Jacquard COMPUTING INC

ROCCC 2.0 Developer's Manual - Revision 0.7.6

May 24, 2013

Contents

1 Introduction

This document describes the current implementation of the Riverside Optimizing Compiler for Configurable Circuits (ROCCC) and describes the code structure and design of all of the transformations and processes that take a C file into VHDL. The information in this document is not needed in order to run and compile code with ROCCC, but is provided if you wish to improve or alter ROCCC to support some new constructs.

ROCCC is currently split up into several distinct phases which are implemented using different languages and toolsets. The compiler proper contains several intermediate representations, which we refer to as Hi-CIRRF and Lo-CIRRF (CIRRF stands for Compiler Intermediate Representation for Reconfigurable Fabrics).

The GUI is written in Java as Eclipse plugins and controls overall compilation and file management. The main plugin is called ROCCCPlugin and the ROCCC PCore Generator is an additional standard plugin. The structure and design of the GUI is described in Section 3. Ports for the Convey HC-1 and the Pico M501 are also implemented as additional Eclipse plugins, but are described in other documents.

The Hi-CIRRF passes are responsible for high level compiler transformations and are implemented using the SUIF (Stanford University Intermediate Format) toolset. SUIF is an older toolset and we have made some modifications to update it, including changing some of the data structures into standard library calls. The Hi-CIRRF passes that transform the compiled code and our additions to SUIF are described in Section 4.

The Hi-CIRRF passes communicate to the Lo-CIRRF passes through the creation of a hi_cirrf.c file. This file is a C file with extra macros and is described in Section 5. While it is possible to create a hi_cirrf file by hand, it is not recommended.

The Lo-CIRRF passes are responsible for hardware generation and pipelining as well as low level optimizations to reduce hardware and increase throughput. The Lo-CIRRF passes are implemented using the Low Level Virtual Machine (LLVM) toolset and are described in Section 6.

If you wish to add a transformation or optimization pass, it is important to place it in the correct location. Passes that transform the code in a traditional compiler sense or something similar should always be located on the Hi-CIRRF side. Any pass that effects the hardware generated in a direct capacity should take place at the Lo-CIRRF side.

ROCCC has several implementation languages, including C, C++, bash scripting, SQL, and Java. Both the Hi-CIRRF and Lo-CIRRF sections are mainly written in C++, and as such any changes made to the compiler require a working knowledge of C++. The GUI is written in Java under the Eclipse plugin framework.

2 ROCCC Organization and Distribution

2.1 Binary distribution

The current version of the GUI looks for executables in a local directory structure that is set up differently from the source distribution directory. In order to construct a binary distribution you must first build from source and then copy executables and libraries into the binary distribution directory structure.

2.2 Building the Binary Distribution From Source

The source distribution can be compiled using the included install bash scripts, by calling rocccInstall.sh from the directory it is located in. There are no parameters necessary to install ROCCC in the current directory.

2.2.1 Requirements

ROCCC has been built from source on 32-bit and 64-bit Ubuntu 12.04 systems, 32-bit and 64-bit CentOS 6.4 systems, and Macintosh OS X systems.

In order to build ROCCC on a base install of 64-bit Ubuntu 12.04, you must add the additional following tools: flex, bison, gcc, g++, autoconf, and make. Once these are installed, you must export the following environment variables by issuing these commands:

- export C_INCLUDE_PATH=/usr/include:/usr/include/x86_64-linux-gnu/
- export CPLUS_INCLUDE_PATH=/usr/include:/usr/include/x86_64-linux-gnu/
- export INCLUDE_PATH=/usr/include:/usr/include/x86_64-linux-gnu/
- export LIBRARY_PATH=/usr/lib:/usr/lib/x86_64-linux-gnu/
- export LD_LIBRARY_PATH=/usr/lib:/usr/lib/x86_64-linux-gnu/

On a 32-bit install of Ubuntu 12.04, replace x86_64-linux-gnu with i386-linux-gnu.

Compiling from source on a base CentOS 6.4 system and Macintosh OS X systems work with no additional steps.

2.3 ROCCC Components

The source distribution is broken into several components that make up the entire ROCCC system. These components are:

- A version of gcc-4.0.2 that has been slightly modified to output an abstract syntax tree that can be processed by ROCCC.
- A version of gcc-4.2 that generates llvm byte code that can be read by the llvm optimization program.
- An Eclipse plugin GUI, which includes the jar files for the Padlock license manager.
- The ROCCC compiler source code, which contains:
  - A modified version of SUIF (called NuSUIF) that generates the appropriate data structures for ROCCC's high level optimizations.
  - A set of optimization passes that use the SUIF libraries specific to ROCCC.
  - A set of helper tools used to initialize the database we are using and create intermediate scripts that call the appropriate passes for hi-cirrf compilation.
  - An unmodified version of sqlite-3.6.11 that is used to create and manage databases.
  - A modified version of llvm-2.3 used to perform low-level optimizations and generate hardware.

2.4 Install scripts

To compile from source, there are several scripts written in bash that will create the appropriate directories and compile the individual sections.

Compilation of ROCCC is performed by running the bash script file ”rocccInstall.sh.” This script calls several subscripts and performs the following tasks:

- Check Requirements

  ROCCC needs to find installed versions of flex, bison, gcc, g++, make, autoconf, grep, and patch in order to fully install. If any of these components are not found then installation halts.

- Install Modified gcc 4.0.2

  The Hi-CIRRF portion of ROCCC requires a file in the SUIF format. In order to generate a SUIF script, we created a program that translates gcc's GENERIC abstract syntax tree into SUIF, which is described in Section 4. The Modified gcc-4.0.2 installed during this step has some changes to the abstract syntax tree output to pass more information to the Hi-CIRRF passes (localized in the file ”tree-dump.c”) as well as some changes to help compilation on multiple machines.

- Unpacks a version of gcc for Lo-CIRRF

  The Lo-CIRRF portion of ROCCC requires a file in the LLVM binary format. in order to generate this binary, we use the prepackaged binaries available on the llvm website in addition to versions we precompiled for distribution. We determine the type of the host system and unpack the appropriate binary while removing all of the others.

- Compile the Hi-CIRRF section

  This step involves compiling both the SUIF distribution as well as all of the passes we have written.

- Compile Lo-CIRRF section

  This step involves compiling both the sqlite3 database we use as well as the LLVM code base.

- Initialize the Database

  At this point in the install script, a program to insert default floating point cores into the database is compiled and run. This program is located under roccc-compiler/src/tools.

- Create Scripts For Compilation

  Several scripts are created by the install script, including a script to perform the Lo-CIRRF compilation (compile_llvmtovhdl.sh), a script to reset the compiler database to its install state (reset-compiler.sh), a script that removes a single module from the roccc-library.h file and corresponding repository (remove_module.sh), a script that adds a single module from the roccc-library.h file and corresponding repository (add_module.sh), and the script that controls overall compilation (compile_hardware.sh). These scripts are unnecessary in the binary distribution and have been folded into the GUI source code.

3 GUI

The GUI is written in Java as an Eclipse plugin. It should be managed as an Eclipse project and developed in Eclipse as well. We also use sqlite3 as our database interface. The format of the database is described in Section 10.

The GUI is responsible for performing several tasks, each constructed with a handler/callback function. Tasks that require user input also have a wizard and wizard page implementation.

3.1 Tasks

The GUI performs the following tasks:

- Build
  Implemented in the CompilationPass.java file. Constructs the commands to call the high and low level compiler executables based upon the chosen optimizations.
- Cancel
  Implemented in the CancelCompile.java file. Calls the cancel script which searches for the executables the GUI spawns off and kills them using Unix commands.
- Manage Intrinsics
  Implemented in the ViewIntrinsicsWizardPage.java file. Interfaces with the sql database and adds the intrinsic information as supplied by the user.
- New Module
  Implemented in the CreateModuleWizardPage.java file. Creates the directory structure to add a module to a project and fills it with default code based upon information supplied by the user.
- New System
  Implemented in the CreateSystemWizardPage.java file. Creates the directory structure to add a system to a project and fills it with default code based upon information supplied by the user.
- New Project
  Implemented in the CreateProjectWizardPage.java file. Creates the directory structure for a new project. Projects are a convenient way to organize ROCCC source code.
- Add IP Core
  Implemented in the ShowIPCoresHandler.java file. Adds elements of a module that is not ROCCC code into the database for use in other applications.
- Import Module
  Implemented in the ImportModuleHandler.java file. Reads in a ROCCC module file that is not currently handled by any project and places it in the proper directory structure.
- Import System
  Implemented in the ImportSystemHandler.java file. Reads in a ROCCC system file that is not currently handled by any project and places it in the appropriate directory structure.
- View ROCCC Library
  Implemented in the ROCCC_IPCores.java file. Opens the view to the database that shows the modules available to be instantiated.
- View roccc-library.h
  Opens the internal file roccc-library.h with the default viewer and gives it focus.

- Debug View
  Implemented in the DebugVariables.java file. Opens a view that shows all the currently declared debug variables.
- Generate Testbench
  Implemented in the TestbenchGenerationPass.java file. Creates a testbench for the file active in default viewer.
- Reset Database
  Implemented in the ResetDatabase.java file. Calls unix commands to remove the current database file and replace it with a fresh copy. All interactions with the database must be locked from multithreaded access and the database must be closed after each interaction in order to maintain stability.
- Preferences
  Implemented in the PreferenceUtils.java file. Sets up global variables for the locations of the unix commands to call based upon user input.
- License information
  Implemented in the LicenseManagementUtils.java file. Utilizes the functions of Padlock in order to verify that the license file is for the given MAC address. If no MAC address is specified, then the license file is good on any machine.
- Load Examples
  Implemented in the ImportExamples.java file. Imports the examples from their default location into a ROCCC eclipse project and creates the appropriate directory structure.
- Check For Updates
  Implemented in the CheckForUpdates.java file. Connects to a web page on jacquardcomputing.com and checks if the version available for downloading is higher than the current version. The current version is stored in the database as well as a local ".VERSION" file and in the lo-cirrf version.h file.

4 Overall Compilation

The GUI selects optimizations and configurations and calls the compiler executables and associated tools with the correct parameters.

4.1 CreateScript

First, the GUI performs a call to the createScript application. This program creates a temporary bash script that will perform the high level optimizations using the SUIF passes and generate a hi-cirrf.c file. The parameters to the createScript application are a high level optimization file, a stream info file, a preferences file, and a debug values file, each of which is created by the GUI and passed to createScript.

4.1.1 High Level Compiler Optimization File

This file is created by the GUI and passed to createScript and specifies the number and order of the optimizations. As the compiler evolves, more optimizations will be added. The general layout of the compiler optimization file takes the form of:

```
Optimization [argument1][ [argument2]
Optimization [argument1][ [argument2]
...
```

All tokens are separated by white space and it is assumed that there is only one optimization per line. Some optimizations require no arguments while others might need one or two. The list of available optimizations with their arguments is:

- SystolicArrayGeneration [CLabel]
- TemporalCommonSubExpressionElimination
- LoopUnrolling [CLabel] [Amount]
- LoopInterchange [CLabel] [CLabel]
- LoopFusion
- MultiplyByConstElimination
- DivisonByConstElimination
- Export
- FullyUnroll
- Redundancy [CLabel] [Type]
- InlineModule [ModuleName]

CLabel must be the names of labels specified in the original C file. Amount should be a positive integer number or the word "FULLY." Type should be either the word "DOUBLE" or "TRIPLE." ModuleName should be the name of a module instantiated in the C code.

4.1.2 Stream Info File

This file is created by the GUI and passed to the createScript program, it describes all of the aspects of the streams. The format is as follows:

```
INPUT Name NumChannels NumRequests MaximizeThroughput
OUTPUT Name NumChannels
```

Name is the original name of the stream in C. NumChannels is the number of incoming or outgoing data channels. NumRequests is the number of outstanding memory requests that can be generated. MaximizeThroughput if set to be 1 will print out address channels for each incoming data channel. If set to 0, then there will only be one address channel. If five incoming data channels are specified but only one address channel is specified, then we still must generate five addresses at one per cycle before we can read any data. Currently, this value is always set.

4.1.3 Preference File

```
COMPILER_VERSION Number
```

There is currently only one global preference that is true for every compilation, and that is the version of the compiler being used. The number can be a string in the form of #.#.# or just an integer.

4.1.4 Debug File

This file contains lines that have three values: the name of a variable that is identified as a debug register, a 0 or 1 indicating if a watchpoint is active, and an integer value the corresponds to the watchpoint.

A debug register will include a port that leads directly outside the circuit to the highest level and can be used to monitor the cycle by cycle value of a certain variable. A debug register with a watchpoint will halt execution when the value is identical to the watchpoint.

4.2 Low level compilation

Once the hi-cirrf.c file has been created, the temporary script is deleted and the LLVM compiler front end is called, translating the hi-cirrf.c file into the LLVM intermediate format. Low level optimizations written as LLVM passes generate hardware based upon the low level optimizations chosen in the GUI.

After the low level optimization passes have finished, a local directory named ”vhdl” will be created and all of the generated hardware files will be placed in that directory.

4.2.1 Timing Information File

The Lo-CIRRF passes expect a specific file called ”*/.ROCCC/.timinginfo” where ”*” is the folder of the file being compiled. This file is laid out in the following way:

```
Operation Weight
Operation Weight
...
```

The specific operations that need to be specified are the following: ”Add”, ”Sub”, ”Mult”, ”Div”, ”Compare”, ”Mux”, ”Copy”, ”Shift”, ”AND”, ”OR”, ”XOR”, and ”DesiredTiming”. All weights must be positive integers. The weight for ”DesiredTiming” must be greater than or equal to any other weights.

This timing information is used by the low level optimizer to determine how many instructions are placed in each pipeline stage and provides a tradeoff between clock speed and area.

5 Gcc2SUIF

In the ROCCC distribution there is a bash script called ”gcc2suif” and a directory called ”gcc2suif” with code that compiles into a program called ”parser.” The script does some cleanup on generated files and calls ”parser,” so the tool described in this section refers to the code implementation in the gcc2suif directory.

The code used in converting gcc into suif can be viewed as a standard compiler in its own right. The input language is the abstract syntax tree of GCC as output by the –dump-tree-original-raw flag, the implementation language is C++, and the target language is SUIF. In order to accomplish this, gcc2suif uses flex and bison to describe the language of the abstract syntax tree and directly creates suif code using the suif toolset. Not every possible node output by gcc is supported.

The code is structured in hierarchical polymorphic classes, with the topmost parent class ”Node” representing a single entry in the original AST as output by gcc. Each possible type of line output by gcc is given its own class, descended from Node and organized based upon a common type, such as ”expression”, ”type”, or ”statement” nodes. Each line in the original gcc AST has a unique identifier, which we refer to as the NodeNumber.

The internal parser tree is constructed in the code created by bison, and once constructed goes through three phases: connecting, flattening, and generating suif.

The connecting phase consists of iterating through every Node and making explicit connections to the appropriate parent and children based upon the NodeNumbers. Since the AST as described by gcc typically refers to nodes before they have been constructed, this must be done after the whole AST has been processed.

The flattening phase removes Option indirection. Every Node in the AST has a number of optional parameters which point to other sections of the tree. When constructing the data structures there is a level of indirection that identifies first if an option exists and then what that option is. During flattening, all options that exist are turned into concrete pointers to the object they refer to and options that do not exist are invalidated.

Once the tree has been connected and flattened, it is recursively traversed and a suif format tree is generated. During this stage all variables are assigned a scope, which is maintained by a ScopeTree structure filled with symbol tables.

6 Hi-CIRRF Compilation

6.1 Hi-CIRRF Passes

All Hi-CIRRF passes are located in a subdirectory under roccc-compiler/src/hi_cirrf_pass/basepasses. Each pass is responsible for one transformation of the C code.

- EvalTransformPass

  Location: global_transforms

  Purpose: Replace all SUIF evaluation statements generated by the gcc2suif tool and replace them with an appropriate Call Statement, Store Statement, or Store Variable Statement.

  This pass exists to because the structure of the abstract syntax tree generated by gcc might produce evaluation statements for call expressions that do not return a value when this is handled by a Call Statement with a NULL destination in SUIF.

- ForLoopPreprocessingPass

  Location: gcc_preprocessing_transforms

  Purpose: Find all for loops and move the statement directly before the for loop into the for loop structure itself.

  This pass exists because of the different way that gcc and SUIF handle for loops. SUIF places the first statement of the for loop as a child of the for loop node, gcc moves the statement before the for loop and does not treat it as a child of the for loop.

- FlattenStatementListsPass

  Location: utility_transforms

  Purpose: Removes blank statement lists and compresses statement lists that contain statement lists into one list.

  Several of our passes create empty statement lists or statement lists that contain other statement lists. This pass is responsible for flattening them out into sensible statement lists.

- NormalizeStatementListsPass

  Location: utility_transforms

  Purpose: Makes sure that every for loop and every if statement has a statement list as a body and not just a single statement.

  In a lot of the ROCCC passes we make certain assumptions about the structure of the code we are processing. One of those is that every loop and if statement will contain a statement list and not a singular statement as its body. This pass makes sure that the program is transformed into such a style.

- InliningPass

  Location: global_transforms

  Purpose: Inlines specific function calls.

  When compiling ROCCC code that uses modules, the modules are treated as black boxes and instantiated into the generated pipeline. If inlined, the internal expressions of the corresponding C code are exposed to more optimizations and results in different hardware, with no black box instantiation.

- MarkRedundantPass

  Location: global_transforms

  Purpose: Mark all call statements that were specified as redundant with the proper type of redundancy, double or triple.

Creating redundant code is a three step process that first involves finding the locations of the modules and marking them for a later pass to deal with.

- RedundancyPass

  Location: global_transforms

  Purpose: Creates duplicate instances of module calls and instantiates voters to manage the outputs.

- RedundantToRedundantPass

  Location: global_transforms

  Purpose: When data from a redundant module flows into another redundant module, a more robust circuit can be created if the voters are redundant as well. This pass creates redundant voters for these cases.

- PointerConversionPass

  Location: global_transforms

  Purpose: Turn all pointer accesses into array references.

  Generic pointers are unsupported in ROCCC, but pointers are supported as long as they act as array references. This pass transforms any pointer usage into an equivalent array access.

- InterchangePass

  Location: loop_transforms

  Purpose: Perform loop interchange on two loops.

  This switches the loop induction variables on two loops. This is useful when accessing two dimensional arrays is accessed in row-major order but stored in column-major order or vice versa. This pass takes two arguments, which are the labels associated with the loops to be interchanged.

- UnrollPass2

  Location: loop_transforms

  Purpose: Unrolls a given loop by a given amount. Does not create other loops to handle excess loop iterations.

  This pass takes two arguments, the label associated with the loop that we wish to unroll and the number of times to unroll the loop. If the number of times to unroll is greater than the end limit of the loop we fully unroll. This pass assumes that all loops start at 0 and count up to some number in the condition, which is checked with a less than comparison.

- HandleCallStatements

  Location: loop_transforms

  Purpose: When loops are unrolled, all call statements that have outputs need to be replicated. This pass takes care of that.

  Code that is passed to LLVM is translated into single static assignment form. To the LLVM side, multiple uses of the same variable are not transformed into multiple definitions as we would like, causing problems. This pass makes sure that if a variable is written to from two distinct call statements these values are split into distinct variables so they cause no conflict when passed to LLVM.

- HandleCopyStatements

  Location: loop_transforms

  Purpose: When loops are unrolled, copy statements require things to be put in a sort of SSA state.

  This pass performs a very similar operation for store variable statements. All variables that have multiple definitions (not if statements) are split into distinct variables. This is mainly a problem thanks to loop unrolling.

- IdentificationPass

  Location: verifyPass

  Purpose: Analyzes the code and determines if we are compiling a module, system, or composite system.

  Due to previous transformations, the type determined by the IdentificationPass might not be what the code started out as (i.e. Systems transformed into Modules through full loop unrolling).

- ControlFlowSolvePass

  Location: control_flow_analysis

  Purpose: Perform control flow analysis on the graph.

  This pass annotates the SUIF graph with information regarding control flow information. These annotations must be remade after any transformation that duplicates or rearranges code.

- DataFlowSolvePass2

  Location: bit_vector_data_flow_analysis

  Purpose: Perform data flow analysis on the graph and treats call statements as modules correctly.

  This pass annotates the SUIF graph with information regarding data flow. The control flow solve pass must be run before this pass. The information gathered in this pass must be redone after any transformation that duplicates or rearranges code.

- UD_DU_ChainBuilderPass2

  Location: bit_vector_data_flow_analysis

  Purpose: Sets up the use-def and def-use chains between all uses and definitions. All uses are in LoadVariableExpressions and all definitions are in StoreVariableStatements and CallStatements.

  All use/definition chains and definition/use chains are stored as annotations on both the variable symbols and the statements themselves. The data flow solver must be run before this pass. Any information gathered in this pass must be redone after any transformation that duplicates or rearranges code.

- ConstantPropagationAndFoldingPass2

  Location: global_transforms

  Purpose: Performs constant propagation and folding.

  This pass does not propagate or fold any constant value that is located inside of a constant array.

- ConstantArrayPropagationPass

  Location: array_transforms

  Purpose: Propagates constant values that are in constant qualified arrays.

  This pass does not perform any constant folding.

- TransformUnrolledArraysPass

  Location: fifoIdentification

  Purpose: When loops have been unrolled fully, arrays need to be changed from array access into either scalars or array accesses of lower dimension.

  This pass must be called after constant qualified array propagation.

- AvailableCodeEliminationPass

  Location: global_transforms

  Purpose: Removes available code and reduces the size of the generated circuit when applicable.

- LoopInfoPass

  Location: control_flow_analysis

  Purpose: Annotate all of the for loops with the nesting level and any associated label.

- PreprocessingPass

  Location: data_dependence_analysis

  Purpose: Annotates all of the array references with the associated indices.

- ScalarReplacementPass

  Location: array_tranforms

  Purpose: Performs scalar replacement. All array accesses are replaced with scalars.

- IfConversionPass2

  Location: global_transforms

  Purpose: Transforms if statements into boolean select statements.

  This pass converts if statements with exactly one assignment to the same location in both the "then" and "else" portion into a boolean select statement. This pass handles struct accesses and direct stores to memory for ROCCC 2.0 compatability.

- PredicationPass

  Location: global_transforms

  Purpose: Convert arbitrary if statements into predicated statements.

  The only type of if statement ROCCC 2.0 supports in hardware is the boolean select variety, where one variable gets assigned one of two different values based upon a condition. This pass converts arbitrary if statements into a sequence of predicated statements that fit the boolean select pattern.

- SummationPass

  Location: global_transforms

  Purpose: Convert statements that accumulate an integer into a special call that gets transformed into optimized accumulation hardware in the generated circuit.

- PseudoSSA

  Location: loop_transforms

  Purpose: Convert the body of the innermost loop into a single static assignment format and detect all possible feedbacks.

  This pass combines the tasks done in the HandleCopyStatements, HandleCallStatements, SolveFeedbackVariables, and SolveFeedbackCalls into one integrated pass.

- SolveFeedbackCalls

  Location: bit_vector_data_flow_analysis

  Purpose: Detect and split variables that act as feedback in all call statements.

- SolveFeedbackVaraiblesPass3

  Location: bit_vector_data_flow_analysis

  Purpose: Identify all of the variables that are read before they are written in the innermost loop.

  Variables are identified as feedback if they are read before they are written in the innermost loop and they do not have a definition before any uses.

- OutputIdentificationPass

  Location: global_transforms

  Purpose: Identify scalars that are either input or output. Input scalars are variables that are read before they are written in the innermost loop. Output scalars are variables that are written in the innermost loop that have no subsequent use (and are not used as feedback).

- FifoIdentification

  Location: fifoIdentification

  Purpose: Changes fifos into stream functions for the hi-cirrf generation.

- TransformSystemsToModules

  Location: global_transforms

  Purpose: When a system has all of its loops fully unrolled, no streams exist and all input and output values turn into scalars. The resulting code is treated as a module and compiled as such. This pass creates and fills up the struct that gets passed and returned.

- VerifyPass

  Location: verifyPass

  Purpose: Identifies if we are compiling a system or a module and verifies that all of the code was written correctly for the type of function.

- OutputPass

  Location: jasonOutputPass

  Purpose: Output the "hi_cirrf.c" and "roccc.h" files. This performs the actual output of the C code based upon the tree. After this pass all of the information we have collected is discarded.

- StripAnnotesPass

  Location: utility_transforms

  Purpose: Remove all of the annotations that we added to the suif graph.

- LibraryOutputPass

  Location: libraryOutputPass

  Purpose: Update the suif repository file and the "roccc-library.h" file.

- RemoveLoopLabelLocStmtsPass

  Location: utility_transforms

  Purpose: Removes labels from loops. This is necessary when unrolling an outer loop that contains an inner loop with a label and then trying to fuse all of the created loops.

- FusePass

  Location: loop_transforms

  Purpose: Fuse loops that have the same bounds and no dependencies.

- MEM_DP_BoundaryMarkPass

  Location: utility_transforms

  Purpose: Adds a mark statement between the hoisted memory reads and writes from the datapath proper.

- RawEliminationPass

  Location: array_transforms

  Purpose: Removes reads after writes.

- ScalarRenamingPass2

  Location: global_transforms

  Purpose: It eliminates all Anti and Output dependencies between scalar variables by renaming scalar variables.

- CopyPropagationPass2

  Location: global_transforms

  Purpose: Performs copy propagation that is compatible with ROCCC 2.0.

- SolveFeedbackVariablesPass2

  Location: bit_vector_data_flow_analysis

  Purpose: Deprecated pass that identifies feedback variables.

- FifoIdentifiactionSystolicArray

  Location: fifoIdentification

  Purpose: Creates fifos for systolic array generation. This involves splitting the two dimensional array into two separate one dimensional arrays, one for input and one for output.

- RemoveNonPrintablePass

  Location: jasonOutputPass

  Purpose: Finds all statements that have been previously annotated as "NonPrintable" and removes them. Some statements are not removed in certain passes but should not be output to the final hi_cirrf file. These are given the annotation "NonPrintable."

- LeftoverRemovalPass

  Location: global_transforms

  Purpose: Go through and remove the extraneous information generated during systolic array generation that might cause problems if passed to llvm. This includes all of the statements outside of the loops that were generated during an intermediate step.

- TemporalCSEPass

  Location: global_transforms

  Purpose: Perform temporal common subexpression elimination.

- RemoveUnusedVariables

  Location: global_transforms

  Purpose: To remove all of the variables that are no longer used in the resulting hi-cirrf file that is output from the symbol table.

- CleanupStoreStatementsPass

  Location: global_transforms

  Purpose: Change store statements that store into a load variable expression into a store variable statement.

  When arrays have been changed into scalars, we just replace all array reference expressions into load variable expressions. This is fine for most nodes, but store statements must be changed into store variable statements where appropriate.

- CleanupUnrolledCalls

  Location: loop_transforms

  Purpose: Create duplicate variables for destinations of call statements that have been unrolled.

- ReferenceCleanupPass

  Location: global_transforms

  Purpose: Converts reference variables into symbol addresses.

  This pass is required for converting reference variables into symbol address expressions. This is necessary when calling modules in the new way (without structs) and connecting output variables of the function to output variables of instantiated modules (inlined or not).

- CleanupRedundantVotes

  Location: global_transforms

  Purpose: This performs a little cleanup and makes sure that all voters have unique error variables assigned to them.

- CleanupBoolSelsPass

  Location: global_transforms

  Purpose: Handle variables that have an undefined path through boolean selects.

  When performing predication, variables may be assigned a value when in the original C code they would be undefined. We fill in these undefined paths with the constant 0 to avoid generating incorrect hardware.

- PreferencePass

  Location: jasonOutputPass

  Purpose: Process the preference file to control some optimization strategies and pass additional information to the lo-cirrf side.

- CastPass

  Location: jasonOutputPass

  Purpose: Locate all automatic casts between different types (including different bit sizes of integers) and create explicit calls so the lo-cirrf side has the correct information.

- IdentifyDebugPass

  Location: jasonOutputPass

  Purpose: Identify which variables correspond to variables marked for debugging in the GUI and pass that information on to the lo-cirrf side.

  This pass is not yet completed but is essential in the addition of debug variables to the overall ROCCC compilation.

7 Hi-CIRRF File Format

Although you generally should not write Hi-CIRRF files directly, we describe in this section all of the statements that will appear in a Hi-CIRRF file.

The Hi-CIRRF that we generate consists of two files, roccc.h and hi_cirrf.c. The roccc.h file contains all of the declarations of the functions we use, although there is no corresponding definition of any of these functions. Hi-CIRRF itself is ANSI C with additional function calls that have no definition. Expressions and statements appear in hi-cirrf nearly identically to the original C after it has been transformed.

7.1 Fakes

Directly after the declarations in the Hi-CIRRF file are load instructions for most declared variables that we refer to in the code as "Fakes." These load instructions are in the form of "X = *((int*)(0)) ; " These instructions are necessary as the Hi-CIRRF code is not necessarily runnable C code, but rather an internal representation that we expect. When translating Hi-CIRRF into the LLVM binary format, any variable that does not have a definition before a use will be translated into an undefined reference. Because of this we make sure to give every variable a definition, even if it is meaningless.

7.2 Functions

There are several functions that we call in the Hi-CIRRF code that have no definition but do have a specialized meaning to the ROCCC compiler. Almost all of them exist to pass information discovered in the high level transformations directly to the code generation portion of the compiler. These functions are as follows:

- ROCCCCompilerVersion(const char*)

  This function passes the version of the compiler that was used to compile the current file to the low end. This is later stored in the database and used to determine if code is out of date or current and usable.

- ROCCC_loadPrevious(int, int)

  When a variable is identified as a feedback variable, meaning the value calculated at the end of one iteration is used in the next iteration, we pass that value back through a temporary feedback register. A call to load previous tells the Lo-CIRRF passes what variable to associate with the previous iterations value. All loadPrevious calls also imply an initial value that is passed in as a scalar to be used before any values have been calculated.

- ROCCC_storeNext(int, int)

  The second half of feedback identification, a storeNext call will store the value calculated in one iteration into a variable for use in future iterations. All storeNexts happen at the end of the loop iteration at the same time.

- ROCCC_systolicNext(int, int)

  A call to systolicNext has the same function as a storeNext call, with the exception that the systolicNext happens when the data is ready and doesn't get updated at the end of a loop iteration.

- ROCCCSystolicNextInit

  This passes an initial value to the low level optimization passes for any value that has feedback. If this function is called, no input lines for the feedback variables are created.

- ROCCCSystolicPrevious(int, int)

  A call to SystolicPrevious is the matching load that happens at the beginning of the loop for systolic array generation.

- ROCCCFeedbackScalar

  This function identifies all of the variables that are feedback variables and need to be treated as such in the low level transformations.

- ROCCCInputStreams

  This function lists all of the identified input streams in the proper order so that the low level transformations can work correctly.

- ROCCCOutputStreams

  Similar to the ROCCCInputStreams function, this function lists and orders all arrays identified as output streams so the low level can function correctly.

- int ROCCCInfinity()

  ROCCC supports infinite loops, but llvm expects for loops to appear in a very rigid format with a starting value of 0 and a step of 1. The end value for a loop must also be an integer, so when the high level transformations detect an infinite loop a call to the function ROCCCInfinity is output in place of an end value.

- float ROCCCFPToFP(float, int)

  When converting from a floating point value to a floating point value of a different bit width either through automatic or explicit casting, we output a call to a conversion function to instantiate specific conversion hardware. The second parameter is the bit size of the destination.

- int ROCCCFPToInt(float, int)

  When converting from a floating point value to an integer either through automatic or explicit casting we output a call to a conversion function to instantiate specific conversion hardware. The second parameter is the bit size of the destination.

- float ROCCCIntToFP(int, int)

  When converting from an integer to a floating point value either through automatic or explicit casting we output a call to a conversion function to instantiate specific conversion hardware. The second parameter is the bit size of the destination.

- int ROCCCIntToInt(int, int)

  When converting from an integer of one bit size to an integer of another bit size we output a call to a conversion function. On the lo-cirrf side this corresponds either to a sign extension, zero extension, or truncation. The second parameter passed is the bit size of the destination.

- ROCCCDoubleVote(int, ...)

  This function instantiates a voter of the appropriate type for double redundancy.

- ROCCCTripleVote(int, ...)

  This function instantiates a voter of the appropriate type for triple redundancy.

- ROCCC_output_C_scalar

  The high level optimizations detect all scalar values that are to be output once at the end of execution. These are explicitly identified in modules as variables in the struct with the suffix "_out" and inferred from live variables in systems. All variables that are output scalars are listed in a call to the function ROCCC_output_C_scalar in no particular order and used by the lo-cirrf side.

- ROCCC_init_inputscalar

  Similar to output scalars, the high level optimizations detect all scalar values that are read once at the beginning of execution. These are explicitly identified in modules as variables in the struct with the suffix "_in" and inferred in systems as all variables that are read but not written in the body of the innermost loop. All identified input variables are listed in no particular order in a function call to ROCCC_init_inputscalar.

- ROCCCName(const char*, ...)

  Every variable gets changed by LLVM into a temporary name. Passing the name as a string allows LLVM to find the original name and store it away.

- ROCCCSigned(int, ...)

  This function informs the low level transformations whether a variable is signed or unsigned.

- ROCCCModuleStructName(const char*)

  For generating a PCore, the GUI must know the name of the struct that is created with modules. The way we get that information back is through the database. The Hi-CIRRF section does not modify the database, so we pass the information along to the Lo-CIRRF side in this function.

- ROCCCPortOrder(const char*)

  The order of the ports in the structs for modules get transformed by the gcc abstract syntax tree and may not be in the same order as when they were defined in the original C. The Hi-CIRRF takes the port order that it knows as one big string and passes it to the LLVM side. The LLVM side then puts this information in the database so the GUI may access this to ease use of module instantiation.

- ROCCCNumAddressChannels(int, ...)

  This specifies the number of address channels to be created for each individual input stream that generate an address each clock cycle. This number cannot be larger than the number of incoming data channels.

- ROCCCInvokeHardware(const char*, ...)

  Every call to a hardware block is done by passing the name of the IP core as the first element of the InvokeHardware call. If this is not found in the database during the lo-cirrf compilation, an error will be raised. The values passed after the name consist of all of the inputs to the module followed by all of the outputs to the module.

- ROCCCDebugScalarOutput

  This function will specify that certain variables are debug variables and should have direct output lines to the outside world. This feature is currently in development and not completely supported.

- ROCCC_boolsel(int, int, int)

  If statements in the C code are transformed into calls to boolean select hardware (muxes) at the VHDL level. The actual transformation occurs at the hi-cirrf level so when the lo-cirrf manages the code there is no control flow resulting in phi nodes when dealing with SSA form.

- ROCCCSize(int, ...)

  Every variable has a bit width associated with it. If no bit width was explicitly specified then this value is the default of the machine compiling on. This function is used to pass the bit width information of each individual variable to the lo-cirrf passes.

- ROCCCStep(int, int)

  In order for LLVM to identify a for loop and collect information about that loop correctly, the for loop must have a step size of one. ROCCC supports any step size, so the hi-cirrf for loops are output with a step size of one and the actual step size is passed to the lo cirrf side through this function, which is called once per loop index.

- ROCCCOutputFifoX(int, ...)

  Any array we have identified as being written to is passed to the lo cirrf side in this function. The array should have had all of its memory accesses replaced with scalars. This function sends the original array name, the indices that all accesses are based off of, and the scalars that correspond to different offsets. The X is either 1 or 2, based upon the dimensionality of the array being passed.

- ROCCCInputFifoX(int, ...)

  Nearly identical to the output fifo call, this function passes information about any arrays identified as input arrays.

- ROCCCInputSBX(int, ...)

  This is identical to the output fifo call with the exception that there should be reuse between different iterations.

- ROCCCFunctionType(int)

  This passes 1 if we are compiling a module and 2 if we are compiling a system.

- ROCCC_state_scalar()

  This function is currently output but ignored and has no function.

- ROCCCNumDataChannels(int, ...)

  This is called for every input and output stream and specifies the number of simultaneous incoming or outgoing words that the stream expects every clock cycle.

- ROCCCNumMemReq(int, ...)

  This is called for every input stream and identifies the number of outstanding memory requests supported.

- ROCCCMaximizePrecision(int)

  This passes 1 to the low level optimizations to indicate if all arithmetic operations should increase their precision and round only on assignment or passes 0 to indicate rounding at every step.

8 Lo-CIRRF Passes

All lo-cirrf passes are located under the directory roccc-compiler/src/llvm-2.3.

8.1 LLVM Changes

LLVM was modified slightly to allow ROCCC to built on top of the framework that was already in place in LLVM. Specifically, the Function class and BasicBlock class were modified to allow two classes to each derive from them, DFFunction and DFBasicBlock respectively. These two classes contain additional ROCCC specific information, and are used to construct the ROCCC DFG using the LLVM CFG framework. By extending these two classes, it is also possible to detect when a given Function or BasicBlock object actually represents a ROCCC DFG construct, and not an LLVM CFG construct.

Because of linking issues with the version of gcc we use to compile LLVM, DFFunctions and DFBasicBlocks do not retain their runtime type information across module boundaries. To fix this, we had to add a single function to each of Function and BasicBlock that returned a DFFunction pointer or a DFBasicBlock pointer if the base object was actually of that type. Instead of dynamic_casting a Function or BasicBlock object to a DFFunction or DFBasicBlock, it is necessary to call the getDFFunction() or getDFBasicBlock() in the base class.

8.2 Passes

All of the Lo-CIRRF passes are implemented as a class derived from ModulePass or FunctionPass and are called from the program "opt". The passes we call, in order, are as follows:

- maximizePrecision

  Location: lib/Transforms/RocccCFGtoDFG/MaximizePrecision.cpp

  Purpose: Calling this pass tells the low level that all integer operations should use the maximum precision possible, truncating as a final step. Whether or not to call this pass is handled by the compile_llvmtovhdl.sh script.

- renameMem

  Location: lib/Transforms/RocccCFGtoDFG/RocccCFGtoDFG.cpp

  Purpose: Renames each load instruction to be a derivative of the first operand. This is not strictly necessary, but makes the resulting VHDL more readable.

- mem2reg

  Location: Builtin

  Purpose: LLVM by default places all C variables into a memory location and need to be explicitly transformed into register uses. This pass performs this transformation and lumps all values without an initialization together as undefined.

- removeExtends

  Location: lib/Transforms/RocccCFGtoDFG/RemoveExtends.cpp

  Purpose: The low level can handle truncation and extension of integers to different sizes without the use of explicit extension operations. This pass removes any explicit extend operations, so as to minimize the overhead of unnecessary operations.

- ROCCCfloat

  Location: lib/Transforms/FloatPass/FloatPass.cpp

Purpose: Floating point operations and integer divides are supported in the VHDL through IP cores. However, to support naturally written C code, these are written as C operators and have to be transformed into calls to an IP core. The FloatPass searches for instructions that are either integer divides or floating point operations, then replaces them with a call to the correct IP core from the database.

- flattenOperations

  Location: lib/Transforms/RocccCFGtoDFG/FlattenOperations.cpp

  Purpose: This optional pass performs tree balancing on commutative and associative operations, namely addition and multiplication, to perform as many possible operations in parallel, rather than serially.

- dce

  Location: Builtin

  Purpose: After flattening operations, dead code is generated. This pass eliminates that dead code, and cleans up for later passes.

- undefDetect

  Location: lib/Transforms/RocccCFGtoDFG/UndefDetectionPass.cpp

  Purpose: After running the mem2reg pass any variables that were declared in C code that have uses before any definition are referenced by the variable "undef." This pass searches all instructions and looks for any instance of an undefined variable and errors out if any are detected. If all goes well, this pass does not modify the code in any way.

- functionVerify

  Location: lib/Transforms/RocccCFGtoDFG/FunctionNameVerifyPass.cpp

  Purpose: This pass looks through the instructions and checks that any call instructions adhere to three rules: no call instructions are allowed except for ROCCC specific call instructions, all calls to ROCCCInvokeHardware must exist in the database as an IP core, and all calls to ROCCCInvokeHardware must have the same number of arguments as the IP core has ports.

- rocccCFGtoDFG

  Location: lib/Transforms/RocccCFGtoDFG/RocccCFGtoDFG.cpp

  Purpose: This pass has several duties, all of which are related to transforming the LLVM-created control flow graph into a data flow graph with ROCCC specific additions. It requires two sub-passes, rocccFunctionInfo and rocccIfCompact, which are responsible for detecting and saving loop-specific information and for compacting the boolSelect calls, respectively. The ROCCC DFG uses one BasicBlock per instruction and connects the BasicBlocks to one another using switch statements. All of the ROCCC inputs, such as input FIFOs, loadPrevious calls, and input scalars are pulled to the top of the DFG and are direct successors of an empty BasicBlock called the sourceHead. All ROCCC outputs, such as output FIFOs, storeNext calls, and output scalars are pulled to the bottom of the DFG and are direct predecessors of an empty BasicBlock called the sinkHead.

  The ROCCC DFG is not a direct translation of the already-existing LLVM DFG ; several functions, such as ROCCCInputScalar are uses in LLVM, but are considered definitions in the ROCCC DFG.

  When finished, the resulting DFG should have a source at the top of the graph and a sink at the bottom of the graph, and then each internal node corresponds to an element in the datapath. The depth of each node from the source can specify which pipeline stage to place the hardware in.

- detectLoops

  Location: lib/Transforms/PipelinePass/DetectLoopsPass.cpp

Purpose: Most of the later passes assume that the graph we create is acyclic, so this pass makes sure that no loops exist and errors out if there are any loops. This pass is a prerequisite for pipeline numbering, retiming, and inserting copies.

- pipeline

  Location: lib/Transforms/PipelinePass/RetimingPass.cpp

  Purpose: Up to this point, the pipeline separates all dependent instructions into their own pipeline stage. The pipelining pass combines instructions into a single pipeline level, allowing for a shallower pipeline at the potential cost of frequency. This is often a desirable optimization as there are usually other constraints that limit the frequency and so a shallower pipeline can reduce area and latency. This is a renumbering algorithm and does not reshape the DFG.

  The algorithm is based off of the FEAS algorithm presented in ”Retiming Synchronous Circuitry” by Charles E. Leiserson and James B. Saxe.

  We first convert the ROCCC DFG into a more suitable, lightweight form to perform the retiming calculations on. We construct a graph in which both the nodes and edges have an associated weight. The weight of each node is the delay of that node, which is user configurable. The weight of each edge is initially set to 0 and is used to represent the number of registers at that edge. The algorithm inserts registers until the period of the graph is less than or equal to the desired period.

- minimizeCopies

  Location: lib/Transforms/PipelinePass/CopyMinimization.cpp

  Purpose: This optional pass attempts to minimize the total number of register copies created, by moving the pipeline level that operations are performed in.

- insertCopy

  Location: lib/Transforms/PipelinePass/InsertCopyPass.cpp

  Purpose: For each edge in the DFG, if the difference in pipeline level is greater than 1 along that edge, copies need to be inserted to maintain the value across pipeline stages. The InsertCopyPass first calls the AnalyzeCopyPass, which goes through and sees which values need to be copied at each pipeline level. Multiple copies of the same value at the same pipeline level are combined into the same copy. Then, the InsertCopyPass creates a copy block and a copy instruction for each value that needs to be copied at each pipeline level, and replaces each instruction that uses that copy value as an operand with the newly created copy value.

- arrayNorm

  Location: lib/Transforms/VHDLOutputPass/ArrayAccessNormalization.cpp

  Purpose: Input arrays in ROCCC offsets based off of an index plus a negative number. However, the InputControllerPass which outputs the FSM for handling all of the input streams, input scalars, and load previous values expects streams to be normalized greater than zero. This pass looks at each input stream, and if the stream has any indices in the form [i-n] with i being the loop induction variable and n being a constant int, then the largest value n found is added to all of the indices that use that loop induction variable.

- vhdl

  Location: lib/Transforms/VHDLOutputPass/VHDLOutputPass.cpp

  Purpose: This pass encompasses three difference passes: InputControllerPass, OutputControllerPass, and VHDLOutputPass. Each pass uses a common VHDL library to construct a representation of the final VHDL purely in memory and then outputs the VHDL after it has been checked for basic syntactical correctness. A loopController object is used to manage the iteration of the loop induction variables;

this is used by the inputController and outputController to manage accesses to stream elements. The inputController takes all input scalars and input streams, and makes sure that they are pushed onto the datapath simultaneously. The outputController serializes the output streams, as well as manages when the overall component completes processing.

- componentInfo

  Location: lib/Transforms/RocccCFGtoDFG/ComponentInformationPass.cpp

  Purpose: The database contains information on each system and module compiled. This pass writes that information to the database. Specifically, for modules it writes the delay of the component, the original struct name use, and the port ordering necessary to call the module in C. There is currently no system information written to the database in this pass, although area and frequency estimations will be written to the database here.

- printPortNames

  Location: lib/Transforms/VHDLOutputPass/PortNamePrinter.cpp

  Purpose: This pass writes the list of ports for each system and module to the database. Ports are organized as either stream or register ports. If they are register ports, the direction, size, original C name, and VHDL name are written to the database. For stream ports, the direction size, original C name of the stream, and VHDL name of each component of the stream are written to the database. Streams have the following components - any number of data channels, a 1-bit valid port, a 1-bit pop port, a 32-bit address port, and a 1-bit address ready port. For each component that belongs to the same stream, the C name will be the same name as the name of the original stream.

- deleteAll

  Location: lib/Transforms/RocccCFGtoDFG/RemoveExtends.cpp

  Purpose: All of the previous transformations leave the llvm tree in a state that is not internally consistant. This pass deletes everything, so as to leave the tree in an internally consistant state.

9 VHDL Library Interface

The VHDL library that is used to constuct the in-memory model of the generated VHDL structure performs many compile-time and run-time purposes, mostly debug and error-checking related; at both times, there are checks to guarantee that the generated VHDL structure is well formed. The VHDL library was designed with two purposes: easing the generation of VHDL code by mimicking the syntax tree of the generated VHDL, and fitting within the framwork of ROCCC. Because of this, the library is not a general-purpose VHDL generation library; it is specifically targeted towards error-checking and easing the generation of ROCCC datapaths. The most basic elements of the library are:

- Value

  Base class for all value-type classes in the library. Many operations take Value* objects as thier right-hand-side argument. A value-typed object with base class Value must have a compile-time constant size. Additionally, this class contains the mechanisms for being set as written-to or read-from, as well as being ”owned” by a ValueOwner object; see ValueOwner for more information.

- Attribute

  Base class for all annotation-type classes in the library. Can contain further information about some value or method.

- VHDLAttribute

  Mirrors the VHDL mechanism for annotating elements in the generated VHDL code, by adding ”attributes” which can be of many different value types. The value type, and attribute value, are specified as strings, and no additional checks are done to verify that they are of compatible types. An example use is to generate the following code:

  attribute syn_keep : boolean;
  attribute syn_keep of cumulative1760_registered : signal is true; //tells the synthesis tool not to optimize away this register.

- Attributeable

  Base class for all classes that can have attributes (possibly VHDLAttributes, but not necessarily) attached to them. This is generally used for value-type classes, although it is not required. Adds functionality for adding, setting, and querying Attributes that are attached.

- Variable

  Base class for value-type classes that can be written to at ”run-time”; these include, but are not limited to, Signals, Ports, and ProcessVariables. Derived from Value, and extends the notions of size. In order to integrate easily into the ROCCC framework, holds a pointer to an llvm::Value, which is the corresponding llvm::Value that this Variable is representing. For uses other than matching the ROCCC datapath, this pointer can be NULL.

- Wrap

  Helper class that wraps around a Value*, thus allowing a Value* to exist as the left-hand-side operand to various value-type operators.

- VWrap

  Helper class that wraps around a Variable*, thus allowing a Variable* to exist as the left-hand-side operand to various value-type operators.

- BitRange

  Helper class that will slice a Variable*, given an integer range, and return a VWrap that is useable as the result. The name of the returned result will be the same name as the sliced Variable, but the size will be the integer range.

- Port

  This class extends Variable, and is linked to a ComponentDeclaration; see ComponentDeclaration for its uses there. A port can only be an input or an output, and the library checks that inputs are only read from, and outputs only written to.

- Generic

  This class represents a ComponentDeclaration's generic declaration; the type of the generic is hardcoded as an integer, and is not checked or enforced by the library. As a Value, this can be used as the right-hand-side operand to many operations, but not as the left-hand-side operand.

- ComponentDeclaration

  This class represents a component declaration, such as in a component's .vhdl file, with associated Ports and Generics. This is then used by an ComponentDefinition to actually instantiate components; it is also created by an Entity to pass to other functions and classes. The ComponentDeclaration has a list of associated Ports. Some of these are considered "standard ports", which are ports specific to ROCCC that always exist, such as "clk" and "inputReady". Additional ports, such as data ports, are considered non-standard ports.

- Signal

  This class extends Variable, and extends the notion of checking read/write permissions. Signals are used whenever a VHDL signal is needed, including both registers, and the targets of concurrent statements.

- ConstantInt

  Helper class to create a value-typed Value object that is the VHDL representation of an integer. The size of a ConstantInt value is not defined by the user; instead, it is minimized to the smallest signed-bit representation that will hold the value of that integer.

- ConstantFloat

  Helper class to create a value-typed Value object that is the VHDL representation of an integer. The size of a ConstantInt value is not defined by the user; instead, it is minimized to the smallest floating point representation that will hold the value of that integer.

- NamedConstant

  Helper class that extends Signal to represent VHDL named constants. Multiple operations can all use a NamedConstant; the size of the NamedConstant is the size of the largest operation on that NamedConstant. NamedConstants cannot be written to, unlike Signals. The reason they are derived from Signal, and not Value, is that they need to be declared, similar to Signals, and by deriving them from Signals, they can be placed in the same list in the Entity that Signals are placed in.

- State

  Similar to NamedConstants, this is a helper class that represents a constant state value. Can only be used as the right-hand-side operand of an operation with a StateVar as the left-hand-side operand.

- StateVar

  This class, derived from Signal, represents a VHDL signal that is used in state machines to hold the current state, ie, a state variable. It is declared as a VHDL type that is a list of all of the State's added to the StateVar, and can only be assigned States that are in the list of its type. It is used in CaseStatements to control the currently executing state in a state machine.

- Value operators (+,−,∗,/,&,‖,,sll,srl,sra,bit_concat)

  These operators take two Wrap's and return a Wrap, and allow for operator syntax when dealing with Value*s. Addition and subtraction are the same implementation regardless of the signedness of the operands; multiplication is generated as a signed multiply if either operand is signed. Division throws an error if it is attempted to be generated. '&' implements bitwise and, '‖' implements bitwise or, '̂' implements bitwise xor, sll is a shift left logical, srl is a shift right logical, sra is a shift right arithmetic, and bit_concat concantenates two values.

- HasReplaceAllUsesOfWith

  Base class for all classes that support replacing uses of a Value with uses of another Value. This allows Statements to replace a Value with another Value, even if they themselves do not hold that Value, but rather a sub-Statement that they hold holds it. Additionally, this class supports cloning to upcasted derived classes. There are two templatized functions that ease working with this base class: getClonedCopyIfPossible¡¿(), and ReplaceAllUsesOfWithIn¡¿(). GetClonedCopyIfPossible¡¿() attempts to clone the argument, and returns the cloned version if it succeeds. It returns the original version if it fails. ReplaceAllUsesOfWithIn¡¿() attemps to call replaceAllUsesOfWith() if the passed argument "of" is a HasReplaceAllUsesOfWith; then, if the argument "of" is the same value as the argument "in", it returns the argument "with". In all other cases, it returns the argument "in". These two functions allow arbitrary class objects to be passed to them, and if the passed object does not have enough capabilities, they will not perform the action on them.

- Condition

  Base class for all comparisons between two Value's. Cannot be assigned to Value's, and thus are not Value's themselves.

- CWrap

  Wrapper class around a Condition*, thus allowing a Condition* to exist as the left-hand-side operand to various logical condition operators.

- ConstCondition

  Helper class to convert C-booleans, true and false, to the VHDL equivalent.

- EventCondition

  Helper class to instantiate the rising_edge or falling_edge condition on a Variable*.

- Condition operators(==,<,>,!=,>=,<=,SLT,SGT,SLTE,SGTE,and,or,not)

  These operators take two CWrap's and return a CWrap, and allow for operator syntax when dealing with Condition's. Equals, less-than, greater-than, not-equal, greater-than-or-equal, less-than-or-equal, signed-less-than, signed-greater-than, signed-less-than-or-equal, and signed-greater-than-or-equal, are all implemented with a helper VHDL library. And, or, and not are implemented with the VHDL logical-primitives of the same name.

- ValueOwner

  Base class for all classes that are responsible for "owning" a Value, generally because they drive it. This class additionally declares that ValueOwners must be able to transfer ownership to a suitable ValueOwner; this is done with the canTransferOwnershipOfTo() virtual function.

- ProcessVariable

  This class extends Variable to implement a VHDL process's "variable" syntax. VHDL variables are local scope to the process, and their value changes immediately when it is set.

- Process

  This class extends both ValueOwner and HasReplaceAllUsesOfWith, and implements a clocked VHDL process. The clock domain can be set by the library user. Additionally, a reset signal can be specified, or the process can be generated without a reset signal. This is a pure virtual class - classes extending this class are expected to implement generateSteadyState(), which generates the clocked non-reset path of the body of the process.

- Statement

  Base class for all statement-type classes, including classes that implement the notions of assigment (both in and out of processes), case statements (for state machines), if statements, and comment statements. A statement can belong to a Process, or it can be outside of a Process.

- IfStatement

  This class implements the VHDL construct of ”if (condition) then block; else block; endif;”. It takes a Condition, a true-path Statement, and a false-path Statement, although the false-path Statement is optional. This construct cannot exist outside of a Process.

- MultiStatement

  This class implements a block of statements; it can be used anywhere a single statement can be used, but it itself is several statements.

- AssignmentStatement

  This class implements several types of behavior; it can be used as a single assignment statement inside of a process, such as ”val ¡= rhs_val;”; it can be used as the same construct outside of a process, in a concurrent statement; multiple cases (Condition/Value pairs) can be added to it, and it can be used inside of a process in a if-elseif-else block; or multiple cases (Condition/Value pairs) can be added to it, and it can be used outside of a process in a concurrent statement, such as ”val ¡= rhs_val0 when rhs_val0 ¿ 3 else rhs_val1;”. Adding a case must be done in a specific order; the ”else” case must be added last, and once it is added, no other cases may be added.

- CaseStatement

  This class implements the VHDL construct of case statements that implement state machines, which can only exist inside of processes. State/Statement pairs can be added to the state machine CaseStatement with either the addCase() or addStatement() function. Querying a Statement from the corresponding State is possible with the getCase() function.

- MultiStatementProcess

  This class is a specific implementation of a generic Process, extending Process by implementing generateSteadyState(). This process contains a single MultiStatement*, which Statements can be added to with the function addStatement(). This class is mainly a helper class that implements a common use case for Processes.

- CommentStatement

  This class performs no function other than generating comments embedded in the code; whether these comments actually show up or not is based on current compiler settings. These comments currently are limited to existing inside of a Process.

- ComponentDefinition

  This class implements the VHDL construct of a component definition, ie a subcomponent of the main entity. Both Generics and Ports are mapped to individual Values and Variables, respectively. The ComponentDefinition must be instantiated with what ComponentDeclaration it is defining a component of, and the Ports mapped to by the ComponentDefinition must exist in the ComponentDeclaration.

- Entity

  This class implements the VHDL construct of an entity; it is generally the topmost node in the VHDL syntax tree generated. It contains a list of signals, a list of concurrent statements (incorrectly named

```
// Create the top level entity, named "AddTwo"
Entity* addTwo = new VHDLInterface::Entity("AddTwo");

// The top level entity has two input ports ('A' and 'B') and one output port ('result')
Port* a = addTwo->addPort("A", 32, Port::INPUT);
Port* b = addTwo->addPort("B", 32, Port::INPUT);
Port* result = addTwo->addPort("result", 32, Port::OUTPUT);

// Create a process to handle the addition of the two ports
MultiStatementProcess* addProcess = addTwo->createProcess<MultiStatementProcess>();
addProcess->addStatement(new AssignmentStatement(result, Wrap(a) + Wrap(b), addProcess));
addProcess->addStatement(new AssignmentStatement(addTwo->getDeclaration()->
  getStandardPorts().outputReady, ConstantInt::get(1), addProcess));

// Associate a process that will contain many statements with the entity
addProcess->addStatement(new AssignmentStatement(addTwo->getDeclaration()->
  getStandardPorts().done, ConstantInt::get(1), addProcess));

std::cout << addTwo->generateCode();
```

Figure 1: Example Library Usage

synchronous statements), a list of processes, a list of components, and a list of attributes that it holds. It also has a pointer to a ComponentDeclaration, so as not to redo the exact same work done in the ComponentDeclaration class, as well as to ease passing a generated Entity to a ComponentDefinition in order to instantiate the generated Entity as a subcomponent in another Entity. The Entity class contains several functions dealing with creating Processes, Signals, Attributes, Components, and concurrent statements, which generally set the created object's parent as the Entity as well as store the created object in the correct list of held objects. Several helper functions exist, which can connect subcomponent's ports and the Entity's ports to each other, automatically generating the intermediary Signals. Finally, there are several functions that aid in querying the Entity for Ports and Signals that are either of a certain name, or that are linked to a certain llvm::Value.

As this library is a work-in-progress, any classes other than the above may be unfinished, or unsupported; it is up to the developer to determine the safety and validity of additional classes.

9.1 Example Library Use

The short snippet of C++ code that appears in Figure 1 describes the construction of a VHDL entity that takes two 32-bit inputs, adds them together, and outputs the result. The resulting VHDL that is generated from this code is shown in Figure 2.

The code in Figure 3 creates an entity that instantiates two of the AddTwo components as created by the code in Figure 1. The entity described takes four inputs and maps them to internal AddTwo components. The generated VHDL is shown in Figure 4.

Generating a state machine, as shown in Figure 5, will create a specially formatted block of VHDL code as shown in Figure 6. Case statements in the VHDL library are not generic Case statements but instead exclusively create state machines and must take a State variable.

```
-- This file was automatically generated by ROCCC version 0.7.4,  DO NOT EDIT
library IEEE ;
use IEEE.STD_LOGIC_1164.all ;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.all ;
use work.HelperFunctions.all;
use work.HelperFunctions_Unsigned.all;
use work.HelperFunctions_Signed.all;

entity AddTwo is
port (
       clk : in STD_LOGIC;
       rst : in STD_LOGIC;
       inputReady : in STD_LOGIC;
       outputReady : out STD_LOGIC;
       done : out STD_LOGIC;
       stall : in STD_LOGIC;
       A_in : in STD_LOGIC_VECTOR(31 downto 0);
       B_in : in STD_LOGIC_VECTOR(31 downto 0);
       result_out : out STD_LOGIC_VECTOR(31 downto 0)
     ) ;
end entity;

architecture Synthesized of AddTwo is
begin
process(clk, rst)
begin
  if (rst = '1') then
    result_out <= "00000000000000000000000000000000";
    outputReady <= '0';
    done <= '0';
  elsif( clk'event and clk = '1' ) then
    result_out <= ROCCCADD(A_in, B_in, 32);
    outputReady <= '1';
    done <= '1';
  end if;
end process;
end Synthesized;
```

Figure 2: Example Library Usage 2

```
Entity* addFour = new VHDLInterface::Entity("AddFour");
Port* a4 = addFour->addPort("A", 32, Port::INPUT);
Port* b4 = addFour->addPort("B", 32, Port::INPUT);
Port* c4 = addFour->addPort("C", 32, Port::INPUT);
Port* d4 = addFour->addPort("D", 32, Port::INPUT);
Port* result4 = addFour->addPort("result", 32, Port::OUTPUT);
ComponentDefinition* addTwoInst1 = addFour->createComponent("addTwoInst1", addTwo->getDeclaration());
ComponentDefinition* addTwoInst2 = addFour->createComponent("addTwoInst2", addTwo->getDeclaration());
addFour->mapPortToSubComponentPort(addFour->getDeclaration()->getStandardPorts().clk,
  addTwoInst1, addTwoInst1->getDeclaration()->getStandardPorts().clk);

addFour->mapPortToSubComponentPort(addFour->getDeclaration()->getStandardPorts().clk,
  addTwoInst2, addTwoInst2->getDeclaration()->getStandardPorts().clk);

addFour->mapPortToSubComponentPort(addFour->getDeclaration()->getStandardPorts().rst,
  addTwoInst1, addTwoInst1->getDeclaration()->getStandardPorts().rst);

addFour->mapPortToSubComponentPort(addFour->getDeclaration()->getStandardPorts().rst,
  addTwoInst2, addTwoInst2->getDeclaration()->getStandardPorts().rst);
addFour->mapPortToSubComponentPort(addFour->getDeclaration()->getStandardPorts().inputReady,
  addTwoInst1, addTwoInst1->getDeclaration()->getStandardPorts().inputReady);

addFour->mapPortToSubComponentPort(addFour->getDeclaration()->getStandardPorts().inputReady,
  addTwoInst2, addTwoInst2->getDeclaration()->getStandardPorts().inputReady);

addFour->mapPortToSubComponentPort(addFour->getDeclaration()->getStandardPorts().stall,
  addTwoInst1, addTwoInst1->getDeclaration()->getStandardPorts().stall);

addFour->mapPortToSubComponentPort(addFour->getDeclaration()->getStandardPorts().stall,
  addTwoInst2, addTwoInst2->getDeclaration()->getStandardPorts().stall);

addFour->mapPortToSubComponentPort(a4, addTwoInst1, a);
addFour->mapPortToSubComponentPort(b4, addTwoInst1, b);
addFour->mapPortToSubComponentPort(c4, addTwoInst2, a);
addFour->mapPortToSubComponentPort(d4, addTwoInst2, b);
addFour->createSynchronousStatement(result4,
  Wrap(addFour->getVariableMappedTo(addTwoInst1, result)) +
  Wrap(addFour->getVariableMappedTo(addTwoInst2, result)));

addFour->createSynchronousStatement(addFour->getDeclaration()->getStandardPorts().outputReady,
  ConstantInt::get(1));

addFour->createSynchronousStatement(addFour->getDeclaration()->getStandardPorts().done,
  ConstantInt::get(1));

std::cout << addFour->generateCode();
```

Figure 3: Library Usage Example 3

```
-- This file was automatically generated by ROCCC version 0.7.4, DO NOT EDIT
library IEEE ;
use IEEE.STD_LOGIC_1164.all ;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.all ;
use work.HelperFunctions.all;
use work.HelperFunctions_Unsigned.all;
use work.HelperFunctions_Signed.all;
entity AddFour is
port (
       clk : in STD_LOGIC;
       rst : in STD_LOGIC;
       inputReady : in STD_LOGIC;
       outputReady : out STD_LOGIC;
       done : out STD_LOGIC;
       stall : in STD_LOGIC;
       A_in : in STD_LOGIC_VECTOR(31 downto 0);
       B_in : in STD_LOGIC_VECTOR(31 downto 0);
       C_in : in STD_LOGIC_VECTOR(31 downto 0);
       D_in : in STD_LOGIC_VECTOR(31 downto 0);
       result_out : out STD_LOGIC_VECTOR(31 downto 0)
     ) ;
end entity;
architecture Synthesized of AddFour is
signal addTwoInst2_result : STD_LOGIC_VECTOR(31 downto 0) ;
signal addTwoInst1_result : STD_LOGIC_VECTOR(31 downto 0) ;
component AddTwo is
port (
       clk : in STD_LOGIC;
       rst : in STD_LOGIC;
       inputReady : in STD_LOGIC;
       outputReady : out STD_LOGIC;
       done : out STD_LOGIC;
       stall : in STD_LOGIC;
       A_in : in STD_LOGIC_VECTOR(31 downto 0);
       B_in : in STD_LOGIC_VECTOR(31 downto 0);
       result_out : out STD_LOGIC_VECTOR(31 downto 0)
     ) ;
end component;
begin
addTwoInst1: AddTwo port map (clk => clk, rst => rst, inputReady => inputReady,
  outputReady => open, done => open, stall => stall, A_in => A_in, B_in => B_in,
  result_out => addTwoInst1_result) ;
addTwoInst2: AddTwo port map (clk => clk, rst => rst, inputReady => inputReady,
  outputReady => open, done => open, stall => stall, A_in => C_in, B_in => D_in,
  result_out => addTwoInst2_result) ;
result_out <= ROCCCADD(addTwoInst1_result, addTwoInst2_result, 32);
outputReady <= '1';
done <= '1';
end Synthesized;
```

Figure 4: Library usage 4

```
Entity* stateEntity = new Entity("StateEntity");
Port* a = stateEntity->addPort("A", 32, Port::INPUT);
Port* result = stateEntity->addPort("result", 32, Port::OUTPUT);
StateVar* curStateVar = stateEntity->createSignal<StateVar>("curStateVar", 0);
State* start = curStateVar->addState("start");
State* end = curStateVar->addState("end");
MultiStatementProcess* stateProcess = stateEntity->createProcess<MultiStatementProcess>();
CaseStatement* cs = new CaseStatement(stateProcess, curStateVar);
stateProcess->addStatement(cs);
cs->addStatement(start, new AssignmentStatement(result, Wrap(a), stateProcess));
cs->addStatement(start, new AssignmentStatement(curStateVar, end, stateProcess));
cs->addStatement(end, new AssignmentStatement(stateEntity->getDeclaration()->getStandardPorts().outputR
cs->addStatement(end, new AssignmentStatement(stateEntity->getDeclaration()->getStandardPorts().done, C
std::cout << stateEntity->generateCode();
```

Figure 5: Library Usage 5

```
-- This file was automatically generated by ROCCC -- version 0.6.0.0, released on Monday, January 17, 20
library IEEE ;
use IEEE.STD_LOGIC_1164.all ;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.all ;
use work.HelperFunctions.all;
use work.HelperFunctions_Unsigned.all;
use work.HelperFunctions_Signed.all;

entity StateEntity is
port (
clk : in STD_LOGIC;
rst : in STD_LOGIC;
inputReady : in STD_LOGIC;
outputReady : out STD_LOGIC;
done : out STD_LOGIC;
stall : in STD_LOGIC;
A_in : in STD_LOGIC_VECTOR(31 downto 0);
result_out : out STD_LOGIC_VECTOR(31 downto 0)
);
end entity;

architecture Synthesized of StateEntity is
type curStateVar_STATE_TYPE is (start, end) ;
signal curStateVar : curStateVar_STATE_TYPE ;
begin
process(clk, rst)
begin
  if (rst = '1') then
    result_out <= "00000000000000000000000000000000";
    curStateVar <= start;
    outputReady <= '0';
    done <= '0';
  elsif( clk'event and clk = '1' ) then
    case curStateVar is
      when start =>
        result_out <= A_in;
        curStateVar <= end;
      when end =>
        outputReady <= '1';
        done <= '1';
      when others =>
        --error if we get here
        curStateVar <= start;
    end case;
  end if;
end process;
end Synthesized;
```

Figure 6: Library Usage 6

10 Database

The database used to store information used at all stages of the compiler is implemented using sqlite3 and contains 6 tables - CompileInfo, ComponentInfo, FileInfo, Ports, ResourcesUsed, and StreamInfo.

CompileInfo contains information related to the GUI about how the file was compiled. The columns of CompileInfo are as follows:

- id : This is a unique key that is used as a foreign key in many other tables. Every compilation instance, whether it be the same file or different files, will have a unique id.
- configurationName : If the GUI allows for compilation settings to be saved by name, this column stores the name of the configuration that the user saved it as.
- timestamp : Contains the timestamp that the compilation was started at.
- compilerVersion : Contains the version of the compiler that this compilation ran on.
- targetFlags : Contains any additional flags related to compilation, that are not included in any of the other flag columns.
- highFlags : Contains the flags selected at the GUI level and passed on to the high-cirrf stage of compilation.
- lowFlags : Contains the flags selected at the GUI level and passed on to the low-cirrf stage of compilation.
- pipeliningFlags : Contains the weights and settings selected at the GUI level that are necessary to control low-cirrf pipelining.
- streamFlags : Contains any flags selected at the GUI level necessary for configuring stream parameters at the low-cirrf code generation level.
- isCompiled : Set by the low-cirrf when code generation is completely finished; this lets the GUI know there were no errors.

ComponentInfo contains information about every component that has been compiled with ROCCC. The columns of the ComponentInfo table are as follows:

- componentName : Contains the name of the generated component.
- id : Foreign key into the CompileInfo table - should be a unique link to the actual component being compiled.
- type : Contains either the string "MODULE" or the string "SYSTEM", or contains an intrinsic identifier. The intrinsic identifiers currently are: "INT_DIV", "INT_MOD", "FP_ADD", "FP_SUB", "FP_MUL", "FP_DIV", "FP_EQUAL", "FP_NOT_EQUAL", "FP_LESS_THAN", "FP_GREATER_THAN", "FP_LESS_THAN_EQUAL", "FP_GREATER_THAN_EQUAL", "FP_TO_INT", "FP_TO_FP", "INT_TO_FP", and "INT_TO_INT".
- area : Currently unused.
- frequency : Current unused.
- portOrder : Contains a string that is used by the GUI to initialize an instantiation of a module. When the user double clicks on a module in the IPCore view, the module is inserted into the currently open file with the parameter list being populated with the portOrder string.

- structName : Contains the name of the struct that was originally compiled when creating modules. Used when generating C level code from the GUI, such as PCore generation.
- delay : Contains the number of clock cycles required to generate data for a module. Used by the Lo-CIRRF passes to properly pipeline module calls.
- active : Configuration to turn on and off a component in the database. Set by the GUI when setting intrinsics active, only active components are used by the low-cirrf when searching for subcomponents.
- description : User-supplied description, currently only used for intrinsics, that aids the user in organizing the library of intrinsics they may have.

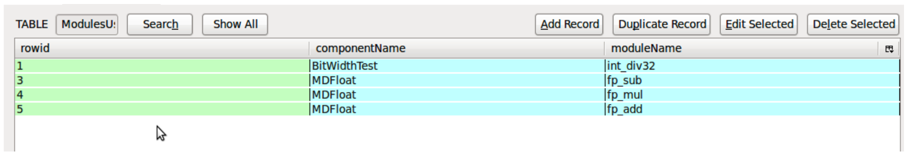
Whenever a component is compiled, all rows in the ComponentInfo table with the componentName equal to the compiled component's name are dropped, and then the relevant information is inserted. This assures that there only ever exists one row with any given componentName.

FileInfo contains information about the files that ROCCC knows about, whether source files or generated files. The columns are as follows:

- id : Foreign key into the CompileInfo table - should be a unique link to the actual component being compiled.
- fileName : Contains the file name, minus path, that is associated with the component being compiled.
- fileType : Contains an identifier that tells the GUI what type of file this is; some options are C_SOURCE, VHDL_SOURCE, and REPORT.
- location : Contains the full path of the file.

The Ports table lists all the ports of every component compiled. Each row of the Ports table is a single port. The columns of the Ports table are as follows:

- id : Foreign key into the CompileInfo table - should be a unique link to the actual component being compiled.
- readableName : Contains the name of the original C value (if available) that this port is associated with. This value is not unique due to the multiple ports generated from the same C value for streams.
- type : Contains one of the following strings - "REGISTER" which is a single, registered value; "STREAM_CROSS_CLK", which is the data clock for a stream; "STREAM_STOP_ACCESS", which is either the full or empty port of a stream; "STREAM_ENABLE_ACCESS", which is either the writeEn or readEn port of the stream; "STREAM_CHANNEL", which is a data port of the stream; "STREAM_ADDRESS_CLK", which is the address clock for the stream; "STREAM_ADDRESS_RDY", which is the address_valid port for the stream; "STREAM_ADDRESS_STALL", which is the address stall port for the stream; "STREAM_ADDRESSS_BASE", which is the base address of a stream channel; and "STREAM_ADDRESS_COUNT", which is the count amount for a stream address channel. There can be multiple STREAM_CHANNEL, STREAM_ADDRESS_BASE, and STREAM_ADDRESS_COUNT ports for any given id and readableName; the rest of the types are unique to a given id and readableName.
- portNum : Contains an integer value that lists in what order the ports come in the component. This is used by the GUI to correctly generate components. In particular, this value is the only way to know the ordering of the STREAM_CHANNELS for a given stream.
- vhdlName : Contains the name of the port in the actual VHDL.
- direction : Contains either the string "IN" or "OUT" to denote the actual direction of the port.



rowid	componentName	moduleName
1	BitWidthTest	int_div32
3	MDFloat	fp_sub
4	MDFloat	fp_mul
5	MDFloat	fp_add

Figure 7: Resources Used Table

- bitwidth : Contains the integer value that is the width of the port in bits.
- dataType : Contains the data type of the original C variable this port is associated with, if any. For example, a floating point input scalar port may have dataType = ”float”, while the data port of a two dimensional integer output stream may have dataType = ”int**”.

The ResourcesUsed table lists the dependencies between components. The columns are as follows:

- id : Foreign key into the CompileInfo table - should be a unique link to the actual component being compiled.
- resourceID : Either a foreign key into the CompileInfo to show what resource this is, or NULL to show it is a non-compiled resource.
- resourceType : A string that explains the type of resource this row is, can be one of the following - ”MODULE”, ”ADD”, ”SUB”, ”COPY”, ”EQ”, ”MUL”, ”MUX”, or ”REGISTER”. For resources other than MODULE’s, the type additionally lists the bitwidth of the resource; a 32 bit add, for example, would be listed as ”ADD 32”.
- numUsed : For this given resource, this contains how many instances of this resource the component being compiled contains.

Each row of the ResourcesUsed table is one dependency. For example, as shown in Figure 7, BitWidthTest only instantiates int_div32, so there is one row with ”BitWidthTest” as the componentName. On the other hand, MDFloat uses fp_sub, fp_mul, and fp_add, so there are three separate rows with ”MDFloat” as the componentName of the row in CompileInfo with the same id.

Whenever a component is compiled, all rows in the ResourcesUsed table with the same id as a row in the CompileInfo table with the same component name as the compiled component’s name are dropped, and then a row for each dependency is inserted. Thus it is possible to have multiple rows with the same id, but they are from a single compilation.

The StreamInfo table has additional information regarding each stream as found by the ROCCC compiler. The columns of this table are as follows:

- id : Foreign key into the CompileInfo table - should be a unique link to the actual component being compiled.
- readableName : the readable name of the stream, as taken from the original c source.
- NumElementsCalculationFormula : This contains a string which is the formula for the number of elements in that stream; for example, if the stream is a 3x3 window indexed with loop induction variables that go from 0 to height and from 0 to width, then the NumElementsCalculationFormula will contain ”(height + 2) * (width + 2)”. This string is used in the PicoInterface generation to transfer the correct amount of data from the software to the hardware.
- NumTotalWindowElements : This is currently unused.
- NumAccessedWindowElements : This is currently unused.