AutoESL Reference Guide

UG868 (v2012.1) April 24, 2012

XILINX®

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
04/24/12	2012.1	Initial Xilinx release.

XILINX®

Table of Contents

Using AutoESL Commands

Before using any AutoESL commands in an active design project it is worthwhile to understand and appreciate a few basic AutoESL concepts.

1. AutoESL stores design in a project based structure.
2. AutoESL optimizations are specified on regions, or locations, within the code.
3. AutoESL optimizations can be specified as Tcl commands or pragmas in the source code (and in addition to editing text files, both options can be performed from within the AutoESL GUI).

Each of these concepts is fully explained in the remaining sections of this chapter.

Managing Projects

AutoESL uses a project based database to manage the synthesis and verification processes and to store the results. Every design must be represented in a project which may itself contain multiple solutions.

The source code and testbench are stored in the project. Solutions can be used to specify a different target technology (different FPGA families and devices), apply different directives and create different implementations of the same source code, with a view to selecting the most optimum implementation.

AutoESL projects and solutions are directly reflected in the directory structure used by AutoESL. Figure 1-1 shows an example AutoESL directory after results have been generated.

In the example in Figure 1-1:

- The project, as shown on the top line, is called `project.prj` and all project data will be stored in a project directory of the same name.
- The project contains source code and testbench files.
- There are 2 solutions in this project (`solution1` and `solution2`).
- Currently **`solution1`** (highlighted in bold) is the active solution.
- Simulation results have been generated, as shown by the `sim` directory.

- Synthesis results have been generated, as shown by the `syn` directory.
  - The `syn` directory shows RTL output has been created in verilog, VHDL and SystemC.
  - The `syn` directory shows reports have been generated.

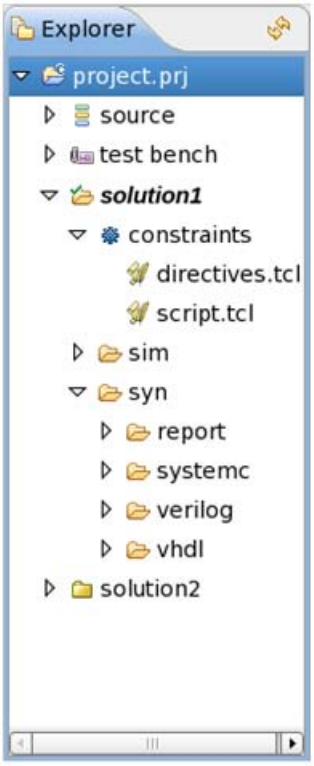

Figure 1-1: **Project and Solution Structure**

When using AutoESL commands, it is important to understand that some commands can only be used inside an active project or solution. In general, projects are used for the same set of source code and solutions are used to create different implementations of that source code.

Note, opening a new project automatically closes the existing project and opening a new solution automatically closes the existing solution.

AutoESL Optimization Locations

Within AutoESL optimizations can be specified on functions, loops, regions, arrays and interface parameters. This section explains how optimizations are applied and the locations they affect.

Optimizations are specified in one of three ways:

1. Using the directives tab in the GUI.
2. Tcl commands can be used at the interactive prompt, or with a batch file, to specify directives, provided the object can be uniquely identified (loops and regions require a label).
3. Insert pragmas directives directly into the source code.

The optimizations specified by all of these techniques are applied to the specified location (scope) within the source code.

The following example shows the outline of some source code.

```
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 1-2 shows how this code is represented in the GUI directives tab. The directives tab can be viewed by selecting the source code within the Project Explorer, then selecting the directives tab in the Auxiliary Pane (right hand side of the GUI).

Figure 1-2: **GUI Directives Objects**

1. Interfaces - Directives applied to interfaces are applied to that interface object (function parameter, global or return value) and nothing else.
2. Functions - Directives applied to functions will be performed on all objects within the scope of the function. The effect of any directive will stop at the next level of function hierarchy except in the case of PIPELINE which recursively flattens and unrolls everything, or if the directive supports a recursive option and it is applied.
   a. In the example in Figure 1-2, a directive applied on `foo_top` will not affect the operations in `foo_sub_A` or `foo_sub_B` (other than the exceptions stated above).
3. Loops - Directives applied on loops will impact all objects within the scope of the loop.

a. If a MERGE directive is applied to a loop, the directive will apply to any sub-loops but not to the loop itself. The loop will not be merged with siblings at the same level of hierarchy but any sub-loops will be merged.

b. A PIPELINE directive will also apply to objects within the loop: they will be pipelined, which is essentially the same as pipelining the loop itself.

4. Arrays - Directives can be applied to arrays directly, in which case they only apply to the array itself, or they can be applied to functions, loops or regions which contain multiple arrays, in which case the directive will apply to all arrays enclosed.

5. Regions - Regions of code can be created by inserting a pair of braces: {the code between is a region}. Any directive applied to a labeled region will apply to the objects within that region.

Note: To apply directives using Tcl commands, loops and regions must have a named label as shown in the above example (loop label `for_top` and region label `my_label`)

Other than interfaces and arrays, the other objects (function, loops and regions) represent "areas" of the code which can cover multiple statements. For this reason, the command pages within this manual will list the target as "location". Keep in mind when a directive is applied to a location it is applied to all objects within the location (unless otherwise explicitly specified).

Commands and Pragmas

AutoESL optimizations are primarily based on locations within the code, as discussed in the previous section. This model for specifying optimizations supports the use of pragmas in the code.

In this example, a pragma is inserted into the code to unroll a for-loop. After the pragma, keyword `AP` is always specified and then the directive:

```
for (int i = 0; i < 64; ++i) {
#pragma AP UNROLL
  ...
}
```

Any directives entered in the C code are shared by every implementation which uses the C specification. This could be a desired feature, to have the C specification contain all the optimization directives, however in many cases designers may wish to separate the pragmas from the source to allow different solutions to be created (when pragmas are used, every solution which uses the source will have the same optimizations performed).

In the above example, the command `set_directive_loop_unroll` could have been used at the command line interface provided the loop had a named label or the unroll directive could have been added to the source code using the GUI directives tab.

The following can be viewed using the `help` command within AutoESL and lists the associated pragma for each optimization directive:

```
>autopilot help
...
set_directive_allocation           - Directive ALLOCATION
set_directive_array_map            - Directive ARRAY_MAP
set_directive_array_partition      - Directive ARRAY_PARTITION
set_directive_array_reshape        - Directive ARRAY_RESHAPE
set_directive_array_stream         - Directive ARRAY_STREAM
set_directive_dataflow             - Directive DATAFLOW
set_directive_dependence           - Directive DEPENDENCE
set_directive_expression_balance   - Directive EXPRESSION_BALANCE
set_directive_function_instantiate - Directive FUNCTION_INSTANTIATE
set_directive_inline               - Directive INLINE
set_directive_interface            - Directive INTERFACE
set_directive_latency              - Directive LATENCY
set_directive_loop_flatten         - Directive LOOP_FLATTEN
set_directive_loop_merge           - Directive LOOP_MERGE
set_directive_loop_tripcount       - Directive LOOP_TRIPCOUNT
set_directive_occurrence           - Directive OCCURRENCE
set_directive_pipeline             - Directive PIPELINE
set_directive_platform             - Directive PLATFORM
set_directive_power                - Directive POWER
set_directive_protocol             - Directive PROTOCOL
set_directive_resource             - Directive RESOURCE
set_directive_top                  - Directive TOP
set_directive_unroll               - Directive UNROLL
...
```

The easiest way to determine what can be optimized and what the command or pragma is:

1. Open the code in the GUI.
2. Select the **Directive** tab, as shown in Figure 1-2, which shows all objects which can be optimized.
3. With the object selected, right-click with the mouse to specify a directive.
   a. Select the **Destination** as **Into Directive File** to see the command in `constraints/directive.tcl`.

   OR

   b. Select the **Destination** as **Into Source File** to see the pragma inserted directly into the code.

AutoESL Commands

add_file

Syntax

```
add_file [OPTIONS] <src_files>
```

Description

The `add_file` command adds design source files to the current project. The current directory is automatically searched for any header files included in the design source. To use header files stored in directories other than the current directory, use the **`-cflags`** option to add those directories to the search path.

<*src_files*> - A list of source files with the description of the design.

Options

`-tb` - Specify any files used as part of the design testbench. These files are not synthesized but used when post-synthesis verification is executed by the `autosim` command. No design files can be included in the list of source files when this option is used: use a separate `add_file` command to add design files and testbench files.

`-cflags` <*string*> - A string with any desired GCC compilation options.

`-type` (**`c`** | **`sc`**) - Specify if the source files are a C/C++ (c) or a SystemC (sc). The default is C/C++ source files.

Pragma

There is no pragma equivalent of the `add_file` command.

Examples

This example adds three design files to the project.

```
add_file a.cpp
add_file b.cpp
add_file c.cpp
```

Multiple files can be added with a single command line.

```
add_file "a.cpp b.cpp c.cpp"
```

The following example adds a SystemC file with compiler flags to enable macro USE_RANDOM and specifies an additional search path, sub-directory `./lib_functions`, for header files.

```
add_file -type sc top.cpp -cflags "-DUSE_RANDOM -I./lib_functions"
```

The **`-tb`** option is used to add testbench files to the project. This example adds multiple files with a single command, including the testbench `a_test.cpp` and all data files read by the testbench, `input_stimuli.dat` and `out.gold.dat`.

```
add_file -tb "a_test.cpp input_stimuli.dat out.gold.dat"
```

If the testbench data files in the previous example are stored in a separate directory, for example `test_data`, the directory can be added to the project in place of the individual data files.

```
add_file -tb a_test.cpp
add_file -tb test_data
```

autoimpl

Syntax

```
autoimpl [OPTIONS]
```

Description

Automatically creates and executes the scripts to create a gate-level implementation of the RTL.

You can specify that a specific user-provided script to be used for the implementation in place of the default AutoESL generated script.

The implementation scripts are created in sub-directory `impl/<rtl>` of the active solution. By default, `autoimpl` automatically executes the scripts in this same directory, to produce a gate level implementation. The **-setup** option can be used to create the scripts only: logic synthesis is not invoked.

The **-export** and **-custom_ports** options can be used to create a pcore implementation for use within the EDK environment. The option **-xil_coregen** will call the Core Generator tool to create and instantiate optimized components (such as BRAM, Floating-Point blocks) prior to logic synthesis.

Options

-export - This option creates an additional implementation of the design, with the appropriate wrappers and external adapters, which can be imported into EDK as a core model (can be directly copied to a project pcores directory).

-par - This option can only be used with the **-tool** option when specifying Synopsys Synplify products for RTL synthesis. When this option is specified the place and route implementation will be executed using the ISE Design Suite (otherwise place and route is not performed).

-rtl (**verilog**|**vhdl**) - Determines which HDL is used for the RTL implementation. The default is **verilog**.

-setup - When this option is specified all the implementation files will be created in the `impl/<rtl>` directory of the active solution but the implementation will not be executed.

-tool - Specify which RTL synthesis tool is used to create a gate-level implementation. The options are:

- **ise** - Xilinx ISE Design Suite (default)
- **synplify** - Synopsys Synplify

The **-tool** option can also be used to pass a string with additional tool options. This option is only available using the command line and is not available in the GUI. For example, to specify that Synplify Professional use a specific license, the following can be used:

```
autoimpl -tool "synplify_pro -licensetype synplifypro_xilinx"
-xil_coregen
```

The string option can also be used to specify the exact executable. For example, synplify_pro, synplify_pro_dp or synplify_premier:

```
autoimpl - tool "synplify_premier"
```

This option uses the CORE Generator tool flow to implement optimized netlists for components in the RTL (such as BRAM/FP).

Pragma

There is no pragma equivalent of the `autoimpl` command.

Examples

This example creates all the scripts for implementation tool, but does not start implementation.

```
autoimpl -setup
```

The following example implements the design using Synplify (with the default Verilog RTL):

```
autoimpl -tool synplify
```

This example creates the scripts for implementing the VHDL using the ISE Design Suite, but does not execute the ISE Design Suite, calls the CORE Generator tool to create optimized modules (such as memories, FIFOs) which are instantiated in the RTL prior to synthesis and creates a pcore directory for use with EDK.

```
autoimpl -rtl vhdl -setup -xil_coregen -export
```

autosim

Syntax

autosim [*OPTIONS*]

Description

Executes post-synthesis co-simulation of the synthesized RTL with the original C-based testbench. The files for the testbench are specified with the `add_file` **-tb** command and option. The simulation is run in sub-directory `sim/`<*HDL*> of the active solution, where <*HDL*> is selected by the **-rtl** option.

For a design to be verified with `autosim`, the design must use interface mode **ap_ctrl_hs** and each output port must use one of the following interface modes which identify, with a write valid signal, when an output is written: **ap_vld**, **ap_ovld**, **ap_hs**, **ap_memory**, **ap_fifo** or **ap_bus**.

Options

-o - Enables optimize compilation of the C testbench and RTL wrapper. Without optimization, `autosim` will compile the testbench as quickly as possible. Enable

optimization to improve the run time performance, if possible, at the expense of compilation time.

Note: Although the resulting executable may potentially run much faster the run time improvements are design dependent. Optimizing for run time may require large amounts of memory for large functions.

-argv <*string*> - Option to specify the argument list for the behavioral testbench. The <string> will be passed onto the main C function.

-coverage - This option enables the coverage feature during simulation with the VCS simulator.

-ignore_init <*integer*> - Disables comparison checking for the first <integer> number of clock cycles. This is useful when it is know the RTL will initially start with unknown ('hX) values.

-ldflags <*string*> - Provides for the specification of options that need to be passed to the linker for co-simulation. This is typically used to pass on include path information or library information for the C test bench.

-mflags <*string*> - Provides for the specification of options that need to be passed to the compiler for SystemC simulation. This is typically used to speed up compilation.

-rtl (**systemc**|**vhdl**|**verilog**) - Selects which RTL is to be used for verification with the C testbench. For Verilog and VHDL a simulator must be specified with the **-tool** option. The default is **systemc**.

-setup - When this option is specified all the simulation files will be created in the sim/<*HDL*> directory of the active solution but the simulation will not be executed.

-tool (**vcs**|**modelsim** |**riviera**) - Option to select the simulator to be used to co-simulate the RTL with the C testbench. No tool needs to be specified for SystemC co-simulation: AutoESL will use its included SystemC kernel.

-trace_level (**none**|**all**) - Determines the level of VCD output which is performed. Option **all** results in all ports and signals being saved to the VCD file. The VCD file is saved in the sim/<*HDL*> directory of the current solution when the simulation executes. The default is **none**.

Pragma

There is no pragma equivalent of the autosim command.

Examples

Perform verification using the SystemC RTL.

```
autosim
```

Use the VCS simulator to verify the Verilog RTL, enable the VCD coverage feature and save all signals in VCD format.

```
autosim -tool VCS -rtl verilog -coverage -trace_level all
```

In this example, the VHDL RTL is verified using ModelSim and values 5 and 1 are passed to the testbench function and used in the RTL verification.

```
autosim -tool modelsim -rtl vhdl -argv "5 1"
```

This final example creates an optimized simulation model for the SystemC RTL but does not execute the simulation. To run the simulation, execute `run.sh` in the `sim/systemc` directory of the active solution.

```
autosim -O -setup
```

autosyn

Syntax

```
autosyn
```

Description

The `autosyn` command synthesizes the AutoESL database for the active solution. The command can only be executed in the context of an active solution. The elaborated design in the database is scheduled and mapped onto RTL, based on any constraints that are set.

Pragma

There is no pragma equivalent of the `autosyn` command.

Examples

Run AutoESL synthesis on the top-level design.

```
autosyn
```

close_project

Syntax

```
close_project
```

Description

The `close_project` command closes the current project, so this project is no longer active in the AutoESL session. The command prevents the designer from entering any project or solution specific commands, but is not really required since opening or creating a new project will automatically close the current active project.

Pragma

There is no pragma equivalent of the `close_project` command.

Examples

Close the current project. All results are automatically saved.

```
close_project
```

close_solution

Syntax

```
close_solution
```

Description

The `close_solution` command closes the current solution, so this solution is no longer active in the AutoESL session. The command prevents the designer from entering any solution specific commands, but is not really required since opening or creating a new solution will automatically close the current active solution.

Pragma

There is no pragma equivalent of the `close_solution` command.

Examples

Close the current solution. All results are automatically saved.

```
close_solution
```

config_array_partition

Syntax

```
config_array_partition [OPTIONS]
```

Description

This command allows the default behavior for array partitioning to be specified.

OPTIONS

-auto_partition_threshold <*int*> - Sets the threshold for automatically partitioning arrays (including those without constant indexing). Arrays which have fewer elements than the specified threshold limit will be automatically partitioned into individual elements, unless interface/core specification is applied on the array. The default is 4.

-auto_promotion_threshold <*int*> - Sets the threshold for automatically partitioning arrays with constant-indexing. Arrays which have fewer elements than the specified threshold limit and have constant-indexing (the indexing is not variable) will be automatically partitioned into individual elements. The default is 64.

-exclude_extern_globals - Excludes external global arrays from throughput driven auto-partitioning. By default, external global arrays are partitioned when option **-throughput_driven** is selected. This option has no effect unless option **-throughput_driven** is selected.

-include_ports - Enables auto-partitioning of IO arrays. This has the effect of reducing an array IO port into multiple ports, each port the size of the individual array elements.

-scalarize_all - This option partitions all arrays in the design into their individual elements.

-throughput_driven - Enables auto-partitioning of arrays based on the throughput. AutoESL will automatically determine if partitioning the array into individual elements will allow it to meet any specified throughput requirements.

Pragma

There is no pragma equivalent of the `config_array_partition` command.

Examples

In this example, all arrays in the design with less than 12 elements, but not global arrays, are automatically partitioned into individual elements.

```
config_array_partition auto_partition_threshold 12 -exclude_extern_globals
```

This example instructs AutoESL to automatically determine which arrays to partition, including arrays on the function interface, to improve throughput.

```
config_array_partition -throughput_driven -include_ports
```

Partition all arrays in the design, including global arrays, into individual elements.

```
config_array_partition -scalarize_all
```

config_bind

Syntax

`config_bind` [*OPTIONS*]

Description

This command allows the default options for micro-architecture binding to be set. Binding is the process where operators, such as addition, multiplication, and shift are mapped to specific RTL implementations; for example a mult operation implemented as a combinational or pipelined RTL multiplier.

Options

`-effort` (**`low`**|**`medium`**|**`high`**) - The optimizing effort level controls the trade off between run-time and optimization. The default is **`medium`** effort.

A **`low`** effort optimization will improve the run time and may be useful for cases where little optimization is possible, for example when all if-else statements have mutually exclusive operators in each branch and no operator sharing can be achieved.

A **`high`** effort optimization will result in increased run time but will typically provide better quality of results.

-min_op <*string*> - This option tells AutoESL to minimize the number of instances of a particular operator. If there are multiple such operators in the code, this will result in them being shared onto the fewest number of RTL resources (cores).

The following operators can be specified as arguments to this command option:

- **add** - Addition
- **sub** - Subtraction
- **mul** - Multiplication
- **icmp** - Integer Compare
- **sdiv** - Signed Division
- **udiv** - Unsigned Division
- **srem** - Signed Remainder
- **urem** - Unsigned Remainder
- **lshr** - Logical Shift-Right
- **ashr** - Arithmetic Shift-Right
- **shl** - Shift-Left

Pragma

There is no pragma equivalent of the `config_bind` command.

Examples

This example tells AutoESL to spend more effort in the binding process, try more options for implementing the operators, and try to produce a design with better resource usage.

```
config_bind -effort high
```

In this example, the number of multiplication operators is minimized, resulting in RTL with the fewest number of multipliers.

```
config_bind -min_op mul
```

config_dataflow

Syntax

```
config_dataflow [OPTIONS]
```

Description

This command specifies the default behavior of dataflow pipelining (implemented by the `set_directive_dataflow` command). This configuration command allows you to specify the default channel memory type and depth.

Options

`-default_channel` (**`fifo`**|**`pingpong`**) - By default a RAM memory, configured in **`pingpong`** fashion, is used to buffer the data between functions or loops when dataflow pipelining is used. When streaming data is used (where the data is always read and written in consecutive order), a FIFO memory will be more efficient and can be selected as the default memory type.

Note: Arrays must be set to streaming using the set_directive_array_stream command in order to perform FIFO accesses.

`-fifo_depth` <*`integer`*> - An integer value specifying the default depth of the FIFOs. This option has no effect when pingpong memories are used. If not specified the FIFOs used in the channel will be set to the size of the largest producer or consumer (whichever is largest). In some cases this may be too conservative and introduce FIFOs which are larger than they need to be. This option can be used when you know the FIFOs are larger than they are required to be. Be careful when using this option as incorrect use may result in a design which fails to operate correctly.

Pragma

There is no pragma equivalent of the `config_dataflow` command.

Examples

Change the default channel from ping-pong memories to a FIFO.

```
config_dataflow -default_channel
```

Change the default channel from ping-pong memories to a FIFO with a depth of 6.

Note: If the design implementation requires a FIFO with greater than 6 elements, this setting will result in which fails RTL verification: this option is a user over-ride and care should be taken when using it.

```
config_dataflow -default_channel fifo -fifo_depth 6
```

config_interface

Syntax

```
config_interface [OPTIONS]
```

Description

The `config_interface` command specifies the default interface used to implement the RTL port of each function argument during interface synthesis. The function arguments can be pass-by-value variables, pointers, arrays and pass-by-reference variables (as permitted by the input language).

In addition, the `config_interface` command can be used to specify the default interface for function level control (such as start, done) and to expose any global variables used by the function as ports on the RTL design.

The default interface is used if none is explicitly specified for the function argument or if an incompatible interface type is specified. A complete list of all interface types is provided below and a detailed explanation of each interface is provided in the *AutoESL User Guide (UG867)*.

Interface Types

`ap_none` - This interface provides no additional handshake or synchronization ports and can be applied to any function argument type except arrays. This is the default interface type for all read-only arguments (input ports), except array arguments.

`ap_ack` - Can be specified on any function argument except arrays and provides an additional acknowledge port to indicate input data has been read by this RTL block or confirm output data has been read by a downstream RTL block.

`ap_vld` - Can be specified on any function argument except arrays and provides an additional data-valid port to indicate when input data is valid and can be read or when output data is valid.

`ap_ovld` - Identical to the **`ap_vld`** interface except that it only applies to write-only arguments (RTL output ports). This is the default interface type for all write-only arguments, except array arguments.

`ap_hs` - Implements each argument with an RTL port supported by a full two-way acknowledge and valid handshake. This can be specified for any function argument except arrays.

ap_fifo - An **ap_fifo** interface can be specified for pointer, array or pass-by-reference arguments. An **ap_fifo** interface implements the data accesses as reads and writes to a FIFO, with associated empty, full and data valid signals.

ap_bus - This interface implements pointer and pass-by-reference variables as a general purpose bus access similar to a typical DMA interface.

ap_memory - This interface is the default type for arrays arguments and can only be specified on array arguments. An **ap_memory** interface results in an RTL implementation which accesses the array elements as data values in a RAM, with associated address, chip enable and write enable control signals. The `set_directive_resource` command should be used to identify which RAM resource in the technology library is used for the array: this will in turn specify the number of ports available and which control signals are implemented.

ap_ctrl_none and **ap_ctrl_hs** - These interface types can only be specified on the function return argument. The **ap_ctrl_hs** is the default and adds function level control signals: an input start signal, output idle and done signals. If there is a function return argument, the done signal signifies when the return value is valid. The **ap_ctrl_none** type ensures these control signals are not added to the design.

Options

-all (**ap_none**|**ap_stable**|**ap_ack**|**ap_vld**|**ap_ovld**|**ap_hs**|**ap_ctrl_none**|**ap_ctrl_hs**|**ap_fifo**|**ap_bus**|**ap_memory**) - The default interface type for all ports types (input, output and inout) and function-level handshakes. The default is **ap_none**.

-clock_enable - Add a clock-enable port (ap_ce) to the design. The clock enable prevents all clock operations when it is active low: disables all sequential operations.

-expose_global - Expose global variables as I/O ports. If a variable is created as a global but all read and write accesses are local to the design, the resource will be created in the design and there is no need for an IO port in the RTL. If however, the global variable is expected to be an external source or destination outside the RTL block, ports should be created using this option.

-in (**ap_none**|**ap_stable**|**ap_ack**|**ap_vld**|**ap_hs**|**ap_fifo**|**ap_bus**|**ap_memory**) - Specify the default interface type for all input (read-only) arguments. The default is **ap_none**.

-inout(**ap_none**|**ap_stable**|**ap_ack**|**ap_vld**|**ap_ovld**|**ap_hs**|**ap_fifo**|**ap_bus**|**ap_memory**) - Specify the default interface type for all inout (read-write) arguments. The default is **ap_none**.

-out (**ap_none**|**ap_ack**|**ap_vld**|**ap_ovld**|**ap_hs**|**ap_fifo**|**ap_bus**|**ap_memory**) - Specify the default interface type for all output (write-only) arguments. The default is **ap_none**.

-return (**ap_ctrl_none**|**ap_ctrl_hs**) - Specify if function level handshakes are used or not. The default is **ap_ctrl_hs**.

Pragma

There is no pragma equivalent of the `config_interface` command.

Examples

Use an acknowledge interface for input ports.

```
config_interface -in ap_ack
```

Specify all outputs to use a handshake interface.

```
config_interface -out ap_hs
```

Do not implement any function level handshakes signals.

```
config_interface -return ap_ctrl_none
```

Configure all IO ports (read-write) to be implemented as valid interfaces and add a clock enable port to the design.

```
config_interface -inout ap_vld -clock_enable
```

config_rtl

Syntax

config_rtl [*OPTIONS*] <*model_name*>

Description

This configures various attributes of the output RTL, such as the type of reset used, the encoding of the state machines and allows a user specific identification to be used in the RTL.

By default, these options are applied to the top-level design and all RTL blocks within the design. Optionally, a specific RTL model may be specified.

<*model_name*> - The RTL module to configure. If none is provided, the top-level design (and all sub-blocks) is assumed.

Options

-header <*string*> - This option places the contents of file <*string*> at the top (as comments) of all output RTL and simulation files. This can be used to ensure the output RTL files contain user specified identification.

-prefix <*string*> - Specify a prefix to be added to all RTL entity/module names.

-reset (**none**|**control**|**state**|**all**) - Variables initialized in the C code are always initialized to the same value in the RTL and hence in the bit-stream. This initialization however is only performed at power-on and not repeated when a reset is applied to the design. The setting applied with the **-reset** option determines how registers/memories are reset. The default is **control**.

- **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.

-reset_async - This option causes all registers to use a asynchronous reset. If this option is not specified a synchronous reset is used.

-reset_level (**low**|**high**) - This option allows the polarity of the reset signal to be either active low or active high. The default is **high**.

-encoding (**bin**|**onehot**|**gray**) - Specify the encoding style used by the design's state machine. The default is **bin**.

Pragma

There is no pragma equivalent of the `config_rtl` command.

Examples

This example configures the output RTL to have all registers reset with an asynchronous active low reset.

```
config_rtl -reset all -reset_async -reset_level low
```

Add the contents of file `my_message.txt` as a comment to all RTL output files.

```
config_rtl -header my_mesage.txt
```

config_schedule

Syntax

```
config_schedule [OPTIONS]
```

Description

This configures the default type of scheduling performed by AutoESL.

Options

-effort (**high**|**medium**|**low**) - Specify the effort used during scheduling operations. The default is **medium** effort. A **low** effort optimization will improve the run time and may be useful for cases where when there are few choices for the design implementation. A **high** effort optimization will result in increased run time but will typically provide better quality of results.

-verbose - The verbose option will print out the critical path when scheduling fails to satisfy any directives or constraints.

Pragma

There is no pragma equivalent of the `config_schedule` command.

Examples

Change the default schedule effort to low to reduce run time.

```
config_schedule -effort low
```

create_clock

Syntax

```
create_clock -period <number> [OPTIONS]
```

Description

The `create_clock` command creates a virtual clock for the current solution. The command can only be executed in the context of an active solution. The clock period is a constraint that drives Autopilot's optimization (chaining as many operations as feasible in the given clock period).

For C and C++ designs, only a single clock is supported. For SystemC designs, multiple named clocks can be created and applied to different SC_MODULEs using the `set_directive_clock` command.

Options

`-name` <*`string`*> - Specify the name of the clock. If no name is given a default name is used.

`-period` <*`number`*> - Specify the clock period in ns or Mhz. If no units are specified then ns are assumed. If no period is specified a default period of 10 ns is used.

Pragma

There is no pragma equivalent of the `create_clock` command.

Examples

Specify a clock period of 50ns.

```
create_clock -period 50
```

This example uses the default period of 10ns to specify the clock.

```
create_clock
```

For a SystemC designs, multiple named clocks can be created (and applied using `set_directive_clock`).

```
create_clock -period 15 fast_clk
create_clock -period 60 slow_clk
```

To specify clock frequency in MHz:

```
create_clock -period 100MHz
```

delete_project

Syntax

```
delete_project <project>
```

Description

The `delete_project` command deletes the directory associated with the project.

The command checks the corresponding project directory `<project>` to ensure it is a valid AutoESL project before deleting it. If no directory `<project>` exists in the current work directory the command has no effect.

`<project>` - Specify the name of the project.

Pragma

There is no pragma equivalent of the `delete_project` command.

Examples

Delete project `Project_1` by removing the directory `./Project_1` and all contents.

```
delete_project Project_1
```

delete_solution

Syntax

```
delete_solution <solution>
```

Description

The `delete_solution` command removes a solution from an active project, and deletes the `<solution>` sub-directory from the project directory.

If the solution does not exist in the project directory this command has no effect.

`<solution>` - Specify the name of the solution to be deleted.

Pragma

There is no pragma equivalent of the `delete_solution` command.

Examples

Delete solution `Solution_1` from the active project by removing the sub-directory Solution_1 from the active project directory.

```
delete_solution Solution_1
```

elaborate

Syntax

```
elaborate [OPTIONS]
```

Description

The elaborate command compiles the source files and creates the AutoESL database for the active solution. The command can only be executed in the context of an active solution.

Some initial processing of functions, loops and arrays is done, based on any directives that are set.

Options

`-effort` (**`low`**|**`medium`**|**`high`**) - By default, the **`medium`** effort is used: this should generally provide the best balance of run time and Quality-of-Results (QoR). The **`high`** effort uses transformations that will lead to the best QoR but will take longer to run. A **`low`** effort will run faster but may result in a less optimal design and should only be used for cases where little or no optimization is possible.

Pragma

There is no pragma equivalent of the `elaborate` command.

Examples

After specifying all input source files and libraries, elaborate the design to create an internal model which can be synthesized into RTL.

```
elaborate
```

Elaborate the design using high effort.

```
elaborate -effort high
```

help

Syntax

help [*OPTIONS*] <*cmd*>

Description

When used without any <*cmd*> the `help` command lists all the AutoESL Tcl commands.

When used with an AutoESL command as an argument, the `help` command provides information on the specified command. Auto-completion using the tab key, for legal AutoESL commands, is active when typing the command argument.

Options

<*cmd*> - Name of the command to provide more help on.

Pragma

There is no pragma equivalent of the `help` command.

Examples

List the help page for command `add_file`.

```
help add_file
```

List all commands and directives used in AutoESL.

```
help
```

list_core

Syntax

```
list_core [OPTIONS]
```

Description

List all the cores in the currently loaded library. Cores are the components used to implement operations in the output RTL (such as adders, multipliers, memories).

After elaboration the operations in the RTL are represented as operators in the internal database. During scheduling operators are mapped to cores from the library to implement the RTL design. Multiple operators may be mapped on the same instance of a core, sharing the same RTL resource.

The `list_core` command allows the available operators and cores to be listed by using the relevant option:

- Operation - This shows which cores in the library can be used to implement each operation.
- Type - Lists the available cores by type, for example those which implement functional operations or those which implement memory/storage operations.

If no options are provided, the command will list all cores in the library.

The information provided by the `list_core` command can be used with the `set_directive_resource` command to implement specific operations onto specific cores.

Options

`-operation` (**`opers`**) - List the cores in the library which can implement the specified operation. The following is the list of operations:

- **`add`** - Addition
- **`sub`** - Subtraction
- **`mul`** - Multiplication
- **`udiv`** - Unsigned Division
- **`urem`** - Unsigned Remainder (Modulus operator)
- **`srem`** - Signed Remainder (Modulus operator)

- **icmp** - Integer Compare
- **shl** - Shift-Left
- **lshr** - Logical Shift-Right
- **ashr** - Arithmetic Shift-Right
- **mux** - Multiplexor
- **load** - Memory Read
- **store** - Memory Write
- **fiforead** - FIFO Read
- **fifowrite** - FIFO Write
- **fifonbread** - Non-Blocking FIFO Read
- **fifonbwrite** - Non-Blocking FIFO Write

-type (**functional_unit**|**storage**|**connector**|**interface**|**ip_block**) - This option will only list cores of the specified type.

- Function Units - Cores which implement standard RTL operations (such as add, multiply, compare)
- Storage - Cores which implement storage elements such as registers or memories.
- Connectors - Cores used to implement connectivity within the design. This included direct connections and streaming storage elements.
- Adapter - Cores which implement interfaces used to connect the top-level design when IP is generated. These interfaces are implemented in the RTL wrapper used in the IP generation flow (Xilinx® EDK).
- IP Blocks - Any IP cores added by you.

Pragma

There is no pragma equivalent of the `list_core` command.

Examples

This example lists all core in the currently loaded libraries which can implement a **add** operation.

```
list_core -operation add
```

Here, all available memory (storage) cores in the library are listed. The `set_directive_resource` command can be used to implement an array using one of the available memories.

```
list_core -type storage
```

list_part

Syntax

list_part *[OPTIONS]*

Description

This command returns the supported device families or supported parts for a given family. If no option is provided, it will return all supported families. To return parts of a family, specify one of the supported families which were listed when no option was provided to the command: this will list the parts supported within that family.

Pragma

There is no pragma equivalent of the `list_part` command.

Examples

This following example returns all supported families.

```
list_part
```

Here, all supported 'virtex6' parts are returned as a list.

```
list_part virtex6
```

open_project

Syntax

open_project [*OPTIONS*] <*project*>

Description

The `open_project` command opens an existing project, or creates a new one.

There can only be one project active at any given time in an AutoESL session. A project can contain multiple solutions. A project can be closed with the `close_project` command or by starting another project with the `open_project` command. The `delete_project` command will completely delete the project directory (removing it from the disk) and any solutions associated it.

<*project*> - Specify the name of the project.

Options

`-reset` - Resets the project by removing any project data which already exists. This option should be used when executing AutoESL with Tcl scripts, otherwise each new `add_file` or `add_library` command will add additional files to the existing data.

Any previous project information on design source files, header file search paths and the top level function is removed. The associated solution directories and files are kept (but may now have invalid results: the `delete_project` command will accomplish the same as the **`-reset`** option and remove all solution data).

Pragma

There is no pragma equivalent of the `open_project` command.

Examples

Open a new or existing project named `Project_1`.

```
open_project Project_1
```

Open a project and remove any existing data (preferred method when using Tcl scripts, to prevent adding source or library files to the existing project data).

```
open_project -reset Project_2
```

open_solution

Syntax

`open_solution` [*OPTIONS*] <*solution*>

Description

The `open_solution` command opens an existing solution or creates a new one in the currently active project. Trying to open or create a solution when there is no active project results in an error. There can only be one solution active at any given time in an AutoESL session.

Each solution is managed in a sub-directory of the current project directory. If the solution does not exist yet in the current work directory, then a new solution is created. A solution can be closed by the `close_solution` command or by opening another solution with the `open_solution` command. The `delete_solution` command will remove them from the project and delete the corresponding subdirectory.

<solution> - Specify the name of the solution.

Options

`-reset` - Resets the solution data if the solution already exists. Any previous solution information on libraries, constraints and directives is removed. Synthesis, verification and implementation results are also removed.

Pragma

There is no pragma equivalent of the `open_solution` command.

Examples

Open a new or existing solution in the active project named `Solution_1`.

```
open_solution Solution_1
```

Open a solution in the active project and remove any existing data (preferred method when using Tcl scripts, to prevent adding to the existing solution data).

```
open_solution -reset Solution_2
```

set_clock_uncertainty

Syntax

`set_clock_uncertainty` *<uncertainty> <clock_list>*

Description

The `set_clock_uncertainty` command sets a margin on the clock period defined by `create_clock`. The margin is subtracted from the clock period to create an effective clock period. If the clock uncertainty is not defined, it will default to 12.5% of the clock period.

AutoESL will optimize the design based on the effective clock period, providing a margin for downstream tools to account for logic synthesis and routing. The command can only be executed in the context of an active solution. AutoESL will still use the specified clock period in all output files for verification and implementation.

For SystemC designs where multiple named clocks are specified by the `create_clock` command, a different clock uncertainty can be specified on each named clock by specifying the named clock.

<*uncertainty*> - A value, specified in ns, which represents how much of the clock period is used as a margin.

<*clock_list*> - A value, specified in ns, which represents how much of the clock period is used as a margin.

Pragma

There is no pragma equivalent of the `set_clock_uncertainty` command.

Examples

Specify an uncertainty/margin of 0.5ns on the clock: this effectively reduces the clock period AutoESL can use by 0.5ns.

```
set_clock_uncertainty 0.5
```

In this SystemC example, two clock domains and created and a different clock uncertainty is specified on each domain. (Multiple clocks are supported in SystemC designs: the `set_directive_clock` command is used to apply the clock to the appropriate function).

```
create_clock -period 15 fast_clk
create_clock -period 60 slow_clk
set_clock_uncertainty 0.5 fast_clock
set_clock_uncertainty 1.5 slow_clock
```

set_directive_allocation

Syntax

```
set_directive_allocation [OPTIONS] <location> <instances>
```

Description

Specify instance restrictions for resource allocation. This defines, and can limit, the number of RTL instances used to implement specific functions or operations.

For example, if the C source has four instances of a function `foo_sub`, the `set_directive_allocation` command can be used to ensure there is only one instance of `foo_sub` in the final RTL (all four instances will be implemented using the same RTL block).

<*location*> - The location string in the format of function[/label].

<*instances*> - A function or operator.

The function can be any function in the original C code and which has not been inlined by the `set_directive_inline` command or inlined automatically by AutoESL.

The list of operators is as follows (provided there is an instance of such an operation in the C source code):

- **add** - Addition
- **sub** - Subtraction
- **mul** - Multiplication
- **icmp** - Integer Compare
- **sdiv** - Signed Division
- **udiv** - Unsigned Division
- **srem** - Signed Remainder
- **urem** - Unsigned Remainder
- **lshr** - Logical Shift-Right
- **ashr** - Arithmetic Shift-Right
- **shl** - Shift-Left

Options

-limit <*integer*> - A maximum limit on the number of instances (of the type defined by the **-type** option) to be used in the RTL design.

-type (**function** | **operation**) - The instance type can be **function** or **operation**. The default is **function**.

Pragma

The pragma should be placed in the C source within the boundaries of the required location.

The format and options are as shown:

```
#pragma AP allocation \
        instances=<Instance Name List> \
        limit=<Integer Value> \
        <operation, function>
```

Examples

Given a design `foo_top` with multiple instances of function `foo`, this command (the pragma is also shown) limits the number of instances of `foo` in the RTL to 2.

```
set_directive_allocation -limit 2 -type function foo_top foo
#pragma AP allocation instances=foo limit=2 function
```

The following command (the pragma is also shown) limits the number of multipliers used in the implementation of `My_func` to 1. Note, this limit does not apply to any multipliers which may reside in sub-functions of `My_func`. To limit the multipliers used in the implementation of any sub-functions, specify an allocation directive on the sub-functions or inline the sub-function into function `My_func`

```
set_directive_allocation -limit 1 -type operation My_func mul
#pragma AP allocation instances=mul limit=1 operation
```

set_directive_array_map

Syntax

set_directive_array_map [*OPTIONS*] <*location*> <*array*>

Description

This command maps a smaller array into a larger array. The typical usage is to use multiple `set_directive_array_map` commands (with the same -instance target) to map multiple smaller arrays into a single larger array which can then be targeted to a single larger memory (RAM or FIFO) resource.

The **-mode** option is used to determine if the new target is a concatenation of elements (horizontal mapping) or bit-widths (vertical mapping). The arrays are concatenated in the order the `set_directive_array_map` commands are issued starting at target element zero in horizontal mapping or bit zero in vertical mapping.

<*location*> - The name of the location, in the format function[/label], which contains the array variable.

<*variable*> - Specify the name of the array variable to be mapped into the new target array instance.

Options

-instance <*string*>- Specify the new array instance name, where the current array variable is to be mapped.

-mode (**horizontal** | **vertical**) - Horizontal mapping concatenates the arrays to form a target with more elements. Vertical mapping concatenates the array to form a target with longer words. The default is **horizontal**.

-offset <*integer*> - This is only relevant for horizontal mapping and specifies an integer value that indicates the absolute offset in the target instance for current mapping operation: element 0 of the array variable will map to element <*int*> of the new target (for example, other elements will map to <*int+1*>, <*int+2*>). If the value is not specified, AutoESL will calculate the required offset automatically, to avoid any overlap (for example, concatenate the arrays starting at the next unused element in the target).

Pragma

The pragma should be placed in the C source within the boundaries of the required location.

The format and options are as shown:

```
#pragma AP array_map \
        variable=<variable> \
        instance=<instance> \
        <horizontal, vertical> \
        offset=<int>
```

Examples

The following commands (the equivalent pragmas are also shown) map arrays A[10] and B[15] in function `foo` into a single new array AB[25]. Element AB[0] will be the same as A[0], element AB[10] will be the same as B[0] (since no -offset option is used) and the bit-width of array AB[25] will be the maximum bit-width of A[10] or B[15].

```
set_directive_array_map -instance AB -mode horizontal foo A
set_directive_array_map -instance AB -mode horizontal foo B
#pragma AP array_map variable=A instance=AB horizontal
#pragma AP array_map variable=B instance=AB horizontal
```

This example concatenates arrays C and D into a new array CD with same number of bits as C and D combined. The number of elements in CD will be maximum of C or D.

```
set_directive_array_map -instance CD -mode vertical foo C
set_directive_array_map -instance CD -mode vertical foo D
#pragma AP array_map variable=C instance=CD vertical
#pragma AP array_map variable=D instance=CD vertical
```

set_directive_array_partition

Syntax

set_directive_array_partition [*OPTIONS*] <*location*> <*array*>

Description

Partitions an array into smaller arrays or individual elements. This will result in RTL with multiple small memories or multiple registers instead of one large memory. This effectively increases the amount of read and write ports for the storage, potentially improving the throughput of the design, but will require more memory instances or registers.

<*location*> - The name of the location, in the format function[/label], which contains the array variable.

<*array*> - Specify the name of the array variable to be partitioned.

Options

-dim <*integer*> - This is only relevant for multi-dimensional arrays and specifies which dimension of the array is to be partitioned. If a value of 0 is used, all dimensions will be

partitioned with the specified options. Any other value will partition only that dimension, for example, if a value 1 is used, only the first dimension will be partitioned.

-factor <*integer*> - Integer number to specify the number of smaller arrays which are to be created. This option is only relevant for type **block** or **cyclic** partitioning.

-type (**block**|**cyclic**|**complete**) - Block partitioning creates smaller arrays from consecutive blocks of the original array. This effectively splits the array into N equal blocks where N is the integer defined by the -factor option. The default is **complete**.

Cyclic partitioning creates smaller arrays by interleaving elements from the original array. For example, if **-factor** 3 is used, element 0 is assigned to the first new array, element 1 to the second new array, element 3 is assigned to the third new array and then element 4 is assigned to the first new array again.

Complete partitioning decomposes the array into individual elements. For a one-dimensional array this corresponds to resolving a memory into individual registers. For multi-dimensional arrays, specify the partitioning of each dimension or use -**dim** 0 to partition all dimensions.

Pragma

The pragma should be placed in the C source within the boundaries of the required location.

The format and options are as shown:

```
#pragma AP array_partition \
        variable=<variable> \
        <block, cyclic, complete> \
        factor=<int> \
        dim=<int>
```

Examples

The following command (the equivalent pragma is also shown) partitions array AB[13] in function `foo` into four arrays. Because 4 is not an integer multiple of 13, three of the arrays will have 3 elements and one will have 4 (containing elements AB[9:12]).

```
set_directive_array_partition -type block -factor 4 foo AB
#pragma AP array_partition variable=AB block factor=4
```

This example partitions array AB[6][4] in function `foo` into two arrays, each of dimension [6][2].

```
set_directive_array_partition -type block -factor 2 -dim 2 foo AB
#pragma AP array_partition variable=AB block factor=2 dim=2
```

All dimensions of AB[4][10][6] in function `foo` are partitioned into individual elements by this command.

```
set_directive_array_partition -type complete -dim 0 foo AB
#pragma AP array_partition variable=AB complete dim=0
```

set_directive_array_reshape

Syntax

set_directive_array_reshape [*OPTIONS*] <*location*> <*array*>

Description

This command combines array partitioning with vertical array mapping, to create a single new array with fewer elements but wider words.

It will first split the array into multiple arrays (in an identical manner as `set_directive_array_partition`) and then automatically recombine the arrays vertically (as per `set_directive_array_map` **-type** vertical) to create a new array with wider words.

<*location*> -The name of the location, in the format function[/label], which contains the array variable.

<*array*> - Specify the name of the array variable to be reshaped.

Options

-dim <*integer*> - This is only relevant for multi-dimensional arrays and specifies which dimension of the array is to be reshaped. If a value of 0 is used, all dimensions will be partitioned with the specified options. Any other value will partition only that dimension, for example, if a value 1 is used, only the first dimension will be partitioned.

-factor <*integer*> - Integer number to specify the number of temporary smaller arrays which are to be created; Only relevant for type **block** or **cyclic** reshaping.

-type (**block**|**cyclic**|**complete**) - Block reshaping creates smaller arrays from consecutive blocks of the original array. This effectively splits the array into N equal blocks where N is the integer defined by the -factor option and then combines the N blocks into a single array with word-width*N. The default is **complete**.

Cyclic reshaping creates smaller arrays by interleaving elements from the original array. For example, if **-factor** 3 is used, element 0 is assigned to the first new array, element

1 to the second new array, element 3 is assigned to the third new array and then element 4 is assigned to the first new array again. The final array is a vertical concatenation (word concatenation, to create longer words) of the new arrays into a single array.

Complete reshaping decomposes the array into temporary individual elements and then recombines them into an array with a wider word. For a one-dimension array this is equivalent to creating a very-wide register (if the original array was N elements of M bits, the result is a register with N*M bits).

Pragma

The pragma should be placed in the C source within the boundaries of the required location.

The format and options are as shown:

```
#pragma AP array_reshape \
        variable=<variable> \
        <block, cyclic, complete> \
        factor=<int> \
        dim=<int>
```

Examples

The following command (the equivalent pragma is also shown) reshapes 8-bit array AB[17] in function `foo`, into a new 32-bit array with 5 elements. Since 4 is not an integer multiple of 13, AB[17] will be in the lower 8-bits of the 5th element (the remainder of the 5th element is unused).

```
set_directive_array_reshape -type block -factor 4 foo AB
#pragma AP array_reshape variable=AB block factor=4
```

This example partitions array AB[6][4] in function `foo`, `i`nto a new array of dimension [6][2], where dimension 2 is twice the width.

```
set_directive_array_reshape -type block -factor 2 -dim 2 foo AB
#pragma AP array_reshape variable=AB block factor=2 dim=2
```

This command reshapes 8-bit array AB[4][2][2] in function `foo`, into a new single element array (a register), 4*2*2*8(=128)-bits wide.

```
set_directive_array_reshape -type complete -dim 0 foo AB
#pragma AP array_reshape variable=AB complete dim=0
```

set_directive_array_stream

Syntax

```
set_directive_array_stream [OPTIONS] <location> <variable>
```

Description

By default, array variables are implemented as RAM (random access) memories:

- Top-level function array parameters are implemented as a RAM interface port.
- General arrays are implemented as RAMs for read-write access.
- In sub-functions involved in dataflow optimizations, the array arguments are implemented using a RAM ping-pong buffer channel.
- Arrays involved in loop-based dataflow optimizations are implemented as a RAM ping-pong buffer channel.

However, if the data stored in the array is consumed/produced in a sequential manner, a more efficient communication mechanism is to use streaming data, where FIFOs are used instead of RAMs.

Note: When an argument of the top-level function is specified as interface type `ap_fifo`, the array is automatically identified as streaming.

<*location*> - The name of the location, in the format function[/label], which contains the array variable.

<*variable*> - Name of the array variable to be implemented as a FIFO.

Options

-depth <*integer*> - Only relevant for array streaming in dataflow channels, this is used to override the default FIFO depth specified (globally) by the `config_dataflow` command.

-off - This option is only relevant for array streaming in dataflow channels. If used, the `config_dataflow -default_channel fifo` command globally implies a `set_directive_array_stream` on all arrays in the design. This option allows streaming to be turned off on a specific array (and default back to using a RAM ping-pong buffer based channel).

Pragma

The pragma should be placed in the C source within the boundaries of the required location.

The format and options are as shown:

```
#pragma AP array_stream
        variable=<variable> \
        off \
        depth=<int>
```

Examples

The following command (the equivalent pragma is also shown) specifies array A[10] in function `foo` to be streaming, and implemented as a FIFO.

```
set_directive_array_stream foo A
#pragma AP array_reshape variable=A
```

In this example, array B in named loop `loop_1` of function `foo`, is set to streaming with a FIFO depth of 12. In this case, the pragma should be placed inside `loop_1`.

```
set_directive_array_stream -depth 12 foo/loop_1 B
#pragma AP array_reshape variable=B depth=12
```

Here, array C has streaming disabled (it's assumed enabled by `config_dataflow` in this example)

```
set_directive_array_stream -off foo C
#pragma AP array_reshape variable=C off
```

set_directive_clock

Syntax

set_directive_clock *<location> <domain>*

Description

Applies the named clock to the specified function.

In C and C++ designs, only a single clock is supported and the clock period specified by `create_clock` is automatically applied to all functions in the design.

SystemC designs support multiple clocks. Multiple named clocks may be specified using the `create_clock` command and applied to individual SC_MODULEs using the `set_directive_clock` command. Each SC_MODULE is synthesized using a single clock.

<*location*> - The name of the function where the named clock is to be applied.

<*domain*> - The name of the clock, as specified by the **-name** option of the `create_clock` command.

Pragma

The pragma should be placed in the C source within the boundaries of the required location.

The format and options are as shown:

```
#pragma AP clock domain=<string>
```

Examples

Given a SystemC design, where top-level `foo_top` has clocks ports `fast_clock` and `slow_clock` but only uses `fast_clock` within its function and sub-block `foo` which only uses `slow_clock`, the following commands create both clocks, apply `fast_clock` to `foo_top` and `slow_clock` to sub-block foo. The equivalent pragmas are also shown and should be placed in the scope of the appropriate function: note there is no pragma equivalent of `create_clock`.

```
create_clock -period 15 fast_clk
create_clock -period 60 slow_clk

set_directive_clock foo_top fast_clock
set_directive_clock foo slow_clock
#pragma AP clock domain=fast_clock
#pragma AP clock domain=slow_clock
```

set_directive_dataflow

Syntax

```
set_directive_dataflow [OPTIONS] <location>
```

Description

The `set_directive_dataflow` command specifies that dataflow optimization be performed on the functions or loops, improving the concurrency of the RTL implementation.

In a C description, all operations are performed in a sequential manner. AutoESL automatically seeks to minimize latency and improve concurrency (in the absence of any directives which limit resources, such as `set_directive_allocation`) however data dependencies can limit this. For example, functions or loops which access arrays must finish all read/write accesses to the arrays before they complete, preventing the next function or loop, which consumes the data, from starting operation.

It may however be possible for the operations in a function or loop to start execution before the previous function or loop completes all its operations.

When dataflow optimization is specified, AutoESL will analyze the dataflow between sequential functions or loops, and seek to create channels (based on ping-pong RAMs or FIFOs) which allow consumer functions or loops to start operation before the producer functions or loops have completed, enabling functions or loops to operate in parallel, decreasing the latency and improving the throughput of the RTL design.

If no initiation interval (number of cycles between the start of one function or loop and the next) is specified, AutoESL will seek to minimize the initiation interval and start operation as soon as data is available.

<*location*> - The name of the location, in the format function[/label], where dataflow optimization is to be performed.

Options

-interval <*integer*> - An integer value specifying the desired initiation interval (II): the number of cycles between the first function or loop executing and the start of execution of the next function or loop.

Pragma

The pragma should be placed in the C source within the boundaries of the required location.

The format and options are as shown:

```
#pragma AP dataflow interval=<int>
```

Examples

This example specifies dataflow optimization within function `foo`. The equivalent pragma is also shown.

```
set_directive_dataflow foo
#pragma AP dataflow interval=3
```

In this example, dataflow is specified in function `My_Func`, with a target initiation interval of 3.

```
set_directive_dataflow -interval 3 My_func
#pragma AP dataflow interval=3
```

set_directive_data_pack

Syntax

set_directive_data_pack [*OPTIONS*] <*location*> <*variable*>

Description

This directive packs the data fields of a struct into a single scalar with a wider word width. Any arrays declared inside the struct will be completely partitioned and reshaped into a wide scalar and be packed with other scalar fields.

The bit alignment of the resulting new wide-word can be inferred from the declaration order of the struct fields. The first field takes the least significant sector of the word and so forth until all fields are mapped.

<*location*> - The name of the location, in the format function[/label], which contains the variable which will be packed.

<*variable*> - Specify the name of the variable to be packed.

Options

-instance <*string*> - Specify the name of resultant variable after packing. If none is provided, the name of the input `variable` will be used.

Pragma

The pragma should be placed in the C source within the boundaries of the required location.

The format and options are as shown:

```
#pragma AP data_pack variable=<variable> instance=<string>
```

Examples

The following command (the equivalent pragma is also shown) packs struct array AB[17] with three 8-bit field fields (typedef struct {unsigned char R, G, B;} pixel) in function `foo`, into a new 17 element array of 24-bits.

```
set_directive_data_pack foo AB
#pragma AP data_pack variable=AB
```

This example (the equivalent pragma is also shown) packs struct pointer AB with three 8-bit fields (typedef struct {unsigned char R, G, B;} pixel) in function `foo`, into a new 24-bit pointer.

```
set_directive_data_pack foo AB
#pragma AP data_pack variable=AB
```

set_directive_dependence

Syntax

set_directive_dependence [*OPTIONS*] <*location*>

Description

AutoESL automatically detects dependencies within loops (a loop-independent dependency) or between different iterations of a loop (a loop-carry dependency). These dependencies impact when operations can be scheduled, especially during function and loop pipelining.

Loop-independent dependence: the same element gets accessed in the same loop iteration.

```
for (i=0;i<N;i++) {
       A[i]=x;
       y=A[i];
}
```

Loop-carry dependence: the same element gets accessed in a different loop iteration.

```
for (i=0;i<N;i++) {
       A[i]=A[i-1]*2;
}
```

Under certain circumstances such as variable dependent array indexing or when an external requirement needs enforced (for example, two inputs are never the same index) the dependence analysis may be too conservative. The `set_directive_dependence` command helps allows you to explicitly specify the dependence and resolve a false dependence.

<*location*> - The name of the location, in the format function[/label], where the dependence is to be specified.

Options

-class (**array**|**pointer**) - Specify a class of variables where the dependence need clarification. This is mutually exclusive with the option **-variable**.

-dependent (**true**|**false**) - Specify if a dependence needs to be enforced (true) or removed (false). The default is **false**.

-direction (**RAW**|**WAR**|**WAW**) - This is only relevant for loop-carry dependencies. Specify the direction for a dependence:

- **RAW** (Read-After-Write - true dependence) - the write instruction uses a value used by the read instruction.
- **WAR** (Write-After-Read - anti dependence) - the read instruction gets a value that is overwritten by the write instruction.
- **WAW** (Write-After-Write - output dependence) - two write instructions write to the same location, in a certain order.

-distance <*integer*> - This is only relevant for loop-carry dependencies where option **-dependent** is set to `true`. A positive integer to specify the inter-iteration distance for array access.

-type (**intra**|**inter**) - Specify if the dependence is within the same loop iteration (intra) or between different loop iterations (inter). The default is **inter**.

-variable <*variable*> - Specify the specific variable to consider for the dependence directive. This is mutually exclusive with the option **-class**.

Pragma

The pragma should be placed in the C source within the boundaries of the required location.

The format and options are as shown:

```
#pragma AP dependence \
        variable=<variable> \
        <array, pointer> \
        <inter, intra> \
        <RAW, WAR, WAW> \
        distance=<int> \
        <false, true>
```

Examples

This example removes the dependence between `Var1` in the same iterations of `loop_1` in function `foo`. The equivalent pragma is also shown.

```
set_directive_dependence -variable Var1 -type intra \
-dependent false foo/loop_1
#pragma AP dependence variable=Var1 intra false
```

Here, the dependence on all arrays in `loop_2` of function foo, is set to inform AutoESL all reads must happen after writes in the same loop iteration.

```
set_directive_dependence -class array -type inter \
-dependent true -direction RAW foo/loop_2
#pragma AP dependence array inter RAW true
```

set_directive_expression_balance

Syntax

set_directive_expression_balance [*OPTIONS*] <*location*>

Description

Sometimes, a C-based specification is written with a sequence of operations. This can result in a lengthy chain of operations in RTL, and with a small clock period, this could increase the design latency.

By default, AutoESL will rearrange the operations, through associative and commutative properties, to create a balanced tree which can shorten the chain, potentially reducing latency at the cost of extra hardware.

The `set_directive_expression_balance` command allows this expression balancing to be turned off or on within with a specified scope.

<location> - The name of the location, in the format function[/label], where the balancing should be enabled or disabled.

Options

`-off` - Turn off expression balancing at this location.

Pragma

The pragma should be placed in the C source within the boundaries of the required location.

The format and options are as shown:

```
#pragma AP expression_balance <off>
```

Examples

This example disables expression balancing within function `My_Func`. The equivalent pragma is also shown.

```
set_directive_expression_balance -off My_Func
#pragma AP expression_balance off
```

Conversely, this example explicitly enables expression balancing in function `My_Func2`.

```
set_directive_expression_balance My_Func2
#pragma AP expression_balance
```

set_directive_function_instantiate

Syntax

`set_directive_function_instantiate` *<location> <variable>*

Description

By default,

- Functions remain as separate hierarchy blocks in the RTL.

- All instances of a function, at the same level of hierarchy, will use the same RTL implementation (block).

The **set_directive_function_instantiate** command is used to create a unique RTL implementation for each instance of a function, allowing each instance to be optimized.

By default, the following code would result in a single RTL implementation of function foo_sub for all three instances.

```
char foo_sub(char inval, char incr)
{
       return inval + incr;
}
void foo(char inval1, char inval2, char inval3,
       char *outval1, char *outval2, char * outval3)
{
       *outval1 = foo_sub(inval1, 1);
       *outval2 = foo_sub(inval2, 2);
       *outval3 = foo_sub(inval3, 3);
}
```

Using the directive, in the manner shown in the example section below, would result in three versions of function foo_sub, each independently optimized for variable incr.

<*location*> - The name of the location, in the format function[/label], where the instances of a function are to be made unique.

variable <*string*> - Specify which function argument <*string*> is to be specified as constant.

Pragma

The pragma should be placed in the C source within the boundaries of the required location.

The format and options are as shown:

```
#pragma AP function_instantiate variable=<variable>
```

Examples

For the example code shown above, the following Tcl (or pragma placed in function foo_sub) allows each instance of function foo_sub to be independently optimized with respect to input **incr.**

```
set_directive_function_instantiate incr foo_sub
#pragma AP function_instantiate variable=incr
```

set_directive_inline

Syntax

```
set_directive_inline [OPTIONS] <location>
```

Description

This command removes a function as a separate entity in the hierarchy. After inlining the function will be dissolved and no longer appear as a separate level of hierarchy.

In some cases inlining a function will allow operations within the function to be shared and optimized more effectively with surrounding operations. An inlined function however, cannot be shared: in some cases this may increase area.

By default, inlining is only performed on the next level of function hierarchy.

<*location*> - The name of the location, in the format function[/label], where inlining is to be performed.

Options

-off - This disables function inlining and is used to prevent particular functions from being inlined. For example, if the **-recursive** option is used in a caller function, this option can prevent a particular callee function from being inlined when all others are.

-recursive - By default only one level of function inlining is performed: the functions within the specified function are not inlined. The **-recursive** option inlines all functions recursively down the hierarchy.

-region - All functions in the specified region are to be inlined.

Pragma

The pragma should be placed in the C source within the boundaries of the required location.

The format and options are as shown:

```
#pragma AP inline <region | recursive | off>
```

Examples

This example inlines all functions in `foo_top` (but not any lower level functions).

```
set_directive_inline -region foo_top
#pragma AP inline region
```

Here, only function `foo_sub1` is inlined.

```
set_directive_inline foo_sub1
#pragma AP inline
```

These commands inline all functions in `foo_top`, recursively down the hierarchy, except function `foo_sub2`. The first pragma is placed in function `foo_top`. The second pragma is placed in function `foo_sub2`.

```
set_directive_inline -region -recursive foo_top
set_directive_inline -off foo_sub2
#pragma AP inline region recursive
#pragma AP inline off
```

set_directive_interface

Syntax

`set_directive_interface` [*OPTIONS*] <*location*> <*port*>

Description

The `set_directive_interface` command specifies how RTL ports are created from the function description during interface synthesis.

The ports in the RTL implementation are derived from:

- Any function-level protocol which is specified.
- Function arguments.
- Global variables - accessed by the top-level function and defined outside its scope.

Function-level handshakes are used to control when the function starts operation and to indicate when function operation ends, is idle and when (in the case of a pipelined function) it is ready for new inputs. The implementation of a function-level protocol is controlled by modes **`ap_ctrl_none`** or **`ap_ctrl_hs`** and only requires the top-level function name (the function `return` should be specified for the pragma).

Each function argument can be specified to have its' own IO protocol (such as valid handshake, acknowledge handshake).

If a global variable is accessed but all read and write operations are local to the design, the resource will be created in the design and there is no need for an IO port in the RTL. If however, the global variable is expected to be an external source or destination it should have its' interface specified in a similar manner as standard function arguments (refer to the examples section).

When `set_directive_interface` is used on sub-functions only the **`-register`** option can be used: the **`-mode`** option is not supported on sub-functions.

<*location*> - The name of the location, in the format function[/label], where the function interface or registered output is to be specified.

<*port*> - The parameter (function argument or global variable) for which the interface has to be synthesized. This is not required when modes **`ap_ctrl_none`** or **`ap_ctrl_hs`** are used.

Options

`-mode` (**`ap_ctrl_none`**|**`ap_ctrl_hs`**|**`ap_none`**|**`ap_stable`**|**`ap_vld`**|**`ap_ovld`**|**`ap_ack`**|**`ap_hs`**|**`ap_fifo`**|**`ap_memory`**|**`ap_bus`**) - Select the appropriate protocol.

Function protocol is implemented by the following **`-mode`** values:

- **`ap_ctrl_none`** - No function-level handshake protocol.
- **`ap_ctrl_hs`** - This is the default behavior and implements a function-level handshake protocol. Input port `ap_start` must go high for the function to begin operation. (All function-level signals are active high). Output port `ap_done` indicates the function is finished (and if there is a function return value, indicates when the return value is valid) and output port `ap_idle` indicates when the function is idle. In pipelined functions, an additional output port `ap_ready` is implemented and indicates when the function is ready for new input data.

For function arguments and global variables, the following default protocol is used for each argument type:

- Read-only (Inputs) - **`ap_none`**
- Write-only (Outputs) - **`ap_vld`**
- Read-Write (Inouts) - **`ap_ovld`**
- Arrays - **`ap_memory`**

The RTL ports to implement function arguments and global variables are specified by the following **`-mode`** values:

- **`ap_none`** - No protocol in place. This corresponds to a simple wire.

- **ap_stable** - Only applicable to input ports, this informs AutoESL that the value on this port is stable after reset and is guaranteed not to change until the next reset. The protocol is implemented as mode **ap_none** but this allows internal optimizations to take place on the signal fanout (note, this is not considered a constant value, simply an unchanging value).
- **ap_vld** - An additional valid port is created (<*port_name*>_vld) to operate in conjunction with this data port. For input ports a read will stall the function until its associated input valid port is asserted. An output port will have its output valid signal asserted when it writes data.
- **ap_ack** - An additional acknowledge port is created (<*port_name*>_ack) to operate in conjunction with this data port. For input ports, a read will assert the output acknowledge when it reads a value. An output write will stall the function until its associated input acknowledge port is asserted.
- **ap_hs** - Additional valid (<*port_name*>_vld) and acknowledge (<*port_name*>_ack) ports are created to operate in conjunction with this data port. For input ports a read will stall the function until its input valid is asserted and will assert its output acknowledge signal when data is read. An output write will assert an output valid when it writes data and stall the function until its associated input acknowledge port is asserted.
- **ap_ovld** - For input signals, this acts as mode **ap_none** and no protocol is added. For output signals, this acts as mode **ap_vld**. For inout signals, the input gets implemented as mode **ap_none** and the output as mode **ap_vld**.
- **ap_memory** - This mode implements array arguments as accesses to an external RAM. Data, address and RAM control ports (such as CE, WE) are created to read from and write the external RAM. The specific signals and number of data ports are determined by the RAM which is being accessed. The array argument should be targeted to a specific RAM in the technology library using the `set_directive_resource command` (or AutoESL will automatically determine the RAM to use).
- **ap_fifo** - Implements array, pointer and pass-by-reference ports as a FIFO access. The data input port will assert its associated output read port (<*port_name*>_read) when it is ready to read new values from the external FIFO and will stall the function until its input available port (<*port_name*>_empty_n) is asserted to indicate a value is available to read. An output data port will asset an output write port (<*port_name*>_write) to indicate it has written a value to the port and will stall the function until its associated input available port (<*port_name*>_full_n) is asserted to indicate there is space available in the external FIFO for new outputs. This interface mode should use the **-depth** option.
- **ap_bus** - implements pointer and pass-by-reference ports as a bus interface. Both input and output ports are synthesized with a number of control signals to support burst access to and from a standard FIFO bus interface. Refer to the *AutoESL User Guide (UG867)* for a detailed description of this interface. This interface mode should use the –depth option.

-depth - The depth option is required for pointer interfaces using **ap_fifo** or **ap_bus** modes. This option should be used to specify the maximum number of samples which will be processed by the testbench. This is required to inform AutoESL about the maximum size of FIFO needed in the verification adapter created for RTL co-simulation.

-register - For the top-level function, this option is relevant for scalar interfaces **ap_none**, **ap_ack**, **ap_vld**, **ap_ovld**, **ap_hs** and causes the signal (and any relevant protocol signals) to be registered and persist until at least the last cycle of the function execution. This option requires the `ap_ctrl_hs` function protocol to be enabled. If this option is used with `ap_ctrl_hs` is results in the function return value being registered.

This option can be used in sub-functions to register the outputs and any control signals until the end of the function execution.

Pragma

The pragma should be placed in the C source within the boundaries of the required location.

The format and options are as shown:

```
#pragma AP interface <mode> register port=<string>
```

Examples

This example turns off function-level handshakes for function `foo`.

```
set_directive_interface -mode ap_ctrl_none foo
#pragma AP interface ap_ctrl_none port=return
```

Here, argument InData in function `foo` is specified to have a **ap_vld** interface and the input should be registered.

```
set_directive_interface -mode ap_vld -register foo InData
#pragma AP interface ap_vld register port=InData
```

This example exposes global variable `lookup_table` used in function `foo` as a port on the RTL design, with an **ap_memory** interface.

```
set_directive_interface -mode ap_memory foo look_table
```

set_directive_latency

Syntax

```
set_directive_latency [OPTIONS] <location>
```

Description

Allows a maximum and/or minimum latency value to be specified on a function, loop or region. AutoESL will always aim for minimum latency. The behavior of AutoESL when minimum and maximum latency values are specified is explained below.

- Latency is less than the minimum - If AutoESL can achieve less than the minimum specified latency, it will extend the latency to the specified value and potentially increasing sharing.
- Latency is greater than the minimum - The constraint is satisfied and no further optimizations are performed.
- Latency is less than the maximum - The constraint is satisfied and no further optimizations are performed.
- Latency is greater than the maximum- -If AutoESL cannot schedule within the maximum limit it will automatically increase effort to achieve the specified constraint. If it still fails to meet the maximum latency a warning is issued and AutoESL will proceed to produce a design with the smallest achievable latency.

<*location*> - The name of the location (function, loop or region), in the format function[/label], to be constrained.

Options

-max <*integer*> - An integer value specifying the maximum latency.

-min <*integer*> - An integer value specifying the minimum latency.

Pragma

The pragma should be placed in the C source within the boundaries of the required location.

The format and options are as shown:

```
#pragma AP latency \
        min=<int> \
```

```
max=<int>
```

Examples

In this example, function `foo` is specified to have a minimum latency of 4 and a maximum latency of 8.

```
set_directive_latency -min=8 -max=8 foo
#pragma AP latency min=4 max=4
```

In function `foo`, loop `loop_row` is specified to have a maximum latency of 12. The pragma should be placed in the loop body.

```
set_directive_latency -max=12 foo/loop_row
#pragma AP latency max=12
```

set_directive_loop_flatten

Syntax

set_directive_loop_flatten [*OPTIONS*] <*location*>

Description

This command is used to flatten nested loops into a single loop hierarchy. In the RTL implementation it will cost a clock cycle to move between loops in the loop hierarchy and flattening nested loops allows them to be optimized as a single loop, saving clock cycles and potentially allowing greater optimization of the loop-body logic.

This directive should be applied to the inner-most loop in the loop hierarchy. Only perfect and semi-perfect loops can be flattened in this manner.

- Perfect loop nest - only the inner-most 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 inner-most 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.

<*location*> - The name of the location (inner-most loop), in the format function[/label].

Options

-off - This option prevents flattening from taking place. This can be used to prevent some loops from being flattened while all others in the specified location are flattened.

Pragma

The pragma should be placed in the C source within the boundaries of the required location.

The format and options are as shown:

```
#pragma AP loop_flatten off
```

Examples

This example flattens `loop_1` in function `foo` and all (perfect or semi-perfect) loops above it in the loop hierarchy, into a single loop. The pragma should be placed in the body of `loop_1`.

```
set_directive_loop_flatten foo/loop_1
#pragma AP loop_flatten
```

Here, loop flattening is prevented in `loop_2` of function `foo`. The pragma should be placed in the body of `loop_2` .

```
set_directive_loop_flatten -off foo/loop_2
#pragma AP loop_flatten off
```

set_directive_loop_merge

Syntax

set_directive_loop_merge *<location>*

Description

Merge all the loops into a single loop. Merging loops reduces the number of clock cycles required in the RTL to transition between the loop-body implementations) and allows the loops be implemented in parallel (if possible).

The rules for loop merging are:

- If the loop bounds are variables, they must have the same value (number of iterations).

- 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.

<*location*> - The name of the location, in the format function[/label], where the loops reside.

Options

-force - This option forces loops to be merged even when AutoESL issues a warning. In this case user takes responsibility that the merged loop will function correctly.

Pragma

The pragma should be placed in the C source within the boundaries of the required location.

The format and options are as shown:

```
#pragma AP loop_merge force
```

Examples

This example merges all consecutive loops in function `foo` into a single loop.

```
set_directive_loop_merge foo
#pragma AP loop_merge
```

In this example, all loops inside `loop_2` (but not `loop_2` itself) of function `foo` are merged by using the **-force** option. The pragma should be placed in the body of `loop_2`.

```
set_directive_loop_merge -force foo/loop_2
#pragma AP loop_merge force
```

set_directive_loop_tripcount

Syntax

```
set_directive_loop_tripcount [OPTIONS] <location>
```

Description

The total number of iterations performed by a loop is referred to as the loop tripcount. AutoESL reports the total latency of each loop: the number of cycles to execute all iterations of the loop. This loop latency is therefore a function of the tripcount (number of loop iterations).

The tripcount could be a constant value, may depend on the value of variables used in the loop expression (for example, x<y) or control statements used inside the loop. In some cases, such as when the variables used to determine the tripcount are input arguments or variables calculated by dynamic operation, AutoESL may not be able to determine the tripcount and hence the loop latency might be unknown.

To help with the design analysis which drives optimization, the `set_directive_loop_tripcount` command allows you to specify minimum, average and maximum tripcounts for a loop and see how the loop latency contributes to the total design latency in the reports.

<*location*> - The name of the location of the loop, in the format function[/label], where the tripcount needs to be specified.

Options

-avg <*integer*> - Specify the average latency.

-max <*integer*> - Specify the maximum latency.

-min <*integer*> - Specify the minimum latency.

Pragma

The pragma should be placed in the C source within the boundaries of the required location.

The format and options are as shown:

```
#pragma AP loop_tripcount \
        min=<int> \
```

```
        max=<int> \
        avg=<int>
```

Examples

In this example, `loop_1` in function `foo` is specified to have a minimum tripcount of 12, an average of 14 and maximum of 16.

```
set_directive_loop_tripcount -min 12 -max 14 -avg 16 foo/loop_1
#pragma AP loop_tripcount min=12 max=14 avg=16
```

set_directive_loop_unroll

Syntax

set_directive_unroll [*OPTIONS*] <*location*>

Description

Transforms loop by creating multiples copies of the loop body.

A loop is executed for the number of iterations specified by the loop induction variable. The number of iterations may also be impacted by any logic inside the loop body (for example, break or modifications to any loop exit variable). The loop is implemented in the RTL by a block of logic, which represents the loop-body, which is executed for the same number of iterations.

The `set_directive_loop_unroll` command allows the loop to be fully unrolled, creating as many copies of the loop-body in the RTL as there are loop iterations, or partially unrolled by a factor N, creating N copies of the loop body and adjusting the loop iteration accordingly.

If the factor N used for partial unrolling is not an integer multiple of the original loop iteration count, the original exit condition needs to be checked after each unrolled fragment of the loop body.

To unroll a loop completely, the loop bounds need to be known at compile time. This is not required for partial unrolling.

<*location*> - The location of the loop, in the format function[/label], to be unrolled.

Options

-factor <*integer*> - Non-zero integer indicating that partial unrolling is requested. The loop body will be repeated this number of times, and the iteration information will be adjusted accordingly.

-region - This option should be specified when seeking to unroll all loops within a loop without unrolling the enclosing loop itself.

Take the example where loop `loop_1` contains multiple loops at the same level of loop hierarchy, loops `loop_2` and `loop_3`. A named loop, such as `loop_1` is also a region/location in the code: a section of code enclosed by braces **{}**. If the unroll directive is specified on location <*function*>`/loop_1`, it will unroll `loop_1`.

The **-region** option specifies that the directive be applied only to the loops enclosed the named region: this results in `loop_1` being left rolled, but all loops inside it (`loop_2` and `loop_3`) being unrolled.

-skip_exit_check - This option is only effective if a factor is specified (partial unrolling).

- Fixed bounds - No exit condition check is performed if the iteration count is a multiple of the factor. For cases where the iteration count is not an integer multiple of the factor, unrolling will be prevented and a warning issued (the exit check must be performed in order to proceed).
- Variable bounds - The exit condition check is removed. The designer is responsible for ensuring the variable bounds is an integer multiple of the factor and that no exit check is in fact required.

Pragma

The pragma should be placed in the C source within the boundaries of the required location.

The format and options are as shown:

```
#pragma AP unroll \
        skip_exit_check \
        factor=<int> \
        region
```

Examples

This example unrolls loop `L1` in function `foo`. The pragma should be placed in the body of loop `L1`.

```
set_directive_loop_unroll foo/L1
```

```
#pragma AP unroll
```

In this example, an unroll factor of 4 is specified on loop `L2` of function `foo` and the exit check is removed. The pragma should be placed in the body of loop `L2`.

```
set_directive_loop_unroll -skip_exit_check -factor 4 foo/L2
#pragma AP unroll skip_exit_check factor=4
```

Here, all loops inside loop `L3` in function `foo` are unrolled, but not loop `L3` itself. The **`-region`** option specifies the location be considered an enclosing region and not a loop label.

```
set_directive_loop_unroll -region foo/L3
#pragma AP unroll region
```

set_directive_occurrence

Syntax

`set_directive_occurrence` [*OPTIONS*] <*location*>

Description

Used when pipelining functions or loops, to specify that the code in a location in executed at a lesser rate than the code in the enclosing function or loop. This allows the code which is executed at the lesser rate to be pipelined a slower rate and potentially shared within the top-level pipeline.

For example, a loop iterates N times, but part of the loop is protected by a conditional statement and only executes M times, where N is an integer multiple of M. The code protected by the conditional is said to have an occurrence of N/M.

If N is pipelined with an initiation interval II, any function or loops protected by the conditional statement may be pipelined with a higher initiation interval than II (at a slower rate: this code is not executed as often) and can potentially be shared better within the enclosing higher rate pipeline.

Indentifying a region with an occurrence allows the functions and loops in this region to be pipelined with an initiation interval which is slower than the enclosing function or loop.

<*location*> - Specify the location which has a slower rate of execution.

Options

-cycle <*int*> - Specify the occurrence N/M, where N is the number of times the enclosing function or loop is executed and M is the number of times the conditional region is executed. N must be an integer multiple of M.

Pragma

The pragma should be placed in the C source within the boundaries of the required location.

The format and options are as shown:

```
#pragma AP occurrence cycle=<int>
```

Examples

In the following example, region `Cond_Region` in function `foo` has an occurrence of 4: it executes at a rate 4 times slower than the code which encompasses it.

```
set_directive_occurrence -cycle 4 foo/Cond_Region
#pragma AP occurrence cycle=4
```

set_directive_pipeline

Syntax

set_directive_pipeline [*OPTIONS*] <*location*>

Description

The `set_directive_pipeline` command specifies the details for pipelining:

- Function pipelining
- Loop pipelining

A pipelined function or loop can process new inputs every N clock cycles, where N is the initiation interval (II). The default initiation interval is 1, which process a new input every clock cycle, or it can be specified by the **-II** option.

If AutoESL cannot create a design with the specified II it will issue a warning and create a design with the lowest possible II: this design can then be analyzed with the warning

message to determine what steps must be taken to create a design which can satisfy the required initiation interval.

<*location*> - The name of the location, in the format function[/label], to be pipelined.

Options

-II <*integer*> - An integer specifying the desired initiation interval for the pipeline. AutoESL will try to meet this request: based on data dependencies, the actual result might have a larger II.

-enable_flush - This option implements a pipeline that can flush pipeline stages if the input of the pipeline stalls. This feature implements additional control logic, has greater area and is optional.

-rewind - This option is only applicable to a loops and enables rewinding, which enables continuous loop pipelining (with no pause between one loop iteration ending and the next starting). Rewinding is effective only if there is one single loop (or a perfect loop nest) inside the top-level function. The code segment before the loop is considered as initialization, will be executed only once in the pipeline and cannot contain any conditional operations (if-else).

Pragma

The pragma should be placed in the C source within the boundaries of the required location.

The format and options are as shown:

```
#pragma AP pipeline \
        II=<int> \
        enable_flush \
        rewind
```

Examples

In the following example, function `foo` is pipelined with an initiation interval of 1.

```
set_directive_pipeline foo
#pragma AP pipeline
```

Here, loop `loop_1` in function `foo` is pipelined with an initiation interval of 4 and pipelining flush is enabled.

```
set_directive_pipeline -II 4 -enable_flush foo/loop_1
#pragma AP pipeline II=4 enable_flush
```

set_directive_protocol

Syntax

```
set_directive_protocol [OPTIONS] <location>
```

Description

This commands specifies a region of the code, a protocol region, in which no clock operations will be inserted by AutoESL unless explicitly specified in the code. A protocol region can be used to manually specify an interface protocol: AutoESL will not insert any clocks between any operations including those which read from or write to function arguments. The order of read and writes will therefore be obeyed at the RTL.

A clock operation may be specified in C using an `ap_wait()` statement (include `ap_utils.h`) and may be specified in C++ and SystemC designs by using the `wait()` statement (include `systemc.h`). The `ap_wait` and `wait` statements have no effect on the simulation of C and C++ designs respectively: they are only interpreted by AutoESL.

A region of code can be created in the C code by enclosing the region in braces **{}** and naming it. (for example, `io_section:{..lines of C code...}` defines a region called `io_section`).

<location> - The name of the location, in the format function[/label], to be implemented in a cycle-accurate manner, corresponding to external protocol requirements.

Options

-mode (**floating** | **fixed**) - The default mode (**floating**) allows the code corresponding to statements outside the protocol region to overlap with the statements in the protocol statements in the final RTL: the protocol region remains cycle accurate but other operations can occur at the same time.

The fixed mode ensures there is no overlap.

Pragma

The pragma should be placed in the C source within the boundaries of the required location.

The format and options are as shown:

```
#pragma AP protocol \
        <floating, fixed>
```

Examples

This example defines region `io_section` in function `foo` as a fixed protocol region. The pragma should be placed inside of region `io_section`.

```
set_directive_protocol -mode fixed foo/io_section
#pragma AP protocol fixed
```

set_directive_resource

Syntax

set_directive_resource -core *<string> <location> <variable>*

Description

Specify that a specific library resource (core) be used to implement a variable (array, arithmetic operation or function argument) in the RTL.

AutoESL will implement the operations in the code using the cores available in the currently loaded library. When multiple cores in the library can be used to implement the variable, the `set_directive_resource` command specifies which core is used. Use the `list_core` command to list the available cores in the library. If no resource is specified, AutoESL will determine the resource to use.

The most common use of `set_directive_resource` is to specify which memory element in the library is used to implement an array. This allows you to control whether, for example, the array is implemented as a single or dual-port RAM. This usage is particularly important for arrays on the top-level function interface, since the memory associated with the array determines the ports in the RTL.

<location> - The location, in the format function[/label], where the variable can be found.

<variable> - Specify the name of the variable.

Options

-core *<string>* - Specify the name of the core, as defined in the technology library.

-port_map *<string>* - This option is used to specify port mappings when using the IP generation flow to map ports on the design with ports on the adapter. The argument to this option is a Tcl list of the design port and adapter ports.

-metadata <*string*> -This option is used to specify bus options when using the IP generation flow. The argument to this option is quoted list of bus operation directives.

Pragma

The pragma should be placed in the C source within the boundaries of the required location.

The format and options are as shown:

```
#pragma AP resource \
        variable=<variable> \
        core=<core>
```

Examples

In this example, variable `coeffs[128]` is an argument to top-level function `foo_top`. This directive specifies coeffs be implemented with core RAM_1P from the library. The ports created in the RTL to access the values of coeffs, will be those defined in the core RAM_1P.

```
set_directive_resource -core RAM_1P foo_top coeffs
#pragma AP resource variable=coeffs core=RAM_1P
```

Given code `Result=A*B` in function `foo`, this example specifies the multiplication be implemented with two-stage pipelined multiplier core, Mul2S.

```
set_directive_resource -core Mul2S foo Result
#pragma AP resource variable=Result core=Mul2S
```

set_directive_top

Syntax

set_directive_top [*OPTIONS*] <*location*>

Description

This directive attaches a name to a function, which can then be used for the `set_top` command. This is typically used to synthesize member functions of a class in C++.

The directive should be specified in an active solution and then the `set_top` command should be used with the new name.

<*location*> - The name of the function to be renamed.

Options

-name <*string*> - Specify the name to be used by the `set_top` command.

Pragma

The pragma should be placed in the C source within the boundaries of the required location.

The format and options are as shown:

```
#pragma AP top \
        name=<string>
```

Examples

In this example, function `foo_long_name` is renamed to `DESIGN_TOP`, which is then specified as the top-level. If the pragma is placed in the code, the `set_top` command must still be issued or the top-level specified in the GUI project settings.

```
set_directive_top -name DESIGN_TOP foo_long_name
#pragma AP top name=DESIGN_TOP
set_top DESIGN_TOP
```

set_directive_unroll

Syntax

set_directive_unroll [*OPTIONS*] <*location*>

Description

This command has been renamed `set_directive_loop_unroll` (same arguments and options as detailed here) and will be deprecated.

Transforms loop by creating multiples copies of the loop body.

A loop is executed for the number of iterations specified by the loop induction variable. The number of iterations may also be impacted by any logic inside the loop body (for example, break or modifications to any loop exit variable). The loop is implemented in the RTL by a block of logic, which represents the loop-body, which is executed for the same number of iterations.

The `set_directive_unroll` command allows the loop to be fully unrolled, creating as many copies of the loop-body in the RTL as there are loop iterations, or partially unrolled by a factor N, creating N copies of the loop body and adjusting the loop iteration accordingly.

If the factor N used for partial unrolling is not an integer multiple of the original loop iteration count, the original exit condition needs to be checked after each unrolled fragment of the loop body.

To unroll a loop completely, the loop bounds need to be known at compile time. This is not required for partial unrolling.

<*location*> - The location of the loop, in the format function[/label], to be unrolled.

Options

-factor <*integer*> - Non-zero integer indicating that partial unrolling is requested. The loop body will be repeated this number of times, and the iteration information will be adjusted accordingly.

-region - This option should be specified when seeking to unroll all loop within a loop without unrolling the enclosing loop itself.

Take the example where loop `loop_1` contains multiple loops at the same level of loop hierarchy, loops `loop_2` and `loop_3`. A named loop, such as `loop_1` is also a region/location in the code: a section of code enclosed by braces **{ }**. If the unroll directive is specified on location <*function*>`/loop_1`, it will unroll `loop_1`.

The -region option specifies that the directive be applied only to the loops enclosed the named region: this results in `loop_1` being left rolled, but all loops inside it (`loop_2` and `loop_3`) being unrolled.

-skip_exit_check - This option is only effective if a factor is specified (partial unrolling).

- Fixed bounds - No exit condition check is performed if the iteration count is a multiple of the factor. For cases where the iteration count is not an integer multiple of the factor, unrolling will be prevented and a warning issued (the exit check must be performed in order to proceed).
- Variable bounds - The exit condition check is removed. The designer is responsible for ensuring the variable bounds is an integer multiple of the factor and that no exit check is in fact required.

Pragma

The pragma should be placed in the C source within the boundaries of the required location.

The format and options are as shown:

```
#pragma AP unroll \
        skip_exit_check \
        factor=<int> \
        region
```

Examples

This example unrolls loop `L1` in function `foo`. The pragma should be placed in the body of loop `L1`.

```
set_directive_unroll foo/L1
#pragma AP unroll
```

In this example, an unroll factor of 4 is specified on loop `L2` of function `foo` and the exit check is removed. The pragma should be placed in the body of loop `L2`.

```
set_directive_unroll -skip_exit_check -factor 4 foo/L2
#pragma AP unroll skip_exit_check factor=4
```

Here, all loops inside loop `L3` in function `foo` are unrolled, but not loop `L3` itself. The **`-region`** option specifies the location be considered an enclosing region and not a loop label.

```
set_directive_unroll -region foo/L3
#pragma AP unroll region
```

set_part

Syntax

`set_part` *<device_specification>*

Description

The `set_part` command sets a target device for the current solution. The command can only be executed in the context of an active solution.

< device_specification> - A device specification sets the target device for AutoESL synthesis and implementation.

<device_family> - The device specification can be simply the device family name, which will use the default device in the family.

<device><package><speed_grade> - The device specification can also be the target device name including device, package and speed-grade information.

Pragma

There is no pragma equivalent of the `set_part` command.

Examples

The FPGA libraries provided with AutoESL can be added to the current solution by simply providing the device family name as shown. In this case, the default device, package and speed-grade specified in the AutoESL FPGA library for this device family are used.

```
set_part virtex6
```

The FPGA libraries provided with AutoESL can optionally specify the specific device with package and speed-grade information.

```
set_part xc6vlx240tff1156-1
```

set_top

Syntax

```
set_top <top>
```

Description

The `set_top` command defines the top-level function to be synthesized. Any functions called from this function will also be part of the design.

<*top*> - Name of the function to be synthesized.

Pragma

There is no pragma equivalent of the `set_top` command.

Examples

This example sets the top-level function as `foo_top`.

```
set_top foo_top
```

Arbitrary Precision Data Types

This chapter provides the details on the Arbitrary Precision (AP) types provided by AutoESL. The sections in the chapter provide details on the associated functions (for C `int#w` types), classes and methods (for C++ `ap_int` and `ap_fixed` types).

Note: When `[u]int#W` types are used, the `autocc` option must be selected in the project settings, to ensure the types are correctly simulated. Functions with these types cannot be analyzed in the debugger.

For information on how to use arbitrary precision types, refer to the *AutoESL User Guide (UG867)*.

C Arbitrary Precision (AP) Types

Compiling [u]int#W Types

In order to use the `[u]int#W` type one must include the `ap_cint.h` header file in all source files which reference `[u]int#W` variables.

When compiling software models that use these types, it may be necessary to specify the location of the AutoESL header files, for example by adding the **`-I`**/<*`AutoESL_HOME`*>**`/include`** option for `gcc` compilation.

Also note that best performance will be observed for software models when compiled with `gcc` **`-O3`** option.

Declaring/Defining [u]int#W Variables

There are separate signed and unsigned C types, respectively:

- int#W
- uint#W

The number #W specifies the total width of the variable being declared.

As usual, user defined types may be created with the C/C++ `typedef` statement as shown among the following examples:

```
include "ap_cint.h"          // use [u]int#W types

typedef uint128 uint128_t;   // 128-bit user defined type
int96 my_wide_var;           // a global variable declaration
```

The maximum width allowed is 1024 bits.

Initialization and Assignment from Constants (Literals)

A `[u]int#W` variable can be initialized with the same integer constants which are supported for the native integer data types. The constants will be zero or sign extended to the full width of the `[u]int#W` variable.

```
#include "ap_cint.h"

uint15   a    = 0;
uint52   b    = 1234567890U;
uint52   c    = 0o12345670UL;
uint96   d    = 0x123456789ABCDEFULL;
```

For bit-widths greater than 64-bit, the following functions can be used.

- `apint_string2bits()`
- `apint_string2bits_bin()`
- `apint_string2bits_oct()`
- `apint_string2bits_hex()`

These functions convert a constant character string of digits, specified within the constraints of the radix (decimal, binary, octal, hexadecimal), into the corresponding value with the given bit-width N. For any radix, the number can be preceded with the minus sign (-), to indicate a negative value.

```
int#N   apint_string2bits[_radix](const char*, int N)
```

This is used to construct integer constants with values that are bigger than what the C language already permits. Smaller values work too, however are easier to specify with the existing C language constant value constructs:

```
#include <stdio.h>
#include "ap_cint.h"

int128 a;

// Set a to the value hex 00000000000000000123456789ABCDF0
a = a-apint_string2bits_hex("-123456789ABCDEF",128);
```

In addition, values can be assigned directly from a character string.

- `apint_vstring2bits()`

This function converts a character string of digits, specified within the constraints of the hexadecimal radix, into the corresponding value with the given bit-width N. The number can be preceded with the minus sign -, to indicate a negative value.

This is used to construct integer constants with values that are larger than what the C language already permits. The function is typically used in a test bench, to read information from a file.

Given file `test.dat` contains the following data:

```
123456789ABCDEF
-123456789ABCDEF
-5
```

The function, used in the test bench, would supply the following values.

```
#include <stdio.h>
#include "ap_cint.h"

typedef data_t;

int128 test (
  int128 t a
  ) {
  return a+1;
}

int main () {
  FILE *fp;
  char vstring[33];

  fp    = fopen("test.dat","r");

  while (fscanf(fp,"%s",vstring)==1) {

    // Supply function "test" with the following values
    // 00000000000000000123456789ABCDF0
    // FFFFFFFFFFFFFFFFFEDCBA9876543212
    // FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFC

    test(apint_vstring2bits_hex(vstring,128));
    printf("\n");
  }

  fclose(fp);
  return 0;
}
```

Support for console I/O (Printing)

A `[u]int#W` variable can be printed with the same conversion specifiers that are supported for the native integer data types. Only the bits that fit according to the conversion specifier will be printed:

```
#include "ap_cint.h"

uint164                    c = 0x123456789ABCDEFULL;

printf("  d%40d\n",c);     // Signed integer in decimal format
//  d               -1985229329
printf(" hd%40hd\n",c);    // Short integer
// hd                                -12817
printf(" ld%40ld\n",c);    // Long integer
// ld                     81985529216486895
printf("lld%40lld\n",c); // Long long integer
// lld                    81985529216486895

printf("  u%40u\n",c);     // Unsigned integer in decimal format
//  u                     2309737967
printf(" hu%40hu\n",c);
// hu                  52719
printf(" lu%40lu\n",c);
// lu                   81985529216486895
printf("llu%40llu\n",c);
// llu                  81985529216486895

printf("  o%40o\n",c);     // Unsigned integer in octal format
//  o                   21152746757
printf(" ho%40ho\n",c);
// ho               146757
printf(" lo%40lo\n",c);
// lo                      4432126361152746757
printf("llo%40llo\n",c);
// llo                    4432126361152746757

printf("  x%40x\n",c);     // Unsigned integer in hexadecimal format [0-9a-f]
// x                     89abcdef
printf(" hx%40hx\n",c);
// hx                    cdef
printf(" lx%40lx\n",c);
// lx                   123456789abcdef
printf("llx%40llx\n",c);
// llx                 123456789abcdef

printf("  X%40X\n",c);     // Unsigned integer in hexadecimal format [0-9A-F]
// X                     89ABCDEF
}
```

As with initialization and assignment to `[u]int#W` variables, features are provided to support printing values which require more than 64-bits to represent.

`apint_print()` - This is used to print integers with values that are larger than what the C language already permits. This function prints a value to `stdout`, interpreted according to the radix (2, 8, 10, 16).

```
void  apint_print(int#N value, int radix)
```

The following example shows the results when `apint_printf()` is used:

```
#include <stdio.h>
#include "ap_cint.h"

int65 Var1 = 44;

apint_print(tmp,2);
//00000000000000000000000000000000000000000000000000000000000101100
apint_print(tmp,8); // 0000000000000000054
apint_print(tmp,10); // 44
apint_print(tmp,16); // 000000000000002C
```

`apint_fprint()` - This is used to print integers with values that are bigger than what the C language already permits. This function prints a value to a file, interpreted according to the radix (2, 8, 10, 16).

```
void  apint_fprint(FILE* file, int#N value, int radix)
```

Expressions Involving [u]int#W types

Variables of `[u]int#W` types may, for the most part, be used freely in expressions involving any C operators. However, there are some behaviors that may seem unexpected and bear detailed explanation.

Zero- and sign-extension on assignment from narrower to wider variables

When assigning the value of a narrower bit-width signed variable to a wider one, the value will be sign-extended to the width of the destination variable, regardless of its signedness.

Similarly, an unsigned source variable will be zero-extended before assignment.

Explicit casting of the source variable may be necessary in order to ensure expected behavior on assignment.

Truncation on assignment of wider to narrower variables

Assigning a wider source variables value to a narrower one will lead to truncation of the value, with all bits beyond the most significant bit (MSB) position of the destination variable being lost.

There is no special handling of the sign information during truncation, which may lead to unexpected behavior. Again, explicit casting may help avoid unexpected behavior.

Binary Arithmetic Operators

In general, any valid operation that may be done on a native C integer data type, is supported for `[u]int#w` types.

Standard binary integer arithmetic operators are overloaded to provide arbitrary precision arithmetic. All of the following operators take either two operands of `[u]int#W` or one `[u]int#W` type and one native C integer data type; for example, char, short, int.

The width and signedness of the resulting value is determined by the width and signedness of the operands, before sign-extension, zero-padding or truncation are applied based on the width of the destination variable (or expression). Details of the return value are described for each operator.

Note that when expressions contain a mix of `[u]int#W` and native C integer types, the C++ types will assume the following widths:

- char – 8-bits
- short – 16-bits
- int – 32-bits
- long – 32-bits
- long long – 64-bits

Addition:

expr1 + *expr2*

This expression produces the sum of two `[u]int#W` (or one `[u]int#W` and a native C integer type).

The width of the sum value will be one bit more than the wider of the two operands (two bits if and only if the wider is unsigned and the narrower is signed).

The sum will be treated as signed if either (or both) of the operands is of a signed type.

Subtraction:

expr1 + *expr2*

This expression produces the difference of two integers.

The width of the difference value will be one bit more than the wider of the two operands (two bits if and only if the wider is unsigned and the narrower signed), before assignment, at which point it will be sign-extended, zero-padded or truncated based on the width of the destination variable.

The difference will be treated as signed regardless of the signedness of the operands.

Multiplication:

expr1 + expr2

This expression returns the product of two integer values.

The width of the product is the sum of the widths of the operands.

The product will be treated as a signed type if either of the operands is of a signed type.

Division:

expr1 + expr2

This expression returns the quotient of two integer values.

The width of the quotient is the width of the dividend if the divisor is an unsigned type; otherwise it is the width of the dividend plus one.

The quotient will be treated as a signed type if either of the operands is of a signed type.

Note: AutoESL synthesis of the divide operator will lead to lead to instantiation of appropriately parameterized Xilinx LogiCORE divider core(s) in the generated RTL.

Modulus:

expr1 + expr2

This expression returns the modulus, or remainder of integer division, for two integer values.

The width of the modulus is the minimum of the widths of the operands, if they are both of the same signedness; if the divisor is an unsigned type and the dividend is signed then the width is that of the divisor plus one.

The quotient will be treated as having the same signedness as the dividend.

Note: AutoESL synthesis of the modulus (%) operator will lead to lead to instantiation of appropriately parameterized Xilinx LogiCORE divider core(s) in the generated RTL.

Bitwise Logical Operators

The bitwise logical operators all return a value with a width that is the maximum of the widths of the two operands and will be treated as unsigned if and only if both operands are unsigned, otherwise it will be of a signed type.

Note: Sign-extension (or zero-padding) may occur, based on the signedness of the expression, not the destination variable.

Bitwise OR:

expr1 + *expr2*

Returns the bitwise OR of the two operands.

Bitwise AND:

expr1 + *expr2*

Returns the bitwise AND of the two operands.

Bitwise XOR:

expr1 + *expr2*

Returns the bitwise XOR of the two operands.

Shift Operators

Each shift operator comes in two versions, one for unsigned right-hand side (RHS) operands and one for signed RHS.

A negative value supplied to the signed RHS versions reverses the shift operations direction, for example, a shift by the absolute value of the RHS operand in the opposite direction will occur.

The shift operators return a value with the same width as the left-hand side (LHS) operand. As with C/C++, if the LHS operand of a shift-right is a signed type, the sign bit will be copied into the most significant bit positions, maintaining the sign of the LHS operand.

Unsigned Integer Shift Right:

expr1 + *expr2*

Integer Shift Right:

expr1 + *expr2*

Unsigned Integer Shift Left:

expr1 + *expr2*

Integer Shift Left:

expr1 + *expr2*

Beware when assigning the result of a shift-left operator to a wider destination variable, as some (or all) information may be lost. It is recommended to explicitly cast the shift expression to the destination type in order to avoid unexpected behavior.

Compound Assignment Operators

The following compound assignment operators are supported:

- *=
- /=
- %=
- +=
- -=
- <<=
- >>=
- &=
- ^=
- |=

The RHS expression is first evaluated then supplied as the RHS operand to the base operator, the result of which is assigned back to the LHS variable. The expression sizing, signedness and potential sign-extension or truncation rules apply as detailed above for the relevant operations.

Relational Operators

All relational operators are supported and return a Boolean value based on the result of the comparison. Variables of `[u]int#W` types may be compared to native C integer types with these operators.

Equality:

expr1 + *expr2*

Inequality:

expr1 + *expr2*

Less than:

expr1 + *expr2*

Greater than:

expr1 + *expr2*

Less than or equal:

expr1 + *expr2*

Greater than or equal:

expr1 + *expr2*

Bit-Level Operation: Support Function

The `[u]int#W` types allow variables to be expressed with bit-level accuracy. It is often desirable with (hardware) algorithms that bit-level operations be performed. AutoESL provides the following functions to enable this.

Bit Manipulation

The following methods are provided in order to facilitate common bit-level operations on the value stored in `[u]int#W` type variable(s):

Length:

`apint_bitwidthof()`

```
int  pint_bitwidthof(type_or_value)
```

This function returns an integer value that provides the number of bits in an arbitrary precision integer value; it can be used with a type or a value:

```
int5 Var1, Res1;

Var1= -1;
Res1 = apint_bitwidthof(Var1); // Res1 is assigned 5
Res1 = apint_and_reduce(int7); // Res1 is assigned 7
```

Concatenation:

`apint_concatenate()`

```
int#(N+M) apint_concatenate(int#N first, int#M second)
```

Concatenates two `[u]int#W` variables, the width of the returned value is the sum of the widths of the operands.

The high and low arguments will be placed in the higher and lower order bits of the result respectively.

C native types, including integer literals, should be explicitly cast to an appropriate `[u]int#W` type before concatenating in order to avoid unexpected results.

Bit selection:

`apint_get_bit()`

```
int apint_get_bit(int#N source, int index)
```

This operation function selects one bit from an arbitrary precision integer value and returns it.

The source must be an `[u]int#W` type and the index argument must be an int value. It specifies the index of the bit to select. The least significant bit has index 0. The highest permissible index is one less than the bit-width of this `[u]int#W`.

Set bit value:

`apint_set_bit()`

```
int#N apint_set_bit(int#N source, int index, int value)
```

This function sets the specified bit, index, of the [u]int#W instance source to the value specified (zero or one).

Range selection:

`apint_get_range()`

```
int#N apint_get_range(int#N source, int high, int low)
```

This operation returns the value represented by the range of bits specified by the arguments.

The Hi argument specifies the most significant bit (MSB) position of the range and Lo the least significant (LSB).

The LSB of the source variable is in position 0. If the Hi argument has a value less than Lo, then the bits are returned in reverse order.

Set range value:

`apint_set_range()`

```
int#N apint_set_range(int#N source, int high, int low, int#M part)
```

This function sets the bits specified of source between, high and low, to the value of part.

Bit Reduction

AND reduce:

`apint_and_reduce()`

```
int apint_and_reduce(int#N value)
```

This function applies the AND operation on all bits in the value, and returns the resulting single bit as an integer value (which can be cast onto a bool):

```
int5 Var1, Res1;

Var1= -1;
Res1 = apint_and_reduce(Var1); // Res1 is assigned 1

Var1= 1;
Res1 = apint_and_reduce(Var1); // Res1 is assigned 0
```

This operation is equivalent go comparing to -1, and return a 1 if it matches, 0 otherwise. Another interpretation is to check that all bits are one.

OR reduce

`apint_or_reduce()`

```
int apint_or_reduce(int#N value)
```

This function applies the OR operation on all bits in the value, and returns the resulting single bit as an integer value (which can be cast onto a bool). This operation is equivalent go comparing to 0, and return a 0 if it matches, 1 otherwise.

```
int5 Var1, Res1;

Var1= 1;
Res1 = apint_or_reduce(Var1); // Res1 is assigned 1

Var1= 0;
Res1 = apint_or_reduce(Var1); // Res1 is assigned 0
```

XOR reduce

`apint_xor_reduce()`

```
int apint_xor_reduce(int#N value)
```

This function applies the OR operation on all bits in the value, and returns the resulting single bit as an integer value (which can be cast onto a bool). This operation is equivalent go counting the ones in the word, and return 1 if there are an even number, or 0 if there are an odd number (even parity).

```
int5 Var1, Res1;

Var1= 1;
Res1 = apint_xor_reduce(Var1); // Res1 is assigned 0

Var1= 0;
Res1 = apint_xor_reduce(Var1); // Res1 is assigned 1
```

NAND reduce

`apint_nand_reduce()`

```
int apint_nand_reduce(int#N value)
```

This function applies the NAND operation on all bits in the value, and returns the resulting single bit as an integer value (which can be cast onto a bool). This is equivalent to comparing this value against -1 (all ones) and returning false if it matches, true otherwise.

```
int5 Var1, Res1;

Var1= 1;
Res1 = apint_nand_reduce(Var1); // Res1 is assigned 1

Var1= -1;
Res1 = apint_nand_reduce(Var1); // Res1 is assigned 0
```

NOR reduce

`apint_nor_reduce()`

```
int apint_nor_reduce(int#N value)
```

This function applies the NOR operation on all bits in the value, and returns the resulting single bit as an integer value (which can be cast onto a bool). This is equivalent to comparing this value against 0 (all zeros) and returning true if it matches, false otherwise.

```
int5 Var1, Res1;

Var1= 0;
Res1 = apint_nor_reduce(Var1); // Res1 is assigned 1

Var1= 1;
Res1 = apint_nor_reduce(Var1); // Res1 is assigned 0
```

XNOR reduce

`apint_xnor_reduce()`

```
int apint_xnor_reduce(int#N value)
```

This function applies the XNOR operation on all bits in the value, and returns the resulting single bit as an integer value (which can be cast onto a bool). This operation is equivalent go counting the ones in the word, and return 1 if there are an odd number, or 0 if there are an even number (odd parity).

```
int5 Var1, Res1;

Var1= 0;
Res1 = apint_xnor_reduce(Var1); // Res1 is assigned 0

Var1= 1;
Res1 = apint_xnor_reduce(Var1); // Res1 is assigned 1
```

C++ Arbitrary Precision (AP) Types

AutoESL provides a C++ template class, `ap_[u]int<>`, that implements arbitrary precision (or bit-accurate) integer data types with consistent, bit-accurate behavior between software and hardware modeling.

This class provides all arithmetic, bit-wise, logical and relational operators allowed for C/C++ fundamental integer types. In addition this class provides methods to handle some useful hardware operations, such as allowing initialization and conversion of variables of widths greater than 64 bits. Details for all operators and class methods are detailed below.

Compiling ap_[u]int<> Types

In order to use the `ap_[u]int<>` classes one must include the `ap_int.h` header file in all source files which reference `ap_[u]int<>` variables.

When compiling software models that use these classes, it may be necessary to specify the location of the AutoESL header files, for example by adding the **`-I`**/<*AutoESL_HOME*>/**`include`** option for `g++` compilation.

Also note that best performance will be observed for software models when compiled with `g++` **`-O3`** option.

Declaring/Defining ap_[u]int<> Variables

There are separate signed and unsigned classes: `ap_int<int_W>` and `ap_uint<int _W>` respectively. The template parameter `int_W` specifies the total width of the variable being declared.

As usual, user defined types may be created with the C/C++ `typedef` statement as shown among the following examples:

```
include "ap_int.h" // use ap_[u]int<> types

typedef ap_uint<128> uint128_t; // 128-bit user defined type
ap_int<96> my_wide_var; // a global variable declaration
```

The default maximum width allowed is 1024 bits; this default may be overridden by defining the macro `AP_INT_MAX_W` with a positive integer value less than or equal to 32768 before inclusion of the `ap_int.h` header file.

Note: Setting the value of `AP_INT_MAX_W` too high may cause slow software compile and run times.

Example of overriding `AP_INT_MAX_W`:

```
#define AP_INT_MAX_W 4096 // Must be defined before next line
```

```
#include "ap_int.h"

ap_int<2048> very_wide_var;
```

Initialization and Assignment from Constants (Literals)

The class constructor and assignment operator overloads, allows initialization of and assignment to `ap_[u]int<>` variables using standard C/C++ integer literals.

However, this method of assigning values to `ap_[u]int<>` variables is subject to the limitations of C++ and the system upon which the software will run, typically leading to a 64-bit limit on integer literals (for example, LL or ULL suffixes).

In order to allow assignment of values wider than 64-bits, the `ap_[u]int<>` classes provide constructors that allow initialization from a string of arbitrary length (less than or equal to the width of the variable).

By default, the string provided will be interpreted as a hexadecimal value as long as it contains only valid hexadecimal digits 0-9 and a-f. In order to assign a value from such a string, an explicit C++ style cast of the string to the appropriate type must be made.

Examples of initialization and assignments, including for values greater than 64-bit, are:

```
ap_int<42> a_42b_var(-1424692392255LL); // long decimal format
a_42b_var = 0x14BB648B13FLL; // hexadecimal format

a_42b_var = -1; // negative int literal sign-extended to full width

ap_uint<96> wide_var("76543210fedcba9876543210"); // Greater than 64-bit
wide_var = ap_int<96>("0123456789abcdef01234567");
```

The `ap_[u]int<>` constructor may be explicitly instructed to interpret the string as representing the number in radix 2, 8, 10, or 16 formats. This is accomplished by adding the appropriate radix value as a second parameter to the constructor call.

If the string literal provided contains any characters that are invalid as digits for the radix specified a compilation error will occur.

Examples using different radix formats:

```
ap_int<6> a_6bit_var("101010", 2); // 42d in binary format
a_6bit_var = ap_int<6>("40", 8);   // 32d in octal format
a_6bit_var = ap_int<6>("55", 10);  // decimal format
a_6bit_var = ap_int<6>("2A", 16);  // 42d in hexadecimal format

a_6bit_var = ap_int<6>("42", 2);   // COMPILE-TIME ERROR! "42" is not binary
```

The radix of the number encoded in the string can also be inferred by the constructor, when it is prefixed with a zero (0) followed by one of the following characters: "b", "o" or "x"; the prefixes "0b", "0o" and "0x" correspond to binary, octal and hexadecimal formats respectively.

Examples using alternate initializer string formats:

```
ap_int<6> a_6bit_var("0b101010", 2);// 42d in binary format
a_6bit_var = ap_int<6>("0o40", 8);    // 32d in octal format
a_6bit_var = ap_int<6>("0x2A", 16);  // 42d in hexidecimal format

a_6bit_var = ap_int<6>("0b42", 2);   // COMPILE-TIME ERROR! "42" is not binary
```

Support for console I/O (Printing)

As with initialization and assignment to `ap_[u]int<>` variables, features are provided to support printing values which require more than 64-bits to represent.

The easiest way to output any value stored in an `ap_[u]int` variable is to use the C++ standard output stream, `std::cout` (`#include <iostream>` or `<iostream.h>`). The stream insertion operator, `<<`, is overloaded to correctly output the full range of values possible for any given `ap_[u]int` variable.The stream manipulators `dec`, `hex` and `oct` are also supported, allowing formatting of the value as decimal, hexadecimal or octal respectively.

Example using `cout` to print values:

```
#include <iostream.h>
// Alternative: #include <iostream>

ap_uint<72> Val("10fedcba9876543210");

cout << Val << endl;        // Yields: "313512663723845890576"
cout << hex << val << endl; // Yields: "10fedcba9876543210"
cout << oct << val << endl; // Yields: "41773345651416625031020"
```

It is also possible to use the standard C library (`#include <stdio.h>`) to print out values larger than 64-bits, by first converting the value to a C++ `std::string`, then to a C character string. The `ap_[u]int` classes provide a method, `to_string()` to do the first conversion and the `std::string` class provides the `c_str()` method to convert to a null-terminated character string.

The `ap[u]int::to_string()` method may be passed an optional argument specifying the radix of the numerical format desired. The valid radix argument values are 2, 8, 10 and 16 for binary, octal, decimal and hexadecimal respectively; the default radix value is 16.

A second optional argument to `ap_[u]int::to_string()` specifies whether to print the non-decimal formats as signed values. This argument is boolean and the default value is `false`, causing the non-decimal formats to be printed as unsigned values.

Examples for using printf to print values:

```
ap_int<72> Val("80fedcba9876543210");

printf("%s\n", Val.to_string().c_str(); // => "80FEDCBA9876543210"
printf("%s\n", Val.to_string(10).c_str(); // => "-2342818482890329542128"
```

```
printf("%s\n", Val.to_string(8).c_str(); // => "401773345651416625031020"
printf("%s\n", Val.to_string(16, true).c_str(); // => "-7F0123456789ABCDF0"
```

Expressions Involving ap_[u]int<> types

Variables of `ap_[u]int<>` types may, for the most part, be used freely in expressions involving any C/C++ operators. However, there are some behaviors that may seem unexpected and bear detailed explanation.

Zero- and sign-extension on assignment from narrower to wider variables

When assigning the value of a narrower bit-width signed (`ap_int<>`) variable to a wider one, the value will be sign-extended to the width of the destination variable, regardless of its signedness.

Similarly, an unsigned source variable will be zero-extended before assignment.

Explicit casting of the source variable, as shown below, may be necessary in order to ensure expected behavior on assignment.

```
ap_uint<10> Result;

ap_int<7> Val1 = 0x7f;
ap_uint<6> Val2 = 0x3f;

Result = Val1;              // Yields: 0x3ff (sign-extended)
Result = Val2;              // Yields: 0x03f (zero-padded)

Result = ap_uint<7>(Val1); // Yields: 0x07f (zero-padded)
Result = ap_int<6>(Val2);  // Yields: 0x3ff (sign-extended)
```

Truncation on assignment of wider to narrower variables

Assigning a wider source variables value to a narrower one will lead to truncation of the value, with all bits beyond the most significant bit (MSB) position of the destination variable being lost.

There is no special handling of the sign information during truncation, which may lead to unexpected behavior. Again, explicit casting may help avoid unexpected behavior.

Class Operators and Methods

In general, any valid operation that may be done on a native C/C++ integer data type, is supported, via operator overloading, for `ap_[u]int` types.

In addition to these overloaded operators, some class specific operators and methods are included to ease bit-level operations.

Binary Arithmetic Operators

Standard binary integer arithmetic operators are overloaded to provide arbitrary precision arithmetic. All of the following operators take either two operands of `ap_[u]int` or one `ap_[u]int` type and one native C integer data type, for example char, short, int.

The width and signedness of the resulting value is determined by the width and signedness of the operands, before sign-extension, zero-padding or truncation are applied based on the width of the destination variable (or expression). Details of the return value are described for each operator.

Note: When expressions contain a mix of `ap_[u]int` and native C integer types, the C++ types will assume the following widths:

- char – 8-bits
- short – 16-bits
- int – 32-bits
- long – 32-bits
- long long – 64-bits

Addition:

```
ap_[u]int::RType ap_[u] int::operator + (ap_[u]int op)
```

This operator produces the sum of two `ap_[u]int` (or one `ap_[u]int` and a native C integer type).

The width of the sum value will be one bit more than the wider of the two operands (two bits if and only if the wider is unsigned and the narrower is signed).

The sum will be treated as signed if either (or both) of the operands is of a signed type.

Subtraction:

```
ap_[u]int::RType ap_[u]int::operator - (ap_[u]int op)
```

This operator produces the difference of two integers.

The width of the difference value will be one bit more than the wider of the two operands (two bits if and only if the wider is unsigned and the narrower signed), before assignment, at which point it will be sign-extended, zero-padded or truncated based on the width of the destination variable.

The difference will be treated as signed regardless of the signedness of the operands.

Multiplication:

```
ap_[u]int::RType ap_[u]int::operator * (ap_[u]int op)
```

This operator returns the product of two integer values.

The width of the product is the sum of the widths of the operands.

The product will be treated as a signed type if either of the operands is of a signed type.

Division:

```
ap_[u]int::RType ap_[u]int::operator / (ap_[u]int op)
```

This operator returns the quotient of two integer values.

The width of the quotient is the width of the dividend if the divisor is an unsigned type; otherwise it is the width of the dividend plus one.

The quotient will be treated as a signed type if either of the operands is of a signed type.

Note: AutoESL synthesis of the divide operator will lead to lead to instantiation of appropriately parameterized Xilinx LogiCORE divider core(s) in the generated RTL.

Modulus:

```
ap_[u]int::RType ap_[u]int::operator % (ap_[u]int op)
```

This operator returns the modulus, or remainder of integer division, for two integer values.

The width of the modulus is the minimum of the widths of the operands, if they are both of the same signedness; if the divisor is an unsigned type and the dividend is signed then the width is that of the divisor plus one.

The quotient will be treated as having the same signedness as the dividend.

Note: AutoESL synthesis of the modulus (%) operator will lead to lead to instantiation of appropriately parameterized Xilinx LogiCORE divider core(s) in the generated RTL.

Examples of arithmetic operators

```
ap_uint<71> Rslt;

ap_uint<42> Val1 = 5;
ap_int<23> Val2 = -8;

Rslt = Val1 + Val2; // Yields: -3 (43 bits) sign-extended to 71 bits
Rslt = Val1 - Val2; // Yields: +3 sign extended to 71 bits
Rslt = Val1 * Val2; // Yields: -40 (65 bits) sign extended to 71 bits
Rslt = 50 / Val2; // Yields: -6 (33 bits) sign extended to 71 bits
Rslt = 50 % Val2; // Yields: +2 (23 bits) sign extended to 71 bits
```

Bitwise Logical Operators

The bitwise logical operators all return a value with a width that is the maximum of the widths of the two operands and will be treated as unsigned if and only if both operands are unsigned, otherwise it will be of a signed type.

Note: Sign-extension (or zero-padding) may occur, based on the signedness of the expression, not the destination variable.

Bitwise OR:

```
ap_[u]int::RType ap_[u]int::operator | (ap_[u]int op)
```

Returns the bitwise OR of the two operands.

Bitwise AND:

```
ap_[u]int::RType ap_[u]int::operator & (ap_[u]int op)
```

Returns the bitwise AND of the two operands.

Bitwise XOR:

```
ap_[u]int::RType ap_[u]int::operator ^ (ap_[u]int op)
```

Returns the bitwise XOR of the two operands.

Unary Operators

Addition:

```
ap_[u]int::RType ap_[u]int::operator + ()
```

Returns the self copy of the `ap_[u]int` operand.

Subtraction:

```
ap_[u]int::RType ap_[u]int::operator - ()
```

This operator returns the negated value of the operand with the same width if it is a signed type or its width plus one if it is unsigned.

The return value is always a signed type.

Bit-wise Inverse:

```
ap_[u]int::RType ap_[u]int::operator ~ ()
```

This operator returns the bitwise-NOT of the operand with the same width and signedness.

Equality Zero:

```
bool ap_[u]int::operator ! ()
```

This operator returns a Boolean `false` value if and only if the operand is equal to zero (0), `true` otherwise.

Shift Operators

Each shift operator comes in two versions, one for unsigned right-hand side (RHS) operands and one for signed RHS.

A negative value supplied to the signed RHS versions reverses the shift operations direction, for example, a shift by the absolute value of the RHS operand in the opposite direction will occur.

The shift operators return a value with the same width as the left-hand side (LHS) operand. As with C/C++, if the LHS operand of a shift-right is a signed type, the sign bit will be copied into the most significant bit positions, maintaining the sign of the LHS operand.

Unsigned Integer Shift Right:

```
ap_[u]int ap_[u]int::operator << (ap_uint<int_W2> op)
```

Integer Shift Right:

```
ap_[u]int ap_[u]int::operator << (ap_int<int_W2> op)
```

Unsigned Integer Shift Left:

```
ap_[u]int ap_[u]int::operator >> (ap_uint<int_W2> op)
```

Integer Shift Left:

```
ap_[u]int ap_[u]int::operator >> (ap_int<int_W2> op)
```

Beware when assigning the result of a shift-left operator to a wider destination variable, as some (or all) information may be lost. It is recommended to explicitly cast the shift expression to the destination type in order to avoid unexpected behavior.

Example for shift operations:

```
ap_uint<13> Rslt;

ap_uint<7> Val1 = 0x41;

Rslt = Val1 << 6;     // Yields: 0x0040, msb of Val1 is lost
Rslt = ap_uint<13>(Val1) << 6; // Yields: 0x1040, no info lost

ap_int<7> Val2 = -63;
Rslt = Val2 >> 4;     // Yields: 0x1ffc, sign is maintained and extended
```

Compound Assignment Operators

The compound assignment operators are supported:

- *=
- /=
- %=
- +=
- -=
- <<=
- >>=
- &=
- ^=
- |=

The RHS expression is first evaluated then supplied as the RHS operand to the base operator, the result of which is assigned back to the LHS variable. The expression sizing, signedness and potential sign-extension or truncation rules apply as detailed above for the relevant operations.

Example of a compound assignment statement:

```
ap_uint<10> Val1 = 630;
ap_int<3> Val2 = -3;
ap_uint<5> Val3 = 27;

Val1 += Val2 - Val3;      // Yields: 600 and is equivalent to:

// Val1 = ap_uint<10>(ap_int<11>(Val1) +
//        ap_int<11>((ap_int<6>(Val2) -
//        ap_int<6>(Val3))));
```

Increment and Decrement Operators

The increment and decrement operators are provided.All return a value of the same width as the operand and which is unsigned if and only if both operands are of unsigned types and signed otherwise.

Pre-increment:

```
ap_[u]int& ap_[u]int::operator ++ ()
```

This operator returns the incremented value of the operand as well as assigning the incremented value to the operand.

Post-increment:

```
const ap_[u]int ap_[u]int::operator ++ (int)
```

This operator returns the value of the operand before assignment of the incremented value to the operand variable.

Pre-decrement:

```
ap_[u]int& ap_[u]int::operator -- ()
```

This operator returns the decremented value of, as well as assigning the decremented value to, the operand.

Post-decrement:

```
const ap_[u]int ap_[u]int::operator -- (int)
```

This operator returns the value of the operand before assignment of the decremented value to the operand variable.

Relational Operators

All relational operators are supported and return a Boolean value based on the result of the comparison. Variables of `ap_[u]int` types may be compared to C/C++ fundamental integer types with these operators.

Equality:

```
bool ap_[u]int::operator == (ap_[u]int op)
```

Inequality:

```
bool ap_[u]int::operator != (ap_[u]int op)
```

Less than:

```
bool ap_[u]int::operator < (ap_[u]int op)
```

Greater than:

```
bool ap_[u]int::operator > (ap_[u]int op)
```

Less than or equal:

```
bool ap_[u]int::operator <= (ap_[u]int op)
```

Greater than or equal:

```
bool ap_[u]int::operator >= (ap_[u]int op)
```

Other Class Methods and Operators

Bit-level Operations

The following methods are provided in order to facilitate common bit-level operations on the value stored in `ap_[u]int` type variable(s).

Length:

```
int ap_[u]int::length ()
```

This method returns an integer value providing the total number of bits in the `ap_[u]int` variable.

Concatenation:

```
ap_concat_ref ap_[u]int::concat (ap_[u]int low)
ap_concat_ref ap_[u]int::operator , (ap_[u]int high, ap_[u]int low)
```

Concatenates two `ap_[u]int` variables, the width of the returned value is the sum of the widths of the operands.

The high and low arguments will be placed in the higher and lower order bits of the result respectively; the `concat()` method places the argument in the lower order bits.

When using the overloaded comma operator, the parentheses are required.The comma operator version may also appear on the LHS of assignment.

C/C++ native types, including integer literals, should be explicitly cast to an appropriate `ap_[u]int` type before concatenating in order to avoid unexpected results.

Examples of concatenation:

```
ap_uint<10> Rslt;

ap_int<3> Val1 = -3;
ap_int<7> Val2 = 54;

Rslt = (Val2, Val1);      // Yields: 0x1B5
Rslt = Val1.concat(Val2); // Yields: 0x2B6
(Val1, Val2) = 0xAB;      // Yields: Val1 == 1, Val2 == 43
```

Bit selection:

```
ap_bit_ref ap_[u]int::operator [] (int bit)
```

This operation function selects one bit from an arbitrary precision integer value and returns it.

The returned value is a reference value, which can be used to set or clear the corresponding bit in this `ap_[u]int`.

The bit argument must be an `int` value. It specifies the index of the bit to select. The least significant bit has index 0. The highest permissible index is one less than the bit-width of this `ap_[u]int`.

The result type `ap_bit_ref` represents the reference to one bit of this `ap_[u]int` instance specified by bit.

Range selection:

```
ap_range_ref ap_[u]int::range (unsigned Hi, unsigned Lo)
ap_range_ref ap_[u]int::operator () (unsigned Hi, unsigned Lo)
```

This operation returns the value represented by the range of bits specified by the arguments.

The Hi argument specifies the most significant bit (MSB) position of the range and Lo the least significant (LSB).

The LSB of the source variable is in position 0. If the Hi argument has a value less than Lo, then the bits are returned in reverse order.

Examples using range selection:

```
ap_uint<4> Rslt;

ap_uint<8> Val1 = 0x5f;
ap_uint<8> Val2 = 0xaa;

Rslt = Val1.range(3, 0); // Yields: 0xF
Val1(3,0) = Val2(3, 0);  // Yields: 0x5A
Val1(4,1) = Val2(4, 1);  // Yields: 0x55
Rslt = Val1.range(7, 4); // Yields: 0xA; bit-reversed!
```

AND reduce:

```
bool ap_[u]int::and_reduce ()
```

This operation function applies the `AND` operation on all bits in this `ap_[u]int` and returns the resulting single bit. This is equivalent to comparing this value against -1 (all ones) and returning `true` if it matches, `false` otherwise.

OR reduce:

```
bool ap_[u]int::or_reduce ()
```

This operation function applies the OR operation on all bits in this ap_(u)int and returns the resulting single bit. This is equivalent to comparing this value against 0 (all zeros) and returning false if it matches, true otherwise.

XOR reduce:

```
bool ap_[u]int::xor_reduce ()
```

This operation function applies the XOR operation on all bits in this ap_int and returns the resulting single bit. This is equivalent to counting the number of 1 bits in this value and returning false if the count is even or true if the count is odd.

NAND reduce:

```
bool ap_[u]int::nand_reduce ()
```

This operation function applies the NAND operation on all bits in this ap_int and returns the resulting single bit. This is equivalent to comparing this value against -1 (all ones) and returning false if it matches, true otherwise.

NOR reduce:

```
bool ap_[u]int::nor_reduce ()
```

This operation function applies the NOR operation on all bits in this ap_int and returns the resulting single bit. This is equivalent to comparing this value against 0 (all zeros) and returning true if it matches, false otherwise.

XNOR reduce:

```
bool ap_[u]int::xnor_reduce ()
```

This operation function applies the XNOR operation on all bits in this ap_(u)int and returns the resulting single bit. This is equivalent to counting the number of 1 bits in this value and returning true if the count is even or false if the count is odd.

Examples of the various bit reduction methods:

```
ap_uint<8> Val = 0xaa;

bool t = Val.and_reduce(); // Yields: false
t = Val.or_reduce();       // Yields: true
t = Val.xor_reduce();      // Yields: false
t = Val.nand_reduce();     // Yields: true
t = Val.nor_reduce();      // Yields: false
t = Val.xnor_reduce();     // Yields: true
```

Bit reverse:

```
void ap_[u]int::reverse ()
```

This member function reverses the contents of ap_[u]int instance, for example, the LSB becomes the MSB and vice versa.

Example of reverse method:

```
ap_uint<8> Val = 0x12;

Val.reverse(); // Yields: 0x48
```

Test bit value:

```
bool ap_[u]int::test (unsigned i)
```

This member function checks whether the specified bit of ap_[u]int instance is 1, returning true if so, false otherwise.

Example of test method:

```
ap_uint<8> Val = 0x12;

bool t = Val.test(5); // Yields: true
```

Set bit value:

```
void ap_(u)int::set (unsigned i, bool v)
void ap_(u)int::set_bit (unsigned i, bool v)
```

This member function sets the specified bit of the ap_[u]int instance to the value of integer V.

Set bit (to 1):

```
void ap_(u)int::set (unsigned i)
```

This member function sets the specified bit of the ap_[u]int instance to the value 1 (one).

Clear bit (to 0):

```
void ap_(u)int:: clear(unsigned i)
```

This member function sets the specified bit of the ap_[u]int instance to the value 0 (zero).

Invert bit:

```
void ap_(u)int:: invert(unsigned i)
```

This member function inverts *i*th bit of the ap_[u]int instance. The *i*th bit will become 0 if its original value is 1 and vice versa.

Example of bit set, clear and invert bit methods:

```
ap_uint<8> Val = 0x12;
Val.set(0, 1);          // Yields: 0x13
Val.set_bit(5, false); // Yields: 0x03
Val.set(7);             // Yields: 0x83
Val.clear(1);           // Yields: 0x81
Val.invert(5);          // Yields: 0x91
```

Rotate Right:

```
void ap_[u]int:: rrotate(unsigned n)
```

This member function rotate the `ap_[u]int` instance *n* places to right.

Rotate Left:

void ap_[u]int:: lrotate(unsigned n)

This member function rotate the `ap_[u]int` instance *n* places to left.

Examples of rotate methods:

```
ap_uint<8> Val = 0x12;

Val.rrotate(3); // Yields: 0x42
Val.lrotate(6); // Yields: 0x90
```

Bitwise NOT:

```
void ap_[u]int:: b_not()
```

This member function complements every bit of the `ap_[u]int` instance.

Example:

```
ap_uint<8> Val = 0x12;

Val.b_not(); // Yields: 0xED
```

Test sign:

```
bool ap_[u]int:: sign()
```

This member function checks whether the `ap_int` instance is negative, returning `true` if negative and return `false` if positive.

Explicit Conversion Methods

To C/C++ "(u)int":

```
int ap_int::to_int ()
unsigned ap_int::to_uint ()
```

These methods return native C/C++ (32-bit on most systems) integers with the value contained in the ap_[u]int. If the value is greater than can be represented by an "[unsigned] int", then truncation will occur.

To C/C++ 64-bit "(u)int64":

```
long long ap_[u]int::to_int64 ()
unsigned long long ap_[u]int::to_uint64 ()
```

These methods return native C/C++ 64-bit integers with the value contained in the ap_[u]int. If the value is greater than can be represented by an "[unsigned] int", then truncation will occur.

To C/C++ "double":

```
double ap_(u)int::to_double ()
```

This method returns a native C/C++ "double" 64-bit floating point representation of the value contained in the ap_[u]int. If the ap_[u]int is wider than 53 bits (the number of bits in the mantissa of a "double"), the resulting "double" may not have the exact value expected.

C++ Arbitrary Precision Fixed Point Types

AutoESL provides support for fixed point types which allow fractional arithmetic to be easily handled. The advantage of fixed point arithmetic is shown in the following example.

```
ap_fixed<10, 5> Var1 = 22.96875; // 10-bit signed word, 5 fractional bits
ap_ufixed<12,11> Var2 = 512.5; // 12-bit word, 1 fractional bit
ap_fixed<13,5> Res1; // 13-bit signed word, 5 fractional bits

Res1 = Var1 + Var2; // Result is 535.46875
```

Even though Var1 and Var2 have different precisions, the fixed point type ensures the decimal point is correctly aligned before the operation (an addition in this case), is performed. You are not required to perform any operations in the C code to align the decimal point.

The type used to store the result of any fixed point arithmetic operation must be large enough, in both the integer and fractional bits, to store the full result.

If this is not the case, the ap_[u]fixed<> type automatically performs overflow handling (when the result has more MSBs than the assigned type supports) and quantization (or rounding: when the result has fewer LSBs than the assigned type supports). The ap_[u]fixed<> type provides a number of options, detailed below, on how the overflow and quantization are performed.

The ap_[u]fixed Representation

In `ap_[u]fixed<>` types, a fixed-point value is represented as a sequence of bits with a specified position for the binary point. Bits to the left of the binary point represent the integer part of the value and bits to the right of the binary point represent the fractional part of the value.

```
ap_[u]fixed type is defined as follows:
ap_[u]fixed<int W,
int I,
ap_q_mode Q
ap_o_mode O,
ap_sat_bits N>;
```

- The `W` attribute takes one parameter: the total number of bits for the word. Only a constant integer expression can be used as the parameter value.
- The `I` attribute takes one parameter: the number of bits to represent the integer part. The value of `I` must be less than or equal to `W`. The number of bits to represent the fractional part is `W` minus `I`. Only a constant integer expression can be used as the parameter value.
- The `Q` attribute takes one parameter: quantization mode. Only predefined enumerated value can be used as the parameter value. The default value is `AP_TRN`.
- The `O` attribute takes one parameter: overflow mode. Only predefined enumerated value can be used as the parameter value. The default value is `AP_WRAP`.
- The `N` attribute takes one parameter: the number of saturation bits considered used in the overflow wrap modes. Only a constant integer expression can be used as the parameter value. The default value is zero.

Note: If the quantization, overflow and saturation parameters are not specified, as in the first example above, the default setting are used.

The quantization and overflow modes are explained below.

Quantization modes

- Rounding to plus infinity: `AP_RND`
- Rounding to zero: `AP_RND_ZERO`
- Rounding to minus infinity: `AP_RND_MIN_INF`
- Rounding to infinity: `AP_RND_INF`
- Convergent rounding: `AP_RND_CONV`
- Truncation: `AP_TRN`
- Truncation to zero: `AP_TRN_ZERO`

AP_RND

The AP_RND quantization mode indicates that the value should be rounded to the nearest representable value for the specific ap_[u]fixed<> type.

For example:

```
ap_fixed<3, 2, AP_RND, AP_SAT> UAPFixed4 = 1.25;    // Yields: 1.5
ap_fixed<3, 2, AP_RND, AP_SAT> UAPFixed4 = -1.25;   // Yields: -1.0
```

AP_RND_ZERO

The AP_RND_ZERO quantization mode indicates the value should be rounded to the nearest representable value and rounding should be towards zero. That is, for positive value, the redundant bits should be deleted, while for negative value, the least significant bits should be added to get the nearest representable value.

For example:

```
ap_fixed<3, 2, AP_RND_ZERO, AP_SAT> UAPFixed4 = 1.25;    // Yields: 1.0
ap_fixed<3, 2, AP_RND_ZERO, AP_SAT> UAPFixed4 = -1.25;   // Yields: -1.0
```

AP_RND_MIN_INF

The AP_RND_MIN_INF quantization mode indicates that the value should be rounded to the nearest representable value, and the rounding should be towards minus infinity. That is, for positive value, the redundant bits should be deleted, while for negative value, the least significant bits should be added.

For example:

```
ap_fixed<3, 2, AP_RND_MIN_INF, AP_SAT> UAPFixed4 = 1.25;   // Yields: 1.0
ap_fixed<3, 2, AP_RND_MIN_INF, AP_SAT> UAPFixed4 = -1.25;  // Yields: -1.5
```

AP_RND_INF

The AP_RND_INF quantization mode indicates that the value should be rounded to the nearest representable value, and the rounding depends on the least significant bit.

- For positive values, if the least significant bit is set, round towards plus infinity, otherwise, round towards minus infinity.
- For negative value, if the least significant bit is set, round towards minus infinity, otherwise, round towards plus infinity.

For example:

```
ap_fixed<3, 2, AP_RND_INF, AP_SAT> UAPFixed4 = 1.25;    // Yields: 1.5
ap_fixed<3, 2, AP_RND_INF, AP_SAT> UAPFixed4 = -1.25;  // Yields: -1.5
```

AP_RND_CONV

The AP_RND_CONV quantization mode indicates that the value should be rounded to the nearest representable value, and the rounding depends on the least significant bit. If the least significant bit is set, round towards plus infinity, otherwise, round towards minus infinity.

For example:

```
ap_fixed<3, 2, AP_RND_CONV, AP_SAT> UAPFixed4 = 0.75;    // Yields: 1.0
ap_fixed<3, 2, AP_RND_CONV, AP_SAT> UAPFixed4 = -1.25; // Yields: -1.0
```

AP_TRN

The AP_TRN quantization mode indicates that the value should be rounded to the nearest representable value, and the rounding should always towards minus infinity.

For example:

```
ap_fixed<3, 2, AP_TRN, AP_SAT> UAPFixed4 = 1.25;   // Yields: 1.0
ap_fixed<3, 2, AP_TRN, AP_SAT> UAPFixed4 = -1.25;  // Yields: -1.5
```

AP_TRN_ZERO

The AP_TRN_ZERO quantization mode indicates that the value should be rounded to the nearest representable value.

- For positive values the rounding is same as mode AP_TRN.
- For negative values, round towards zero.

For example:

```
ap_fixed<3, 2, AP_TRN_ZERO, AP_SAT> UAPFixed4 = 1.25;    // Yields: 1.0
ap_fixed<3, 2, AP_TRN_ZERO, AP_SAT> UAPFixed4 = -1.25;  // Yields: -1.0
```

Overflow modes:

- Saturation: AP_SAT
- Saturation to zero: AP_SAT_ZERO
- Symmetrical saturation: AP_SAT_SYM
- Wrap-around: AP_WRAP
- Sign magnitude wrap-around: AP_WRAP_SM

AP_SAT

The AP_SAT overflow mode indicates the value should be saturated to the maximum value in case of overflow, or to the negative maximum value in case of negative overflow.

For example:

```
ap_ufixed<4, 4, AP_RND, AP_SAT> UAPFixed4 = 19.0;    // Yields: 15.0
ap_fixed<4, 4, AP_RND, AP_SAT> UAPFixed4 = 19.0;     // Yields: 7.0
ap_ufixed<4, 4, AP_RND, AP_SAT> UAPFixed4 = -19.0;  // Yields: 0.0
ap_fixed<4, 4, AP_RND, AP_SAT> UAPFixed4 = -19.0;   // Yields: -8.0
```

AP_SAT_ZERO

The AP_SAT_ZERO overflow mode indicates the value should be forced to in case of overflow, or negative overflow.

For example:

```
ap_ufixed<4, 4, AP_RND, AP_SAT_ZERO> UAPFixed4 = 19.0;    // Yields: 0.0
ap_fixed<4, 4, AP_RND, AP_SAT_ZERO> UAPFixed4 = 19.0;     // Yields: 0.0
ap_ufixed<4, 4, AP_RND, AP_SAT_ZERO> UAPFixed4 = -19.0;  // Yields: 0.0
ap_fixed<4, 4, AP_RND, AP_SAT_ZERO> UAPFixed4 = -19.0;   // Yields: 0.0
```

AP_SAT_SYM

The AP_SAT_SYM overflow mode indicates the value should be saturated to the maximum value in case of overflow, or to the minimum value (negative maximum for signed ap_fixed types and zero for unsigned ap_ufixed types) in case of negative overflow.

For example:

```
ap_ufixed<4, 4, AP_RND, AP_SAT_SYM> UAPFixed4 = 19.0;   // Yields: 15.0
ap_fixed<4, 4, AP_RND, AP_SAT_SYM> UAPFixed4 = 19.0;     // Yields: 7.0
ap_ufixed<4, 4, AP_RND, AP_SAT_SYM> UAPFixed4 = -19.0;  // Yields: 0.0
ap_fixed<4, 4, AP_RND, AP_SAT_SYM> UAPFixed4 = -19.0;   // Yields: -8.0
```

AP_WRAP

The AP_WRAP overflow mode indicates that the value should be wrapped around in case of overflow.

For example:

```
ap_ufixed<4, 4, AP_RND, AP_WRAP> UAPFixed4 = 19.0;      // Yields: 3.0
ap_fixed<4, 4, AP_RND, AP_WRAP> UAPFixed4 = 31.0;       // Yields: -1.0
ap_ufixed<4, 4, AP_RND, AP_WRAP> UAPFixed4 = -19.0;    // Yields: 13.0
ap_fixed<4, 4, AP_RND, AP_WRAP> UAPFixed4 = -19.0;     // Yields: -3.0
```

If the value of N is set to zero (the default overflow mode):

- Any MSB bits outside the range are deleted.
- For unsigned numbers - after the maximum it wraps around to zero.
- For signed numbers - after the maximum, it wraps to the minimum values.

If N>0:

- When N > 0, N MSB bits are saturated or set to 1.
- The sign bit is retained, so positive numbers remain positive and negative numbers remain negative.
- The bits that are not saturated are copied starting from the LSB side.

AP_WRAP_SM

The `AP_WRAP_SM` overflow mode indicates that the value should be sign-magnitude wrapped around.

For example:

```
ap_fixed<4, 4, AP_RND, AP_WRAP_SM> UAPFixed4 = 19.0;   // Yields: -4.0
ap_fixed<4, 4, AP_RND, AP_WRAP_SM> UAPFixed4 = -19.0;  // Yields: 2.0
```

If the value of N is set to zero (the default overflow mode):

- This mode uses sign magnitude wrapping.
- Sign bit set to the value of the least significant deleted bit.
- If the most significant remaining bit is different from the original MSB, all the remaining bits are inverted.
- If MSBs are same, the other bits are copied over.
  a. First delete redundant MSBs.

     The new sign bit is the least significant bit of the deleted bits. 0 in this case.

  b. Compare the new sign bit with the sign of the new value.
- If different, invert all the numbers. They are different in this case.

If N>0:

- Uses sign magnitude saturation.
- N MSBs will be saturated to 1.
- Behaves similar to case where N = 0, except that positive numbers stay positive and negative numbers stay negative.

Compiling ap_[u]fixed<> Types

In order to use the `ap_[u]fixed<>` classes one must include the `ap_fixed.h` header file in all source files which reference `ap_[u]fixed<>` variables.

When compiling software models that use these classes, it may be necessary to specify the location of the AutoESL header files, for example by adding the **`-I`**/<*AutoESL_HOME*>**`/include`** option for `g++` compilation.

Also note that best performance will be observed for software models when compiled with `g++` **`-O3`** option.

Declaring/Defining ap_[u]fixed<> Variables

There are separate signed and unsigned classes: `ap_fixed<W,I>` and `ap_ufixed<W,I>` respectively.

As usual, user defined types may be created with the C/C++ `typedef` statement as shown among the following examples:

```
#include "ap_fixed.h"                    // use ap_[u]fixed<> types

typedef ap_ufixed<128,32> uint128_t; // 128-bit user defined type,
                                     //  32 integer bits
```

Initialization and Assignment from Constants (Literals)

Any `ap_[u]fixed<>` variable may be initialized with normal floating point constants of the usual C/C++ width (32 bits for type `float`, 64 bits for type `double`). That is, typically, a floating point value that is single precision type or in the form of double precision. Such floating point constants will be interpreted and translated into the full width of the arbitrary precision fixed-point variable depending on the sign of the value.

For example:

```
#include <ap_fixed.h>

ap_ufixed<30, 15> my15BitInt = 3.1415;
ap_fixed<42, 23> my42BitInt = -1158.987;
ap_ufixed<99, 40> = 287432.0382911;
```

Support for console I/O (Printing)

As with initialization and assignment to `ap_[u]fixed<>` variables, features are provided to support printing values which require more than 64-bits to represent.

The easiest way to output any value stored in an `ap_[u]fixed<>` variable is to use the C++ standard output stream, `std::cout` (`#include <iostream>` or `<iostream.h>`).

The stream insertion operator, <<, is overloaded to correctly output the full range of values possible for any given ap_[u]fixed<> variable, similarly to the way that floating point values are printed. Note that the stream manipulators dec, hex and oct don't change the behavior of printing other floating point values and hence don't modify the output of ap_[u]fixed<> values either.

Example using cout to print values:

```
#include <iostream.h>
// Alternative: #include <iostream>
ap_uint<72> Val("10fedcba9876543210");
cout << Val << endl; // Yields: "313512663723845890576"
cout << hex << val << endl; // Yields: "10fedcba9876543210"
cout << oct << val << endl; // Yields: "41773345651416625031020"
```

It is also possible to use the standard C library (#include <*stdio.h*>) to print out arbitrary ap_[u]fixed<> values, by first converting the value to a C++ std::string, then to a C character string. The ap_[u]fixed<> classes provide a method, to_string() to do the first conversion and the std::string class provides the c_str() method to convert to a null-terminated character string.

The ap_[u]fixed<>::to_string() method may be passed an optional argument specifying the radix of the numerical format desired. The valid radix argument values are 2, 8, 10 and16 for binary, octal, decimal and hexadecimal respectively; the default radix value is 16.

A second optional argument to ap_[u]fixed<>::to_string() specifies whether to print the non-decimal formats as signed values. This argument is boolean and the default value is false, causing the non-decimal formats to be printed as unsigned values.

Examples for using printf to print values:

```
ap_int<72> Val("80fedcba9876543210");

printf("%s\n", Val.to_string().c_str(); // => "80FEDCBA9876543210"
printf("%s\n", Val.to_string(10).c_str(); // => "-2342818482890329542128"
printf("%s\n", Val.to_string(8).c_str(); // => "401773345651416625031020"
printf("%s\n", Val.to_string(16, true).c_str(); // => "-7F0123456789ABCDF0"
```

Expressions Involving ap_[u]fixed<> types

Arbitrary precision fixed-point values can participate in expressions that use any operators supported by C/C++. Once an arbitrary precision fixed-point type or variable is defined, their usage is the same as for any floating point type or variable in the C/C++ languages. However, there are a few caveats:

Zero and Sign Extensions

Remember that all values of smaller bit-width will be zero or sign extended depending on the sign of the source value. You may need to insert casts to obtain alternative signs when assigning smaller bit-width to larger.

Truncations

When you assign an arbitrary precision fixed-point of larger bit-width than the destination variable, truncation will occur.

Class Operators and Methods

In general, any valid operation that may be done on a native C/C++ integer data type, is supported, via operator overloading, for `ap_[u]fixed<>` types.

In addition to these overloaded operators, some class specific operators and methods are included to ease bit-level operations.

Binary Arithmetic Operators

Addition:

```
ap_[u]fixed<> ap_[u]fixed::operator + (ap_[u]fixed op)
```

This operator function adds this arbitrary precision fixed-point with a given operand op to produce a result.

Note: The operands can be `ap_[u]fixed`, `ap_[u]int` or C/C++ integer types.

The result type `ap_[u]fixed::RType` depends on the type information of the two operands.

For example:

```
ap_fixed<76, 63> Result;

ap_fixed<5, 2> Val1 = 1625.153;
ap_fixed<75, 62> Val2 = 6721.355992351;

Result = Val1 + Val2; // Yields 6722.480957
```

Since `Val2` has the larger bit-width on both integer part and fraction part, the result type has the same bit-width and plus one, in order to be able to store all possible result values.

Subtraction:

```
ap_[u]fixed<> ap_[u]fixed::operator - (ap_[u]fixed op)
```

This operator function subtracts this arbitrary precision fixed-point with a given operand op to produce a result.

The result type `ap_[u]fixed::RType` depends on the type information of the two operands.

For example:

```
ap_fixed<76, 63> Result;

ap_fixed<5, 2> Val1 = 1625.153;
ap_fixed<75, 62> Val2 = 6721.355992351;

Result = Val2 - Val1; // Yields 6720.23057
```

Since `Val2` has the larger bit-width on both integer part and fraction part, the result type has the same bit-width and plus one, in order to be able to store all possible result values.

Multiplication:

```
ap_[u]fixed::RType ap_[u]fixed::operator * (ap_[u]fixed op)
```

This operator function multiplies this arbitrary precision fixed-point with a given operand op to produce a result.

For example:

```
ap_fixed<80, 64> Result;

ap_fixed<5, 2> Val1 = 1625.153;
ap_fixed<75, 62> Val2 = 6721.355992351;

Result = Val1 * Val2; // Yields 7561.525452
```

We have the multiplication of `Val1` and `Val2`. The result type has sum of their integer part bit-width and their fraction part bit width.

Division:

```
ap_[u]fixed::RType ap_[u]fixed::operator / (ap_[u]fixed op)
```

This operator function divides this arbitrary precision fixed-point by a given operand op to produce a result.

For example:

```
ap_fixed<84, 66> Result;

ap_fixed<5, 2> Val1 = 1625.153;
ap_fixed<75, 62> Val2 = 6721.355992351;

Result = Val2 / Val1; // Yields 5974.538628
```

We have the division of `Val1` and `Val2`. In order to preserve enough precision:

- The integer bit-width of the result type is sum of the integer = bit-width of `Val1` and the fraction bit-width of `Val2`.
- The fraction bit-width of the result type is sum of the fraction bit-width of `Val1` and the whole bit-width of `Val2`.

Bitwise Logical Operators

Bitwise OR:

```
ap_[u]fixed<> ap_[u]fixed::operator | (ap_[u]fixed op)
```

This operator function applies or bitwise operation on this arbitrary precision fixed-point and a given operand op to produce a result.

For example:

```
ap_fixed<75, 62> Result;

ap_fixed<5, 2> Val1 = 1625.153;
ap_fixed<75, 62> Val2 = 6721.355992351;

Result = Val1 | Val2; // Yields 6271.480957
```

Bitwise AND:

```
ap_[u]fixed<> ap_[u]fixed::operator & (ap_[u]fixed op)
```

This operator function applies and bitwise operation on this arbitrary precision fixed-point and a given operand op to produce a result.

For example:

```
ap_fixed<75, 62> Result;

ap_fixed<5, 2> Val1 = 1625.153;
ap_fixed<75, 62> Val2 = 6721.355992351;

Result = Val1 & Val2; // Yields 1.00000
```

Bitwise XOR:

```
ap_[u]fixed::RType ap_[u]fixed::operator ^ (ap_[u]fixed op)
```

This operator function applies `xor` bitwise operation on this arbitrary precision fixed-point and a given operand op to produce a result.

For example:

```
ap_fixed<75, 62> Result;
```

```
ap_fixed<5, 2> Val1 = 1625.153;
ap_fixed<75, 62> Val2 = 6721.355992351;

Result = Val1 ^ Val2; // Yields 6720.480957
```

Increment and Decrement Operators

Pre-Increment:

```
ap_[u]fixed<> ap_[u]fixed::operator ++ ()
```

This operator function prefix increases this arbitrary precision fixed-point variable by 1 to produce a result.

For example:

```
ap_fixed<25, 8> Result;
ap_fixed<8, 5> Val1 = 5.125;

Result = ++Val1; // Yields 6.125000
```

Post-Increment:

```
ap_[u]fixed<> ap_[u]fixed::operator ++ (int)
```

This operator function postfix increases this arbitrary precision fixed-point variable by 1, returns the original val of this arbitrary precision fixed-point.

For example:

```
ap_fixed<25, 8> Result;
ap_fixed<8, 5> Val1 = 5.125;

Result = Val1++; // Yields 5.125000
```

Pre-Decrement:

```
ap_[u]fixed<> ap_[u]fixed::operator -- ()
```

This operator function prefix decreases this arbitrary precision fixed-point variable by 1 to produce a result.

For example:

```
ap_fixed<25, 8> Result;
ap_fixed<8, 5> Val1 = 5.125;

Result = --Val1; // Yields 4.125000
```

Post-Decrement:

```
ap_[u]fixed<> ap_[u]fixed::operator -- (int)
```

This operator function postfix decreases this arbitrary precision fixed-point variable by 1, returns the original val of this arbitrary precision fixed-point.

For example:

```
ap_fixed<25, 8> Result;
ap_fixed<8, 5> Val1 = 5.125;

Result = Val1--; // Yields 5.125000
```

Unary Operators

Addition:

```
ap_[u]fixed<> ap_[u]fixed::operator + ()
```

This operator function returns self copy of this arbitrary precision fixed-point variable.

For example:

```
ap_fixed<25, 8> Result;
ap_fixed<8, 5> Val1 = 5.125;

Result = +Val1; // Yields 5.125000
```

Subtraction:

```
ap_[u]fixed<> ap_[u]fixed::operator - ()
```

This operator function returns negative value of this arbitrary precision fixed-point variable.

For example:

```
ap_fixed<25, 8> Result;
ap_fixed<8, 5> Val1 = 5.125;

Result = -Val1; // Yields -5.125000
```

Equality Zero:

```
bool ap_[u]fixed::operator ! ()
```

This operator function compares this arbitrary precision fixed-point variable with 0, and returns the result.

For example:

```
bool  Result;
ap_fixed<8, 5> Val1 = 5.125;

Result = !Val1; // Yields false
```

Bitwise Inverse:

```
ap_[u]fixed::RType ap_[u]fixed::operator ~ ()
```

This operator function returns bitwise complement of this arbitrary precision fixed-point variable.

For example:

```
ap_fixed<25, 15> Result;
ap_fixed<8, 5> Val1 = 5.125;

Result = ~Val1; // Yields -5.25
```

Shift Operators

Unsigned Shift left:

```
ap_[u]fixed<> ap_[u]fixed::operator << (ap_uint<_W2> op)
```

This operator function shifts left by a given integer operand, and returns the result. The operand can be a C/C++ integer type (char, short, int, or long).

The return type of the shift left operation is the same width as type being shifted.

Note: Shift currently cannot support overflow or quantization modes.

For example:

```
ap_fixed<25, 15> Result;
ap_fixed<8, 5> Val = 5.375;

ap_uint<4> sh = 2;

Result = Val << sh; // Yields -10.5
```

The bit-width of the result is (W = 25, I = 15), however, since the shift left operation result type is same as the type of `Val`, the high order two bits of `Val` are shifted out, and the result is -10.5.

If a result of 21.5 is required, `Val` must be cast to `ap_fixed<10, 7>` first: for example, `ap_ufixed<10, 7>(Val)`.

Signed Shift Left:

```
ap_[u]fixed<> ap_[u]fixed::operator << (ap_int<_W2> op)
```

This operator shifts left by a given integer operand, and returns the result. The shift direction depends on whether operand is positive or negative.

- If operand is positive, a shift right is performed.

- If operand is negative, a shift left (opposite direction) is performed.

The operand can be a C/C++ integer type (char, short, int, or long).

The return type of the shift right operation is the same width as type being shifted.

For example:

```
ap_fixed<25, 15,  false> Result;
ap_uint<8, 5> Val = 5.375;

ap_int<4> Sh = 2;
Result = Val << sh; // Shift left, yields -10.25

Sh = -2;
Result = Val << sh; // Shift right, yields 1.25
```

Unsigned Shift Right:

```
ap_[u]fixed<> ap_[u]fixed::operator >> (ap_uint<_W2> op)
```

This operator function shifts right by a given integer operand, and returns the result. The operand can be a C/C++ integer type (char, short, int, or long).

The return type of the shift right operation is the same width as type being shifted.

For example:

```
ap_fixed<25, 15> Result;
ap_fixed<8, 5> Val = 5.375;

ap_uint<4> sh = 2;

Result = Val >> sh; // Yields 1.25
```

If it is required to preserve all significant bits, extend fraction part bit-width of the `Val` first, for example `ap_fixed<10, 5>(Val)`.

Signed Shift Right:

```
ap_[u]fixed<> ap_[u]fixed::operator >> (ap_int<_W2> op)
```

This operator shifts right by a given integer operand, and returns the result. The shift direction depends on whether operand is positive or negative.

- If operand is positive, a shift right is performed.
- If operand is negative, a shift left (opposite direction) is performed.

The operand can be a C/C++ integer type (char, short, int, or long).

The return type of the shift right operation is the same width as type being shifted.

For example:

```
ap_fixed<25, 15, false> Result;
ap_uint<8, 5> Val = 5.375;

ap_int<4> Sh = 2;
Result = Val >> sh; // Shift right, yields 1.25

Sh = -2;
Result = Val >> sh; // Shift left, yields -10.5

1.25
```

Relational Operators

Equality:

```
bool ap_[u]fixed::operator == (ap_[u]fixed op)
```

This operator compares the arbitrary precision fixed-point variable with a given operand, and returns `true` if they are equal and `false` if they are not equal.

The type of operand op can be `ap_[u]fixed<>`, `ap_[u]int` or C/C++ integer types.

For example:

```
bool Result;

ap_ufixed<8, 5> Val1 = 1.25;
ap_fixed<9, 4> Val2 = 17.25;
ap_fixed<10, 5> Val3 = 3.25;

Result = Val1 == Val2; // Yields  true
Result = Val1 == Val3; // Yields  false
```

Non-Equality:

```
bool ap_[u]fixed::operator != (ap_[u]fixed op)
```

This operator compares this arbitrary precision fixed-point variable with a given operand, and returns `true` if they are not equal and `false` if they are equal.

The type of operand op can be `ap_[u]fixed<>`, `ap_[u]int` or C/C++ integer types.

For example:

```
bool Result;

ap_ufixed<8, 5> Val1 = 1.25;
ap_fixed<9, 4> Val2 = 17.25;
ap_fixed<10, 5> Val3 = 3.25;

Result = Val1 != Val2; // Yields false
Result = Val1 != Val3; // Yields true
```

Greater-than-or-equal:

```
bool ap_[u]fixed::operator >= (ap_[u]fixed op)
```

This operator compares a variable with a given operand, and returns `true` if they are equal or if the variable is greater than the operator and `false` otherwise.

The type of operand op can be `ap_[u]fixed<>`, `ap_[u]int` or C/C++ integer types.

For example:

```
bool Result;

ap_ufixed<8, 5> Val1 = 1.25;
ap_fixed<9, 4> Val2 = 17.25;
ap_fixed<10, 5> Val3 = 3.25;

Result = Val1 >= Val2; // Yields true
Result = Val1 >= Val3; // Yields false
```

Less-than-or-equal:

```
bool ap_[u]fixed::operator <= (ap_[u]fixed op)
```

This operator compares a variable with a given operand, and return `true` if it is equal to or less than the operand and `false` if not.

The type of operand op can be `ap_[u]fixed<>`, `ap_[u]int` or C/C++ integer types.

For example:

```
bool Result;

ap_ufixed<8, 5> Val1 = 1.25;
ap_fixed<9, 4> Val2 = 17.25;
ap_fixed<10, 5> Val3 = 3.25;

Result = Val1 <= Val2; // Yields true
Result = Val1 <= Val3; // Yields true
```

Greater-than:

```
bool ap_[u]fixed::operator > (ap_[u]fixed op)
```

This operator compares a variable with a given operand, and return `true` if it is greater than the operand and `false` if not.

The type of operand op can be `ap_[u]fixed<>`, `ap_[u]int` or C/C++ integer types.

For example:

```
bool Result;

ap_ufixed<8, 5> Val1 = 1.25;
```

```
ap_fixed<9, 4> Val2 = 17.25;
ap_fixed<10, 5> Val3 = 3.25;

Result = Val1 > Val2; // Yields false
Result = Val1 > Val3; // Yields false
```

Less-than:

```
bool ap_[u]fixed::operator < (ap_[u]fixed op)
```

This operator compares a variable with a given operand, and return true if it is less than the operand and false if not.

The type of operand op can be `ap_[u]fixed<>`, `ap_[u]int` or C/C++ integer types.

For example:

```
bool Result;

ap_ufixed<8, 5> Val1 = 1.25;
ap_fixed<9, 4> Val2 = 17.25;
ap_fixed<10, 5> Val3 = 3.25;

Result = Val1 < Val2; // Yields false
Result = Val1 < Val3; // Yields true
```

Bit Operator

Bit-Select-and-Set:

```
af_bit_ref ap_[u]fixed::operator [] (int bit)
```

This operator selects one bit from an arbitrary precision fixed-point value and returns it.

The returned value is a reference value, which can be used to set or clear the corresponding bit in the `ap_[u]fixed<>` variable. The bit argument must be an integer value and it specifies the index of the bit to select. The least significant bit has index 0. The highest permissible index is one less than the bit-width of this `ap_[u]fixed<>` variable.

The result type is `af_bit_ref` with a value of either 0 or 1.

For example:

```
ap_int<8, 5> Value = 1.375;

Value[3];  // Yields  1
Value[4];  // Yields  0

Value[2] = 1; // Yields 1.875
Value[3] = 0; // Yields 0.875
```

Bit Range:

```
af_range_ref af_[u]fixed::range (unsigned Hi, unsigned Lo)
af_range_ref af_[u]fixed::operator () (unsigned Hi, unsigned Lo)
```

This operation is similar to bit-select operator `[]` except that it operates on a range of bits instead of a single bit.

It selects a group of bits from the arbitrary precision fixed point variable. The `Hi` argument provides the upper range of bits to be selected. The `Lo` argument provides the lowest bit to be selected. If `Lo` is larger than `Hi` the bits selected will be returned in the reverse order.

The return type af_range_ref represents a reference in the range of the `ap_[u]fixed<>` variable specified by `Hi` and `Lo`.

For example:

```
ap_uint<4> Result = 0;
ap_ufixed<4, 2> Value = 1.25;
ap_uint<8> Repl = 0xAA;

Result = Value.range(3, 0); // Yields: 0x5
Value(3, 0) = Repl(3, 0);    // Yields: -1.5

// when Lo > Hi, return the reverse bits string
Result = Value.range(0, 3); // Yields: 0xA
```

Range Select:

```
af_range_ref af_[u]fixed::range ()
af_range_ref af_[u]fixed::operator ()
```

This operation is the special case of the range select operator `[]`. It selects all bits from this arbitrary precision fixed point value in the normal order.

The return type `af_range_ref` represents a reference to the range specified by Hi = W - 1 and Lo = 0.

For example:

```
ap_uint<4> Result = 0;

ap_ufixed<4, 2> Value = 1.25;
ap_uint<8> Repl = 0xAA;

Result = Value.range(); // Yields: 0x5
Value() = Repl(3, 0); // Yields: -1.5
```

Length:

```
int ap_[u]fixed::length ()
```

This function returns an integer value that provides the number of bits in an arbitrary precision fixed-point value. It can be used with a type or a value.

For example:

```
ap_ufixed<128, 64> My128APFixed;

int bitwidth = My128APFixed.length(); // Yields 128
```

Explicit Conversion to Methods

Fixed-toDouble:

```
double ap_[u]fixed::to_double ()
```

This member function returns this fixed-point value in form of IEEE double precision format.

For example:

```
ap_ufixed<256, 77> MyAPFixed = 333.789;
double Result;

Result = MyAPFixed.to_double(); // Yields 333.789
```

Fixed-to-ap_int:

```
ap_int ap_[u]fixed::to_ap_int ()
```

This member function explicitly converts this fixed-point value to `ap_int` that captures all integer bits (fraction bits are truncated).

For example:

```
ap_ufixed<256, 77> MyAPFixed = 333.789;
ap_uint<77> Result;

Result = MyAPFixed.to_ap_int(); // Yields 333
```

Fixed-to-integer:

```
int ap_[u]fixed::to_int ()
unsigned ap_[u]fixed::to_uint ()
ap_slong ap_[u]fixed::to_int64 ()
ap_ulong ap_[u]fixed::to_uint64 ()
```

This member function explicitly converts this fixed-point value to C built-in integer types.

For example:

```
ap_ufixed<256, 77> MyAPFixed = 333.789;
unsigned int  Result;
```

```
Result = MyAPFixed.to_uint(); // Yields 333

unsigned long long Result;
Result = MyAPFixed.to_uint64(); //Yields 333
```

Appendix A

Additional Resources

Xilinx Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see the Xilinx Support website at:

www.xilinx.com/support.

For a glossary of technical terms used in Xilinx documentation, see:

www.xilinx.com/company/terms.htm.

Solution Centers

See the Xilinx Solution Centers for support on devices, software tools, and intellectual property at all stages of the design cycle. Topics include design assistance, advisories, and troubleshooting tips.