AutoESL Operator and Core Guide

UG870 (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.

Table of Contents

AutoESL Operator and Core Guide

AutoESL transforms a C, C++ or SystemC design specification into a Register Transfer Level (RTL) implementation which in turn can be synthesized into a Xilinx Field Programmable Gate Array (FPGA).

To perform this task AutoESL, does the following:

- First elaborate the C, C++ or SystemC source code into an internal database containing operators.
  - The operators represent operations in the C code such as additions, multiplications, array reads and writes etc.
- During synthesis, AutoESL maps the operators to cores from the AutoESL library.
  - Cores are the specific hardware components used to create the design (such as adders, multipliers, pipelined multipliers, and block RAMs)
  - A separate library is provided for each Xilinx technology (Spartan®-6, Virtex®-7, and other Xilinx® devices)

This document provides details on the operators supported by AutoESL and the accompanying libraries.

Synthesis Overview

When High-level Synthesis (HLS) is performed to transform C source code into a Register Transfer Level (RTL) description, the source code is elaborated into an internal database which contains operators.

The operators are explained in this *AutoESL Operator and Core Guide*, but basically consist of all the operations which can occur in the C source code, such as: additions, shifts, multiplications, bit-slicing, array accesses, and so forth.

AutoESL uses this internal database when it synthesizes the design. Synthesis is a two-step process consisting of *scheduling* and *binding*.

About Scheduling

Scheduling is where AutoESL determines in which cycles an operation is to occur.

If, for example, two addition operations are scheduled in the same clock cycle they cannot use the same hardware adder; however, if they are scheduled in different clock cycles, they could use the same adder and save resources. In the absence of any constraints or directives AutoESL first tries to schedule the design to achieve the minimum possible latency: this could require scheduling two additions in the same clock cycle.

About Binding

When scheduling completes, binding is the process where the scheduled operations are bound to specific hardware implementations (or cores) from the technology library.

For example, a multiplication operation in the source code could be implemented by a standard combinational multiplier, while another multiplication could be implemented using a pipelined multiplier: the fact that a pipelined multiplier requires two stages would have been considered during scheduling.

In AutoESL the effects of binding (knowledge of the specific cores that are used to implement operations) are considered during the scheduling process. Figure 1, page 6 shows the process of scheduling and binding. This prevents decisions made during binding from requiring that the design be re-scheduled, preventing endless iterations.

The types of cores available during the binding process depend on the selected device. AutoESL provides a unique library for each Xilinx device and hence cores from different libraries have different delays and timing. The delays associated with each core affect which cores can be scheduled in a single clock cycle.

The created schedule depends upon to what cores the operators are bound. As such, the scheduling process takes into consideration the effects binding has on the design.

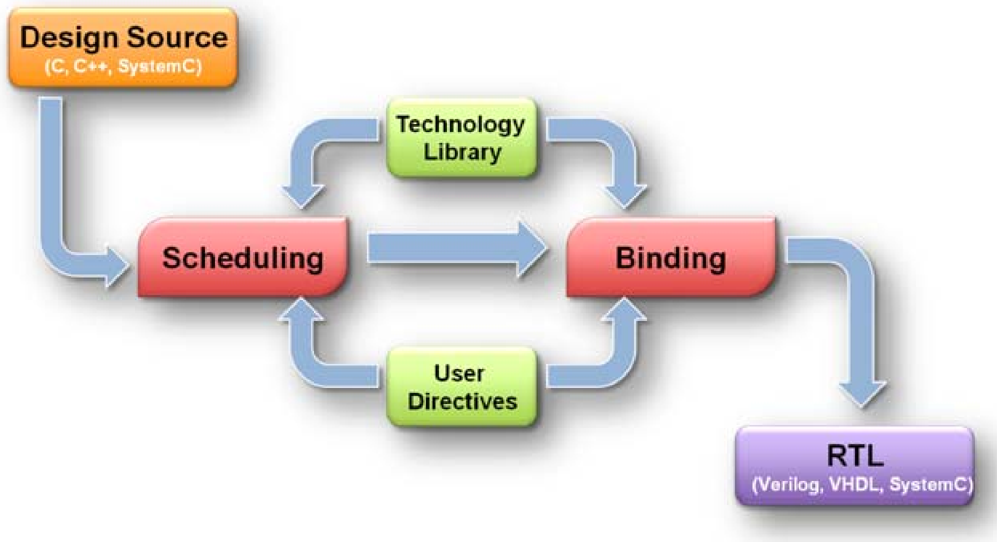

Figure 1: **Synthesis Process**

In addition to device selection, AutoESL provides a number of commands and directives that let you control the scheduling and binding process.

The process of synthesis, although straightforward to describe, is complicated in implementation, taking account of a number of factors when creating a designs, such as the:

- Operations in the code
- Design schedule
- Timing delays of the cores
- User constraints and directives
- Binding process

Understanding which cores are available for the RTL implementation is often crucial to achieving a high performance design. The following subsections describes the AutoESL operations and cores.

Understanding Operators, Cores & Directives

During High-Level Synthesis (HLS) the operations in the C, C++ or SystemC source code are identified and represented in the internal database. There are no commands to list or access the internal database; the operations can be seen in the Design Viewer that is available in the AutoESL GUI. Figure 2, page 7 shows an example from the Design Viewer.

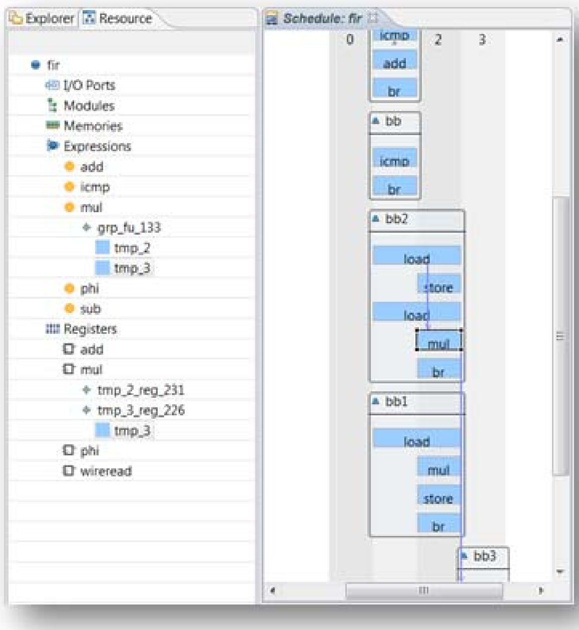

Figure 2: **Operations in the Design Viewer**

In Figure 2,a number of operations can be seen. For example, there are additions represented by an `add` operation, multiplications represented by a `mul` operation, array reads (`load` operation), comparisons (`cmp` operation) and breaks represented by a `br` operation. for information about the Design Viewer, refer to the `AutoESL_Tutorial_Introduction.pdf`.

Figure 2 also shows how the Design Viewer can show the result of operator binding. When the `mul` operation in block `bb2` is selected in the schedule viewer window (right side of the figure) it automatically cross-highlights in the resource viewer window (left side of the figure) showing this `mul` operation is bound to hardware multiplier resource and instance `grp_fu_133`. Instance `grp_fu_133` (the same instance name is used in the RTL) also shows it is used for a second multiplier operation: there are two operations inside `grp_fu_133`, `tmp2` and `tmp3`, indicating this single hardware instance is being used for two `mul` operations.

- Operations such as multiplications are implemented by a specific hardware multiplier in the RTL design using a specific core. Not all operations map to cores.
- Operations such as the break from a loop or switch statement indicate a control flow action and are implemented using logic rather than a core from the library. Operations such as these cannot be controlled by user directives.

Controlling Operators & Cores

AutoESL provides the following ways to control the use of operations and cores:

- Directing the allocation process for operations
- Directing the specific resources used for operations
- Scheduling efforts
- Control the binding process of operators to cores
- Listing details on the cores

Limiting Operators

AutoESL lets you limit how many operators are used in a design. For example, if a design called `foo` has 317 multiplications but the FPGA only has 256 multipliers, the following allocation command can be used to direct AutoESL to only create a design schedule with maximum of 256 multiplication (`mul`) operators:

```
set_directive_allocation -limit 256 -type operation foo mul
```

The `-type` option has specified `operation` because the allocation directive can be used in the same manner to limit the number of instances of cores and specific sub-functions.

Explicitly limiting the number of operators to reduce area might be required in some cases because the default operation of AutoESL is to first maximize performance:

- By default, in the absence of any constraints or directives, AutoESL tries to create a design with the lowest latency (the fewest number of cycles from input to output).
- When directives are applied to specify a maximum or minimum latency, AutoESL seeks to satisfy these latency constraints.
- When directives are applied for pipelining AutoESL first seeks to satisfy these constraints, and then minimizes or satisfies any latency constraints.

Minimizing latency can often mean using more cores in the final design, for example, using two adders in the same clock cycle rather than taking two clock cycles and sharing the same adder.

Limiting the number of operators ensures fewer cores are used in the final design and might force an increase in latency.

Table 1, page 12 lists all the operations which can be controlled using the `set_directive_allocation` command.

Controlling Resources

The `set_directive_resource` command specifies which core to use for an operation. This directive ensures the exact core is known during the scheduling process: no effort on binding is performed because this directive explicitly specifies the binding.

Listing the details on the technology library shows which operations can be implemented with each core (resource), and is explained Core Details, page 11.

The `set_directive_resource` directive is most typically used to specify which memory element is to be used to implement an array.

AutoESL can determine through analysis which cores can and should be used for each operation.

KEY CONCEPT: *For arrays, a specific memory from the technology library should be specified for each array: if none is specified AutoESL will automatically select a memory, single or dual-port, which provides the highest throughput and lowest latency.*

The resource directives uses the assigned variable as the target for the resource. 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
```

If the variable is used with multiple operators, the code must be modified to ensure there is a single variable for each operator. For example:

```
Result = (A*B) + C;
```

Should be changed to:

```
Result_tmp = (A*B);
Result = Result_tmp + C;
```

And the directive specified on `Result_tmp` to control the multiplier resource or on `Result` to control the adder resource.

Controlling Schedule

The `config_schedule` command controls the effort during scheduling and can identify the exact path when the design fails to satisfy the constraints.

Scheduling Effort

The following general information can be used for all effort level controls in AutoESL, which are: **high**, **low**, and **medium**.

With an effort level set to **high**, AutoESL uses additional CPU cycles and memory, even after satisfying the constraints, to determine if it can create an even smaller or faster design. This exploration might, or might not, result in a better quality design but it does take more time and memory to complete. For designs which are just failing to meet their goals or for designs where many different optimization combinations are possible, this could be a useful strategy.

For some designs, the nature of the code does not allow much optimization. For example, if the code says move data from one variable to the next, there are no other actions to be done, and the code just needs to be implemented: the AutoESL software, however, does not determine that spending time doing exploration will not result in much improvement, and it spends time researching more possibilities. For designs such as this, an effort level set to **low** will most likely come to the same result much faster.

In summary, it is a better practice to leave the effort level at the default **medium** setting, however:

TIP: If the design has little room for various combinations of operators and cores *and it is running a long time*, using a low effort may give the same results much faster.

TIP: If the design *is just failing to meet the require performance in area or timing*, it is worth using a high effort to see what is possible with further exploration and trials.

Critical Path Analysis

The `config_schedule` command also has a `-verbose` option. When you specify this option AutoESL prints out the full critical path when scheduling is unable to meet the constraints.

Controlling Binding

The `config_bind` command provides control over the binding process. The command lets you direct how much effort is spent when binding cores to operators and enables direct control for operator minimization in area sensitive designs.

Binding Effort

The same general guidelines for scheduling effort also apply to binding. In this case, designs with operations for which there are a large number of possible cores will benefit more from higher efforts than design were there are few choice.

The `list_core` command, described in Core Details, can be used to determine the number of possible cores with which each operator can be implemented.

Minimizing operators

You can use the `config_bind` command to force more sharing on the design.

- The `-min_op` option to the `config_bind` command instructs AutoESL to create a design with the minimum number of specified operators.

For example, the following instructs AutoESL to create a design with the minimum number of `add` operators.

```
config_bind -min_op add
```

Because this command affects the binding process it only has an impact when operators are scheduled in the different clock cycles. If the operators are scheduled in the same clock cycle to satisfy other constraints (latency and/or throughput) the `config_bind` command has no effect on these operators. The allocation directive which impacts the result of scheduling should be used to first limit the number of operators.

This command option is typically used to override any consideration of MUXing costs. When operations are shared onto the same core, the additional MUXes, which are implemented in LUTs, can have a significant impact on both timing and area. AutoESL typically defaults to being conservative with MUXes if there is a potential that it will violate timing.

Core Details

The `list_core` command is used to obtain details on the cores available in the library.

To use the `list_core` a device must be selected using the `set_part` command. If no device has been selected, the command will have no effect.

- The `-operation` option of the list core command lists all the cores in the library that can be implemented with the specified operation.

Table 1, page 12 gives a complete list of the operations which can be queried. Using the command without any option lists all the cores available for the target device..

- The `-type` option can be used to further refine the cores by category:

- **Function Units**: Cores that implement standard RTL operations (such as add, multiply, compare)
- **Storage** - Cores that implement storage elements such as registers or memories.
- **Adapter** - Cores that 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, such as the Embedded Development Kit (EDK).
- **IP Blocks** -Any IP cores added by the user.
- **Connectors** - Cores used to implement connectivity within the design. This includes direct connections and streaming storage elements.

Tables in AutoESL Cores, page 13 list the standard cores used in Xilinx devices.

AutoESL Operators

Table 1 lists the operators used by AutoESL.

The columns in the table indicate whether the operator is:

- Available for viewing in the Design Viewer
- Can be controlled by the `set_directive_allocation`, and `set_directive_resource` directives or the `config_bind` command
- If the associated cores can be listed from the library

Table 1: **AutoESL Operators**

Operator	Description	Design Viewer	Controlled by Directives	Library Core listed
add	Addition	X	X	X
ashr	Arithmetic Shift-Right	X	X	X
br	Break operation	X		
fiforead	FIFO Read	X		X
fifowrite	FIFO Write	X		X
fifonbread	Non-Blocking FIFO Read	X		X
fifonbwrite	Non-Blocking FIFO Write	X		X
icmp	Integer Compare	X	X	X
load	Memory Read	X		X
lshr	Logical Shift-Right	X	X	X
mul	Multiplication	X	X	X

Table 1: **AutoESL Operators *(Cont'd)***

Operator	Description	Design Viewer	Controlled by Directives	Library Core listed
mux	Multiplexor	X		X
phi	Multiplexor	X		
sdiv	Signed Divider		X	
shl	Shift-Left	X	X	X
srem	Signed Remainder	X		X
store	Memory Write	X		X
sub	Subtraction	X	X	X
udiv	Unsigned Division	X	X	X
urem	Unsigned Remainder	X		X
srem	Signed Remainder	X		X
wireread	IO read operation	X		
wirewrite	IO write operation	X		

AutoESL Cores

The AutoESL cores can be listed in the following categories (also used in the `list_core` command):

- Functional Unit Cores
- Storage Cores
- Connector Cores
- Adapter Cores
- Floating Point Cores
- IP Blocks

IMPORTANT: IP blocks are blocks added to the library by the user; consequently, they are not listed in this document.

The cores are explained in the Table 2 through Table 6, page 16 in the following subsections.

Functional Unit Cores

Table 2 lists the cores that implement standard RTL logic operations (such as add, multiply, and compare).

Table 2: **Functional Cores**

Core	Description
AddSub	This core is used to implement both adders and subtractors.
AddSubnS	An N-stage pipelined adder or subtractor. AutoESL will automatically determine how many pipeline stages are required.
Cmp	Comparator.
Div	Divider.
Mul	Combinational multiplier.
Mul2S	2-stage pipelined multiplier.
Mul3S	3-stage pipelined multiplier.
Mul4S	4-stage pipelined multiplier.
Mul5S	5-stage pipelined multiplier.
Mul6S	6-stage pipelined multiplier.
MulnS	N-stage pipelined multiplier. AutoESL will automatically determine how many pipeline stages are required.
Sel	Generic selection operator, typically implemented as a mux.

Storage Cores

Table 3 lists the cores that implement storage elements such as registers or memories.

Table 3: **Storage Cores**

Core	Description
FIFO	A FIFO. AutoESL will determine whether to implement this in the RTL with a BRAM or as distributed RAM.
FIFO_ BRAM	A FIFO implemented with a BRAM.
FIFO_LUTRAM	A FIFO implemented as distributed RAM.
FIFO_SRL	A FIFO implemented as with an SRL.
RAM_1P	A single-port RAM. AutoESL will determine whether to implement this in the RTL with a BRAM or as distributed RAM.
RAM_1P_BRAM	A single-port RAM, implemented with a BRAM.
RAM_1P_LUTRAM	A single-port RAM, implemented as distributed RAM.
RAM_2P	A dual-port RAM, using separate read and write ports. AutoESL will determine whether to implement this in the RTL with a BRAM or as distributed RAM.

Table 3: **Storage Cores *(Cont'd)***

Core	Description
RAM_2P_BRAM	A dual-port RAM, using separate read and write ports, implemented with a BRAM.
RAM_2P_LUTRAM	A dual-port RAM, using separate read and write ports, implemented as distributed RAM.
RAM_T2P_BRAM	A true dual-port RAM, with support for both read and write on both the input and output side, implemented with a BRAM.
RAM_2P_1S	A dual-port asynchronous RAM: implemented in LUTs.
ROM_1P	A single-port ROM. AutoESL will determine whether to implement this in the RTL with a BRAM or with LUTs.
ROM_1P_BRAM	A single-port ROM, implemented with a BRAM.
ROM_1P_LUTRAM	A single-port ROM, implemented as distributed ROM.
ROM_1P_1S	A single-port asynchronous ROM: implemented in LUTs.
ROM_2P	A dual-port ROM. AutoESL will determine whether to implement this in the RTL with a BRAM or as distributed ROM.
ROM_2P_BRAM	A dual-port ROM implemented with a BRAM.
RAM_2P_LUTRAM	A dual-port ROM implemented as distributed ROM.

Connector Cores

Table 4 lists the core used to implement connectivity within the design. This includes direct connections and streaming storage elements.

Table 4: **Connector Cores**

Core	Description
Mux	Multiplexor.

Adapter Cores

Table 5 lists the cores that 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 in EDK.

Table 5: **Adapter Cores**

Core	Description
FSL	Standard Xilinx FSL interface.
NPI64M	Native multi-port memory controller interface.
PLB46S	Standard PLB46 slave interface.
PLB46M	Standard PLB46 master interface.

Table 5: **Adapter Cores *(Cont'd)***

Core	Description
AXI4LiteS	AXI4 Lite slave interface.
AXI4M	AXI4 master interface.
AXI4Stream	AXI4 stream interface.

Floating Point Cores

AutoESL supports the following floating point cores for each Xilinx device. If no floating point core exists for an operator or function, AutoESL will not be able to synthesis the floating point operator and synthesis will halt.

Table 6: **Floating Point Cores**

Core	7 Series	Virtex-6	Virtex-5	Virtex-4	Spartan-6	Spartan-3
FAddSub	X	X	X	X	X	X
FAddSub_nodsp	X	X	X	-	-	-
FAddSub_fulldsp	X	X	X	-	-	-
FCmp	X	X	X	X	X	X
FDiv	X	X	X	X	X	X
FMul	X	X	X	X	X	X
FMul_nodsp	X	X	X	-	X	X
FMul_meddsp	X	X	X	-	X	X
FMul_fulldsp	X	X	X	-	X	X
FMul_maxdsp	X	X	X	-	X	X
FRSqrt	X	X	X	-	-	-
FRSqrt_nodsp	X	X	X	-	-	-
FRSqrt_fulldsp	X	X	X	-	-	-
FRecip	X	X	X	-	-	-
FRecip_nodsp	X	X	X	-	-	-
FRecip_fulldsp	X	X	X	-	-	-
FSqrt	X	X	X	X	X	X
DAddSub	X	X	X	X	X	X
DAddSub_nodsp	X	X	X	-	-	-
DAddSub_fulldsp	X	X	X	-	-	-
DCmp	X	X	X	X	X	X
DDiv	X	X	X	X	X	X
DMul	X	X	X	X	X	X
DMul_nodsp	X	X	X	-	X	X

Table 6: **Floating Point Cores *(Cont'd)***

Core	7 Series	Virtex-6	Virtex-5	Virtex-4	Spartan-6	Spartan-3
DMul_meddsp	X	X	X	-	-	-
DMul_fulldsp	X	X	X	-	X	X
DMul_maxdsp	X	X	X	-	X	X
DRSqrt	X	X	X	X	X	X
DRecip	X	X	X	-	-	-
DSqrt	X	X	X	-	-	-

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.