AUTOESL USER GUIDE

UG867 (v 2012.1) April 24, 2012

Table of Contents

Preface

The preface includes the syntax conventions used in this manual.

Conventions

The following conventions are used in this document:

Convention	Description
`Command`	Indicates command syntax, menu element, or a keyboard key.
`<variable>`	Indicates a user-defined value.
`choice1 | choice2`	Indicates a choice of alternatives. The underlined choice is the default.
`[option]`	Indicates an optional object.
`{repeat}`	Indicates an object repeated 0 or more times.
`Ctrl+c`	Indicates a keyboard combination, such as holding down the `Ctrl` key and pressing `c`.
`Menu>Item`	Indicates a path to a menu command, such as `Item` cascading from `Menu`.
`RMB`	Right Mouse Button, gives access to context-sensitive menu in the GUI.

`<variable> ::= choice`
`$ bash_command`
`% tcl_command`
	Indicates syntax and scripting examples (bash shell, Perl script, Tcl script,...).

Important	Indicates an important tip, note or warning.

Table 1 Syntax conventions

Third Party Tool & Version Support

AutoESL is supported in the operating systems shown in Table 2.

Operating System	Version
Windows	XP
	Windows 7
Red Hat Linux OS	RHEL/CentOS 5
SUSE Linux OS	SLED 11

Table 2 Supported Operating Systems

Table 3 shows the version of C compilers supported by AutoESL. The C/C++ and SystemC design files should be compiled and validated with these compiler versions prior to synthesis: synthesis will match the functionality of these versions.

Compiler	Version
C Compiler gcc	4.1 (4.1.2)
C Compiler g++	4.1 (4.1.2)
Microsoft Visual Studio Compiler	2010
OSCI SystemC	2.2

Table 3 C-Language Support

Table 4 details the simulators and versions supported for RTL verification from within AutoESL.

RTL Simulator	Version
Mentor Graphics ModelSim	6.6c
Synopsys VCS[1]	vcs-C-2009.06

Table 4 Supported Simulation Tools

Note: RTL simulators used in the AutoESL flow require the ability to compile and simulate both HDL and C.

C simulation will typically be an option to the HDL simulator: contact the appropriate vendor for details on installing any required C packages and licenses.

RTL implementation executed from within AutoESL supports the following RTL synthesis tools and versions.

RTL Synthesis	Version
ISE	13.1 (13.4)
Synopsys Synplify Pro	E-2010.09-SP3 (E-2011.03-SP2)
Synopsys Synplify Premier	E-2010.09-SP3 (E-2011.03-SP2)

[1] This is only supported on Linux operating systems.

Table 5 Supported RTL Synthesis Tools

Introduction

The introduction explains different concepts associated with High-Level Synthesis (HLS) and gives a basic overview of AutoESL, the Xilinx® HLS tool.

Functional Abstraction Level

Definitions

The FPGA design community has moved through a few abstraction levels, to manage the complexity of the designs. Each new abstraction level hides some of the complexity of a design implementation step, offering productivity at the cost of less visibility in the challenges associated with the lower abstraction level:

- A transistor layout database hides the challenges in mask making and wafer processing. The focus of the layout abstraction layer is to respect Design Rule Checks (DRC) which models the basic layout.
- For FPGA design, a netlist avoids a detailed layout effort: the netlist is constructed with instances from a pre-built library. The focus of the netlist abstraction layer is to define the Boolean functionality of the design with appropriate area, performance and power.
- A Register Transfer Level (RTL) description captures the desired functionality by defining datapath and logic between boundaries of registers. RTL synthesis creates a netlist of Boolean functions to implement the design. The focus of the RTL abstraction layer is to define a model for the hardware which is functionally correct.
- A functional specification removes the need to the define register boundaries (and the specific logic required between them) to implement the desired algorithm. The focus of the designer is only on specifying the desired functionality.

As with previous moves up the abstraction level, using a functional specification with high-level synthesis (HLS) to automatically create the RTL design provides productivity benefits in both verification and design optimization.

- The significant benefits of acceleration in simulation time by using a functional C language based specification and the resultant earlier detection of design errors has been embraced for quite a while.
- High Level Synthesis shortens the previous manual RTL creation process and avoids translation errors by automating the creation of the RTL from the functional specification.
- High Level Synthesis automates the optimization of the RTL architecture, allowing multiple architectures to quickly and easily be evaluated before committing to an optimum solution.

C-based Specification

C-based entry is the most popular mechanism to create functional specifications. Currently, ANSI-C (with C99), C++ and SystemC are standards deployed by many

system architects to define the functionality of systems intended to be implemented on an FPGA.

AutoESL provides comprehensive support for C, C++ and SystemC, the IEEE standard (IEEE-1666) used for modeling and concurrent simulation of hardware. The constructs which cannot be synthesized are those which unbounded at elaboration time and for which a finite sized description cannot be determined.

Native C data types live within the classic boundaries of 8-bit, 16-bit, 32-bit and 64-bit words (`char`, `short`, `int`, `long`, `long long`). Neither ANSI-C nor C++ has built-in data types to deal with bit-accurate calculations, where the exact bit-width of the data type is used (and which results in optimally sized hardware). AutoESL provides support for arbitrary precision data types in both C and C++. AutoESL fully supports the arbitrary precision data types provided by SystemC.

High-Level Synthesis (HLS)

The synthesis of C into RTL employs many advanced transformations working on all aspects of the design area and performance. AutoESL provides synthesizable support for a large subset of all three input C standards (C, C++ and SystemC) enabling it to synthesize the C code with minimal modifications.

AutoESL performs two distinct types of synthesis upon the design:

- Algorithm Synthesis takes the content of the functions, and synthesizes the functional statements into RTL statements over a number of clock cycles.

- Interface Synthesis transforms the function arguments (or parameters) into RTL ports with specific timing protocols, allowing the design to communicate with other designs in the system.
    - Interface synthesis can be performed on global variables, top-level function arguments and the return value of the top-level function.
    - The types of available interfaces are:
        - Wire
        - Register
        - One-way & two-way handshakes
        - Bus
        - FIFO
        - RAM
    - In addition, a function level protocol can be synthesized to the top-level function. The function level protocol includes signals which control when the function can start operation and indicate when it has completed.

AutoESL synthesis is executed in multiple steps. The effect of interface synthesis impacts what is achievable in algorithm synthesis and vice versa. Like the numerous decisions made during any manual RTL design, the number of available implementations and optimizations is large and the combinations of how they

impact each other is very large. AutoESL abstracts the user away from these details and allows the user to productively get to the best design in the shortest time.

To better understand how AutoESL is able to abstract the designer away from the implementation details, it is recommended to review the remainder of this section which explains some of the fundamental concepts of HLS and type of optimizations AutoESL provides:

- Control and Datapath Extraction
- Scheduling & Binding
- Arbitrary Precision Data Types
- Optimizations
- Design Constraints

Control and Datapath Extraction

The first thing which is performed during HLS is to extract the control and datapath inferred by the code. Figure 1 shows a small example on how this is performed.

The control functionality is provided by the loops and conditional branches in the code. Figure 1 shows how the control behavior can be extracted from the code. Each time the function requires an entry or exit from a loop, it is equivalent to entering or exiting a state in an RTL Finite State Machine (FSM)[2].

In Figure 1 it is assumed that all operations take a single cycle (or state) to complete. In reality, timing delays and the clock frequency may require more cycles to complete the operations, for example state 1 may expand to states 11, 12 and 13, the control logic may be impacted by the IO protocols inferred by interface synthesis and AutoESL may create a more complex and optimized state machine.

The datapath extraction is more straightforward and can be determined by unrolling all the loops and evaluating the conditional statements in the design.

[2] This loop to state correlation may not be exact, as AutoESL may optimize the FSM.

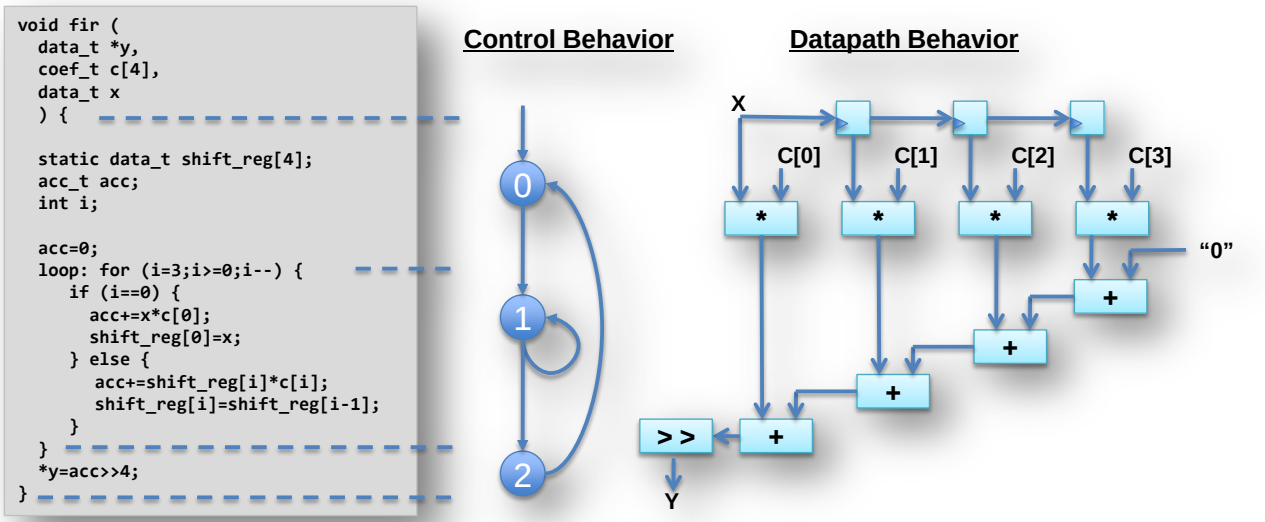

Figure 1 Control and Data Extraction

The final datapath implementation in the RTL is unlikely to be as simple as that shown in Figure 1: AutoESL will easily determine that the first adder is not required since the final shift operation is a power of 2 and requires no hardware. More complex optimizations and decisions will be made when the design is scheduled.

Scheduling & Binding

Scheduling and binding are the processes at the heart of high-level synthesis. AutoESL will determine during the scheduling process in which cycle operations will occur. The decisions made during scheduling take into account, among other things, the clock frequency and clock uncertainty, timing information from the device technology library, as well as area, latency and throughput directives.

For the same example code shown in Figure 1, multiple RTL implementations are possible. Figure 2 shows just 3 possible implementations.

1. Using 4 clock cycles means a single adder and multiplier can be used, as AutoESL can share the adder and multiplier across clock cycles: 1 adder, 1 multiplier and 4 clock cycles to complete.
2. If analysis of the target technology timing indicates the adder chain can complete in 1 clock cycle, a design which uses 3 adders and 4 multipliers but which finish in 1 clock cycle can be realized (faster but larger than option 1).
3. Take 2 clock cycles to finish but use only 2 adders and 2 multipliers (smaller than option 2 but faster than option 1).

AutoESL quickly creates the most optimum implementation based on its own default behavior and the constraints and directives specified by the user. Later chapters explain how to set constraints and directives to quickly arrive at the most ideal solution for the specific requirements.

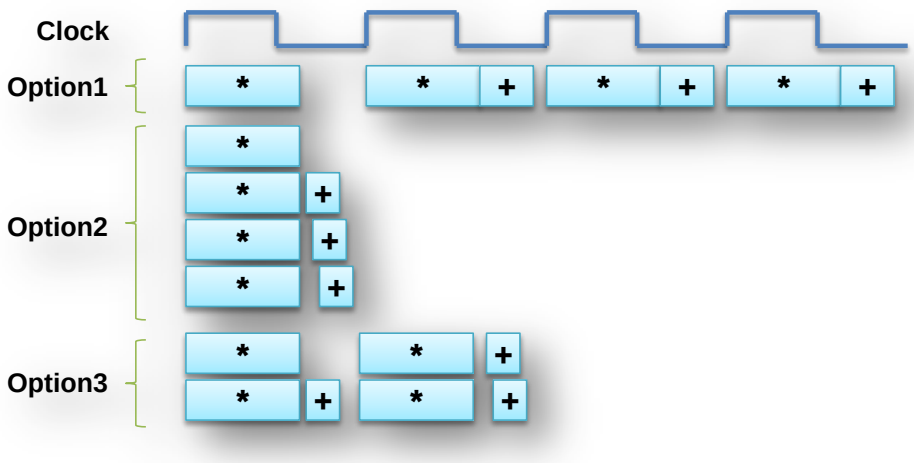

Figure 2 Scheduling

Binding is the process that determines which hardware resource, or core, is used for each schedule operation. For example, AutoESL will automatically determine if an adder and subtractor will used or if a single adder-subtractor can be used for both operations.

Since the decisions in the binding process can influence the scheduling of operations, for example, using a pipelined multiplier instead of a standard combinational multiplier, binding decisions are considered during scheduling.

Arbitrary Precision Data Types

Native C data types are on 8-bit boundaries (8, 16, 32, 64 bits). RTL operations (corresponding to hardware) support arbitrary widths. HLS needs a mechanism to allow the specification of arbitrary precision bit-widths or the RTL design may use 32-bit multipliers when only 17-bit multipliers are required (not an issue to a C program, but a major issue in an RTL design).

AutoESL provides arbitrary precision integer and fixed-point data types (Table 6).

Language	Integer Data Type	Required Header
C	[u]int<precision> (1024 bits)	#include "ap_cint.h"
C++	ap_[u]int<W> (1024 bits)	#include "ap_int.h"
	ap_[u]fixed<W,I,Q,O,N>	#include "ap_fixed.h"
SystemC	sc_[u]int<W> (64 bits) sc_[u]bigint<W> (512 bits)	#include "systemc.h"
	sc_[u]fixed<W,I,Q,O,N>	#include "sc_fixed.h"

Table 6 Integer data types

These arbitrary types are supported by functions which provide hardware like operations, such as bit-slicing, concatenation and range-selection. Refer to the section “Arbitrary Precision Data Types" section in this User Guide.

Optimizations

AutoESL can perform a number of optimizations on the design to produce high quality RTL satisfying the performance and area goals. This section introduces a few of the optimization techniques to give an overview of the capabilities.

Pipelining is an optimization which allows one of the major performance advantages of hardware over software, concurrent or parallel operation, to be automatically implemented in the RTL design.

A C program operates in a sequential manner. Given the function "top" shown on the left-hand side of Figure 3, every sub-function from “func_A” to “func_C” must complete its operation before “func_A” can once again execute.

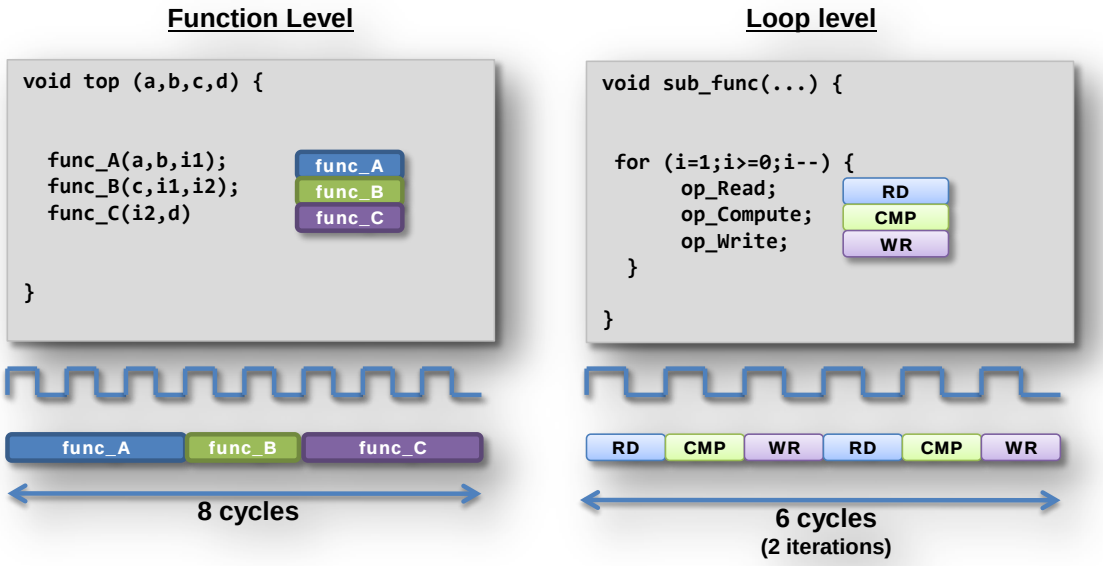

Figure 3 Functions & Loops without pipelining

Even if “func_A” is ready to process the next set of operations as soon as it is finished, functions "func_B" and "func_C" must complete execution before “func_A” can once again begin operation.

As function “sub_func” on the right-hand side of Figure 3 shows, it is the same at the operator level: the first operation cannot re-execute until the last is complete.

The sequential nature of the C language, or in other words its lack of concurrency, puts artificial dependencies on operations which must wait their turn for execution. AutoESL provides the ability to automatically pipeline both functions and loops to ensure the RTL design does not suffer from such limitations.

By default, AutoESL will seek to execute these operations in parallel and reduce the overall latency of the design. In addition to this, AutoESL can improve the throughput by pipelining these operations, allowing different executions of the function or different loop iterations to overlap in time.

Figure 4 shows the result when AutoESL is used to pipeline the sub-functions and/or operations in a loop.

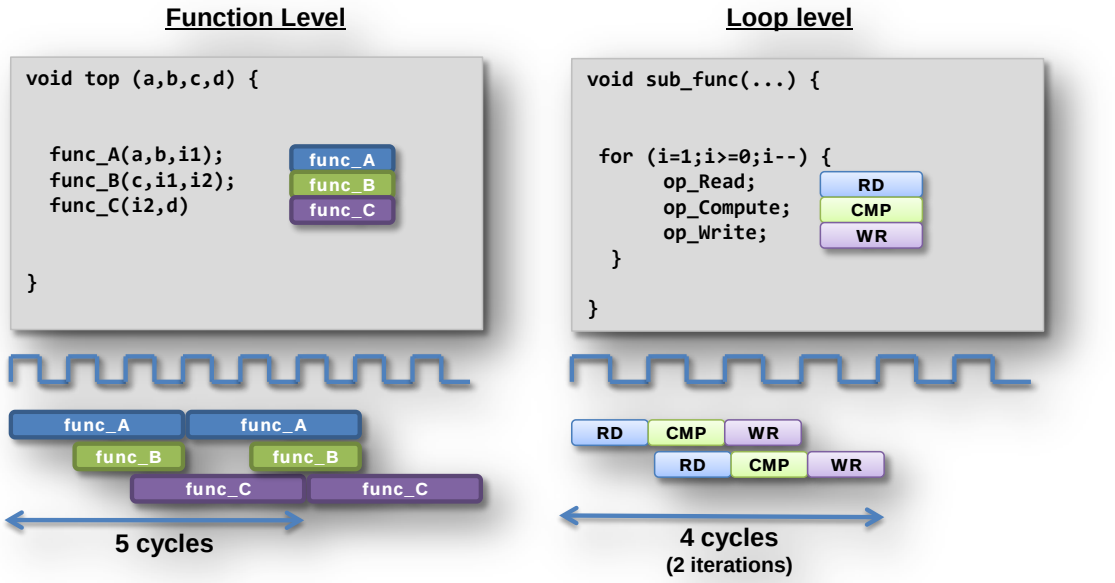

Figure 4 Functions & Loops with pipelining

- At the function level, dataflow optimization allows the sub-functions ("func_A", "func_B" and "func_C") to execute as soon as data is available.
  - Function "func_A" starts it's next operation "before func_C" has completed its first execution.
  - Compared with the previous implementation in Figure 3, the 8 clock cycles it took to execute the function is now only 5 cycles and "func_A" starts a new operation every 3 clock cycles instead of every 8.
- Pipelining the loop allows the operations in a loop to execute concurrently.
  - Figure 4 shows how loop pipelining can also positively performance compared with Figure 3: the loop completes in only 4 clock cycles and processes an new input (RD operation) every clock cycle instead of waiting for 3 clock cycles.

Another example of a design optimization which can be automatically implemented by AutoESL is array partitioning.

Within C language descriptions, arrays are used as a convenient way to group similar elements together. When the elements of arrays are synthesized as storage elements (that is, when the value must be maintained across clock cycles) these array elements can be grouped at the RTL in RAMs or they can be broken into their constituent parts and implemented as individual registers.

- If the elements of an array are accessed one at a time, an efficient implementation in hardware is to keep them grouped together and mapped into a RAM.

- If multiple elements of an array are required simultaneously, it may be more advantageous for performance to implement them as individual registers: allowing parallel access to the data.

Implementing an array of storage elements as individual registers may help performance but this loses the substantial benefits of RAMs: area efficient in all technologies and they are readily available in the device as BRAMs (separate from the LUTs and registers).

AutoESL provides a variety of techniques to ensure arrays are implemented in the most optimal manner:

- Partitioning large arrays into multiple smaller arrays, which can be mapped to different instances of RAM (allowing multiple reads or writes in the same cycle).
- Enabling multiple small arrays to be implemented onto the same RAM resource.

The application of a few simple directives provides for a large number of different implementations, from pipelining to the manipulation of arrays, ensuring that the most optimal implementation for the particular design can be quickly and easily found.

Design Constraints

Finally, in addition to the clock period and clock uncertainty, AutoESL offers a number of constraints including the ability to:

- Specify a specific latency across functions, loops and regions.
- Specify a limit on the number of resources used.
- Override the inherent or implied dependencies in the code and permit operations (for example, a memory read before write)

These constrains can be applied using AutoESL directives to create a design with the desired attributes.

Designing with AutoESL is a HLS flow allows the designer to quickly implement an initial architecture, which will be defined by the dependencies in the code and the default AutoESL interpretation of C language constructs, and then easily direct the design with directives towards the desired high performance implementation.

AutoESL Overview

As shown (Figure 5) AutoESL accepts as input, a C-based design description, and directives and constraints, specified using the Graphical User Interface (GUI) or a Tcl batch script. A technology library specifying the timing and area details of all supported Xilinx device is built-in and is not required to be supplied.

AutoESL outputs RTL design files in Verilog, VHDL and SystemC. In addition verification and implementation scripts, used to automated the RTL verification and RTL synthesis steps are also created.

This section provides an overview of these various inputs and outputs.

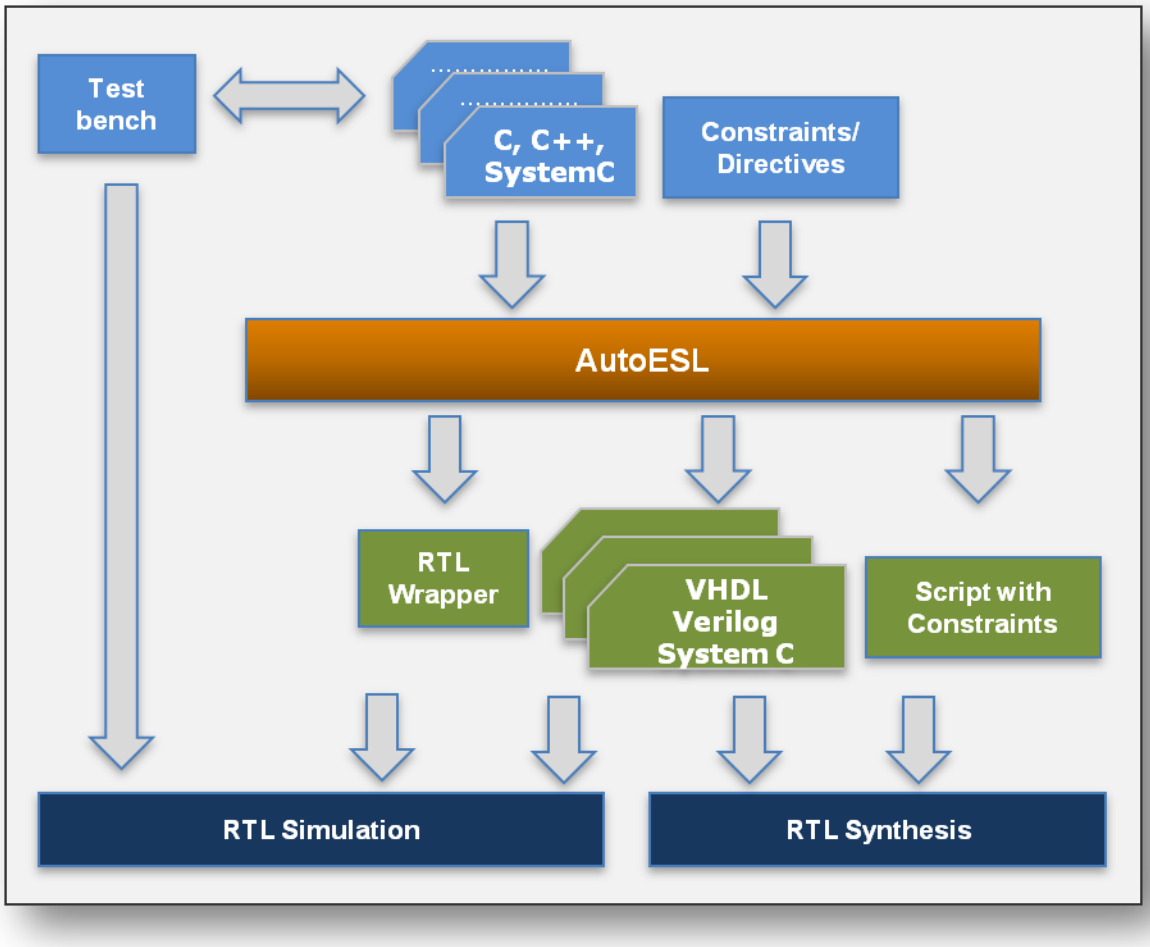

Figure 5 AutoESL use model

Design Files

When referring to C, or C-based design, AutoESL covers all 3 standards:

- ANSI-C enhanced with a data type for arbitrary integer precision.
- C++ enhanced with classes for arbitrary integer precision and fixed point precision.
- SystemC (IEEE-1666)

The documentation will elaborate on how to simulate the input specification, including explanations of the provided arbitrary precision enhancements.

The C-based input can include a test bench. If provided, a C test bench can be re-used to verify the output RTL: improving designer productivity by removing the need to create RTL test benches for RTL verification. AutoESL supports multiple input files and while the recommended flow separates the test bench from the design to be synthesized in separate files, this is not required.

Device Technology Library

A device technology library models the area and timing of each supported Xilinx device, enabling the optimization engine to make the appropriate trade-offs. The device technology library is built-in to AutoESL and does not need to be supplied.

Directives and Constraints

The directives and constraints are specified in the AutoESL GUI or with the Tcl-based command language and drive the optimization engine towards the desired performance goals and RTL architecture.

RTL Output

The RTL output is written automatically after the successful completion of synthesis. AutoESL supports three hardware description language standards:

- VHDL (IEEE 1076-2000)
- Verilog (IEEE 1364-2001)
- SystemC (IEEE 1666-2006 -Version 2.2-)

Note: The SystemC output from AutoESL is the design implementation at the Register Transfer Level (RTL).

Simulation Output (RTL co-simulation)

AutoESL creates the scripts required to verify the generated RTL through co-simulation with the original test bench and a variety of RTL simulators. The following RTL simulators are supported:

- ModelSim
- VCS
- OSCI SystemC

The SystemC output can be verified using the built-in SystemC kernel and requires not third part simulator or license. The supported HDL simulators require a license from the appropriate vendor.

Implementation Output

The scripts and constraint files required for processing the design through RTL synthesis and P&R on the FPGA are provided. These scripts ensure the RTL

synthesis process can be completed in a push-button manner from the AutoESL GUI.

Using AutoESL

This section provides an introduction to AutoESL, explaining how to invoke AutoESL, create a project, use solutions to manage the RTL implementation and apply directives for optimization. After this introduction, details on a tutorial example are provided.

AutoESL can be invoked as a Graphical User Interface (GUI) or as a Command Line Interface (CLI) which accepts Tcl commands in interactive or batch mode.

AutoESL Graphical User Interface

Windows

To invoke AutoESL on a PC Windows platform double-click on the desktop icon as shown in Figure 6.

Figure 6 AutoESL GUI Icon

Linux

To invoke AutoESL on a Linux platform execute the following command at the Linux command prompt.

```
$ autoesl
```

The AutoESL GUI invokes as shown in Figure 7.

Figure 7 GUI mode

The `Getting Started` options in Figure 7 allow the following tasks to be performed:

- Create New Project
  - This will launch the project setup wizard.
- Open project
  - Navigate to an existing project.
- Open Recent Project
  - Select from a list of recent projects.

The `Documentation` tasks available directly from the `Welcome Screen` (Figure 7) are:

- Browse Examples
  - Open AutoESL examples. These can also be found in the examples directory in the AutoESL installation area.
- Release Note Guide

    - Open the Release Notes for this version of software.
- User Guide
    - Open the AutoESL User Guide.
- AutoESL Tutorial
    - Select a tutorial to open.

AutoESL Command Line Interface

On Windows the AutoESL Command Line Interface (CLI) can be invoked from the desktop AutoESL Command Prompt icon as shown in Figure 8.

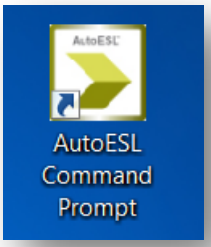

Figure 8 AutoESL CLI Icon

On Windows and Linux, using the `-i` option with the `autoesl` command will open AutoESL in interactive mode. AutoESL will wait for Tcl commands to be entered.

```
$ autoesl -i [-l <log_file>]

autoesl>
```

By default, AutoESL creates an `autoesl.log` file in the current directory. To specify a different file, the `-l <log_file>` option can be used.

AutoESL supports auto-completion: press the `<TAB>` key after the initial few letters of a command and AutoESL will offer a list of candidates that match the command.

```
autoesl> open<TAB>
open
open_project
open_solution
```

The AutoESL commands have built-in help, which can be accessed with the `help` command in AutoESL. The command name can still be provided through auto-complete:

```
autoesl> help <command>
```

Type the `exit` command to quit interactive mode, and return to the shell prompt:

```
autoesl> exit
$
```

Commands also can be embedded in a Tcl script and executed in batch mode with the `-f <script_file>` option.

```
$ autoesl –f script.tcl
```

To further help with script automation AutoESL provides options which will return details on the environment in which it is running, namely the `–version` option which returns the version number of AutoESL, `-system` which returns operating system AutoESL is running on, the `-machine` option which returns the current machine architecture and `–root_dir` which returns the name of the directory where AutoESL is installed.

Using the CLI Shell on Windows

On the Windows OS, the CLI shell is implemented using the Minimalist GNU for Windows (minGW) environment which allows both standard Windows DOS commands to be used and/or a subset of Linux commands to be used.

Figure 9 shows that both (or either) the Linux `ls` command and the DOS `dir` command can be used to list the contents of a directory.

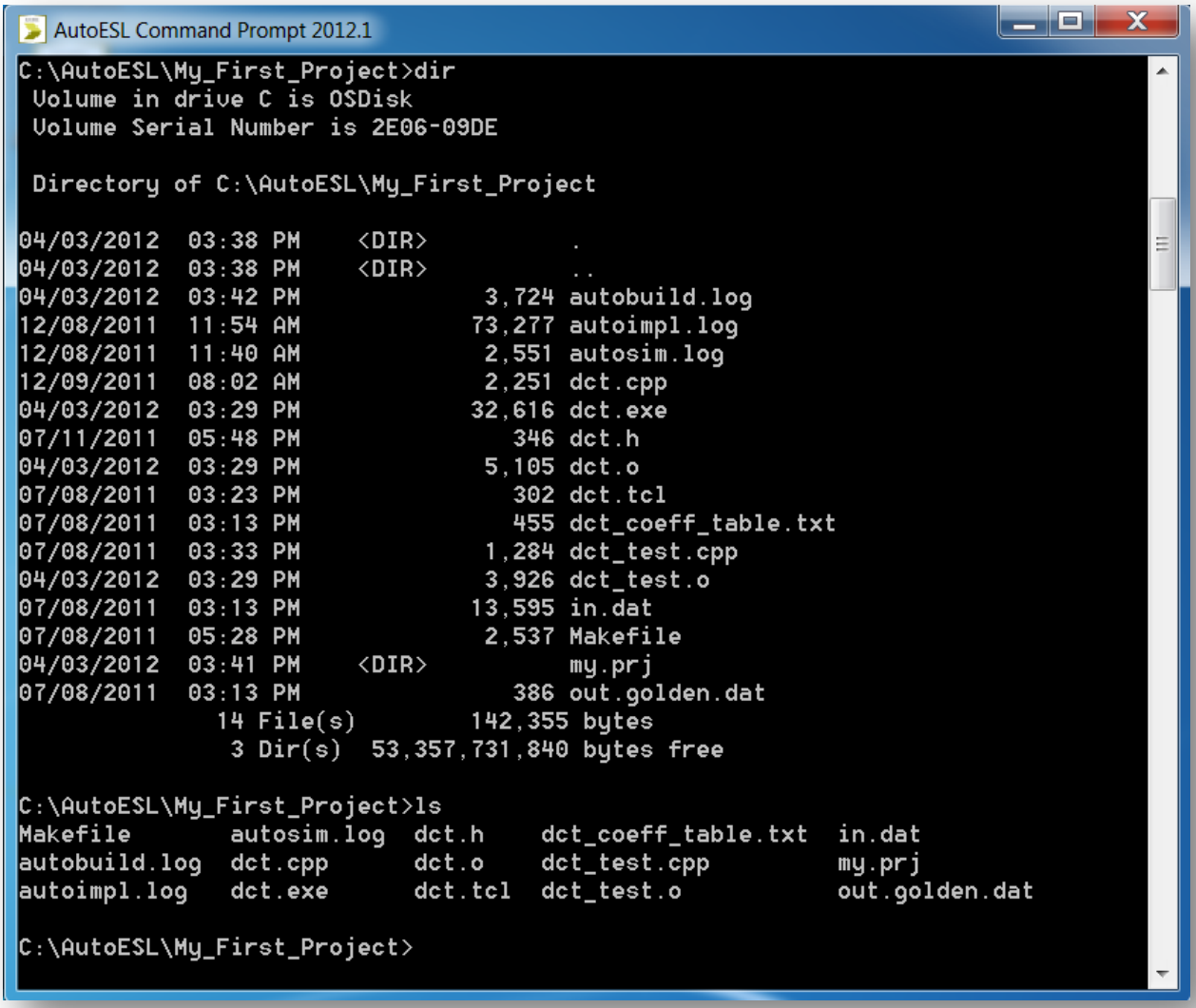

Figure 9 CLI Shell Example

Be aware that not all Linux commands and behaviors are supported in the minGW environment. The following represent some know common differences in support:

- The Linux `which` command is not supported.
- Linux paths in Makefile will be automatically expanded to minGW paths. In all Makefile, replace any Linux style pathnames assignments such as `FOO := :/` with versions where the pathname is quoted such as `FOO := ":/"` to prevent any path substitutions.

Creating an AutoESL Project

The first step in using AutoESL is to create a new project or open an existing project. As shown in Figure 7 when the AutoESL GUI invokes the menu commands for performing these operations are `File -> New Project` and `File -> Open Project`.

When `File -> New Project` is selected the AutoESL project wizard invokes. The first screen of the project wizard asks for details on the project specification as shown in Figure 10.

The fields for entering the project specification are:

- Project Name: In addition to being the project name this will be the name of the directory when the project details are stored. The use of a file extension, such as the `.prj` extension shown in Figure 10, makes the directory easily identifiable as a project directory but is not a requirement.

- Location: This is when the project will be stored.

- Top Level: If the top-level module is a SystemC SC_MODULE, select `SystemC`. Otherwise select `C/C++` (the default). If the source files use SystemC types but the top-level is not an SC_MODULE, select `C/C++`.

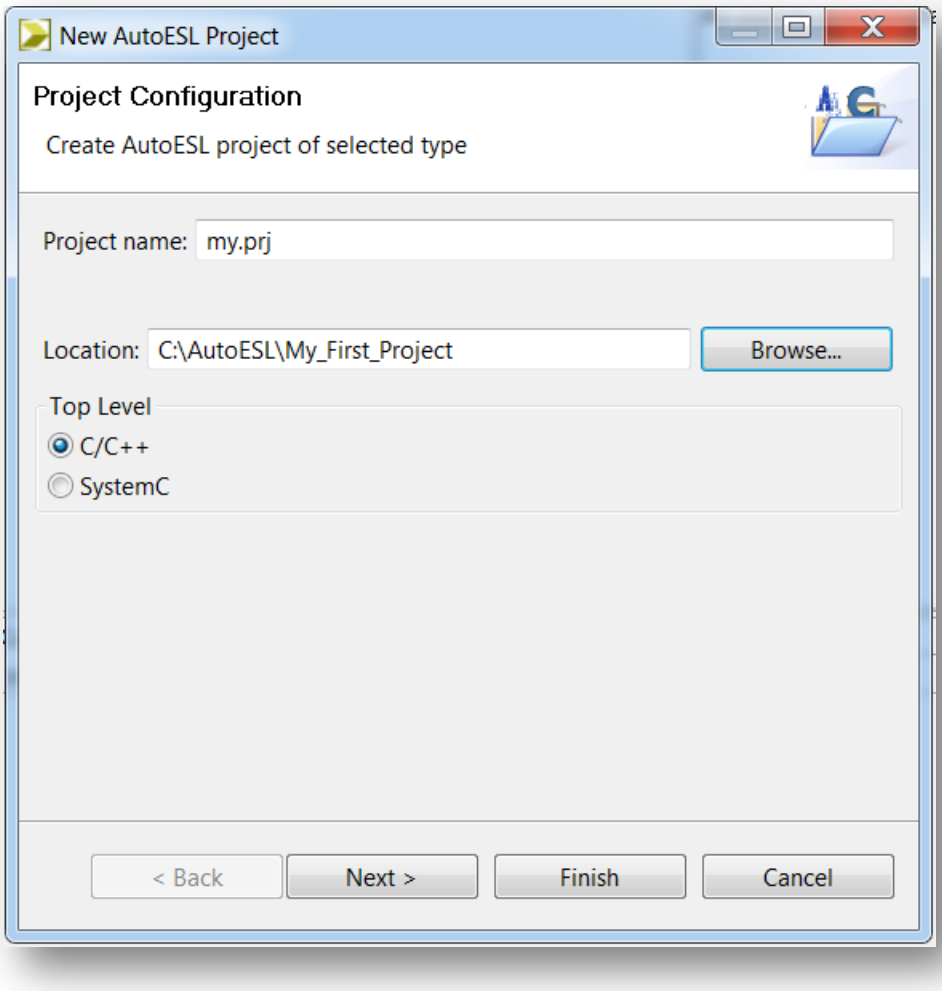

Figure 10 Project Specification

Pressing the `Next>` button will move the wizard to the second screen where details of the project C sources files can be entered (Figure 11).

The name of the top-level function to be synthesized should be specified.

> **Note**: This is not required when the project is specified as SystemC.

Use the `Add Files…` button to add the source code files to the project.

The `Edit CFLAGS` button allows any C compiler flags required to successfully compile the source files, to be added to the project.

Examples of C compiler flags include macro specifications such as `-DMACRO_1` which defines macro `MACRO_1` during compilation, `-fnested-functions` which is required for any design which contains nested functions and `-I/project/source/headers` which provides the search path for any associated header files. Any headers which exist in the local directory (as specified by the Location in Figure 10) are automatically found and included.

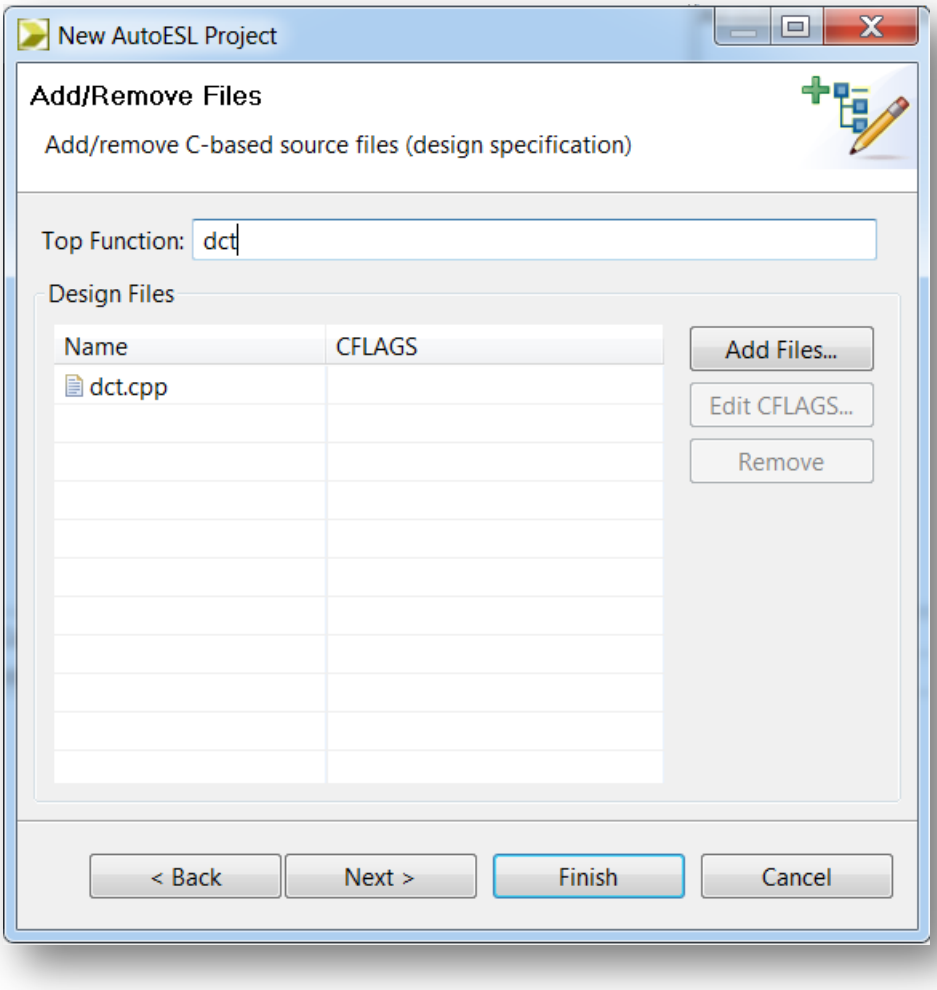

Figure 11 Project Source Files

The next window in the project wizard allows the files associated with the test bench to be added to the project.

The C test bench used to validate the C algorithm can be reused to verify the output RTL. AutoESL automatically creates the adapters and wrappers to instantiate the RTL design into the C test bench and verify the RTL through co-simulation of C and HDL, negating the requirement to create an RTL test bench.

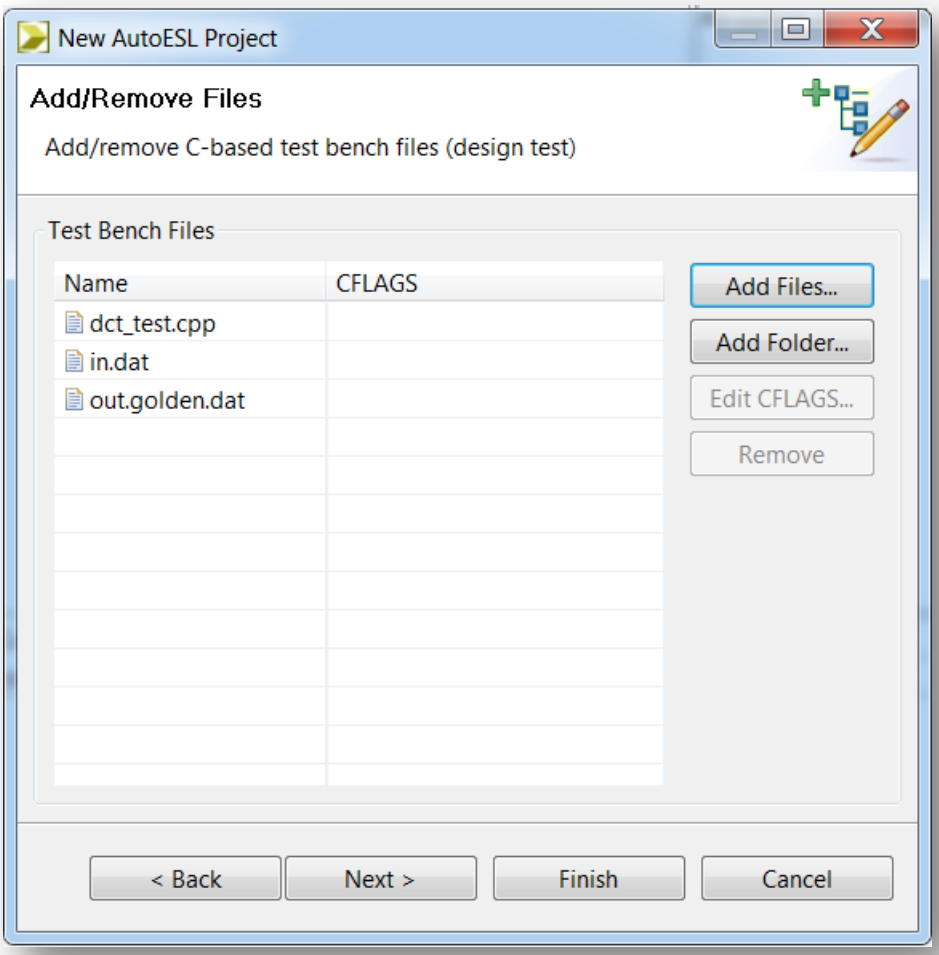

Figure 12 Project Test bench Files

As with the C source files, the `Add Files…` button is used to add the C test bench and the `Edit CFLAGS` button to include any C compiler options.

In addition to the C source files, all files read by the test bench should be added to the project. In the example shown in Figure 12, the test bench opens file `in.dat` to supply input stimuli to the design and file `out.golden.dat` to read the expected results. Since the test bench accesses these files, both are (and must be) included in the project.

If the test bench files exist in a directory, the entire directory may be included rather than the individual files.

If there is no C test bench, there is no requirement to enter any information here and the `Next>` button will open the final window of the project wizard which allows the details for the first solution to be specified (Figure 13)

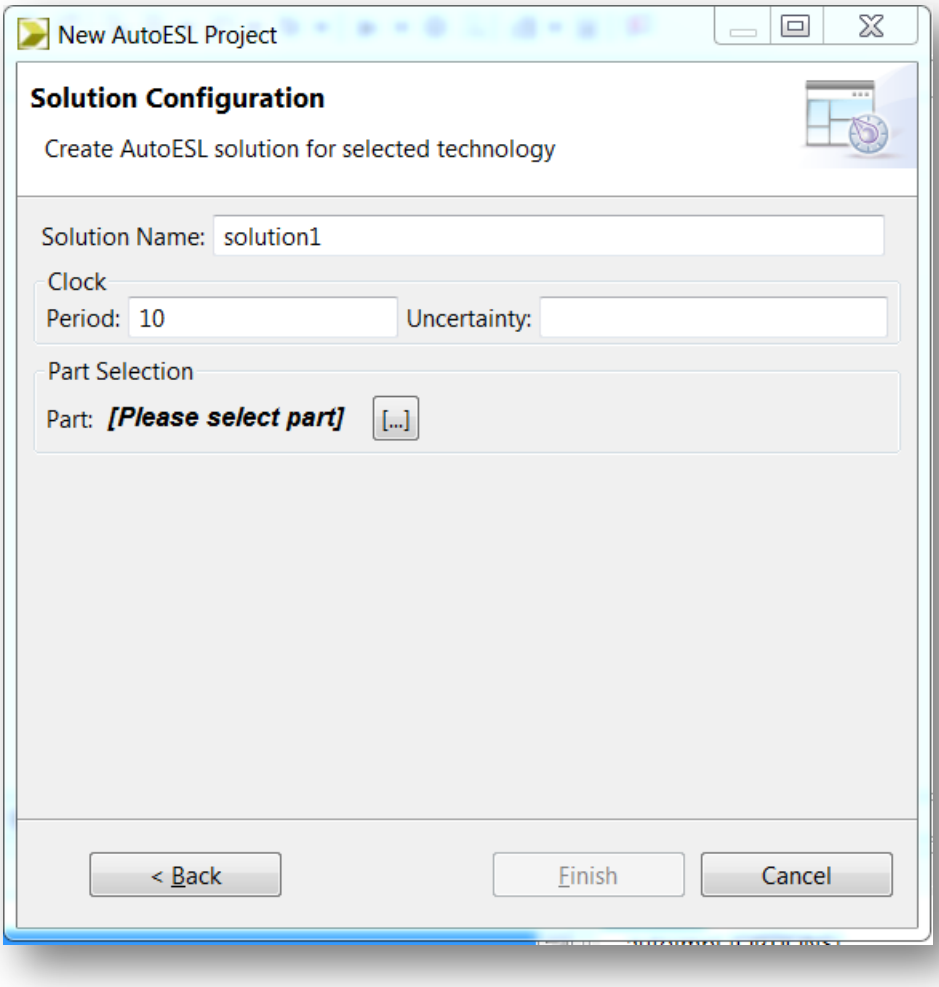

Figure 13 Initial Solution Settings

The fields in Figure 13 allow the following details to be specified:

- Solution Name: AutoESL provides the initial default name `solution1` but any name can be specified for the solution.

- Clock: The clock period is specified using units of ns. The clock period used for synthesis is the clock period minus the clock uncertainty. AutoESL uses the timing information in the technology library to create the RTL design. The clock uncertainty value allows a user controllable margin to account for any

increases in net delays due to RTL synthesis, place and route. If not specified in ns, the clock uncertainty defaults to 12.5% of the clock period.

- Part: Press to select the appropriate technology (Figure 14).

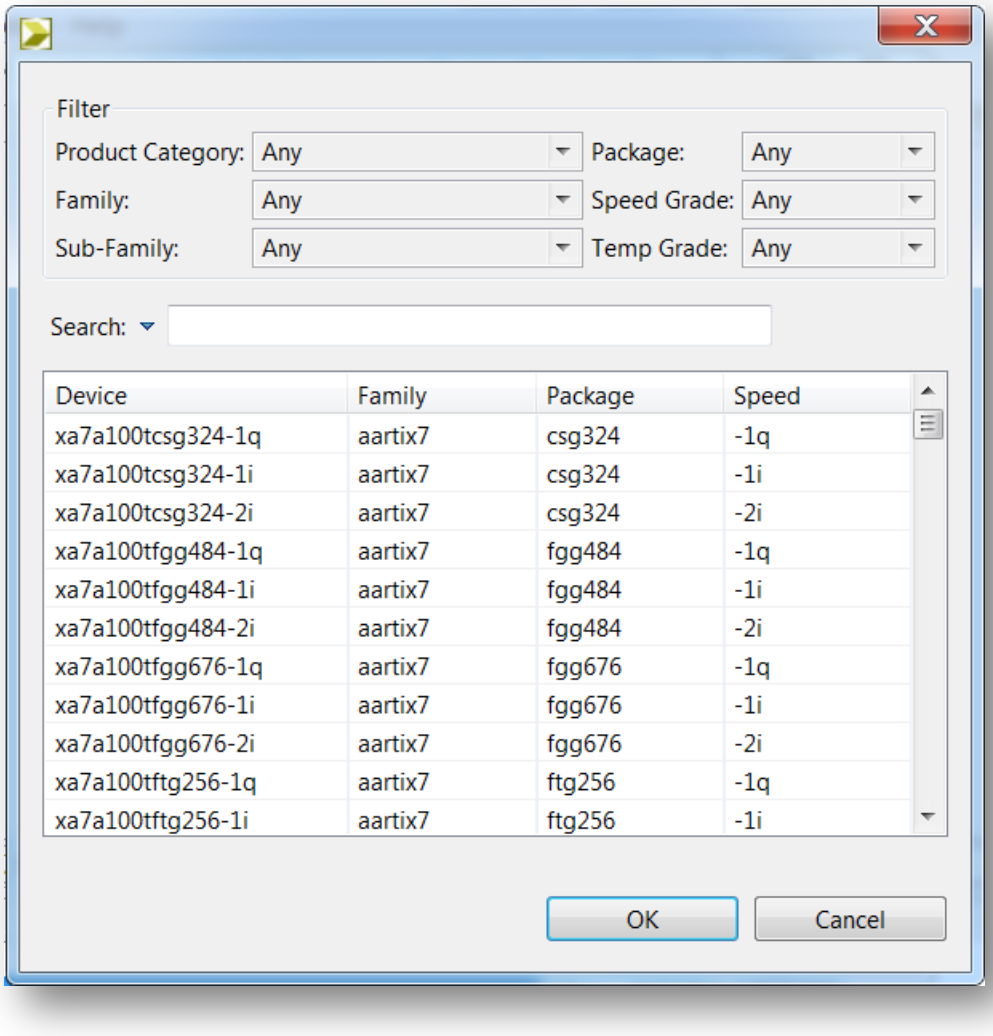

Figure 14 Part Selection

Selecting Finish will open the project as shown in Figure 15.

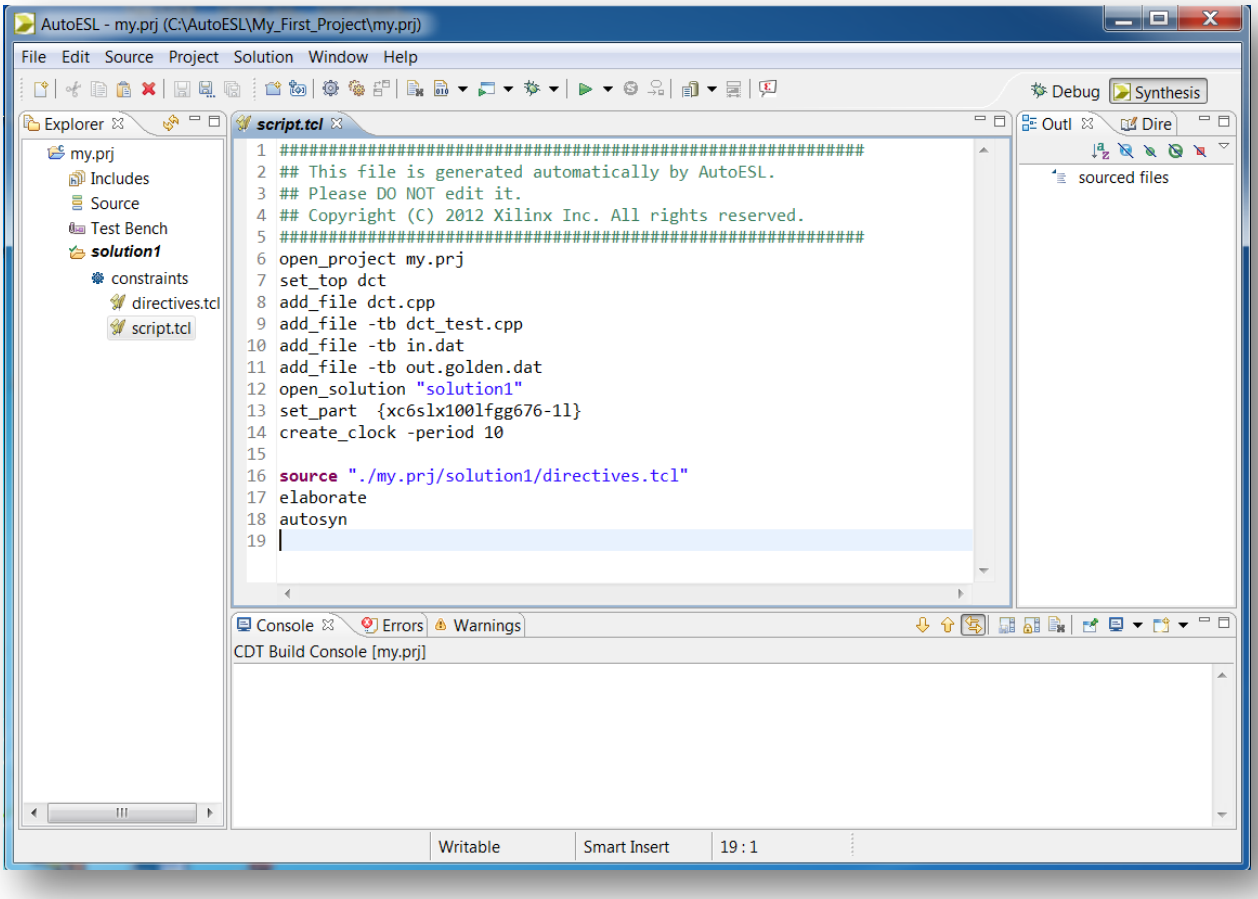

Figure 15 New Project

All the Tcl commands for creating the project are stored in the `script.tcl` file within the solution. Double-clicking on the `script.tcl` file in the Explorer pane (right-hand side of Figure 15) opens the file in the Information pane, as shown in Figure 15.

For users wishing to develop Tcl batch scripts, the `script.tcl` file is an ideal starting point. The containers shown in the Explorer pane can be found in the project directory: simply copy the file from solution directory.

The primary commands for using AutoESL are provided in the toolbar (Figure 16). Project control ensures only commands which can be currently executed are highlighted. For example, synthesis must be performed before RTL simulation can be executed and thus the simulation toolbar button will remain grey until synthesis completes.

The primary command buttons, shown within the red box in Figure 16, are (in left to right order):

- Create a `New Project`
- Create a `New Solution`
- Edit the existing `Project Settings`

- Edit the existing `Solution Settings`
- `Compare Solution` reports
- Cleanup the C simulation environment.
- Build the C/C++/SystemC executable.
- Run the C/C++/SystemC executable.
- Run the C/C++/SystemC executable in debug mode.
- Execute `Synthesize`
- Execute RTL `Simulation`
- Execute RTL synthesis (`Implementation`)
- `Open Reports` on synthesis, simulation or implementation

Figure 16 Tool Bar

Each of the buttons on the tool bar has an equivalent command in the menus.

The first step after creating a project is to validate the C function. Pressing the Build toolbar button will compile the C design (debug or optimized release version), as shown in Figure 17.

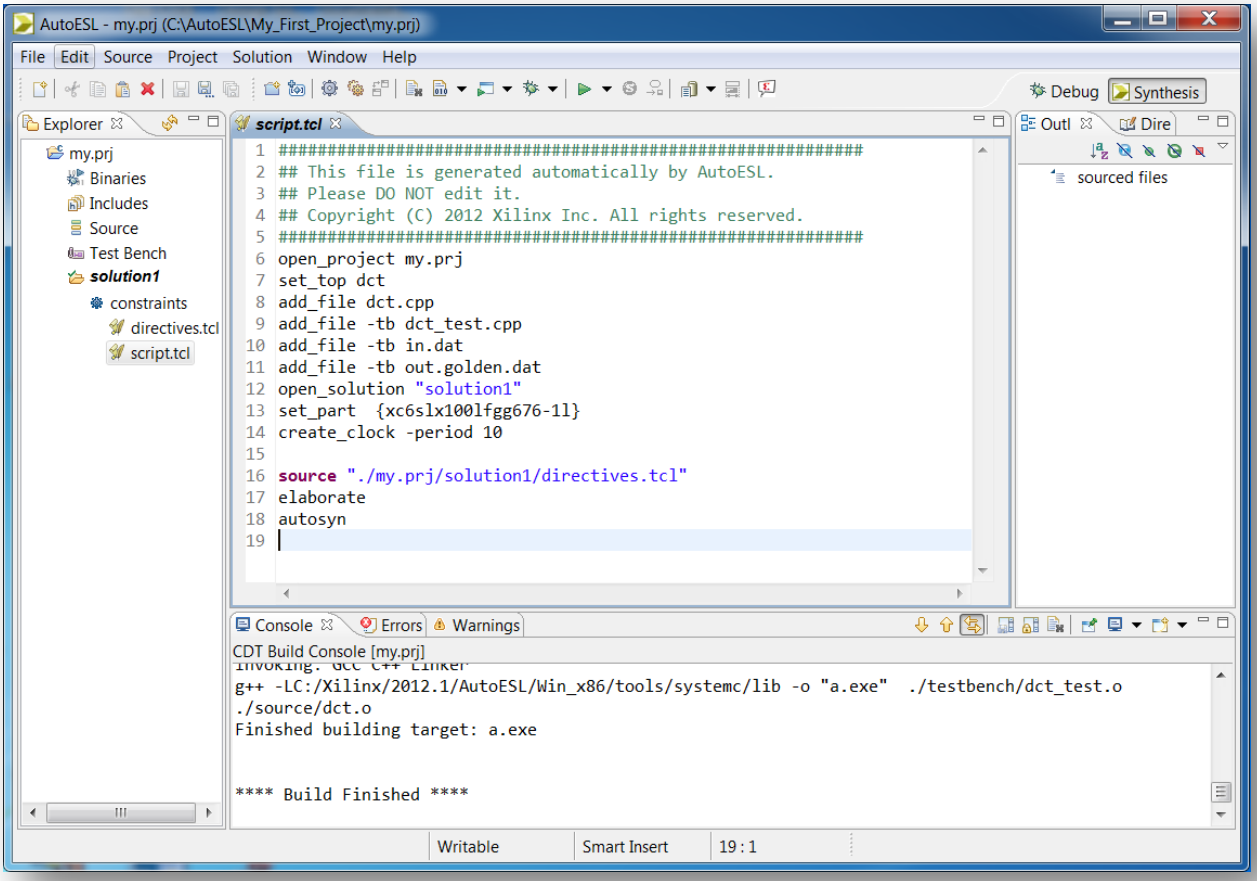

Figure 17 C Compiled with Build

The build can then be run or optionally viewed in the debug environment. If the Debug toolbar button is used the debug environment can be opened (Figure 18) and the debug step buttons (red box in Figure 18) to step through the code and analyze its operation.

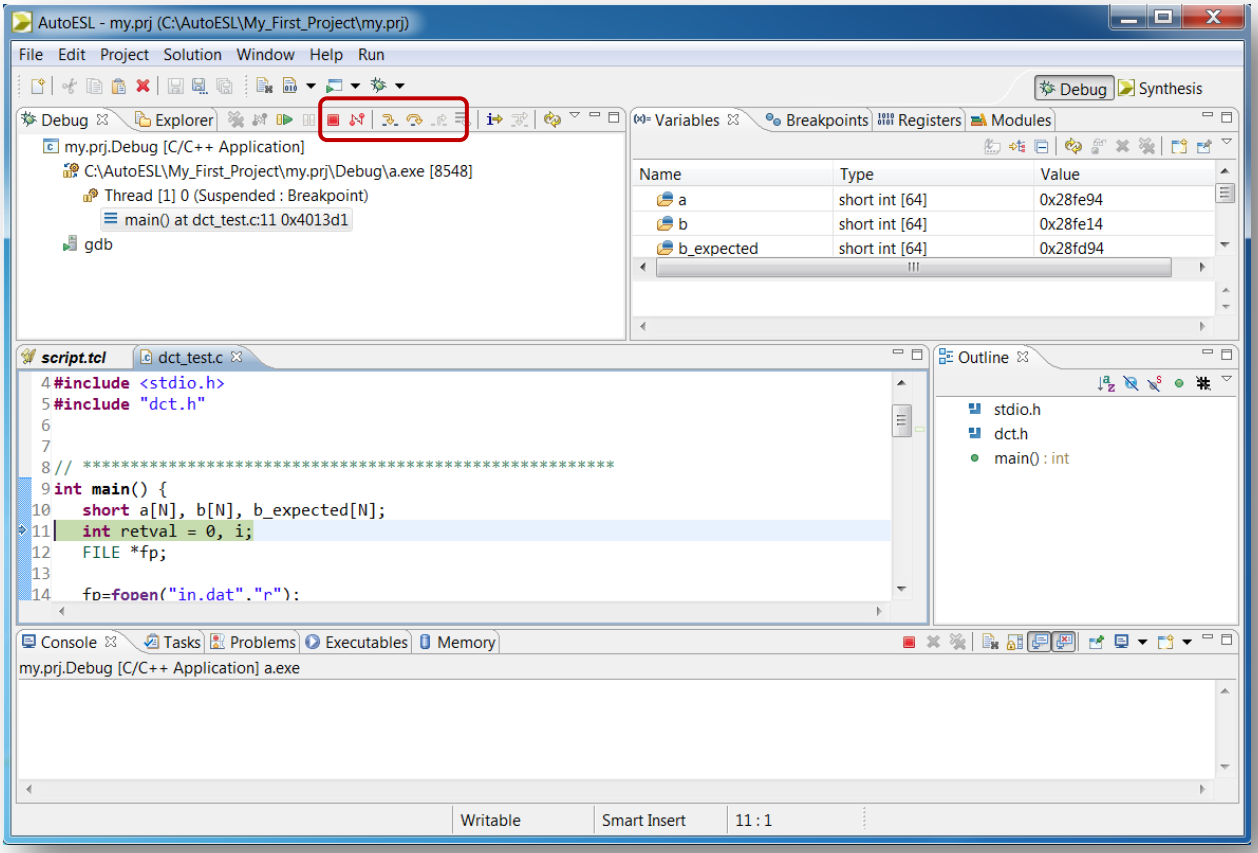

Figure 18 C Debug Environments

The next step is to execute synthesis. When synthesis completes the synthesis report is available, it will open automatically in the information pane, and results can be analyzed. (Figure 19)

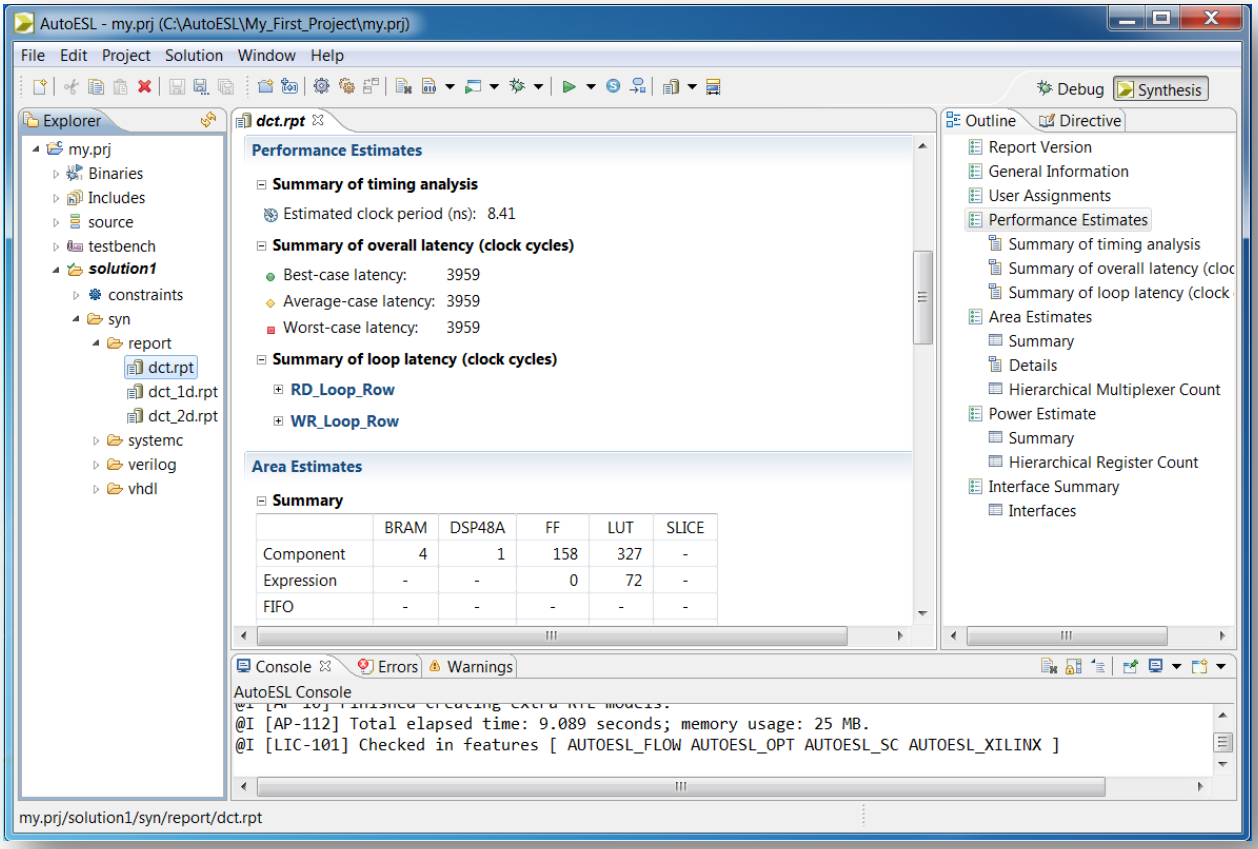

Figure 19 Synthesis Report

The report provides details on both the performance and area of the RTL design. The outline tab can be used to navigate through the report. Reports are created for each function in the hierarchy (unless the function was inlined: an optimization discussed in later chapters). The report for the top-level function provides details for the entire design.

Table 7 explains the categories in the synthesis report.

Category	Sub-Category	Description
Report Version	---	Details on the version of AutoESL used to create the results.
General Information	---	Project name, solution name and when the solution was executed.
User Assignments	---	Details on the technology, target device attributes and the target clock period.
Performance Estimates	Summary of timing analysis	The estimate of the fastest achievable clock frequency. This is an estimate because logic synthesis and P&R are still to be

Category	Sub-Category	Description
		performed.
	Summary of overall latency	The latency of the design: the number of clock cycles from the start of execution until the final output is written. If the latency of loops can vary, the best, average and worse case latencies will be different. If the design is pipelined this section will show the throughput (Without pipelining the throughput is the same as the latency: the next input will be read when the final output is written).
	Summary of loop latency	This shows the latency of individual loops in the design. The trip count is the number of iterations of the loop. The loop latency is the latency to complete all iterations of the loop.
Area Estimates	Summary	This shows the resources (LUTS, Flip-Flops, DSP48s etc.) used to implement the design. The sub-categories are explained in the Details section of this table.
	Utilization	Shows the utilization of resources for the selected device.
	Details: Component	The resources specified here are used by the components (sub-blocks) within the top-level design. Components are created by sub-functions in the design. Unless inlined, each function becomes it's own level of hierarchy. In this example there are no sub-blocks: the design has one level of hierarchy.
	Details: Expression	This category shows the area used by any expressions such as multipliers, adders, comparators etc. at the current level of hierarchy.
	Details: FIFO	The resources listed here are those used in the implementation of FIFOs at this level of the hierarchy.
	Details: Memory	The resources listed here are those used in the implementation of memories at this level of the hierarchy.
	Details: Multiplexors	All the resources used to implement multiplexors at this level of hierarchy are shown here.
	Details: Registers	This category shows the register resources used at this level of hierarchy.
	Hierarchical Multiplexor Count	A summary of the multiplexors throughput the hierarchy.
Power Estimate	Summary	The expected power used by the device. At this level of abstraction the power is an estimate and should be used for comparing the efficiently of different solutions.
	Hierarchical Register Count	The estimated power used by resisters throughput the design hierarchy.
Interface Summary	Interface	This section shows the details on type of interfaces used for the function and the ports: port names, directions, bit-widths, etc.

Table 7 Report

The most typical use of AutoESL is to create an initial design, then perform optimizations to meet the desired area and performance goals. Solutions offer a convenient way to ensure the results from earlier synthesis runs can be both preserved and compared.

Using Solutions

The `New Solution` tool bar button (Figure 16) or the menu `Project->New Solution` can be used to create a new solution. This opens the solution wizard (Figure 20).

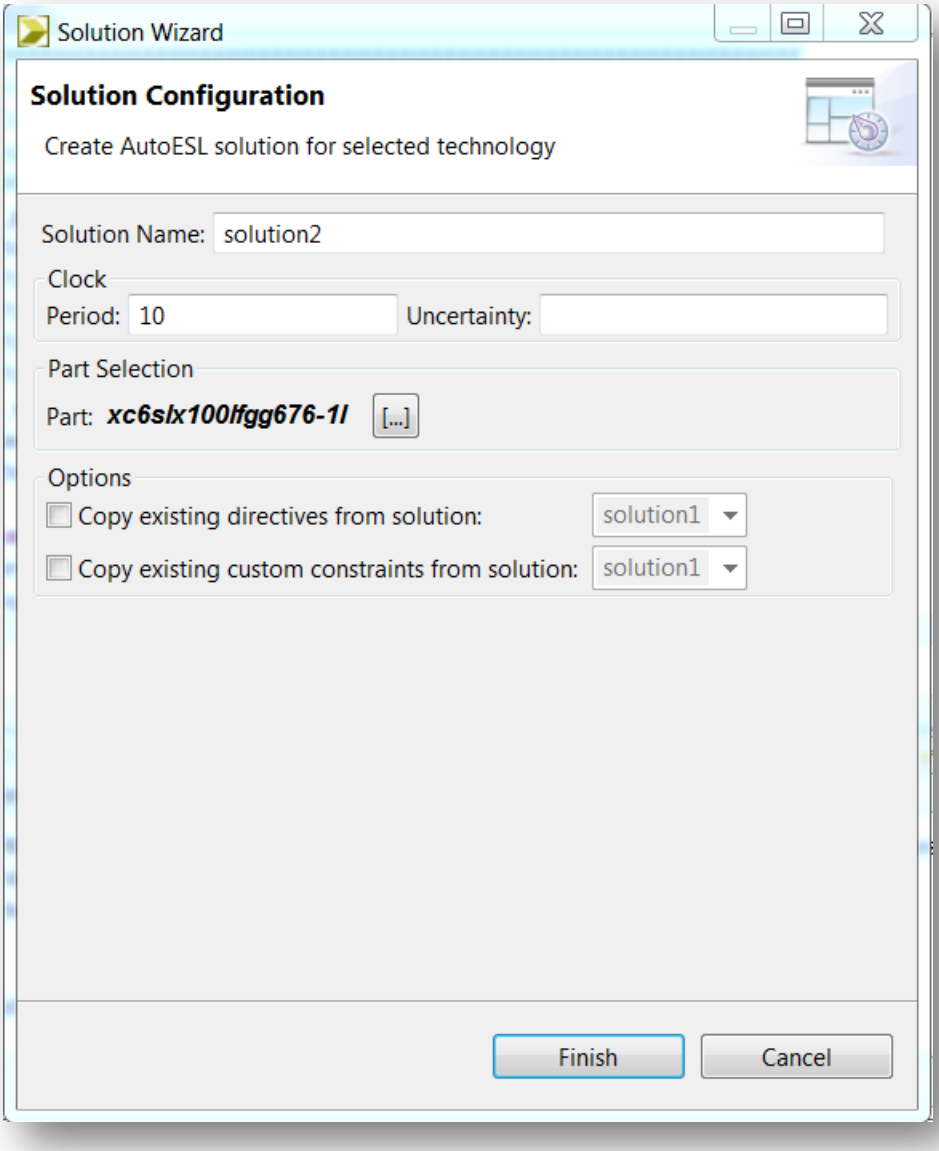

Figure 20 Solution Wizard

The solution setting window has the same options as the final window in the `New Project` wizard (Figure 13) plus two additional options which allow directives and customs commands which were applied to a solution to be conveniently copied to the new solution, where they may be modified or removed. The next section explains how directives can be added to solutions.

Using The AutoESL GUI

Before discussing how optimizations are performed, it is worth spending some time to review how the AutoESL GUI displays information and how it can be customized.

In some cases the default setting of the AutoESL GUI may prevent certain information from being shown. This relates to the following:

- Information defined in header files.
- Comments in the source written in a language other than English.

Resolving Header File Information

By default, the AutoESL GUI does not automatically parse all header files to resolve all coding constructs. The symptoms of this can be:

- Annotations in the code viewer which say a variable or value is unknown or cannot be defined.
- Variables in the code which do not appear in the directives window.

In both cases, the definitions for the unknown values and missing variables will be defined in a header file (a file with extension .h or .hpp). The solution to resolving the missing information is to edit the project setting using menu item `Project` ➔ `Project Settings…` and enable the Parse All Header Files as shown in FIGURE.

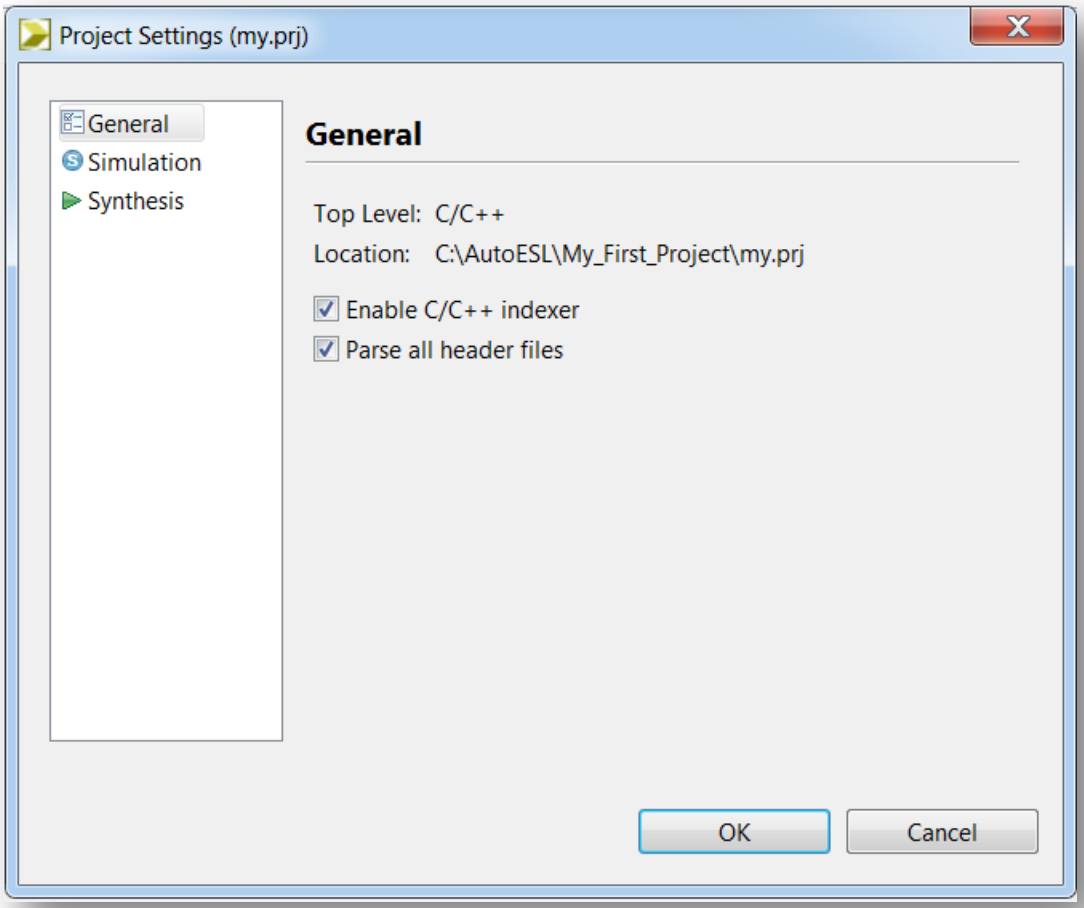

Figure 21 Enabling Header File Parsing

Note: When option `Parse all header files` is selected, the AutoESL GUI will continuously pole all header files for any potential changes.

This may result in a reduced response time from the GUI as CPU cycles are used to check the header files.

Resolving Comments in the Source Code

In some localizations, non-English comments in the source file may appear as strange characters. This can be corrected by:

1. Selecting the project in the `Explorer Pane`.
2. Right-click and select the appropriate language encoding using `Properties` ➔ `Resource`. In the section titled `Text File Encoding` select `Other` and choose appropriate encoding from the drop-down menu.

Customizing the GUI Behavior

The behavior of the AutoESL GUI can be customized using the menu `Windows` ➔ `Preferences` and new user defined tool settings saved.

As an example on how detailed customizations can be performed using the Preferences menu, the following change will be made: The default setting for the key combination CTRL-TAB, is to make the active tab in the Information Pane toggle between the source code and the header file. This will be changed to make the CTRL-TAB combination make each tab in turn the active tab.

- In the Preferences menu, sub-menu General ➔ Keys allows the Command value Toggle Source/Header to be selected and the CTRL-TAB combination removed by using the Unbind Command key.
- Selecting Next Tab in the Command column, placing the cursor in the Binding dialog box and pressing the CTRL key and then the TAB key, will cause the operation CTRL-TAB to be associated with making the Next Tab active.

Reviewing the sub-menus in the Preferences menu allows every aspect of the AutoESL GUI environment to be customized to ensure the highest levels of productivity.

Using Directives to Optimize

Directives can be used to perform various optimizations on the design. This section explains how optimizations are added to the solution. The various optimizations are discussed in detail in later chapters of this User Guide

The first step in adding optimization directives is to open the source code in the Information pane.

As shown in Figure 22, expand the source container, located at the top of the Explorer pane, and double-click on the source file to open it for editing in the Information pane.

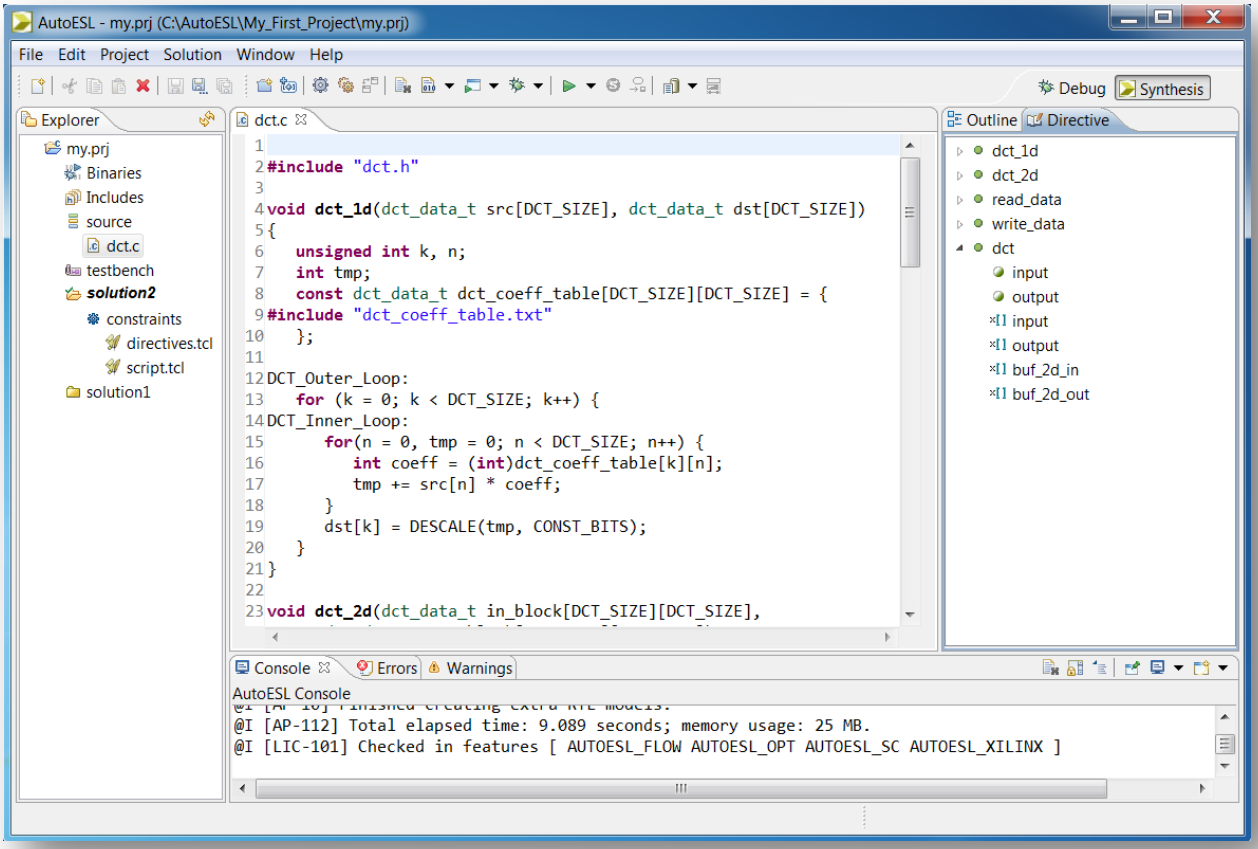

Figure 22 Source and Directive

With the source code active in the Information pane, the directives tab on the right-hand side becomes active.

Directives Tab

The directives tab contains all the objects in the opened source code upon which directives can be applied.

- Functions
- Interfaces
  - Interfaces are the arguments to the top-level function: these will become ports on the RTL design and directives can be specified on these to specify the IO protocol ports.
- Arrays
- Loops
- Regions
  - A region is any named region of code surrounded by braces.

Note: The objects shown in the directives tab are only those from the file currently shown in the information pane (current active file): not all files in the design.

The following example shows the outline of some source code, highlighting each of the scopes and objects upon which directives can be applied and optimizations performed.

```
int foo_sub_A (int mem_1[64],..) {
    for_A: for (int n = 0; n < 3; ++n) {
              ...
    }
    ...
}
int foo_sub_B (int mem_1[64], int i) {
    for_B:for (int n = 0; n < 4; ++n) {
              ...
    }
    ...
}
void foo_top (int mem_1[64], int mem_2[64]) {
      ...
       for_top: for (int i = 0; i < 64; ++i) {
              my_label: {
                     ...
              }
       }
}
```

Figure 23 shows how this example code is represented in the directives tab.

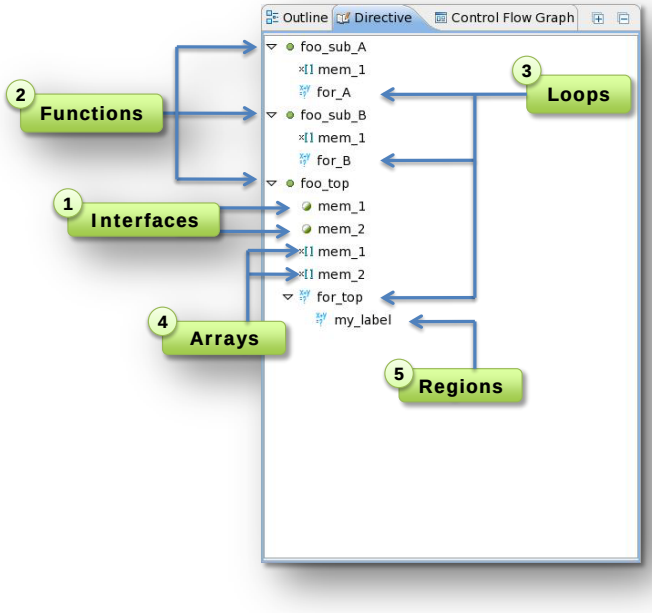

Figure 23 GUI Directives Objects

Applying Directives

Directives are applied by selecting an object in the directives tab and clicking with the right-hand button of mouse to open the directives window, as shown in Figure 24.

The drop-down menu allows the appropriate directives to be added. The example in Figure 24 shows the DATAFLOW directive being added. In addition to the options for the directive (discussed in later chapters) the directives window allows the directive to be inserted into the directive file as a Tcl command or to be inserted directly into the code as a pragma.

> **Note**: To apply directives to objects in a header file, such as a class:
>
> To add a directive to a class member or global variable, open the Directives Editor on a function that uses the variable and enter the variable name manually in Directives Editor.
>
> To add a directive to a local scalar, open the Directives Editor on a function that contains the variable. and enter the variable manually in Directives Editor.

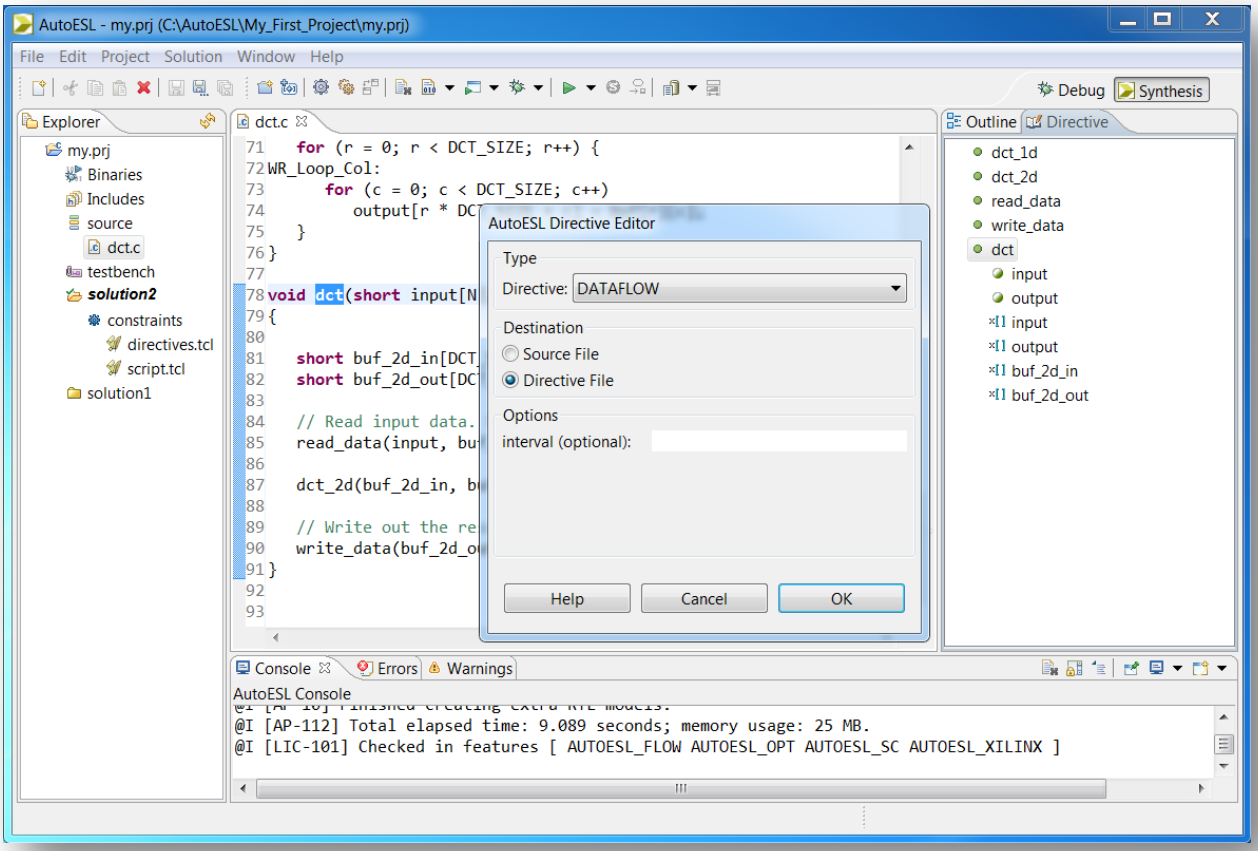

Figure 24 Adding Directives

When the `Into Directive File` option is selected in the directives dialog box, the directive is written to file directives.tcl in the solution directory. The two advantages for this approach are:

- Each solution can have its own directives.
- Users wishing to create Tcl batch files can simply copy the directive from the directives.tcl file

Figure 25 shows the directive being added to the `directives.tcl` file and shows the resulting `directives.tcl` opened in the information pane.

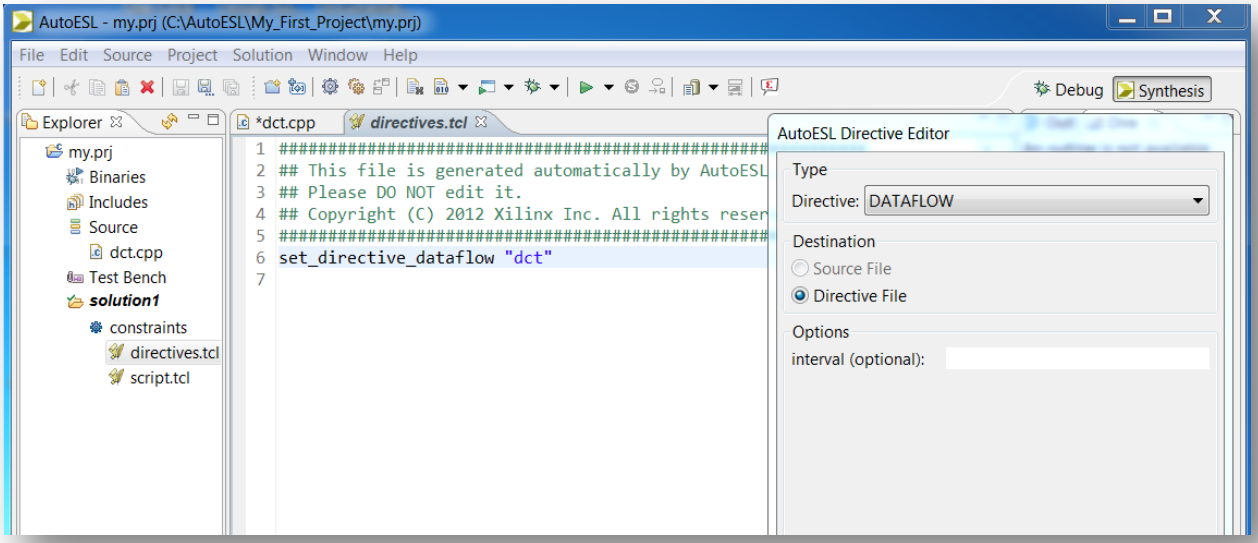

Figure 25 Adding Tcl Directives

The alternative option for directives is to add a pragma to the code. The advantage to this option is that the directive is permanently applied to the code and no additional files are required. This is an ideal approach for releasing IP and for directives which will never changed based on the technology target, such as the TRIPCOUNT directive.

Figure 26 shows the directive from the previous example being applied as a pragma, by selecting option `Source File` in the `Destination` section of Directives Editor, and the resultant source code open in the information pane.

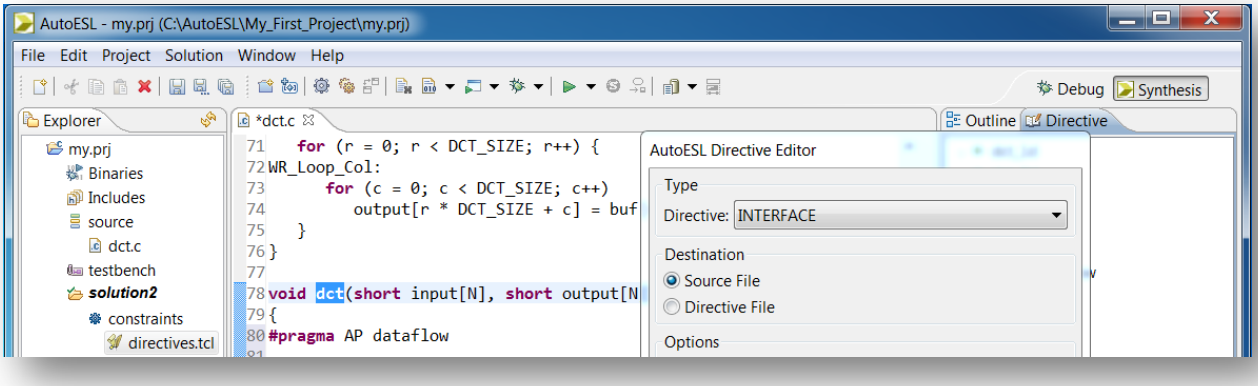

Figure 26 Adding Pragma Directives

In both cases, the directive will be applied and the optimization performed when synthesis is executed.

The only disadvantage to using pragmas is that the directive is now permanently embedded with the source code and will be used for every new solution.

First Example

A tutorial introduction on using AutoESL is available via the help menu.

The tutorial uses a design example to provide a good understanding of the following topics associated with using AutoESL:

- Perform validation of the C design
- Create an AutoESL project
- Perform Synthesis & Design Analysis
- Address Bit-accurate design
- Perform Design Optimization
- Understand how to perform RTL verification and implementation
- Review using AutoESL with Tcl scripts

The example design for use with the tutorial can be found in the `examples` directory in the AutoESL installation area.

C Validation and Coding Styles

Verification in an HLS flow can be separated into two discrete processes. Pre-synthesis validation which validates the C program correctly implements the required functionality and post-synthesis verification which verifies the RTL is correct. It is not uncommon for both processes to be referred to as simulation: C simulation and RTL simulation.

Pre-Synthesis Validation

Prior to synthesis, the function to be synthesized should be verified using a test bench. An ideal test bench has the following attributes:

- The test bench is self-checking.
- The test bench is in a separate file from the design (not a requirement, as discussed next, but advised).

Having the test bench and the function to be synthesized in separate files keeps a clean separation between the process of simulation and synthesis. If the test bench is in the same file as the function to be synthesized, there is a minor modification to the general AutoESL flow: the file with the test bench and the function to be synthesized should be added to the AutoESL project as a source file **and** as a test bench file.

Similarly, if the file with the function to be synthesized has functions above which are not in the test bench file, the file(s) with the functions above the top-level function must added to the project as a test bench file.

Typically the entire process of compiling C designs for pre-synthesis validation can be performed inside AutoESL as shown in Figure 17 and Figure 18. The C validation can however also be performed at the command line in the AutoESL Command Prompt.

C Validation outside the AutoESL GUI

Given a top-level design file "foo_top.c" and test bench file "tb_foo_top.c" the following commands can be used to compile and execute the test bench:

```
$ gcc -o foo_top foo_top.c tb_foo_top.c
$ ./foo_top
```

C++ Validation outside the AutoESL GUI

Given a top-level C++ design file "foo_top.cpp" and test bench file "tb_foo_top.cpp" the following commands can be used to compile and execute the test bench:

```
$ g++ -o foo_top foo_top.cpp tb_foo_top.cpp
$ ./foo_top
```

SystemC Validation outside the AutoESL GUI

Given a top-level design file "foo_top.cpp" and test bench file "tb_foo_top.cpp" the following commands can be used to compile and execute the test bench (the command options for the first gcc command are shown split over multiple lines for clarity but should appear on the same command line):

```
$ g++ -o foo_top foo_top.cpp tb_foo_top.cpp
         -I$AUTOESL_ROOT/Linux_x86_64/tools/systemc/include/
         -lsystemc
         -L$AUTOESL_ROOT/Linux_x86_64/tools/systemc/lib-linux64
$ ./foo_top
```

Since SystemC is being used the "systemc.h" header file must be included in all compilations.

Using a non-standard version of GCC

The version of gcc defined in Table 3 should be used to compile the C code prior to synthesis. AutoESL will create RTL to match the functionality of this version of gcc and this is the version which will be used to co-simulate the C test bench with the RTL.

AutoESL can be instructed to use a different version of gcc for RTL simulation by setting the environment variable AP_SIM_GCC prior to invoking AutoESL. The variable should be defined with the path to the directory which contains the local version of gcc.

Visual Studio Compiler

Microsoft Visual Studio Compiler (MVSC) can be used to compile the code prior to using AutoESL.

When the functions are to be compiled with AutoESL header files, such as those used with arbitrary precision integers (these are discussed in section “Arbitrary Precision Data Types” later in this chapter) there are special considerations to be aware of.

Compile C

C functions using arbitrary precision integers, as defined by AutoESL header file “ap_cint.h” must be compiled with autocc as discussed in the section “Arbitrary Precision Types with C”. MVSC cannot be used for C designs which use AutoESL arbitrary precision types.

Compiling C++

C++ functions which include AutoESL header files must have the location of the header file specified in MVSC.

To specify the location of the AutoESL header files in MVSC,

1. Click Project
2. Click Properties

3. In the panel that opens, select C/C++
4. Select general
5. Click on additional include directories and add the path as show in Figure 27.

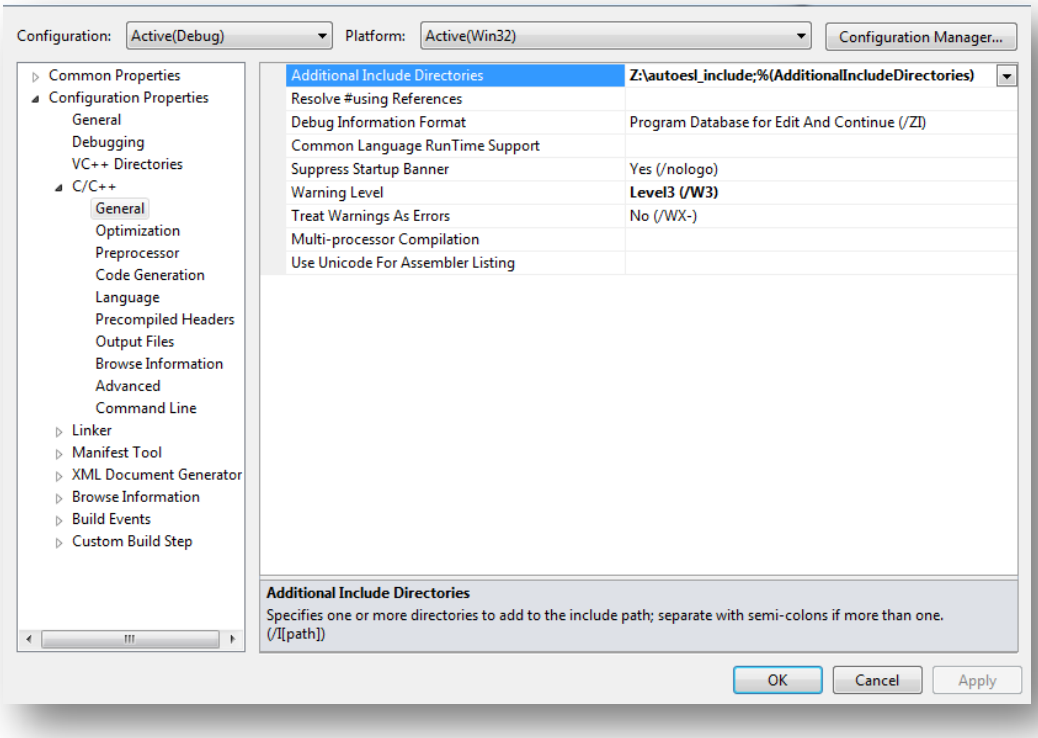

Figure 27 Setting AutoESL include path in Visual Studio

Unsupported C Language Constructs

While AutoESL is able to synthesize a large subset of all three C modeling standards (C, C++ and SystemC) there are some constructs which cannot be synthesized. This section outlines the constructs which cannot be synthesized.

In order to be synthesized, the C function must contain the entire functionality of the design (none of the functionality can be performed by system calls to the OS), the C constructs must be of a fixed/bounded size and the implementation of those constructs unambiguous. The following constructs fail to satisfy one or more of these characteristics.

System Calls

System calls cannot be synthesized since they are actions which related to performing some task upon the operating system in which the C program is running. A few examples show how system calls cannot be synthesized into anything within the hardware design itself.

- The `printf()` call prints information to the system console: this is useful during C simulation but cannot be a feature of the final hardware design and as such cannot be synthesized.
- The `fprintf()` call accesses files in the system upon which the program is executing. Again, this cannot be performed by the final hardware: access to external data must be performed via the top-level function arguments or to global variables.

Other examples of such calls are `getc()`, `time()`, `sleep()` etc. all of which make calls to the operating system.

In general, most system calls cannot be synthesized. Some commonly used system calls are automatically ignored by AutoESL (e.g. `printf` and `cout`) but in general they should be removed from consideration by synthesis by using the __SYNTHESIS__ macro.

Some system calls, such as those which allocate memory for the program to access, are part of the functionality of the design and must be both removed and transformed to maintain the functionality.

Dynamic Memory & Functions

Any system calls which manage memory allocation within the system, for example `malloc()`, `alloc()` and `free()`, must be removed from the design code prior to synthesis.

The reason for this is that they are implying resources which are created and released during runtime: to be able to synthesize a hardware implementation the design must be fully self-contained, specifying all required resources.

However, since they also typically used to define the functionality of the design, they must be transformed into equivalent bounded representation. The following examples show how dynamic memory allocations are transformed into equivalent bounded representations.

```
#ifndef __SYNTHESIS__
       // If synthesis is not required, use this code
       long long x = malloc (sizeof(long long));
       int* arr = malloc (64 * sizeof(int));
#else
       // For synthesis, use this code
       static long long x;
       int arr[64];
#end
```

The recommended approach is to make the above changes and re-execute the C simulation to verify the simulation with the synthesizable code is identical to the original (by adding the option -D__SYNTHESIS__ to the gcc or g++ compilation).

Similarly dynamic virtual function calls are not synthesizable. The following cannot be synthesized since it create new function at run time.

```
Class A {
public:
  virtual void bar() {…};
};

void fun(A* a) {
    a->bar();
}
A* a = 0;
if (base)
       A= new A();
else
       A = new B();

foo(a);
```

Pointer Casting

Pointer casting is not supported in the general case but is supported between native C types. The following is not synthesizable and must be transformed as shown in the example, where values are assigned using the original type.

```
struct {
  short first;
  short second;
} pair;
#ifndef __SYNTHESIS__
       // If synthesis is not required, use this code
       *(unsigned*)pair = -1U;
#else
       // For synthesis, use this code
       pair.first = -1U;
       pair.second = -1U;
#end
```

Recursive Functions

To create a hardware implementation the C function, AutoESL must be able to determine the resources which be required to implement the functionality. Recursive functions cannot be synthesize since the recursion may be endless (and have no bounds).

```
unsigned foo (unsigned n)
{
    if (n == 0 || n == 1) return 1;
    return (foo(n-2) + foo(n-1));
}
```

Tail recursion is synthesizable. In this example, the recursion will reach a maximum limit and is therefore synthesizable.

```
unsigned foo (unsigned m, unsigned n)
{
    if (m == 0) return n;
```

```
    if (n == 0) return m;
    return foo(n, m%n);
}
```

Standard Template Libraries

Many of the C++ Standard Template Libraries (STLs) contain function recursion and use dynamic memory allocation. For this reason the STLs cannot be synthesized.

The solution with STLs is to create a local function with identical functionality which does not exhibit these characteristics.

Top-Level Function Name Limitations

A design written in C, C++ or SystemC can use names which are illegal in the Verilog or VHDL Hardware Description Languages (HDLs).

AutoESL can automatically change names within the design to legal Verilog and VHDL names but it cannot change the name of the top-level function or the names of any arguments used in the top-level function (those which become IO ports in the RTL design).

The top-level function and the associated arguments cannot use the Verilog as listed in Table 8.

Verilog Keywords	A-E	F-N	O-S	T-W
	and	for	or	table
	always	force	output	task
	assign	forever	parameter	tran
	attribute	fork	pmos	tranif0
	begin	function	posedge	tranif1
	buf	highhz0	primitive	time
	bufif0	highhz1	pulldown	tri
	bufif1	if	pullup	triand
	bufif1	initial	pull0	trior
	case	inout	pull1	trireg
	cmos	input	rcmos	tri0
	deassign	integer	reg	tri1
	default	join	release	vectored
	defparam	large	repeat	wait
	disable	medium	rnmos	wand
	else	module	rpmos	weak0
	endattribute	nand	rtran	weak1
	end	negedge	rtranif0	while
	endcase	nor	rtranif1	wire
	endfunction	not	scalared	wor
	endprimitive	notif0	small	
	endmodule	notif1	specify	
	endtable	nmos	specparam	

Verilog Keywords	A-E	F-N	O-S	T-W
	endtask event		strong0 strong1 supply0 supply1	

Table 8 Verilog Keywords

The top-level function and the associated arguments cannot use the VHDL keywords shown in Table 9.

VHDL Keywords	A-E	F-N	O-S	T-W

	abs
	access
	after
	alias
	all
	and
	architecture
	array
	assert
	attribute
	begin
	block
	body
	buffer
	bus
	case
	component
	configuration
	constant
	disconnect
	downto
	else
	elsif
	end
	entity
	exit
		file
		for
		function
		generate
		generic
		group
		guarded
		if
		impure
		in
		inertial
		inout
		is
		label
		library
		linkage
		literal
		loop
		map
		mod
		nand
		new
		next
		nor
		not
		null
			of
			on
			open
			or
			others
			out
			package
			port
			postponed
			procedure
			process
			pure
			range
			record
			register
			reject
			return
			rol
			ror
			select
			severity
			signal
			shared
			sla
			sli
			sra
			srl
			subtype
				then
				to
				transport
				type
				unaffected
				units
				until
				use
				variable
				wait
				when
				while
				with
				xnor
				xor

Table 9 VHDL Keywords

Arbitrary Precision Data Types

C-based native data types are on 8-bit boundaries (8, 16, 32, 64 bits). RTL busses (corresponding to hardware) support arbitrary lengths. HLS needs a mechanism to allow the specification of arbitrary precision bit-width and not rely on the artificial

boundaries of native C data types: if a 17-bit multiplier is required the user should not be force to implement this with a 32-bit multiplier.

AutoESL provides arbitrary precision data types for C, C++ and supports the arbitrary precision data types which are part of SystemC.

The advantage of arbitrary precision data types is that they allow the C code to be updated to use variables with smaller bit-widths and then for the C simulation to be re-executed to validate the functionality remains identical.

The __SYNTHESIS__ macro can be used to add arbitrary precision data types to the source code while retaining the original types for reference:

```
void foo {
ifdef __SYNTHESIS__
       // use bit accurate type
       int8 a,
#else
       // Original Source
       int a,
#endif
       ...
};
```

AutoESL provides both integer and fixed point data types.

Integer Data Types

AutoESL provides arbitrary precision integer data types (Table 10) which manage the value of the integer numbers within the boundaries of the specified width.

Language	Integer data type		Required Header	
C	[u]int<precision>	(1024 bits)	autocc gcc	none #include “ap_cint.h”
C++	ap_[u]int<W>	(1024 bits)		#include “ap_int.h”
SystemC	sc_[u]int<W> sc_[u]bigint<W>	(64 bits) (512 bits)		#include “systemc.h”

Table 10 Integer data types

Arbitrary Precision Types with C

For the C language, the header file “ap_cint.h” defines the arbitrary precision integer data types `[u]int`. The “ap_cint.h” file is located in $AUTOESL_ROOT/include (where $AUTOESL_ROOT is the AutoESL installation directory).

To use arbitrary precision integer data types in a C function,

- Add header file "ap_cint.h" to the source code.
- Change the bit types to `intN` or `uintN`, where N is a bit-size from 1 to 1024.
- Compile using the `autocc` compiler
    - Select the `Use AutoCC for Compiling C Compiler` option in the GUI.
    - Use `autocc` in place of `gcc` at the command prompt

The following example shows how the header file is added and the two variables are implemented to use 9-bit integer and 10-bit unsigned integer types:

```
#include ap_cint.h

void foo_top (…) {

    int9     var1;          // 9-bit
    uint10   var2;          // 10-bit unsigned
```

Note: Standard C compilers will not correctly simulate C arbitrary precision types and the AutoESL `autocc` utility must be used.

Standard C compilers such as `gcc` will compile the attributes used in the header file to define the bit sizes, but they do know what they means. The final executable created by standard C compiler will issue messages such as the following

```
$AUTOESL_ROOT/include/etc/autopilot_dt.def:1036: warning: bit-width attribute directive ignored
```

and proceed to use native C data types for the simulation and producing results which do not reflect the bit-accurate behavior of the code.

AutoESL includes a compiler, `autocc`, which overcomes this limitation and allows the function to be compiled and verified in a bit-accurate manner.

The `autocc` compiler can be enabled in the project setting using menu Project ➔ Project Settings ➔ Simulation and select `Use AutoCC for Compiling C Files` as shown in Figure 28.

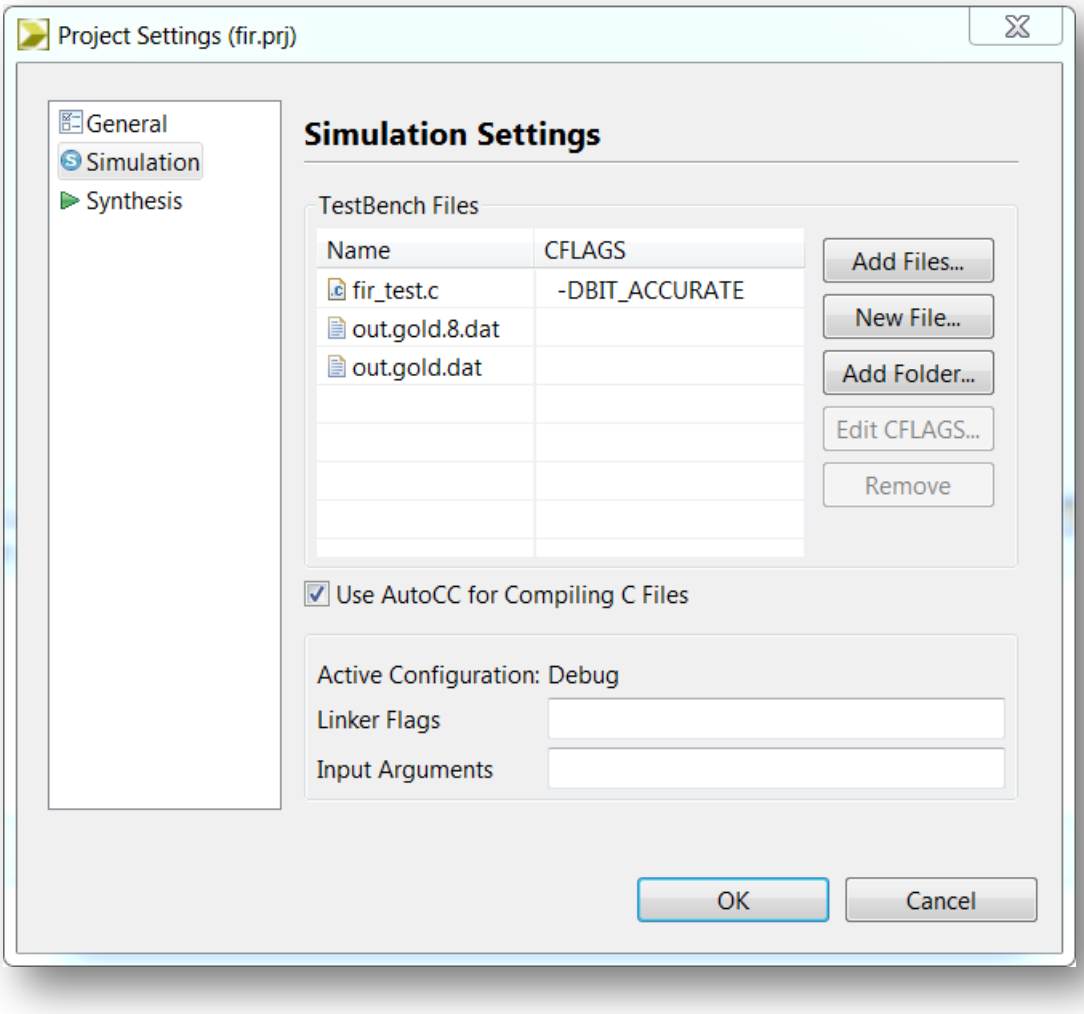

Figure 28 Enabling the AutoCC Compiler

Note: When option `Use AutoCC for Compiling C Files` is selected, the design can no longer be analyzed in the debugger: this is a side-effect of using arbitrary procession type in C code.

For functions specified using C++ or SystemC there are no such limitations when using arbitrary precision types. This limitation only exists with C, not C++ or SystemC.

AutoCC should not be used to compile C++ or SystemC functions (it will be ignored if selected).

If compiling at the command prompt, the `autocc` compiler should be used at the shell prompt: it is command line compatible with `gcc` and will process the arbitrary precision arithmetic correctly (respecting the boundaries imposed by the bit-width information).

When autocc is used the AutoESL header files are automatically included (no need to use –I$AUTOESL_ROOT/include) and the design will simulate with the correct bit-accurate behavior.

```
$ autocc -o foo_top foo_top.c tb_foo_top.c
$ ./foo_top
```

Arbitrary Precision Types with C++

For the C++ language ap_[u]int data types, the header file “ap_int.h” defines the arbitrary precision integer data. The “ap_int.h” file is located at $AUTOESL_ROOT/include (where $AUTOESL_ROOT is the AutoESL installation directory).

To use arbitrary precision integer data types in a C++ function,

- Add header file "ap_int.h" to the source code.
- Change the bit types to ap_int<N> or ap_uint<N>, where N is a bit-size from 1 to 1024.

The following example shows how the header file is added and two variables have been implemented to use 9-bit integer and 10-bit unsigned integer types:

```
#include ap_int.h

void foo_top (…) {

    ap_int<9>       var1;           // 9-bit
    ap_uint<10>     var2;           // 10-bit unsigned
```

If arbitrary precision integers are used, the simulation must include the path to header file "ap_int.h":

```
$ g++ -o foo_top foo_top.cpp tb_foo_top.cpp -I$AUTOESL_ROOT/include
```

Arbitrary Precision Types with SystemC

The arbitrary precision types used by SystemC are defined in the "systemc.h" header file which is required to be included in all SystemC designs. They include the SystemC sc_int<>, sc_uint<>, sc_bigint<> and sc_biguint<> types.

The path to the SystemC header file must be included when simulating SystemC designs. Given a top-level design file "foo_top.cpp" and test bench file "tb_foo_top.cpp" the following commands can be used to compile and execute the test bench on a Linux system (the command options for the first gcc command are shown split over multiple lines for clarity but should appear on the same command line):

```
$ g++ -o foo_top foo_top.cpp tb_foo_top.cpp
         -I$AUTOESL_ROOT]\Win_x86\tools\systemc\include
         -lsystemc
         -L$AUTOESL_ROOT\Win_x86\tools\systemc\lib
$ ./foo_top
```

Fixed Point Data Types

The use of fixed-point types is of particular importance when using HLS since the behavior of the C++/SystemC simulations performed using fixed-point data types will match that of the resulting hardware created by synthesis, allowing analysis of the effects of bit-accuracy, quantization and overflow to be analyzed with fast C-level simulation.

AutoESL offers arbitrary precision fixed point data types (Table 11) for use with C++ and SystemC functions.

Language	Fixed Point Datatype	Required Header
C	*-- Not Applicable --*	*-- Not Applicable --*
C++	ap_[u]fixed<W,I,Q,O,N>	#include "ap_fixed.h"
SystemC	sc_[u]fixed<W,I,Q,O,N>	#define SC_INCLUDE_FX [#define SC_FX_EXCLUDE_OTHER] #include "systemc.h"

Table 11 Fixed point data types

These data types manage the value of floating point numbers within the boundaries of a specified total width and integer width (Figure 29).

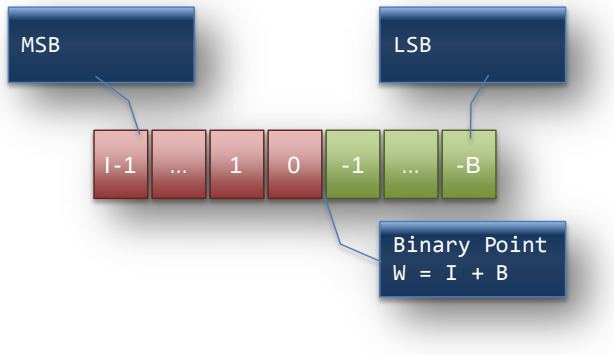

Figure 29 Fixed point data type

Table 12 provides a brief overview of operations supported by fixed point types.

Identifier	Description		
W	Word length in bits		
I	The number of bits used to represent the integer value (the number of bits above the decimal point)		
Q	Quantization mode dictates the behavior when greater precision is generated than can be defined by smallest fractional bit in the variable used to store the result.		
	SystemC Types	**AP_Fixed Types**	**Description**
	SC_RND	AP_RND	Rounding to plus infinity

Identifier	Description		
	SC_RND_ZERO	AP_RND_ZERO	Rounding to zero
	SC_RND_MIN_INF	AP_RND_MIN_INF	Rounding to minus infinity
	AP_RND_INF	AP_RND_INF	Rounding to infinity
	AP_RND_CONV	AP_RND_CONV	Convergent rounding
	AP_TRN	AP_TRN	Truncation to minus infinity
	AP_TRN_ZERO	AP_TRN_ZERO	Truncation to zero (default)
O	Overflow mode dictates the behavior when more bits are generated than the variable to store the result contains.		
	SystemC Types	**AP_Fixed Types**	**Description**
	SC_SAT	AP_SAT	Saturation
	SC_SAT_ZERO	AP_SAT_ZERO	Saturation to zero
	SC_SAT_SYM	AP_SAT_SYM	Symmetrical saturation
	SC_WRAP	AP_WRAP	Wrap around (default)
	SC_WRAP_SM	AP_WRAP_SM	Sign magnitude wrap around
N	The number of saturation bits in wrap modes.		

Table 12 Fixed Point Identifier Summary

ap_fixed

In this example the AutoESL `ap_fixed` type is used to define an 18-bit variable with 6 bits representing the numbers above the decimal point and 12-bits representing the value below the decimal point. The variable is specified as signed, rounding is specified as the quantization mode and the default wrap-around mode is used for overflow.

```
#include <ap_fixed.h>
...
ap_fixed<18,6,AP_RND > my_type;
...
```

sc_fixed

In this `sc_fixed` example a 22-bit variable is shown with 21 bits representing the numbers above the decimal point: enabling only a minimum accuracy of 0.5. Rounding to zero is used, such that any result less than 0.5 will round to 0 and saturation is specified.

```
#define  SC_INCLUDE_FX
#define SC_FX_EXCLUDE_OTHER
#include <systemc.h>
...
sc_fixed<22,21,SC_RND_ZERO,SC_SAT> my_type;
...
```

Floating Point Types

To synthesize any design, AutoESL coverts the operations in the design into operators which are then mapped to cores from the technology library. For floating point designs not all operators have an associate floating point core in the library.

If there is no core in the technology library which can be mapped to, AutoESL synthesis will halt synthesis with a message that there is no library core to map to. A complete list of core in the AutoESL library is provided in the "AutoESL Library Guide".

Floating Point Arithmetic

In order to use a floating point core from the library, all arguments of the operation must be floating point argument. The following example code,

```
A = B/2;
```

Must be converted to

```
A =B * 0.5;
```

in order for a floating point operator to be used (for the multiplication). AutoESL will not automatically convert the constant data type.

The standard arithmetic operations (+, -, *, / and %) are supported by floating point cores in the technology library. Simply defining a variable as a float and then using it with the standard arithmetic operators will result in a floating point core being used in the implementation.

Floating Point Math Functions

For C/C++ math functions, the function must be declared. For example, to use the `sqrtf()` function the following must be added to the code:

```
#include <math.h>
extern "C" float sqrtf(float);
```

The `sqrtf()` function can then be used with floating point variables.

Multi-Access Pointer Interfaces

Designs which use pointers in the argument list of the top-level function need special consideration when multiple accesses are performed using the pointers. Multiple accesses occur when a pointer is read from or written to, multiple times in the same function.

In the following example, (input) pointer "d_i" is read from four times and (output) "d_o" is written to twice.

```
#include "fifo.h"
```

```
void fifo ( int *d_o,  int *d_i) {
  static int acc = 0;
  int cnt;

    acc += *d_i;
    acc += *d_i;
    *d_o = acc;
    acc += *d_i;
    acc += *d_i;
    *d_o = acc;

}
```

When multi-access pointers are used, the following must be performed:

- It is a requirement to use the `volatile` qualifier.
- Specify the number of accesses on the port interface if verifying the RTL with `autosim`.
- Be sure to validate the C prior to synthesis to confirm the intent and the C model is correct

Understanding Volatile Interfaces

The code above is written with intent that input pointer "d_i" and output pointer "d_o" will be implemented in RTL as interfaces with handshakes (such as FIFO or handshake ports) which will ensure:

- Upstream producer blocks will supply new data each time a read is performed on RTL port "d_i".
- Downstream consumer blocks will accept new data each time there is a write to RTL port "d_o".

However, when this code is compiled by C compilers, the multiple accesses to each pointer will be reduced to a single access: as far as the compiler is concerned, there is no indication that the data on "d_i" changes during the execution of the function and only the final write to "d_o" is relevant (the other writes will be over-written by the time the function completes).

AutoESL will match the behavior of the C compiler and optimize these reads and writes into a single read operation and a single write operation.

This design can be made to work as intended at the RTL by using the `volatile` qualifier in the code, as shown in the next example.

```
#include "fifo.h"

void fifo ( volatile int *d_o,  volatile int *d_i) {
  static int acc = 0;
  int cnt;

    acc += *d_i;
    acc += *d_i;
    *d_o = acc;
```

```
    acc += *d_i;
    acc += *d_i;
    *d_o = acc;

}
```

The `volatile` qualifier tells the C compiler, and AutoESL, to make no assumptions about the pointer accesses (the data is volatile and may change and the compiler should therefore not optimize pointer accesses).

This example results in an RTL design which will perform the expected four reads on input port "d_i" and two writes to output port "d_o".

However, even if the `volatile` keyword is used, this coding style (accessing a pointer multiple times) introduces two additional issues associated with the test bench and verification.

It can easily result in:

- RTL `autosim` simulation failures.
- Modeling and Simulation mismatches.

RTL autosim simulation failures

The following test bench can be used to validate the algorithm above. This test bench models four executions of the function, or four transactions, to highlight the operation of the code.

Note: This test bench is not self-checking: the self-checking code is omitted for clarity.

```
#include <stdio.h>

#include "foo.h"

int main () {
  int d_o, d_i;

  for (d_i=0;d_i<4;d_i++) {
    foo(&d_o,&d_i);
    printf("%d %d\n", d_i, d_o);
  }

  return 0;
}
```

The header file “foo.h” used with this example would be the following, where the function is simply declared, and allowing function “foo” to be defined in a separate file:

```
#ifndef FOO_H_
#define FOO_H_
```

```
void foo ( volatile int *d_o,  volatile int *d_i);

#endif
```

The issue with this test bench is that it only supplies a single value to the function each transaction.

In each transaction, the test bench will apply a single value, but since the `volatile` keyword is used, the function `foo` will perform four reads and hence will read the same value four times.

The test bench will validate the algorithm with the following results, showing the output is the accumulation of four input reads plus the accumulation (or output) from the previous transaction:

```
Di Do
0  0
1  4
2  12
3  24
```

When RTL verification is performed the `volatile` qualifier tells AutoESL to create RTL which performs four reads and if synthesized with a handshake interface, this will ensure the interface signals for new data each time port “d_i” is read.

To verify the RTL with `autosim`, AutoESL creates a SystemC wrapper with adapters around the RTL and instantiates this (C code) wrapper into the existing C test bench, as shown in Figure 30.

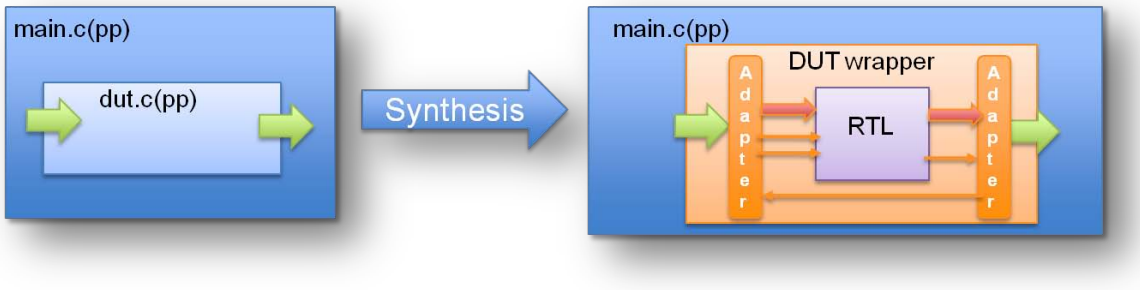

Figure 30 Autosim Wrapper Overview

The wrapper created by AutoESL models any required handshakes on the RTL interface and as such must ensure the input values to the DUT, presented by the test bench, are ready when required by the RTL design. This requires storage.

AutoESL cannot determine from this type of function interface, using pointers, how many reads or writes are performed. Neither of the arguments in the function interface tells AutoESL how many values will be read or written.

```
void foo ( volatile int *d_o,  volatile int *d_i)
```

Unless something on the interface informs AutoESL as to how many values are required, such the maximum size of an array, AutoESL will assume a single value and only create simulation wrappers for a single input and single output.

If the RTL ports are actually reading or writing multiple values, this will result in the RTL `autosim` simulation stalling: since the wrapper is modeling the producer and consumer blocks which will be connected to the RTL design, the RTL design will try to read or write the next value but the handshake interfaces will tell the design to wait, since there is currently no value to read or no space to write.

When multi-access pointers are used at the interface, AutoESL must be informed of the maximum number of reads or writes on the interface. When specifying the interface, use the depth option on the INTERFACE directive as shown in Figure 32.

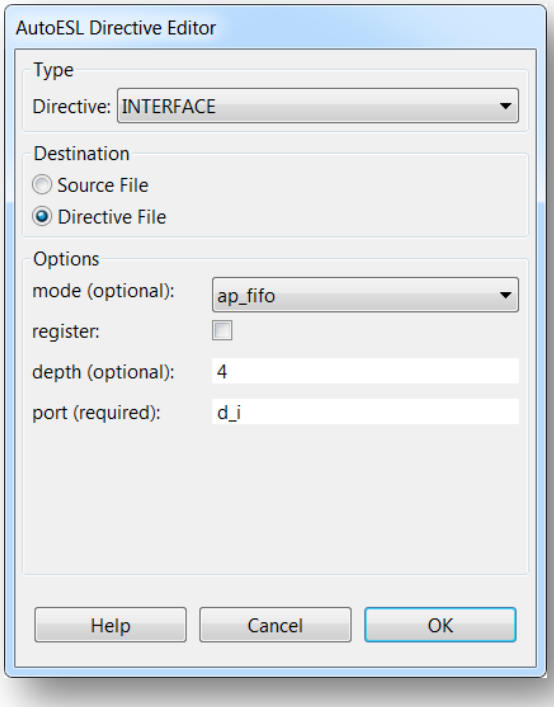

Figure 31 Interface Directive Dialog: Depth Option

In the above example, argument/port "d_i" is set to have a (FIFO interface) depth of 4, ensuring that `autosim` will provide enough values to correctly verify the RTL.

Simulation Mismatches and C Modeling

Once the ports are defined to have a maximum depth and `autosim` can be used to verify the RTL crated from the function, the next issue will be a simulation mismatch.

The RTL input interface will not supply the same four values in each transaction as the test bench does. At the RT level, the wrapper (modeling the RTL producer block the design will eventually be connected to) will supply a new value each time one is

requested by the handshake interface. It therefore supply 4 values in the first transaction of this example: 0 the value supplied by the test bench and then three undefined values , x, y and z since the test bench only provided one value to the RTL wrapper.

There will be a simulation mismatch between the C and RTL.

> **Note:** The issue here is the inability of this C code and test bench to correctly model multiple a situation where multiple reads or writes is required.

At the start of this example, the function was verified by reading the same four values from the test bench. This was done with the assumption that at the RT level, the data would be updated but somehow things would work out. This was the mistake.

This example was not contrived. It is something seen all too often with C function to be synthesized into hardware: the code is often written and synthesized but it does not execute as expected and a great deal of time is wasted debugging RTL.

Note: Always validate the C code with a C simulation prior to synthesis.

The limitations of this coding style can be overcome by re-writing the code.

The code shown in the next example has been updated to ensure it will read four unique values from the test bench. This is achieved by having the code access explicit values defined in the test bench. Since the pointer accesses are sequential and start at location zero, a streaming interface type can be used during synthesis.

```
#include "foo.h"

void foo ( volatile int *d_o,  volatile int *d_i) {
  static int acc = 0;
  int cnt;

    acc += *d_i;
    acc += *(d_i+1);
    *d_o = acc;
    acc += *(d_i+2);
    acc += *(d_i+3);
    *(d_o+1) = acc;

}
```

The test bench is updated to model the fact that the function will read four unique values in each transaction. (Note, to keep the example small and easy to understand, this new test bench only models a single transaction and the self-checking code is omitted).

```
#include "foo.h"
```

```
int main () {
  int d_o[4], d_i[4];
  int i;

  for (i=0;i<4;i++) {
    d_i[i]=i;
  }

  foo(d_o,d_i);

  printf("Di Do\n");
  for (i=0;i<4;i++) {
    if (i<2)
       printf("%d %d\n", d_i[i], d_o[i]);
    else
       printf("%d\n", d_i[i]);
  }

       return 0;
}
```

The test bench will validate the algorithm with the following results, showing there are two outputs from a single transaction and they are an accumulation of the first two input reads, plus an accumulation of the next two input reads and the previous accumulation:

```
Di Do
0 1
1 6
2
3
```

This design will synthesize and the RTL will be verified by `autosim` if the port interfaces are set to a depth of four (port “d_i”) and two (port “d_o”).

However, this updated example suffers from an additional limitation. Each time the function is called, it will start reading data from position 0 in the external array (d_i[0] in the test bench). This function requires that all data be available to read when the function is called

Streams can be used to work around this limitation. AP_STREAMs are a coding construct provided by AutoESL, which allows streaming data, and hence applications which use streaming data, to be more easily modeled in C.

Since, as has been seen in this example, streaming data is not easily modeled in C but is often used in hardware designs (video, communications, etc.), AP_STREAMs may be the most intuitive way to model the hardware in C. These are discussed in the “Coding with Streams” section, discussed next.

Coding Techniques for Modeling Hardware

C code is written to satisfy a number of different requirements: reuse, readability, performance. Until now, it is unlikely the C code was written to result in the most ideal hardware after High-Level Synthesis.

However, like the requirements for reuse, readability and performance, certain coding techniques or pre-defined constructs can be used to ensure the synthesis output results in more optimal hardware or to better model hardware in C for easier validation of the algorithm.

User Defined Registers in C++

In general, the only variables in a C function which are guaranteed to be implemented as registers are those proceeded by the `static` qualifier, those defined in the global scope (unless the global is exposed as a port) and arrays which are targeted to memory resources.

For the other variables, they may be implemented in a register, or they may not: it depends on the decisions made during of synthesis. Synthesis may determine that a particular variable should be registered over multiple cycles, or it may determine the variable can be used in the same cycle it's created and therefore is not required to be registered. Only the variable types mentioned above are guaranteed to be registers in the RTL.

The C++ function, "Reg", used in the following example is a useful technique to guarantee (or force) that a particular variable is a register in the final RTL design.

```
#include "foo.h"

template<class T>
T Reg(T in) {
#pragma AP INLINE off
#pragma AP INTERFACE port=return register
  return in;
}

int foo (int in1, int in2 ) {
  int tmp=in1*(0x0800-in2);
  // int out = tmp >> 10;
  int out = Reg(tmp >> 10);
  return( out );
}

int foo_top(int inA, int inB) {

  int res1 = foo(inA, inB);
  return(res1);
}
```

The important aspects of this code are:

- Function “Reg” is created and has its output, the function `return` value, registered.
    - Registering function arguments (the RTL ports) is the only allowed use of the interface directive on sub-functions. The interface directive cannot be used to specify the IO protocol of a sub-function argument.
- Function “Reg” has inlining disabled in case AutoESL decides to automatically inline this small function.
    - Inlining function “Reg” negate the register operation performed on the function, since it would no longer exist as a separate function if it is inlined.
- The output of the multiplier and shift operation in sub-function “foo” is used as the input to the “Reg” function.
    - This guarantees the result of this operation is registered in the RTL output, since the function “Reg” has a registered output.
- The original source code for this variable is shown commented out. This is shown how easy it is to apply this technique

In general, AutoESL will determine which variables should be registered in the final RTL, but this coding technique can be used to force particular variables to be registered without modifying the functionality of the code.

Coding with Streams

Many hardware designs, from video to communication designs, process the data as streaming data. Streaming data is data which is read or written in a sequential manner: there are no arbitrary or random data accesses.

A natural way to code such streaming designs is to use pointers; however because this requires accessing pointer multiple times, as was seen in the chapter “Multi-Access Pointer Interfaces”, this can unexpected results if the `volatile` qualifier is missing from the code and even when specified correctly can cause issues with validation and RTL verification. Streaming data can however be safely and concisely modeled using the AutoESL AP_STREAM construct.

An AP_STREAM is a predefined coding construct which behaves like a FIFO of infinite size in the C code, allowing applications which rely on streaming data to be easily modeled in C, verified efficiently and result in efficient hardware after synthesis.

The easiest way to understand AP_STREAMs is to review an example.

In this example, two streams are defined in function “foo”. The data is written into one stream (“d_i”) in the test bench and consumed in the function, while the other stream (“d_o”) is written to in the function and read in the test bench.

The header file “foo.h” is as follows:

```
#ifndef FOO_H_
#define FOO_H_

#include "ap_stream.h"
```

```
void foo (int cnt);

#endif
```

- This header file, to be included by both the function to be synthesized and test bench, ensures the AP_STREAM header file “ap_stream.h” is defined in both the function and test bench.
- The AP_STREAM header file resides in the “include” directory of the AutoESL installation and defines the AP_STREAM macros.

Streams are defined in the global scope and do not appear as arguments in any function. Streams are automatically exposed as ports unless they are defined with the `static` qualifier.

The design:

```
#include "foo.h"

AP_STREAM(int,d_o);
AP_STREAM(int,d_i);

void foo (int cnt) {
  static int acc = 0;
  int data;
  int i;

  for (i=0;i<cnt;i++) {
    AP_STREAM_READ(d_i,data);
    acc += data;

    if (i%2==1) {
      AP_STREAM_WRITE(d_o,acc);
    }
  }
}
```

- Two streams, called “d_i” and “d_o”, are defined outside of the function in the global scope; in this example, both use integer (int) types.
- Stream are defined in the global scope and do not appear as arguments to function.
- Samples are read from input stream “d_i” into local variable “data”.
- The value of local variable “acc” is written into the output stream “d_o”.

Since the streams in the design were not defined with the `static` qualifier, they will be exposed as ports in the RTL design after synthesis. The test bench should be used to fill the input streams with data and empty and verify the data in the output streams.

To communicate between the test bench and the design, the macro AP_STREAM_INTERFACE is used. This macro allows an AP_STREAM defined in the design file, to be accessed in the test bench file (the stream itself cannot be defined

in two places). If the test bench and the design are in the same file, the AP_INTERFACE_STREAM declaration is not required.

To keep this example simple, this test bench is not self-checking but instead simply prints the values: use a self-checking test bench to ensure the RTL is automatically verified.

The test bench:

```
#include "foo.h"

AP_STREAM_INTERFACE(int,d_o);
AP_STREAM_INTERFACE(int,d_i);

  int main () {
  int data;
  int i;

  for (i=0;i<10;i++) {
    AP_STREAM_WRITE(d_i,i);
  }

   foo(4);

  printf("Unprocessed data:\n");
  while (!AP_STREAM_EMPTY(d_i)) {
    AP_STREAM_READ(d_i,data);
    printf("%d\n", data);
  }

  printf("Output data:\n");
  while (!AP_STREAM_EMPTY(d_o)) {
    AP_STREAM_READ(d_o,data);
    printf("%d\n", data);
  }
  return 0;
}
```

- Since both the streams are to be exposed as RTL ports (they were not defined with the `static` qualifier) use the AP_STREAM_INTERFACE macro to communicate between the test bench and the design.
- Data is written into stream “d_i”.
- After the function executes, stream “d_i” is checked to ensure all values were read by the function “foo”: any values which were not read remain in the stream.
- Finally, stream “d_o” is read and the output data is verified (this test bench omits the results checking: always check the results, as this ensures the RTL simulation results are checked automatically).

Note: If a stream is defined without the `static` qualifier it will be implemented as a port in the RTL. However, to communicate between the

test bench and the design, the stream must be defined as an AP_INTERFACE_STREAM in the test bench.

Simply defining the streams as ports is not enough to communicate with the test bench when performing C validation.

The following explains, in detail, all the attributes of stream and each aspect of using them.

Defining Streams

When streams are implemented in the RTL, they result in a FIFO which can use either a standard FIFO interface (input and output data, empty and full controls) or a standard handshake interface (input and output data, valid and acknowledge controls).

The type of interface used in the stream is determined by the how the stream is defined, as shown in Table 13.

Macro	Function
AP_STREAM(type, name)	Defines a named stream of the specified data type. After RTL synthesis the stream communicates using a FIFO protocol and has a depth of 1.
AP_STREAM_HS(type, name)	Defines a named stream of the specified data type. After RTL synthesis the stream communicates using a two-way handshake protocol and has a depth of 1.

Table 13 Defining Streams

As stated earlier, an AP_STREAM behaves like a FIFO of infinite size in the C code. Streams created using the macros in Table 13 are implemented in the RTL as FIFOs with a depth of one.

Defining Streams: RTL Sizes

In most applications which use streaming data, the function or loop is pipelined with a throughput of one and the stream implementation (a FIFO in the RTL) only requires a depth of one.

However, accesses to a stream can be conditional: in some executions of the function there may be more writes than reads. In this case, the FIFO in the RTL may need to hold more than a single value.

An AP_STREAM can be implemented in the RTL using a FIFO of any arbitrary size by using the macros shown in Table 14 (in place of those shown in Table 13). If the stream implementation FIFO is required to be greater than one, failure to specify the size correctly will result in a simulation mismatch when verifying the RTL.

Macro	Function
AP_STREAM_SIZE(type, name, depth)	Defines a named stream of the specified data type and user specified depth. After RTL synthesis the stream communicates using a FIFO protocol and has the specified user defined depth.
AP_STREAM_HS_SIZE(type,name,depth)	Defines a named stream of the specified data type and user specified depth. After RTL synthesis the stream communicates using a two-way handshake protocol and has the specified user defined depth.

Table 14 Defining Stream Sizes in the RTL

Interface and Local Streams

As shown in the example above, streams are not defined as arguments to any function in the way other input and output are. If the stream is to be implemented at the top-level interface, the appropriate macro from Table 13 or Table 14 should be used to define the stream. In addition, the stream should be accessed in the test bench using the AP_STREAM_INTERFACE macro defined in Table 15.

Any struct used in a stream which is exposed as an interface will be decomposed into individual ports in the RTL interface.

If a stream will only be used inside the function, and it will not be implemented on the function interface, it should be defined with the `static` qualifier as shown in Table 15.

> Note: If a stream is declared with the static qualifier, any functions in which it is used must be defined in the same file: global statics are limited to the scope of the file in which they appear.

Macro	Function
AP_STREAM_INTERFACE(type, name)	Allows the named stream and type to be accessed by the test bench.
static AP_STREAM(type, name), static AP_STREAM_HS(type, name) static AP_STREAM_SIZE(type,name, size) static AP_STREAM_HS_SIZE(type, name, size)	If the stream is only to be used internally, and will not appear on the function interface, it should be defined as shown in Table 13 or Table 14 but with the static qualifier shown here (and in the global scope outside of any function).

Table 15 Defining Local Streams

It is highly recommended to use AP_STREAM_SIZE or AP_STREAM_HS_SIZE for internal streams and size them to the maximum possible value. Then, when the actual maximum size can be determined via RTL simulation, reduce the specified size to the minimum required.

Blocking and Non-Blocking RTL behavior

Streams can support both blocking and non-blocking behavior in the RTL.

- Blocking reads: If the stream is empty, the RTL implementation will stall and wait for new data to become available.
- Blocking writes: If the stream is full, the RTL implementation will stall and wait for space to become available.

- Non-Blocking reads: If the stream is empty, the RTL implementation will not read any new data (existing values will not be updated) but will continue operation.
- Non-Blocking writes: If the stream is full, the RTL implementation will not write the current value into the stream but will continue operation.

Any stall will complete when the steam interface signals indicate that new data is available to read or that the consumer has space available to write.

> **Note**: In the C code, all AP_STREAMs will execute as non-blocking: the C simulation will never stall.

To define if the read or write access to the AP_STREAM will be blocking or non-blocking in the RTL, the macros Table 16 can be used.

Macro	Function	Details
AP_STREAM_WRITE(name, variable)	Will write the variable value into the named stream.	The RTL write operation will stall if the stream is full: will wait until it can write.
AP_STREAM_READ(name, variable)	Read from the name stream and store it in variable.	The RTL read operation will stall and wait, if the stream is empty.
AP_STREAM_WRITE_NB(name, variable)	Non-Blocking: Will write the variable value into the named stream.	If the stream is full, there will be no write operation in the RTL, the design will continue to operate and the data will be lost.
AP_STREAM_READ_NB(name, variable)	Non-Blocking: Read data from the named stream and store it in variable.	If the stream is empty, nothing is read in the RTL and the design will continue to execute.

Table 16 Accessing Streams

Querying Streams

Streams can be queried, to determine if they are full or empty, as shown in Table 17. The query macros make streams ideal for cases where control logic is required based upon the level of activity in the stream.

Macro	Function
AP_STREAM_FULL(name)	Returns TRUE if the named stream is full and FALSE if more data can be written to the stream
AP_STREAM_EMPTY(name)	Returns TRUE if the named stream is empty (has no data) and FALSE if the stream contains data to read.

Table 17 Query Streams

Mapping Directly into SRL resources

Many C algorithms sequentially shift data through arrays: add a new value to the start of the array, shift the existing data through array and drop the oldest data value. This operation is implemented in hardware as a shift-register.

This most common way to implement a shift-register from C into hardware is to completely partition the array into individual elements, and allow the data dependencies between the elements in the RTL to imply a shift-register.

Logic synthesis will typically implement the RTL shift-register into a Xilinx SRL resource, which efficiently implements shift-registers. The problem is that sometimes logic synthesis does not implement the RTL shift-register using an SRL component:

- When data is accessed in the middle of the shift-register, logic synthesis cannot directly infer an SRL.
- Sometimes, even when the RTL is ideal, logic synthesis may implement the shift-resister in flip-flops, due to other factors. (Logic synthesis is also a complex process).

AutoESL provides a C++ class, `ap_shift_reg`, which ensures the shift-register defined in the C code, is always implemented using an SRL resource. The `ap_shift_reg` class has two methods to perform the read and write accesses.

Read From the Shifter

The `read` method allows a specified location to be read from the shifter register.

```
// Include the Class
#include "ap_shift_reg.h"

// Define a variable of type ap_shift_reg<type, depth>
// - Sreg must use the static qualifier
// - Sreg will hold integer data types
// - Sreg will hold 4 data values
static ap_shift_reg<int, 4> Sreg;
int var1;

// Read location 2 of Sreg into var1
var1 = Sreg.read(2);
```

Read and Shift Data

A `shift` method allows a read and shift operation to be performed.

```
// Include the Class
#include "ap_shift_reg.h"

// Define a variable of type ap_shift_reg<type, depth>
// - Sreg must use the static qualifier
// - Sreg will hold integer data types
// - Sreg will hold 4 data values
static ap_shift_reg<int, 4> Sreg;
int var1;

// Read location 3 of Sreg into var1
// THEN shift all values up one and load In1 into location 0
var1 = Sreg.shift(In1,3);
```

Read and Enable-Shift

The shift method also supports an enable input, allowing the shift process to be controlled/enable by a variable.

```
// Include the Class
#include "ap_shift_reg.h"

// Define a variable of type ap_shift_reg<type, depth>
// - Sreg must use the static qualifier
// - Sreg will hold integer data types
// - Sreg will hold 4 data values
static ap_shift_reg<int, 4> Sreg;
int var1, In1;
bool En;

// Read location 3 of Sreg into var1
// THEN if En=1
// Shift all values up one and load In1 into location 0
var1 = Sreg.shift(In1,3,En);
```

When using the ap_shift_reg class, AutoESL will create a unique RTL component for each shifter. When logic synthesis is performed, this component is synthesized into an SRL resource.

The SRL implementation will be shown in the ISE output:

```
Processing Unit <foo_top> :
Found 3-bit shift register for signal <..._U/ShiftRegMem_3_63>.
Found 3-bit shift register for signal <..._core_U/ShiftRegMem_3_62>.
Found 3-bit shift register for signal <..._core_U/ShiftRegMem_3_61>.

Etc..

# Shift Registers                                       : 32
```

```
3-bit shift register                                : 32
```

Interface Management

In C based design, all input and output operations are performed, in zero time, through formal function arguments. In an RTL design these same input and output operations must be performed via a port in the design interface and must operate using a specific IO (input-output) protocol.

AutoESL supports two solutions for specifying the type of IO protocol used:

- Interface synthesis, where the port interface is automatically created based on efficient, safe and standard interfaces.

- Manual interface specification where the interface behavior is explicitly described in the input source code. This allows any arbitrary IO protocol to be used, hence allows the function to interface with any hardware resource.
    - This solution is more typical with SystemC designs, where the IO control signals are specified in the interface declaration and their behavior specified in the code.
    - AutoESL also supports this mode of interface specification for C and C++ designs. This is detailed later in this chapter.

Interface Synthesis

When a C program is synthesized into an RTL design, the C arguments are synthesized into RTL data ports. Interface synthesis allows an interface protocol to be automatically added to the RTL data port. The interface protocol could be as simple as an output valid signal indicating when an output is ready or it could include all the ports required to interface with a BRAM, such that the date could be read from or written to a BRAM.

The type of interfaces which can be created by interface synthesis depend on the C argument. For example, for an output valid signal to be created by interface synthesis, the C argument must be a point or C++ reference, since pass-by-value scalars can only be inputs. Table 18 summarizes the types of interface which are supported for each type of C function argument.

> **Note**: Interface synthesis is not generally supported for SystemC designs.
>
> In SystemC designs all IO protocol signals are declared in the interface declaration and their behavior is fully specified in the code.
>
> The exception to this is for memory interfaces, as discussed in “SystemC Interface Synthesis”.

If no interface type is specified for the port, the interface will be implemented with the default interface as detailed in Table 18. If an unsupported interface type is specified, as also shown in Table 18, AutoESL will issue a warning message and revert to the default interface type.

Argument Type	Variable			Pointer Variable			Array			Reference Variable		
	Pass-by- value			Pass-by-reference			Pass-by-reference			Pass-by-reference		
Interface Type	I[1]	IO[2]	O[2]	I	IO	O	I	IO	O	I	IO	O
ap_none	D			D						D		
ap_stable												
ap_ack												
ap_vld						D						D
ap_ovld[3]					D						D	
ap_hs												
ap_memory							D	D	D			
ap_fifo												
ap_bus												
ap_ctrl_none[4]												
ap_ctrl_hs[4]			D									

Key:
I : input
IO : inout
O : output
D : Default Interface

Supported Interface

Unsupported Interface

Table 18 Data Type and Interface Synthesis Support

The notes in Table 18 are explained as follows:

1. The concept of inputs and outputs is somewhat different between C functions and RTL blocks. The following convention is used here for the purposes of explaining interface synthesis:
   - A function argument which is read and never written to, like an RTL input port, is referred to as an input (I).
   - A function argument which is both read and written to, like an RTL inout port, is referred to as an inout (IO)
   - A function argument which is written to but never read, like an RTL output port, is referred to as an output (O)

2. A standard pass-by-value argument cannot be used to output a value to the calling function. The value of an argument such as this can only be returned (or output from the function) by the function `return` statement.
    - Any pass-by-value function argument which is written to but never read, (like an RTL output port) will be synthesized as an RTL input port with no fanout.
    - Any pass-by-value function argument which is written to and read, (like an RTL inout port) will be synthesized as an RTL input port only.
3. The `ap_ovld` interface type is only valid for output ports.
4. The interface types `ap_ctrl_none` and `ap_ctrl_hs` are used to control the synthesis of function level interface protocols. These interface types are specified on the function itself (all other interface types are specified on the function arguments).

Interface Types

This section details each of the interface types supported by AutoESL. The details on how to specify an interface are discussed after this section: first comes an explanation of the interfaces.

There are two distinct types on interface synthesis. Interface synthesis which is performed on C function arguments and interface synthesis which is performed at the function or block level.

Block level interface synthesis applies an IO protocol to the entire block, adding control signals to control when the block can begin operation, when it is ready for new data and when it completes. Block level synthesis is controlled by interface modes `ap_ctrl_hs` and `ap_ctrl_none`, which are applied to the function or the function return port.

Standard port level interface synthesis is specified by applying the appropriate interface mode to a function argument. A function argument which is both read from and written to (an RTL inout port) is synthesized in the following manner:

- For interface types `ap_none`, `ap_stable`, `ap_ack`, `ap_vld`, `ap_ovld` and `ap_hs`: as separate input and output ports. For example, if function argument `arg1` was both read from and written to, it would be synthesized as RTL input data port `arg1_i` and output data port `arg1_o` and any specified or default IO protocol is applied to each port individually.
- For interface types `ap_memory` and `ap_bus`: a single interface is created. Both these RTL interfaces support read and write.
- For interface type `ap_fifo`: read and write are not supported for ap_fifo interfaces.

Structs on the interface are flattened and all hierarchy is removed before being implemented as ports: the first argument in the struct hierarchy is implemented in the LSBs of the port and the last argument implemented in the port MSBs. The implementation of arrays inside structs depends on whether the struct is a pass-by-value or pointer argument.

- In pass-by-value structs, arrays are completely scalarized with all elements inlined to create single-wide bus.
- In pointer structs, arrays ports are maintained and can be implemented in the same manner as any other array (as discussed below).

If a design is to be verified using the `autosim` feature, the following must hold true:

- The design must use block-level handshakes, as specified by `ap_ctrl_hs`.
- Each output port must use an interface type which indicates when a write operation has occurred: `ap_vld`, `ap_ovld`, `ap_hs`, `ap_memory`, `ap_fifo` or `ap_bus`.

The default interface types ensure the design can be verified by the `autosim` feature. SystemC designs do use interface synthesis and there are no such requirements to verify a SystemC design with `autosim`.

In the following explanations, producer blocks are those RTL blocks which provide data to the current block inputs and consumer blocks are those which consume the output data of the current block.

ap_ctrl_none & ap_ctrl_hs

Interface types `ap_ctrl_none` and `ap_ctrl_hs` are used to specify if the RTL is implemented with block-level handshake signals or not. Block-level handshake signals specify when the design can start to perform its standard operation and when that operation ends. These interface types are specified on the function or the function return.

Figure 32 shows the resulting RTL ports and behavior when `ap_ctrl_hs` is specified on a function (note, this is the default operation). In this example the function returns a value using the `return` statement and thus output port ap_return is created in the RTL design: if there is no function `return` statement this port is not created.

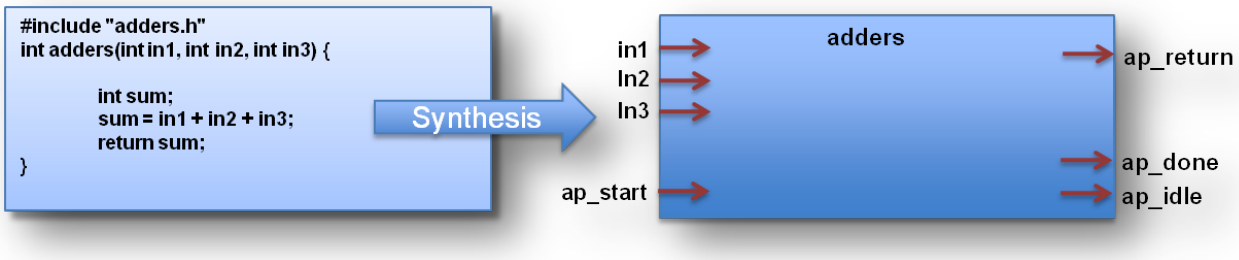

Figure 32 Example ap_ctrl_hs interface

If `ap_ctrl_none` is specified, none of the handshake signal ports ("ap_start", "ap_idle" and "ap_done") shown in Figure 32 are created and the block cannot be verified with the `autosim` feature.

The behavior of the block level handshake signals created by interface mode `ap_ctrl_hs` are shown in Figure 33 and summarized below.

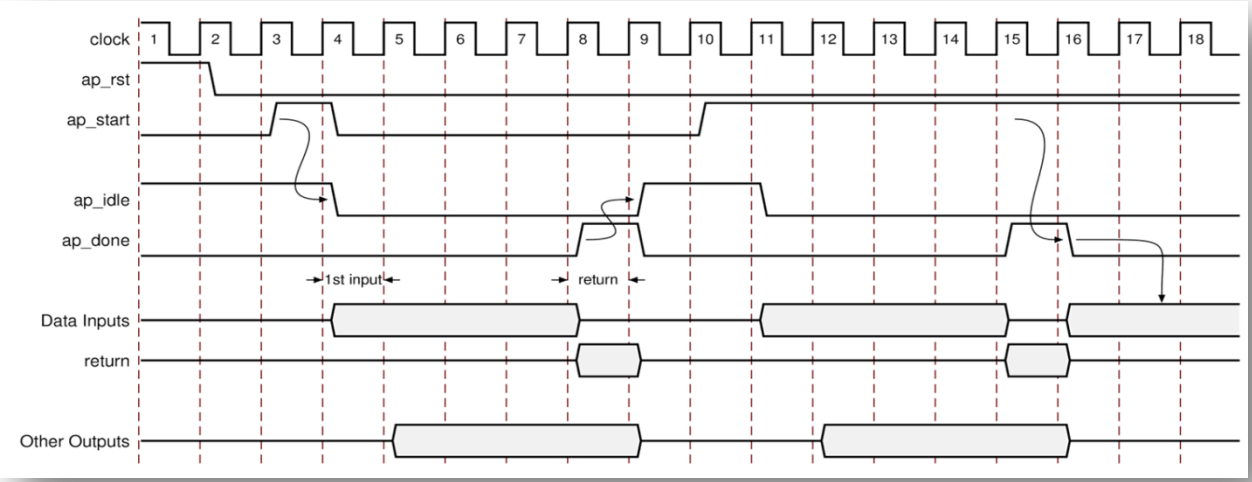

Figure 33 Behavior of ap_ctrl_hs interface

After reset:

- The block will wait for “ap_start” to go high before it begins operation.
- Output “ap_idle” goes low when “ap_start” is sampled high.
- Data can now be read on the input ports.
    - The first input data may be sampled on the first clock edge after “ap_idle” goes low.
- When the block completes all operations, any return value will be written to port ap_return
    - If there was no function return, there will be no ap_return port on the RTL block.
    - Other outputs may be written to at any time until the block completes and are independent of this IO protocol.
- Output “ap_done” goes high when the block completes operation.
    - If there is an ap_return port, the data on this port will be valid when “ap_done” is high.
    - The “ap_done” signal can therefore used to validate when the function return value (output on port ap_return) is valid.
- The idle signal goes high one cycle after “ap_done” and remains high until the next time “ap_start” is sampled high (indicating the block should once again begin operation).

If the “ap_start” signal is high when “ap_done” goes high:

- The “ap_idle” signal will remain low.
- The block will immediately start its next execution (or next transaction).
- The next input may be read on the next clock edge.

AutoESL supports pipelining, allowing the next transaction to begin before the current one ends. In this case, the block can accept new inputs before the first transaction completes and output port “ap_done” is asserted high.

If the function is pipelined, or if the top-level loop is pipelined with the `-rewind` option, an additional output port "ap_ready" is created to indicate when new inputs can be applied. Figure 34 shows the behavior of the block level interface when pipelining is used.

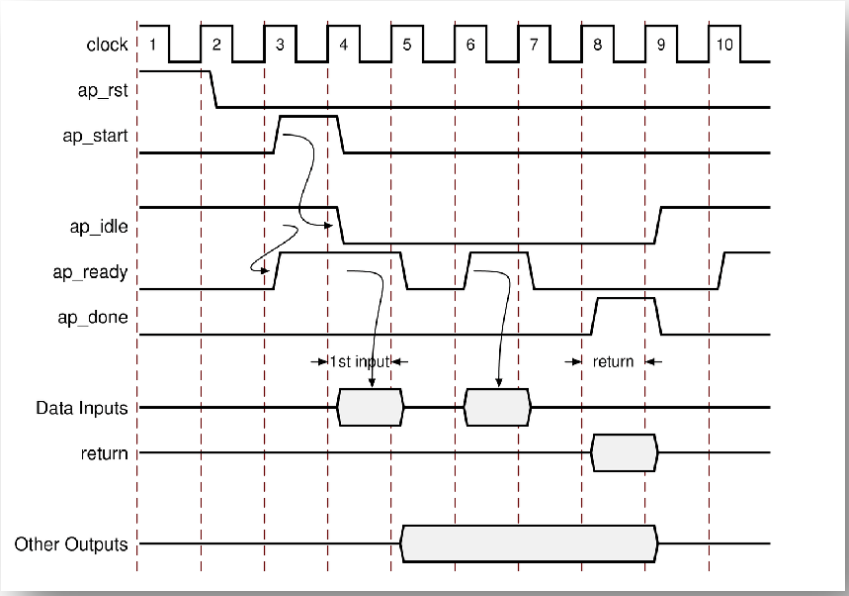

Figure 34 Behavior of ap_ctrl_hs interface with pipelining

After reset:

- If "ap_idle" is high, the "ap_ready" signal will go high.
- The block will wait for "ap_start" to go high before it begins operation.
- Output "ap_idle" goes low when "ap_start" is sampled high.
- Data will be read on the input ports when "ap_ready" is high.

Note: Since "ap_ready" can be high before "ap_idle" is low, both the ready and idle signals must be used by producer blocks for applying new data.

The remainder of the behavior is the same as that described for Figure 33.

Note: The only data which is guaranteed to be valid when "ap_done" is asserted is the data on the optional ap_return port (this port will only exist if a value is returned from the function using the `return` statement).

The other outputs in the design may be valid at this time, the end of the transaction when "ap_done" is asserted, but that is not guaranteed. If it is a requirement that an output port must have an associated valid signal, it should be specified with one of the port-level IO protocols discussed in the remainder of this section.

ap_none

The ap_none interface type is simplest interface and has no other signals associated with it. Neither the input nor output signals have any associated control ports indicating when data is read or written. The only ports in the RTL design are those specified in the source code.

An ap_none interface requires no additional hardware overhead but does require that producer blocks provide data to the input port at the correct time or hold it for the length of a transaction (until the design completes) and consumer blocks are required to read output ports at the correct time: as such, a design with interface mode ap_none specified on an output cannot be automatically verified using the autosim feature.

The ap_none interface cannot be used with array arguments, as shown in Table 18.

ap_stable

The ap_stable interface type, like type ap_none, does not add any interface control ports to the design. The ap_stable type informs AutoESL that the data applied to this port will remain stable during normal operation, but is not a constant value which could be optimized, and the port is not required to be registered.

The ap_stable type is typically used for ports which will provide configuration data - data which may change but which will remain stable during normal operation (configuration data is typically only changed during or before a reset).

The ap_stable type can only be applied to input ports. When applied to inout ports, only the input part of the port is considered to be stable.

ap_hs (ap_ack, ap_vld and ap_ovld)

An ap_hs interface provides both an acknowledge signal to say when data has been consumed and a valid signal to indicate when data has been read. This interface type is a superset of types ap_ack, ap_vld and ap_ovld.

- Interface type ap_ack only provides an acknowledge signal.
- Interface type ap_vld only provides a valid signal.
- Interface type ap_ovld only provides a valid signal and only applies to output ports or the output half of an inout pair.

Figure 35 shows how an ap_hs interface behaves for both an input and output port. In this example the input port in named “in” and the output port named “out”. Note how the control signals are automatically named, based on the original port name (For example, the valid port for input “in” is added and named “in_vld”).

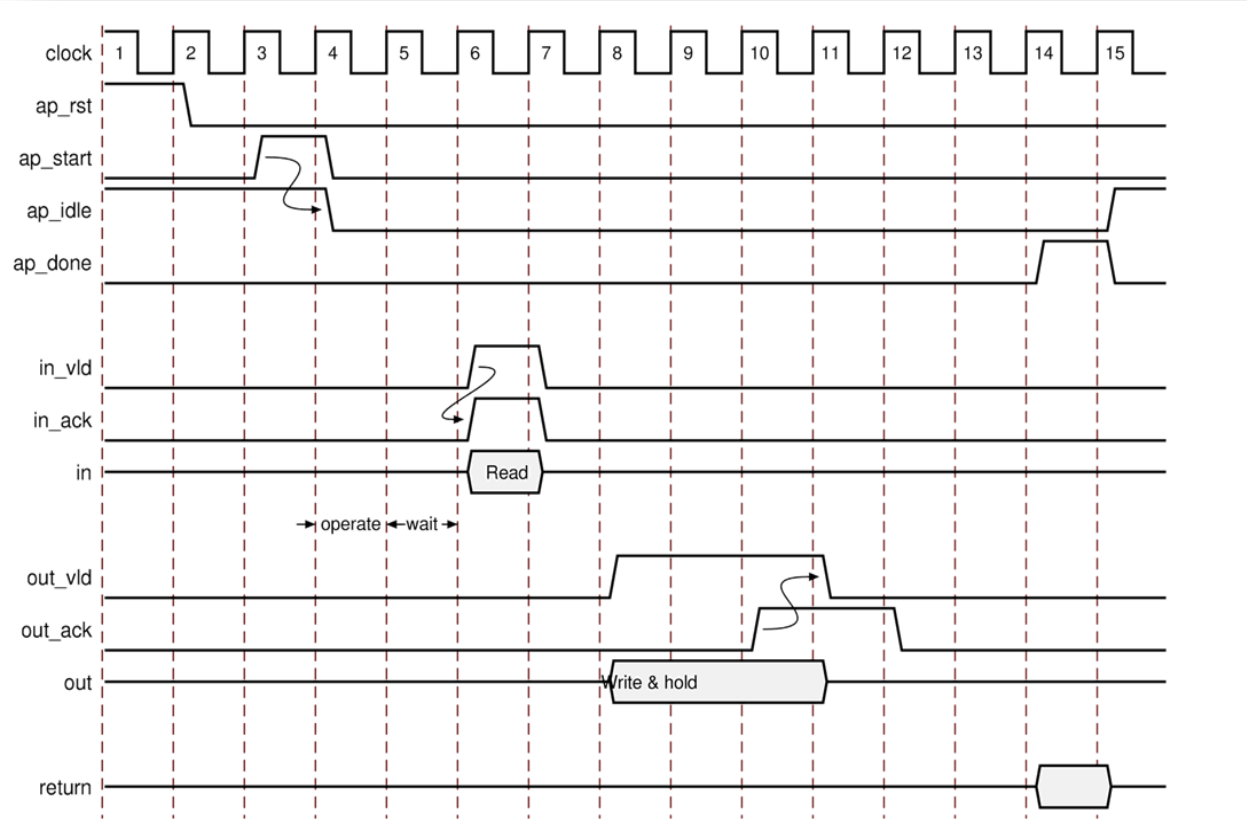

Figure 35 Behavior of ap_hs interface

For inputs:

- After reset and start have been applied, the block will begin normal operation.
- If the input port is to be read but the input valid is low, the design will stall and wait for the input valid to be asserted; indicating a new input value is present.
- As soon as the input valid is asserted high, an output acknowledge will be asserted high to indicate the data was read.

For outputs:

- After reset and start have been applied, the block will begin normal operation.
- When an output port is written to, its associated output valid signal is simultaneously asserted to indicate valid data is present on the port.
- If the associated input acknowledge is low, the design will stall and wait for the input acknowledge to be asserted.
- When the input acknowledge is asserted, the output valid is de-asserted on the next clock edge.

Designs which use ap_hs interfaces can be verified with autosim and provide the greatest flexibility in the development process, allowing both bottom-up and top-down design flows: all intra-block communication is safely performed by two-way handshakes, with no manual intervention or assumptions required for correct operation.

The ap_hs interface is a safe interface protocol but requires a two port overhead, with associated control logic.

With an ap_ack interface, only an acknowledge port is synthesized.

- For input arguments this results in an output acknowledge port which is active high in the cycle in which the input is read.
- For output arguments this results in an input acknowledge port.
    - After a write operation, the design will stall and wait until the input acknowledge has been asserted high, indicating the output has been read by a consumer block.
    - However, there is no associated output port to indicate when the data can be consumed.

Care should be taken when specifying output ports with ap_ack interface types. Designs which use ap_ack on an output port cannot be verified by autosim.

Specifying interface type ap_vld results in a single associated valid port in the RTL design.

- For output arguments this results in an output valid port, indicating when the data on the output port is valid.

  Note: For input arguments this valid port behaves in a different manner than the valid port implemented with ap_hs.

- If ap_vld is used on an input port (there is no associated output acknowledge signal), the input port will be read as soon as the valid is active: even if the design is not ready to read the port, the data port will be sampled and held internally until needed.
- The input data will be read each cycle the input valid is active.

An ap_ovld interface type is the same an ap_vld but can only be specified on output ports. This is a useful type for ensuring pointers which are both read from and written to, will only be implemented with an output valid port (and the input half will default to type ap_none).

ap_memory

Array arguments are typically implemented using the ap_memory interface. This type of port interface is used to communicate with memory elements (RAMs, ROMs) when the implementation requires random accesses to the memory address locations. Array arguments are the only arguments which support a random access memory interface.

If sequential access to the memory element is all that is required, the `ap_fifo` interface discussed next can be used to reduce the hardware overhead: no address generation is performed in an `ap_fifo` interface.

When using an `ap_memory` interface, the array targets should be specified using the `set_directive_resource` command as shown in section "Memory Resource Selection". If no target is specified for the arrays, AutoESL will automatically determine if a single or dual-port RAM interface will be used.

> Ensure array arguments are targeted to the correct memory type using `RESOURCE directive` before synthesis as re-synthesizing with new, correct, memories may result in a different schedule and RTL.

Figure 36 shows an example where an array named “d” has the resource specified as a single-port BRAM.

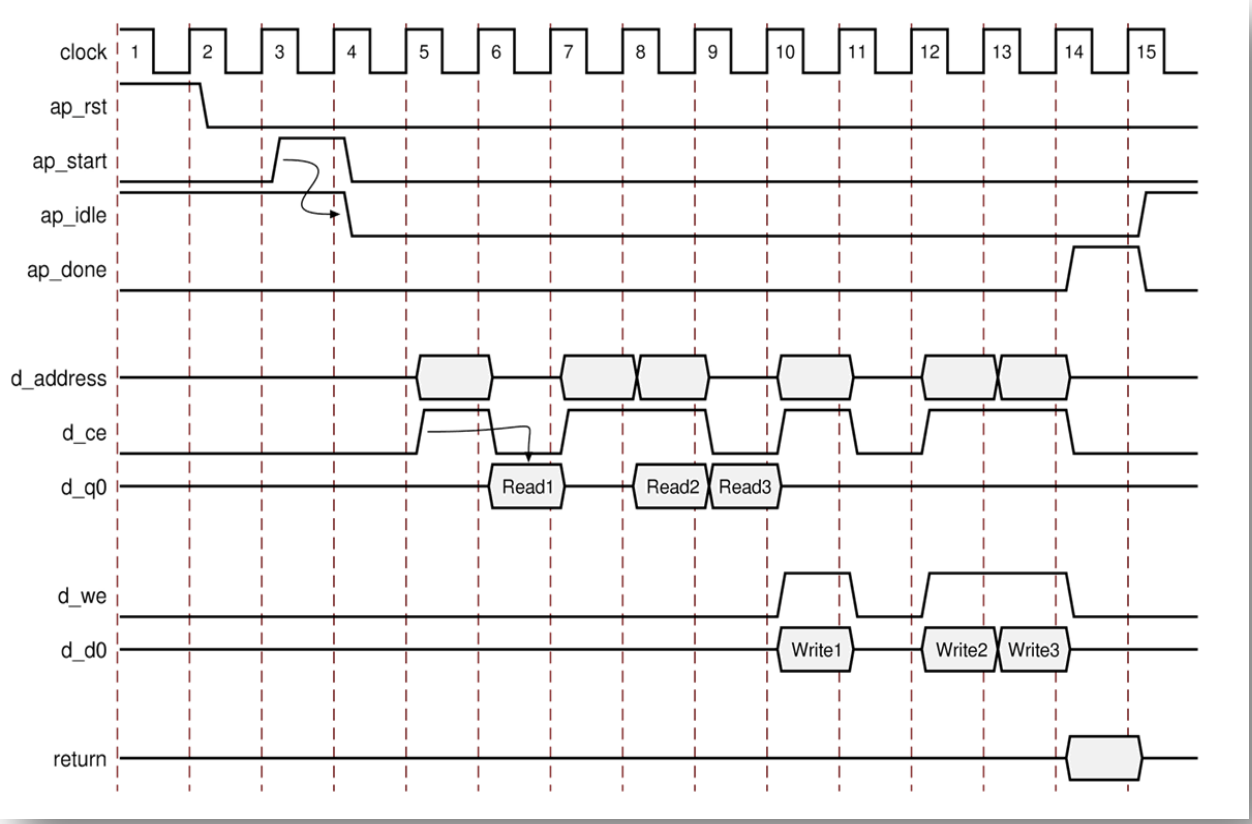

Figure 36 Behavior of ap_memory interface

After reset:

- After reset and start have been applied, the block will begin normal operation.

- Reads will be performed by applying an address on the output address ports while asserting the output signal “d_ce”.
    - For this BRAM target, the design expects the input data to be available in the next clock cycle.
- Write operations will be performed by asserting output ports “d_ce” and “d_we” while simultaneously applying the address and data.

A memory interface cannot be stalled by external signals, provides an indication of when output data is valid and can therefore be verified using `autosim`.

ap_fifo

If access to a memory element is required and the access is only ever performed in a sequential manner (no random access required) an `ap_fifo` interface is the most hardware efficient. The `ap_fifo` interface allows the port to be connected to a FIFO, supports full two-way empty-full communication and can be specified for array, pointer and pass-by-reference argument types.

Functions which can use an `ap_fifo` interface will often use pointers and may access the same variable multiple times. Refer to "Multi-Access Pointer Interfaces" to understand the importance of the `volatile` qualifier when this coding style is used.

> **Note**: An `ap_fifo` interface assumes that all reads and writes are sequential in nature.
>
> If AutoESL can determine this is not the case, it will issue an error and halt.
>
> If AutoESL cannot determine that the accesses are always sequential, it will issue a warning that it is unable to confirm this and proceed.

In the following example “in1” is a pointer which accesses the current address, then two addresses above the current address and finally one address below.

```
void foo(int* in1, ...) {
       int data1, data2, data3;
        ...
       data1= *in1;
       data2= *(in1+2);
       data3= *(in1-1);
       ...
}
```

If “in1” is specified as an `ap_fifo` interface, AutoESL will check the accesses and determine the accesses are not in sequential order, issue an error and halt. To read from non-sequential address locations use an `ap_memory` interface as this random accessed or use an `ap_bus` interface.

An `ap_fifo` interface cannot be specified on an argument which is both read from and written to (an inout port). Only input and output arguments can be specified with an `ap_fifo` interface. An interface with input argument “in” and output argument “out” specified as `ap_fifo` interfaces will behave as follows:

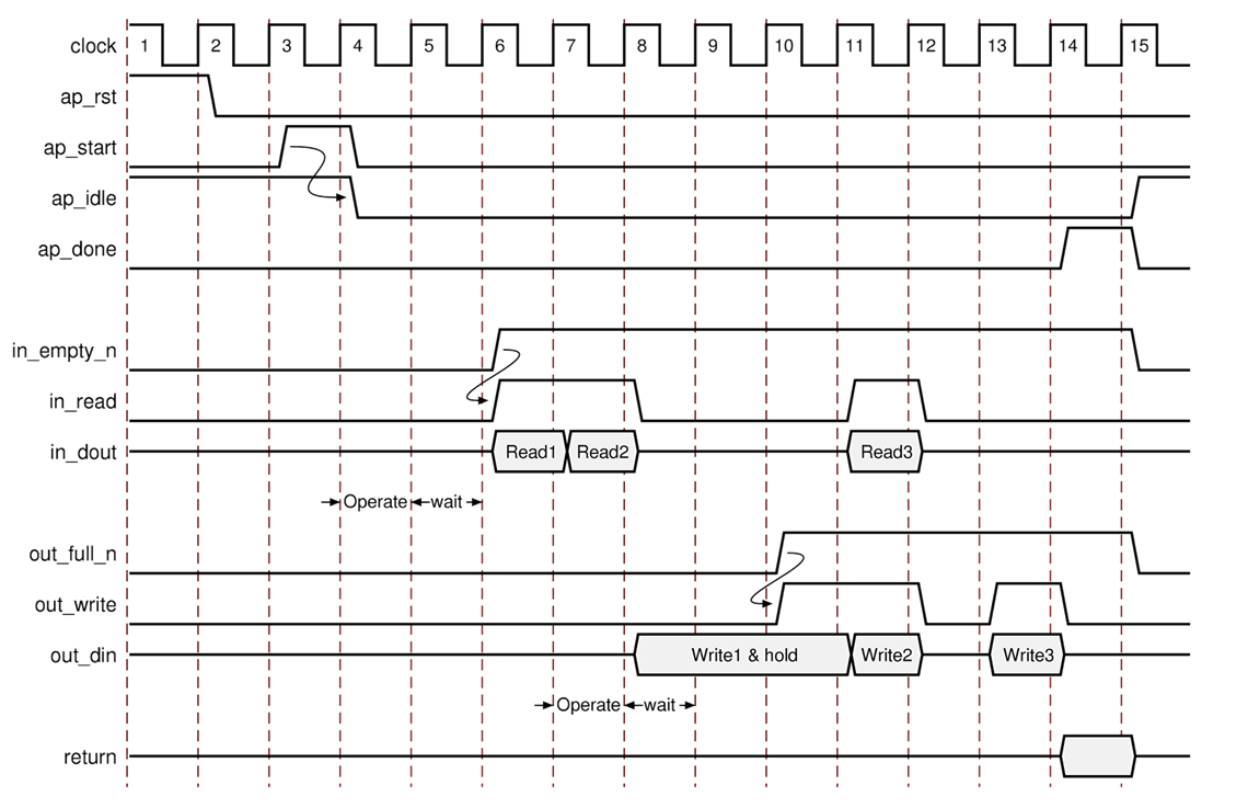

Figure 37 Behavior of ap_fifo interface

- After reset and start have been applied, the block will begin normal operation.

For reads:

- If the input port is to be read but the FIFO is empty, as indicated by input port "in_empty_n" low, the design will stall and wait for data to become available.
- As soon as the input port "in_empty_n" is asserted high to indicate data is available, an output acknowledge ("in_read") is asserted high to indicate the data was read in this cycle.
- If data is available in the FIFO when a port read is required, the output acknowledge will be asserted to acknowledge data was read in this cycle.

For outputs:

- If an output port is to written to but the FIFO is full, as indicated by "out_full_n" low, the data will be placed on the output port but the design will stall and wait for the space to become available in the FIFO.

- When space becomes available in the FIFO (input “out_full_n” goes high) the output acknowledge signal “out_write” is asserted to indicate the output data is valid.
- If there is space available in the FIFO when an output is written to, the output valid signal is asserted to indicate the data is valid in this cycle.

In an ap_fifo interface the output data port has an associated write signal: ports with an ap_fifo interface can be verified using autosim.

ap_bus

An ap_bus interface can be used to communicate with a bus bridge. The interface does not adhere to any specific bus standard but is generic enough to be used with a bus bridge which in-turn arbitrates with the system bus. The bus bridge must be able to cache all burst writes.

Functions which can employ an ap_bus interface will often use pointers and may access the same variable multiple times. Refer to "Multi-Access Pointer Interfaces" to understand the importance of the volatile qualifier when this coding style is used.

An ap_bus interface can be used in two specific ways.

- Standard Mode: The standard mode of operation is to perform individual read and write operations, specifying the address of each.
- Burst Mode: If the C function memcpy is used in the C source code, burst mode will be used for data transfers. In burst mode, the base address and the size of the transfer is indicated by the interface: the data samples are then quickly transferred in consecutive cycles.

Figure 38 and Figure 39 show the behavior for read and write operations in standard mode, when an ap_bus interface is applied to argument “d” in the following example:

```
void foo (int *d) {
  static int acc = 0;
  int i;

  for (i=0;i<4;i++) {
    acc += d[i+1];
    d[i] = acc;
  }
}
```

While Figure 40 and Figure 41 show the behaviors when the C memcpy function and burst mode is used, as in this example code:

```
void bus (int *d) {
  int buf1[4], buf2[4];
  int i;

       memcpy(buf1,d,4*sizeof(int));
```

```
      for (i=0;i<4;i++) {
    buf2[i] = buf1[3-i];
  }

      memcpy(d,buf2,4*sizeof(int));
}
```

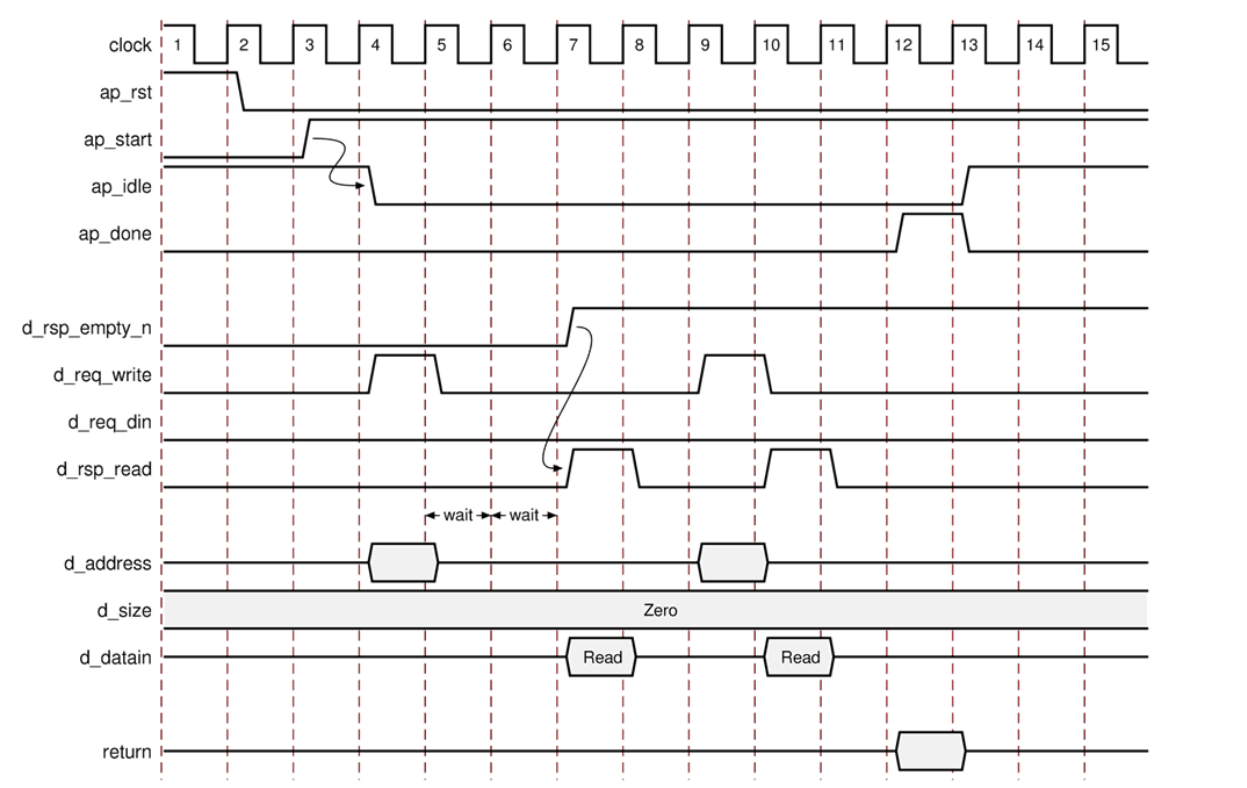

Figure 38 Behavior of ap_bus interface: Standard read

After reset:

- After reset and start have been applied, the block will begin normal operation.
- If a read is to be performed, but there is no data in the bus bridge FIFO ("d_rsp_empty_n" is low):
  - Output port "d_req_write" is asserted with port "d_req_din" de-asserted, to indicate a read operation.
  - The address is output.
  - The design will stall and wait for data to become available.
- When data becomes available for reading the output signal "d_rsp_read" is immediately asserted and data is read at the next clock edge.
- If a read is to be performed, and data is available in the bus bridge FIFO ("d_rsp_empty_n" is high):

- Output port “d_req_write” is asserted and port “d_req_din” is de-asserted, to indicate a read operation.
- The address is output.
- “d_rsp_read” is asserted in the next clock cycle and data is read at the next clock edge.

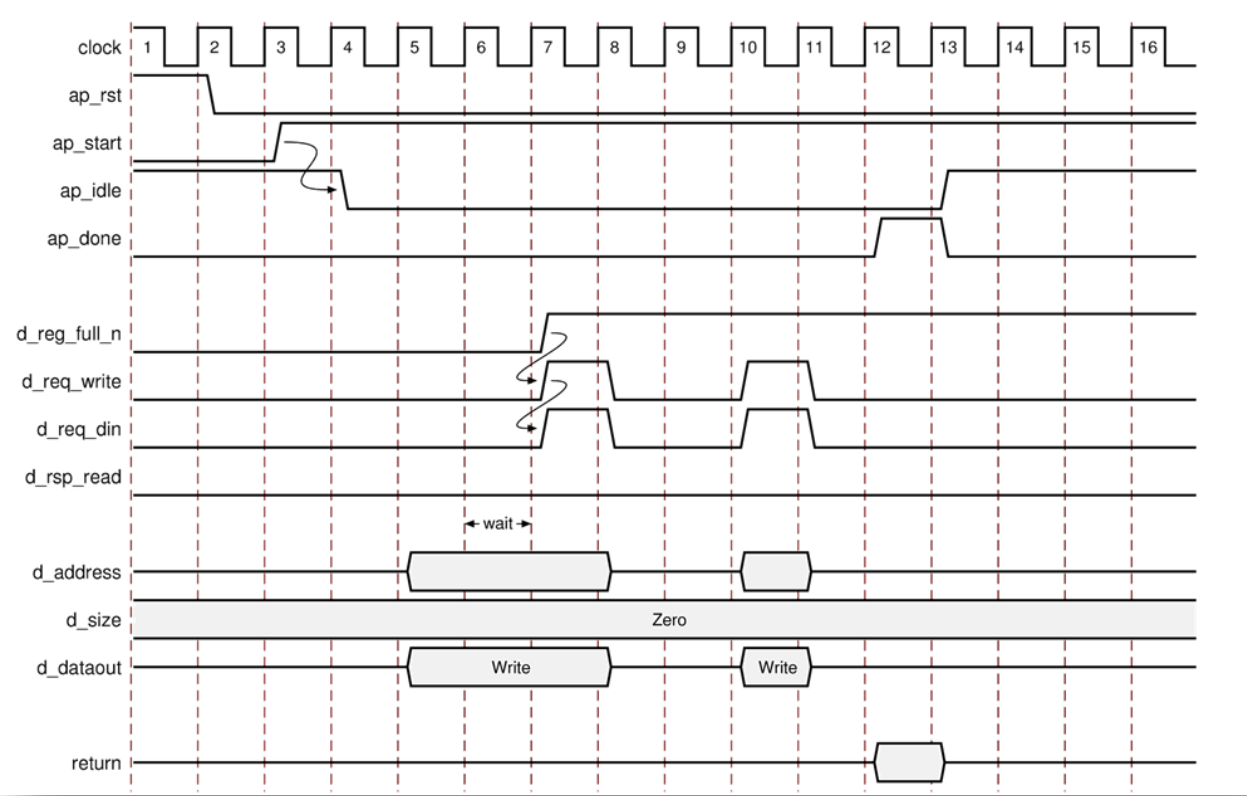

Figure 39 Behavior of ap_bus interface: Standard write

After reset:

- After reset and start have been applied, the block will begin normal operation.
- If a write is to be performed, but there is no space in the bus bridge FIFO (“d_req_full_n” is low):
  - The address and data are output.
  - The design will stall and wait for space to become available.
- When space becomes available for writing:
  - Output ports “d_req_write” and “d_req_din” are asserted, to indicate a write operation.
  - The output signal “d_req_din” is immediately asserted to indicate the data is valid at the next clock edge.
- If a write is to be performed, and space is available in the bus bridge FIFO (“d_rsp_full_n” is high):

- Output ports “d_req_write” and “d_req_din” are asserted, to indicate a write operation.
- The address and data are output.
- “d_req_din” is asserted to indicate the data is valid at the next clock edge.

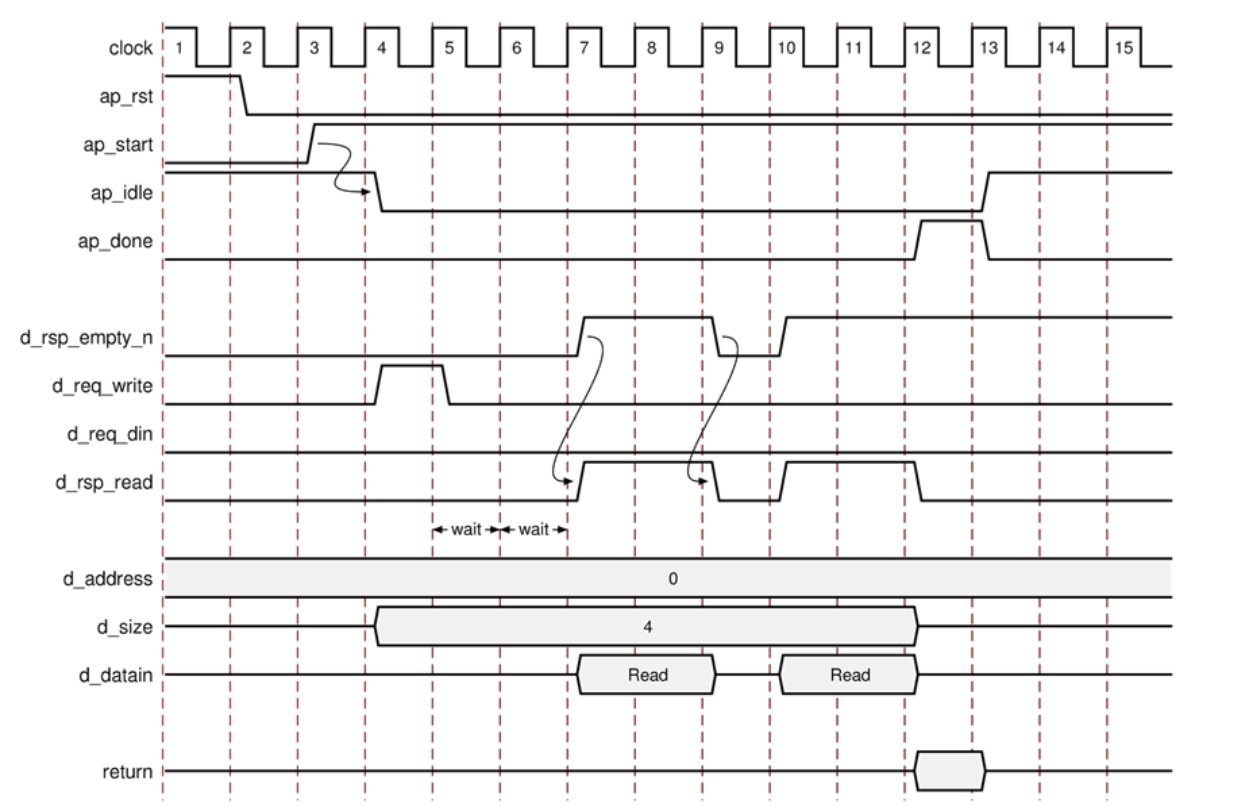

Figure 40 Behavior of ap_bus interface: Burst read

After reset:

- After reset and start have been applied, the block will begin normal operation.
- If a read is to be performed, but there is no data in the bus bridge FIFO (“d_rsp_empty_n” is low):
  - Output port “d_req_write” is asserted with port “d_req_din” de-asserted, to indicate a read operation.
  - The base address for the transfer and the size are output.
  - The design will stall and wait for data to become available.
- When data becomes available for reading the output signal “d_rsp_read” is immediately asserted and data is read at the next N clock edges, where N is the value on output port size.

- If the bus bridge FIFO runs empty of values, the data transfers stop immediately and wait until data is available before continuing where it left off.
- If a read is to be performed, and data is available in the bus bridge FIFO
    - Transfer begin as above and the design will stall and wait if the FIFO empties

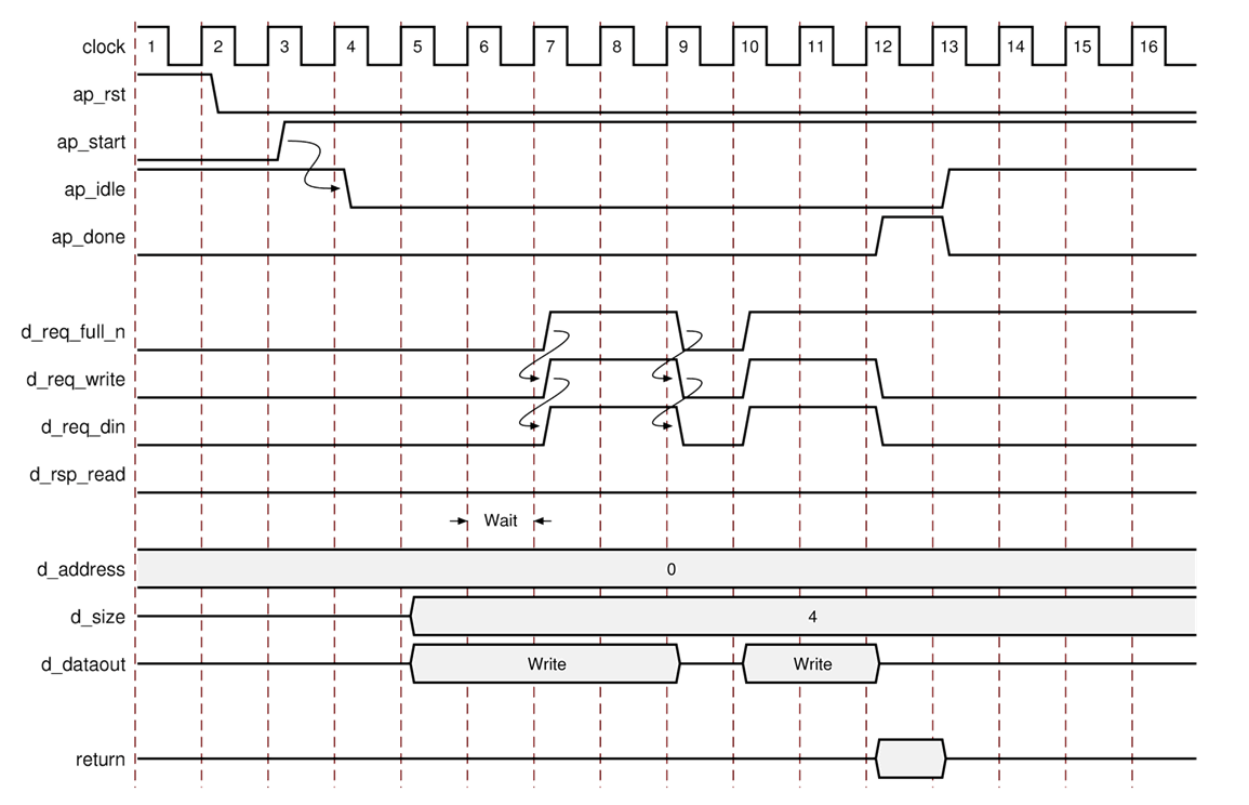

Figure 41 Behavior of ap_bus interface: Burst write

After reset:

- After reset and start have been applied, the block will begin normal operation.
- If a write is to be performed, but there is no space in the bus bridge FIFO ("d_req_full_n is low):
    - The base address, transfer size and data are output.
    - The design will stall and wait for space to become available.
- When space becomes available for writing:
    - Output ports "d_req_write" and "d_req_din" are asserted, to indicate a write operation.
    - The output signal "d_req_din" is immediately asserted to indicate the data is valid at the next clock edge.

    - Output signal “d_req_din” will be immediately de-asserted if the FIFO becomes full and re-asserted when space is available.
    - The transfer will stop after N data values have been transferred, where N is the value on the size output port.
- If a write is to be performed, and space is available in the bus bridge FIFO (“d_rsp_full_n” is high):
    - Transfer begins as above and the design will stall and wait when the FIFO is full.

The `ap_bus` interface can be verified by `autosim`.

Controlling Interface Synthesis

Interface synthesis is controlled by the INTERFACE directive or by using a configuration setting.

Configuration settings can be used to specify the default operation for creating RTL ports and interfaces. The `INTERFACE` directive is used to specify the explicit interface type of a particular port and overrides any default or global configuration.

Configuring Default Port Interface Types

Configuration settings can be used to:

- Add a global clock enable to the RTL design.
- Create RTL ports for any global variables.
- Set a default interface type for all ports of the specified type.

The configuration can be set using the GUI option `Solution` → `Solution Settings…` → `General` → `config_interface`, as shown in Figure 42.

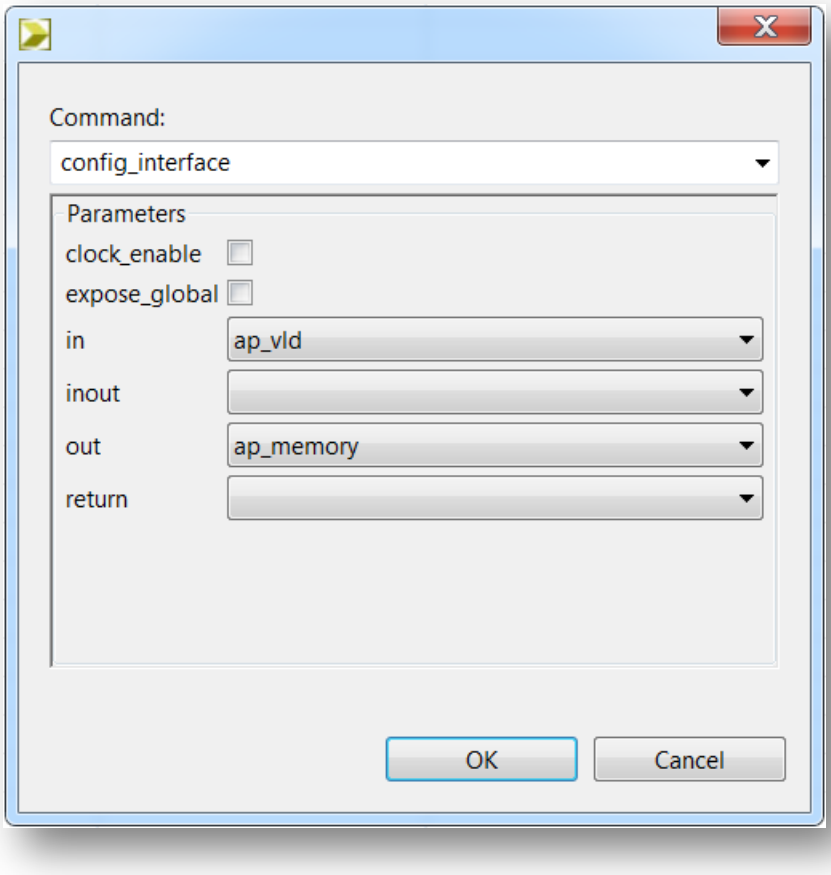

Figure 42 Setting a Configuration

The interface configuration setting `clock_enable` adds global clock enable (CE) to be added to the RTL design when selected. This adds port `ap_ce` to the top-level ports: when port `ap_ce` is active low, the clock is inhibited to all registers in the design.

Any C function can use global variables: those variables defined outside the scope of any function. By default, global variables do not result in the creation of RTL ports: AutoESL will assume the global variable is inside the final design. The configuration setting `expose_global` tells AutoESL the global variable will actually be implemented outside the scope of the function/design and to create an RTL port.

Finally, the default interface type for all ports was shown in Table 18. The interface configuration allows these default interfaces to be specified by the user. The following example, shown in Figure 42 and shown below using the associated Tcl command, sets the interface type on all input ports to type `ap_vld` and the default type on all output ports to type `ap_memory`.

```
config_interface -mode in ap_vld
config_interface -mode out ap_memory
```

Note: If a port is specified with an unsupported interface type, the configuration will be ignored and the interface will be set to the default type for that port: the unsupported port types and the default port types are shown in .

For example, if an array port is specified to be implemented with an unsupported IO protocol type such as ap_ack, the directive or configuration will be ignored.

Specifying Port Interface Types

The type of interface can be set on specific ports. Simply select the port in the GUI directives tab and right-click on the mouse to open the directives menu as shown in Figure 43.

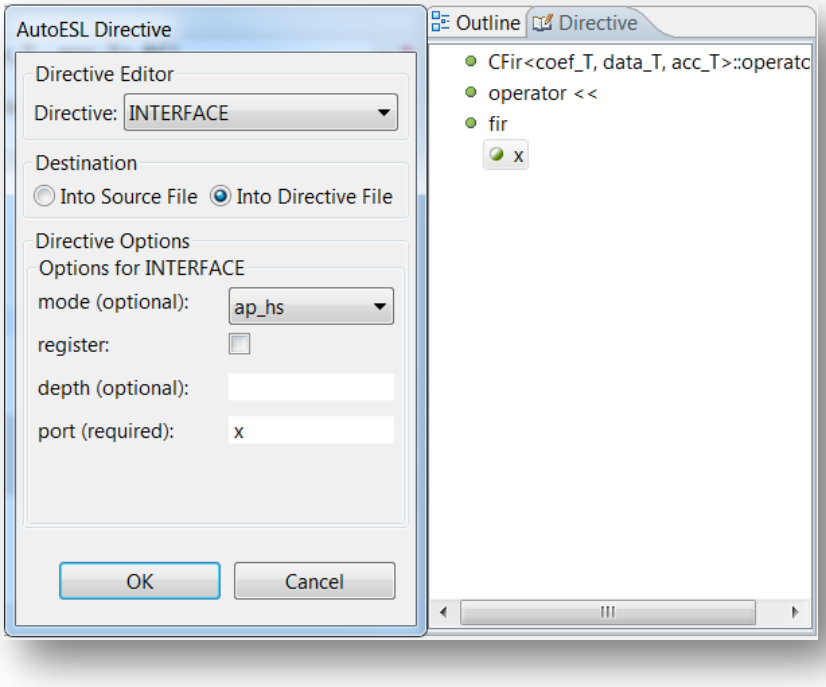

Figure 43 Specifying Port Interfaces

The interface can also be specified in Tcl command file. In this example, the same operation as Figure 43 is specified and port x in function foo is set to type ap_hs.

```
set_directive_interface -mode ap_hs fir x
```

If an unsupported port interface type is selected, a message will be issued and the default port type as shown in Table 18 will be implemented.

Specifying Port Options

The INTERFACE directive has two options. The first specifies if the port should be registered or not (the default) and second specifies the number of values read from or written to the port.

By default AutoESL does not register ports. It will read input ports when the data is required saving the area overhead of a register. If the register option is selected AutoESL will register all pass-by-value reads in the first cycle of operation. For pointer read it will register the read but it will perform the read one cycle before the data is required. For output ports, the register option will guarantee the output is registered. For memory, FIFO and bus interfaces the register option has no effect.

For cases where a pointer is read from or written to multiple times within a single transaction, the depth option should be used to specify the maximum number of values read or written. Failure to specify this correctly may results in a verification failure when the `autosim` feature is used to verify the RTL.

SystemC Interface Synthesis

In general, interface Synthesis is not supported for SystemC designs. The IO ports for SystemC designs should be fully described in the SC_MODULE interface and in behavior of the ports fully described in the source code.

An exception to this is memory ports. Given a design with standard SystemC array ports,

```
SC_MODULE(my_design) {
  //”RAM” Port
       sc_uint<20>  my_array[256];
       …
```

The port “my_array” would be synthesized into internal RAMs.

Including the AutoESL header file “ap_mem_if.h” allows the port to be specified as an `ap_mem_port<data_width, address_bits>` port. An `ap_mem_port` will be synthesized into a standard RAM interface port with the specified data and address bus-widths.

The code example above, can be transformed into the following,

```
#include "ap_mem_if.h"

SC_MODULE(my_design) {
  //”RAM” Port
       ap_mem_port<sc_uint<20>,sc_uint<8>, 256> my_array;
       …
```

This will ensure interface port “my_array” is implemented as a RAM interface.

When an `ap_mem_port` is added to a SystemC design, an associated `ap_mem_chn` must be added to the SystemC test bench to drive the `ap_mem_port`.

In the test bench, an `ap_mem_chn` is defined and attached to the instance as shown:

```
#include "ap_mem_if.h"
…
ap_mem_chn<int,int, 68> bus_mem;
```

```
…

// Instantiate the top-level module
my_design U_dut (“U_dut”)
U_dut.my_array.bind(bus_mem);
…
```

The header file “ap_mem_if.h” is located at “$AUTOESL_ROOT\include \ap_sysc\ap_mem_if.h” and must be included during SystemC simulation.

Manual Interface Specification

AutoESL has the ability to identify blocks of code which defines a specific IO protocol. This allows an IO protocol to be specified without using interface synthesis or SystemC (the protocol directive explained below can also be used with SystemC designs to provide greater IO control).

In following example code,

- input "response[0]" is read
- output "request" is written
- input "response[1]" is read.
- AND it is required that the final design perform the IO accesses in this order

```
void test (
  int   *z1,
  int   a,
  int   b,
  int   *mode,
  volatile int  *request,
  volatile int  response[2],
  int   *z2
  ) {

  int   read1, read2;
  int   opcode;
  int   i;


  P1: {
    read1       = response[0];
    opcode      = 5;
    *request    = opcode;
    read2       = response[1];
  }
  C1: {
          *z1   = a + b;
          *z2   = read1 + read2;
  }
}
```

When AutoESL implements this code there is no reason the "request" write should be between the two reads on "response". The code is written with this IO behavior but there are no dependencies in the code which enforce it. AutoESL may schedule the IO accesses in the same manner or choose some other access pattern.

In this case the use of a protocol block can be used to enforce a specific IO protocol behavior. Since the accesses occur in the scope defined by block "P1", a protocol can be applied to this block as follows:

- Include "ap_utils.h" header file which defines `ap_wait()`.
- Place an `ap_wait()` statement after the write to "request", but before the read on "response[1]".
    - The `ap_wait()` statement will not cause the simulation to behave any differently, but it will instruct AutoESL to insert a clock here during synthesis.
- Specify that block P1 is a protocol region.
    - This will instruct AutoESL that the code within this region is to be scheduled as is: no re-ordering of the IO or `ap_wait()` statements.

Applying the directive as shown,

```
set_directive_protocol test P1 -mode floating
```

to the modified code

```
#include "ap_utils.h"                    // Added include file

void test (
  int   *z1,
  int   a,
  int   b,
  int   *mode,
  volatile int  *request,
  volatile int  response[2],
  int   *z2
  ) {

  int   read1, read2;
  int   opcode;
  int   i;


  P1: {
    read1       = response[0];
    opcode      = 5;
    ap_wait();                           // Added ap_wait statement
    *request    = opcode;
    read2       = response[1];
  }
  C1: {
          *z1   = a + b;
          *z2   = read1 + read2;
```

```
  }
}
```

will result in exact IO behavior specified in the code:

- input "response[0]" is read
- output "request" is written
- input "response[1]" is read.

The `-mode floating` option allows other code to execute in parallel with this block, if allowed by data dependencies. The `fixed` mode would prevent this.

Design Optimization

This chapter outlines the various optimizations and techniques which can be employed to direct AutoESL to produce a micro-architecture which satisfies the desired performance, area and power goals.

The layout of the topics chapter is designed to help minimize the effort required in finding the correct optimizations to apply and maximize productivity. It is recommended to approach design optimization by reviewing the topics in the following order:

- Checklist & Guidelines
- Clocks, Timing & RTL output
- Arbitrary Precision Data Types
- Performing Optimizations

The "Checklist & Guidelines" section provides strategies for applying the optimizations discussed in this chapter. Optimization strategies are provided for a number of standard design goals such as area, throughput, latency and power. However, even if your overall goals are different from one of these typical design goals, a review of this section will provide a solid foundation for applying AutoESL's features and capabilities to meet your goals.

In high-level synthesis, like logic synthesis, subtle changes to the input source code or the constraints can result in a somewhat different output design, only more so. To prevent the need to "start over again" review the sections on "Clocks, Timing & RTL output" and "Arbitrary Precision Data Types" to ensure the basic design parameters and design description are both correct and as ideal as possible before spending time and effort time applying more complex optimizations.

When the time comes to apply more complex and powerful optimization techniques a top-down or bottom-up approach can be used.

- If the design is far from meeting its latency or throughput goals, work from the top-down. Applying optimizations at higher levels of abstraction and operation is more likely to result in large improvements: start reviewing the design at the function level and work down towards the logic level. This is also the order presented in this User Guide.
- If the design is almost achieving its goals, focus on trying to reduce a few clock cycles or resources. In this case, it may be more productive to start at the level of logic structures, reviewing if better component selection would allow more operations in a cycle or if array accesses are causing bottlenecks and work up toward the function level.

Checklist & Guidelines

This section outlines some basic strategies for quickly reaching your optimization goals.

Design Basics

Review the basic designs parameters discussed in the section “Clocks, Timing & RTL output” to ensure the RTL is being created with the

- ✓ Correct clock and clock uncertainty.
- ✓ Reset style.
- ✓ State encodings.

Interface Synthesis

It is important to pay attention to both algorithm and interface synthesis (the interface has to provide the data, and could very well be the bottleneck to achieve required throughput rates).

Before proceeding to perform design optimizations, ensure the correct interfaces are being used:

- ✓ Confirm the current interfaces on the design are compatible with whatever design they will communicate with.
- ✓ Review the "Interface Management" chapter to select and define the correct interface.
- ✓ Confirm that the chosen interfaces work with the post-synthesis verification methodology, as changing the interfaces later may result in a different schedule. Refer to the “Verification” chapter for more details on the verification flow.

Finally, it should be appreciated that to meet performance requirements it may be necessary to change how the interface is implemented or the type of interface that is used. For example, an array interface may benefit from:

- Being changed from a single to dual-port RAM implementation
- Being changed from a RAM to streaming FIFO implementation
- Being changed from an array to a pointer which can support a DMA-like bus interface.

These types of decisions and changes are no different from those made when optimizing by hand: sometimes the biggest gain or only way forward is to consider a change at the level above the design and hence change an interface protocol. AutoESL will quickly allow you to determine if such changes will result in a beneficial architecture.

Data Types and Bit-Widths

AutoESL will propagate data types based on arithmetic properties; however it is safer to be explicit where possible.

The bit-widths of the variables in the C function directly impact the size of the storage elements and operators used in the RTL implementation. If a variables only requires 8-bits but is specified as an integer type (32-bit) it may result in larger and slower 32-bit operators being used, reducing the number of operations which can be performed in a clock cycle and potentially increasing latency and throughput.

- ✓ Use the appropriate precision for the data types. Refer to section “Arbitrary Precision s” in this chapter.
- ✓ Confirm the size of any arrays which are to be implemented as RAMs or registers. The area impact of any over-sized elements in the array is magnified because there are multiple elements in each array.
- ✓ Pay special attention to multiplications, divisions, modulus or other complex arithmetic operations. If these variables are larger than they need to be, they will negatively impact both area and performance.
- ✓ If using ANSI C, use `autocc` after modifying any bit-widths to confirm existing simulations give the same results. Refer to section “Arbitrary Precision ” below for an explanation of why `autocc` is required.

Minimum Area Designs

When trying to minimize the area in a design concentrate on re-using functions and loops. Functions and loops will iterate over the same hardware resources each time they execute: this maximizes sharing at a level above the operator.

- ✓ Before beginning work on minimizing the area, ensure the design already meets, or is close to meeting, its performance metrics.
- ✓ If a function is called multiple times, it will share the same hardware. This is a great way to save area through coding style. Check to see if function inlining will allow more functions to be shared (Refer to the section on “Function Re-use, Inlining”).
- ✓ Similarly, a loop will iterate over the same hardware. This implements the loop functionality with a small area but a large latency. Ensure for-loops are not unrolled: the default is to keep them rolled. (Refer to the section "Loop Optimizations").
- ✓ If the design is pipelined, see if a different initiation interval still satisfies the throughput requirements but allows greater re-use of components. (Refer to the section on “Function Optimizations” and "Loop Optimizations").
- ✓ Check if arrays are being optimally implemented on the existing RAMs or if array partitioning or reshaping could make more optimal use of the available RAM resources: allowing more parallel accesses. (Refer to "Array Optimizations").

Maximum Throughput Designs

In a maximum throughput design, the challenge is to process as much data as possible in as few cycles as possible.

- ✓ Starting at the function level, and especially in C and C++ designs (SystemC can support concurrency natively) examine the section on “Function Dataflow Pipelining”.
- ✓ Review the "Loop Pipelining" section later in this chapter to determine if pipelining can be applied on loops.

If all pipelining is complete or cannot be considered, the next step is to look at minimizing the latency through each function and loop: if it requires less cycles to complete the operations, the next input can be read sooner. The techniques for

minimizing the latency are given in the next section but the following are areas of importance when dealing with design throughput.

- ✓ Review the summary section of the reports. Examine the report on each function in the hierarchy for critical loops, those having the greatest impact, and use some of the techniques in the “Minimum Latency Designs” section below to reduce the latency of critical loops.
  - o A loop which takes 25 clocks but is executed twice has a 50 cycle impact.
  - o A loop which requires only 4 clocks but is executed 256 times has a 1024 cycle impact and should be considered more “critical”.
- ✓ Finally, if the limitation is at the design ports, consider changing the type of interface. This will require confirming with the other designs in the system that such changes can be supported.

Minimum Latency Designs

To reduce the latency through a design:

- ✓ Starting at the function and loop levels, and especially in C and C++ designs (SystemC can support concurrency natively) examine the sections on “Function Dataflow Pipelining” and "Loop Dataflow Pipelining" to improve concurrency.
- ✓ If there are any for-loops, check to see if unrolling or partially unrolling them, as described in section "Unrolling Loops", will reduce the latency. Unrolling allows more operations to be done in parallel using less cycles (but more resources and larger area).
- ✓ If there are multiple loops, remember that it costs 1 clock cycle to move from one loop-body to another. Review the sections on “Merging Loops” and “Flattening Nested Loops” to reduce loop overheads.
- ✓ Be careful with arrays. They typically map onto memories, which have a limited access capabilities (read ports and write ports). This can result in dependencies in the hardware that can increase the latency. For example, a dual-port RAM, or reconfigured RAM as discussed in “Array Optimizations”, may allow more reads and writes in the same clock cycle.

Minimum Power Designs

AutoESL automatically performs a number of power optimizations. Examples of these include operator gating, which is performed during scheduling and the addition of pipeline enables added to the beginning of pipelines. All power optimizations are applied automatically, if they do not negatively impact performance constraints.

AutoESL does not allow performance to be sacrificed to improve power however there are a number of methodologies which can be followed to further reduce power.

- ✓ Review the section above on “Minimum Area Designs” as reducing area can have a great impact on power.

- ✓ Do not forget to use the architectural exploration capabilities of AutoESL. AutoESL allows you change the clock period and quickly re-generate a new micro-architecture.

Clocks, Timing & RTL output

For C and C++ designs only a single clock is supported. The same clock is applied to all functions in the design. For SystemC designs, each SC_MODULE may be specified with a different clock.

To specify multiple clocks in a SystemC design, use the `-name` option of the `create_clock` command to create named clocks and use the `set_directive_clock` command or pragma to specify which function contains the SC_MODULE to be synthesized with the specified clock. Each SC_MODULE can only be synthesized using a single clock: clocks may be distributed through functions, such as when multiple clocks are connected from the top-level ports to individual blocks, but each SC_MODULE can only be sensitive to a single clock).

The clock period, in ns, is set in the solutions setting of the GUI. In addition AutoESL uses the concept of a clock uncertainty to provide a user defined timing margin.

AutoESL estimates the timing of operations in the design but it cannot know the final component placement and net routing: as such, it cannot know the exact delays. The clock uncertainty is a value which is subtracted from the clock period to give the usable clock period as shown in Figure 44. AutoESL will use the usable clock period to schedule the operations in the design.

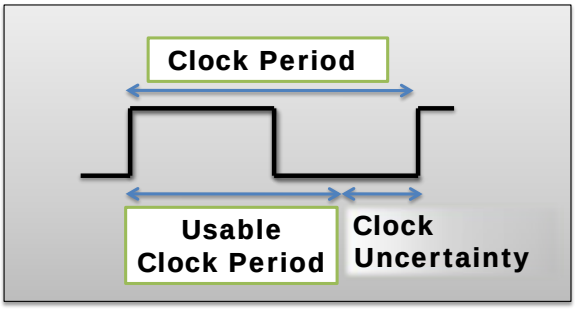

Figure 44 Clock Period and Margin

This provides some user specified slack to ensure downstream processes, such as logic synthesis and place & route, have enough timing margin to complete their operations and limits the design iterations associated with timing closer.

By default, the clock uncertainty is 12.5% of the cycle time; however the clock uncertainty can be defined by the user, in the solutions settings menu beside the clock period. The advantage of the automatic setting is that it is updated if the clock period is changed (else the user must manually update the clock uncertainty when the clock period is changed).

Timing

The timing information used for the RTL operators and registers is defined by the library. The libraries are all pre-characterized and stored within AutoESL.

AutoESL will always aim to meet latency, throughput (initiation interval) and the timing constraints. However, even when AutoESL cannot meet constraints, it will always output an RTL design.

- If AutoESL cannot meet a throughput constraint due to a data dependency (for example, if a throughput of one is required but it requires two cycles to read a value from memory) it will automatically increase the throughput until a design can be realized.
- If the timing constraints inferred by the clock period cannot be met AutoESL will issue message SCHED-644, as shown below, and output a design with the best achievable performance.

```
@W [SCHED-644] Max operation delay (<operation_name> 2.39ns) exceeds the
effective cycle time
```

Even if AutoESL cannot meet timing requirements for a particular path, or paths, it will still endeavor to achieve timing on all other paths. This behavior allows the user to evaluate if higher optimization levels or special handling of those failing paths by downstream processes can pull-in and ultimately satisfy the timing.

It is important to review the constraint report after synthesis to determine if all constraints have been met: ***the fact that AutoESL produces an output design does not guarantee the design meets all performance constraints***.

Review the "Performance Estimates" section of the design report.

A design report is generated automatically for each function in the hierarchy when synthesis completes and can be viewed in the solution reports folder and is stored in the solution directory in file ./syn/report/<function name>.rpt.

The worse case timing for the entire design is reported as the worst case in each function report. (There is no need to review every report in the hierarchy). If the timing violations are too severe to be further optimized and corrected by downstream processes, review the techniques in the remainder of this chapter before considering a faster target technology.

RTL Output

Various characteristics of the RTL output by AutoESL can be controlled using the `config_rtl` command. These include

- The ability to specify the type of FSM encoding used in the RTL state machines.
- The ability to add an arbitrary comment string, such as a copyright notice, to all RTL files using the –header option.

- Using the –prefix option to add a unique prefix to all output files, allowing RTL files generated from the same top-level function (and which would have the same name by default) to be easily combined in the same directory.

The RTL configuration can be defined via the Solution menu, using Solution ➔ Solution Settings ➔ General tab ➔ config_rtl as shown in Figure 45.

using the menu

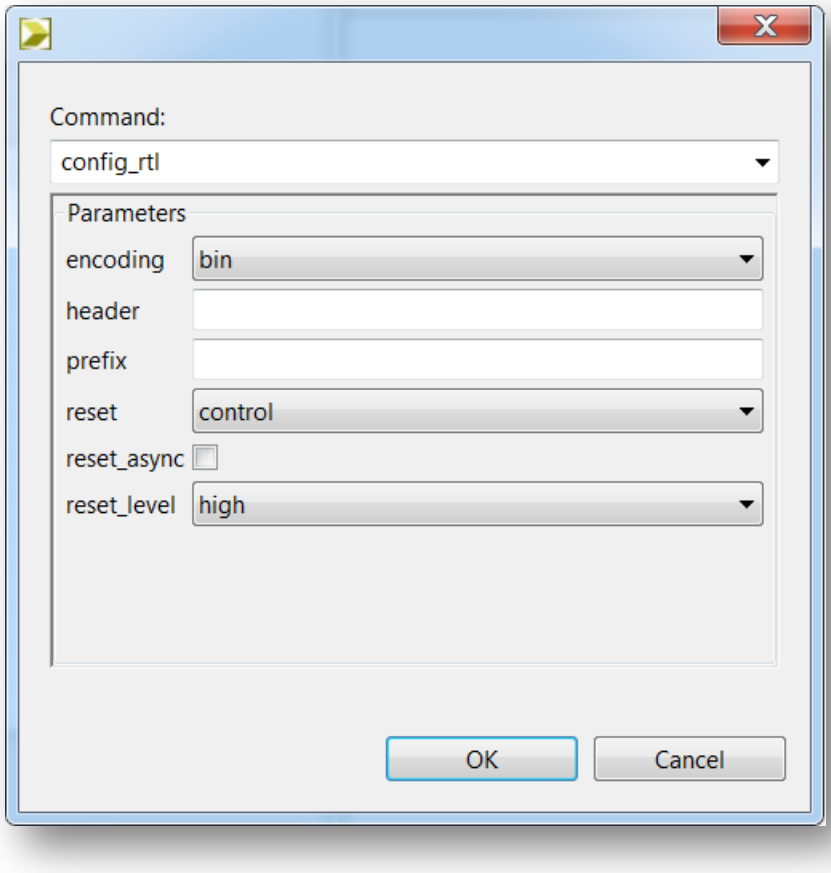

Figure 45 RTL Configurations

Typically the most important aspect of RTL configuration is selecting the reset behavior.

Selecting RTL Reset Behavior

When discussing reset behavior it is important to understand the difference between initialization and reset.

In C, variables defined with the static qualifier and those defined in the global scope, are by default initialized to zero. Optionally, these variables may be assigned a specific initial value. For these type of variables, the initial value in the C code is assigned at compile time (at time zero) and never again. In both cases, the same

initial value will be implemented in the RTL after synthesis: this ensures the FPGA bit-stream will initialize the device with the same initial value(s).

Any variable not defined with the static qualifier or defined in the global scope, but given an initial value, is assigned its initial value each time the function executes. As such, the initialization of such variables will occur each time the RTL design starts and will be part of normal operation. An example of such an initialization is the accumulator in an FIR filter. When a new data samples arrives, the accumulator is set to zero, values are then calculated and accumulated but the result of the accumulation is not carried forward to the next transaction. When a new sample arrives the accumulator is once again set to zero.

In addition to replicating the initial value of static and global variables in the RTL, a reset can optionally be applied to the RTL (the current default is to add a reset).

> **Note:** Reset is separate and in addition to initialization.
>
> Some configurations of the reset do not re-initialize variables to their initial power-up state when the reset signal is applied. In these cases, the initial value is only applied at power-up time.
>
> The current default operation is not to re-initialize static and global variables when a reset occurs.

The behavior of the RTL reset is controlled using the RTL configuration settings shown in Figure 45.

This includes being able to set the polarity of the reset and whether the reset is synchronous or asynchronous (Xilinx technologies favor the use of a synchronous reset: the default) but more importantly it determines, via the `reset` option, which registers are reset when the reset signal is applied.

The `reset` option has four settings:

- **none**: no reset is added to the design.
- **control**: reset control registers, such as those used in state machines and to generate IO protocol signals.
- **state**: reset control registers and registers/memories derived from static/global variables in the C code. Any static/global variable initialized in the C code is reset to its initialized value.
- **all**: reset all registers and memories in the design. Any static/global variable initialized in the C code is reset to its initialized value.

The default is setting `control`.

> **Note**: When option state is used, any arrays implemented with RAMs will typically be initialized after reset. Remember, most arrays implemented as a RAM are defined as statics and therefore imply that all elements be initialized to zero: even if the elements do not explicitly initialization.

For a large memory, this reset behavior may take many clock cycles and required more area resources to implement.

Example Tcl commands specifying all the attributes discussed in this section are shown below (additional options can be reviewed in the `config_rtl` man page). These commands perform the following:

- Set clock period as 10 ns
- Add a technology library
- Set the clock uncertainty to 2ns (if not specified, this defaults to 12.5% of the clock period).
- Create an RTL output with active low asynchronous reset on all registers.
- Use one-hot encoding for all state machines.

```
open_solution solution2

set_part  {xc6vlx365tff1759-3}

create_clock -period 10ns
set_clock_uncertainty 2

config_rtl my_func -encoding onehot –reset all -reset_level low -reset_async
```

Once these basic specifications are set and defined, the design can be optimized as outlined in the remaining sections of this chapter.

Function Optimizations

A complete list of the optimizations which can be specified upon a function can be seen in the GUI (Figure 46):

1. Select the source code in the Project Explorer window.
2. View the directives tab in the Auxiliary Pane.
3. Select a function, right-click with the mouse & select "Insert Directives"
4. Choose a directive from the menu and select the appropriate options

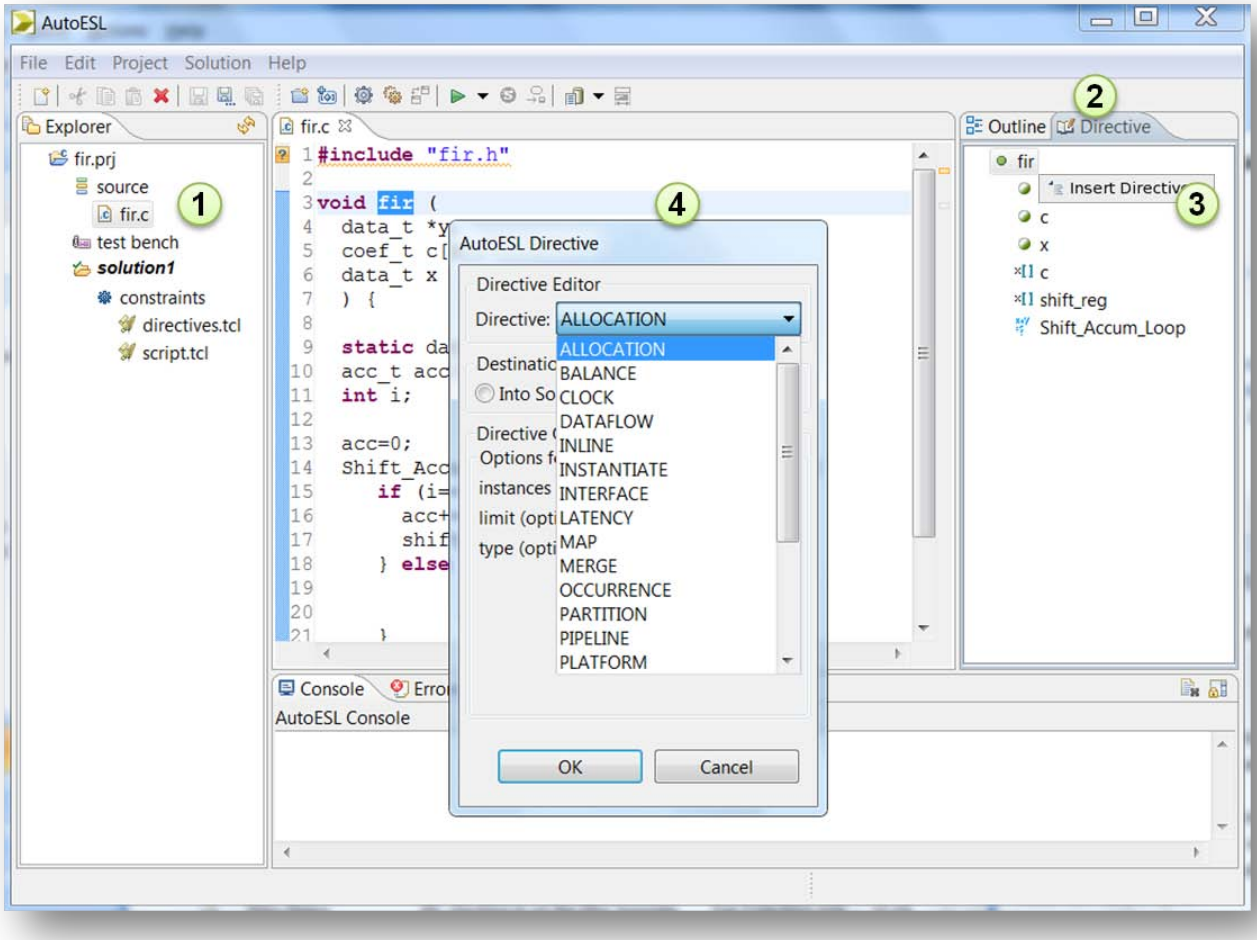

Figure 46 GUI Function Directives

Not all of the directives which can be applied at the function level are related to the optimizations of functions.

For example, when applied to a function, the RESHAPE directive is applied to all arrays in the function (in effect, a short-hand way to apply the directive to multiple arrays simultaneously). Array optimizations are discussed in a subsequent chapter.

Table 19 lists the directives which impact the behavior and optimization of functions themselves and the order in which the optimizations are discussed in this chapter.

GUI Directive	Description
Inline	Inlines a function, removing all function hierarchy. Used to help improve latency and throughput by reducing function call overhead.
Instantiate	Allows functions to be locally optimized.
Dataflow	Enables concurrency at the function level and used to improve throughput and latency.
Pipeline	Improves throughput of the function by allowing the concurrent execution of operations within a function.
Latency	Allows a minimum and maximum latency constraint to be specified on the function.
Interface	Applies function level handshaking. The synthesized control ports start, done and idle enables test bench re-use. This option is more fully discussed in the "Interface Management" chapter.

Table 19 Function Optimizations

Function Re-use, Inlining & Instantiation

AutoESL supports the synthesis of a hierarchy of function calls. By default, each function gets mapped to a specific hardware implementation. Calling the same function multiple times within the same enclosing function reuses the same hardware, just like instantiating a block in an RTL design.

Function re-use is a highly effective way of ensuring resource sharing and keeping the design smaller, since it guarantees all operations within the function are shared. If the function hierarchy is removed, it is unlikely the same level of sharing can be achieved by trying to re-assemble the individual operations at multiple locations (it's more likely that local optimizations at different locations will limit sharing).

Function inlining

There is typically a cycle overhead to enter and exit functions and removing the function hierarchy can mean improved latency and throughput.

Function inlining can be used to remove function hierarchy, often at the expense of area. If however a function is only called a few times and/or is small, inlining the function may actually improve area by allowing the few components within the function to be better shared. There are other situations where inlining a function provides benefits.

- Improve function sharing
- Allow architecture exploration

All instances of a function will have the same implementation but to ensure functions are shared they must be called within the same enclosing function and at the same level of hierarchy. This may require inlining some other functions.

In this code example, function calls “foo_1” and “foo_2” may be shared but not function call “foo_3” which is at a different level of hierarchy.

```
foo {x,y} {
       ...
}
foo_sub (p, q) {
       int q1 = q + 10;
       foo(p1,q);   // foo_3
       ...
}
void foo_top { a, b, c, d} {
       ...
       foo(a,b);    //foo_1
       foo(a,c);    //foo_2
       foo_sub(a,d);
       ...
}
```

In the above example, function inlining can be used to increase sharing by removing the hierarchy created by function “foo_sub”. This can be performed in the GUI as shown in Figure 46 or using the `set_directive_inline` command.

```
set_directive_inline foo_sub
```

The inlining directive optionally allows all functions below the specified function to be recursively inlined: if the `-recursive` option was used in the above example, function “foo_sub” would be inlined as would function call “foo_3”. If the recursive option is used on the top-level function, all function hierarchy in the design will be removed.

The `-off` option can optionally be applied to functions to prevent them being inlined. If the following commands are applied to the example above:

```
set_directive_inline –region -recursive foo_top
set_directive_inline -off foo_sub
```

All functions in the region of “foo_top” would be inlined, recursively down the hierarchy. Function “foo” would be inlined for function calls “foo_1” and “foo_2” but not “foo_3”, which is inside “foo_sub” since the `-off` option applied on function “foo_sub” will prevent inlining.

Function inlining is a powerful way to substantially modify the structure of the code without actually performing any modifications to the source and provides a very powerful method for architectural exploration.

Function Instantiation

Function instantiation is a an optimization technique which has the area benefits of maintaining the function hierarchy but provides an additional powerful option: performing targeted local optimizations on specific instances of a function. This can simplify the control logic around the function call and potentially improve latency and throughput.

Function instantiation exploits the fact that some inputs to a function may be a constant value when the function is called and uses this to both simplify the surrounding control structures (which are typically creating the constants) and produce smaller more optimized function blocks. This is best explained by example.

Given the following code:

```
void A(){
  B(true);
  B(false);
  B(true);
  B(false);
}

void B(bool mode){
  if (mode) {
     // code segment 1
  } else {
     // code segment 2
  }
}
```

It is clear that function “B” has been written to perform multiple but exclusive operations (depending on whether mode is true or not). Each instance of function “B” will be implemented in an identical manner: this is great for function re-use and area optimization but means that the control logic inside the function must be more complex.

Function instantiation can be performed from the directives tab in the GUI, by inserting pragmas into the code or with the following Tcl command:

```
set_directive_function_instantiate -variable mode=true B
```

After function instantiation, the code will effectively be transformed to have two separate functions, each optimized for different possible values of mode, as shown:

```
void A(){
  B1();
  B2();
  B1();
  B2();
}

void B1() {
  // code segment 1
```

```
}

void B2() {
  // code segment 2
}
```

Each version of the new function “B”, that is functions “B1” and “B2”, have simplified control structures.

If the function were called at different levels of hierarchy such that function sharing is difficult without extensive inlining or code modifications, function instantiation can provide the best means of improving area: many small locally optimized copies are better than many large copies which cannot be shared.

Function Dataflow Pipelining

Dataflow pipelining takes a sequential functional description (Figure 47) and creates a parallel process architecture from it (Figure 48). Dataflow pipelining is a very powerful method for improving design throughput.

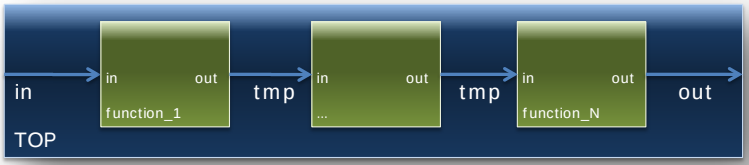

Figure 47 Sequential functional description

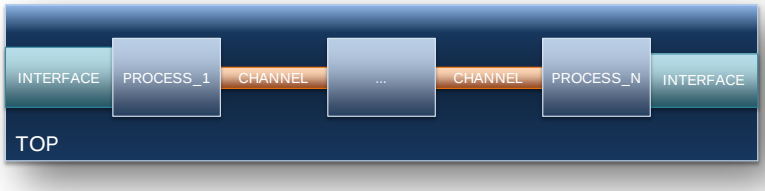

Figure 48 Parallel process architecture

The channels shown in Figure 48 ensure a function is not required to wait until the previous function has finished all operations before it can begin. Figure 49 shows how dataflow pipelining allows the execution of functions to overlap, increasing the overall throughput of the design and reducing latency.

In Figure 49(A) the implementation with dataflow pipelining requires 8 cycles before a new input can be processed by "func_A" and 8 cycles before an output is written by "func_C" (assume the output is written at the end of "func_C").

In Figure 49(B), "func_A" can begin processing a new input every 3 clock cycles (increased throughput) and it only requires 5 clocks to output a final value (shorter latency).

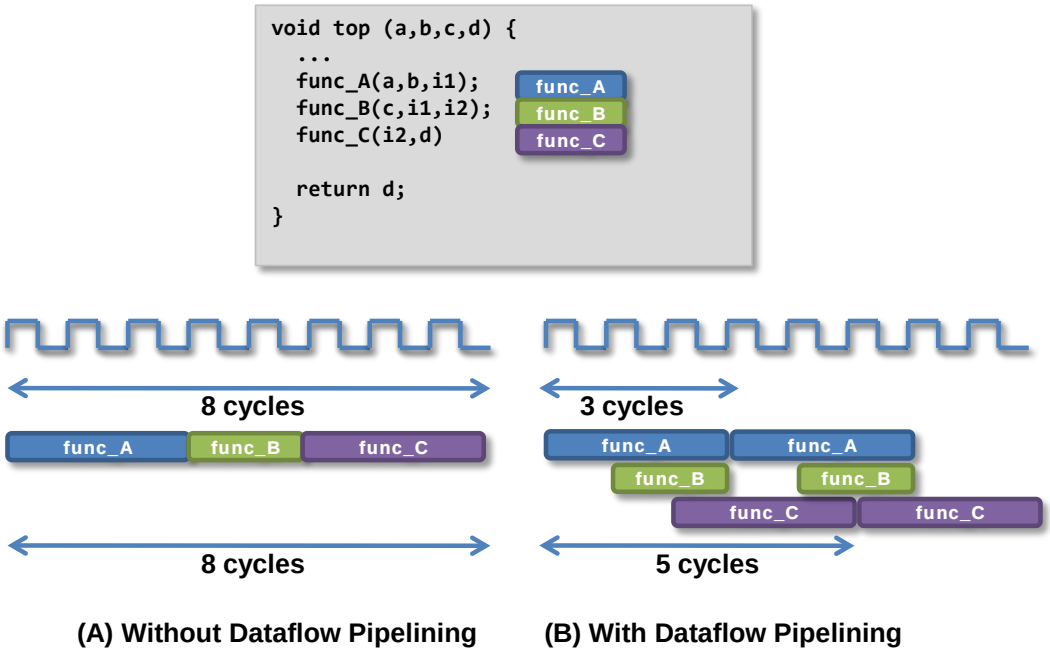

Figure 49 Dataflow pipelining behavior

The channels between the processes are implemented as either ping-pong buffers or FIFOs, depending on the access patterns of the producer and the consumer of the data.

- If the function parameter (producer or consumer) is an array the channel is implemented as a ping-pong buffer using standard memory accesses (with associated address and control signals).
- For scalar, pointer and reference parameters and the function return, the channel is implemented as a FIFO, which uses less hardware resources (no address generation) but requires that the data is accessed sequentially.

To use dataflow pipelining the arguments in each function must appear only twice: once as a producer from one function call (including return arguments) and once as a consumer in another function argument.

In addition to using the GUI directive tab, dataflow pipelining can be specified using the `set_directive_dataflow` command. When the directive is specified on a function, AutoESL will seek to improve the concurrency of the functions within it. For the example shown in Figure 49 the following command would perform function dataflow pipelining on functions “func_A”, “func_B” and “func_C”.

```
set_directive_dataflow top
```

When dataflow pipelining is used, AutoESL will by default attempt to execute each RTL block starting at the same clock edge: maximum parallel behavior. If data dependencies prevent the RTL implementation of a function from executing until an earlier function provides data (as in the Figure 49(B) where “func_B” must wait 1 clock cycle for “func_A” to generate the data for “i1”) AutoESL will automatically adjust the interval between one block starting execution and the next block starting execution, to the minimum possible number of cycles.

The `–interval` option can be used to specify exactly how many cycles there will be between an RTL block beginning execution and the next RTL block beginning execution. For example, if an interval of 3 is specified, there would be 3 cycles between the start of each function in Figure 49(B).

For scalar values, the maximum channel size will be one: only one value is passed from one function to another. When arrays are used as function arguments the number of elements in the channel (memory) is defined by the maximum size of the consumer or producer array. AutoESL does however provide a means to specify a default channel depth (refer to "Configuring " below).

When dataflow pipelining is applied to a function only the sub-functions at the current level of hierarchy will be pipelined. If a sub-function itself contains additional functions which could also benefit from dataflow pipelining, the sub-function should be inlined to ensure all functions are at the same level of hierarchy.

Configuring Dataflow Memories

The default channel used between function interfaces can be specified using the `config_dataflow` command. Configuration commands allow a default operation to be set for a solution. This command allows the default channel size and implementation to be set for all channels in a design.

```
config_dataflow -default_channel (fifo | *pingpong*) -fifo_depth <FIFO size>
```

The size of a channel is defined by the maximum size of the consumer or producer array. In some cases this may be overly conservative. The -fifo_depth option provides a means for the user to override the default behavior.

> If an array parameter is specified to use a FIFO channel, the array must be set to a streaming type array (refer to "Array Streaming" for an explanation of streaming).

If the default channel type is FIFO but a specific array has been specified as non-streaming using `set_directive_array_stream` command, the channel implementation for that array will default to a ping-pong channel (an explicit directive overrides a configuration).

Function Pipelining

Where dataflow pipelining allows the optimization of the communication between functions to improve throughput, function pipelining optimizes the operations within a function and has a similarly positive effect on throughput.

The throughput improvements in function pipelining are shown in Figure 50. Function pipelining allows operations to happen concurrently: the function does not have to complete all operations before it begin the next operation.

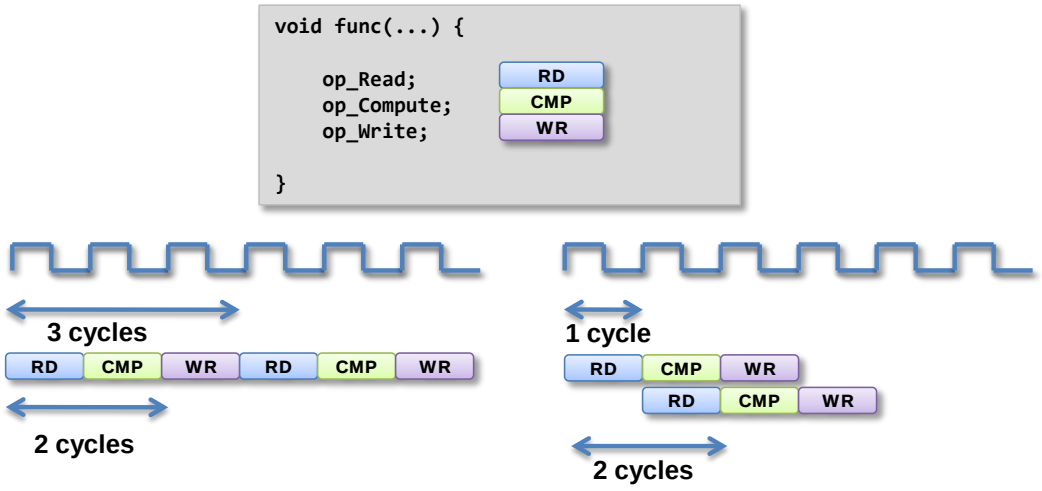

Figure 50 Function pipelining behavior

Without pipelining the function reads an input every 3 clock cycles and outputs a value every 2 clock cycles. With pipelining, a new input is read every cycle with no change to the output latency or resources used.

Function pipelining is only possible so long as there is no resource contention or data dependency which prevents pipelining. For example, in Figure 51 below, assume the input array "m[2]" is implemented with a single-port RAM. The function cannot be pipelined, as shown in Figure 51(A) because the two reads operations on input "m[2]" ("op_Read_m[0]" and "op_Read_m[1]") cannot be performed in the same clock cycle.

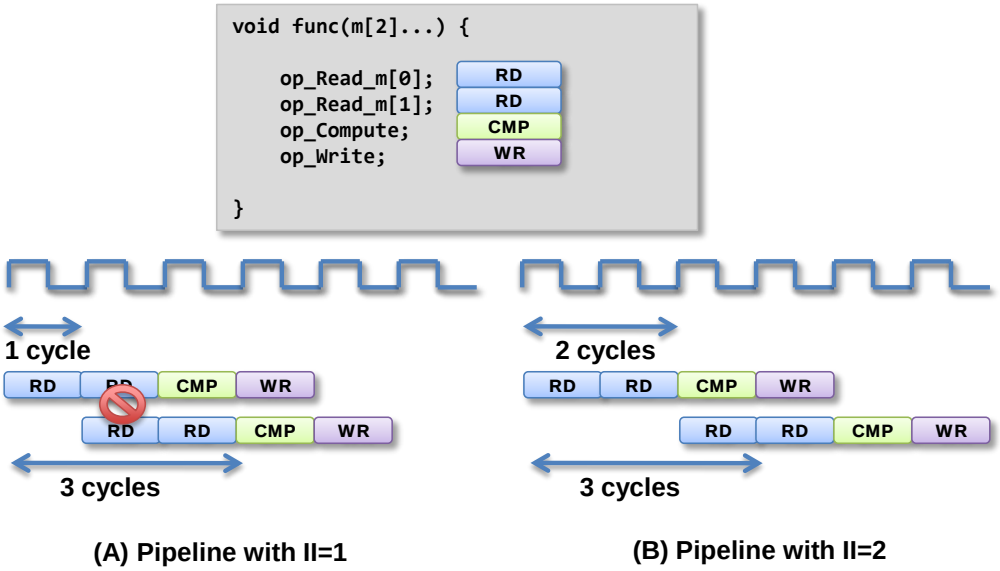

Figure 51 Pipeline Resource Contention

The function can be pipelined, as shown in Figure 51(B), by increasing the initiation interval of the pipeline. The initiation interval is the number of cycles between new input reads.

In Figure 51(A) the initiation interval is 1 because a new operation is performed every clock cycle. In Figure 51(B) an initiation interval of 2 is used - there is no longer any resource contention on the input port - and the function can be successfully pipelined.

> **Note**: Resource contentions can occur due to reads and writes on ports, access to limited resources (for example, if only 1 multiplier is available) or reads and writes on arrays mapped to a RAM or FIFO.

Review sections "Array Optimizations" and "Controlling Hardware Resources" for more details on analyzing resource contention if pipelining fails to achieve the desired initiation interval. The resource contention problem in Figure 51 could also be solved by using a dual-port RAM for array “m[2]", allowing both reads to be performed in the same clock cycle.

Function pipelining can be applied using the Tcl command

```
set_directive_pipeline function_foo
```

or applied using the directives tab in the GUI. Pipelining is applied hierarchically to the specified function: all sub-functions in the hierarchy below the specified function are automatically (and recursively) inlined and any loops in the hierarchy are automatically unrolled.

The default operation for function pipelining is to create a pipeline which runs forever and never ends. In some cases, it is desirable to have a pipeline which can

be "emptied" or "flushed" and the option `-flush` is provided to perform this. When a pipeline is "flushed" the pipeline stops reading new inputs when none are available (as determined by a data valid signal at the start of the pipeline) but continues processing, shutting down each successive pipeline stage, until the final input has been processed through to the output of the pipeline.

Latency Constraints

AutoESL supports the use of latency constraints upon a function. When a maximum and/or minimum constraint is placed on the function, AutoESL will try to ensure all operations in the function complete within the range of clock cycles specified.

Latency constraints can be applied to a function as shown in Figure 46 or they can be specified using the `set_directive_latency` command.

```
set_directive_latency -min 3 -max 5 function_foo
```

If AutoESL is unable to meet a latency constraint it will allow the timing in one of the clock cycles to be violated, as described in “Clocks, Timing & RTL output”. AutoESL will produce a design with the minimum timing violation to facilitate a strategy of meeting timing using downstream logic synthesis. If the timing violation is too great to be met using logic synthesis, review the techniques in the “Logic Structure Optimizations” chapter to reduce logic delays.

To confirm that all latency constraints have been satisfied, review the constraint report.

Function Interface Protocol

AutoESL provides the capability of automatically creating a function interface protocol. When a function is synthesized each of the function parameters, any function return value and any global variables accessed by the function are implemented as input or output ports in the final RTL design. In addition to these ports, AutoESL can synthesize function control ports which allow the RTL implementation to be more easily integrated into a surrounding system.

The interface protocol provides an input start signal ("ap_start") which must be set to logic 1 before the function will begin execution, an output signal to indicate when the function has completed all operations ("ap_done") and an output idle signal ("ap_idle") to indicate that no operations are currently being performed by the function.

The ability to automatically add a function level interface protocol means the implementation details of the protocol can be omitted from the source code description, allowing it to remain a high-level specification of the algorithm.

An interface protocol can be applied to any function in the hierarchy but it is recommended to only apply an interface protocol to the top-level function and allow AutoESL to schedule the most optimum communication between sub-functions.

A complete description of the function interface protocol is provided in the "Interface Management" chapter and a detailed waveform diagram of the protocol is shown in Figure 32.

Loop Optimizations

Within functions, C language descriptions are typically implemented as a series of loops. Understanding how loops are implemented in HLS, can be optimized and the impact of loop hierarchy is crucial to achieving optimal performance at the RT level.

A complete list of the directives which can be applied to loops can be seen in the GUI (Figure 52):

1. Select the source code in the Project Explorer window.
2. View the directives tab in the Auxiliary Pane.
3. Select a loop, right-click with the mouse & select "Insert Directives"
4. Choose a directive from the menu and select the appropriate options

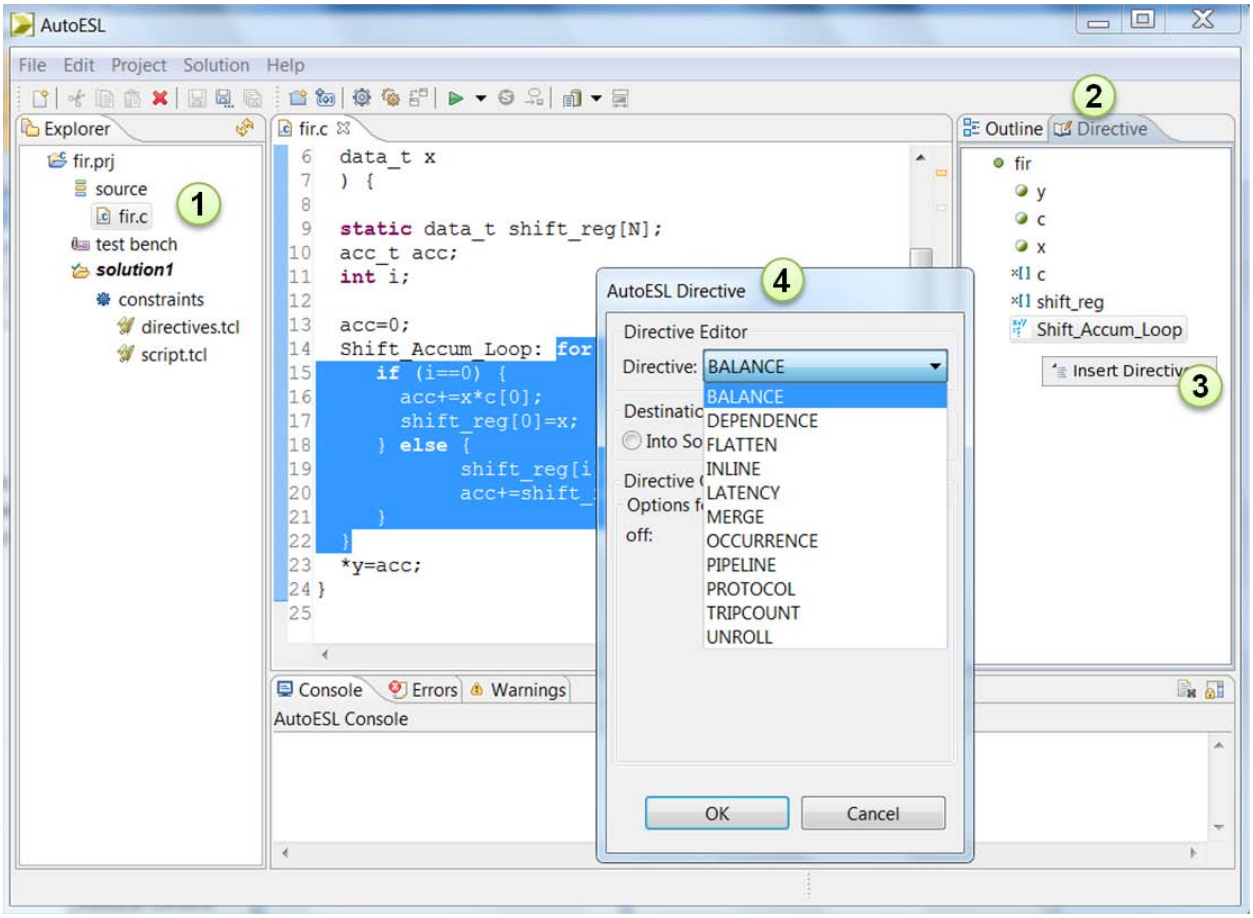

Figure 52 GUI Function Directives

Not all of the directives which can be applied to loops are related to the optimization of loops.

For example, when applied to a loop, the BALANCE directive is applied to the logic structures created with the loop. Logic optimizations are discussed in a subsequent chapter.

Table 20 lists the optimizations which can be performed on loops and the order in which the optimizations are discussed in this chapter.

GUI Directive	Description
Unrolling	Unroll for-loops to create multiple independent operations rather than a single collection of operations.
Merging	Merge consecutive loops to reduce overall latency, increase sharing and optimization.
Flattening	Allows nested loops to be collapsed into a single loop with improved latency and logic optimizations
Dataflow	Allows sequential loops to operate concurrently
Pipelining	Used to increase throughput by performing concurrent operations
Dependence	Used to provide additional information which can be used to overcome loop-carry dependencies.
Tripcount	Provides user override of iteration analysis
Latency	Specify a cycle latency for the loop operation

Table 20 Loop Level Optimizations

Unrolling Loops

By default loops are kept rolled in AutoESL. That is to say that the loops are treated as a single entity: all operations in the loop are implemented using the same hardware resources for iteration of the loop.

AutoESL provides the ability to unroll or partially unroll for-loops.

Figure 53 shows both the powerful advantages of loop unrolling and the implications which must be considered when unrolling loops. The example in Figure 53 assumes the arrays a[i], b[i] and c[i] are mapped to RAMs. If the arrays were not mapped to sequential elements, the number of cycles in the example would be determined by the combinational delay of the multiplier.

The first conclusion which can be drawn from Figure 53 is how easy it is to create many different implementations by the simple application of loop unrolling.

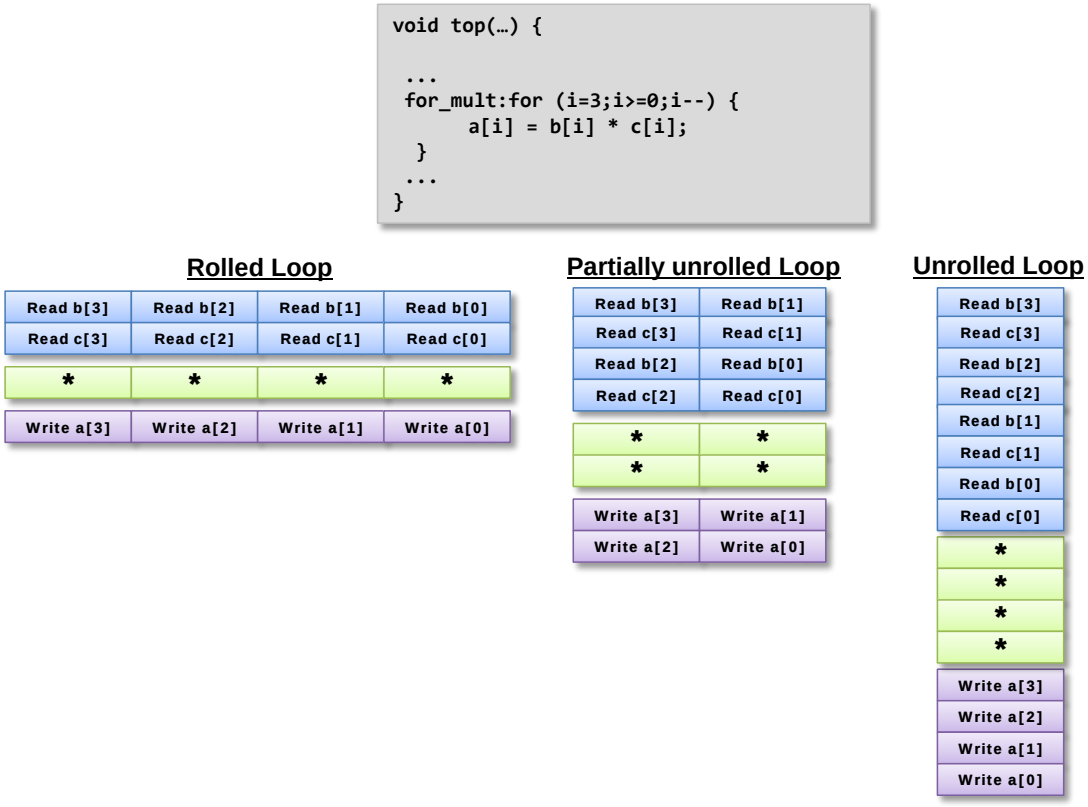

Figure 53 Loop Unrolling Details

- Rolled Loop: When the loop is rolled, each iteration will be performed in a separate clock cycle. This implementation takes four clock cycles, only requires one multiplier and each RAM can be a single port RAM.
- Partially Unrolled Loop: In this example, the loop is partially unrolled by a factor of 2. This implementation required two multipliers and dual-port RAMs to support two reads or writes to each RAM in the same clock cycle. This implementation does however only take 2 clock cycles to complete: twice the throughput and half the latency of the rolled loop version.
- Unrolled loop: In the fully unrolled version the entire loop operation can be performed in a single clock cycle. This implementation however requires four multipliers. More importantly, this implementation requires the ability to perform 4 reads and 4 write operations in the same clock cycle. Since quad-port RAMs are not common, this implementation may require the arrays be implemented as register arrays rather than RAMs, or that array partitioning and re-shaping be used.

It is safe to say that depending on how the arrays are implemented (some or all mapped to RAMs) and the delay of the multiplier, there could be many more possible implementations of this simple example.

Loop unrolling can be performed using the GUI as shown in Figure 46 by applying directives to individual loops in the design. Alternatively, loop directives can be applied to all for-loops in a function by applying the unroll directive to the function

itself as shown in Figure 46. The Tcl command can also be used to unroll specific loops:

```
set_directive_unroll -skip_exit_check -factor 2 top/for_mult
```

The `set_directive_unroll` command can only be applied to loops which are labeled, as shown in Figure 53, unless the directive is applied as a `pragma` inserted into the source code (which would apply to all versions of the code).

Partial loop unrolling does not require the unroll factor to be an integer multiple of the maximum iteration count. AutoESL will automatically add any exit checks to ensure partially unrolled loops are functionally identical to the original loop. For example, given the following code:

```
for(int i = 0; i < N; i++) {
  a[i] = b[i] + c[i];
}
```

Loop unrolling by a factor of 2 will effectively transform the code to look like the following example where the "break" construct is used, and implemented in the RTL, to ensure the functionality remains the same:

```
for(int i = 0; i < N; i += 2) {
  a[i] = b[i] + c[i];
  if (i+1 >= N) break;
  a[i+1] = b[i+1] + c[i+1];
}
```

Since N is a variable, AutoESL may not be able to determine its maximum value (it could be driven from an input port). If it is known that the unrolling factor, 2 in this case, is an integer multiple of the maximum iteration count N, the `-skip_exit_check` option can be used to remove the exit check. The effect of unrolling can now be represented as:

```
for(int i = 0; i < N; i += 2) {
  a[i] = b[i] + c[i];
  a[i+1] = b[i+1] + c[i+1];
}
```

This helps minimize the area and simplify the control logic.

Unrolling Loops in C++ Classes

When loops are used in C++ classes, care should be taken to ensure the loop induction variable is not a data member of the class as this prevents the loop for being unrolled.

In this example, loop induction variable “k” is a member of class “foo_class”.

```
template <typename T0, typename T1, typename T2, typename T3, int N>
class foo_class {
private:
```

```
       pe_mac<T0, T1, T2> mac;
public:
       T0 areg;
       T0 breg;
       T2 mreg;
       T1 preg;
       T0 shift[N];
       int k;              // Class Member
       T0 shift_output;
       void exec(T1 *pcout, T0 *dataOut, T1 pcin, T3 coeff, T0 data, int col)
       {
Function_label0:;
#pragma AP inline off
       SRL:for (k = N-1; k >= 0; --k) {
#pragma AP unroll          // Loop will fail UNROLL
              if (k > 0)
                     shift[k] = shift[k-1];
              else
                     shift[k] = data;
       }

       *dataOut = shift_output;
       shift_output = shift[N-1];
       }

       *pcout = mac.exec1(shift[4*col], coeff, pcin);
};
```

For AutoESL to be able to unroll the loop as specified by the UNROLL pragma directive, the code should be re-written to remove “k” as a class member.

```
template <typename T0, typename T1, typename T2, typename T3, int N>
class foo_class {
private:
       pe_mac<T0, T1, T2> mac;
public:
       T0 areg;
       T0 breg;
       T2 mreg;
       T1 preg;
       T0 shift[N];
       T0 shift_output;
       void exec(T1 *pcout, T0 *dataOut, T1 pcin, T3 coeff, T0 data, int col)
       {
Function_label0:;
       int k;              // Local variable
#pragma AP inline off
       SRL:for (k = N-1; k >= 0; --k) {
#pragma AP unroll          // Loop will unroll
              if (k > 0)
                     shift[k] = shift[k-1];
              else
                     shift[k] = data;
       }
```

```
        *dataOut = shift_output;
        shift_output = shift[N-1];
        }

        *pcout = mac.exec1(shift[4*col], coeff, pcin);
};
```

Merging Loops

All rolled loops imply and create at least one state in the design Finite-State-Machine (FSM). When there are multiple sequential loops this can sometimes create additional unnecessary clock cycles and prevent further optimizations.

Figure 54 shows a simple example where a seemingly intuitive coding style has a negative impact on the performance of the RTL design.

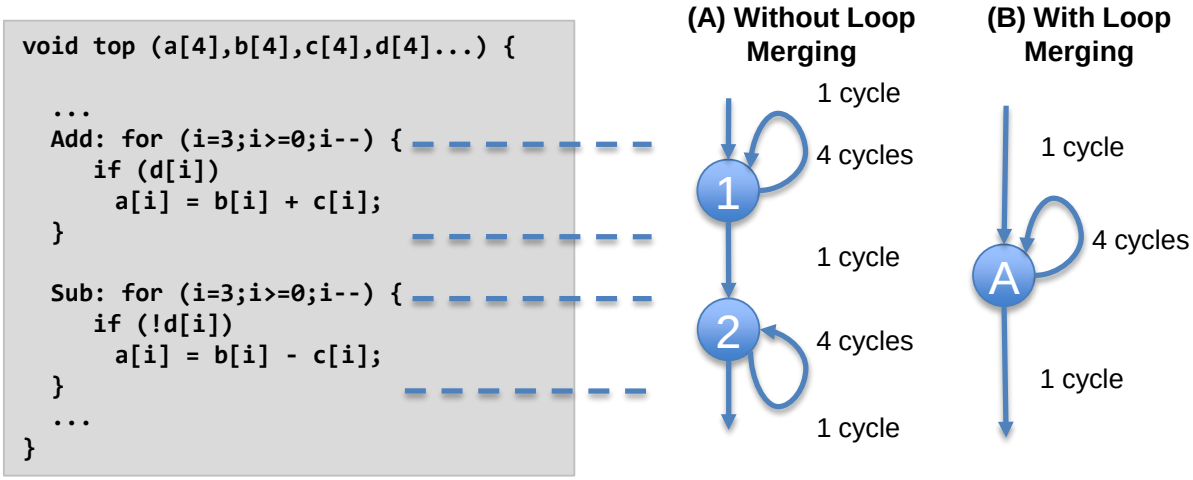

Figure 54 Loop Merging

Figure 54(A) shows how by default, each rolled loop in the design creates at least one (but perhaps more, depending on the number of operations to be performed) state in the FSM[3]. Moving between those states costs clock cycles: assuming each loop iteration requires one clock cycle, it take a total of 11 cycles to execute both loops:

- 1 clock cycle to enter the ADD loop.
- 4 clock cycles to execute the add loop.
- 1 clock cycle to exit ADD and enter SUB.
- 4 clock cycles to execute the SUB loop.
- 1 clock cycle to exit the SUB loop.

[3] There may not be a one-to-one correlation between loops and states as AutoESL may optimize and combine states.

- For a total of 11 clock cycles.

In this simple example it should become obvious that an else branch in the ADD loop would also solve the problem but in a more complex example it may be less obvious and the more intuitive coding style may have greater advantages, such as allowing colleagues in the same design team to more easily understand a very complex algorithm. (Adding statements to perform the same merge operation, such as multiple if-else statements, may make the code very unreadable).

AutoESL provides a feature to automatically merge loops. Using the directives tab in the GUI to add a MERGE directive to the function (or a region which contains both loops) would instruct AutoESL to merge the loops and create a control structure similar to that shown in Figure 54(B) which requires only 6 clocks to complete.

Currently, loop merging in AutoESL has the following restrictions:

- If loop bounds are all variables, they must have the same value.
- If loops bounds are constants, the maximum constant value is used as the bound of the merged loop.
- Loops with both variable bound and constant bound cannot be merged.
- The code between loops to be merged cannot have side effects: multiple execution of this code should generate the same results (a=b is allowed, a=a+1 is not).
- Loops cannot be merged when they contain FIFO reads: merging would change the order of the reads and reads from a FIFO or FIFO interface must always be in sequence.

Loop merging can be performed at the command line using the `set_directive_loop_merge` command (on a function as shown below or on a labeled loop or region) or it can be performed using the GUI to apply the MERGE directive.

```
set_directive_loop_merge top
```

Flattening Nested Loops

In a similar manner to the consecutive loops discussed in the previous section, it requires additional clock cycles to move between rolled nested loops. It requires one clock cycle to move from an outer loop to an inner loop and from an inner loop to an outer loop.

In the small example shown here, this implies 200 extra clock cycles to execute loop "Outer".

```
void foo_top { a, b, c, d} {
       ...
       Outer: while(j<100)
              Inner: while(i<6)   // 1 cycle to enter inner
                     ...
                     LOOP_BODY
```

```
                ...
            }                   // 1 cycle to exit inner
        }
        ...
}
```

In addition, nested loops prevent the outer loop from being pipelined, as discussed in the next section on "Loop Dataflow Pipelining".

AutoESL provides the `set_directive_loop_flatten` command to allow labeled perfect and semi-perfect nested loops to be automatically flattened, removing the need to re-code for optimal hardware performance and reducing the number of cycles it takes to perform the operations in the loop.

- Perfect loop nest: only the innermost loop has loop body content, there is no logic specified between the loop statements and all the loop bounds are constant.
- Semi-perfect loop nest: only the innermost loop has loop body content, there is no logic specified between the loop statements but the outermost loop bound can be a variable.

For imperfect loop nests, where the inner loop has variables bounds or the loop body is not exclusively inside the inner loop, designers should try to restructure the code, or unroll the loops in the loop body to create a perfect loop nest.

When the directive is applied to a set of nested loops it should be applied to the inner most loop which contains the loop body.

```
set_directive_loop_flatten top/Inner
```

Loop flattening can also be performed using the directive tab in the GUI, either by applying it to individual loops or applying it to all loops in a function by applying the directive at the function level.

Loop Dataflow Pipelining

Dataflow pipelining can be applied to loops in similar manner as it can be applied to functions. It allows loops which are sequential in nature to operate concurrently at the RTL. Dataflow pipelining should be applied to a function, loop or region which contains all function or all loops: do not apply on a scope which contains a mixture of loops and functions.

Figure 55 shows the advantages dataflow pipelining can produce when applied to loops.

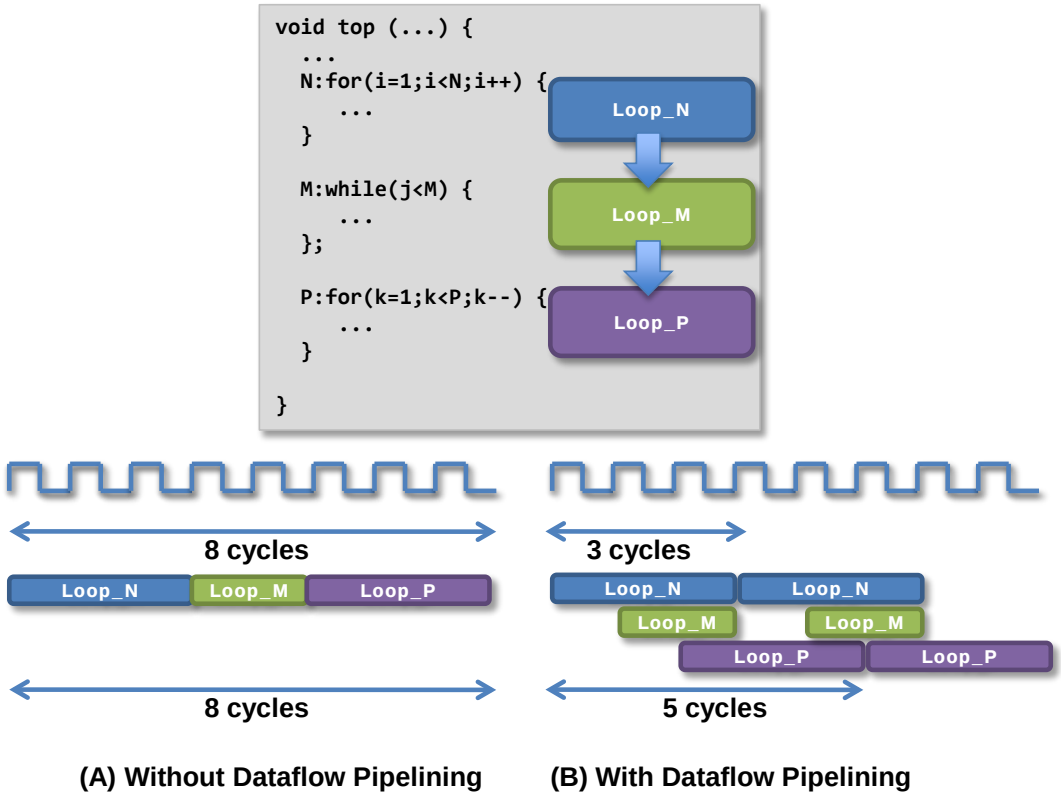

Figure 55 Loop dataflow pipelining

Without dataflow pipelining, loop N must execute and complete all iterations before loop M can begin. The same applies to the relationship between loops M and P. In this example, it is 8 cycles before loop N can start processing the next value and 8 cycles before an output is written (assuming the output is written when loop P finishes).

With dataflow pipelining, these loops can be allowed to operate in parallel, accepting new inputs every 3 cycles and outputting a value every 5 cycles: using the same hardware resources. AutoESL automatically inserts channels between the loops to ensure data can flow asynchronously from one loop to the next.

The channels between the loops are implemented as either ping-pong buffers or FIFOs.

- If the variable is an array the channel is implemented as a ping-pong buffer using standard memory accesses (with associated address and control signals).
- For all other variable types, and streaming arrays, the channel is implemented as a FIFO, which uses less hardware resources (no address generation), but requires the data is accessed sequentially.

To use dataflow pipelining the variables must be produced by one loop and consumed by only one other loop.

In addition to using the GUI directive tab, dataflow pipelining can be specified using the `set_directive_dataflow` command. When the directive is specified in a region AutoESL will seek to improve the concurrency of the loops within it. For the example shown in Figure 55 the following command would perform loop dataflow pipelining on loops “Loop_N”, “Loop_M” and “Loop_P”.

```
set_directive_dataflow -interval 2 top
```

The `–interval` option can be used to specify exactly how many cycles there will be between the start of one loop implementation and the next. By default AutoESL will try to minimize this time. Ideally AutoESL will try to have them all start of the same clock edge and execute in parallel, but data dependencies will typically prevent this. For example, if an interval of 3 is specified, there would be 3 cycles between the start of loop implementation in Figure 55(B).

The number of elements in the channel (memory) is defined by the maximum size of the consumer or producer array size but AutoESL provides a means to specifying a default channel depth (refer to "Configuring " below).

Configuring Default Channels

The default channel used between loops can be specified using the `config_dataflow` command. Configuration commands allow a default operation to be set for a solution. This command allows the default channel size and implementation to be set for all channels in a design.

```
config_dataflow -default_channel (fifo | *pingpong*) -size <FIFO size>
```

The size of a channel is defined by the maximum size of the consumer or producer array. In some cases this may be overly conservative. The -size option provides a means for the user to override the default behavior.

If an array parameter is specified to use a FIFO channel, the array is automatically converted to a streaming type (refer to "Array Streaming" for an explanation of streaming). If the default channel type is FIFO but a specific array has been specified as non-streaming using `set_directive_array_stream` command, the channel implementation for that array will default to a ping-pong channel. (An explicit directive overrides a configuration).

Loop Pipelining

In a C language description the operations in a loop are executed sequentially and the next iteration of the loop can only begin when the last operation in the loop is complete. An RTL design can execute multiple operations concurrently and it is often desirable that the RTL be implemented to perform in this manner.

Loop pipelining allows the operations in a loop to be implemented in a concurrent manner as shown in Figure 56. The default sequential operation is shown in Figure 56(A) where there are 3 clock cycles between each input read and it requires 8 clock cycles before the last output write is performed.

In the pipelined version of the loop, shown in Figure 56(B), a new input sample is read every cycle and the final output is written after only 4 clock cycles: substantially improving both the throughput and latency while using the same hardware resources, since the only changes to the design are in the control logic.

The number of cycles between new input reads is called the pipeline initiation interval and is specified by the user but defaults to 1 if not specified.

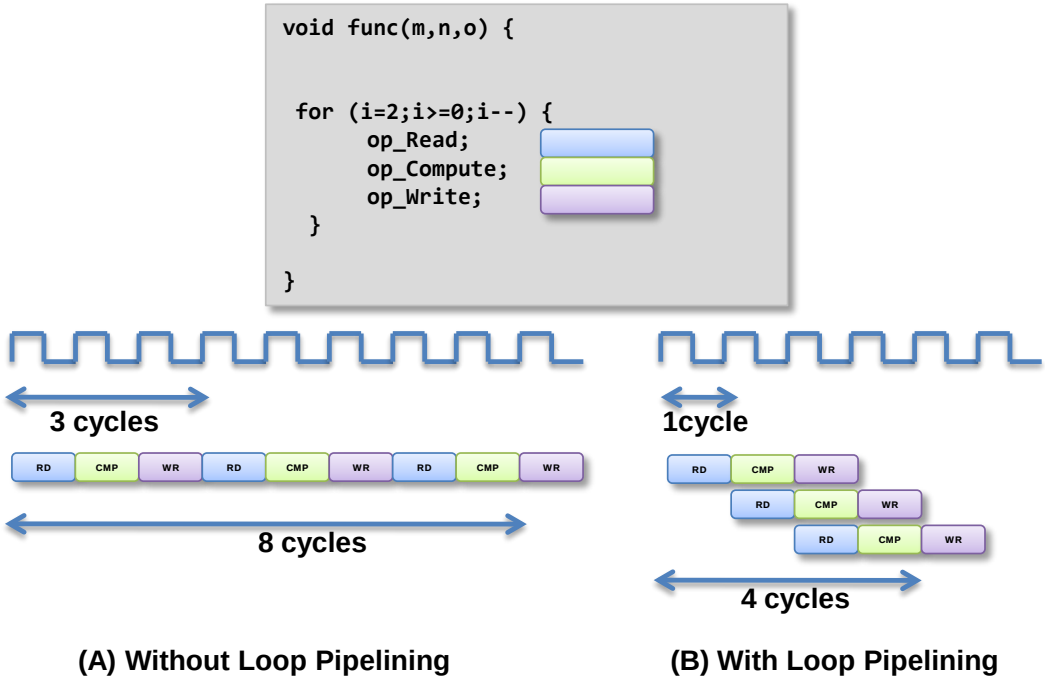

Figure 56 Loop pipelining

Loop pipelining can be prevented due to resource contention, as shown for pipelined functions back in Figure 51, and data dependencies.

Only the inner-most loop in a series of nested loops can be pipelined, however pipelining will automatically flatten any loops in the hierarchy below the pipelined region.

Data dependencies are a much harder issue to resolve and often require changes to the source code. A scalar data dependency could look like:

```
while (a != b) {
   if (a > b) a -= b;
   else b -= a;
 }
```

Obviously, the next iteration of this loop can not start until the current iteration has calculated the updated the values of `a` and `b` (Figure 57).

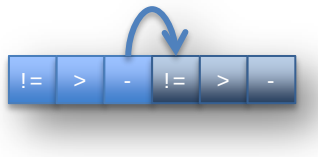

Figure 57 Scalar dependency

Dependencies are common with memory accesses:

```
for (i = 1; i < N; i++)
  mem[i] = mem[i-1] + i;
```

In this case, the next iteration of the loop must wait until the current iteration updates the content of the array (Figure 58).

Figure 58 Memory dependency

If the result of the previous loop iteration must be available before the current iteration can begin, loop pipelining is not possible. If AutoESL cannot pipeline with the specified initiation interval it will automatically increase the initiation internal (in effect, pulling the next iteration into the current iteration to remove the dependency). If it cannot pipeline at all, as shown by the above examples in Figure 57 and Figure 58, it halts pipelining and proceeds to output a non-pipelined design with reduced throughput.

Loop pipelining can be specified on loops using the directives tab in the GUI and labeled loops can be pipelined using the Tcl command as shown in this example:

```
set_directive_pipeline -II 5 -enable_flush foo/sum_loop
```

The initiation interval of the pipeline is specified as 5 in this example, meaning a new input will be read every 5 clock cycles and flushing is enabled. Flushing means the pipeline is constructed such that it can be shutdown and cleanly emptied. The default, with no flushing, creates a pipeline which will run continuously until the design is reset.

Loop Carry Dependencies

Loop pipelining can be prevented by loop carry dependencies as explained in the previous section ("Loop Pipelining"). However, under certain complex scenarios

automatic dependence analysis can be too conservative and fail to filter out false dependencies.

For instance:

```
void foo(int A[3*N], int x)
{
   LF: for (i = 0; i < N; i++)
            A[i+x] = A[i] + i; // User knows that 2*N > x >= N
}
```

In the above example, the AutoESL does not have any knowledge about the range of input parameter "x" and will conservatively assume that there is always a dependence between the write to "A[i+x]" and the read from "A[i]" and schedule the loop iterations sequentially.

To overcome this deficiency, the user can specify the dependence directive to provide AutoESL with additional information about the loop-carried dependencies on one or multiple variables. Here we can inform the tool that no loop-carried dependencies would exist if we know in advance that x is no less than N (and no greater than 2*N).

```
set_directive_dependence -variable x -type inter -dependent false foo/LF
```

When specifying dependencies there are two main types:

- Inter: specifies the dependency is between different iterations of the same loop. If this is specified as false it will allow AutoESL to perform operations in parallel if the loop is unrolled or partially unrolled and prevents such concurrent operation when specified as false.
- Intra: specifies dependence within the same iteration of a loop, for example an array being accessed at the start and end of the same iteration. When intra dependencies are specified as false AutoESL may move operations freely within the loop, increasing their mobility and potentially improving performance or area. Obviously when the dependency is specified as true, the operations must be performed in the order specified.

Loop Iteration Control

AutoESL performs analysis to automatically determine the maximum possible iteration of a loop, however it is not possible for AutoESL to determine the actual maximum iteration of a loop.

In the following example, the maximum iteration of the for-loop is determined by the value of input "num_samples". AutoESL can determine that the maximum possible value of this variable is 15 (it is of type uint4), but it cannot know that the actual value of "num_samples" is, for example, never above 9 due to an external constraint.

```
void foo (uint4 num_samples, ...);

void foo (num_samples, ...) {
   int i;
   ...
   loop_1: for(i=0;i< num_samples;i++) {
       ...
       result = a + b;
   }
}
```

If the latency or throughput of the design is dependent on a loop with a variable index, AutoESL will not be able to report the correct values for throughput or latency. AutoESL will report the latency of the loop as being unknown (represented in the reports by a question mark "?").

Specifying the loop iterations, or tripcount, ensures the report contains valid numbers. This can be specified using the TRIPCOUNT directive which can be applied via the GUI using the directives tab or specified on labeled loops using the Tcl command line:

```
set_directive_loop_tripcount -min 3 -max 8 -avg 5 foo/loop_1
```

The `-max` option tells AutoESL the maximum number of iterations the loop will ever iterate. The `-min` option specifies the minimum number of iterations which will be performed and the `-avg` option specifies an average tripcount. This ensures the AutoESL reports accurately reflect the maximum and minimum design throughput.

The tripcount directive does not impact the results of synthesis. The tripcount values are only used for reporting, where they help to ensure the reports generated by AutoESL show meaningful ranges for latency and throughput.

Loop Latency

The maximum and minimum latency for a loop can be specified as a constraint. This is a means of ensuring the performance targets are met and a powerful way to control how the resources in a loop are used.

By default AutoESL will seek to meet timing and then minimize latency. If timing cannot be met, latency will be extended until timing can be met. If there is a latency constraint which prevents latency being extended, AutoESL will allow local timing violations as detailed in the "Clocks, Timing & RTL output" section. Given the latency achieved with the default settings:

- Setting a maximum latency which is less than this will cause AutoESL to work harder to meet the lower latency value by trying to improve operator sharing and chaining.

- Setting a minimum latency which is higher than this will allow AutoESL to take more clock cycles to complete the operations, allowing increased sharing and potentially meeting timing on any paths which are currently failing.

Latency constraints are applied on each loop separately via the GUI or using the `set_directive_latency` command using the loop label.

Array Optimizations

The memory configurations in a design have a great impact on the performance and area of the overall design. Arrays in a C language description are typically mapped to memories and so the optimizations performed on arrays have a great impact on both area and performance.

A complete list of the directives which can be applied to arrays can be seen in the GUI (Figure 59):

1. Select the source code in the Project Explorer window.
2. View the directives tab in the Auxiliary Pane.
3. Select an array, right-click with the mouse & select "Insert Directives"
4. Choose a directive from the menu and select the appropriate options

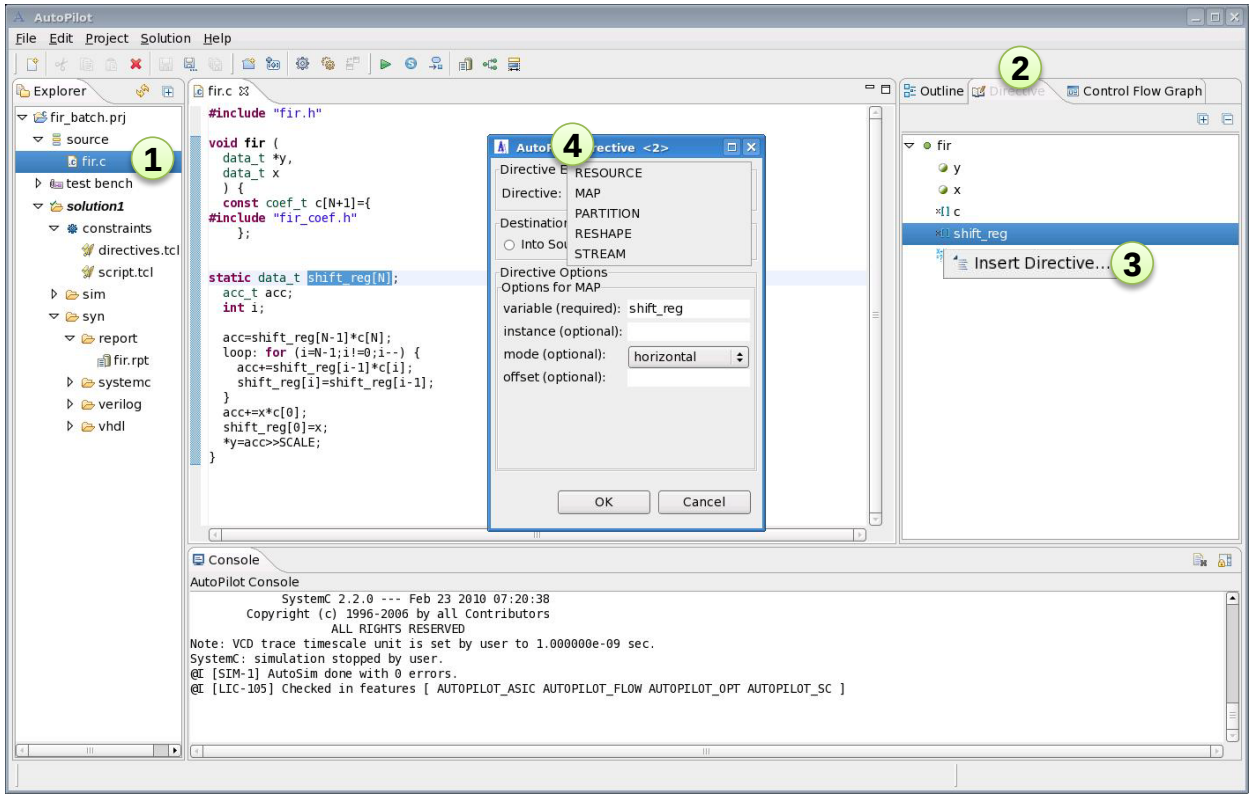

Figure 59 GUI Array Directives

Table 21 lists the optimizations which can be performed on arrays and the order in which they are discussed in this chapter.

GUI Directive	Description
Resource	Specify which hardware resource (RAM component) a array maps to.
Map	Reconfigures array dimensions by combining multiple smaller arrays into a single large array to help reduce RAM resources and area.
Partition	Control how large arrays are partitioned into multiple smaller arrays to reduce RAM access bottleneck. Also used to ensure arrays are implemented as registers and not RAMs.
Reshape	Can reshape an array from one with many elements to one with greater word-width. Useful for improving RAM accesses without using many RAMs.
Stream	Specifies that an array should be implemented as a FIFO rather than RAM.

Table 21 Array Optimizations

Arrays in a C language description are generally implemented using memory resources at the RTL. This chapter explains how arrays can be implemented using specific RAMs or ROMs, how they can be transformed (split horizontally, split vertically, aggregated or combinations of these operations) to ensure they map efficiently into the available memory resource and how they can be decomposed into individual registers.

Arrays which are specified as arguments to the top-level function are synthesized in a slightly different manner. In this case it is assumed the memory resource is outside the design and the array is synthesized into ports which can access this resource. The same transformations can be performed upon such arrays: the result of synthesis will simply be ports which can access the transformed array.

Array Initialization & Reset

AutoESL supports the initialization of arrays in the source code. The initialization value of arrays is replicated in the RTL and in the FPGA, where the bitstream is used to ensure the RAM (or registers if the array is partitioned) is initialized to the same values when the device is powered-up.

The following code example shows,

- An example where the array is initialized from file `fir_coef.h`.
- An example where the `const` qualifier is used: AutoESL will know the array should be implemented as a ROM.

If the `const` qualifier is not used AutoESL can generally detect when arrays are only read from and thus should be implemented as a ROM, however it is always best to

specify explicitly which memory resource should be used to implement the array. Resource selection is discussed in the next section.

```
typedef int  coef_t;
typedef int  data_t;
typedef int  acc_t;

     data_t fir (
      data_t x
      ) {
      static data_t shift_reg[N];
      acc_t acc;
      int i;

     const coef_t c[N+1]={
#include "fir_coef.h"
      };

      acc=0;
      mac_loop: for (i=N-1;i>=0;i--) {
         if (i==0) {
           acc+=x*c[0];
           shift_reg[0]=x;
         } else {
           shift_reg[i]=shift_reg[i-1];
           acc+=shift_reg[i]*c[i];
         }
      }
      return=acc;
     }
```

Arrays will be initialized to the values specified in the source code at power-up, however subsequent applications of the reset port to the device will not return the array to this initial power-on state: the reset of arrays is not supported through any reset added by AutoESL.

If arrays must be returned to some initial (reset) state when an external signal is applied, this must be explicitly coded into the design. An example of this is:

```
...
     const coef_t c_temp[N+1]={
#include "fir_coef.h"
      };
     coef_t c[N+1];

     if (rst_array==1) {
       for (i=0;i<N;i++) {
              c[i] = c_temp[i];
       }
     }
...
```

This however may result in two undesirable attributes in the RTL design:

- Unlike a power-up initialization, this type of explicit reset requires that the RTL design iterate through each address in the array/RAM to set the value: this can take many clock cycles if N is large.
- This requires that an additional variable, `rst_array` in this case, is added to the top-level function interface. This port would be in addition to any reset signal which is added during synthesis.

Memory Resource Selection

If no memory resource is specified for an array, AutoESL will automatically determine which memory resource (single-port, dual-port, etc.) will be used. The same applies to arrays specified as function arguments on the top-level function: AutoESL may create an interface to a dual-port memory since it allows higher throughput. This is not guaranteed to be the best choice and so users are encouraged to specify exactly which memory resource each array should map to.

Arrays are mapped to a specific RAM resource using the `set_directive_resource` command as shown below or this can be specified in the GUI as shown in Figure 59 (or inserted as a pragma in the code).

Given the following example with three arrays, one specified as a function parameter and two defined within the function,

```
void foo (in[16], ...) {
   int8  array1[16];
   int12 array2[48];
   ...
       loop_1: for(i=0;i<8;i++) {
              array1[i] = in[i];
              ...
       }
}
```

Interface Resources

The following will specify the type of RAM which supplies the data to port in[16]:

```
set_directive_resource -core RAM_1P foo in
```

The `-core` option specifies the RAM core: a complete list of RAM cores is available in the "AutoESL Library Guide" and can be selected from the RESOURCE directive window in the GUI.

If parameter "in[16]" is specified as having an `ap_memory` interface, the ports created will match the ports on RAM_1P. If RAM_1P has a chip-enable (CE) port, AutoESL will create an interface with a CE port. If the RAM_1P has separate address ports for reading and writing, AutoESL will create an address port to read the RAM and an address port to write to the RAM. If it has a single address port for both, AutoESL will create a design with a single address port for both read and write operations.

If port in[16] is specified as an `ap_fifo` the type of memory resource specified is of less importance since an `ap_fifo` interface is always the same: data ports with read, write, empty and full ports.

Refer to chapter "Interface Management" for more details on selecting an interface.

Design Resources

Using the same code example above, the following commands specify the type of RAM used to implement arrays "array1" and "array2".

```
set_directive_resource -core RAM_1P foo array1
set_directive_resource -core RAM_2P foo array2
```

In this case, "array1" is mapped to core RAM_1P and "array2" is mapped to core RAM_2P. At this stage, the following must be satisfied:

- RAM_1P must have more than 16 elements (addresses) and each element must be greater than 8-bits, since int8 is an 8-bit datatype.
- RAM_2P must have more than 28 elements and each element must be more than 12-bits (int12 is a 12-bit datatype).

Even if port "in1" has already been specified as communicating with a RAM_1P component outside the function, there is no requirement that "array1" be targeted to the same type of RAM component. AutoESL will perform any transformation necessary to read from one type of RAM or RAM port and write to another.

Array Mapping

In most technology libraries the RAMs provided have pre-defined sizes (e.g. power-of-2 depth, with 1,8,16-bit words). When there are many small arrays in the original specification, mapping them into a single large array before specifying a target resource may reduce the storage overhead. If each small array gets a separate memory, a lot of memory space is potentially wasted and the design will be unnecessarily large.

The AutoESL `set_directive_array_map` command supports two ways of mapping small arrays into a larger one:

- Horizontal mapping: this corresponds to creating a new array by concatenating the original arrays. Physically, this gets implemented as a single array with more elements.

- Vertical mapping: this corresponds to creating a new array by concatenating the original words in the array. Physically, this gets implemented by a single array with a larger bit-width.

Horizontal Mapping

The following code example has two arrays which would result in two RAM components.

```
void foo (...) {
   int8  array1[M];
   int12 array2[N];
   ...
       loop_1: for(i=0;i<M;i++) {
              array1[i] = ...;
              array2[i] = ...;
              ...
       }
   ...
}
```

Arrays "array1" and "array2" can be combined into a common array, specified as "array3" in the following example:

```
set_directive_array_map -instance array3 -mode horizontal foo array1
set_directive_array_map -instance array3 -mode horizontal foo array2
```

The MAP directive, which can also be specified in the GUI by selecting the individual arrays, transforms the arrays as shown in Figure 60.

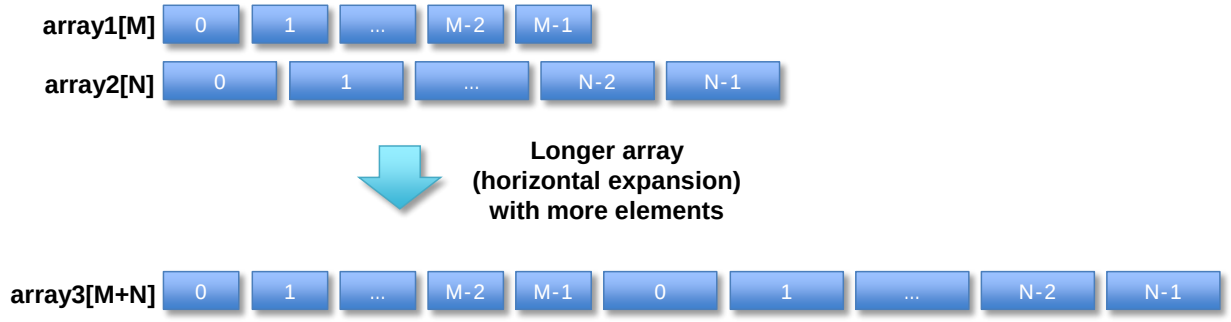

Figure 60 Horizontal mapping

When using horizontal mapping the smaller arrays are mapped into a larger array, starting at location 0 in the larger array, and in the order the commands are issued (in the AutoESL GUI it is the order the arrays are specified using the menu). The `-offset` option can be used to add additional elements between the original arrays.

To repeat the previous example, but reversing the order of the commands (adding "array2" then "array1") and adding an offset, as shown below,

```
set_directive_array_map -instance array3 -mode horizontal foo array2
set_directive_array_map -instance array3 -mode horizontal -offset 2 foo array1
```

would result in the transformation shown in Figure 61.

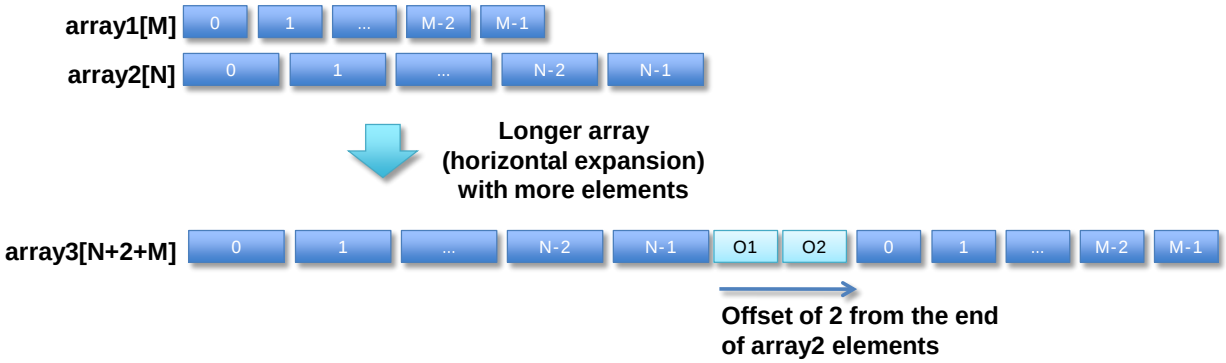

Figure 61 Horizontal mapping with offset

After mapping the newly formed array, "array3" in the above examples, can be mapped into a single RAM component.

```
set_directive_resource -core RAM_1P foo array3
```

The RAM implementation shown in Figure 62 corresponds to the mapping in Figure 60 (no offset is used).

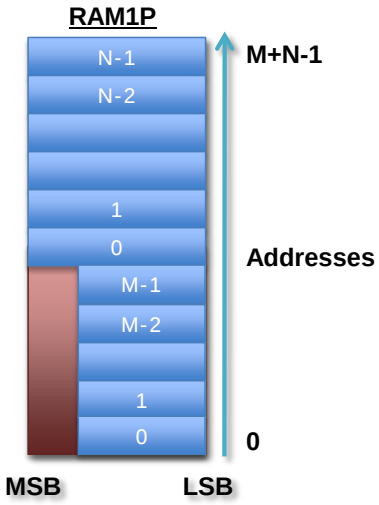

Figure 62 Memory for horizontal mapping

Although horizontal mapping can result in using less RAM components and hence improve area, it can have an impact on throughput and performance. In the above example both the accesses to "array1" and "array2" in "loop_1" can be performed in the same clock cycle. If both arrays are mapped to the same RAM this will now require a separate access, and clock cycle, for each read operation.

To overcome this limitation, AutoESL provides vertical mapping.

Vertical Mapping

In vertical mapping, arrays are concatenated by to produce an array with higher bit-widths. Figure 63 shows how the same example used in horizontal mapping discussion is transformed when vertical mapping mode is applied.

```
set_directive_array_map -instance array3 -mode vertical foo array2
set_directive_array_map -instance array3 -mode vertical foo array1
```

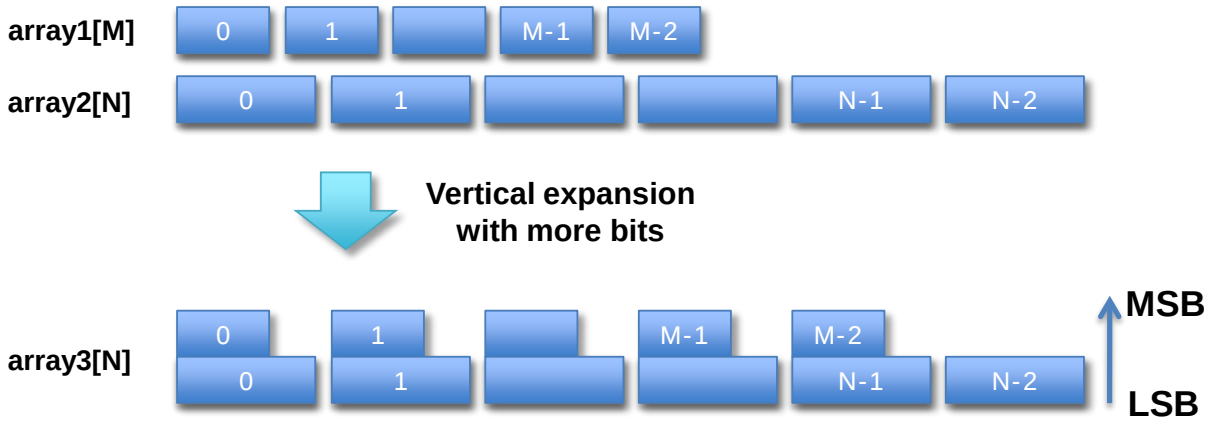

Figure 63 Vertical mapping

In vertical mapping the arrays are concatenated in the order specified by the command, with the first arrays starting at the LSB and the last array specified ending at the MSB. (Note, "array2" was specified first by the set_directive_array_map command. In the AutoESL GUI it is the order the arrays are specified using the menus).

After mapping the newly formed array, "array3" in the above examples, can be mapped into a single RAM component (Figure 64).

```
set_directive_resource -core RAM_1P foo array3
```

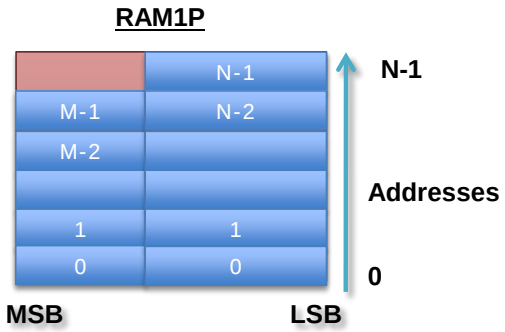

Figure 64 Memory for vertical mapping

Mapping & Global Arrays

It is possible to map a global array. However the resulting array instance will be global and any local arrays mapped onto this same array instance will become global.

When local arrays of different functions get mapped onto the same target array, then the target array instance becomes global.

When array function parameters are mapped, they need to be parameters of the same function.

Array Partitioning

Arrays can also be partitioned into smaller arrays. Memories only have a limited amount of read ports and write ports which can limit the throughput of a load/store intensive algorithm. The bandwidth can sometimes be improved by splitting up the original array (a single memory resource) into multiple smaller arrays (multiple memories), effectively increasing the number of ports.

Partitioning a larger array into smaller arrays with the `set_directive_array_partition` command can hence improve the throughput.

AutoESL provides three types of array partitioning, as shown (Figure 65). The various types are specified with the `-type` option:

- block: the original array is split into equally sized blocks of consecutive elements of the original array.
- cyclic: the original array is split into equally sized blocks interleaving the elements of the original array.
- complete: the default operation is to split the array into its individual elements. This corresponds to resolving a memory into registers.

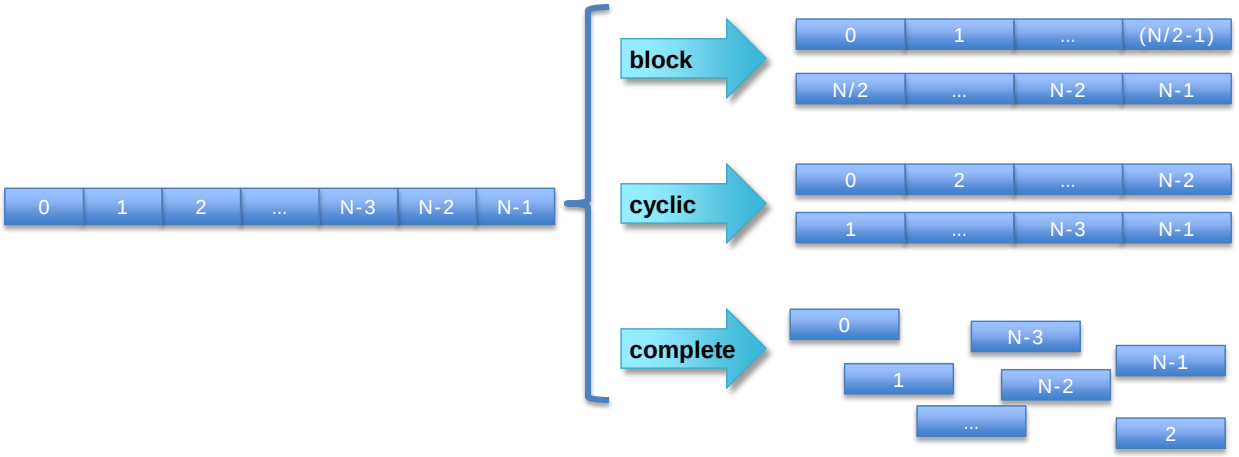

Figure 65 Array partitioning

For block and cyclic partitioning the `-factor` option can be used to specify the number of array which are created. In Figure 65 a factor of 2 is used - the array is divided into two smaller arrays. If the number of elements in the array is not an integer multiple of the factor, the final array has fewer elements.

When partitioning multi-dimensional arrays, the `-dim` option can be used to specify which dimension is partitioned. In this example,

```
void foo (...) {
   int  array1[L][M][N];
   ...
}
```

"array1" is split into three arrays, each of which has dimension 1 of size L/3.

```
set_directive_array_partition -type block -dim 1 -factor 3 foo array1
```

If zero is specified as the dimension (`-dim 0`) all dimensions are partitioned.

Array Reshaping

The command `set_directive_array_reshape` combines array partitioning with vertical mapping. This ultimately takes different elements from a dimension in the original array, and combines them into a single element in the reshaped array.

Given the following example,

```
void foo (...) {
   int  array1[N];
   int  array2[N];
   int  array3[N];
   ...
}
```

The following commands can be used to reshape arrays "array1", "array2" and "array3" into a three new arrays, using the default factor of 2 to create block, cyclic and complete types.

```
set_directive_array_reshape -type block -instance array4 foo array1
set_directive_array_reshape -type cyclic -instance array5 foo array2
set_directive_array_reshape -type complete -instance array6 foo array3
```

Figure 66 shows the result of the above commands.

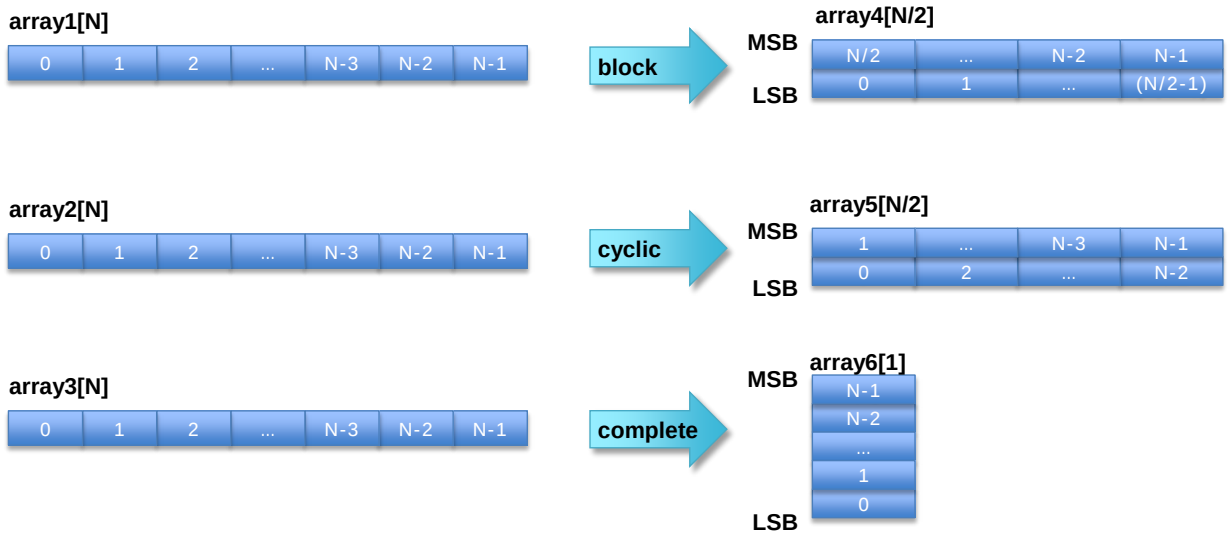

Figure 66 Array Reshaping

Array Streaming

By default all arrays are implemented as memory elements, unless complete partitioning reduces them to individual registers. This means all arrays will be assigned to a RAM resource and accessed with the data, address, chip and write enable signals specified on the RAM core in the technology library.

To use a FIFO instead of a RAM, the array must be specified as streaming. The following arrays are automatically specified as steaming:

- If an array on an interface is set as interface type `ap_fifo` it is automatically set as streaming.

All other array must be specified as streaming if a FIFO interface is required. This includes if streaming interface is required between function or loops when dataflow pipelining is specified.

An array can be specified to be streaming by

```
set_directive_array_stream  foo array1
```

The only options to the command are used when applying the command to arrays involved in dataflow channels. The `-depth` option overrides the default FIFO depth (the size of the largest array) and the `-off` option overrides the `config_dataflow` command: when the default channel is specified as a FIFO the `set_directive_array_stream -off` option can be used to prevent the array from streaming and ensure it uses a pingpong buffer.

Logic Structure Optimizations

The logic structures created by AutoESL are of the utmost importance. Once functions and loops have been optimized for concurrent operation, merged, inlined and flattened for minimum latency, pipelined for maximum throughput and array accesses have been analyzed to reduce bottlenecks, it is often ultimately the logic structures which dictate or limit the performance of a design and understanding how to improve them, the key to success.

A number of items dictate the type of logic structures implemented in a design:

- Clock Rate
- Target Device
- Operator Selection
- Controlling Hardware Resources
- Struct Packing
- Expression balancing
- Elaboration Effort

The impact of the clock, target device, state machine encoding and the reset are discussed in the "Design Optimization" chapter. These items should be reviewed and confirmed before applying any other optimization techniques discussed here.

Operator Selection

During synthesis AutoESL selects implementations for operators (+, -, *, /, %, etc.) from the device technology library. By default AutoESL chooses operators which are the best balance between timing and area. The `config_bind` can be used to influence which operators are used and to minimize the number of operators.

```
config_bind -effort [low | medium | high] -min_op <list> -reset
```

The `config_bind` command can only be issued inside an active solution. Once the command has been issued it will apply to all synthesis operations performed in the solution: if the solution is closed and re-opened the specified configuration will still apply to any new synthesis operations.

Any configurations applied with the `config_bind` command can be removed by using the `-reset` option or by using `open_solution -reset` to open the solution.

The default effort level for the binding operation is medium.

- Low Effort: Spend less timing sharing, run time is faster but the final RTL may be larger. Useful for cases where the designer knows there is little sharing possible or desirable and does not wish to waste CPU cycles exploring possibilities.
- Medium Effort: The default, where AutoESL tries to share operations but endeavors to finish in a reasonable time.

- High Effort: Try to maximize sharing and do not limit run time. AutoESL will keep trying until all possible combinations of sharing have been explored.

Effort levels impact every operator in the top-level function - bindings are set for the entire design.

The `-min_op` operation allows a particular operation to be minimized in the RTL. For example, if the design contains 12 multipliers, the following command would seek to minimize the number of multipliers in this design:

```
config_bind -min_op mul
```

Refer to the `config_bind` command in the "AutoESL Reference Guide" for a complete list of available operators.

Controlling Hardware Resources

The resources used to implement the RTL can be specified explicitly during synthesis or a general limit can be put on the resources synthesis is permitted to use. These techniques can be used to both improve timing (and hence latency and throughput) and area.

The resource used for a specific operation can be directly specified. Resources can be specified using the GUI (and as a pragma) and applied to any variable in a function using the `set_directive_resource` command.

When AutoESL reads the code shown in this example,

```
int foo (
  int a,
  int b
  ) {
  int c, d;
  c = a*b;
  d = a*c;
  return d;
}
```

the multiplications, used for variables "c" and "d", will be implemented in the internal database as standard "mul" (multiplier) operators. A complete list of operators is available in the "AutoESL Library Guide".

When synthesis is performed, AutoESL will use the timing constraints specified by the clock, the delays specified by the target device and any user constraints to determine which core is used to implement the operators: it could use the combinational core "multiplier" or it may decide to use a pipeline multiplier core such as "Mul2S". A complete list of available cores is provided in the "AutoESL Library Guide".

The RESOURCE directive can be used to explicitly specify which core should be used. The following command informs AutoESL to use a 2-stage pipelined multiplier for variable "c".

```
set_directive_resource -core Mul2S foo c
```

In addition to selecting specific operators to improve timing or area, the total number of operators used in the design can be limited to force operator sharing and improve area (often at the expense of timing or latency). For the same example code given above, the following command:

```
set_directive_allocation -limit 1 -type operation foo mul
```

limits the implementation to one `mul` operation (by default, there is no limit). This forces AutoESL to use a single multiplier for function "foo".

Struct Packing

Packing the members of a struct into a single wide-word can reduce the control overhead associated with each of the individual elements are result in both a smaller and faster design.

The PACK directive can be used to pack the three 8-bit char elements into a single wide-word element. In the new word, the first element of the struct will occupy the least significant bits and the last element, the most significant bits. If the struct contains arrays, the array with be reshaped with complete partitioning and packed with the other scalars.

Given a struct, “my_data”, used in function foo

```
typedef struct{
  unsigned char A;
  unsigned char B;
  unsigned char C;
}my_data;

void foo(my_data a_in[50], my_data b_out[50])
{

  int i;

  for(i=0; i < 50; i++){
    b_out[i].A = (a_in[i].A >> 1) + 10;
    b_out[i].B = (a_in[i].B >> 2);
    b_out[i].C = (a_in[i].C >> 3) + 100;
  }

}
```

The following commands will pack the members “a_in” into a new variable called “a_in” (it will use the same name if no –instance option is used) and pack the members of struct “b_in” into a new called “new_var”.

```
set_directive_data_pack foo a_in
set_directive_data_pack –instance new_var foo b_out
```

In both cases, the new variables will be 24-bits wide (three 8-bit char types).

Note: The maximum bit-width of any port or bus created by data packing is 8192 bits.

Expression Balancing

During synthesis a number of optimizations, such as strength reduction, bitwidth minimization etc. are performed automatically. Included in the list of automatic optimizations is expression balancing.

One of the optimizations which can be directly controlled is expression balancing. This optimization rearranges operators to construct a balanced tree and reduce latency. Expression balancing is on by default but may be disabled.

Software programmers often will write highly sequential code by using assignment operators (such as += and *=) for convenience. However, this sequential code may have an adverse effect on latency.

Let us look at the following example:

```
int foo_top (short a, short b, short c, short d)
{
  int i;
  int sum;

  sum = 0;
  sum += a;
  sum += b;
  sum += c;
  sum += d;
  return sum;

}
```

The code above must be executed sequentially. Suppose each addition requires one clock cycle, the complete computation for "sum" requires four clock cycles shown in Figure 67.

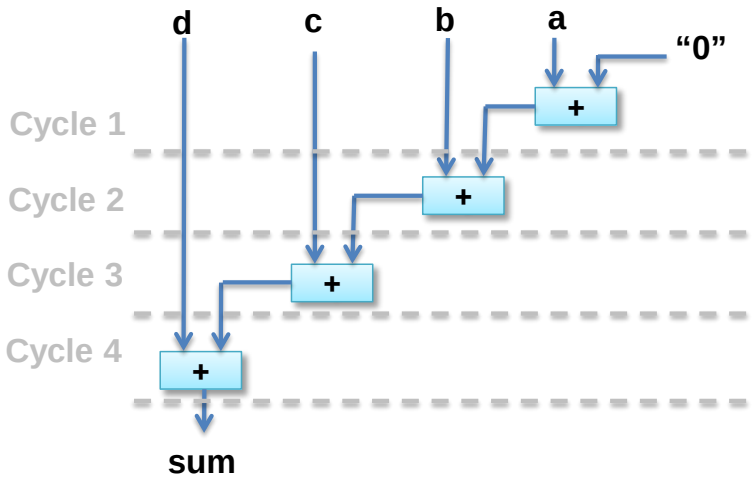

Figure 67 Adder Tree

However additions "(a+b)" and "(c+d)" can be executed in parallel allowing the latency to be reduced. After balancing the computation completes in two clock cycles as shown in Figure 68.

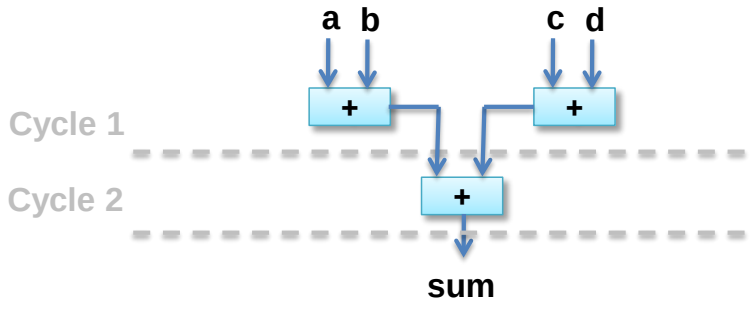

Figure 68 Adder Tree after Balancing

Expression balancing typically prohibits sharing and can result in increased area. In the example from Figure 67 a single adder can be used for the entire design, unless the design is pipelined, because only one adder is required in each clock cycle. In the example in Figure 68 a minimum of two adders are required (and three if the design is pipelined).

The following command turns off expression balancing in function "foo":

```
set_directive_expression_balance -off foo
```

Elaboration Effort

The first process performed on the input function(s) is elaboration. It is during elaboration that the functionality of the C/C++/SystemC is transformed into generic logic structures. The effort level used during elaboration has an impact on the starting point for synthesis.

By default, the effort level used during elaboration is `std` (standard). The standard effort level is typically enough for most design, because it provides the best balance between memory/CPU usage and optimization.

The effort level during elaboration can be set by using the `-effort` option.

```
elaboration -effort [low | std | high]
```

Low Effort Level

A low effort level is only recommended when the time taken to elaborate the design becomes excessively large. No optimizations will be performed on if-else or branch conditions which can result in larger area (due to less sharing).

Standard Effort Level

The default standard effort level seeks a balance between optimization and run time.

High Effort Level

When high effort level is used more optimization is performed on the initial database which helps both timing and area. Anything which increases the search space (a design with many mutually exclusive paths, many opportunities for sharing, a lack of design constraints etc.) will increase the run time.

Verification

Post-synthesis verification can be automated through the use of the `autosim` feature which can re-use the pre-synthesis test bench to seamlessly perform verification on the output RTL.

When synthesis completes AutoESL writes the output RTL to the "syn" directory as shown in Figure 69. These files can be used with an appropriately created RTL test bench to verify the design.

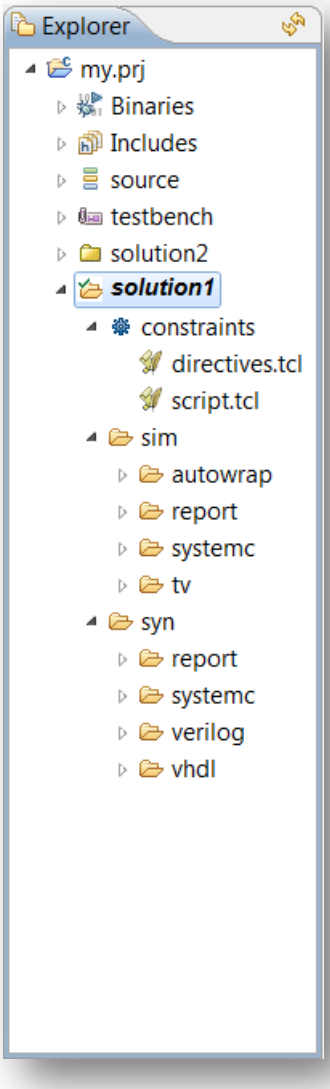

Figure 69 Output Directory Structure

However, AutoESL provides a much more productive method to verify the RTL design: `autosim`.

Automatic Verification of the RTL

The `autosim` feature can re-use the existing C-level test bench created for pre-synthesis verification and automatically verify the RTL using he built-in SystemC RTL simulator or a 3rd party HDL simulator.

The following is required in order to use the `autosim` feature successfully:

- The correct interface synthesis options must be selected.
- The test bench must be self-checking and return a value of 0.
- Any 3rd-party simulators must be available in the search path.

Interface Synthesis Requirements

In order to use the `autosim` feature to automatically verify the RTL design.

- The top-level function of C and C++ designs must be synthesized using an `ap_ctrl_hs` interface.
    - This interface creates a start, done and idle port on the design which are used to control when transactions begin and when to capture data at the end of a transaction.
- C and C++ designs must use one of the following interfaces on each output port. (These are the only interfaces which provide a data valid signal, required to capture the output data):
    - ap_vld
    - ap_ovld
    - ap_hs
    - ap_memory
    - ap_fifo
    - ap_bus

Since SystemC designs do not use interface synthesis, there are no such requirements for SystemC designs.

Unsupported Optimizations

The automatic RTL verification does not support cases where multiple transformations have been performed upon arrays or arrays within structs on the interface.

In order for automatic verification to be performed, arrays on the function interface, or array inside structs on the function interface, can use any of the following optimizations, but not two or more:

- Reshape.
- Partition.
- Data Pack on structs elements.

Test Bench Requirements

To verify the RTL design produces the same results as the original C code, the test bench used to execute the verification should be self-checking. The important features of a self-checking test bench are discussed in the following example:

```
int main () {
  int ret=0;
  …
  // Execute (DUT) Function
  …

  // Write the output results to a file
  …

  // Check the results
  ret = system("diff --brief  -w output.dat output.golden.dat");

  if (ret != 0) {
        printf("Test failed  !!!\n");
        ret=1;
  } else {
        printf("Test passed !\n");
  }
  …
  return ret;
}
```

- Write the output from the function to a file.
- Compare the results to some existing know good (or golden) results.
- If the results are correct, return the value 0.
- If the results are incorrect, return a non-zero value.
    - Any value can be returned. A sophisticated test bench may return different values depending on the type of difference/failure.

A test bench such as the one shown above provides a substantial productivity improvement by automatically checking the results, freeing the user from manually verifying them.

If a zero is returned by the top-level test bench function `main()`, AutoESL will issue a message stating the verification was successful.

```
@I [SIM-1] *** AutoSim finished: PASS ***
```

Note: If the test bench returns a value of zero, but does not self-check the RTL results and confirm the results are indeed correct, AutoESL will still issue message SIM-1 (as above) indicating the simulation test passed: when no results have actually been checked.

Ensure the test bench checks the results against the expected behavior.

If any non-zero value is returned, AutoESL will issue two messages: one stating the value returned and one stating the RTL verification failed.

```
@E [SIM-3] Simulation failed: test bench return error code "20".
```

```
@E [SIM-1] *** AutoSim finished: FAIL ***
```

Note: A return of "20" also means there was no return value in the test bench: the test bench should be enhanced to self-check the results and return a value of zero if they are correct.

Debugging a Simulation Mismatch

If the test bench is self-checking and shows the results from the RTL to be different from the C code, the following methodology can be used to debug the differences and confirm this is a bug in the RTL:

1. Confirm the result from the C validation step was not created by from a `double` or `float` type. When comparing the results of `double` or `float` types, the test bench must be smart enough to compare in ranges and not in absolute values, since associatively optimizations (the order of the operations) can vary the results depending on the level of optimization applied in the C compilation and synthesis.
2. If required, modify the test bench to print at which sample/cycle the difference is first observed.
3. When executing the simulation, as shown below, select the `Dump Trace` option to create a VCD and view the output waveforms in a 3rd-party tool which can open VCD files (e.g. ModelSim, Verdi, etc.).
4. Try to match up common points in the C and RTL and using the debugger in the CDT and the RTL VCD file, determine when and where the two representations diverge.

The debug steps may also include adding `printf` statements or writing specific results to a file in the C source but the above is the basic methodology for debugging differences.

RTL simulator support

With the above requirements in place, `autosim` can verify the RTL design using any of the valid language and simulator combinations shown in Table 22.

Simulator	OSCI	ModelSim	VCS
SystemC	Supported		
Verilog		Supported	Supported
VHDL		Supported	Supported

Table 22 autosim Simulation Support

When verifying the SystemC RTL output, AutoESL uses the built-in SystemC kernel to verify the RTL. This does not require a license, uses the same version of SystemC used in synthesis and means the RTL design can always be verified using AutoESL.

To verify one of the RTL HDL designs (verilog or VHDL) any of the 3rd-party simulators shown in Table 22 may be used.

For the 3rd party simulators, the executable must be available in the OS search path and the simulator must have a C co-simulation license, since the original C test bench must also be simulated with the design.

RTL Verification

The simulation can be launched from the GUI using the simulation toolbar button.

Figure 70 Tool Bar

This in turn opens the simulation wizard window (Figure 71).

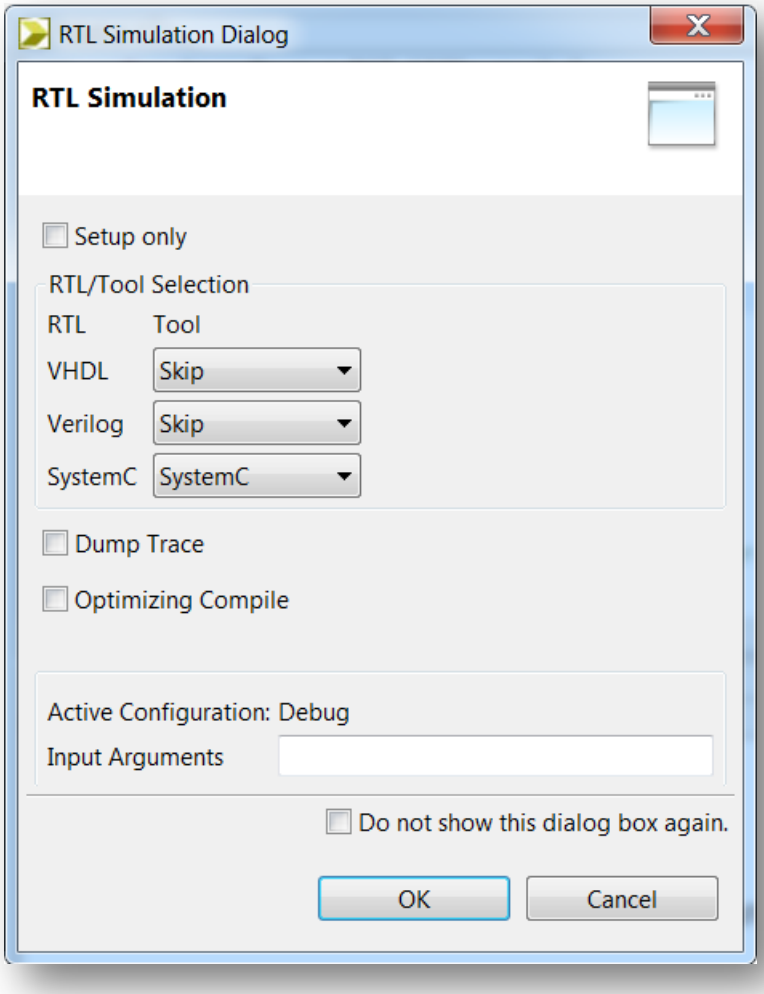

Figure 71 AutoESL simulation wizard

The wizard presents the available RTL languages to simulate (any or all can be simulated). To simulate using a specific language, use the drop-down menu to

select a setting other than "Skip": the drop-down menu will list the simulators supported for that language.

Alternatively the simulator can be run using the AutoESL command line interface using the `autosim` Tcl command:

```
# Simulate VHDL RTL using the ModelSim simulator
autosim -tool modelsim -rtl vhdl

#Simulate systemc RTL using the OSCI simulator
autosim -rtl systemc
```

Once the verification has been executed, the `sim` directory shown in Figure 69 will be populated by the simulation files and the adapters used within test bench: these are not intended for user review but they are not encoded or protected.

If the `Setup Only` option is selected, AutoESL will create the scripts, adapters and wrappers to verify the design but will not execute the simulator. The `Dump Trace` option causes a VCD trace file to be written for each function in the design: these files are written to the appropriate HDL sub-directory of the “sim” directory shown in Figure 69.

The `Optimizing Compiler` option will result in AutoESL using optimized options to compile the C test bench and SystemC adapters: this will result in longer compile time but will improve the run time performance.

The Input Arguments allows the specification of any arguments required by the test bench. The active compile configuration is shown as part of this dialog box.

.

Design Flow Integration

The final step in the AutoESL HLS flow is the implementation phase. AutoESL can create all the scripts, adapters, IP and design files required to process the design through logic synthesis.

This functionality is provided within AutoESL as a convenient way to confirm that the RTL design from AutoESL will satify the timing requirements and conform to the approximate area estimation after the logic synthesis and P&R steps.

The RTL files output by AutoESL should be incorporated into a logic synthesis project and the design processed to FPGA bitstream from there.

> **Note**: When the RTL output from AutoESL is incorporated into a larger RTL logic synthesis project, there may be subtle differences in timing due to placement and routing: the logic synthesis, place and route performed using the implementation step within AutoESL does not include the other RTL blocks in the system.

Implementation can be initiated using the implementation button on the toolbar (Figure 71)

Figure 72 Tool Bar

Or use the `autoimpl` command. When implementation completes, the required files and scripts to process the design through logic synthesis will be present in the "impl" directory as shown in Figure 73.

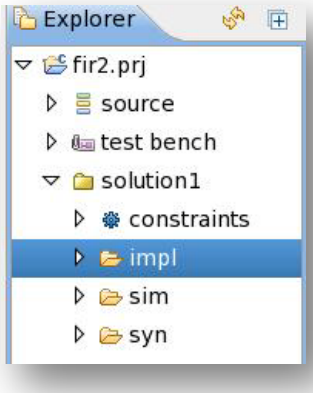

Figure 73 Implementation Directory

Using the `Implementation` button on the toolbar (Figure 72) automatically opens the implementation settings.

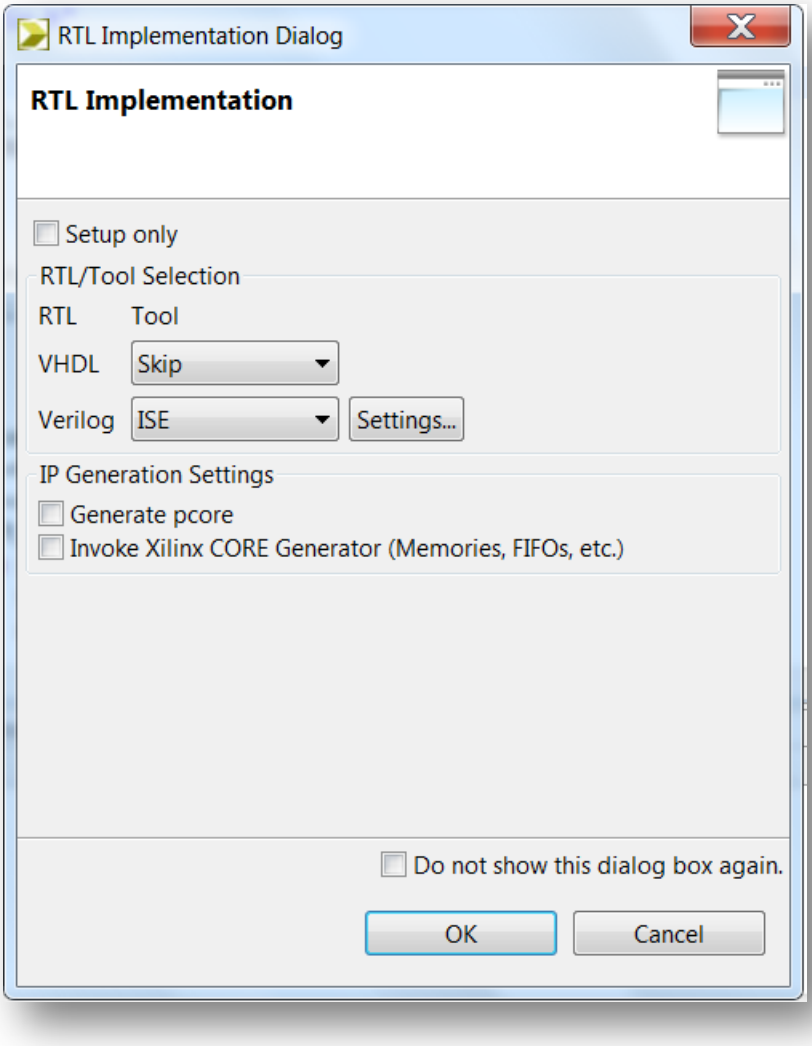

Figure 74 Implementation Settings

If the `Setup Only` option is selected, AutoESL will create scripts, adapters and wrappers to verify the design but will not execute the simulator.

The tool options allow the either the Verilog RTL, VHDL RTL or both to be compiled to gates using either the Xilinx ISE or Synopsys Synplify RTL synthesis tool.

The location of the RTL synthesis tool should be available in the search path.

The `Settings…` options beside the selected RTL synthesis tool can be used to control the RTL synthesis options.

The Generate pcore option creates a pcore implementation of the design which can be imported into the Xilinx EDK environment. More details on this are provided in the next section.

Option Invoke Xilinx CORE Generator is used to force the instantiation of Xilinx CORE generator modules for memory elements. By default, the RTL will contain code the logic synthesis tool will infer as memories. Xilinx ISE may decide to implement these as memory element or it may decide to implement them as registers. This option ensures the memory elements are instantiated into the design as COREs and ISE is forced to implement them as memories.

Once the correct settings have been selected, enter OK to generate the files required to process the design through ISE for logic synthesis.

Manual Execution of RTL Synthesis

If the Setup Only option is selected, the RTL implementation can be launched manually from the CLI interface.

To perform logic synthesis with verilog:

```
$ cd <project>/<solution>/imp/verilog
$ ./impl.sh
```

To perform logic synthesis with VHDL:

```
$ cd <project>/<solution>/imp/vhdl
$ ./impl.sh
```

Generating a Core for EDK

The RTL design created by AutoESL can be exported to the EDK environment as a pcore block. This is accomplished by selecting the `Generate pcore` option as shown in Figure 75.

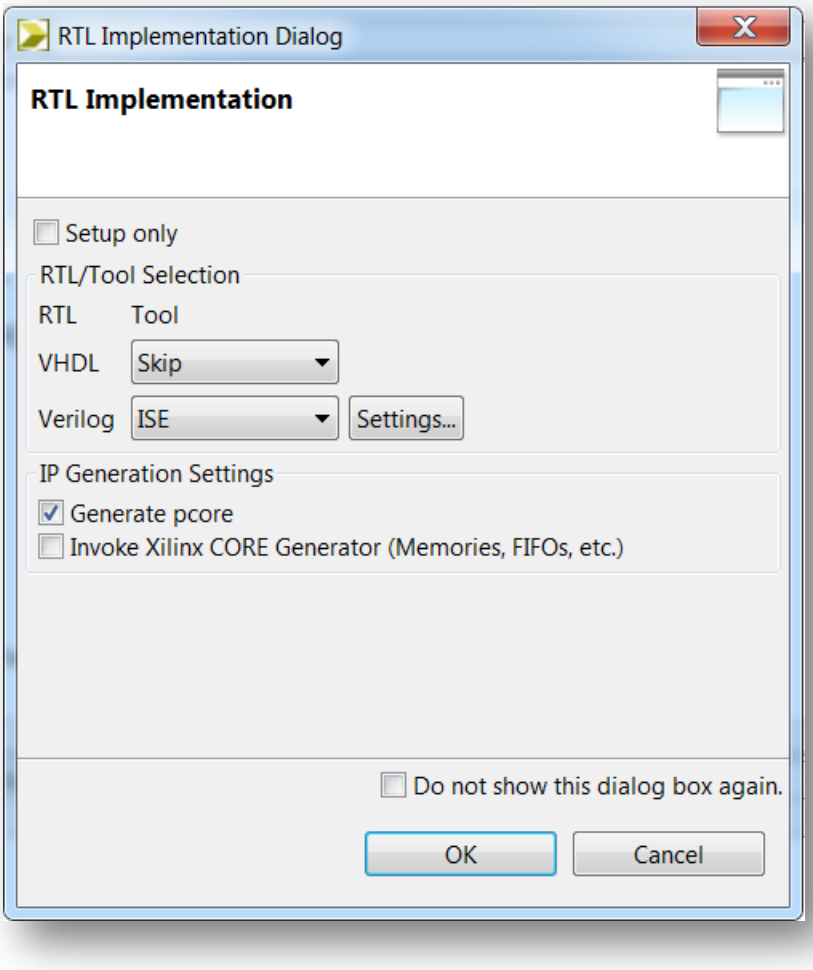

Figure 75 Pcore Creation Settings

The pcore block created by AutoESL automatically consists of a top-level wrapper, in which is placed the RTL design and any specified bus interface adapters to create a pcore as shown in Figure 76. This example shows some of the bus interfaces which can be created between an AutoESL RTL design and the pcore interface (PLB master, AXI4 Stream, NPI, etc.).

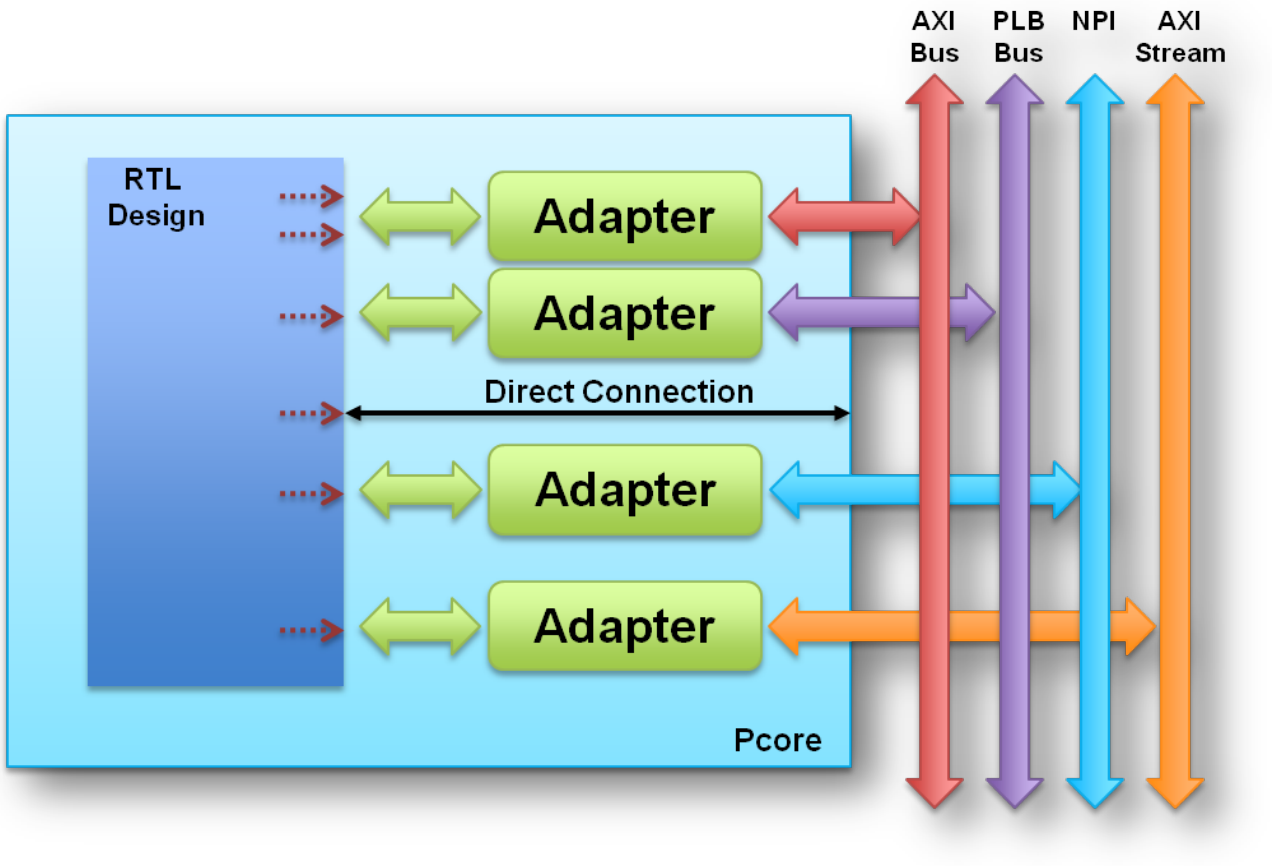

Figure 76 Pcore Block Structure

Overview: Creating a Pcore with AutoESL

The type of bus (or bus adapter) which can be connected to the RTL block depends upon the type of RTL port created by AutoESL.

- Each type of RTL interface can only be connected to certain bus adapters.
- Table 23 shows a list of the RTL interface ports AutoESL creates and bus adapters which can be connected to them.

RTL Interface Type	Bus Adapters						
	PLB 4.6 (master)	PLB 4.6 (slave)	NPI	FSL	AXI4 (Lite Slave)	AXI4 (Stream)	AXI4 (Master)
ap_bus	X	-	X	-	-	-	X
ap_fifo	-	-	-	X	-	X	-
ap_ctrl_hs ap_none ap_vld ap_ack ap_hs	-	X	-	-	X	-	-
ap_ovld ap_memory	-	-	-	-	-	-	-

Table 23 AutoESL Port to Pcore Adapter Mappings

For example, if the port on the RTL design is of type `ap_fifo`, it can be connected to an FSL or AXI4_stream bus adapter and hence to an FSL or AXI4_stream interface. If on the other hand, the RTL port is of type `ap_ovld` or `ap_memory`, it cannot be connected to any bus adapter:

- An ap_memory interface does not require an adapter and can be directly connected to memories (BRAM) in EDK.
- Any port with `ap_ovld` interface should be modified to be one of the supported types, for example `ap_hs`, or it cannot be connected to an adapter.

Any port not connected to a bus adapter, defaults to a Direct Connection interface where the RTL port is simply connected to an identical port on the pcore interface by a wire.

Specifying a Pcore Bus Interface

To connect an RTL interface to a bus adapter, and hence to the appropriate bus interface, an interface resource is specified on the port using the RESOURCE directive. (The same process and directive is used to specify which memory resource is connected to an `ap_memory` port).

A complete list of all bus interface adapters is provided in the “AutoESL Library Guide” and is shown in Table 24.

Bus Adapter	Core	Description
PLB 4.6 (master)	PLB46M	Standard bus-master interface.
PLB 4.6 (Slave)	PLB46S	Standard bus-slave interface.
NPI	NPI64M	Native multi-port memory controller interface.
FSL	FSL	Standard Xilinx FSL interface.
AXI4 (Lite Slave)	AXI4LiteS	AXI4 slave interface
AXI4 (Stream)	AXI4Stream	AXI4 Steam interface
AXI4 (master)	AXI4M	AXI4 Master interface

Table 24 Bus Interface Adapters

Figure 77 shows how a port of type `ap_bus` can be mapped to an NPI interface, allowing the port to communicate with a Multi-Port Memory Controller (MPMC) and hence communicate with a DDR2/DDR3 memory interface.

- The port ("out") must first be synthesized to have an `ap_bus` interface, as shown by the INTERFACE `ap_bus` directive in the directives pane.
- The RESOURCE directive is then used to specify that the port resource is an NPI bus adapter (resource `NPI64M`).
- This will allow the port "out" to be connected to an NPI bus in EDK.

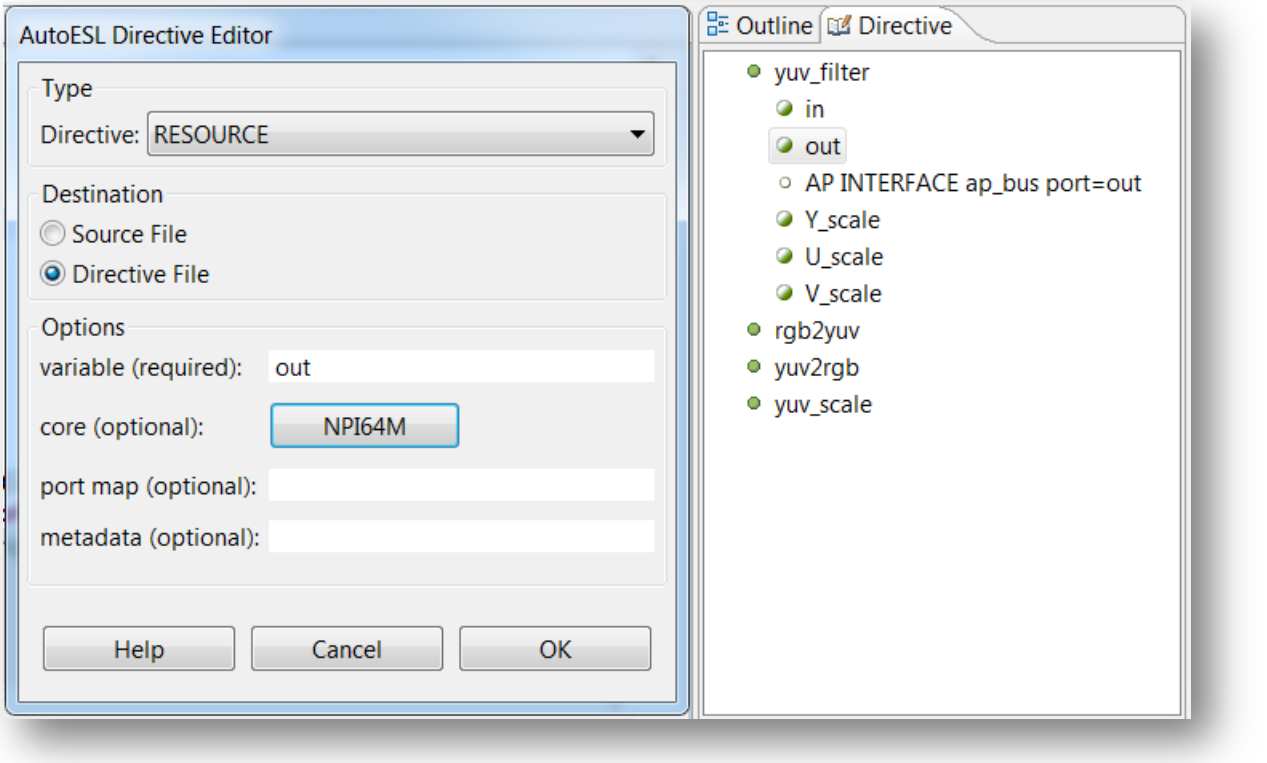

Figure 77 Using RESOURCE directive to specify a pcore adapter

The entire process is summarized by the following short example. If an AXI4-Slave interface is required:

- The port must be synthesized as type which can connect to an AXI4-Slave port.
  - From Table 23 these are interfaces: `ap_ctrl_hs`, `ap_none`, `ap_vld`, `ap_ack` or `ap_hs`.
  - This means the C code must be of a suitable type which can be synthesized as one of these ports (Table 18).
- The interface resource should then be explicitly specified as AXI4LiteS, from Table 24.

If the `Generate pcore` option has been selected (Figure 75), AutoESL will create a "pcores" directory inside the implementation directory, as shown in Figure 78 below.

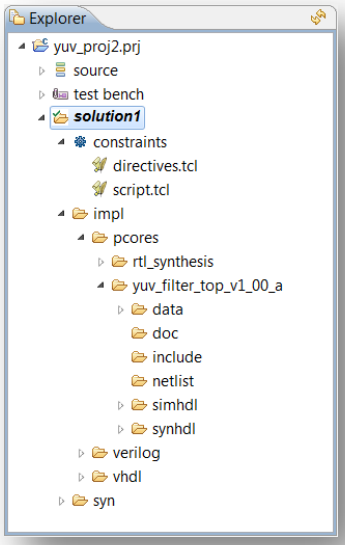

Figure 78 The Pcore Directory

This directory can be copied over to the EDK environment and imported by using the `Add IP` option. The part can then be connected up inside EDK.

Bus Interfaces Examples

This section shows an example of how to create each type of bus interface that can be realized with a pcore.

Each example is shown with example code and using pragmas to specify both the RTL interface (using the INTERFACE directive) and the bus adapter (using the RESOURCE directive): this is done to keep everything required to create the interface under discussion, in the same example file.

These commands can also be applied as Tcl commands in a batch file or using the GUI. Figure 79 shows an RTL interface can be applied using the GUI and Figure 79 shows how the bus interface type can be specified.

Selecting an RTL Interface

In Figure 79, function argument “out” is selected and an INTERFACE directive is applied by right-clicking with the mouse. After selecting INTERFACE from the drop-down menu, the interface mode is specified as ap_bus.

The other options in the directive dialog allow the port to be optionally registered and the required depth to be specified for RTL verification (if the port is a pointer which is accessed multiple times, the number of accesses must be specified here).

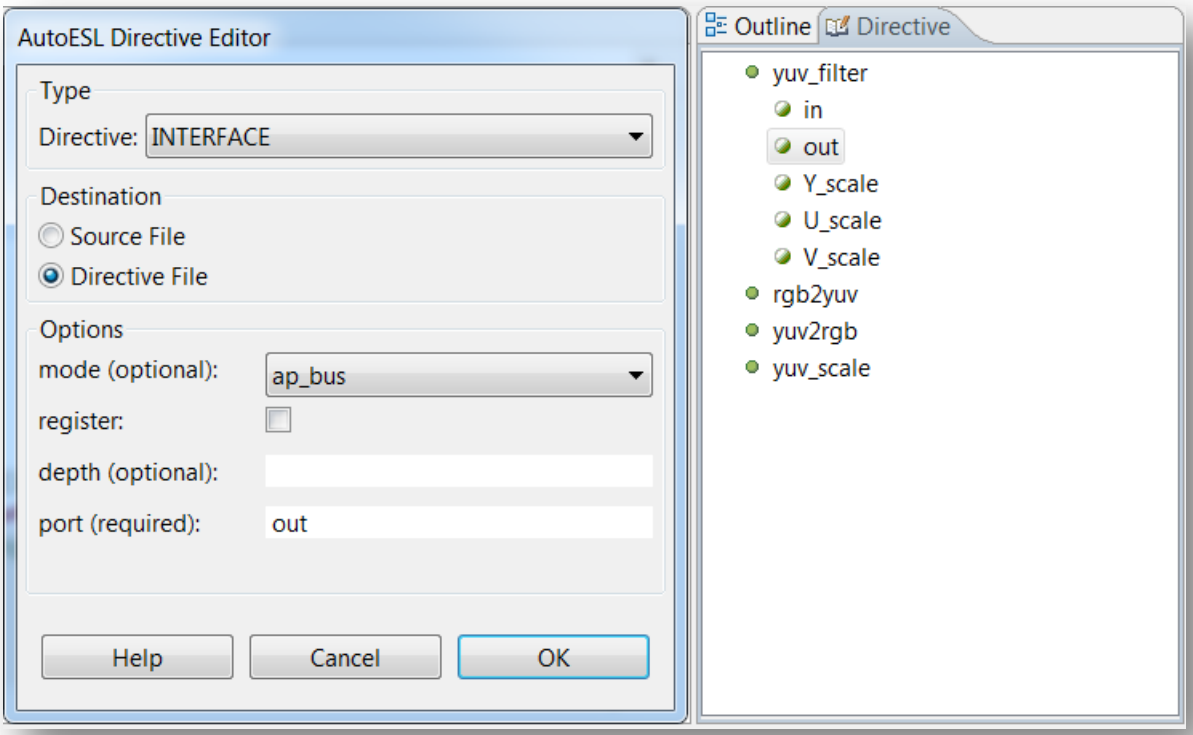

Figure 79 RTL Interface Directive

Selecting a bus adapter

An AXI4 master bus adapter is shown being applied to argument "out" in Figure 80.

The port option in the RESOURCE dialog is used with the AXI4 stream interface to map the RTL port names with the bus adapter names. The usage is shown in the AXI4 Stream example below.

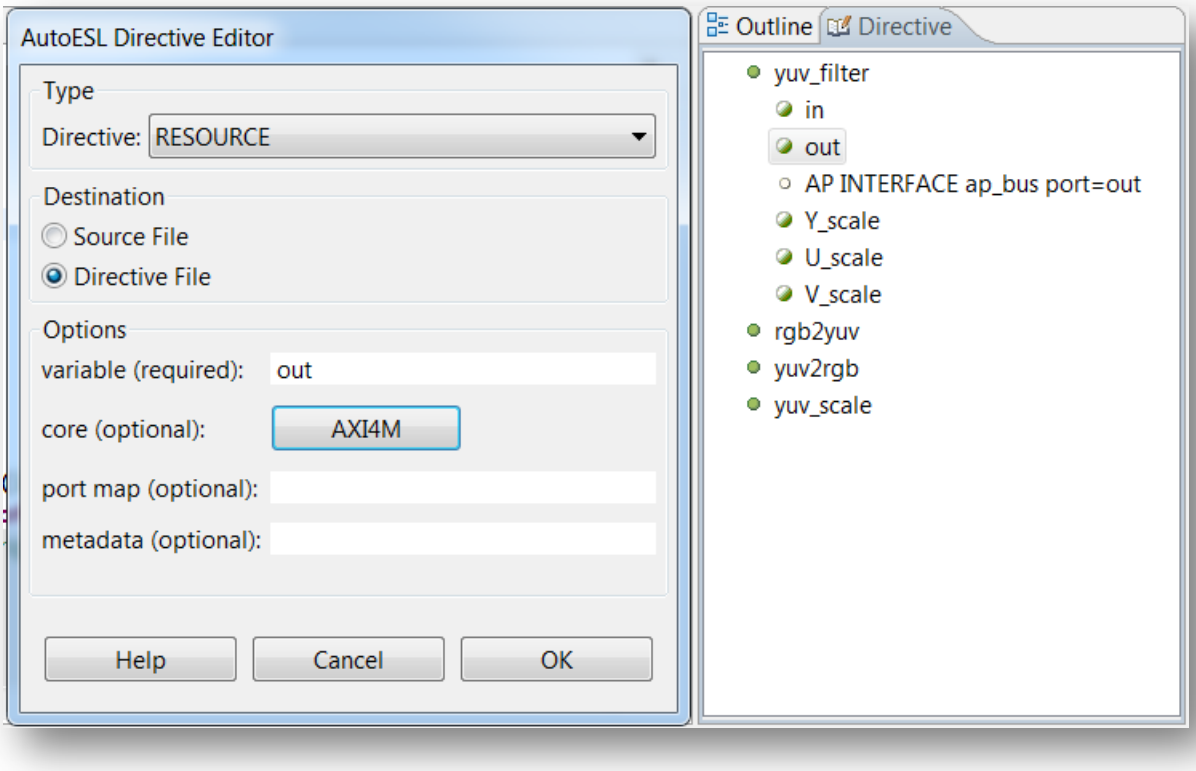

Figure 80 Bus Adapter Interface Directive

The metadata option is used with the AXI stream, AXI slave and PLB slave interfaces. This option allows multiple RTL ports to be grouped, or bundled, in to common bus.

In general, each RTL port is connected to the pcore interface via a specific bus adapter. Bus bundles allow multiple RTL ports to connect via the same adapters, shown as "Bundle Adapter" in Figure 81. This option is often used to bundle the block-level protocol ports (`ap_start`, `ap_done` etc) into a single slave adapter.

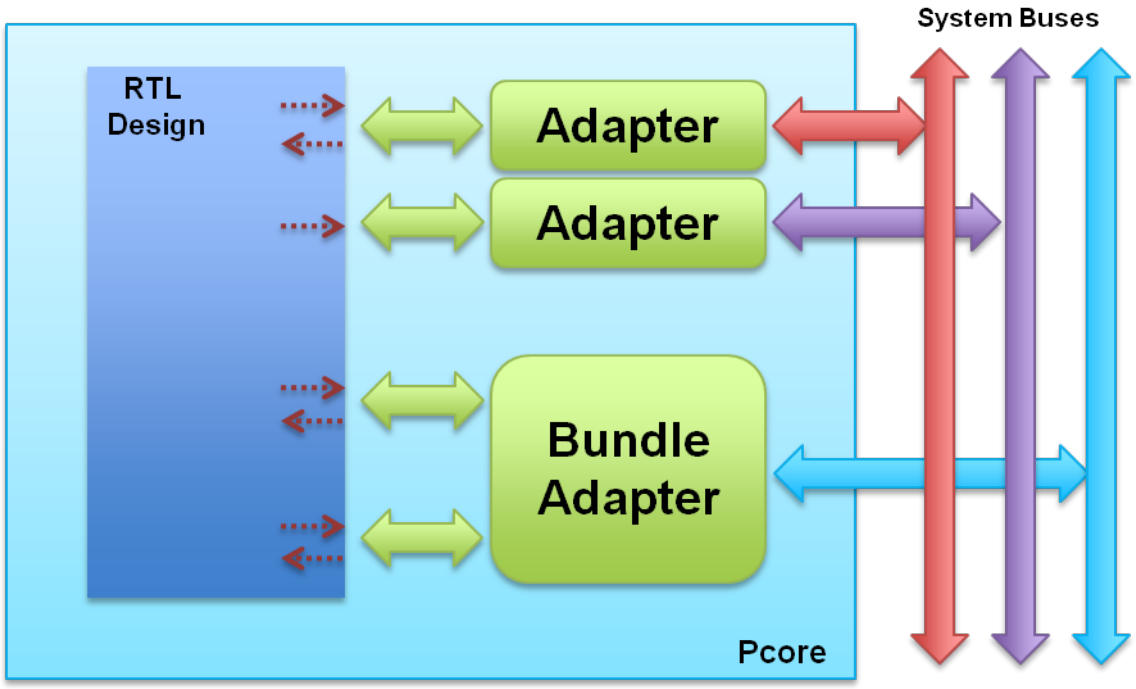

Figure 81 Pcore: Bus Bundles

Direct Connection Interface

If no adapter is assigned to an interface, it will be connected directly to the ports on the pcore. This is default for all interfaces.

Example

In this example, the RTL interface is defined for each port but there are no bus adapters assigned.

```
int func (int a, int b) {

// Define the RTL interfaces
#pragma AP interface ap_hs port=a
#pragma AP interface ap_hs port=b
#pragma AP interface ap_ctrl_hs port=return register

    return (a + b);
}
```

If the `Generate pcore` option is selected, this results in a pcore where all the ports have direct connection interfaces, as shown in Figure 82.

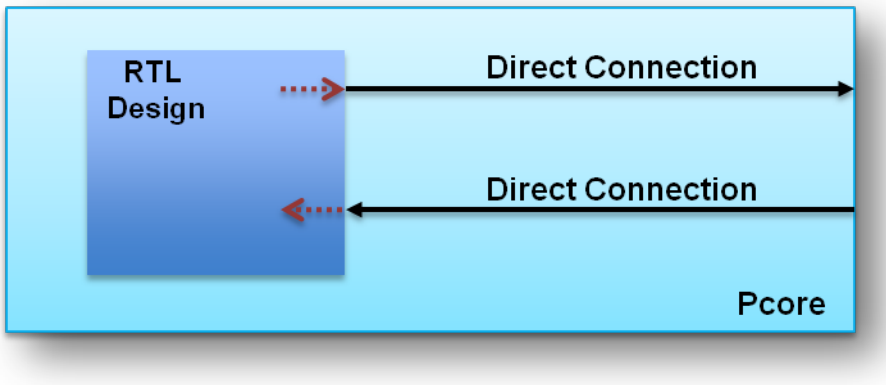

Figure 82 Pcore: Direct Connection Interface

PLB Master Interface

To create a PLB master interface, the RTL ports must be an `ap_bus` interface, as shown in Table 23. This example sets the port "m" as an `ap_bus` and then specifies that the port resource is a "PLB46M" resource.

Example

The block-level interface protocol is removed in this example by setting the IO protocol to `ap_ctrl_none`. This ensures there are no other interface signals in this design (no block-level handshakes signals and there is no return port).

```
#include "ap_cint.h"

#define N 256

typedef uint32 DT;
void func (volatile DT *m) {

// Define the RTL interfaces
#pragma AP interface ap_ctrl_none port=return
#pragma AP interface ap_bus port=m

// Define the pcore interface as a PLB master
#pragma AP resource core=PLB46M variable=m

    DT buff[N], tmp;
    int i, j;
    memcpy(buff, m, N * sizeof(DT));
    for (i = 0, j = N - 1; i < j; i++, j--) {
        tmp = buff[i];
        buff[i] = buff[j];
        buff[j] = tmp;
    }
    memcpy(m, buff, N * sizeof(DT));
}
```

This specification results in a PLB v4.6 master adapter and interface on the pcore. Inside the EDK environment, this port can be connected to PLB bus and will act as a master interface.

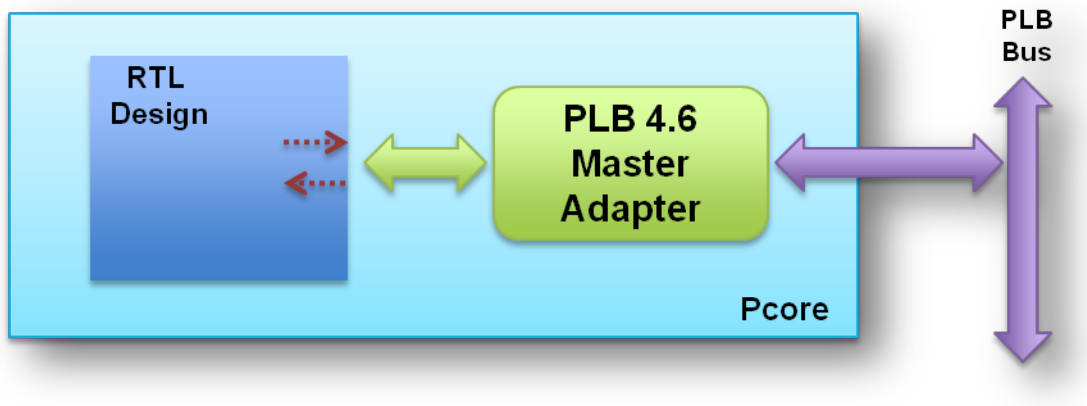

Figure 83 Pcore: PLB Master Interface

AXI4 master and NPI adapters are similar in structure and the number of ports to PLB master adapter.

AXI4 Master Interface

To create an AXI4 master interface, the RTL ports must be an `ap_bus` interface, as shown in Table 23. This example sets the port "m" as an `ap_bus` and then specifies that the port connect to an "AXI4M" resource.

Example

The block-level interface protocol is removed in this example by setting the IO protocol to `ap_ctrl_none`. This ensures there are no other interface signals in this design (no block-level handshakes signals and there is no return port).

```
#include "ap_cint.h"

#define N 256
typedef uint32 DT;

void func (volatile DT *m) {

// Define the RTL interfaces
#pragma AP interface ap_ctrl_none port=return
#pragma AP interface ap_bus port=m

// Define the pcore interface as an AXI4 master
#pragma AP resource core=AXI4M variable=m

    DT buff[N], tmp;
    int i, j;
    memcpy(buff, m, N * sizeof(DT));
    for (i = 0, j = N - 1; i < j; i++, j--) {
```

```
        tmp = buff[i];
        buff[i] = buff[j];
        buff[j] = tmp;
    }
    memcpy(m, buff, N * sizeof(DT));
}
```

When the pcore is generated, an AXI4 master interface will be connected to port "m" and this can connect to an AXI4 system bus, as shown in Figure 84.

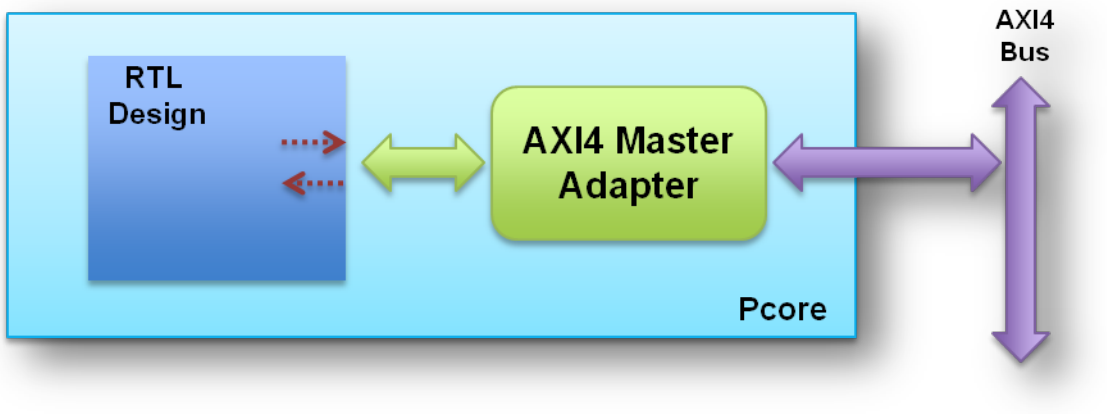

Figure 84 Pcore: AXI4 Master Interface

NPI Interface

An NPI interface is created by first creating an ap_bus on the RTL interface. This interface can then be assigned a "NPI64M" resource as shown in the following example.

Example

```
#include "ap_cint.h"

#define N 256
typedef uint32 DT;

void func (volatile DT *m) {

// Define the RTL interfaces
#pragma AP interface ap_ctrl_none port=return
#pragma AP interface ap_bus port=m

// Define the pcore interface as an NPI master
#pragma AP resource core=NPI64M variable=m

    DT buff[N], tmp;
    int i, j;
    memcpy(buff, m, N * sizeof(DT));
    for (i = 0, j = N - 1; i < j; i++, j--) {
        tmp = buff[i];
        buff[i] = buff[j];
```

```
        buff[j] = tmp;
    }
    memcpy(m, buff, N * sizeof(DT));
}
```

In this example, the block-level interface protocol is removed by setting the IO protocol to `ap_ctrl_none`. This ensures there are no other interface signals in this design (no block-level handshakes signals and there is no return port).

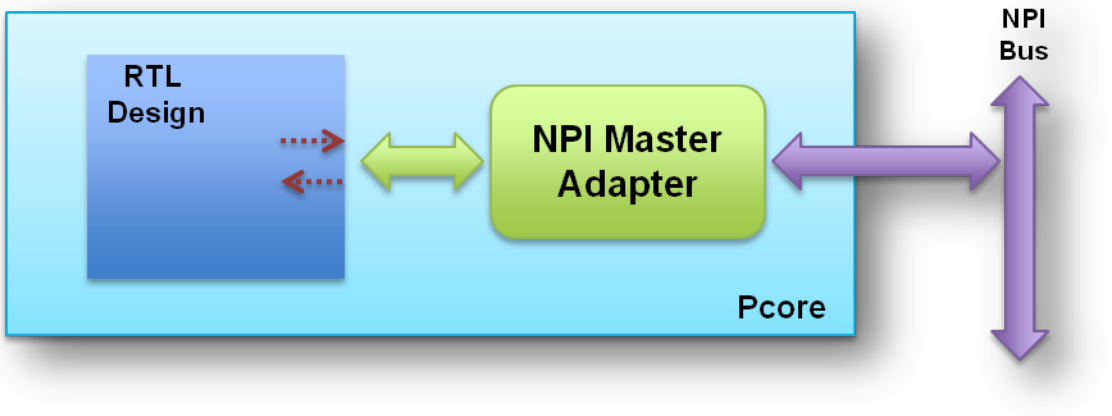

Figure 85 Pcore: NPI Master Interface

FSL Interface

An FSL interface can be connected to an RTL `ap_fifo` interface. The FSL interface is a master/slave interface.

- If the interface is an input, a FSL slave adapter is generated.
- If it is an output, a FSL master adapter is generated.
- FSL adapters cannot be connected to bi-directional ports (an `ap_fifo` interfaces cannot be applied to these ports).

Example

```
#define N 256

void func (int data_i[N], int data_o[N]) {

// Define the RTL interfaces
#pragma AP interface ap_ctrl_none port=return
#pragma AP interface ap_fifo port=data_i
#pragma AP interface ap_fifo port=data_o

// Define the pcore interfaces as FSL types
#pragma AP resource core=FSL variable=data_i
#pragma AP resource core=FSL variable=data_o
```

```
    int buff[N], i;
    for (i = 0; i < N; i++) {
        buff[i] = data_i[i];
    }
    for (i = 0; i < N; i++) {
        data_o[i] = buff[N-1-i];
    }
}
```

In this example, there are two ap_fifo interfaces. One is an input and the other an output. This examples set the block-level protocol to ap_ctrl_none to ensure there are no other ports (there is no function return and the ap_ctrl_none ensures there are no block-level handshake signals).

The input and output FSL interfaces are shown in Figure 86.

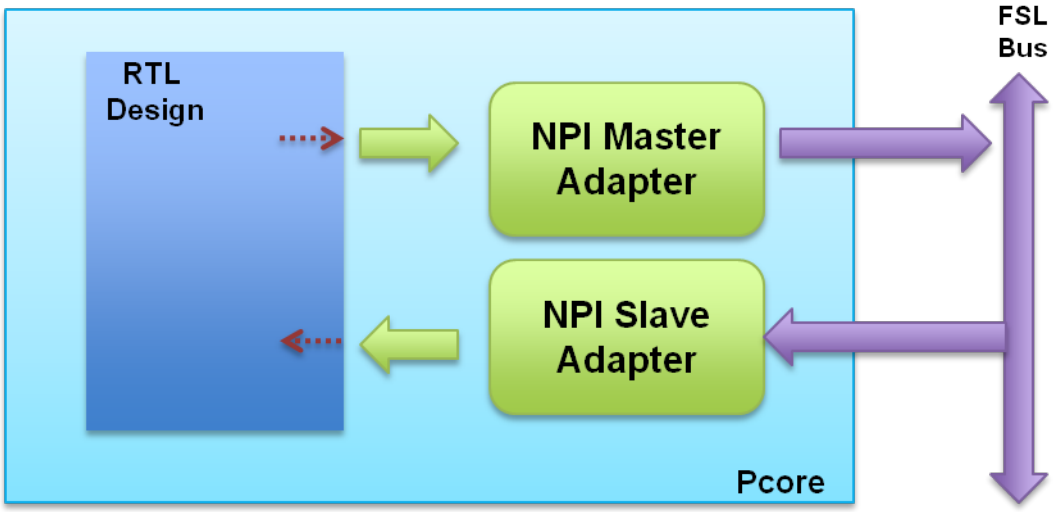

Figure 86 Pcore: FSL Master & Slave Interface

PLB Slave Interface

The following example shows how multiple RTL ports are bundled into a common PLB slave interface. This allows multiple RTL ports to be accessed through a single bus interface.

```
int func (int *a, int *b, int *c, int *d) {

// Define the RTL interfaces
#pragma AP interface ap_hs port=a
#pragma AP interface ap_hs port=b
#pragma AP interface ap_hs port=c
#pragma AP interface ap_hs port=d
#pragma AP interface ap_ctrl_hs port=return register

// Define the pcore interfaces and group into PLB slave “slv0”
#pragma AP resource core=PLB_SLAVE metadata="-bus_bundle slv0" variable=a
```

```
#pragma AP resource core=PLB_SLAVE metadata="-bus_bundle slv0" variable=b

// Define the pcore interfaces and group into PLB slave “slv1”
#pragma AP resource core=PLB_SLAVE metadata="-bus_bundle slv1" variable=c
#pragma AP resource core=PLB_SLAVE metadata="-bus_bundle slv1" variable=d
#pragma AP resource core=PLB_SLAVE metadata="-bus_bundle slv1" variable=return

    *a += *b;
    return (*c + *d);
}
```

In the example above, ports “a” and “b” are grouped into a common PLB slave interface as shown in Figure 87.

The block-level IO protocol is explicitly set by applying interface mode `ap_ctrl_hs` to the function `return`. (This is the default but shown explicitly in this example). All block-level IO interface ports (`ap_start`, `ap_done`, etc) are assigned to a PLB slave interface as a single group along with the function output (the `ap_return` port).

Ports “c” and “d” and grouped with the block-level IO protocol signals and function `return` into group “slv1” as shown in Figure 87.

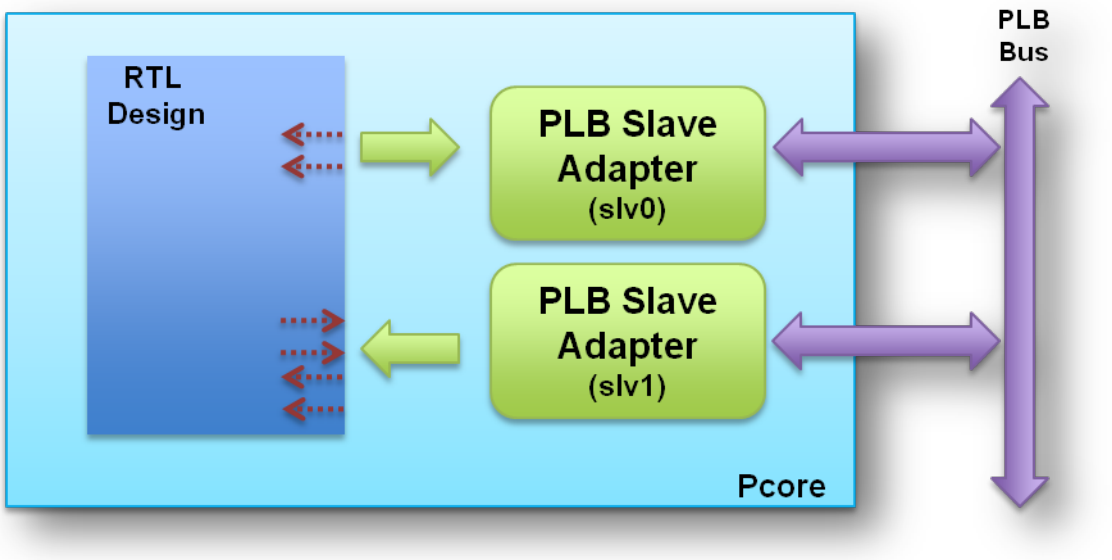

Figure 87 Pcore: PLB Slave interfaces with Bundle Ports

If the RESOURCE directive is applied using the GUI, the entire bus bundle string (e.g. "-bus_bundle slv1") including the quotes should be entered into the metadata option box.

PLB slave interfaces are created with an accompanying header file to define the offset address of the various RTL ports associated with each RTL interface. A header file is created for each slave adapter for use in EDK.

Bus interface slv0 is associated with header file “slv0_addrmap.h”. Since argument “a” is both read from and written to it has both input and output RTL ports. The contents “slv0_addrmap.h” are:

```
// a_i
#define a_i_BASEADDR        0x0000
#define a_i_HIGHADDR        0x0000
#define a_i_ap_vld_BASEADDR 0x0004
#define a_i_ap_ack_BASEADDR 0x0008

// a_o
#define a_o_BASEADDR        0x000c
#define a_o_HIGHADDR        0x000c
#define a_o_ap_vld_BASEADDR 0x0010
#define a_o_ap_ack_BASEADDR 0x0014

// b
#define b_BASEADDR          0x0018
#define b_HIGHADDR          0x0018
#define b_ap_vld_BASEADDR   0x001c
#define b_ap_ack_BASEADDR   0x0020
```

Similarly, bus adapter slv1 has header file “slv1_addrmap.h”:

```
// c
#define c_BASEADDR         0x0000
#define c_HIGHADDR         0x0000
#define c_ap_vld_BASEADDR  0x0004
#define c_ap_ack_BASEADDR  0x0008

// d
#define d_BASEADDR         0x000c
#define d_HIGHADDR         0x000c
#define d_ap_vld_BASEADDR  0x0010
#define d_ap_ack_BASEADDR  0x0014

// ap_start
#define ap_start_BASEADDR  0x0018

// ap_done
#define ap_done_BASEADDR   0x001c

// ap_idle
#define ap_idle_BASEADDR   0x0020

// ap_return
#define ap_return_BASEADDR 0x0024
#define ap_return_HIGHADDR 0x0024
```

AXI4 Slave Interface

An AXI4 slave interface is implemented as an AXI4 Lite slave interface. The following example shows how multiple RTL ports are bundled into a common AXI4

slave interface. This allows multiple RTL ports to be accessed through a single bus interface.

This example is nearly the same as PLB_SLAVE example except that AXI4LiteS is used instead of PLB_SLAVE.

```
int func (int *a, int *b, int *c, int *d) {

// Define the RTL interfaces
#pragma AP interface ap_hs port=a
#pragma AP interface ap_hs port=b
#pragma AP interface ap_hs port=c
#pragma AP interface ap_hs port=d
#pragma AP interface ap_ctrl_hs port=return register

// Define the pcore interfaces and group into AXI4 slave “slv0”
#pragma AP resource core=AXI4LiteS metadata="-bus_bundle slv0" variable=a
#pragma AP resource core=AXI4LiteS metadata="-bus_bundle slv0" variable=b

// Define the pcore interfaces and group into AXI4 slave “slv1”
#pragma AP resource core=AXI4LiteS metadata="-bus_bundle slv1" variable=c
#pragma AP resource core=AXI4LiteS metadata="-bus_bundle slv1" variable=d
#pragma AP resource core=AXI4LiteS metadata="-bus_bundle slv1" variable=return

    *a += *b;
    return (*c + *d);
}
```

In the example above, ports “a” and “b” are grouped into a common AXI4 slave interface as shown in Figure 88.

The block-level IO protocol is explicitly set by applying interface mode `ap_ctrl_hs` to the function `return`. (This is the default but shown explicitly in this example). All block-level IO interface ports (`ap_start`, `ap_done`, etc) are assigned to an AXI4 slave interface as a single group along with the function output (the `ap_return` port).

Ports “c” and “d” and grouped with the block-level IO protocol signals and function `return` into group “slv1” as shown in Figure 88.

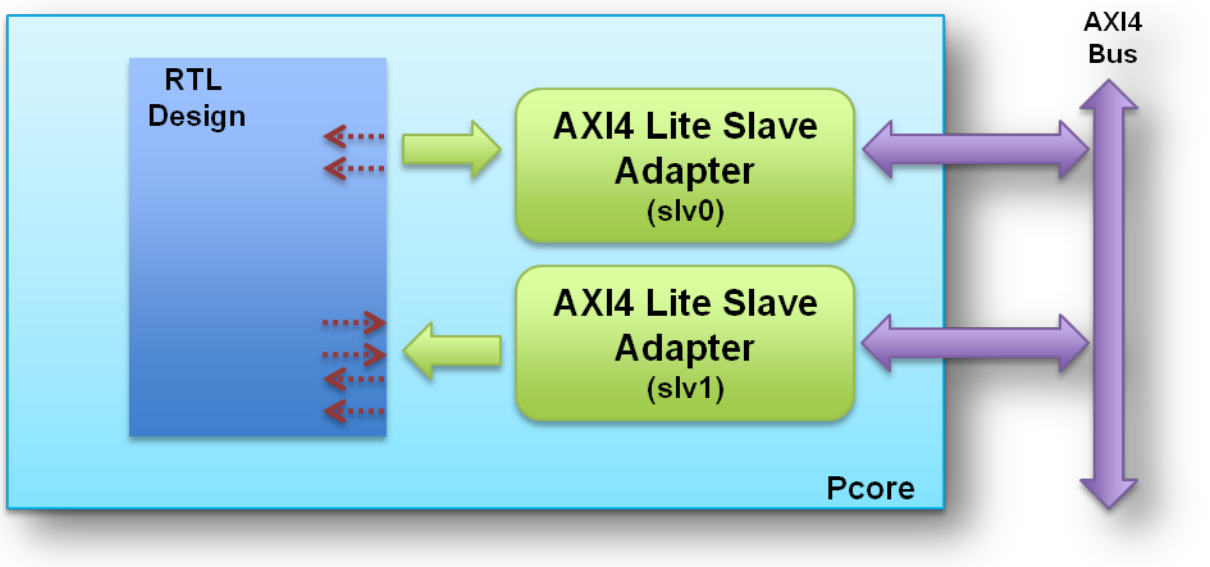

Figure 88 Pcore: AXI4 lite Slave interfaces with Bundle Ports

If the RESOURCE directive is applied using the GUI, the entire bus bundle string (e.g. "-bus_bundle slv1") including the quotes should be entered into the metadata option box.

AXI4 Stream Interface

The AXI4 stream interface is an AXI4-Stream master/slave adapter. This interface can be applied to any `ap_fifo` RTL. AXI4 stream interface can be bundled like PLB and AXI4 slave interfaces.

In an AXI4 stream interface, the RTL Interfaces in the same bundle must be all input or all output. Bidirectional interface are not supported. Output adapters implement a master interface and input adapters a slave interface.

The following example shows how to apply and configure an AXI4 stream interface. In this example, the block-level interface protocol is removed by setting the IO protocol to `ap_ctrl_none`. This ensures there are no other interface signals in this design (no block-level handshakes signals and there is no return port).

```
#include "ap_cint.h"

#define N 256

typedef struct {
    uint32 data;
    uint4  strb;
} DATA;

void func (DATA data_i[N], DATA data_o[N]) {

// Define the RTL interfaces
#pragma AP interface ap_ctrl_none port=return
```

```
#pragma AP interface ap_fifo port=data_i
#pragma AP interface ap_fifo port=data_o

// Define the pcore interfaces AXI4 slave
#pragma AP resource core=AXI4Stream variable=data_i metadata="-bus_bundle
AXI4Stream_S" port_map={{data_i_data TDATA} {data_i_strb TSTRB}}

// Define the pcore interfaces AXI4 master
#pragma AP resource core=AXI4Stream variable=data_o metadata="-bus_bundle
AXI4Stream_M" port_map={{data_o_data TDATA} {data_o_strb TSTRB}}

    int i;
    DATA buf[N];

    for (i = 0; i < N; i++) {
        buf[i] = data_i[i];
    }

    for (i = 0; i < N; i++) {
        data_o[i] = buf[N-1-i];
    }

}
```

The first thing to note about this example is that when an RTL interface is applied to a struct, the RTL interface is applied to all members of the struct.

For an AXI4 stream interface the bus bundle and port map options are mandatory.

When applying a stream interface it is recommended to define a struct whose members correspond to the desired AXI4 stream bus signals. In the example above, struct DATA is defined and contains two members, data and strb. Each member creates a separate RTL interface port and the bus bundle option bundles all ports into the same AXI4 stream adapter.

In an AXI4 stream interface, most signals, (e.g. TREADY, TKEEP, TSTRB) are optional. The AXI4 stream interface provided by AutoESL is designed to fit the flexibility of the bus standard. Only four signals, ACLK, ARESETN, TVALID and TREADY are mandatory (TREADY is optional in the bus standard, but it is used in the AutoESL AXI4 stream interface).

All other signals are optional but if present must be mapped to an RTL interface using the port_map option. In the above example the bundle data signals are mapped to AXI4 stream interface signal TDATA and the strobes signals mapped to TSTRB, for example:

```
port_map={{data_o_data TDATA} {data_o_strb TSTRB}}
```

To generate an AXI4-Stream adapter, all RTL interfaces in the bundle must be mapped with the port_map option to an AXI stream signal. There is no default mapping.

If the port mappings are applied using the GUI, each port map should be enclosed by braces {RTL_PORT AXI4_INTERFACE_SIGNAL} and all maps enclosed by an outer set of braces. The entire string must be entered into the port_map option box. For example:

```
{{data_o_data TDATA} {data_o_strb TSTRB}}
```

Figure 89 shows the pcore created from the example shown above, with RTL inputs and outputs bundled into separate adapters.

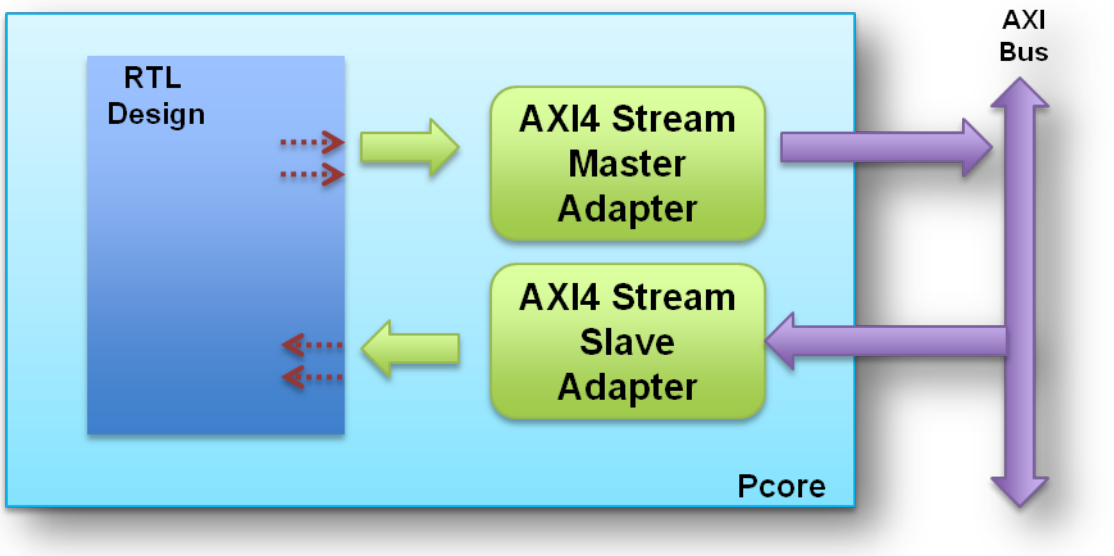

Figure 89 Pcore: AXI4 Stream interfaces