: Copying the Plugins into Eclipse

Once you have moved the ROCCC plugins into the plugins folders inside eclipse, ROCCC should be ready to run on Eclipse. The first time you run Eclipse with the ROCCC plugins installed, ROCCC will set up the perspective best used for working with ROCCC. It will also open up a page welcoming you to ROCCC 2.0 and asking if you would like to register for updates and news.

3.2 Preparing the GUI for using ROCCC

Before we can use the core functionality that is bundled with the GUI, the user must first set the directory path to the ROCCC distribution folder. This can be done by selecting "Preferences" in the ROCCC menu tab at the top of the program as in Figure 2.

Once this is done, a preference page will pop up asking for the ROCCC distribution path. Set the preference value to wherever you had uncompressed the ROCCC distribution folder. The validity of the chosen folder can be checked by clicking the "Verify ROCCC Distribution Folder" button on the preference page as shown in Figure 3.

On this page you must also set the path to the license file in order to use ROCCC. If the license is not set you will not be able to compile or use any of the ROCCC functionality.

Once that is done, the ROCCC GUI should be ready to use. If you ever try to use any of the ROCCC functionality and this preference is not set or that directory is incorrect, the GUI will tell you and ask if you want to set the ROCCC distribution folder in the Preference menu.

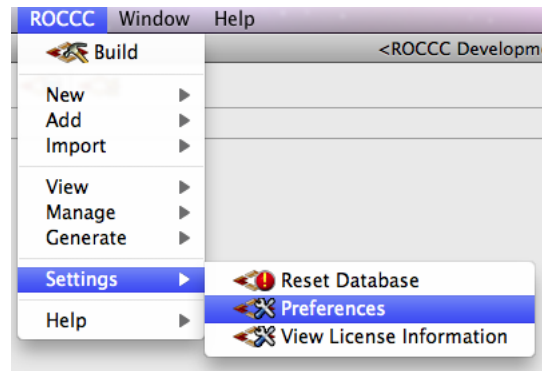


Figure 2: Location of the ROCCC 2.0 Preferences

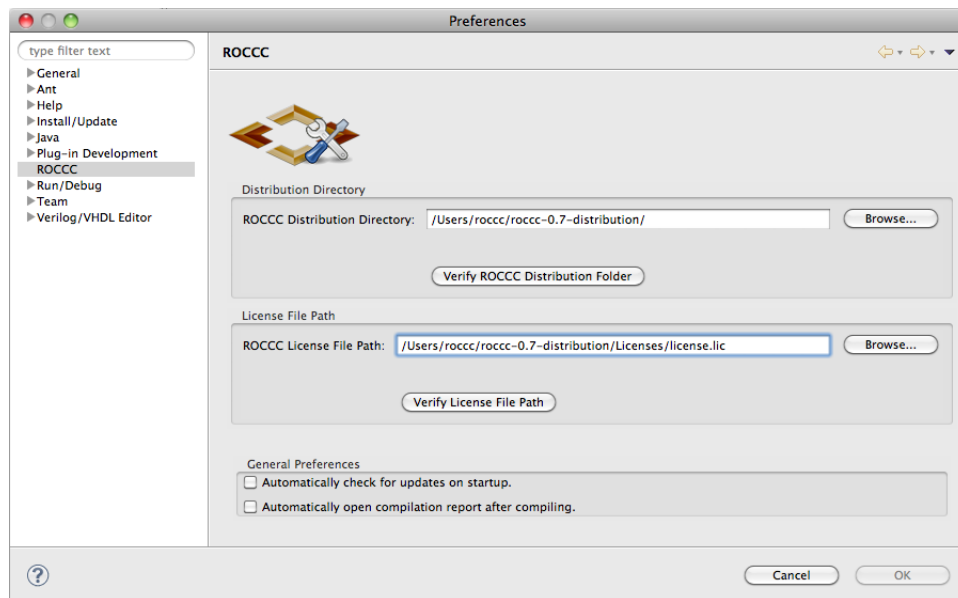



Figure 3: The ROCCC Preferences Page


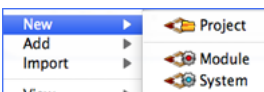
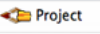
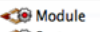
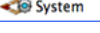
3.3 GUI Menu Overview

This is a quick overview of all the ROCCC buttons and options located on the GUI for future reference. Each of the actions the buttons do will be covered in more detail in the other sections, this is merely so you can see and recognize all the buttons available.

Note: The icons on the menus may not show up if your system preferences are set to not show Menu Icons.

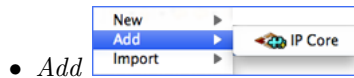
3.3.1 ROCCC Menu

-  **Build:** Compile the open modules or system file.

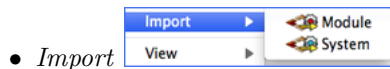
-  
 -  **Project**
 -  **Module**
 -  **System**

- **Project:** Create a new ROCCC project in Eclipse.

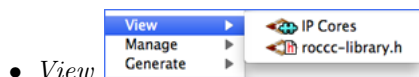
- **Module:** Create starter code for a new ROCCC module.
- **System:** Create starter code for a new ROCCC system.



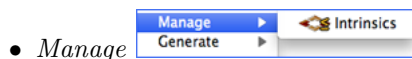
- **IP Core:** Add an IP Core directly to the database for future use.



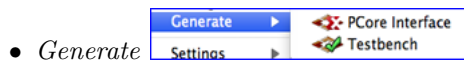
- **Module:** Import an outside ROCCC module C file into a project.
- **System:** Import an outside ROCCC system C file into a project.



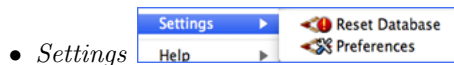
- **IP Cores:** Opens the IP Cores view to see available cores in the ROCCC database.
- **roccc-library.h:** Open the roccc-library.h file in the default editor.



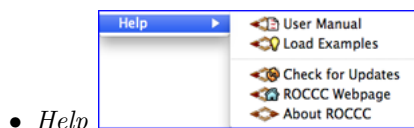
- **Intrinsics:** Open the intrinsic window to add, edit, or delete intrinsics.



- **PCore Interface:** Generate a PCore for a ROCCC module.
- **Testbench:** Generate a hardware testbench file for a ROCCC component.



- **Reset Database:** Reset the database back to its installation configuration.
- **Preferences:** Open the preference page to manage preferences.



- **User Manual:** Opens the ROCCC user manual.
- **Load Examples:** Loads the ROCCC examples in an Eclipse project.
- **Check for Updates:** Check if a new version of ROCCC is available.
- **ROCCC Webpage:** Opens the ROCCC webpage.
- **About ROCCC:** View which version of ROCCC you are using.



Figure 4: ROCCC Toolbar

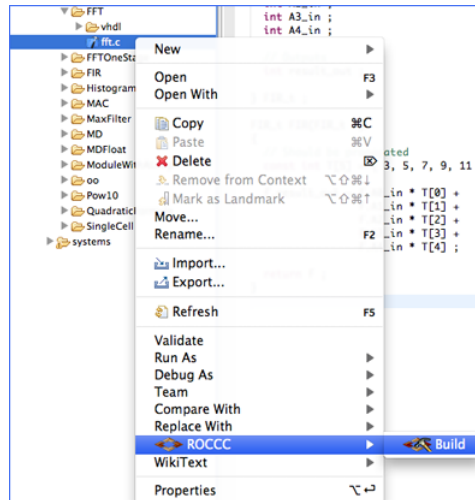


Figure 5: ROCCC Context Menu

3.3.2 ROCCC Toolbar

- **Build:** Compile the open ROCCC module or system file.
- **Cancel:** Stops the current compilation if any are running.
- **New Module:** Create the starter code for a new ROCCC module and add it to a project.
- **New System:** Create the starter code for a new ROCCC system and add it to a project.
- **Manage Intrinsic:** Open the intrinsic management window to add, edit, or delete intrinsics.

3.3.3 ROCCC Context Menu

- **Build:** Compile the open module or system file and run it through the ROCCC compiler.

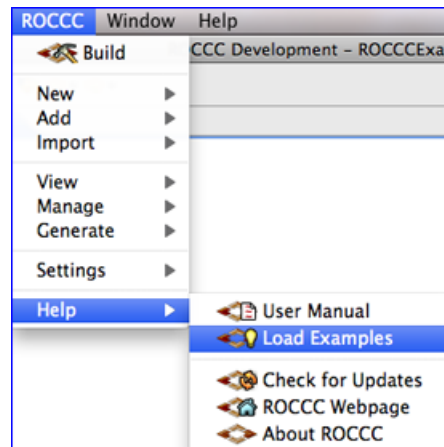


Figure 6: Importing the Examples

3.4 Loading the Example Files

To test ROCCC out on the example files, you need to load the examples that came bundled with the distribution. The first way to do this is after setting the distribution folder for the first time, ROCCC will ask if you would like the examples loaded. Selecting "Yes" will have the ROCCC examples loaded into a new project called "ROCCCExamples." If there is already a project with that name, ROCCC will ask you for a different name for a project to create and import the examples into. If there is an internet connection available, ROCCC will also open the examples webpage to give explanations of how the examples work.

The second way to load the examples, which can be done at any time after the distribution folder has been set, is to do it through the ROCCC menu. Select 'ROCCC → Help → Load Examples and the ROCCC examples will be loaded as mentioned above. This is shown in Figure 6.

Once that is complete, the examples should be loaded into the project that was created. If you look into the projects sub directories, you should see a src folder. Within that folder there should be modules, and systems folders as shown in Figure 7.

The GUI requires ROCCC projects to be arranged according to this directory structure. Any code located in the modules subdirectory is assumed to be module code, and similarly any code in the systems directory is assumed to be systems code.

3.5 IP Cores View

ROCCC maintains its own database of compiled modules that can be viewed at anytime. To view the contents of the database, click ROCCC → View → IPCores on the Eclipse menu. The ROCCC IPCores view will open and display all the inserted modules inside the database.

You can view what ports are on a specific module in the database by selecting a component in the IPCores view. The neighboring table will then display all the port names, directions, port sizes, and types for that selected component. You can delete a compiled component from the database by clicking the component name in the IPCores View and pressing the Delete key. The component will also be removed from the roccc-library.h file.

You can also use any of the components in the ROCCC database by having a valid module or system open and selected, move the cursor to where you want to insert a call to a module, and double click the desired component in the IPCores view. This will add a function call to the double clicked component in the open ROCCC file and will add `#include roccc-library.h` to the top of the file. All that you will have to do after that is put which variables you wish to pass into the desired component function call.

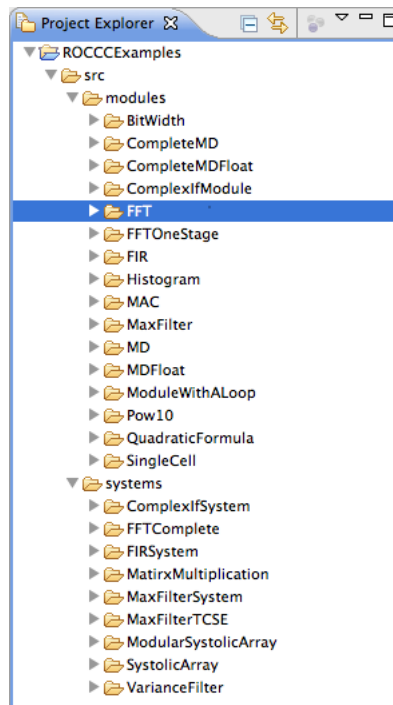


Figure 7: The ROCCExamples Project

Component Name	Latency	Num Ports	Port Name	Direction	Size	Data Type
Histogram	4	8	num1_in	IN	64	int
Location	36	7	num2_in	IN	64	int
LocationWithAngle	46	7	sum_out	OUT	64	int
MAX	2	3				
MD	10	12				
MDFloat	97	12				
MAX	2	4				
PassThrough	2	2				

Figure 8: IP Cores View

3.6 Creating a New ROCCC Project

To start using ROCCC with your own code from scratch, you first need to create a new project. To create a new project, select 'ROCCC → New → Project' as shown in Figure 9.

A window will pop up asking for the name of the new project to make. Type in the desired name of the project and press "Ok." Once that is done a new project will show up in the project explorer with the name you chose. From there, to add new modules or systems you either import them from already made files or create new ones from scratch. To import premade modules or systems into the project, use the Import → Module and Import → System under the ROCCC menu. To create new modules or systems to be added to the project, use the New → Module and New → System under the ROCCC menu.

3.7 Build to Hardware

Once a ROCCC module or system is ready to be compiled into VHDL code, you want to use the Build command. To do this, open the desired module or system inside the Eclipse editor and select the Build command in the ROCCC menu or ROCCC toolbar. After that is selected, a window will open up asking for which high-level compiler optimizations to use as in Figure 10.

The build window consists of several pages which control different levels of compiler optimizations. The user may select finish at any time and any pages not modified will use the default values. The default values

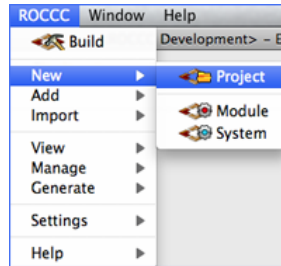


Figure 9: Creating a New Project

may also be set on a page by page basis by selecting the "Save Values As New Defaults" button.

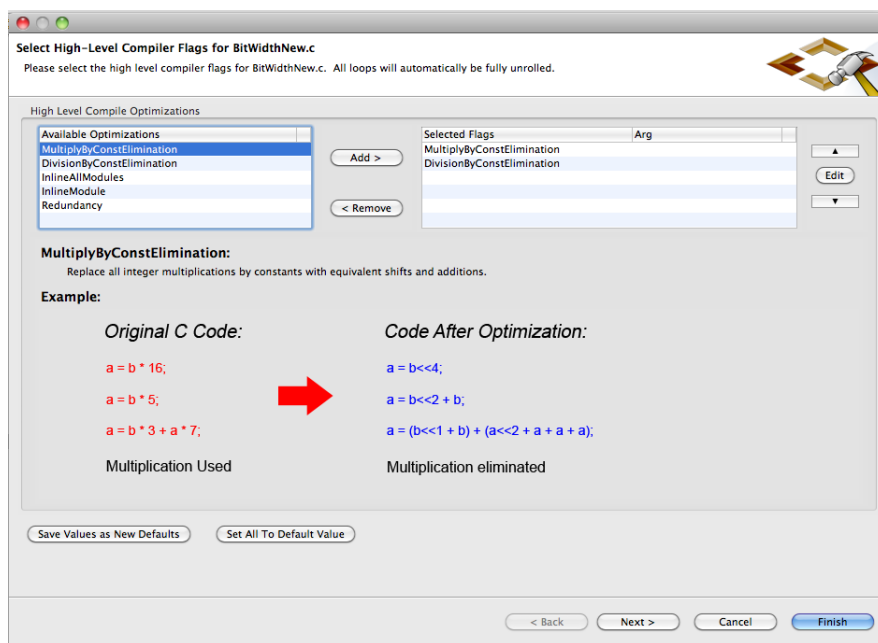


Figure 10: High-Level Optimizations Page

On the first page you can select which high level compiler optimizations to perform. Depending on whether you are compiling a module or a system, you will see a different list of available optimizations to choose.

The second page available when compiling asks for which low-level compiler optimizations to use as in Figure 11. These flags are the same regardless of compiling a module or system.

The third optimization page available controls the extent of pipelining in the generated hardware. As shown in Figure 12, the pipelining may be controlled with a slider that adjusts the generated pipeline from fully pipelined on the left to fully compacted on the right. When fully pipelined, every operation will be placed into a separate pipeline stage, resulting in the largest area but fastest clock. When fully compacted, the compiler will attempt to put every operation into one pipeline stage, resulting in the slowest clock speed but smaller area. When fully compacting code, instantiated modules will retain their delay.

However, not all operations take the same amount of time to execute. To naively have the compiler arbitrarily pack operations together without considering how expensive an operation is would give inconsistent results across different components. Because of this, ROCCC allows you to specify weight values for each basic operation in the advanced mode as shown in Figure 13. A larger weight means that operation is more

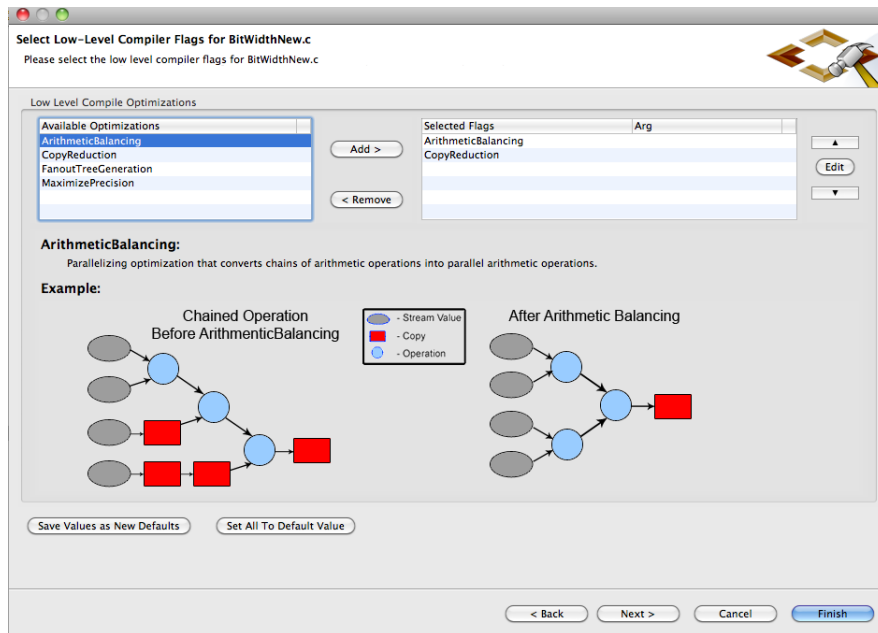


Figure 11: Low-Level Optimizations Page

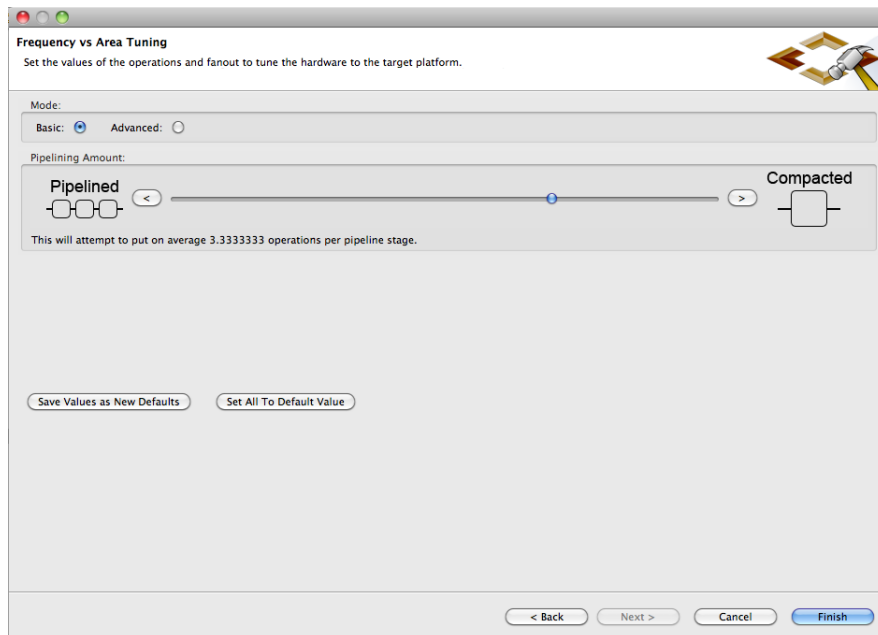


Figure 12: Basic Control of the Pipelining Phase

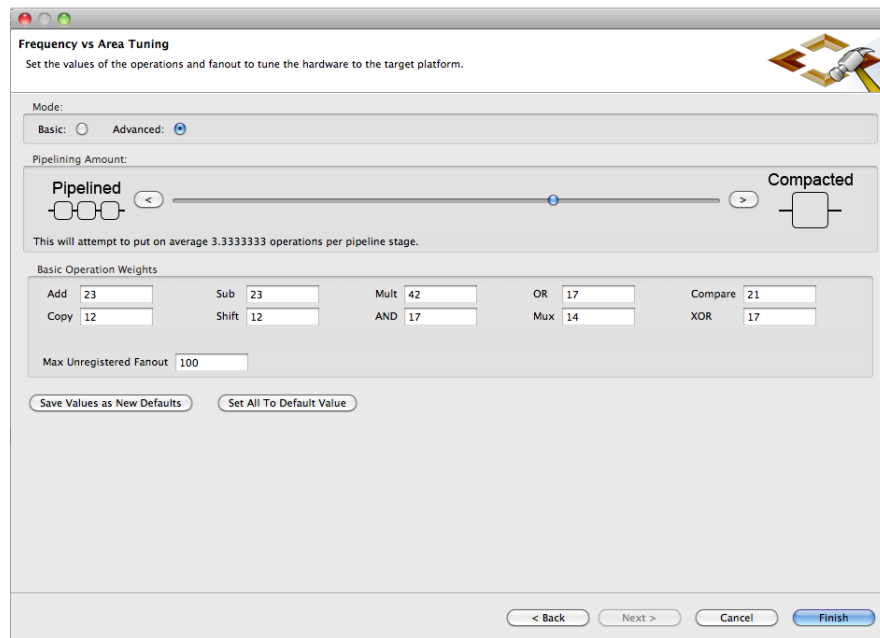


Figure 13: Advanced Control of the Pipelining Phase

expensive in terms of execution time on the desired platform. To edit these values, click the advanced tab at the top of the Area vs Frequency page.

These weight values have no real absolute meaning, they only have meaning relative to each other. For example, if our Mult operation takes twice as long as our Add, we need to make sure we make the weight value for Mult is twice that of Add. This can be done as (100 and 50) or (50 and 25), it doesn't really matter as long as the weights are proportional to each other. In this case when compaction occurs, the compiler would attempt to allow two chained additions to happen together for every multiplication that is done.

If all the weights have the same value, that means that they all take the same amount of execution time. Again, this can be achieved by having the weights as all 1s or even all 500s, as long as they are all the same value. The default weights that were distributed with ROCCC are the values we came up with for targeting 150 MHz on an LX-330. These weights combined with the pipeline slider gives you precise control over how to tune your component in terms of area and frequency.

Also available in the advanced view is control over the maximum allowable fanout. When generating a circuit, if any register has a fanout larger than the specified number, registers are inserted along the paths in order to ease routing constraints.

If compiling a system there is a fourth page in the compilation wizard for managing the ways streams are accessed as shown in Figure 14. From here you can select "Add" to add managing info for either input or output streams. From here, a page will open asking for the stream name, the number of parallel data channels, and the number of parallel address channels. Once pressing "Finish" the values will be added to the stream management page in the corresponding table you pressed "Add" for.

Once these values are in the table, you can edit these values by double clicking individual cells and changing the values. The number of stream channels must be a factor of the window the data is being accessed from for that stream and the step size of the loop.

Once you have selected which optimizations to use and have set the arguments for the optimizations that require them, select Finish. This will run the ROCCC toolchain on the selected open file inside the Eclipse editor. All output from the compilation will be outputted on the console inside of Eclipse as shown in Figure 15.

Additionally, a summary report will open in the main window to report statistics of the compilation.

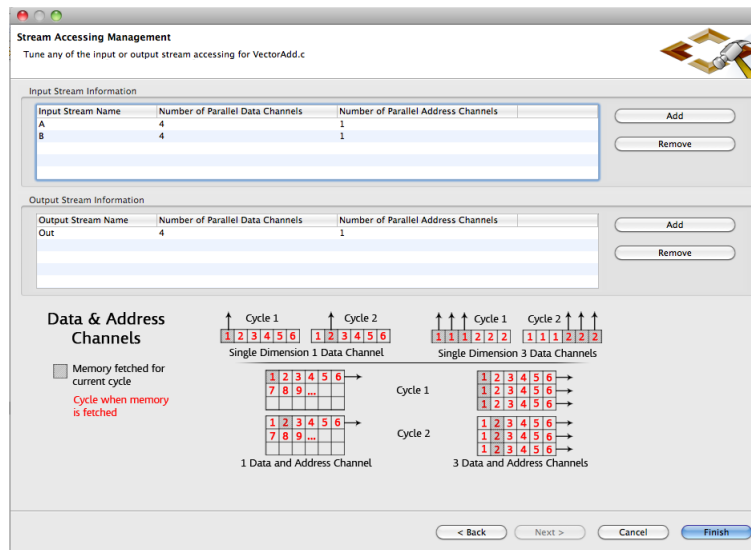


Figure 14: Stream Accessing Management Page

The automatic opening of this report can be controlled in the ROCCC Preferences panel.

Figure 15: Successful compilation

If the compilation finished successfully, you will see a VHDL folder in the project directory next to the file you compiled that will have the generated VHDL code for that system or module as shown in Figure 16

The selected flags for each file are saved so that if you go to recompile a file multiple times, it will load which flags were used during the previous compile.

The other way to compile a file is to right-click the desired file in the Project Navigator and select Build to Hardware in the ROCCC submenu as shown in Figure 5.

3.8 Compiler Optimizations

In addition to standard compiler optimizations such as dead code elimination and constant propagation, when compiling ROCCC code, the first page of the build window will allow the user to select additional high level optimizations to perform on the code. The choice of optimizations is different depending on if the compiled code is a module or system. Note: When compiling a module, all loops are fully unrolled automatically.

The available optimizations are:

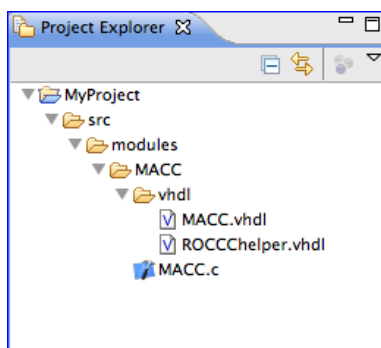


Figure 16: VHDL Subdirectory Created

3.8.1 System Specific Optimizations

- **Systolic Array Generation:** Transform a wavefront algorithm that works over a 2-dimensional array into a one-dimensional hardware structure with feedback at every stage in order to increase the throughput while reducing hardware.

Note: This optimization cannot be combined with other optimizations.

- **Temporal Common Sub Expression Elimination:** Detection and removal of common code across loop iterations to reduce the size of the generated hardware.
- **LoopFusion:** Merge successive loops with the same bounds and no dependencies.
- **LoopInterchange:** Switch the loop induction variables of two nested loops.
- **Loop Unrolling:** Unroll the loop at the given C label by a specified amount. If the loop has constant bounds, the loop can be fully unrolled.

Arguments:

Loop Label - The loop specified by the C label in the code.

Number of times to unroll - The number of times to unroll the loop. If the loop has constant bounds, you can set the value to FULLY to fully unroll the loop. If a system has all of its loops completely unrolled, it will be transformed and compiled as a module.

- **FullyUnroll:** Fully unroll all loops in the original C code. If any of the loops have variable bounds, this pass will stop compilation.

3.8.2 Optimizations for both Systems and Modules

- **MultiplyByConstElimination:** Replace all integer multiplications by constants with equivalent shifts and additions.
- **DivisionByConstElimination:** Replace all integer divisions by constants with equivalent shifts and adds.
- **Redundancy:** Enable dual or triple redundancy for a module at a given C label.
- **InlineModule:** Inline C code of specified modules as opposed to instantiating black boxes.
- **InlineAllModules:** Inline C code of all module instantiations, and if those contain any other calls, continue inlining up to the specified depth.

3.8.3 Low Level Optimizations

- **ArithmeticBalancing:** Change long chains of sequential arithmetic into parallel trees when applicable.
- **CopyReduction:** Rebalance the low-level data flow graph to minimize the number of registers and decrease the size of the generated hardware.
- **FanoutTreeGeneration:** Create a tree when the fanout of a register is greater than the threshold. This will slightly increase the latency and size of the generated circuit in order to tradeoff the large fanout.
- **MaximizePrecision:** Temporary arithmetic results are truncated at every step by default, but this optimization will increase the size of the temporary arithmetic results and only perform rounding as necessary in assignments.

3.9 Add IP Cores

When working on a ROCCC project, you may want to integrate some hardware modules that you have access to outside of ROCCC. Using this component would require you to insert the already created component into the ROCCC database so the compiler can incorporate it as well as using it in future compilations. To do this, select Add → IP Core in the ROCCC menu. A window will pop up asking for the details of the component as shown in Figure 17.

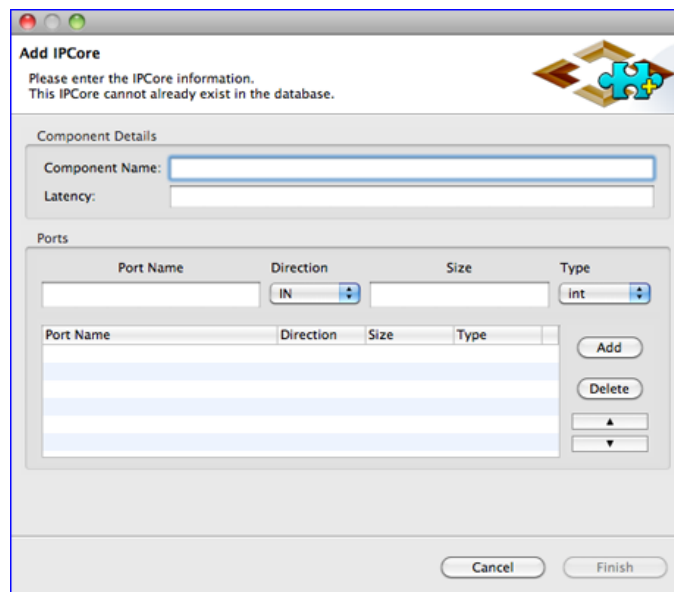


Figure 17: Add Component Wizard

First, specify the name and latency of the component. Next, you need to add all of the ports for the added component. You need to specify at least one input port and one output port before you can click Finish. If you need to edit one of the already added ports, simply double click on the field you wish to edit and you will be able to change the value of that field. Once everything is added correctly, click Finish and the component will be added to the ROCCC database. The component will now also be found in the IP Cores view.

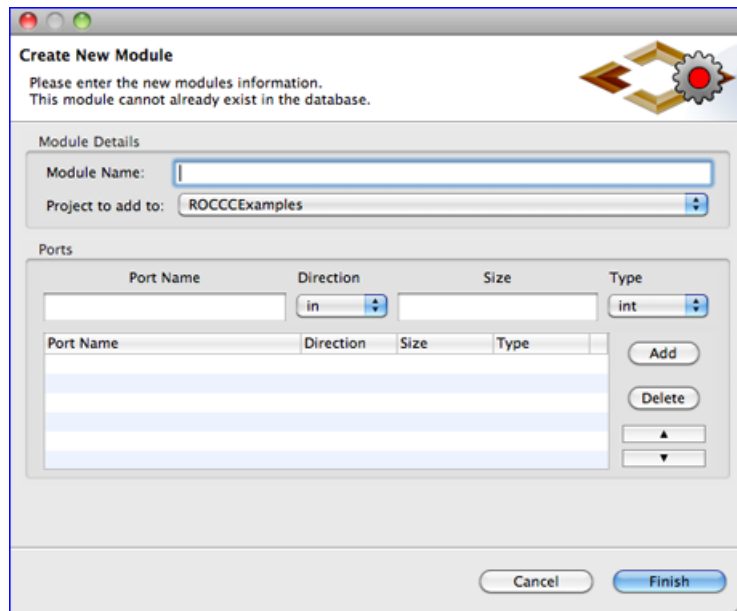


Figure 18: New Module Wizard

3.10 Create New Module

To start a new module from scratch, first make sure you have a valid ROCCC project loaded or have created a new project as described in the Creating a new Project section. Once you have a valid project open, select New → Module under the ROCCC menu or toolbar to begin creating the new module. A new window will open asking for the details of the new module as shown in Figure 18.

Input the name of the module and which project to add the new module to. Next add all the ports that this module will have. If you ever need to edit an already added port, simply double click the field you wish to edit and you will be able to change the value of that field. Once everything is added correctly, click Finish and the module will be added to the project. The new file will open in the editor with the necessary starter code to begin coding the module as shown in Figure 19.

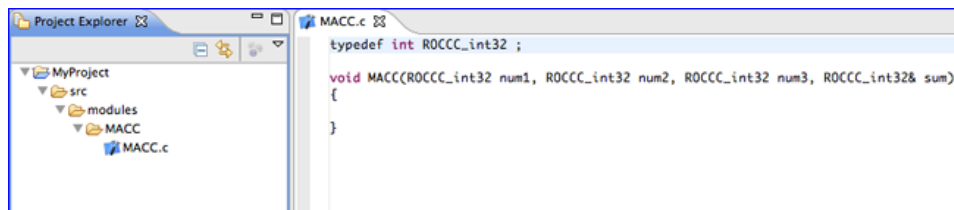


Figure 19: Module Skeleton Code for MACC

3.11 Create New System

To start a new system from scratch, first make sure you have a valid ROCCC project loaded or have created a new project as described in the Creating a new Project section. Once you have a valid project open, select New → System under the ROCCC menu or toolbar to begin creating the new system. A new window will open asking for the details of the new system.

Input the name of the system and which project to add the new system to. Lastly, select how many

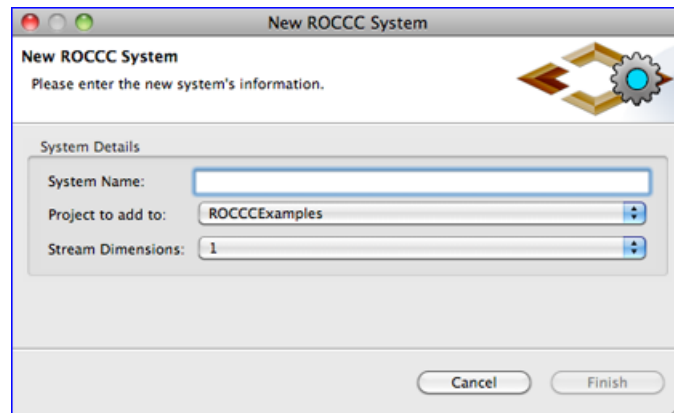


Figure 20: New System Wizard

stream dimensions the system will have. Once everything is added correctly, click Finish and the system will be added to the project. The new file will open in the editor with the necessary starter code to begin coding the system as shown in Figure 21.

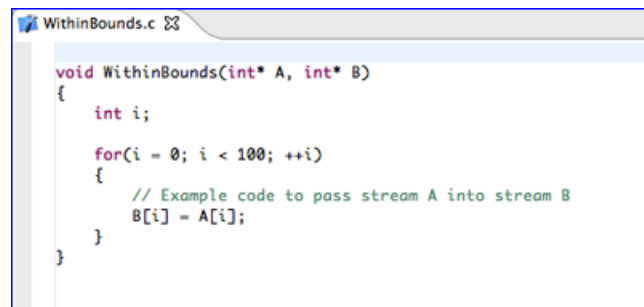


Figure 21: System Skeleton Code for WithinBounds

3.12 Import Module

If you are looking to add an already done ROCCC module C file to the current project you are working on, you can use the Import Module command. To do this, first have a valid project opened to import the module into. Next, click Import → Module under the ROCCC menu. This will open up a window asking for the file to import.

First, browse for the desired ROCCC module file to import. Secondly, type the name of the module you are importing. Lastly, select which project to import the module into. Once finished, click the Finish button at the bottom and the selected module will be imported into the project and will show up in the Project Navigator view. This does not add the module to the database, this solely adds the module C code to the project.

3.13 Import System

If you are looking to add an already done ROCCC system C file to the current project you are working on, you can use the Import System command. To do this, first have a valid project opened to import the system into. Next, click Import → System under the ROCCC menu. This will open up a window asking for the file to import.

First, browse for the desired ROCCC system file to import. Secondly, type the name of the system you are importing. Lastly, select which project to import the system into. Once finished, click the Finish button at the bottom and the selected system will be imported into the project and will show up in the Project Navigator view. This does not create hardware code for the selected system, this solely adds the system C code to the projects.

3.14 Intrinsic Manager

Certain operations in C require hardware blocks on FPGA. These include floating point operations and integer division. By selecting 'Manage → Intrinsic' the user is able to select which IP cores to use. The intrinsic manager is shown in Figure 22. By adding intrinsics the user is able to select which components are inserted into generated datapaths by activating and deactivating individual intrinsics.

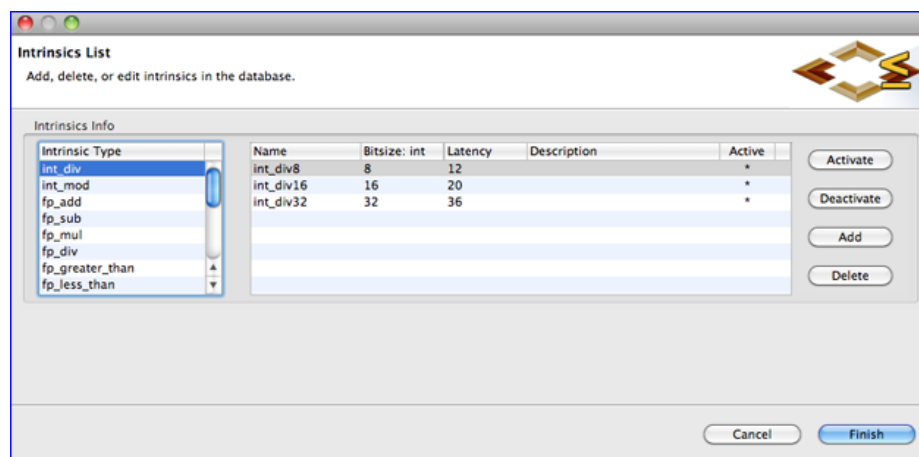


Figure 22: Intrinsic Manager

3.15 Open "roccc-library.h"

Every time a module is compiled, the interface struct and hardware function prototype are added to the roccc-library.h file. If you ever need to view the roccc-library.h file, simply select View → roccc-library.h under the ROCCC menu. This will open up the roccc-library.h file in the default editor.

3.16 Reset Compiler

To reset the ROCCC database to its distribution state, simply click Settings → Reset Database under the ROCCC menu. This will delete any added entries in the ROCCC database and will clear all added modules under the roccc-library.h file.

3.17 Testbench Generation

Once a module or system has been compiled with ROCCC and translated into hardware, you can create a hardware testbench for simulation by selecting Generate → Testbench from the ROCCC menu. For modules, you can enter as many test sets as you wish with their corresponding expected outputs as shown in Figure 23. For systems, you will need to enter values for both the input scalars as well as specify files that contain the values of each input stream and output stream. The stream files must consist of a list of values separated by white space in the order in which they will be read.

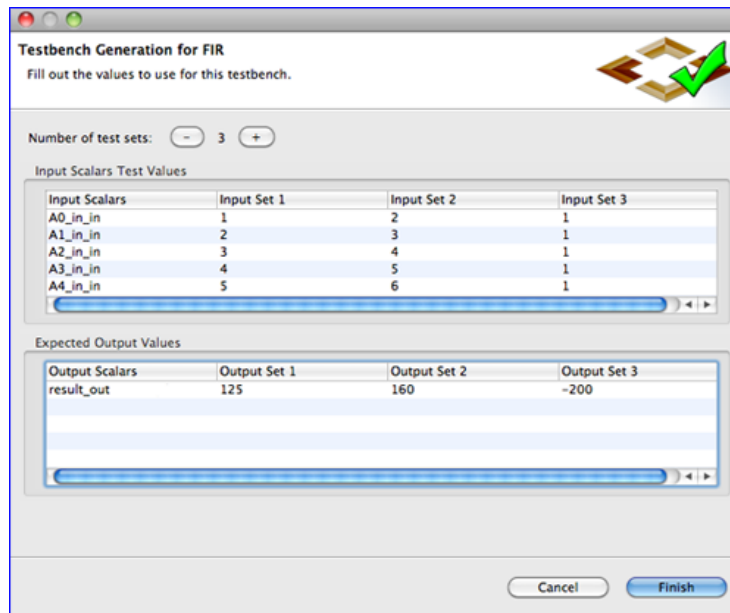


Figure 23: Testbench Generation

3.18 Platform Generation

Once a module or system has been compiled with ROCCC, you can generate a Xilinx PCore from it. You can do this by selecting 'Generate → PCore Interface in the ROCCC menu. ROCCC will then generate all the necessary files and connections to make a PCore. If your component requires any dependent files such as sub components or netlists, a window will pop up asking for those files prior to generating the PCore files. The window will show you all the required components it is looking for and ask for the necessary files for each as in Figure 24.

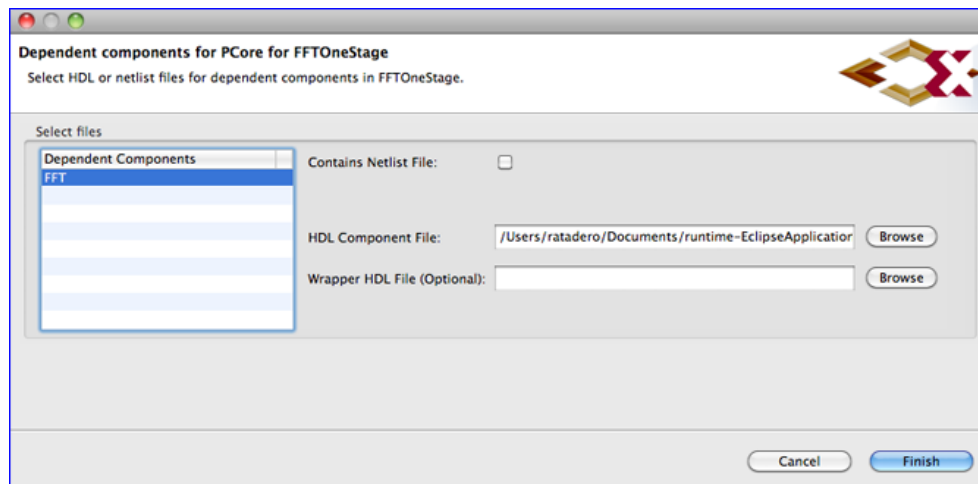


Figure 24: Dependent Files Window

You can either fill these sections out and let ROCCC handle all the moving and packaging of the files or you can continue with the generation without specifying these and place them in the packaged folder later. Once the generation of the PCore interface is complete, a folder named either "PCore" will show up next to

the ROCCC file in the project explorer.

These folders should have all the files necessary to run the PCore on the desired hardware as long as they support PCores on what you chose.

PCores support being generated on all modules but currently not on systems.

3.19 Updating

There are a few ways to keep the ROCCC toolset up to date with the most current version available. The first is by having the ROCCC GUI automatically check for updates each time on startup. You can change whether or not you want ROCCC checking for updates at startup in the preference page as in Figure 3. The other way to check for updates is to manually check for updates by selecting 'Help → Check for Updates' in the ROCCC menu.

In both of these cases, ROCCC will check to see if there is a new version of the compiler and if there is a new version of the GUI plugins. All messaging about checking for updates will show up in the Eclipse console. When there is a new version available you will be informed. Updates are available for download on the jacquardcomputing.com website.

4 C Code Construction

4.1 General Code Guidelines

ROCCC supports two styles of C programs, which we refer to as *modules* and *systems*. Modules represent concrete hardware implementations of purely computational functions. Modules can be constructed using instantiations of other modules in order to create larger components that describe a specific architecture.

System code performs repeated computation on streams of data. System code consists of loops that iterate over arrays. System code may or may not instantiate modules. System code represents the topmost perspective and generates hardware that interfaces to memory systems.

4.1.1 Limitations

ROCCC is not designed to compile entire applications into hardware and has certain general restrictions on both module and system code. ROCCC is continually in development, so these restrictions may fluctuate or be eliminated entirely in future releases. ROCCC 2.0 currently does not support:

- Logical operators that perform short circuit evaluation. The "&" and "|" operators do work and should be used in place of "&&" and "||"
- Generic pointers
- Non-component functions, including C-library calls
- Shifting by a variable amount
- Non-for loops
- The ternary operator (?:)
- Stream accesses other than those based on a constant offset from loop induction variables

4.2 Module Code

Module code represents a hardware building block to be used in larger applications. Modules are computational datapaths and are written as computational functions. All inputs to modules are passed in by value and all outputs are passed by reference. Inputs must only be read from and output ports can only be written to inside the function. We do not support writing to an output port multiple times inside the function. Modules can only process scalar values and cannot have arrays as input or output variables. Internal variables may be created but are not visible outside of the module.

Figure 25a shows a simple FIR filter written as a module. This code takes five inputs and computes a single output. When compiled, the hardware generated will resemble the circuit shown in Figure 25b. The interface to the module is exactly as described by the parameter list, the integer array T is not visible outside of the module.

Modules do not generate addresses or fetch values from memory, but instead have data pushed onto them, and then output scalar values after all computation has been performed. They are completely pipelined and can support processing new data every clock cycle.

If a module contains a loop, it will automatically be fully unrolled. Hence, any loop inside of a module must have an end bound that can be statically determined. Figure 26a provides an example of the supported loop structure inside modules.

After unrolling, constant and copy propagation, we end up with the hardware as shown in Figure 26b which is a single multiply as we would expect. There is no loop control or other control created as the loop has been removed.

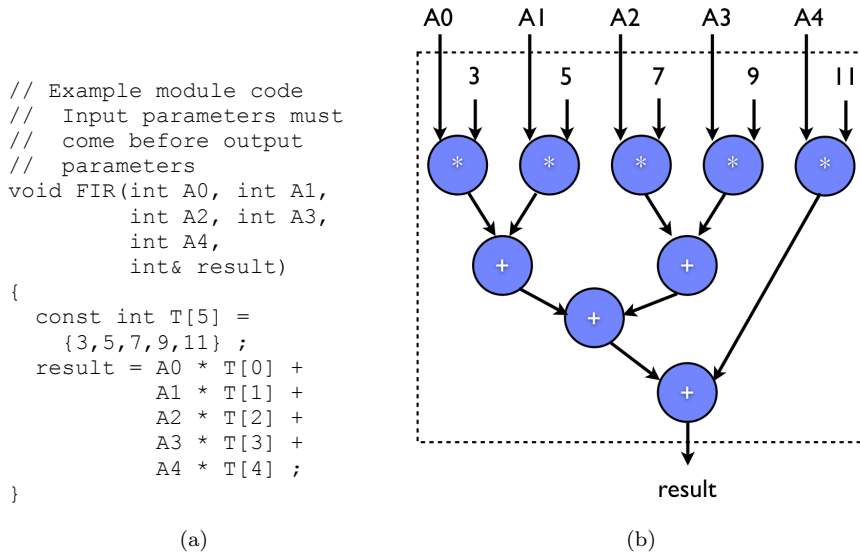


Figure 25: (a) Module Code in C and (b) generated hardware

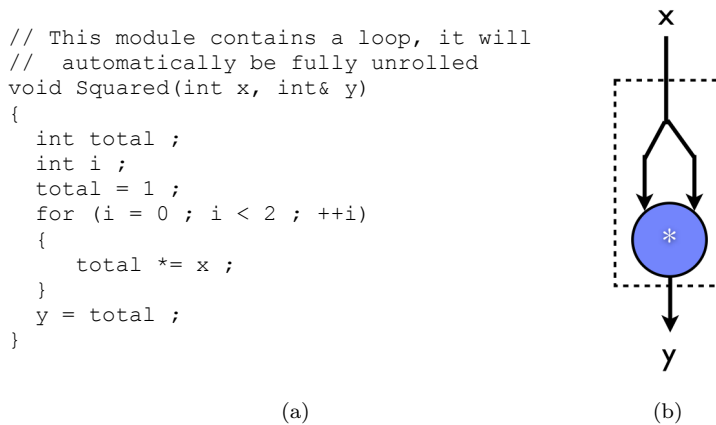


Figure 26: (a) Using a loop in module code and (b) resulting hardware

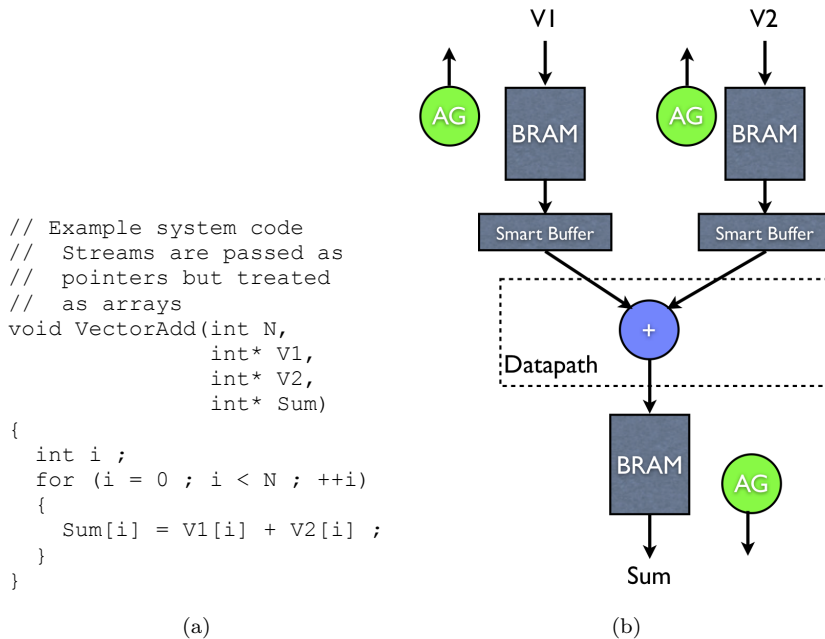


Figure 27: (a) System Code in C and (b) generated hardware

4.3 System Code

System code performs computation on streams of data and produces streams of data. Scalars may also be read as input and generated as output, but as opposed to modules, input scalars are read once at the beginning of computation and output scalars are only generated once at the end of computation.

Similar to module code, system code is written as a void function that takes input and output parameters. Input scalars are passed by value, output scalars are passed by reference, and both input and output streams are passed as pointers. The function definition must declare inputs before outputs. Although passed as pointers, the internal use of streams must be through array accesses.

An example of system code is shown in Figure 27a. This code takes a single input scalar that is used to determine the length of the incoming streams, two input streams V1 and V2, and an output stream Sum. The computation adds all elements of the two input vectors and outputs them to the Sum stream. Like module code, all inputs must be declared in the parameter list before any outputs.

The generated hardware is shown in Figure 27b. Each stream specified in the C code generates a memory interface that includes an address generator (AG) and a BRAM FIFO structure. The specifics of the hardware communication protocols are discussed in Section 5. Data reuse is handled through the creation of smart buffers, which is detailed in Section 4.9.7. The code located in the innermost loop will be translated into a datapath that is separate from the control.

4.3.1 Windows and Generated Addresses

When generating code, we infer the size of the memory we are accessing from both the loop bounds and the size of the accessed window. For example, the loop bounds in Figure 28 suggest a 10x10 memory. However, the code inside the loop accesses a 3x3 window, so we generate code that assumes a 13x13 memory. The addresses we generate will be the same as in C, and note that if run in C on a 10x10 array the results will be undefined.

When fetching the first window, we will therefore generate the offsets 0, 13, and 26 for the first column

```

void WindowSystem(int* A, int* B)
{
    int i, j ;
    for (i = 0 ; i < 10 ; ++i)
    {
        for (j = 0 ; j < 10 ; ++j)
        {
            B[i][j] = A[i][j] + A[i+2][j+2] ;
        }
    }
}

```

Figure 28: Accessing a 3x3 Window

```

// Example N-Dimensional code
void NDimensional(int*** A, int*** B)
{
    int i, j, k ;
    for (i = 0 ; i < 10 ; ++i)
    {
        for (j = 0 ; j < 10 ; ++j)
        {
            for (k = 0 ; k < 10 ; ++k)
            {
                B[i][j][k] = A[i][j][k] ;
            }
        }
    }
}

```

Figure 29: A system with a three dimensional input and output stream

and NOT 0, 10, 20. Similarly, we will generate the offsets 1, 14, and 27 for the second column, and 2, 15, and 28 for the third column of the window.

Additionally, we perform a normalization step on the window accesses to adjust for negative offsets. If the C code accesses an array with a negative offset, for example $A[i-2]$ and $A[i-1]$, we normalize these values to start at location 0, meaning the previous offsets will be adjusted to $A[i]$ and $A[i+1]$. After the normalization, we determine the size of the memory rows we are accessing identically as above.

4.3.2 N-dimensional arrays

ROCCC can accept arbitrary dimension arrays. Figure 29 shows example C code that both inputs a three dimensional array and outputs a three dimensional array. When declaring an N-dimensional array, the parameter must be a N-dimensional pointer.

4.3.3 Feedback detection

Variables whose values are used in multiple iterations of the for loop in system code are detected and turned into feedback variables. Figure 30 shows example code that contains a feedback variable. In this code, the value of `currentMax` is used in the initial loop iteration and is then carried through all of the additional loop iterations.

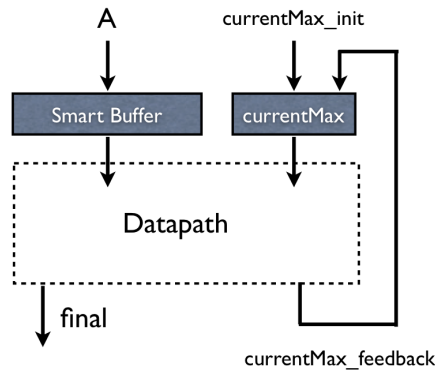
When converting the code in Figure 30a into hardware, we get a circuit that resembles that in Figure 30b. All variables that are determined to be feedback variables will have an additional hardware input port generated for the initial value of the variable. Subsequent iterations of the datapath can only be executed once the value of the feedback variable is known, which in the worst case will be at the bottom of the pipeline.

```

// Example code with feedback
void MaxSystem(int N, int* A,
               int& final)
{
    int i ;
    int currentMax ;
    for (i = 0 ; i < N ; ++i)
    {
        if (A[i] > currentMax)
        {
            currentMax = A[i] ;
        }
        else
        {
            currentMax = currentMax ;
        }
        final = currentMax ;
    }
}

```

(a)



(b)

Figure 30: (a) System Code That Contains Feedback and (b) Generated Hardware

This feedback may potentially decrease circuit throughput if the C code requires the feedback variable to be determined at the bottom of the pipeline and used at the top of the pipeline.

Feedback variables are not output at the end of computation and if you wish to have the final value output you must assign a separate output variable, as shown in Figure 30b.

4.3.4 Summation reduction

A special condition of feedback variables is a summation reduction. When the feedback detected is purely performing a summation reduction the feedback can be performed in one clock cycle and does not necessarily affect the throughput of the circuit.

An example of the code recognized as a summation reduction is shown in Figure 31a. The hardware generated, as shown in Figure 31b, will contain a datapath that handles the feedback internally and can support full throughput on the data streams.

4.4 Instantiating Modules

Both module code and system code can instantiate other modules to be integrated directly into the generated hardware. When a module is compiled, it is exported for use in other code. All modules have header information placed into the file "roccc-library.h." These functions can be called from other ROCCC code and each function call will be translated into a module instantiation.

The system code shown in Figure 32a processes a data stream and instantiates the module that was shown in Figure 25. When compiled, the generated hardware will resemble the circuit shown in Figure 32b.

IMPORTANT NOTE: Currently, array references can be used as inputs to modules but the outputs of modules can not be mapped to array references. If you wish to accomplish this, you must declare an intermediate temporary variable and assign the output of the module to this variable and then assign the variable to the output array. This is shown in Figure 32a as the output of FIR must be mapped to the variable tmp and then assigned to the output stream B.

4.4.1 Inlining Modules

The user has control of if module instantiations are treated as black boxes or inlined. When inlined, the individual operations of the module are exposed to the top level design and can be optimized around at the expense of increased compile time. As an example, Figure 32c shows the resulting circuit structure of the

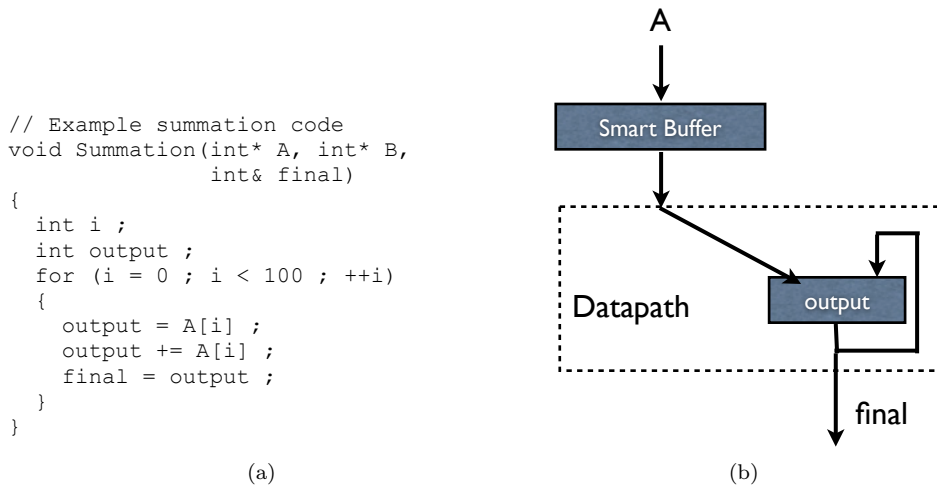


Figure 31: System Code That Results in a Summation Reduction

FIR system code shown in Figure 32a. Note that instead of a black box the top level design has all of the individual operations exposed and may perform additional optimizations on this code.

4.5 Control Flow

ROCCC code supports arbitrary if statements through predication. The quality of the generated circuit is directly affected by the use of predication, so care should be taken in constructing the C code to minimize logic.

In the simplest case, an if statement that determines one of two values to store into a variable will be translated into a boolean select statement. Figure 33 shows the transformation undergone from original C code to intermediate representation and finally to the generated hardware. If statements written in exactly this way will always result in a mux in the generated hardware.

All other combinations of if statements will be reduced to this form through predication. If there are any paths through which a variable might not be initialized, the generated hardware will either choose a default value of 0 or create a feedback variable that requires an initial value. An example of this is shown in Figure 34. The variable *x* is only assigned if the expression (*value* > 5) is true. In the generated hardware we must assign a value to *x* regardless of the expression's result, and so we assign a default value. In modules, this default value is 0, while in systems the default value is itself, which will introduce a feedback variable in a way that the user might not have expected.

4.6 Look Up Tables

ROCCC detects arrays declared locally in functions and translates them into hardware lookup tables. On the hardware, these lookup tables will be placed in BRAMs if any are available, or created as logic if no BRAMs are available or if the underlying platform does not support BRAMs. Depending on the available resources, the actual hardware resources used might be larger than what the C code implies (i.e. we might round up to the nearest power of two in order to allocate actual BRAM resources).

Lookup tables should be declared as a non-const array of any type and support random run-time access as shown in Figure 35. The small example in Figure 35 shows the declaration of a local lookup table that consists of 10 initialized elements. Every clock cycle the generated code will read a value from the input stream *A* and use that value to determine which element of the lookup table to access. If the value in stream

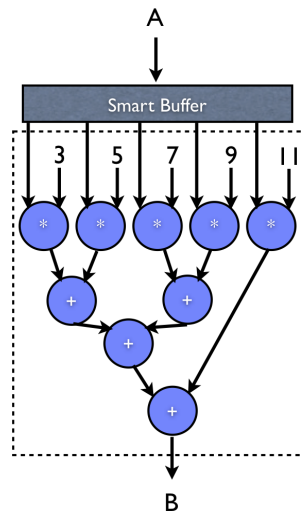
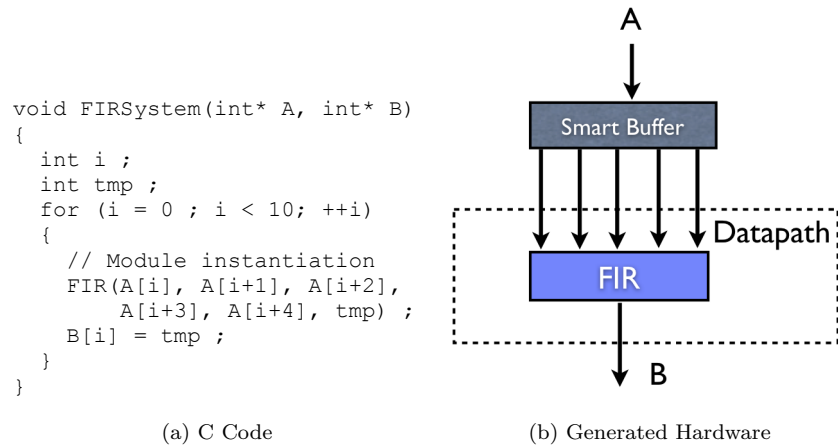


Figure 32: (a) Code That Instantiates a Module, (b) the Generated Hardware, and (c) Generated Hardware After Inlining


```

if (value > 5)
{
  x = 1 ;
}
else
{
  x = 2 ;
}

```

(a)

```

x = ROCCCBoolSelect(1, 2, (value > 5)) ;

```

(b)

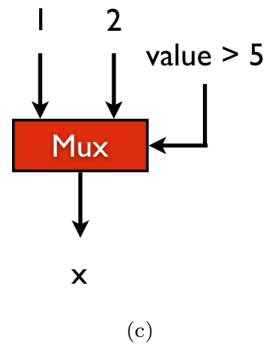


Figure 33: Boolean Select Control Flow. (a) In the original C, (b) in the intermediate representation, and (c) in the generated hardware datapath.

```

if (value > 5)
{
  x = 1 ;
}

```

(a)

```

pred = (value > 5) ;
x = ROCCCBoolSelect(1, x, pred) ;

```

(b)

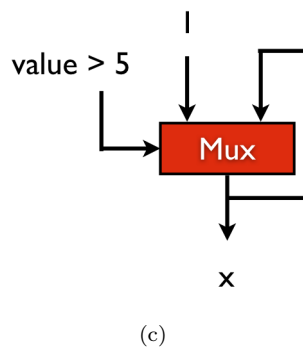


Figure 34: Predicated Control Flow (A) in the original C, (B) in the intermediate representation, and (C) in the generated hardware.

```

// Input Stream is a set of numbers between 0-10
void LUTTestSystem(int* A, int N, int* B)
{
    int i ;
    int LocalLUT[10] = { 1, 3, 9, 0, 5, 6, 4, 7, 8, 2 } ;
    for (i = 0 ; i < N ; ++i)
    {
        B[i] = LocalLUT[A[i]] ;
    }
}

```

Figure 35: Reading From A Preinitialized LUT

```

void LUTTestSystem(int* A, int* B, int* C, int* D, int* E)
{
    // Multidimensional LUTs are supported
    int LocalLUT[5][5] = {
                                                {1, 2, 3, 4, 5 },
                                                {1, 2, 3, 4, 5 },
                                                {1, 2, 3, 4, 5 },
                                                {1, 2, 3, 4, 5 },
                                                {1, 2, 3, 4, 5 }
    } ;

    int i ;

    for (i = 0 ; i < 10 ; ++i)
    {
        LocalLUT[A[i]][B[i]] = C[i] ; // One write per pipeline
        D[i] = LocalLUT[A[i]][0] ;
        E[i] = LocalLUT[0][B[i]] ;
    }
}

```

Figure 36: Multiple Reads/Single Write LUT

A lies out of bounds of the lookup table in C (i.e. is greater than or equal to 10), then the hardware will return an undefined value. Multidimensional Lookup tables are also supported, as shown in in Figure 36.

Lookup tables generated by ROCCC support an arbitrary number of reads, but currently only supports one total write to each lookup table in the C code that corresponds to one write in the generated pipeline. Care must be taken when applying optimizations as unrolling would increase the number of writes to lookup tables resulting in incorrect code.

4.7 Composed System Code

ROCCC now has added preliminary support for the composition of system code into larger systems. The code as shown in Figure 37 allows for the composition of systems in a manner very similar to constructing modules from other modules. The code itself is a system, and as such may be used in larger composed systems.

As shown in Figure 37, input streams are declared and passed as normal. Output streams to composed systems must be identified by being passed by reference. Any intermediate variables or streams that need to

```

#include "roccc-library.h"

void SystemToSystem(int* A, int*& B)
{
    int* internal ;
    PassThrough(A, 96, internal) ;
    PassThrough(internal, 96, B) ;
}

```

Figure 37: Composite Systems

be declared to connect two systems are not accessible outside the composed system. Figure 37 composites two `PassThrough` systems, which take an input stream and a number and return an output stream. The input to the composed system (A) is mapped to the input of the first internal `PassThrough` system. The output stream of the first `PassThrough` system is mapped to the internal stream "internal" and flows into the second `PassThrough` system. The output of the second `PassThrough` system is mapped to the output stream of the composite system (B).

Individual systems in composite systems may be specified as redundant using a label. When a system is made redundant, several copies are instantiated and the each input stream must be split into separate streams for each instance. Similarly, all of the output streams from the multiple copies must be merged into a stream voter. We have added support for these two constructs as intrinsics, so it is the user's responsibility to provide ROCCC with the appropriate stream splitters and voters and ROCCC will instantiate them as appropriate.

4.8 Legacy Code

In previous versions of ROCCC, modules and systems were coded slightly differently. We still support compilation of legacy code, although newer features such as inlining are not supported for legacy code and mixing legacy code with new style code may cause problems in the future as legacy code is deprecated.

4.8.1 Legacy Module Code

Legacy module code must define both an interface and implementation. The interface is described as a struct that identifies all of the inputs and outputs to the module. Input ports must be identified by adding the suffix "_in" and output ports must be identified by adding the suffix "_out."

The implementation function must be a function that returns and receives an instance of this struct by value. Any return statements that are not at the end of the function are ignored and cannot be used as a form of control flow. All computation inside this function will be translated to hardware.

The FIR filter shown in Figure 38 is written in this style. Note that the hardware generated for this code is nearly identical to the hardware generated for the same code written in Figure 25. The only difference will be in the ordering of the ports once compiled.

IMPORTANT NOTE: When compiling Legacy ROCCC modules, the order in which you pass the parameters is not necessarily the order in which you declared them in the struct. The order in which you pass parameters must match the order in which they appear in the struct as exported in the "roccc-library.h" file. If using the GUI, this ordering is available by double-clicking the module in the IPCores view. Modules written in the new style will have the parameters in the same order as written.

4.8.2 Legacy System Code

Legacy system code is nearly identical to the new style system code with the exception that parameters were not accepted. Input and output arrays and scalars are declared locally and inferred during compilation.

```

typedef struct
{
    int A0_in ;
    int A1_in ;
    int A2_in ;
    int A3_in ;
    int A4_in ;
    int result_out ;
} FIR_t ;

FIR_t FIR(FIR_t t)
{
    const int T[5] = { 3, 5, 7, 9, 11 } ;
    t.result_out = t.A0_in * T[0] + t.A1_in * T[1] +
                  t.A2_in * T[2] + t.A3_in * T[3] + t.A4_in * T[4] ;
    return t ;
}

```

Figure 38: Legacy Module Code

```

typedef int ROCCC_int12 ;

void Test(ROCCC_int12 a_in, ROCCC_int12& b_in)
{
    // ...
}

```

Figure 39: Declaring And Using A Twelve Bit Integer Type

4.9 Hardware Specific Optimizations

There are several features specific to ROCCC that allow you to create specific hardware and are not reflected in the software. These include bit-width specification, systolic array generation, and temporal common subexpression elimination.

4.9.1 Specifying Bit Width

Every integer variable you declare in the C code can have a nonstandard bit width tailored to your application. The supported floating point bit widths are 16, 32, and 64, with the default being 32 bits. The choice of cores instantiated in the datapath will be based upon the bit width of the variables passed to them. Smaller bit width variables will be extended to take advantage of the larger cores unless no such core exists, in which case the variables will be truncated to use the largest core available.

The quality and precision of the generated VHDL can vary based upon how the C is specified, so use caution. By default, all operations are expanded to the highest precision before being performed and then truncated if necessary as the last step. In the generated VHDL, an N bit addition is stored into an (N + 1) bit value and a multiplication between two N-bit numbers is stored into a number with 2N bits. The user may select the optimization "MaintainPrecision" to truncate at every step.

Specifying the specific bit width is done by declaring a typedef at the beginning of your program. This typedef must be in the form of ROCCC_intX where X is any positive number, as shown in Figure 39. This type can then be used to declare any variable with the appropriate size.

```

L1: for (i = 0 ; i < 100 ; ++i)
    {
        for (j = 0 ; j < 100 ; ++j)
            {
                A[i][j] = A[i-1][j-1] + A[i][j-1] + A[i-1][j] ;
            }
    }

```

Figure 40: C Code To Generate A Systolic Array

4.9.2 Systolic Array Generation

Systolic array generation is an optimization that takes a wavefront algorithm operating on a two-dimensional array and converts it into hardware consisting of a single dimensional array of elements that feed back to each other. The original C code must be in the form of a doubly nested for loop that calculates the value of a two-dimensional array based upon some function of the previous elements of that array.

In order for systolic array generation to recognize the optimization, the outer loop must be labelled as shown in Figure 40.

The current version of systolic array generation only transforms a precise software architecture into a specific instance of a systolic array. The code must have a single two-dimensional array where the value of every cell is based upon some function of the cells located to the north, west, and northwest. Optionally, the C code may have a constant array of values based upon the outer loop bounds and a single dimensional input array based upon the loop bounds of the innermost loop as seen in the Smith Waterman example. Any other software architecture is not currently supported for the systolic array generation optimization.

After transformation, the resulting hardware will expect a one dimensional input array (A_input) and produces a one dimensional output array (A_output) in place of the original two-dimensional array. The input stream A_input should be the values of the topmost row of the original two-dimensional array. The output stream A_output will generate the bottom row of the original two-dimensional array. All of the intermediate values are discarded and not output in the generated hardware structure. Additionally, the first column of the original two-dimensional array must be passed in as scalars to the resulting hardware.

4.9.3 Temporal Common Subexpression Elimination

Temporal common subexpression elimination (TCSE) analyzes loops and detects common code across loop iterations. For example, if the same value is calculated in loop iteration 1 and loop iteration 2, this will be detected. When generating hardware, we take advantage of this fact and create feedback variables that eliminate redundant computations.

TCSE can only be performed on system code. The code does not have to be written in any special way to take advantage of TCSE.

An example of the difference in hardware generated can be seen in Figures 41 and 42. These block diagrams show the original structure of the Max Filter System hardware that contains four Max Filter modules and operates on a sliding 3x3 window and the Max Filter System after TCSE has been performed. After TCSE, the generated hardware only has two Max Filter modules and two have been replaced with feedback variables.

The generated hardware does require initial values for each piece of hardware eliminated, so you might have to change the way you pass data into the hardware depending on if you perform TCSE or not.

4.9.4 Arithmetic Balancing

The user has the choice of performing arithmetic balancing on the generated hardware. The optimization finds expressions composed of a single operator performed in serial, and changes the order that the subexpressions are calculated in to minimize the time to calculate the expression. Only associative and commutative operators are balanced; currently, addition, multiplication, and bitwise AND, OR, and XOR

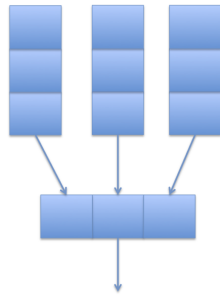


Figure 41: Block Diagram Of Max Filter System

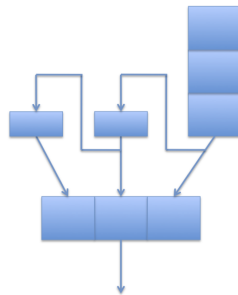


Figure 42: Block Diagram Of Max Filter System After TCSE

are balanced. For example, the statement $a = b + c + d + e$ in software will be calculated serially. By performing arithmetic balancing, the statement is changed into $a = (b + c) + (d + e)$, with $b+c$ and $d+e$ calculated in parallel. Because floating point operators are not strictly associative and commutative, and order of execution matters when dealing with overflow, this optimization may change the final result when using floating point values.

4.9.5 Copy Reduction

ROCCC automatically inserts copy registers in between pipeline stages if a value is not used immediately after it is calculated. If an operation could correctly be calculated in several different pipeline stages, one of those stages will minimize the total bits that are copied (both coming into that operation from previous stages, and leaving that operation to later stages that use the calculated value). This pass attempts to find a placement for operations that minimizes the total number of copied bits. Starting with the edge in the use-def graph that has the most number of bits copied, edges are "tightened" by moving the nodes at the edges ends toward each other. By repeating this process, and saving a snapshot of the graph whenever a minimal number of copied bits is found, eventually a local minimum is found that minimizes the number of copied bits. This can take as long as $O(E)$, where E is the number of edges in the use-def graph, although in practice a minimum is found quickly.

4.9.6 Fanout Tree Generation

When compiling high level code, the amount of parallelism that is generated in hardware may not be readily apparent. High fanout can seriously affect the clock rate or area of the generated circuit, and so we have added user control to specify the maximum allowable fanout for any register in the generated circuit. If the fanout exceeds this number, ROCCC generates a tree of registers in separate pipeline stages, increasing the latency but shortening the clock and simplifying the routing.

```

for(i = 0 ; i < 5 ; ++i )
{
  for (j = 0 ; j < 5; ++j)
  {
    row1 = A[i][j] + A[i][j+1] + A[i][j+2] ;
    row2 = A[i+1][j] + A[i+1][j+1] + A[i+1][j+2] ;
    row3 = A[i+2][j] + A[i+2][j+1] + A[i+2][j+2] ;
    B[i][j] = row1 + row2 + row3 ;
  }
}

```

Figure 43: System Code That Accesses a 3x3 Window

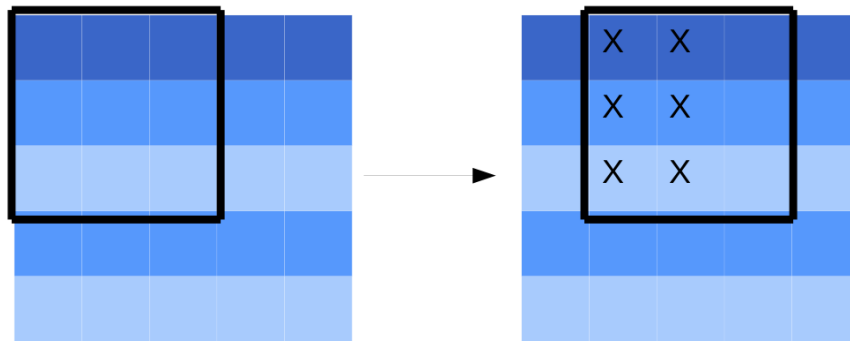


Figure 44: 3x3 Smart Buffer Sliding Along a 5x5 Memory

4.9.7 Smart Buffers

When generating code for systems, array accesses are analyzed looking for possible reuse between loop iterations. These reuse patterns can be exploited and reduce the number of off-chip memory accesses. The generated hardware will contain Smart Buffers to exploit the reuse between loop iterations, which internally consist of registers that cache the portion of memory reused.

The code in Figure 43 requires a 3x3 window from the memory A in order to execute each loop iteration. Note that as in the C code, the ROCCC generated hardware will access rows 0-6 and columns 0-6 of the image even though the loop bounds are < 5 . As shown in Figure 44, code that accesses a sliding 3x3 window over a larger memory can reuse six values between loop iterations (shown with X's in the diagram). The smart buffer initially reads nine values from memory and exports all nine to the datapath for the first loop iteration, and for subsequent iterations only three are read for each loop iteration.

The code as shown in Figure 45 will be analyzed by ROCCC and determined that no reuse occurs between loop iterations. In this case, a FIFO interface is generated. For each loop iteration, two elements are read in, as in Figure 45. No reuse can be exploited between consecutive loop iterations.

```

for (i = 0 ; i < 5 ; i += 2)
{
  B[i] = A[i] + A[i+1] ;
}

```

(a)



(b)

Figure 45: (a) System Code that reads from a FIFO and (b) Memory fetches when using a FIFO

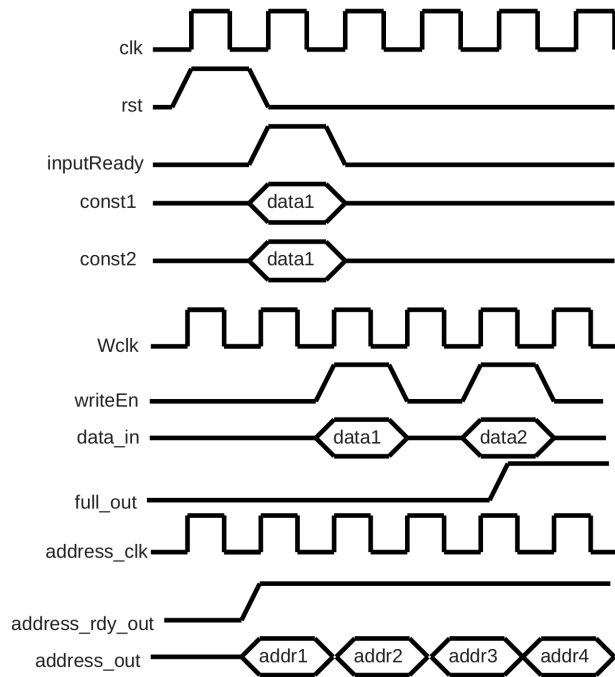


Figure 46: Timing Diagram Of A System With Both Input Scalars And Input Streams

5 Interfacing With Generated Hardware

5.1 Port Descriptions

The VHDL generated by ROCCC communicates with the external platform in a variety of ways described in this section. All inputs and outputs that connect to ROCCC code are assumed to be active-high.

5.1.1 Default Ports

Each hardware module and system generated by ROCCC will contain six ports by default. These default ports are `clk`, `rst`, `inputReady`, `outputReady`, `done`, and `stall`. Their use is described here:

- `clk`
The `clk` port is the clock of the hardware and should be connected to a clock signal. All processes internal to ROCCC code trigger off of the rising edge of the clock. All ROCCC components and systems assume a single clock to drive all the hardware.
- `rst`
The `rst` port is the reset signal to the generated hardware. Driving the reset port high resets the hardware to an initialized state. As long as the reset port is held high, the hardware will remain in the reset state, regardless of the inputs. After bringing the reset port low, the hardware will begin responding to the input signals. The hardware generated by ROCCC requires the reset port to be driven high for at least ten clock cycles for initialization purposes. Not doing so may leave the component in an uninitialized state. The use of the reset signal and the initialization of hardware that contains both input registers and input streams is shown in Figure 46.
- `inputReady`
The `inputReady` signal should be driven high when the signals that correspond to input scalars are

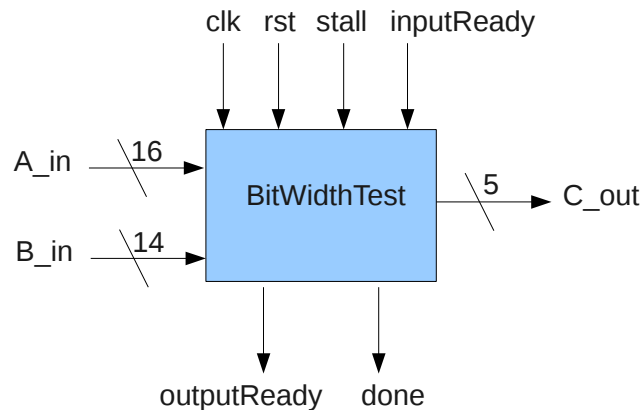


Figure 47: Block Diagram Of A Generated Module

valid. As long as the inputReady signal is high, input scalars will be read on every rising edge of the clock. Setting the input scalars to valid data and setting inputReady high should be the first thing done by any interfacing code.

- outputReady
The outputReady port goes high when valid data is placed on the output scalar ports of the hardware. The output data is valid simultaneously with the outputReady signal being high.
- done
The done port goes high when the hardware generated by ROCCC has finished processing all of the input it was designed to process and remains high until the reset signal is asserted.
- stall
The stall port is used by the interfacing code to stall the pipeline of the generated hardware.

5.1.2 Input And Output Ports

In addition to the default ports, input and output data ports will be generated by ROCCC. These may correspond either to single registers or to streams.

- Registers
For each input register, a single data port will be generated. When generating modules, all inputs are treated as registers. When generating systems, any single variable that acts as input to the main loop will be treated as an input register.

For each output register, a single data port will be generated. When generating modules, all outputs are treated as registers. When generating systems, any single variable that acts as output to the main loop will be treated as an output register.

Figure 47 provides a block diagram of a ROCCC generated module. This module includes both the default ports (located on the top and bottom) but also the user defined ports, which may be variable bit size (located on the left and right).

- Streams
All streams in ROCCC are split into two separate clock domains - a data clock domain, with associated ports, and an address clock domain, with associated ports. On the data clock domain, there is a data valid port, a data read / write enable port, and a positive number of data channels. On the address clock domain, there is an address stall port, an address valid port, and a positive number of pairs of address channel ports; these pairs each consist of a base address port and a count address port.

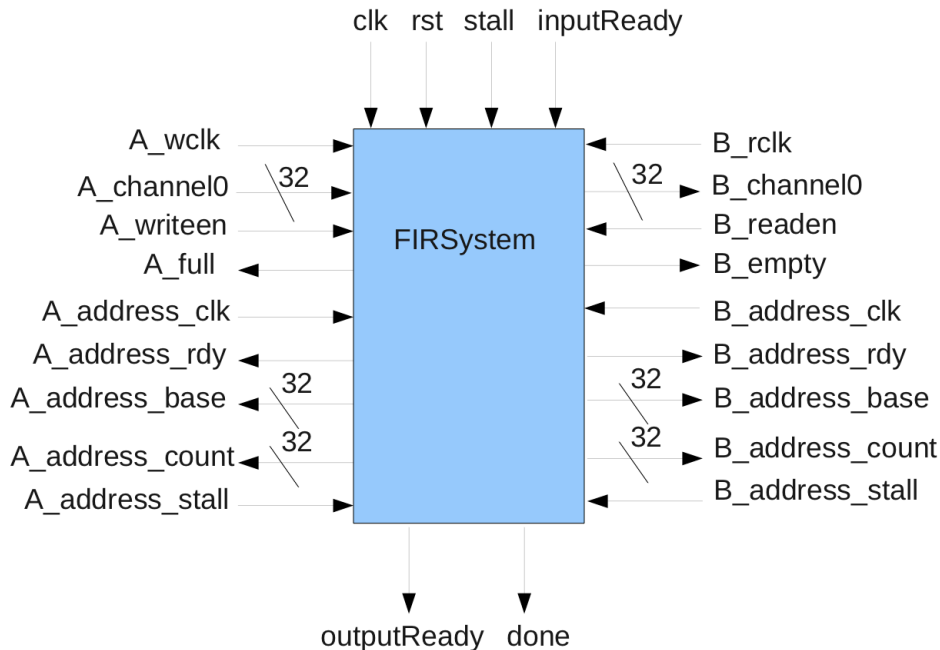


Figure 48: Block Diagram Of A Generated System

The number of data channels will be equal to the number of data channels the user specifies for the stream, and needs to be a factor of both the window size and the step window size, which the number of address channels will be equal to the number of address channels specified by the user, and needs to be a factor of both the window size and the step window size along the dimensions other than the innermost.

Figure 48 shows the block diagram of a generated system that communicates with a multi-dimensional buffer. The default ports are still generated (located on the top and bottom of the figure) as well as the interface to the streams. In addition to the ports generated for streams, input and output registers can be created as well.

For output streams, several ports will be generated: an address clock, an address valid port, an address stall port, a positive number of address channel pairs, a data clock, a data empty port, a data read enable port, and a positive number of data channel ports.

5.2 Interfacing Protocols

5.2.1 Input Registers

Input registers are used by both module and system code. They need to be set when `inputReady` is driven and are sampled on the rising edge of the clock. Driving the input registers is the responsibility of the calling code. In modules, the input registers can be changed every clock cycle. In systems, the input registers may be set only once, and must be set before passing any data to the input streams. See Figure 46 for the timing of interfacing with system code's input registers and Figure 49 for the timing of driving a module's input registers.

5.2.2 Input Streams

The input stream address generation and the input data protocol are decoupled, allowing address generation to happen independent of incoming data. In particular, there are four ports dealing with address

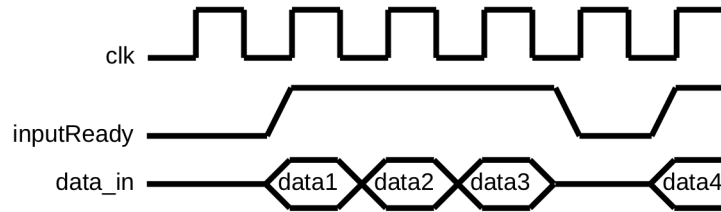


Figure 49: Timing Diagram Of Module Use

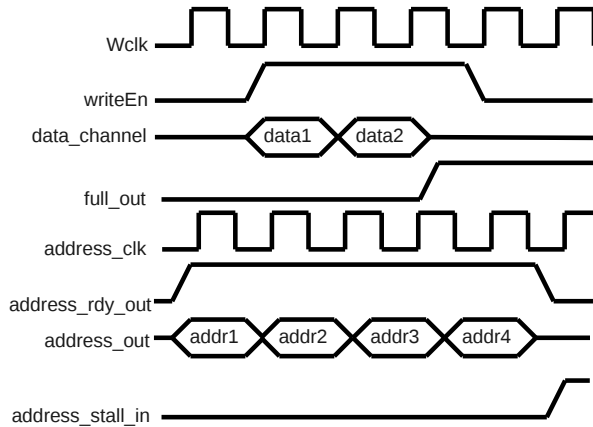


Figure 50: Reading From A Stream

generation and three that deal with input data (assuming 1 channel for both): `address_stall`, `address_rdy`, `address_channel_base`, `address_channel_count`, `data_full`, `data_writeEn`, and `data_channel`.

As long as the `address_stall` port is not held high, addresses will be generated.

When an address is being generated, the `address_rdy` port will be brought high and the `address_channel_base` port will hold the base address of the values needed. The `address_channel_count` port holds the number of consecutive elements from that base address that are desired; it is up to the end user to use these two values to derive the memory locations needed. The `address_rdy` will only be held high for one clock cycle for each individual address being generated. If addresses are being generated in consecutive clock cycles, the `address_rdy` port will be continuously high.

The user defined interfacing code needs to service memory requests in a FIFO fashion. ROCCC generated code expects the data we receive to be in the exact order as requested. When data is ready, if the full port is not currently high, the data must be placed on the input data port(s) and write enable must be asserted and held high for a clock cycle. As long as full remains low, write enable can be kept high and data can be put onto the data port(s) every clock cycle.

The number of outstanding memory requests generated by the ROCCC-generated code is unbounded; if the interfacing code can only handle so many outstanding memory requests, it is up to that code to bring `address_stall` high when that limit is reached.

If the user has specified that a given stream is a multi-channel stream, then it is necessary to set all channels of the input with valid data before asserting the write enable signal. The channels in the ROCCC generated code are numbered from 0 to N and it is up to the user generated interfacing code to place the oldest data in channel 0, the second oldest data in channel 1, and so on. Once all channel data has been fetched, if `data_full` is not high, the interfacing code should set write enable high and hold it high for one clock cycle. An example of this timing protocol is shown in Figure 51.

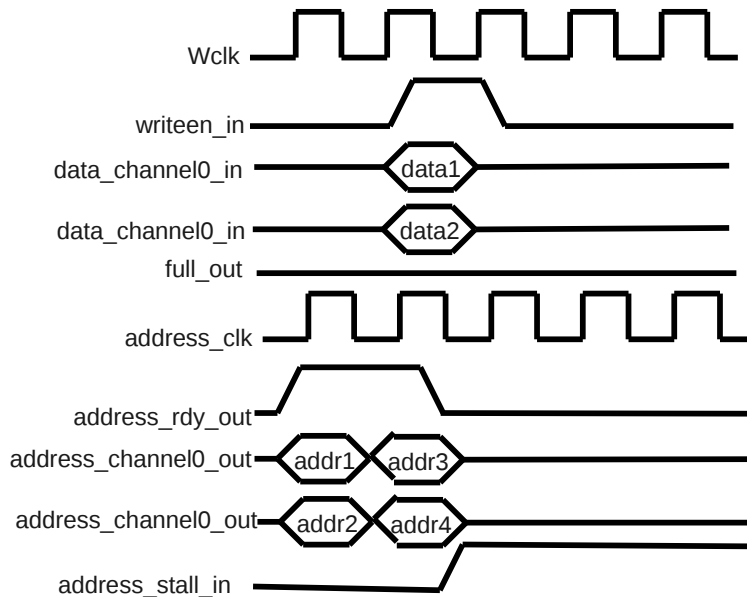


Figure 51: Reading From A Stream With Multiple Channels

5.2.3 Output Scalars

Output scalars are driven when `outputReady` is driven. The number of clock cycles before `outputReady` goes high after driving `inputReady` is based off the delay of the pipeline. Code that interfaces with systems should ignore `outputReady`; if values are to be sampled every iteration of the loop, then a stream should be used. System code that properly uses output scalars should only be interested in a final value, which will be valid when done goes high, not when `outputReady` goes high.

5.2.4 Output Streams

Output streams have four ports dealing with address generation and three that deal with input data (assuming 1 channel for both): `address_stall`, `address_rdy`, `address_channel_base`, `address_channel_count`, `data_empty`, `data_readEn`, and `data_channel`.

As long as the `address_stall` port is not held high, addresses will be generated until the total amount of addresses have been written.

When an address is being generated, the `address_rdy` port will be brought high and the `address_channel_base` port will hold the base address of the values needed. The `address_channel_count` port holds the number of consecutive elements from that base address that are being written; it is up to the end user to use these two values to derive the memory locations needed. The `address_rdy` will only be held high for one clock cycle for each individual address being generated. If addresses are being generated in consecutive clock cycles, the `address_rdy` port will be continuously high.

The fifo interface protocol of the output streams is similar to the fifo interface protocol of the input stream. When the output controller has valid data from the datapath, the first element of the stream is written to the data port(s), and the address of that data element is written to the corresponding address port. The empty port is brought low, signaling there is data in the fifo. When the read enable signal is brought high, the first data element is put onto the data port(s) and the address port is loaded with that element's address. See Figure 52 for an example of the timing protocol for output streams. As shown in Figure 52, the data and addresses are decoupled.

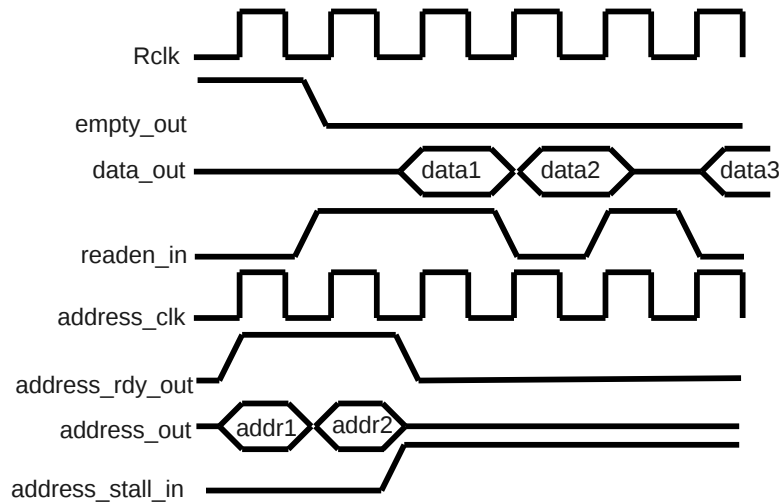


Figure 52: Timing Diagram of Output Streams

Because the output controller may be serializing data calculated in parallel, the datapath must be stalled until all of the data is serialized. This happens entirely internally, but functions equivalently to bringing the stall port high - the datapath is stalled, the input controller continues to read but will not push data onto the datapath, and other output streams may run out of valid data. For this reason, it is important not to rely on a specific timing for any stream interfacing. Rather, the fifo interface should be relied on to guarantee that data is transferred correctly.

If it is imperative that data not be serialized, it is preferred to create several output streams or to create a multi-channel stream.

5.2.5 Done

The done signal works differently, depending on if it is coming from module or system code. Module code will drive the done signal high as soon as the first value is processed; this can safely be ignored by any code interfacing with a ROCCC module, as modules are stateless and can never be considered done. System code will drive the done signal high on the rising edge of the clock after the last output values are set. Figure 53 provides an example of the done signal's behavior in a typical system.

5.2.6 Stall

The stall signal allows the interfacing code to stall the datapath in both modules and systems. Stalls are not instantaneous - it takes 1-2 clock cycles for the stall signal to propagate all the way up the datapath, to both the input and output controller. In hardware, a common use for a stall signal is when interfacing with memory that may become full. However, both input and output streams are two-way handshakes, and any stream can be "stalled" by simply not completing the handshake. For this reason, and because stalls are not instantaneous, stalls should be reserved for the case when there is no alternative.

When the stall signal is brought high, both input and output streams will continue to interact with any interfacing code. However, the datapath will be frozen, and data will not be pushed onto the datapath. Again, prefer to handle full memory in the stream interface, and not with the stall signal.

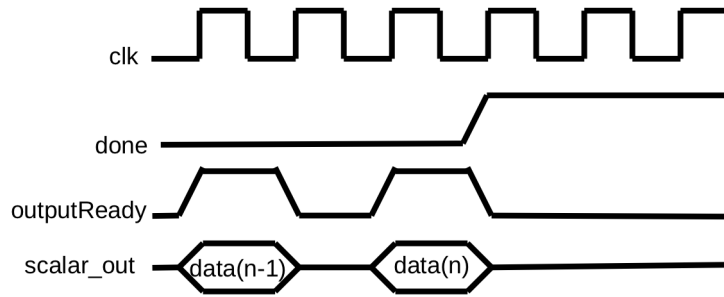


Figure 53: Timing Diagram Of The End Of A System's Processing

```

for(i = 0 ; i < height ; ++i)
{
    for (j = 0 ; j < width ; ++j)
    {
        MAX(window[i][j], window[i][j+1], window[i][j+2], maxCol1) ;
        MAX(window[i+1][j], window[i+1][j+1], window[i+1][j+2], maxCol2) ;
        MAX(window[i+2][j], window[i+2][j+1], window[i+2][j+2], maxCol3) ;
        // Find the maximum of the three columns
        MAX(maxCol1, maxCol2, maxCol3, finalOutput) ;
    }
}

```

Figure 54: C Code For MaxFilterSystem Which Uses A 3x3 Window

5.3 Memory Organization

5.3.1 Input Streams

Input streams will generate a base address and a count for each chunk of contiguous memory they access. Both one dimensional and two dimensional streams generate addresses for each requested value and it is up to the interfacing code to decide how to treat these values.

The addresses that are generated by the system when accessing memory are assuming an input memory of a certain size. This assumed size is based off of several factors, including both the window size of the input and the size of the for loops driving the window. For example, given the C code for MaxFilterSystem as shown in Figure 54, the window size is 3x3 and the for loop size is width x height. Given these values the input memory size is $(width + 3 - 1) * (height + 3 - 1)$. To traverse a memory of size 20x20, the width and height passed in to the hardware need to both be 18 ($20 + 1 - 3$).

When processing the code in Figure 54, both height and width will become input registers and need to be set along with `inputReady`. Only then is it safe to begin returning valid data to the component's request for window elements; not setting height and width to the correct values will result in the wrong addresses being generated.

5.3.2 Output Streams

The memory layout for output streams follows the same rules as the memory layout of input streams. The window size and the for loop end values will both be used to calculate the address of each value's location in memory. For the code in Figure 55, the first iteration through the loop will calculate $B[i]$, $B[i + 1]$, and $B[i + 2]$ with $i = 0$, and so the outputted address for $B[i]$, $B[i + 1]$, and $B[i + 2]$ will be 0, 1, and 2, respectively. On the second iteration through the loop, $i = 1$, so the outputted address for $B[i]$, $B[i + 1]$, and $B[i + 2]$ will be 1, 2, and 3, respectively. Multi-dimensional code works similarly.

```

for(i = 0 ; i < 5; ++i)
{
    B[i] = A[i] + A[i+1] ;
    B[i+1] = A[i+1] + A[i+2] ;
    B[i+2] = A[i+2] + A[i+3] ;
}

```

Figure 55: C Code That Writes To Three Locations In The Same Stream Each Loop Iteration

One note to make is that there are no guarantees made about the order of data coming out, nor are there any guarantees about the number of times a value may be output; in the previous example, it is easy to see that element $B[1]$ was written in both the first and second iteration of the loop. Elements written multiple times in different loop iterations may be actually written to more than once, or values may be cached to eliminate redundant writes to memory. In any case, it is important not to rely on a particular behavior.

5.3.3 Systolic Arrays

After using the systolic array optimization, two input streams and a set of input registers are created as inputs. The input registers should be loaded with the first column of the matrix, and the top row of the matrix is fed in as a stream. The input array T is also fed in as a stream. Refer to Figure 59 for the relationships between the original two dimensional array and the created registers and input streams.

5.4 Pipelining

Pipelining in ROCCC is guided by user-provided weights of basic operations. By varying these numbers, along with a desired clock cycle weight, the aggressiveness of pipelining can be controlled by the user. Under ROCCC, the data flow graph representing each loop body contains no initial registers. Registers are then inserted into the data flow graph until no register to register path has a total weight greater than the desired clock cycle weight.

In Figure 56, the leftmost *mux* has a critical path of one addition operation (assuming $Weight(add) > Weight(compare)$), while the rightmost *mux* has a critical path of one addition operation and one comparison. By choosing a desired delay d such that $Weight(mux) + Weight(add) < d < Weight(mux) + Weight(add) + Weight(compare)$, registers were inserted after the leftmost *mux*, but before the rightmost *mux*. This can be seen in Figure 57. When dealing with complicated multi-operation datapaths and a large pipeline depth, this sort of timing analysis is difficult and error-prone when performed by hand, and time consuming when done at the gate level on large graphs by the synthesis tool.

5.5 Fanout Reduction

A high fanout in a design can severely impact the frequency of the final hardware, especially when that high fanout is exacerbated by not having registers in between the fanout operation and the operations that use it. By specifying the max unregistered fanout, the user can specify at what point registers should be inserted to minimize the impact of a high fanout. As an example, the addition operation in Figure 58 has a high fanout; by inserting registers between it and the operations that use it, the impact on the frequency of the final design is minimized.

5.6 Intrinsic

Unlike in C, integer division, modulus, and floating point operations are expensive to do in hardware. In fact, there is no way to specify "add two 32-bit floats" or "multiply two 16-bit floats", other than implementing the algorithm yourself, or using a hardware IPCore specifically designed for that purpose. These operations

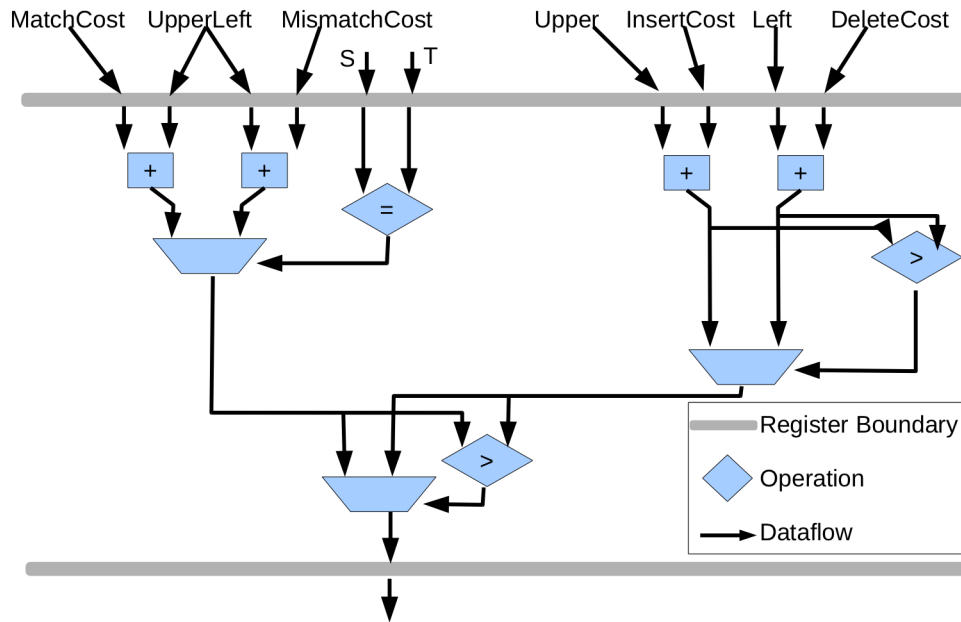


Figure 56: Basic Dataflow

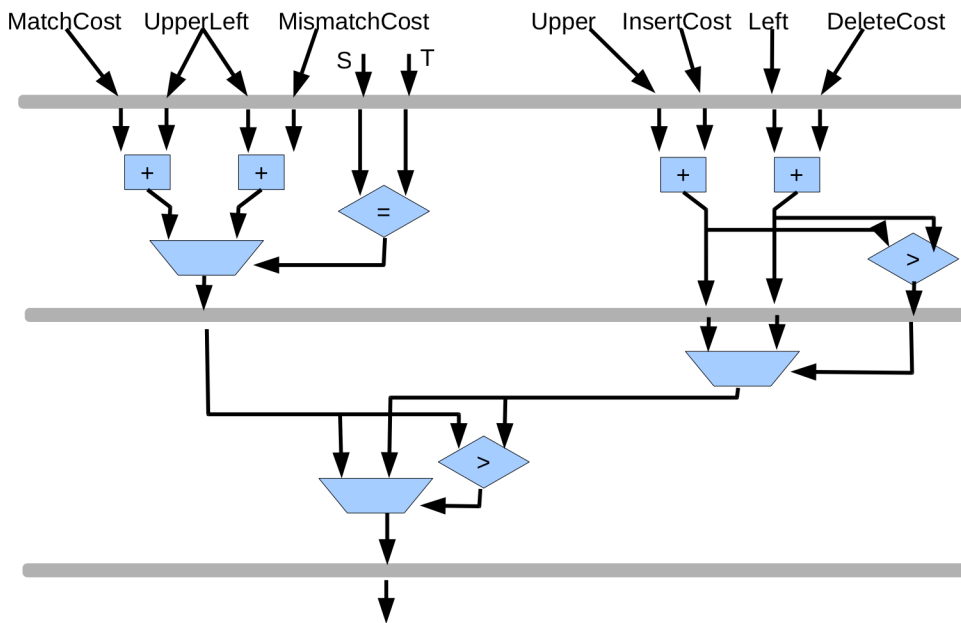


Figure 57: Medium Dataflow

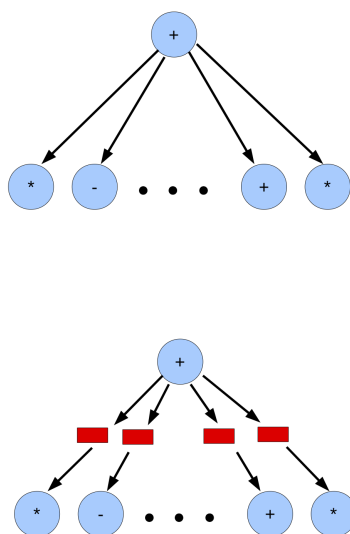


Figure 58: High fanout a) before registering and b) after registering

are significantly more complex than simple operations, such as addition, and because there are several ways to implement division, the synthesis tool does not blindly infer a solution.

In order to simulate or synthesize code generated with ROCCC that uses integer division, integer modulus, or floating point operations, it is necessary to create and include an intrinsic component into your simulation or synthesis project.

It is generally necessary, if you want to use floating point, to find an ipcore for each of the operations you need. Xilinx has the CoreGen utility to provide ipcores, while it is probably also possible to find free ipcores on a site like <http://opencore.org>.

Once you have found an ipcore that implements the operations you require, you will want to utilize it in your project. Traditionally, this would be done by instantiating it in the code that requires it, with each ipcore having slightly different requirements. For example, one divide core may have a reset or enable input, while another may not. Because ROCCC has no knowledge of what ipcore you will end up using, we cannot directly instantiate the ipcore you will use; instead, we instantiate a "wrapper" component. This component must be written by you, and provides a standardized interface that ROCCC can instantiate. However, this component does not have to implement any logic; it can simply instantiate the ipcore to implement the logic. In this way, a standardized interface is presented to ROCCC, but any IPCore can be used to implement the actual logic.

As an example, the declaration for a theoretical 32-bit floating point divide core is shown in Figure 60, with the corresponding wrapper shown in Figure 61.

When choosing an IPCore, it is important to keep several considerations in mind. First, the core should be fully pipelined, as ROCCC assumes all subcomponents are fully pipelined. Second, the core needs to have a way to stall the component; if no core is available that has a way to stall the component, a simple solution is to gate the clock, but this results in poor performance. Thirdly, the component must complete the calculation in a constant number of clock cycles. This number of clock cycles must be told to the GUI when importing the intrinsic into ROCCC.

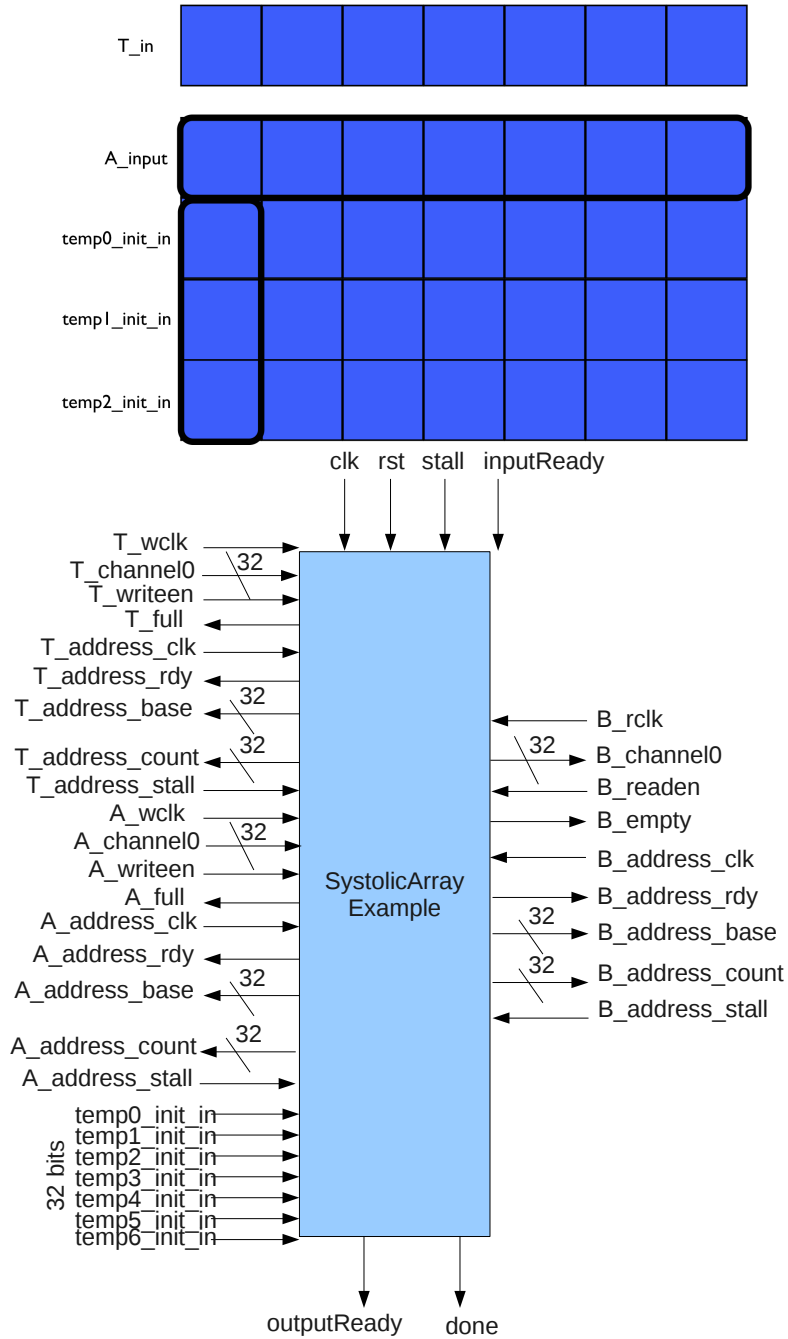


Figure 59: Generated Systolic Array Hardware

```

entity fp_div_gen32 is
  port (
    a : in STD_LOGIC_VECTOR(31 downto 0); --dividend
    b : in STD_LOGIC_VECTOR(31 downto 0); --divisor
    clk : in STD_LOGIC; --clock signal
    ce : in STD_LOGIC; --clock enable, brought low to stall the core
    result : out STD_LOGIC_VECTOR(31 downto 0) --quotient
  );
end fp_div_gen32;

```

Figure 60: Theoretical Interface to a 32-bit Floating Point Divide IPCore

```

entity fp_div32 is
  port (
    clk : in STD_LOGIC; --clock signal
    rst : in STD_LOGIC;
    inputReady : in STD_LOGIC;
    outputReady : out STD_LOGIC;
    done : out STD_LOGIC;
    stall : in STD_LOGIC;
    a : in STD_LOGIC_VECTOR(31 downto 0);
    b : in STD_LOGIC_VECTOR(31 downto 0);
    result : out STD_LOGIC_VECTOR(31 downto 0)
  );
end fp_div32;

architecture Behavioral of fp_div32 is

  component fp_div_gen32 IS
  port (
    a: IN std_logic_VECTOR(31 downto 0);
    b: IN std_logic_VECTOR(31 downto 0);
    operation_rfd: OUT std_logic;
    clk: IN std_logic;
    ce: IN std_logic;
    result: OUT std_logic_VECTOR(31 downto 0)
  );
  END component;

  signal inv_stall : STD_LOGIC;

begin
  inv_stall <= not stall; --when we need to stall, we just stop enabling the clock
  U0 : fp_div_gen32 port map ( a => a, b => b, clk => clk,
                               ce => inv_stall, result => result);
end Behavioral;

```

Figure 61: Wrapper for the Theoretical 32-bit Floating Point Divide

```

void SystemCode(int**A, int**B)
{
    int i ;
    int j ;
    int x ;

    x = 5 ; // Ignored
    for (i = 0 ; i < 10 ; ++i)
    {
        for(j = 0 ; j < 10; ++j)
        {
            B[i][j] = A[i][j] + x ; // Only statement translated into hardware
        }
    }
    x = B[9][9] ; // Ignored
}

```

Figure 62: System Code Sections Translated Into Hardware

6 Generated Specific Hardware Connections

Some optimizations may also create additional input scalars that do not appear in the C code. This section describes in detail how input and output scalar ports are derived from the written C code.

6.1 Basic Assumptions

When compiling systems, we only translate the body of the innermost loop after all loop unrolling has occurred into hardware. This means that any initialization or arbitrary code before or after the loop is ignored. For example, the code in Figure 62 will not translate the statements before or after the loop nest unless all loops are fully unrolled.

Input streams are identified as array read accesses. Output streams are identified by array write accesses. Arrays may not be both read and written to in the body of a loop except in the special case of generating a systolic array.

Input and output scalars must be passed in the parameter list. Local variables used in the innermost loop may also be identified as a feedback variable if the local variable has a read followed by a write. Feedback variables will add an input port to the generated hardware for the initial value.

Figure 63 provides an example of the assumptions we make based upon the C code. The interface we generate for this code is shown in Figure 64.

6.2 Values created by optimizations

The optimizations Temporal Common Subexpression Elimination (TCSE) and Systolic Array Generation also create input ports. TCSE will create a feedback variable and corresponding initialization port for each piece of code eliminated. Systolic Array generation will turn the original two dimensional array into a one dimensional array input (which corresponds to the first row of the two-dimensional array) and will create initialization input ports for every element in the first column of the original two-dimensional array.

```
void SystemCode(int* A,      // Input Stream
                int x,      // Input Scalar
                int endValue, // Input Scalar
                int& z,     // Output Scalar
                int* B,     // Output Stream
                )
{
    int i ;

    int y ; // Read before a write in the innermost loop,
            // is a feedback variable

    int internal ; // Written and then read in the innermost loop,
                  // identified as an internal register

    for (i = 0 ; i < endValue ; ++i)
    {
        y = y + 1 ;
        internal = y * 2 ;
        B[i] = A[i] + x + y + internal ;
        z = A[i+1] ;
    }
}
```

Figure 63: C Code That Infers Ports

```

entity SystemCode is
  port (
    -- Default signals
    clk      : in  STD_LOGIC ;
    rst      : in  STD_LOGIC ;
    inputReady : in  STD_LOGIC ;
    outputReady : out STD_LOGIC ;
    done     : out STD_LOGIC ;
    stall    : in  STD_LOGIC ;

    -- Input Stream signals
    A_WClk_in      : in  STD_LOGIC ;
    A_full_out     : out STD_LOGIC ;
    A_writeEn_in   : in  STD_LOGIC ;
    A_data_channel0_in : in  STD_LOGIC_VECTOR(31 downto 0) ;
    A_address_channel0_base_out : out STD_LOGIC_VECTOR(31 downto 0) ;
    A_address_channel0_count_out : out STD_LOGIC_VECTOR(31 downto 0) ;
    A_address_clk_in : in  STD_LOGIC ;
    A_address_rdy_out : out STD_LOGIC ;
    A_address_stall_in : in  STD_LOGIC ;

    -- Output Stream signals
    B_WClk_in      : in  STD_LOGIC ;
    B_full_out     : out STD_LOGIC ;
    B_writeEn_in   : in  STD_LOGIC ;
    B_data_channel0_in : in  STD_LOGIC_VECTOR(31 downto 0) ;
    B_address_channel0_base_out : out STD_LOGIC_VECTOR(31 downto 0) ;
    B_address_channel0_count_out : out STD_LOGIC_VECTOR(31 downto 0) ;
    B_address_clk_in : in  STD_LOGIC ;
    B_address_rdy_out : out STD_LOGIC ;
    B_address_stall_in : in  STD_LOGIC ;

    -- Feedback Initialization Scalars
    y_init_in : in STD_LOGIC_VECTOR(31 downto 0) ;

    -- Input Scalars
    x_in      : in STD_LOGIC_VECTOR(31 downto 0) ;
    endValue_in : in STD_LOGIC_VECTOR(31 downto 0) ;

    -- Output Scalars
    z_out : out STD_LOGIC_VECTOR(31 downto 0)
  ) ;
end SystemCode ;

```

Figure 64: Generated Ports

7 Examples Provided

Twenty different example codes are provided to demonstrate the current capabilities of ROCCC 2.0. These are located in the Examples subdirectory. The Examples subdirectory contains a directory with all of the Module examples and a directory with all of the System examples.

7.1 Module Examples

The Module examples are listed here:

- **BitonicSort2**
This module takes two scalar numbers and returns two sorted scalar numbers.
- **BitonicSort8**
This module takes eight scalar numbers and returns eight sorted scalar numbers. BitonicSort8 instantiates many BitonicSort2 modules in order to sort the inputs.
- **ColoumbicForceCalculations**
This example performs all of the Coloumbic force calculations between two atoms for one timestep of a molecular dynamics simulation. The calculations performed in this module require floating point values, so floating point cores must be supplied by the user in order to run the generated code.
- **FFT2**
The FFT2 module performs the base calculations for the fast Fourier transform between two complex numbers. The complex numbers are represented as two 64-bit values for the real and imaginary part.
- **FFT4**
The FFT4 module uses several instantiations of the FFT2 module in order to create 4 pairs of FFT base calculations happening in parallel.
- **FFT8**
The FFT8 module uses several instantiations of the FFT4 module and connects them in the butterfly configuration in order to perform the fast Fourier transform on eight input complex numbers and generate eight complex output numbers.
- **MaxFilter**
The Max Filter module takes three input integers and returns the maximum value amongst them.
- **SaturatingAdd**
This example module shows the usage of bit width by performing a saturating addition. The inputs and outputs to the system are eight bits long, but the internal calculations use a nine bit number in order to prevent overflow. If the result of the 8-bit addition is outside the range of an 8-bit number than the value returned from the module will be the maximum value of an eight bit number.
- **SingleSWCell**
This example performs the calculations necessary for a single cell of a wavefront algorithm like Smith-Waterman. This code can then be used as a module in a larger systolic array generation.

7.2 System Examples

The system examples are listed here:

- **IntegralImage**
This system calculates the integral image on a two-dimensional input stream. The integral image is the sum of all the previously seen elements that lie to the north and west of the current element. This is accomplished by instantiating a LUT and storing the accumulated values inside. The output of the system is a two-dimensional stream that represents the integral image.

- **IPIntegration**
This system demonstrates how to integrate external IP with ROCCC generated code. The multiply-accumulate core is not a ROCCC example and must be added into the ROCCC library through the GUI.
- **MatrixMultiplication**
The matrix multiplication system takes as input two NxN two-dimensional matrices and outputs an NxN matrix that is the product of the inputs. The code is three nested loops accessing the two dimensional arrays with different access patterns and shows some ways that ROCCC supports accessing array elements based upon loop indices.
- **MaxFilterSystem**
This example filters out the maximum value on each 3x3 window of a two-dimensional input stream. The module MaxFilter must be compiled before this example. If temporal common subexpression elimination is performed on this example, the amount of hardware generated will be reduced from 4 instances of MaxFilter to 2 instances of MaxFilter.
- **MedianFilter**
The MedianFilter example chooses the median value from each window of size 8 from a one-dimensional stream. In order to choose the median, each window must be sorted, which is accomplished through the BitonicSort8 module.
- **Prewitt**
This example performs Prewitt edge detection on a two-dimensional image by comparing each window with a set 3x3 filter. The output is a two-dimensional matrix that corresponds to a black and white image of just the edges in the original image.
- **SmithWaterman**
An implementation of the Smith-Waterman algorithm that can be compiled with the Systolic Array Generation optimization to create an efficient hardware solution.
- **Sobel**
This example performs the Sobel edge detection algorithm on a two-dimensional image by comparing each 3x3 window with a set 3x3 filter. The output is a two-dimensional matrix that corresponds to a black and white image of just the edges in the original image.
- **VectorAdd**
The VectorAdd example takes two streams of one-dimensional input and outputs a third stream that consists of the sum of the individual elements. This example can be used to demonstrate various stages of unrolling and stream widths.
- **VectorMatrixMultiplication**
This example performs the multiplication of a one-dimensional input stream to a two-dimensional input stream. The size of the streams should be 1xN and NxN, with the resulting output stream being a 1xN.
- **VectorReduction**
The VectorReduction code takes a one-dimensional input stream and outputs a single scalar output that consists of the sum of all the input elements.