AUTOESL CODING STYLE GUIDE

UG923 (v 2012.1) April, 2012

XILINX®

Table of Contents

Preface

The preface includes the syntax conventions used in this document.

Conventions

The following conventions are used in document:

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

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

Important	Indicates an important tip, note or warning.

Table 1 Syntax conventions

Introduction

This coding style guide explains how C code (including C++ and SystemC) can be written for implementation on a Xilinx FPGA device. The first step for accomplishing this is to synthesize the C code to a register transfer level (RTL) description using AutoESL High-Level Synthesis (HLS). The RTL design is then synthesized into Xilinx gate-level primitives.

For more details on AutoESL and the complete tool flow to implementation on a Xilinx FPGA, refer to the AutoESL User Guide.

The initial chapters of this document explain the basics of C programming with AutoESL and how the various constructs in the C programming language are synthesized into a hardware implementation. Guidelines are then presented for extensions to the C language: C++ and SystemC (a class library of C++ routines used for modeling hardware behavior and available from www.accellera.org).

> Since everything which is written about the C language also applied to C++ and SystemC, the term C code is used throughout this document to imply code written in C, C++ or SystemC, unless specifically noted.

Algorithms written in C code are widely used in many applications and execute on many different targets, including standard microprocessors (CPUs), graphics processors (GPUs), microcontrollers used in real-time-operating-systems (RTOS) and digital signal processors (DSPs). In all cases the compiled C code will execute with adequate performance but for high performance operation the C code will be optimized for the target device. It is no different when the target device is a Xilinx FPGA and this document explains how code modification can improve the quality and performance of the hardware.

Coding Examples

Each of the coding examples in this document, for which there is an example number, are provided as part of the AutoESL release. The coding examples can be accessed in the following manner:

- From the `Browse Examples` option in the AutoESL start-up page.
- In the directory `examples/coding` available in the AutoESL installation area.

Each example directory has the same name as the top-level function for synthesis. The coding examples can be opened inside the AutoESL GUI or the Tcl script is provided for use at the command prompt.

The examples in this coding guide often refer to an associated header file. This header file is shown in the document for some examples but in most the header file is not shown but can be viewed in the example directory: it can typically be assumed to simply define the data types for the top-level function and test bench.

C for Synthesis

The top-level of every C program is the function `main()`. In High-Level Synthesis (HLS) any function below the level of `main()` can be synthesized. In AutoESL, the function to be synthesized is referred to as the top-level function, or design file, and any functions above this are collectively referred to as the test bench. The test bench is used to validate the behavior of the top-level function to be synthesized.

The Top-Level Design

In general it is good design practice to separate the top-level function for synthesis from the test bench and to make use of header files.

- The test bench will typically contain operations which cannot be synthesized into hardware, such as file I/O accesses to the disk.
- Header files allow definitions used in the test bench and design files to be shared and easily updated.

Example 1 shows a design where function `hier_func` calls two sub-functions, `sumsub_func` and `shift_func`, to perform addition, subtraction and shift operations. The types `din_t`, `dint_t` and `dout_t` are defined in the header file `hier_func.h` which is discussed shortly.

```
#include "hier_func.h"

int sumsub_func(din_t *in1, din_t *in2, dint_t *outSum, dint_t *outSub)
{
	*outSum = *in1 + *in2;
	*outSub = *in1 - *in2;
}

int shift_func(dint_t *in1, dint_t *in2, dout_t *outA, dout_t *outB)
{
	*outA = *in1 >> 1;
	*outB = *in2 >> 2;
}

void hier_func(din_t A, din_t B, dout_t *C, dout_t *D)
{
	dint_t apb, amb;

	sumsub_func(&A,&B,&apb,&amb);
	shift_func(&apb,&amb,C,D);
}
```

Example 1 Hierarchical Design Example

The top-level function can contain multiple sub-functions but there can only be single top-level function for synthesis. If multiple functions are to be synthesized, they must be grouped into a single top-level function.

To synthesize function `hier_func`, the file shown in Example 1 can be added to an AutoESL project as a design file and the top-level function specified as `hier_func`. As later sections will discuss, the arguments to the top-level function (`A`, `B`, `C` and `D` in Example 1) will be synthesized into RTL ports and the functions within the top-level (`sumsub_func` and `shift_func` in Example 1) will be synthesized into hierarchical blocks.

The header file for Example 1, `hier_func.h`, is shown in Example 2 and shows how macros and the use of `typedef` statements can be used to make the code more portable and readable. As will be shown in later sections, the `typedef` statement allows the types and hence the bit-width of the datapath to be easily refined for both area and performance improvements in the final FPGA implementation.

```
#ifndef _HIER_FUNC_H_
#define _HIER_FUNC_H_

#include <stdio.h>

#define NUM_TRANS 40

typedef int din_t;
typedef int dint_t;
typedef int dout_t;

void hier_func(din_t A, din_t B, dout_t *C, dout_t *D);

#endif
```

Example 2 Hierarchical Design Example Header File

The header file includes some definitions, such as NUM_TRANS, which are not required in the design file. These are used by the test bench which includes the same header file.

The Test Bench

The first step in the synthesis of any block is to validate the C function is correct. This is performed by the test bench and writing a good test bench can greatly increase designer productivity.

C functions execute orders of magnitude faster than RTL simulations. Using C to develop and validate the algorithm prior to synthesis is much more productive than developing at the RT level.

- The key to taking advantage of C development times is to have a test bench which checks the results of the function against known good (or golden) results. This allows any code changes to be quickly validated before synthesis: the algorithm is known to be correct.
- AutoESL can re-use the C test bench to verify the RTL design (no RTL test bench needs to be created when using AutoESL). If the test bench checks the

results from the top-level function, the RTL can be automatically verified by simulation.

Example 3 shows the test bench for the design shown in Example 1.

```
#include "hier_func.h"

int main() {
	// Data storage
	int a[NUM_TRANS], b[NUM_TRANS];
	int c_expected[NUM_TRANS], d_expected[NUM_TRANS];
	int c[NUM_TRANS], d[NUM_TRANS];

	//Function data (to/from function)
	int a_actual, b_actual;
	int c_actual, d_actual;

	// Misc
	int	retval=0, i, i_trans, tmp;
	FILE *fp;

	// Load input data from files
	fp=fopen("tb_data/inA.dat","r");
	for (i=0; i<NUM_TRANS; i++){
		fscanf(fp, "%d", &tmp);
		a[i] = tmp;
	}
	fclose(fp);

	fp=fopen("tb_data/inB.dat","r");
	for (i=0; i<NUM_TRANS; i++){
		fscanf(fp, "%d", &tmp);
		b[i] = tmp;
	}
	fclose(fp);

	// Execute the function multiple times (multiple transactions)
	for(i_trans=0; i_trans<NUM_TRANS-1; i_trans++){

		//Apply next data values
		a_actual = a[i_trans];
		b_actual = b[i_trans];

		hier_func(a_actual, b_actual, &c_actual, &d_actual);

		//Store outputs
		c[i_trans] = c_actual;
		d[i_trans] = d_actual;
	}

	// Load expected output data from files
	fp=fopen("tb_data/outC.golden.dat","r");
	for (i=0; i<NUM_TRANS; i++){
		fscanf(fp, "%d", &tmp);
```

```
                c_expected[i] = tmp;
        }
        fclose(fp);

        fp=fopen("tb_data/outD.golden.dat","r");
        for (i=0; i<NUM_TRANS; i++){
                fscanf(fp, "%d", &tmp);
                d_expected[i] = tmp;
        }
        fclose(fp);

        // Check outputs against expected
        for (i = 0; i < NUM_TRANS-1; ++i) {
                if(c[i] != c_expected[i]){
                        retval = 1;
                }
                if(d[i] != d_expected[i]){
                        retval = 1;
                }
        }

        // Print Results
        if(retval == 0){
                printf("    *** *** *** *** \n");
                printf("    Results are good \n");
                printf("    *** *** *** *** \n");
        } else {
                printf("    *** *** *** *** \n");
                printf("    Mismatch: retval=%d \n", retval);
                printf("    *** *** *** *** \n");
        }

        // Return 0 if outputs are correct
        return retval;
}
```

Example 3 Test Bench Example

Productive Test Bench

Example 3 highlights some of the attributes of a productive test bench.

- The top-level function for synthesis (`hier_func`) is executed for multiple transactions, as defined by macro NUM_TRANS (specified in the header file, Example 2), allowing for many different data values to be applied and verified. The test bench is only as good as the variety of tests it performs.
- The function outputs are compared against known good values. The known good values are read from a file in this example, but could also be computed as part of the test bench.
- The return value of function `main()` is set zero if the results are correctly verified and to a non-zero value if the results do not match known good values.

Note: If the test bench does not return a value of 0, the RTL verification performed by AutoESL will report a simulation failure. To take full advantage of the automatic RTL verification, check the results in the test bench and return a 0 if the test bench has verified the results are correct.

A test bench which exhibits these attributes will quickly test and validate any changes made to the C functions prior to synthesis and be re-usable at the RT level, allowing the RTL to be easily verified.

Design Files & Test Bench Files

Since AutoESL re-uses the C test bench for RTL verification it requires that the test bench and any associated files are denoted as test bench files when they are added to the AutoESL project.

Files associated with the test bench are any files accessed by the test bench and required for the test bench to operate correctly. Examples of such files are the data files `inA.dat`, `inB.dat`, etc. in Example 3: these must also be added to the AutoESL project as test bench files.

The requirement for indentifying test bench files in an AutoESL project does not impose a requirement that the design and test bench to be in separate files (although it is recommended).

The same design from Example 1 is repeated below in Example 4. The only difference is that the top-level function is renamed `hier_func2`, to differentiate the examples.

Using the same header file and test bench (other than the change from `hier_func` to `hier_func2`), the only changes required in AutoESL to synthesize function `sumsum_func` as the top-level function are:

- Set `sumsub_func` as the top-level function in the AutoESL project.
- Add the file in Example 4 as both a design file and project file: the level above `sumsub_func`, function `hier_func2`, is now part of the test bench and must be included in the RTL simulation.

Even though function `sumsub_func` is not explicitly instantiated inside the `main()` function, the remainder of the functions (`hier_func2` and `shift_func`) are confirming it is operating correctly and thus are part of the test bench.

```
#include "hier_func2.h"

int sumsub_func(din_t *in1, din_t *in2, dint_t *outSum, dint_t *outSub)
{
	*outSum = *in1 + *in2;
	*outSub = *in1 - *in2;
}

int shift_func(dint_t *in1, dint_t *in2, dout_t *outA, dout_t *outB)
{
      *outA = *in1 >> 1;
      *outB = *in2 >> 2;
}

void hier_func2(din_t A, din_t B, dout_t *C, dout_t *D)
{
      dint_t apb, amb;

      sumsub_func(&A,&B,&apb,&amb);
      shift_func(&apb,&amb,C,D);
}
```

Example 4 New Top-Level

Combining Test Bench and Design Files

It is also possible to include the design and test bench into a single design file. Example 5 has the same functionality as the Example 1 through Example 3 but everything in captured in a single file (function `hier_func` is renamed `hier_func3` simply to ensure the examples are unique).

Note: If the test bench and design are in a single file, the file must be added to an AutoESL project as both a design file and a test bench file.

```
#include <stdio.h>

#define NUM_TRANS 40

typedef int din_t;
typedef int dint_t;
typedef int dout_t;

int sumsub_func(din_t *in1, din_t *in2, dint_t *outSum, dint_t *outSub)
{
      *outSum = *in1 + *in2;
      *outSub = *in1 - *in2;
}

int shift_func(dint_t *in1, dint_t *in2, dout_t *outA, dout_t *outB)
{
      *outA = *in1 >> 1;
      *outB = *in2 >> 2;
}

void hier_func3(din_t A, din_t B, dout_t *C, dout_t *D)
{
      dint_t apb, amb;

      sumsub_func(&A,&B,&apb,&amb);
      shift_func(&apb,&amb,C,D);
}

int main() {
      // Data storage
```

```
        int a[NUM_TRANS], b[NUM_TRANS];
        int c_expected[NUM_TRANS], d_expected[NUM_TRANS];
        int c[NUM_TRANS], d[NUM_TRANS];

        //Function data (to/from function)
        int a_actual, b_actual;
        int c_actual, d_actual;

        // Misc
        int     retval=0, i, i_trans, tmp;
        FILE *fp;
        // Load input data from files
        fp=fopen("tb_data/inA.dat","r");
        for (i=0; i<NUM_TRANS; i++){
                fscanf(fp, "%d", &tmp);
                a[i] = tmp;
        }
        fclose(fp);

        fp=fopen("tb_data/inB.dat","r");
        for (i=0; i<NUM_TRANS; i++){
                fscanf(fp, "%d", &tmp);
                b[i] = tmp;
        }
        fclose(fp);

        // Execute the function multiple times (multiple transactions)
        for(i_trans=0; i_trans<NUM_TRANS-1; i_trans++){

                //Apply next data values
                a_actual = a[i_trans];
                b_actual = b[i_trans];

                hier_func3(a_actual, b_actual, &c_actual, &d_actual);

                //Store outputs
                c[i_trans] = c_actual;
                d[i_trans] = d_actual;
        }

        // Load expected output data from files
        fp=fopen("tb_data/outC.golden.dat","r");
        for (i=0; i<NUM_TRANS; i++){
                fscanf(fp, "%d", &tmp);
                c_expected[i] = tmp;
        }
        fclose(fp);

        fp=fopen("tb_data/outD.golden.dat","r");
        for (i=0; i<NUM_TRANS; i++){
                fscanf(fp, "%d", &tmp);
                d_expected[i] = tmp;
        }
        fclose(fp);
```

```
       // Check outputs against expected
       for (i = 0; i < NUM_TRANS-1; ++i) {
             if(c[i] != c_expected[i]){
                   retval = 1;
             }
             if(d[i] != d_expected[i]){
                   retval = 1;
             }
       }

       // Print Results
       if(retval == 0){
             printf("    *** *** *** *** \n");
             printf("    Results are good \n");
             printf("    *** *** *** *** \n");
       } else {
             printf("    *** *** *** *** \n");
             printf("    Mismatch: retval=%d \n", retval);
             printf("    *** *** *** *** \n");
       }

       // Return 0 if outputs are correct
       return retval;
}
```

Example 5 Test Bench and Top-Level Design

Top-Level Arguments: RTL Interface Ports

When the top-level function is synthesized the arguments (or parameters) to the function are synthesized into RTL ports. This process is called interface synthesis.

Interface Synthesis

The code shown in Example 6 can be used to provide a comprehensive overview interface synthesis. In this example, there are two pass-by-value inputs, (`in1` and `in2`), a pointer (`sum`) which is both read from and written to, and a function return (the value of `temp`).

```
#include "sum_io.h"

dout_t sum_io(din_t in1, din_t in2, dio_t *sum) {

       dout_t temp;

       *sum = in1 + in2 + *sum;
       temp = in1 + in2;

       return  temp;
}
```

Example 6 Interface Synthesis Example

By default, the design will be synthesized into an RTL block with the ports shown in Figure 1.

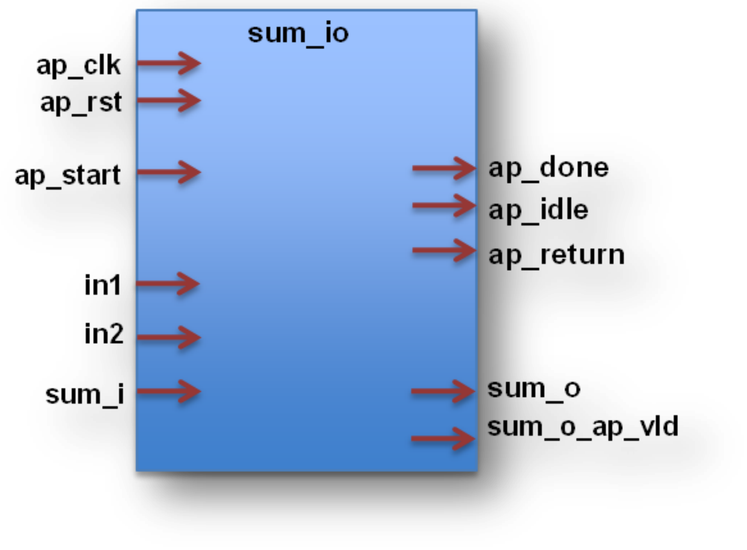


Figure 1 RTL Ports after Default Interface Synthesis

An explanation of the ports is as follows:

- A clock and reset port are added to the design.
- AutoESL adds design-level handshake signals by default: ports `ap_start`, `ap_done` and `ap_idle`.
- If the function has a return value, output port `ap_return` is added to the RTL interface.
- Function arguments which are both read from and written to, are synthesized as separate input and output ports (`sum_i` and `sum_o` in Figure 1).
- By default, input pass-by-value arguments and pointers will be synthesized as simple wire ports with no associated handshaking signal.
- By default, output pointers will be synthesized with an associated output valid signal to indicate when the output data is valid.

When AutoESL synthesizes the RTL ports it automatically creates the necessary hardware to read and write to the ports: whether it takes a single cycle or multiple cycles. For the code shown in Example 6 the timing behavior would be as shown in Figure 2 (assuming the target technology and clock frequency allow a single addition per clock cycle).

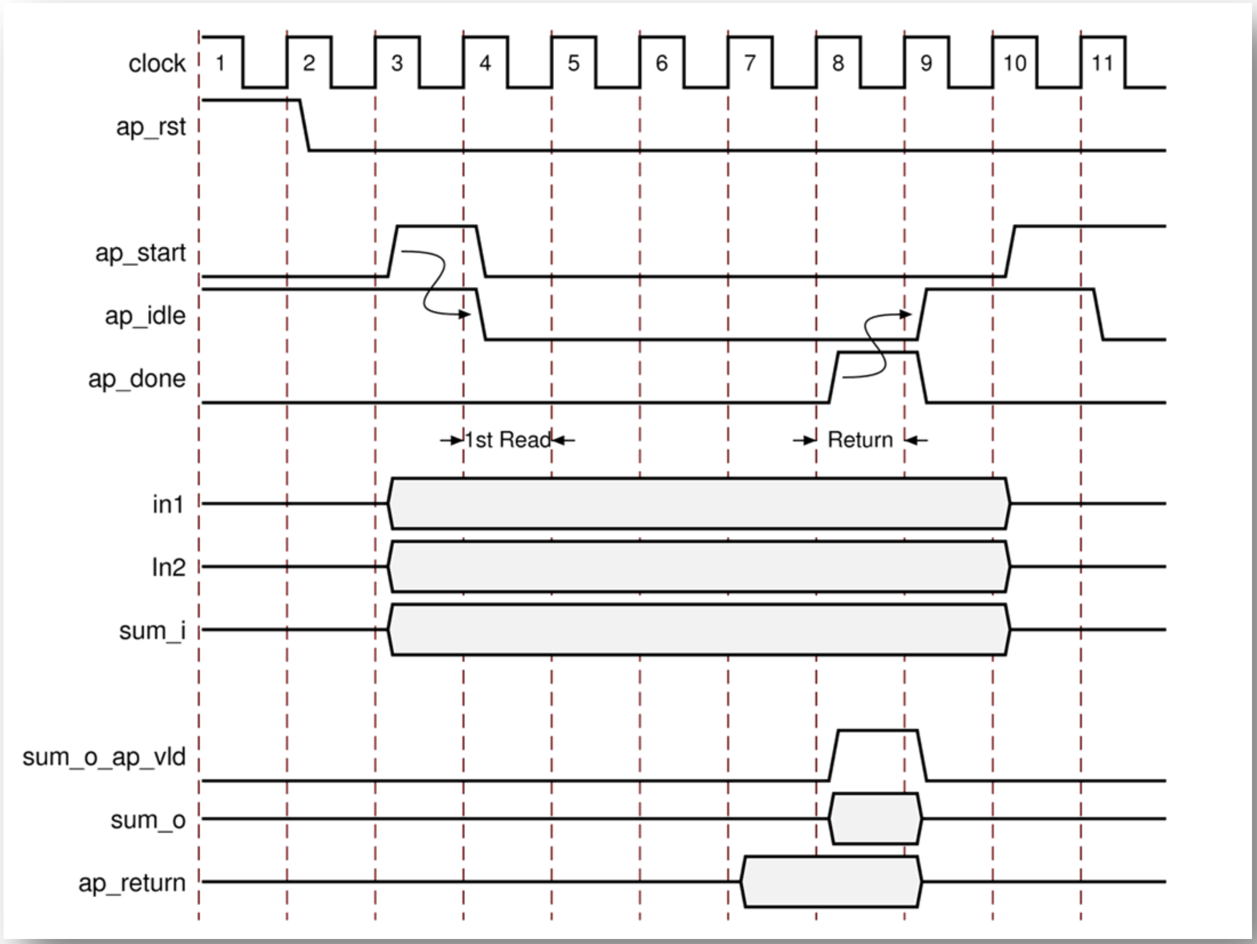


Figure 2 RTL Port Timing with Default Synthesis

- The design starts when `ap_start` is asserted high.
- The `ap_idle` signal is asserted low to indicate the design is operating.
- The input data is read at any clock after the first cycle (AutoESL will automatically schedule when the reads occur).
- When output `sum` is calculated, the associated output handshake (`sum_o_ap_vld`) indicates the data is valid.
- When the function completes, `ap_done` is asserted: this also indicates the data on `ap_return` is valid.
- Port `ap_idle` is asserted high to indicate the design is waiting start again.

A complete explanation of interface synthesis and the various options is provided in the `AutoESL User Guide`. The important points to understand here are:

- Interface synthesis automatically handles the data sequencing to and from the design: the user simply needs to select the appropriate interface.
- Many types of interfaces can be synthesized: wire ports, single and two-way handshakes, RAM access ports and FIFO ports among others.

- Many different types of interface can be synthesized from the same source code.

If the same code is synthesized with `in1`, `in2` and `sum` specified as two-way handshakes, the RTL ports would be as shown in Figure 3.

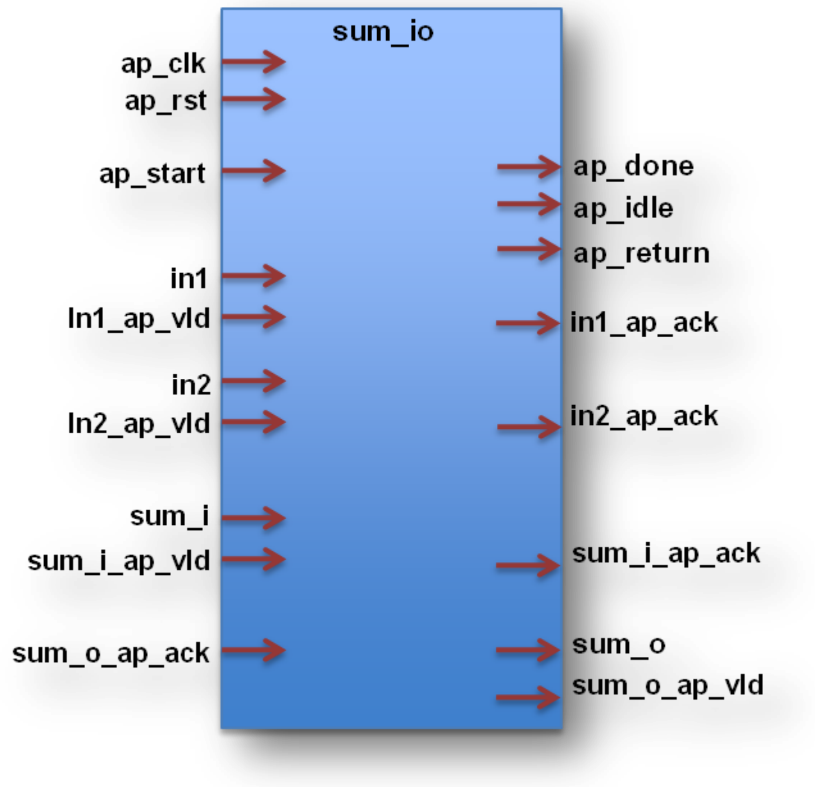


Figure 3 RTL Ports after Specified Interface Synthesis

With the concept of interface synthesis understood, the remainder of this section will discuss issues related to the how the coding style can influence the implementation of RTL ports.

Pointers

Pointers can be used as arguments to the top-level function. It is important to understand how pointers are implemented during synthesis as they can sometimes introduce issues in achieving the desired RTL interface and design after synthesis.

Basic Pointers

A function with basic pointers on the top-level interface, such as shown in Example 7, produces no issues for HLS. The pointer can be synthesized to either a simple wire interface or an interface protocol using handshakes.

Note: To be synthesized as a FIFO interface, a pointer must be read-only or write-only.

```
#include "pointer_basic.h"
```

```
void pointer_basic (dio_t *d) {
       static dio_t acc = 0;

       acc += *d;
       *d  = acc;
}
```

Example 7 Basic Pointer Interface

When used with the test bench shown here in Example 8.

```
#include "pointer_basic.h"

int main () {
       dio_t d;
       int i, retval=0;
       FILE        *fp;

       // Save the results to a file
       fp=fopen("result.dat","w");
       printf(" Din Dout\n");

       // Create input data
       // Call the function to operate on the data
       for (i=0;i<4;i++) {
              d = i;
              pointer_basic(&d);
              fprintf(fp, "%d \n", d);
              printf("  %d   %d\n", i, d);
       }
       fclose(fp);

       // Compare the results file with the golden results
       retval = system("diff --brief -w result.dat result.golden.dat");
       if (retval != 0) {
              printf("Test failed!!!\n");
              retval=1;
       } else {
              printf("Test passed!\n");
       }

       // Return 0 if the test
       return retval;
}
```

Example 8 Basic Pointer Interface Test Bench

The C function and RTL simulation will verify the correct operation (although not all possible cases with this simple data set).

```
Din Dout
  0   0
  1   1
  2   3
  3   6
Test passed!
```

Pointer Arithmetic

When pointer arithmetic is introduced it limits the possible interfaces which can be synthesized in RTL. Example 9 shows the same code but this time some simple pointer arithmetic is used to accumulate the data values (starting from the 2nd value).

```
#include "pointer_arith.h"

void pointer_arith (dio_t *d) {
        static int acc = 0;
        int i;

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

Example 9 Interface with Pointer Arithmetic

Example 10 shows the test bench which supports this example. Since the loop to perform the accumulations is now inside function `pointer_arith`, the test bench simply populates the address space, specified by array `d[5]`, with the appropriate values.

```
#include "pointer_arith.h"

int main () {
        dio_t d[5], ref[5];
        int i, retval=0;
        FILE        *fp;

        // Create input data
        for (i=0;i<5;i++) {
                d[i] = i;
                ref[i] = i;
        }

        // Call the function to operate on the data
        pointer_arith(d);

        // Save the results to a file
        fp=fopen("result.dat","w");
        printf(" Din Dout\n");
        for (i=0;i<4;i++) {
                fprintf(fp, "%d \n", d[i]);
                printf("  %d   %d\n", ref[i], d[i]);
        }
        fclose(fp);
        // Compare the results file with the golden results
        retval = system("diff --brief -w result.dat result.golden.dat");
        if (retval != 0) {
                printf("Test failed!!!\n");
                retval=1;
        } else {
                printf("Test passed!\n");
        }

        // Return 0 if the test
        return retval;
}
```

Example 10 Test Bench for Pointer Arithmetic Function

When simulated, this results in the following output.

```
Din Dout
  0   1
  1   3
  2   6
  3   10
Test passed!
```

The problem with the pointer arithmetic is that it does not access the pointer data in sequence. Wire, handshake or FIFO interfaces have no way of accessing data out of order:

- A wire interface will read data when the design is ready to consume the data or write the data when the data is ready.
- Handshake and FIFO interfaces will read and write when the control signals permit the operation to proceed.

In both cases, the data must arrive (and is written) in order, starting from element zero. In Example 9 the code states the first data value read will be from index 1 (`i` starts at 0, 0+1=1): this is the 2nd element from array `d[5]` in the test bench.

When this is implemented in hardware, this will require some form of data indexing. This is not supported with wire, handshake or FIFO interfaces. The code in Example 9 can only be synthesized with an `ap_bus` interface: this interface supplies an address with which to index the data when the data is accessed (read or write).

Alternatively the code must be modified as shown in Example 11, with an array on the interface instead of a pointer. This can be implemented in synthesis with a RAM (`ap_memory`) interface, which has the capability of indexing the data with an address and can perform out-of-order, or non-sequential accesses.

Wire, handshake or FIFO interfaces can only be used on streaming data and therefore cannot be used in conjunction with pointer arithmetic (unless it indexes the data starting at zero and then proceeds sequentially).

More details on the `ap_bus` and `ap_memory` interface types are available in the AutoESL User Guide and AutoESL Reference Guide.

```
#include "array_arith.h"

void array_arith (dio_t d[5]) {
	static int acc = 0;
	int i;

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

Example 11 Array Arithmetic

Multi-Access Pointer Interfaces: Streaming Data

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

The issues which arise are that:

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

Note: Issues with multiple pointer accesses are generally seen in designs which are written and then synthesized without C validation (simulation) being performed.

The following will help demonstrate why a good methodology should always include validation that the C source gives the expected behavior prior to synthesis.

In the following example, input pointer `d_i` is read from four times and output `d_o` is written to twice, with the intent that the accesses will be implemented by FIFO interfaces (streaming data into and out of the final RTL implementation).

```
#include "pointer_stream_bad.h"

void pointer_stream_bad ( dout_t *d_o,  din_t *d_i) {
	din_t acc = 0;

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

Example 12 Multi-Access Pointer Interface

The test bench to verify this design is shown in Example 13.

```
#include "pointer_stream_bad.h"

int main () {
	din_t d_i;
	dout_t d_o;
	int retval=0;
	FILE *fp;

	// Open a file for the output results
	fp=fopen("result.dat","w");

	// Call the function to operate on the data
	for (d_i=0;d_i<4;d_i++) {
		pointer_stream_bad(&d_o,&d_i);
		fprintf(fp, "%d %d\n", d_i, d_o);
	}
	fclose(fp);

	// Compare the results file with the golden results
	retval = system("diff --brief -w result.dat result.golden.dat");
	if (retval != 0) {
		printf("Test failed  !!!\n");
		retval=1;
	} else {
		printf("Test passed !\n");
	}

	// Return 0 if the test
	return retval;
}
```

Example 13 Multi-Access Pointer Test Bench

Understanding Volatile Data

The code in Example 12 is written with intent that input pointer `d_i` and output pointer `d_o` will be implemented in RTL as FIFO (or handshake) interfaces which will ensure:

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

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

AutoESL matches the behavior of the `gcc` compiler and optimizes these reads and writes into a single read operation and a single write operation. When the RTL is examined, there will only be a single read and write operation on each port.

The fundamental issue with this design is that the test bench and design do not adequately model how the designer expects the RTL ports to be implemented:

- The designer expects RTL ports which read and write multiple times during a transaction (and can stream the data in and out).
- The test bench only supplies a single input value and only returns a single output value.

A C simulation of Example 12 would show the following results, which demonstrates each input is being accumulated 4 times, but it's the same value being read once and accumulated each time: not 4 separate reads.

```
Din Dout
0   0
1   4
2   8
3   12
```

This design can be made read and write to the RTL ports multiple times by using the `volatile` qualifier as shown below in Example 14.

The `volatile` qualifier tells the C compiler, and AutoESL, to make no assumptions about the pointer accesses: the data is volatile, may change and pointer accesses should not be optimized.

```
#include "pointer_stream_better.h"

void pointer_stream_better ( volatile dout_t *d_o,  volatile din_t *d_i) {
      din_t acc = 0;

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

Example 14 Multi-Access Volatile Pointer Interface

Example 14 will simulate the same as Example 12 but the `volatile` qualifier will prevent pointer access optimizations and result in an RTL design which will perform the expected four reads on input port `d_i` and two writes to output port `d_o`.

However, even if the `volatile` keyword is used, this coding style (accessing a pointer multiple times) still has an issue in that the function and test bench do not adequately model multiple distinct reads and writes.

In this case, four reads will be performed, but again, the same data will be read four times. There will be two separate writes, each with the correct data but the test bench will only capture data for the final write. (The intermediate accesses can be seen by enabling autosim to create a trace file during RTL simulation and viewing the VCD file).

> **Note**: Example 14 can be implemented with wire interfaces, but if a FIFO interface is specified AutoESL will create an RTL test bench to stream new data on each read: since there is no new data available from the test bench, the RTL will fail to verify.

The issue here is that the test bench does not correctly model the reads and writes correctly.

Modeling Streaming Data Interfaces

Unlike software, the concurrent nature of hardware systems allows them to take advantage of streaming data, where data is continuously supplied to the design and the design continuously outputs data: an RTL design can accept new data before the design has finished processing the existing data.

As the previous example has shown, modeling streaming data in software is non-trivial, especially when writing software to model an existing hardware implementation (where the concurrent/streaming nature already exists and needs to be modeled).

There are a number of approaches which can be taken here:

- Simply add the `volatile` qualifier as shown in Example 14. The test bench will not model unique reads and writes and RTL simulation using the original C test bench may fail, but viewing the VCD waveforms will show the correct reads and writes are being performed.
- Modify the code to model explicit unique reads and writes. This is shown next in Example 15.
- Modify the code to using a streaming data type. A streaming data type allows hardware using streaming data to be accurately modeled. This is discussed in the AutoESL User Guide.

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

```
#include "pointer_stream_good.h"

void pointer_stream_good ( volatile dout_t *d_o,  volatile din_t *d_i) {
      din_t acc = 0;
```

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

Example 15 Explicit Multi-Access Volatile Pointer Interface

The test bench is updated to model the fact that the function will read four unique values in each transaction. This new test bench only models a single transaction: to model multiple transactions, the input data set would need to be increased to and the function called multiple times.

```
#include "pointer_stream_good.h"

int main () {
  din_t d_i[4];
  dout_t d_o[4];
        int i, retval=0;
        FILE        *fp;

        // Create input data
        for (i=0;i<4;i++) {
                d_i[i] = i;
        }

        // Call the function to operate on the data
        pointer_stream_good(d_o,d_i);

        // Save the results to a file
        fp=fopen("result.dat","w");
        for (i=0;i<4;i++) {
                if (i<2)
        fprintf(fp, "%d %d\n", d_i[i], d_o[i]);
                else
        fprintf(fp, "%d \n", d_i[i]);
        }
        fclose(fp);

        // Compare the results file with the golden results
        retval = system("diff --brief -w result.dat result.golden.dat");
        if (retval != 0) {
                printf("Test failed  !!!\n");
                retval=1;
        } else {
                printf("Test passed !\n");
        }

        // Return 0 if the test
        return retval;
}
```

Example 16 Explicit Multi-Access Volatile Pointer Test Bench

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

```
Din Dout
0   1
1   6
2
3
```

The final issue to be aware of when pointers are accessed multiple time at the function interface is RTL simulation modeling.

Multi-Access Pointers and RTL simulation

To verify the RTL with `autosim`, AutoESL creates a SystemC wrapper with adapters around the RTL and instantiates this wrapper into the existing C test bench, as shown in Figure 4.

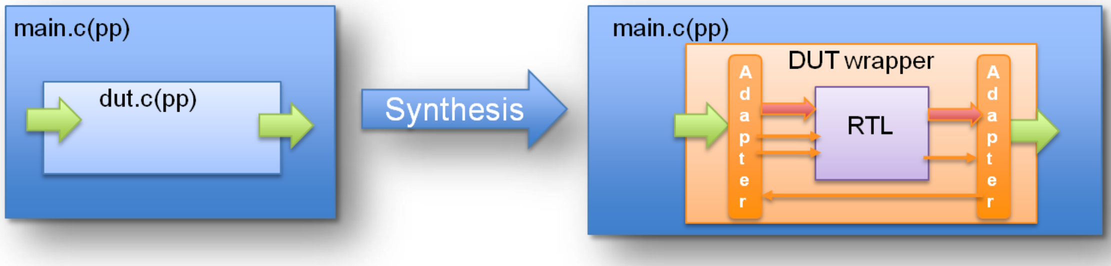


Figure 4 Autosim Wrapper Overview

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

When pointers on the interfaced are accessed multiple times, to read or write, AutoESL cannot determine from the function interface how many reads or writes are performed. Neither of the arguments in the function interface informs AutoESL how many values will be read or written.

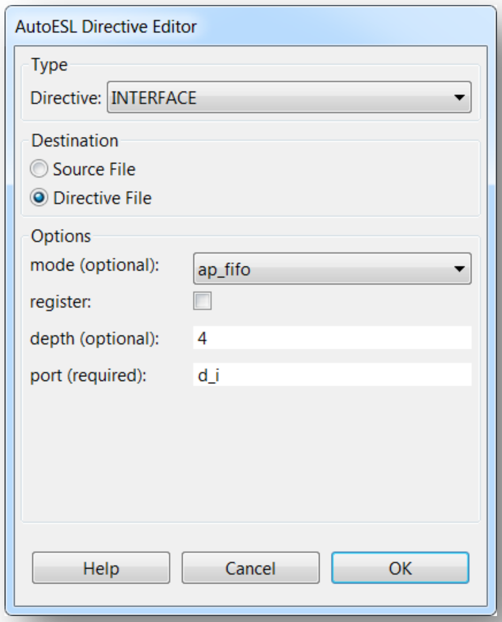
```
void pointer_stream_good ( volatile dout_t *d_o,  volatile din_t *d_i)
```

Example 17 Volatile Pointer Interface

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

If the RTL ports are actually reading or writing multiple values, this will result in the RTL `autosim` simulation stalling: the wrapper is modeling the producer and consumer blocks which will be connected to the RTL design, and if it only models a single value the RTL design will stall when trying to read or write more than one value (since there is currently no value to read or no space to write).

When multi-access pointers are used at the interface, AutoESL must be informed of the maximum number of reads or writes on the interface. When specifying the interface, use the depth option on the INTERFACE directive as shown in Figure 5.


AutoESL Directive Editor

Type

Directive: INTERFACE

Destination

( ) Source File

(●) Directive File

Options

mode (optional): ap_fifo

register: ☐

depth (optional): 4

port (required): d_i

Figure 5 Interface Directive Dialog: Depth Option

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

Arrays on the Interface

In HLS, arrays are synthesized into memory elements by default. When an array is used as an argument to the top-level function the memory is assumed to be "off-chip" and interface ports are synthesized to access the memory.

AutoESL has a rich feature set to configure how these ports are created.

- The memory can be specified as a single or dual port RAM.
- The interface can be specified as a FIFO interface.
- AutoESL array optimization directives (`partition`, `map` and `reshape`) can be used to re-configure the structure of the array and hence number of IO ports.

The primary issue which arrays on the interface will introduce into a design is creating a performance bottleneck because access to the data is limited through a memory (RAM or FIFO) port. These issues can typically be over-come with the use of directives.

The main rule for using arrays in synthesizable code, it is that arrays must be sized. If for example, the declaration `d_i[4]` in Example 18 is changed to `d_i[]`, AutoESL will issue a message that the design cannot be synthesized.

```
@E [SYNCHK-61] array_RAM.c:52: unsupported memory access on variable 'd_i'
which is (or contains) an array with unknown size at compile time.
```

Array Optimization Directives

The `resource` directive can be used to explicitly specify which type of RAM is used, and hence which RAM ports are created: single-port or dual-port. If no resource is specified AutoESL will use a single-port RAM by default and automatically use a dual-port RAM if it improves throughput or reduces latency.

The `partition`, `map` and `reshape` directives can be used to re-configure arrays on the interface. Arrays can be partitioned into multiple smaller arrays, each implemented with its own interface. This includes the ability to partition every element of the array into its own scalar element: on the function interface, this results in a unique port for every element in the array. This provides maximum parallel access but creates many more ports and may introduce routing issues in the hierarchy above.

Similarly, smaller arrays may be combined into a single larger array, resulting in a single interface. This may map better to an "off-chip" BRAM but keep in mind it may also introduce a performance bottleneck. These trade-offs can be made using AutoESL optimization directives and do not impact coding.

RAM interfaces

The array arguments in the function shown in Example 18 will, by default, be synthesized into a single-port RAM interface.

```
#include "array_RAM.h"

void array_RAM (dout_t d_o[4], din_t d_i[4], didx_t idx[4]) {
	int i;

	For_Loop: for (i=0;i<4;i++) {
		d_o[i] = d_i[idx[i]];
	}

}
```

Example 18 RAM Interface

A single-port RAM interface will be used because the for-loop ensures only one element can be read and written in each clock cycle: there is no advantage in using a dual-port RAM interface.

If the for-loop is unrolled, AutoESL will automatically use a dual-port RAM because doing so will allow multiple elements to be read at the same time and increase the throughput. The type of RAM interface can be explicitly set by applying the `resource` directive.

As mentioned earlier, if there are issues related to arrays on the interface, they are typically related to throughput and can be handled with optimization directives. For example, if the arrays in Example 18 are partitioned into individual elements and the for-loop unrolled, all four elements in each array will be accessed simultaneously.

FIFO Interfaces

AutoESL allows array arguments to be implemented as FIFO ports in the RTL. If a FIFO ports is to be used, be sure that the accesses to and from the array ports are sequential. AutoESL will perform analysis to confirm if the accesses are sequential.

- If AutoESL can verify the accesses are sequential it will implement a FIFO port.
- If AutoESL can determine the accesses are not sequential it will issue an error and synthesis will halt.
- If AutoESL cannot determine if the accesses are sequential it will issue a warning and proceed with the implementation of a FIFO port. If the accesses are in fact not sequential it will result in an RTL simulation mismatch.

Example 19 shows a case where AutoESL cannot determine if the accesses are sequential. In this example, both `d_i` and `d_o` are specified to be implemented with a FIFO interface during synthesis.

```
#include "array_FIFO.h"

void array_FIFO (dout_t d_o[4], din_t d_i[4], didx_t idx[4]) {
	int i;

	// Breaks FIFO interface d_o[3] = d_i[2];
	For_Loop: for (i=0;i<4;i++) {
		d_o[i] = d_i[idx[i]];
	}
}
```

Example 19 Streaming FIFO Interface

In this case, it is the behavior of variable `idx` which determines if a FIFO interface can be successfully created or not.

- If `idx` is incremented sequentially a FIFO interface can be created.
- If random values are used for `idx`, a FIFO interface will fail when implemented in RTL.

Since there is a possibility that this interface may work, AutoESL will issue a message during synthesis and proceed to create a FIFO interface.

```
@W [XFORM-124] Array 'd_i': may have improper streaming access(es).
```

If the comments in Example 19 are removed, ("//Breaks FIFO interface") AutoESL can automatically determine the accesses to the arrays are not sequential and will halt with an error message if a FIFO interface is specified.

Note: FIFO ports cannot be synthesized for arrays which are read from and written to: separate input and output arrays (as in Example 19) must be created.

The following general rules apply to arrays which are to be streamed (implemented with a FIFO interface):

- The array must be written/read in only one loop or function. This can be transformed into a point to point connection which matches the characteristics of FIFO links.
- The array reads must be in the same order as the array write. Random access is not supported for FIFO channels, so the array has to be used in the program following first in first out semantics.
- The index used to read and write from the FIFO has to be analyzable at compile time. Array addressing based on runtime computations cannot be analyzed for FIFO semantics and will prevent the tool from converting an array into a FIFO.

Code changes are generally not required to implement or optimize arrays in the top-level interface. The only time arrays on the interface may need coding changes is when the array is part of a struct.

Structs on the Interface

When structs are used as arguments to the top-level function, the ports created by synthesis depend on whether the struct is a pass-by-value argument or a pointer.

In this design example, struct `data_t` is defined in the header file shown in Example 20. This struct has two data members:

- An unsigned vector `A` of type `short` (16-bit).
- An array `B` of four `unsigned char` types (8-bit).

```
typedef struct {
      unsigned short A;
      unsigned char B[4];
      }     data_t;

data_t struct_port(data_t i_val, data_t *i_pt, data_t *o_pt);
```

Example 20 Struct Declaration in Header file

In Example 21 the struct is used as both a pass-by-value argument (from `i_val` to the return of `o_val`) and as a pointer (`*i_pt` to `*o_pt`). Although both methods provide a similar result - passing the input to the output after an addition operation

- the difference is how the pass-by-value and pointer arguments are synthesized as ports.

```
#include "struct_port.h"

data_t struct_port(
       data_t  i_val,
       data_t  *i_pt,
       data_t  *o_pt
       ) {

       data_t  o_val;
       int i;

       // Transfer pass-by-value structs
       o_val.A = i_val.A+2;
       for (i=0;i<4;i++) {
              o_val.B[i] = i_val.B[i]+2;
       }

       // Transfer pointer structs
       o_pt->A = i_pt->A+3;
       for (i=0;i<4;i++) {
              o_pt->B[i] = i_pt->B[i]+3;
       }

       return o_val;
}
```

Example 21 Struct as Pass-by-Value and Pointer

In the case of the pass-by-value input arguments, the arrays in the struct will be completely partitioned into separate elements:

- Struct element A will result in a 16-bit port.
- Struct element B will result in 4 separate 8-bit ports.

For the pointers and the function return, any arrays in the struct will be synthesized in the same manner as standard arrays and will result in memory interface:

- Struct element A will result in a 16-bit port.
- Struct element B will result in a RAM port, accessing 4 elements.

When using structs with large arrays, it may be an advantage to convert any pass-by-value structs to pointers otherwise such arrays will be completely partitioned into individual elements, each implemented with their own port. For example, if the array contains 1024 elements, it will be implemented with 1024 separate RTL ports.

There are no limitation in the size or complexity of structs which can be synthesized by AutoESL. There can be as many array dimensions and as many members in a struct required. The only limitation with the implementation of structs is when arrays are to be implemented as streaming (such as a FIFO interface). In this case, the same general rules which apply to arrays on the interface should be followed (FIFO Interfaces).

Types

The data types used in a C function compiled into an executable impact the accuracy of the result, the memory requirements and can impact the performance.

- A 32-bit integer `int` data type can hold more data and hence provide more precision than an 8-bit `char` type but obviously requires more storage.
- If 64-bit bit `long long` types are used on a 32-bit system the run time will be impacted as it will typically require multiple accesses to read and write such values.

Similarly, when the C function is to be synthesized to an RTL implementation the types impact the precision, the area and the performance of the RTL design: the data types used for variables determine the size of the operators required and hence the area and performance of the RTL.

AutoESL supports the synthesis of all standard C types including exact-width integer types.

- (unsigned) char, (unsigned) short, (unsigned) int
- (unsigned) long, (unsigned) long long
- (unsigned) intN_t (where N is 8,16,32 and 64, as defined in `stdint.h`)
- float, double

Exact-width integers types are useful for ensuring designs are portable across all types of system.

Note: Integer type `(unsigned)long` is implemented as 64-bit on 64-bit operating systems and as 32-bit on 32-bit operating systems. Synthesis matches this behavior and will produce different sized operators, and hence different RTL designs, depending on the type of operating system on which AutoESL is run.

Data type `(unsigned)int` or `(unsigned)int32_t` should be used instead of type `(unsigned)long` for 32-bit.

Data type `(unsigned)long long` or `(unsigned)int64_t` should be used instead of type `(unsigned)long` for 64-bit.

Standard Types

Example 22 shows some basic arithmetic operations being performed.

```
#include "types_standard.h"

void types_standard(din_A  inA, din_B  inB, din_C  inC, din_D  inD,
             dout_1 *out1, dout_2 *out2, dout_3 *out3, dout_4 *out4
  ) {
       // Basic arithmetic operations
       *out1 = inA * inB;
       *out2 = inB + inA;
       *out3 = inC / inA;
       *out4 = inD % inA;

}
```

Example 22 Basic Arithmetic

The data types in Example 22 are defined in the header file `types_standard.h` shown in Example 23 and show how standard signed types, unsigned types, and with the inclusion of header file `stdint.h`, exact-width integer types can be used.

```
#include <stdio.h>
#include <stdint.h>

#define N 9

typedef char din_A;
typedef short din_B;
typedef int din_C;
typedef long long din_D;

typedef int dout_1;
typedef unsigned char dout_2;
typedef int32_t dout_3;
typedef int64_t dout_4;

void types_standard(din_A inA,din_B inB,din_C inC,din_D inD,dout_1
*out1,dout_2 *out2,dout_3 *out3,dout_4 *out4);
```

Example 23 Basic Arithmetic Type Definitions

These different types result in the following operator and port sizes after synthesis:

- The multiplier used to calculate result `out1` will be a 24-bit multiplier: an 8-bit `char` type multiplied by a 16-bit `short` type requires a 24-bit multiplier. This result will be sign-extended to 32-bit to match the output port width.
- The adder used for `out2` will be 8-bit: since the output is an 8-bit `unsigned char` type, only the bottom 8-bits of `inB` (a 16-bit `short)` are added to 8-bit `char` type `inA`.
- For output `out3` (32-bit exact width type), 8-bit `char` type `inA` is sign-extended to 32-bit value and a 32-bit division operation is performed with the 32-bit (`int` type) `inC` input.
- Similarly, a 64-bit modulus operation is performed using the 64-bit `long long` type `inD` and 8-bit `char` type `inA` sign-extended to 64-bit, to create a 64-bit output result `out4`.

As the result of `out1` indicates, AutoESL will use the smallest operator it can and simply extend the result to match the required output bit-width. Similarly, for result out2 even though one of the inputs is 16-bit, an 8-bit adder can be used because only an 8-bit output is required. However, as the results for out3 and out4 demonstrate, if all bits are required, a full sized operator will be synthesized.

Floats and Doubles

AutoESL supports float and double types for synthesis. Both data types are synthesized with IEEE-754 standard compliance.

- Single-precision 32 bit: 24-bit fraction, 8-bit exponent
- Double-precision 64 bit: 53-bit fraction, 11-bit exponent

Example 24 show the header file used with Example 22 updated to define the data types to be double and float types.

```
#include <stdio.h>
#include <stdint.h>
#include <math.h>

#define N 9

typedef double din_A;
typedef double din_B;
typedef double din_C;
typedef float din_D;

typedef double dout_1;
typedef double dout_2;
typedef double dout_3;
typedef float dout_4;

void types_float_double(din_A inA,din_B inB,din_C inC,din_D inD,dout_1
*out1,dout_2 *out2,dout_3 *out3,dout_4 *out4);
```

Example 24 Float and Double Types

This updated header file is used with Example 25, where a sqrtf() function is used.

```
#include "types_float_double.h"

void types_float_double(
      din_A  inA,
      din_B  inB,
      din_C  inC,
      din_D  inD,
      dout_1 *out1,
      dout_2 *out2,
      dout_3 *out3,
      dout_4 *out4
      ) {

      // Basic arithmetic & math.h sqrtf()
      *out1 = inA * inB;
```

```
        *out2 = inB + inA;
        *out3 = inC / inA;
        *out4 = sqrtf(inD);

}
```

Example 25 Use of Floats and Doubles

When Example 25 is synthesized it will result in 64-bit double-precision multiplier, adder and divider operators: these operators will be implemented by the appropriate floating-point Xilinx CORE Generator cores.

The square-root function used `sqrtf()` will be implemented using a 32-bit single-precision floating-point core. It is worth noting that if the double-precision square-root function `sqrt()` was used, it would result in additional logic to cast to and from the 32-bit single-precision float types used for `inD` and `out4`: `sqrt()` is a double-precision (`double`) function, while `sqrtf()` is a single precision (`float`) function.

Note: In C functions, be careful when mixing float and double types as float-to-double and double-to-float conversion units will be inferred in the hardware.

This code:

```
float foo_f    = 3.1459;
float var_f = sqrt(foo_f);
```

Would result in the following hardware:

```
        wire(foo_t)
        ➔ Float-to-Double Converter unit
        ➔ Double-Precision Square Root unit
        ➔ Double-to-Float Converter unit
        ➔ wire (var_f)
```

Using a `sqrtf()` function would remove the need for the type converters in hardware, save area and improve timing.

For float and double types to be synthesized, there must be an associated Xilinx CORE Generator core available to implement the operation. Table 2 shows the cores available for each Xilinx family.

The implications from the cores shown in Table 2 are:

- The synthesis of floating point algorithms such as linear algebra algorithms which use the standard add, subtract, multiply, divide, and compare operations is fully supported.
- Only the square root, inverse square root and reciprocal functions in the `math.h` library are supported for synthesis in C.

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
DMul_meddsp	X	X	X	-	-	-

Core	7-Series	Virtex 6	Virtex 5	Virtex 4	Spartan 6	Spartan 3
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	-	-	-

Table 2 Floating Point Cores

The cores in Table 2 allow the operation, in some cases, to be implemented with a core in which many DSP48s are used or none (e.g. `DMul_nodsp` and `DMul_maxdsp`). By default, AutoESL will implement the operation using the core with the maximum number of DSP48s. Alternatively, the AutoESL resource directive can be used to specify exactly which core should be used.

A final consideration to be aware of when synthesizing `float` and `double` types is that AutoESL will maintain the order of operations performed in the C code to ensure the results are the same as the C simulation. Due to saturation and truncation, the following are not guaranteed to be the same in single and double precision operations:

```
        A=B*C;          A=B*F;
        D=E*F;          D=E*C;
        O1=A*D          O2=A*D;

With float and double types, O1 and O2 are not guaranteed to be the same.
```

Note: In some cases (design dependent), optimizations such as unrolling or partial unrolling of loops, may not be able to take full advantage of parallel computations as AutoESL will maintain the strict order of the operations when synthesizing `float` and `double` types.

For C++ designs, AutoESL provides a bit-approximate implementation of the most commonly used math functions. Refer to section Synthesis of C++ Math Functions in the C++ chapter.

Composite Types

Composite data types are supported for synthesis:

- array
- enum
- struct
- union

Example 26 shows a header file which first defines some `enum` types, uses them in a `struct`, which is in turn used in another `struct`, allowing an intuitive description of a complex type to be captured.

In addition, Example 26 shows how a complex define (MAD_NSBSAMPLES) statement can be specified and synthesized.

```
#include <stdio.h>

enum mad_layer {
      MAD_LAYER_I   = 1,
      MAD_LAYER_II  = 2,
      MAD_LAYER_III = 3
};

enum mad_mode {
      MAD_MODE_SINGLE_CHANNEL = 0,
      MAD_MODE_DUAL_CHANNEL = 1,
      MAD_MODE_JOINT_STEREO = 2,
      MAD_MODE_STEREO = 3
};

enum mad_emphasis {
      MAD_EMPHASIS_NONE = 0,
      MAD_EMPHASIS_50_15_US = 1,
      MAD_EMPHASIS_CCITT_J_17 = 3
};

typedef   signed int mad_fixed_t;

typedef struct mad_header {
      enum mad_layer layer;
      enum mad_mode mode;
      int mode_extension;
      enum mad_emphasis emphasis;

      unsigned long long bitrate;
      unsigned int samplerate;

      unsigned short crc_check;
      unsigned short crc_target;

      int flags;
      int private_bits;

} header_t;

typedef struct mad_frame {
      header_t header;
      int options;
      mad_fixed_t sbsample[2][36][32];
} frame_t;

# define MAD_NSBSAMPLES(header)  \
  ((header)->layer == MAD_LAYER_I ? 12 :  \
   (((header)->layer == MAD_LAYER_III &&  \
     ((header)->flags & 17)) ? 18 : 36))


void types_composite(frame_t *frame);
```

Example 26 Enum, Struct & Complex Define

The `struct` and `enum` types defined in Example 26 are used in Example 27. As with standard C compilation, `enum` types are assumed to be 32-bit values and as such will result in 32-bit values after synthesis.

Example 27 also shows how `printf` statements will be automatically ignored during synthesis.

```
#include "types_composite.h"

void types_composite(frame_t *frame)
{
	if (frame->header.mode != MAD_MODE_SINGLE_CHANNEL) {
		unsigned int ns, s, sb;
		mad_fixed_t left, right;

		ns = MAD_NSBSAMPLES(&frame->header);
		printf("Samples from header %d \n", ns);

		for (s = 0; s < ns; ++s) {
			for (sb = 0; sb < 32; ++sb) {
				left  = frame->sbsample[0][s][sb];
				right = frame->sbsample[1][s][sb];
				frame->sbsample[0][s][sb] = (left + right) / 2;
			}
		}
		frame->header.mode = MAD_MODE_SINGLE_CHANNEL;
	}
}
```

Example 27 Use Complex Types

In Example 28 a `union` is created with a `double` and a `struct`. Unlike C compilation, synthesis will not guarantee to use the same memory (in the case of synthesis, registers) for all fields in the `union`: AutoESL will perform whatever optimization provides the most optimal hardware.

Note: Pointer reinterpretation is not supported for synthesis. As such, a union cannot hold pointers to different types (or arrays of different types).

```
#include "types_union.h"

dout_t types_union(din_t N, dinfp_t F)
{
	union {
		struct {int a; int b; } intval;
		double fpval;
	} intfp;
	unsigned long long one, exp;

	// Set a floating-point value in union intfp
	intfp.fpval = F;

	// Slice out lower bits and add to shifted input
	one = intfp.intval.a;
	exp = (N & 0x7FF);

	return ((exp << 52) + one) & (0x7fffffffffffffffLL);
}
```

Example 28 Unions

Type Qualifiers

The type qualifiers can have a direct impact on the hardware created by high-level synthesis. In general, the qualifiers influence the synthesis results in a predictable manner, as detailed below, however AutoESL is only limited by the interpretation of the qualifier as it affects functional behavior and can perform optimizations to create a more optimal hardware design. Examples of this are shown after an overview of each qualifier.

Volatile

The `volatile` qualifier impacts how many reads or writes are performed in the RTL when pointers are accessed multiple times on function interfaces. Although the `volatile` qualifier impacts this behavior in all functions in the hierarchy the impact of the `volatile` qualifier is discussed in the section on top-level interfaces (Refer to the section, Understanding Volatile Data).

Statics

`Static` types in a function hold their value between function calls. The equivalent behavior in a hardware design is a registered variable (a flip-flop or memory). If a variable is required to be a `static` type for the C function to execute correctly, it will certainly be a register in the final RTL design: the value must be maintained across invocations of the function and design.

It is <u>not</u> true however to state that <u>only</u> `static` types will result in a register after synthesis. AutoESL will determine which variables are required to be implemented as registers in the RTL design. For example, if a variable assignment must be held over multiple cycles, AutoESL will create a register to hold the value, even if the original variable in the C function was <u>not</u> a `static` type.

AutoESL obeys the initialization behavior of statics and automatically assigns the value to zero, or any explicitly initialized value, to the register during initialization. This means the `static` variable will be initialized in the RTL code and in the FPGA bitstream. It does not automatically mean the variable will be re-initialized each time the reset signal is asserted.

Note: Refer to the RTL configuration (`config_rtl` command) to determine how static initialization values are implemented with regard to the system reset.

Const

A `const` type specifies that the value of the variable is never updated. The variable is read but never written to and therefore must be initialized. For most `const` variables, this will typically mean they will be reduced to constants in the RTL design (AutoESL will perform constant propagation and remove any unnecessary hardware).

In the case of arrays however, the `const` variable will be implemented as a ROM in the final RTL design (in the absence of any auto-partitioning performed by AutoESL on small arrays). Arrays specified with the `const` qualifier will, like statics, be initialized in the RTL and in the FPGA bitstream. (There is no need to reset them since they are never written to).

AutoESL Optimizations

Example 29 shows a case where AutoESL will implement a ROM even though the array is not specified with a `static` or `const` qualifier. This highlights how AutoESL will analyze the design and automatically determine the most optimal implementation: the qualifiers, or lack of them, influence but do not dictate the final RTL.

```
#include "array_ROM.h"

dout_t array_ROM(din1_t inval, din2_t idx)
{
	din1_t lookup_table[256];
	dint_t i;

	for (i = 0; i < 256; i++) {
		lookup_table[i] = 256 * (i - 128);
	}

	return (dout_t)inval * (dout_t)lookup_table[idx];
}
```

Example 29 Non-static, Non-const, ROM implementation

In the case of Example 29, AutoESL is able to determine the implementation is best served by having the variable `lookup_table` as a memory element in the final RTL. More details on how this achieved for arrays is discussed in the section Implementing ROMs.

Global Variables

Global variables can be freely used in the code and are fully synthesizable. By default however, global variables are not exposed as ports on the RTL interface. Example 30 helps explain how global variables are synthesized.

In Example 30, three global variables are used (Although this example uses arrays, all types of global variables are supported):

- Values are read from array `Ain`.
- Array `Aint` is used to transform and pass values from `Ain` to `Aout`.
- The outputs are written to array `Aout`.

```
din_t Ain[N];
din_t Aint[N];
dout_t Aout[N/2];

void types_global(din1_t idx) {
	din_t i,lidx;

	// Move elements in the input array
	for (i=0; i<N; ++i) {
		lidx=i;
		if(lidx+idx>N-1)
			lidx=N-1;
		Aint[lidx] = Ain[lidx+idx] + Ain[lidx];
	}

	// Sum to half the elements
	for (i=0; i<(N/2); i++) {
		Aout[i] = (Aint[i] + Aint[i+1])/2;
	}

}
```

Example 30 Global variables

By default, after synthesis, the only port on the RTL design will be port `idx`: global variables are not exposed as RTL ports by default. In the default case, array `Ain` is an internal RAM which is read from and `Aout` an internal RAM which is written to.

The `expose_global` option in the AutoESL interface configuration can be used to instruct AutoESL to expose global variables as ports on the RTL interface. In this case, ports will be created to access both `Ain` and `Aout` as external RAMs. In addition however, ports will also be created showing the accesses to internal RAM `Aint`.

> **Note**: When global variables are exposed, all non-static global variables in the design, including those which only have accesses internal to the design, are exposed as RTL ports. Static global variables are never exposed as ports.

In summary, global variables are supported for synthesis, however a coding style which uses global variables extensively is not recommended.

Pointers

Pointers are used extensively in C code and are well supported for synthesis. The only cases where care needs to be taken when using pointers are:

- When pointers are accessed (read or written) multiple times in the same function. Refer to Multi-Access Pointer Interfaces: Streaming Data for issues related to this.
- When using arrays of pointers, each pointer must point to a scalar or a scalar array: not another pointer.
- Pointer casting is only supported when casting between standard C types, as shown below.

Many previous examples have shown how C pointers can be synthesized using AutoESL. Synthesis support for pointers includes, as shown in Example 31, cases where pointers point to multiple objects.

```
#include "pointer_multi.h"

dout_t pointer_multi (sel_t sel, din_t pos) {
      static const dout_t a[8] = {1, 2, 3, 4, 5, 6, 7, 8};
      static const dout_t b[8] = {8, 7, 6, 5, 4, 3, 2, 1};

      dout_t* ptr;
      if (sel)
            ptr = a;
      else
      ptr = b;

      return ptr[pos];
}
```

Example 31 Multiple Pointer Targets

Double-pointers are also supported for synthesis (Example 32). If a double-pointer is used in multiple functions, AutoESL will inline all functions in which it is used. If multiple functions are inlined, it may cause an increase in run time.

```
#include "pointer_double.h"

data_t sub(data_t ptr[10], data_t size, data_t**flagPtr)
{
      data_t x, i;

      x = 0;
      // Sum x if AND of local index and double-pointer index is true
      for(i=0; i<size; ++i)
            if (**flagPtr & i)
                  x += *(ptr+i);
      return x;
}

data_t pointer_double(data_t pos, data_t x, data_t* flag)
{
      data_t array[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
      data_t* ptrFlag;
      data_t i;
```

```
        ptrFlag = flag;

        // Write x into index position pos
        if (pos >=0 & pos < 10)
       *(array+pos) = x;

        // Pass same index (as pos) as pointer to another function
        return sub(array, 10, &ptrFlag);
}
```

Example 32 Double Pointers

Arrays of pointers can also be synthesized, as shown in Example 33 where an array of pointers is used to store the start location of the 2nd dimension of a global array.

Note: The pointers in an array of pointers can only point to a scalar or to an array of scalars. They cannot point to other pointers.

```
#include "pointer_array.h"

data_t A[N][10];

data_t pointer_array(data_t B[N*10]) {
        data_t i,j;
        data_t sum1;

        // Array of pointers
        data_t* PtrA[N];

        // Store global array locations in temp pointer array
        for (i=0; i<N; ++i)
                PtrA[i] = &(A[i][0]);

        // Copy input array using pointers
        for(i=0; i<N; ++i)
                for(j=0; j<10; ++j)
                        *(PtrA[i]+j) = B[i*10 + j];

        // Sum input array
        sum1 = 0;
        for(i=0; i<N; ++i)
                for(j=0; j<10; ++j)
                        sum1 += *(PtrA[i] + j);

        return sum1;
}
```

Example 33 Pointer Arrays

Pointer casting is supported for synthesis if native C types are used. In Example 34, type `data_t` (`char`) is cast to type `dint_t` (`int`).

```
#define N 1024

typedef int data_t;
typedef char dint_t;

data_t pointer_cast_native (data_t index,  data_t A[N]) {
      dint_t* ptr;
      data_t i =0, result = 0;
      ptr = (dint_t*)(&A[index]);

      // Sum from the indexed value as a different type
      for (i = 0; i < 4*(N/10); ++i) {
            result += *ptr;
            ptr+=1;
      }
      return result;
}
```

Example 34 Pointer Casting with Native Types

Pointer casting is not however supported between general types. For example, if a (struct) composite type of signed values is created, the pointer cannot be cast to assign unsigned values.

```
struct {
  short first;
  short second;
} pair;

// Not supported for synthesis
*(unsigned*)pair = -1U;
```

In such cases, the values must be assigned using the native type(s).

```
struct {
  short first;
  short second;
} pair;

// Assigned value
pair.first = -1U;
pair.second = -1U;
```

C Arbitrary Precision Integer Types

The native data types in C are on 8-bit boundaries (8, 16, 32 and 64 bits). RTL signals and operations however support arbitrary bit-lengths. AutoESL provides arbitrary precision data types for C to allow variables and operations in the C code to be specified with any arbitrary bit-widths: 6-bit, 17-bit, 234-bit etc. up to 1024 bits.

Note: AutoESL also provides arbitrary precision data types in C++ and supports the arbitrary precision data types which are part of SystemC. These types are discussed in the respective chapters on C++ and SystemC coding.

There are two primary advantages of arbitrary precision data types:

- Better quality hardware: If for example, a 17-bit multiplier is required, arbitrary precision types can be used to specify that exactly 17-bit are used in the calculation.
    - Without arbitrary precision data types, such a multiplication (17-bit) must be implemented using 32-bit integer data types and result in the multiplication being implemented with multiple DSP48 components.
- Accurate C simulation/analysis: Arbitrary precision data types in the C code allows the C simulation to be executed using accurate bit-widths and for the C simulation to validate the functionality (and accuracy) of the algorithm before synthesis.

The remainder of this section explains how to use arbitrary precision types and reviews issues where care should be taken. A detailed description of arbitrary precision types is provided in a reference section at the end of this document (C Arbitrary Precision Types) and includes:

- Techniques for assigning constant and initialization values to arbitrary precision integers (including values greater than 64-bit).
- A description of AutoESL helper functions, such as printing, concatenating, bit-slicing and range selection functions.
- A description of operator behavior, including a description of shift operations (a negative shift values, results in a shift in the opposite direction).

Using Arbitrary Precision Types with C

For the C language, the header file `ap_cint.h` defines the arbitrary precision integer data types `[u]int#W`. For example, `int8` represents an 8-bit signed integer data type and `uint234` represents a 234-bit unsigned integer type.

The `ap_cint.h` file is located in the directory `$AUTOESL_ROOT/include`, where `$AUTOESL_ROOT` is the AutoESL installation directory.

The code shown in Example 35, is a repeat of the code shown in the earlier example on basic arithmetic (Example 22). In both examples the data types in the top-level function to be synthesized are specified as `dinA_t`, `dinB_t` etc.

```
#include "apint_arith.h"

void apint_arith(din_A  inA, din_B  inB, din_C  inC, din_D  inD,
                dout_1 *out1, dout_2 *out2, dout_3 *out3, dout_4 *out4
  ) {

      // Basic arithmetic operations
      *out1 = inA * inB;
      *out2 = inB + inA;
      *out3 = inC / inA;
```

```
        *out4 = inD % inA;

}
```

Example 35 Basic Arithmetic Revisited

The real difference between the two examples is in how the data types are defined. To use arbitrary precision integer data types in a C function,

- Add header file `ap_cint.h` to the source code.
- Change the native C types to arbitrary precision types `intN` or `uintN`, where N is a bit-size from 1 to 1024.

The data types are defined in the header `apint_arith.h` as shown in Example 36. Compared with Example 22, the input data types have simply been reduced to represent the maximum size of the real input data (e.g. 8-bit input `inA` is reduced to 6-bit input). The output types however have been refined to be more accurate, for example, `out2`, the sum of `inA` and `inB`, need only be 13-bit and not 32-bit.

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

// Previous data types
//typedef char dinA_t;
//typedef short dinB_t;
//typedef int dinC_t;
//typedef long long dinD_t;
//typedef int dout1_t;
//typedef unsigned int dout2_t;
//typedef int32_t dout3_t;
//typedef int64_t dout4_t;

typedef int6 dinA_t;
typedef int12 dinB_t;
typedef int22 dinC_t;
typedef int33 dinD_t;

typedef int18 dout1_t;
typedef uint13 dout2_t;
typedef int22 dout3_t;
typedef int6 dout4_t;

void apint_arith(dinA_t inA,dinB_t inB,dinC_t inC,dinD_t inD,dout1_t
*out1,dout2_t *out2,dout3_t *out3,dout4_t *out4);
```

Example 36 Basic Arithmetic APINT Types

If Example 36 is synthesized it will result in a design which is functionally identical to Example 22 (given data in the range specified by Example 36). The final RTL design however is smaller in area and has a faster clock speed (smaller bit-widths result in reduced logic).

However, before synthesis, the function must be compiled and validated.

Validating Arbitrary Precision Types in C

To create arbitrary precision types, attributes are added to define the bit-sizes in file `ap_cint.h`. Standard C compilers such as `gcc` will compile the attributes used in the header file, but they do know what the attributes mean. The final executable created by standard C compilers will issue messages such as the following

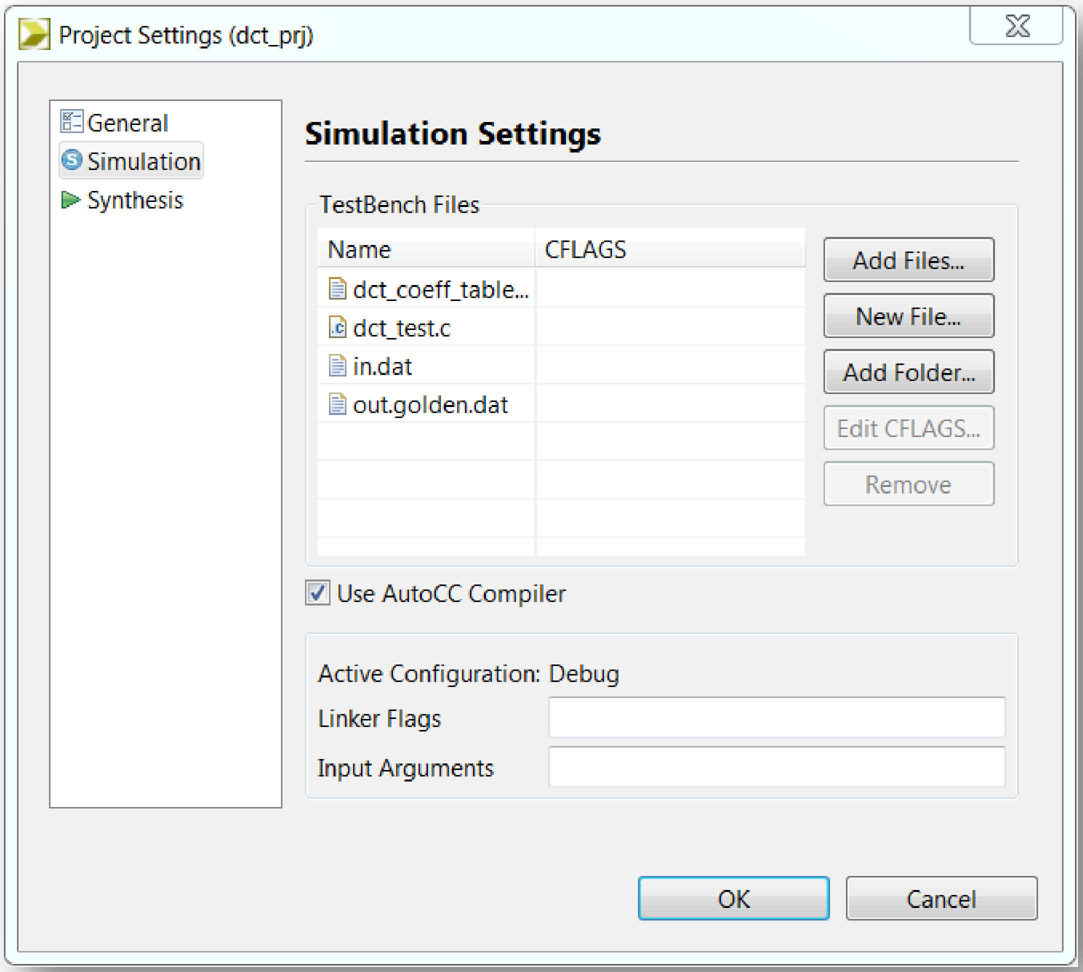
```
$AUTOESL_ROOT/include/etc/autopilot_dt.def:1036: warning: bit-width attribute
directive ignored
```

and proceed to use native C data types for the simulation. This results in computations which do not reflect the bit-accurate behavior of the code. For example, a 3-bit integer value with binary representation `100` will be treated as having a decimal value 4 and not -4.

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

> **Note**: When bit-accurate types are in C, the design must be compiled and simulated using the `autocc` compiler.

The `autocc` compiler can be enabled in the project setting using menu Project ➔ Project Settings ➔ Simulation and selecting `Use AutoCC Compiler` as shown in Figure 6.



Simulation Settings

TestBench Files

Name	CFLAGS
dct_coeff_table...	
dct_test.c	
in.dat	
out.golden.dat	

☑ Use AutoCC Compiler

Active Configuration: Debug

Linker Flags

Input Arguments

Figure 6 Enabling the AutoCC Compiler

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

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

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

In summary, when using arbitrary precision types in C, compile using the `autocc` compiler:

- Select the `Use AutoCC Compiler` option in the GUI.

- Use `autocc` in place of `gcc` at the command prompt

For functions specified using C++ or SystemC there are no such limitations when using arbitrary precision types. (An alternative may be to change the file name exertions to `.cpp` , use C++ arbitrary precision types and compile/simulate using a C++ compiler).

Debugging Arbitrary Precision Types in C

Note: When `autocc` is used to compile C code the design can no longer be analyzed in the AutoESL C debugger: this is a side-effect of using arbitrary procession type in C code.

If there is a requirement to debug the design, the following methodology is recommended:

- If the operation of the algorithm requires analysis in the debugger, use native C types (`int`, `char`, `short` etc.) and open the code in the debugger to verify correct operation of the algorithm.
    - This is easily performed when all types are defined in the header file, allowing the data types throughout to be changed in one location.
- If the operation of the algorithm is known to be correct and it is simply the case that the bit accurate nature of the arbitrary precision types must be analyzed, execute a C simulation and use the `printf` and/or `fprintf` functions to output the data values for analysis.

Integer Promotion Issues

Care should be taken when the result of arbitrary precision operations crosses the native 8, 16, 32 and 64-bit boundaries. In the following example, the intent is that two 18-bit values are multiplied and the result stored in a 36-bit number:

```
#include "ap_cint.h"

int18  a,b;
int36  tmp;

tmp = a * b;
```

However, what happens in this example is that integer promotion occurs and the result in not what is expected.

In integer promotion, the C compiler promotes the result of the multiplication operator from 18-bit, to the next native bit size (32-bit) and then assigns the result to the 36-bit variable `tmp`. This results in the behavior and incorrect result shown in Figure 7.

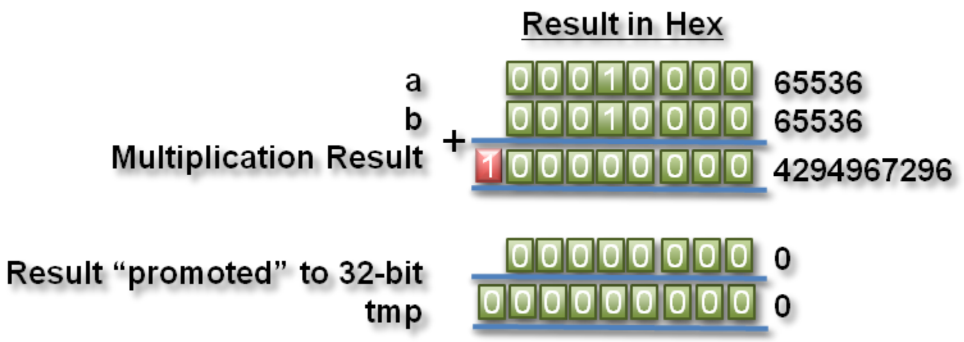


Figure 7 Integer Promotion

Since AutoESL will produce the same results as C simulation, AutoESL will create hardware where a 32-bit multiplier result is sign-extended to a 36-bit result.

The solution to the integer promotion issue is to cast operator inputs to the output size. Example 37 shows where the inputs to the multiplier are cast to 36-bit value before the multiplication. This results in the correct (expected) results during C simulation and the expected 36-bit multiplication in the RTL.

```
#include "ap_cint.h"

typedef int18 din_t;
typedef int36 dout_t;

dout_t apint_promotion(din_t a,din_t b) {
      dout_t  tmp;

      tmp = (dout_t)a * (dout_t)b;
      return tmp;
}
```

Example 37 Cast to avoid Integer Promotion

Casting to avoid integer promotion issue is only required when the result of an operation is greater than the next native boundary (8, 16, 32 or 64). This behavior is more typical with multipliers than with addition and subtraction operations.

Integer promotion issues are not present when using C++ or SystemC arbitrary precision types.

Functions

The top-level function becomes the top-level of the RTL design after synthesis and sub-functions are synthesized into blocks in the RTL design.

Note: The top-level function cannot be a static function.

After synthesis, each function in the design will have its own synthesis report and RTL HDL file (Verilog, VHDL and SystemC). Sub-functions can optionally be inlined to merge their logic with the logic of the surrounding function. Inlining functions can result in better optimizations but can also increase run time, since more logic and more possibilities have to be kept in memory and analyzed. AutoESL may perform automatic inlining of small functions (which can be disabled by setting the `inline` directive to off for that function). If a function is inlined there will be no report or separate RTL file for that function: the logic and loops are merged with the function above it in the hierarchy.

The primary impact of a coding style on functions is on the function arguments and interface.

If the arguments to a function are sized accurately, AutoESL can propagate this information through the design and there is no need to create arbitrary precision types for every variable. In the following example, two integers are multiplied, but only the bottom 24-bits are used for the result.

```
#include "ap_cint.h"

int24 foo(int x, int y) {
  int tmp;

  tmp = (x * y);
  return tmp
}
```

When this code is synthesized the result will be a 32-bit multiplier with the output truncated to 24-bit.

If however the inputs are correctly sized to 12-bit types (`int12`) as shown in Example 38, the final RTL will use a 24-bit multiplier.

```
#include "ap_cint.h"
typedef int12 din_t;
typedef int24 dout_t;

dout_t func_sized(din_t x, din_t y) {
  int tmp;

  tmp = (x * y);
  return tmp
}
```

Example 38 Sizing Function Arguments

Using arbitrary precision types for the two function inputs is enough to ensure AutoESL creates a design using a 24-bit multiplier: the 12-bit types are propagated through the design. It is recommended to correctly size the arguments of all functions in the hierarchy.

In general, when variables are driven directly from the function interface, especially from the top-level function interface, they can prevent some optimizations from taking place. A typical case of this is when an input is used as the upper limit for a loop index.

Loops

Loops provide a very intuitive and concise way of capturing the behavior of an algorithm and are used often in C code. Loops are very well supported by synthesis: loops can be pipelined, unrolled, partially unrolled, merged and flattened.

The optimizations unroll, partially unroll, flatten and merge effectively make changes to the loop structure, as if the code was changed: these optimizations ensure limited coding changes are required when optimizing loops. There are some optimizations however which can only be applied in certain conditions and some coding changes may be required to allow them.

Note: It is not recommended to use global variables for loop index variables as this can inhibit some optimizations.

Variable Loop Bounds

Some of the optimizations AutoESL can apply are prevented when the loop has variable bounds. In Example 39 shown below, the loop bounds are determined by variable `width`, which is driven from a top-level input. In this case the loop is considered to have variables bounds, since AutoESL cannot know when the loop will complete.

```
#include "ap_cint.h"
#define N 32

typedef int8 din_t;
typedef int13 dout_t;
typedef uint5 dsel_t;

dout_t code028(din_t A[N], dsel_t width) {

	dout_t out_accum=0;
	dsel_t x;

	LOOP_X:for (x=0;x<width; x++) {
		out_accum += A[x];
	}

	return out_accum;
}
```

Example 39 Variable Loop Bounds

Trying to optimize the design in Example 39 will uncover the problems which are introduced by variable loop bounds.

The first issue with variable loop bounds is that they prevent AutoESL from determining the latency of the loop. AutoESL can determine the latency to complete one iteration of the loop, but because it cannot statically determine the exact value of variable `width`, it does not know how many iteration are performed and thus cannot report the loop latency (the number of cycles to completely execute every iteration of the loop).

Where variable loop bounds are present, AutoESL will report the latency as a question mark (`?`) instead of using exact values. The following shows the result after synthesis of Example 39.

```
+ Summary of overall latency (clock cycles):
    * Best-case latency:    ?
    * Average-case latency: ?
    * Worst-case latency:   ?
+ Summary of loop latency (clock cycles):
    + LOOP_X:
        * Trip count: ?
        * Latency:    ?
```

The first problem with variable loop bounds is therefore that the performance of the design is unknown.

To overcome this problem AutoESL provides the `tripcount` directive. The `tripcount` directive allows a minimum, average and/or maximum tripcount to be specified for the loop: the `tripcount` is the number of loop iterations. If a maximum `tripcount` of 32 is applied to LOOP_X in Example 39, the report is updated to the following:

```
+ Summary of overall latency (clock cycles):
    * Best-case latency:    2
    * Average-case latency: 18
    * Worst-case latency:   34
+ Summary of loop latency (clock cycles):
    + LOOP_X:
        * Trip count: 0 ~ 32
        * Latency:    0 ~ 32
```

Note: The values provided by the user for the `tripcount` directives are used only for reporting and have impact on synthesis. The tripcount value simply allows AutoESL to report numbers in the report. This allows the user to see the effect of optimizations: solutions can now be compared.

Note: Tripcount directives have no impact on the results of synthesis, only reporting.

The next steps in optimizing Example 39 for high throughput would be:

- Unroll the loop and allow the accumulations to occur in parallel.
- Partition the array input, or the parallel accumulations will be limited, by a single memory port.

If these optimizations are applied, the output from AutoESL will highlight the biggest problem with variable bound loops:

```
@W [XFORM-503] Cannot unroll loop 'LOOP_X' in function 'code028': cannot
completely unroll a loop with a variable trip count.
```

Since variable bounds loops cannot be unrolled, they not only prevent the `unroll` directive being applied, they also prevent pipelining of the levels above the loop.

Note: When a loop or function is pipelined, AutoESL will unroll all loops in the hierarchy below the function or loop. If there is a loop with variable bounds in this hierarchy it will prevent pipelining.

The solution to loops with variable bounds is to make the number of loop iteration a fixed value with conditional executions inside the loop. The code from Example 39 can be re-written as shown in Example 40. Here, the loop bounds are explicitly set to the maximum value of variable `width` and the loop body is conditionally executed.

```
#include "ap_cint.h"
#define N 32

typedef int8 din_t;
typedef int13 dout_t;
typedef uint5 dsel_t;

dout_t loop_max_bounds(din_t A[N], dsel_t width) {

  dout_t out_accum=0;
  dsel_t x;

  LOOP_X:for (x=0;x<N-1; x++) {
       if (x<width) {
              out_accum += A[x];
       }
  }

  return out_accum;
}
```

Example 40 Variable Loop Bounds Re-Written

The for-loop (LOOP_X) in Example 40 can be unrolled: the loop has fixed upper bounds and AutoESL knows how much hardware to create. There will be N (32) copies of the loop body in the RTL design, each copy of the loop body will have conditional logic associated with it and be executed depending on the value of variable `width`.

Loop Pipelining

When pipelining loops, the most optimum balance between area and performance is typically found by pipelining the inner most loop. This is also results in the fastest run time. The code in Example 41 can be used to demonstrate the trade-offs when pipelining loops and functions.

```
#include "loop_pipeline.h"

dout_t loop_pipeline(din_t A[N]) {

  int i,j;
        static dout_t acc;

        LOOP_I:for(i=0; i < 20; i++){
               LOOP_J: for(j=0; j < 20; j++){
                      acc += A[j] * i;
               }
        }

        return acc;
}
```

Example 41 Loop Pipeline

If the inner-most (LOOP_J) is pipelined, there will be one copy of LOOP_J in hardware, (a single multiplier) and AutoESL will use the outer-loop (LOOP_I) to simply feed LOOP_J with new data. Only 1 multiplier operation and 1 array access need to be scheduled, then the loop iterations can be scheduled as single loop-body entity (20x20 loop iterations).

Note: When a loop or function is pipelined, any loop in hierarchy below the loop or function being pipelined must be unrolled.

If the outer-loop (LOOP_I) is pipelined, inner-loop (LOOP_J) will be unrolled creating 20 copies of the loop body: 20 multipliers and 20 array accesses must now be scheduled. Then each iteration of LOOP_I can be scheduled as a single entity.

If the top-level function is pipelined, both loops must be unrolled: 400 multipliers and 400 arrays accessed must now be scheduled. It is very unlikely AutoESL will produce a design with 400 multiplications since in most designs data dependencies often prevent maximal parallelism, for example in this case, even if a dual-port RAM is used for A[N] the design can only access two values of A[N] in any clock cycle (and can therefore only make use of two multipliers in each clock cycle).

The concept to appreciate when selecting at which level of the hierarchy to pipeline it to understand that pipelining the inner-most loop will give the smallest hardware with generally acceptable throughput for most applications. Pipelining the upper-levels of the hierarchy will unroll all sub-loops and can create many more operations to schedule (which could impact run time and memory capacity) but will typically give the highest performance design in terms of throughput and latency.

To summarize the above options:

- Pipeline LOOP_J: Latency will be approximately 400 cycles (20x20) and will require less than 100 LUTs and registers (the IO control and FSM are always present).

- Pipeline `LOOP_I`: Latency will be approximately 20 cycles but will require a few hundred LUTs and registers. About 20 times the logic as first option, minus any logic optimizations which can be made.
- Pipeline function `loop_pipeline`: Latency will be approximately 10 (20 dual-port accesses) but will require thousands of LUTs and registers (about 400 times the logic of the first option minus any optimizations which can be made).

Imperfect Nested Loops

When the inner-loop of a loop hierarchy is pipelined, AutoESL automatically flattens the nested loops, to reduce latency and improve overall throughput by removing any cycles caused by loop transitioning (the checks performed on the loop index when entering and exiting loops). Such checks can result in a clock delay when transitioning from one loop to the next (entry and/or exit). In Example 41, pipelining the inner-most loop would result in the following message from AutoESL.

```
@I [XFORM-541] Flattening a loop nest 'LOOP_I' in function 'loop_pipeline'.
```

Nested loops can only be flattened if the loops are perfect or semi-perfect.

- Perfect Loops
  - Only the inner most loop has body (contents).
  - There is no logic specified between the loop statements.
  - The loop bounds are constant.
- Semi-perfect Loops
  - Only the inner most loop has body (contents)
  - There is no logic specified between the loop statements.
  - The outer most loop bound can be variable.

Example 42 shows a case where the loop nest is imperfect.

```
#include "loop_imperfect.h"

void loop_imperfect(din_t A[N], dout_t B[N]) {

  int i,j;
      dint_t acc;

      LOOP_I:for(i=0; i < 20; i++){
             acc = 0;
             LOOP_J: for(j=0; j < 20; j++){
                          acc += A[i] * j;
             }
             B[i] = acc / 20;
      }
}
```

Example 42 Imperfect Nested Loops

The assignment to acc and array B[N] inside LOOP_I, but outside LOOP_J, prevent the loops from being flattened. If LOOP_J in Example 42 is pipelined, the synthesis report will show the following:

```
+ Summary of loop latency (clock cycles):
    + LOOP_I:
        * Trip count: 20
        * Latency:    480
        + LOOP_J:
            * Trip count:     20
            * Latency:        21
            * Pipeline II:    1
            * Pipeline depth: 2
```

- The pipeline depth shows it takes 2 clocks to execute one iteration of LOOP_J (this will vary with the device technology and clock period).
- A new iteration can begin each clock cycle: Pipeline II is 1 (II is the Initiation Interval: cycles between each new execution of the loop body).
- It takes 2 cycles for the first iteration to output a result. Due to pipelining each subsequent iteration executes in parallel with the previous one and outputs a value after 1 clock. The total latency of the loop is 2 plus 1 for each of the remaining 19 iterations: 21.
- LOOP_I, requires 480 clock cycles to perform 20 iterations, thus each iteration of LOOP_I is 24 clocks cycles: this means there are 3 cycles of overhead to enter and exit LOOP_J (24 – 21 = 3).

Imperfect loop nests, or the inability to flatten loop them, results in additional clock cycles to enter and exit the loops. The code in Example 42 can be re-written to make the nested loops perfect and allow them to be flattened.

Example 43 shows how conditionals can be added to loop LOOP_J to provide the same functionality as Example 42 but allow the loops to be flattened.

```
#include "loop_perfect.h"

void loop_perfect(din_t A[N], dout_t B[N]) {

  int i,j;
      dint_t acc;

      LOOP_I:for(i=0; i < 20; i++){
             LOOP_J: for(j=0; j < 20; j++){
                    if(j==0) acc = 0;
                    acc += A[i] * j;
                    if(j==19) B[i] = acc / 20;
             }
      }
}
```

Example 43 Perfect Nested Loops

When Example 43 is synthesized, the loops are flattened:

```
@I [XFORM-541] Flattening a loop nest 'LOOP_I' in function 'loop_perfect'.
```

And the synthesis report shows an improvement in latency.

```
+ Summary of loop latency (clock cycles):
    + LOOP_I_LOOP_J:
        * Trip count:     400
        * Latency:        401
        * Pipeline II:    1
        * Pipeline depth: 2
```

When the design contains nested loops, analyze the results to ensure as many nested loops as possible have been flattened: review the log file or look in the synthesis report for cases, as shown above, where the loop labels have been merged (`LOOP_I` and `LOOP_J` are now reported as `LOOP_I_LOOP_J`).

Loop Parallelism

AutoESL will schedule logic and functions are early as possible to reduce latency. To perform this it will schedule as many logic operations and functions as possible in parallel. It will not however schedule loops to execute in parallel.

If Example 44 is synthesized, loop `SUM_X` will be scheduled and then loop `SUM_Y` will be scheduled: even though loop `SUM_Y` does not need to wait for loop `SUM_X` to complete before it can begin its operation, it will be scheduled after `SUM_X`.

```
#include "loop_sequential.h"

void loop_sequential(din_t A[N], din_t B[N], dout_t X[N], dout_t Y[N],
             dsel_t xlimit, dsel_t ylimit) {

  dout_t X_accum=0;
  dout_t Y_accum=0;
  int i,j;

  SUM_X:for (i=0;i<xlimit; i++) {
      X_accum += A[i];
      X[i] = X_accum;
  }

  SUM_Y:for (i=0;i<ylimit; i++) {
      Y_accum += B[i];
      Y[i] = Y_accum;
  }
}
```

Example 44 Sequential Loops

Since the loops have different bounds (`xlimit` and `ylimit`) they cannot be merged. However by placing the loops in separate functions, as shown in Example 45, the exact same functionality can be achieved and both loops (inside the functions), can be scheduled in parallel.

```
#include "loop_functions.h"

void sub_func(din_t I[N], dout_t O[N], dsel_t limit) {
      int i;
      dout_t accum=0;

      SUM:for (i=0;i<limit; i++) {
            accum += I[i];
            O[i] = accum;
      }
}

void loop_functions(din_t A[N], din_t B[N], dout_t X[N], dout_t Y[N],
            dsel_t xlimit, dsel_t ylimit) {

      dout_t X_accum=0;
      dout_t Y_accum=0;
      int i,j;

      sub_func(A,X,xlimit);
      sub_func(B,Y,ylimit);
}
```

Example 45 Sequential Loops as Functions

If Example 45 is synthesized, the latency will be half the latency of Example 44 because the loops (as functions) can now execute in parallel.

The `dataflow` optimization could also be used in Example 44. The principle of capturing loops in functions to exploit parallelism is presented here for cases where `dataflow` optimization cannot be used. For example, in a larger example, `dataflow` optimization would be applied to <u>all</u> loops and functions at the top-level and memories placed between every top-level loop and function.

Loop Dependencies

Loop dependencies are data dependencies which prevent optimization of loops, typically pipelining. They can be within a single iteration of a loop and or between different iteration of a loop.

The simplest way to understand loop dependencies is to examine an extreme example. In the following example, the result of the loop is used as the loop continuation/exit condition: each iteration of the loop must finish before the next can start.

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

In this case, this loop cannot be pipelined: the next iteration of the loop cannot begin until the previous iteration ends.

Not all loop dependencies are as extreme as this, but this example highlights the problem: some operation cannot begin until some other operation has completed. The solution is to try ensure the initial operation is performed as early as possible.

Loop dependencies can occur with any and all types of data; however they are particularly common when using arrays. As such, they are discussed in the next section on arrays.

Arrays

Arrays are typically implemented as a memory (RAM, ROM or FIFO) after synthesis. As discussed in the section Arrays on the Interface, arrays on the top-level function interface are synthesized as RTL ports which access a memory outside. Arrays internal to the design are synthesized to internal BRAM, LUTRAM or registers, depending on the optimization settings.

Like loops, arrays are an intuitive coding construct and so they are often found in C programs. Also like loops, AutoESL has a number of optimizations and directives which can be applied to optimize their implementation in RTL without any need to modify the code.

Cases where arrays can introduce problems in the RTL are:

- Array accesses can often create bottlenecks to performance. When implemented as a memory, the number of memory ports limits access to the data.
- Array initialization, if not performed carefully, can result in undesirably long reset and initialization in the RTL.
- Some care must be taken to ensure arrays which only require read accesses are implemented as ROMs in the RTL.

As discussed in Pointers, arrays of pointers are supported, however each pointer can only point to a scalar or an array of scalars.

Note: Arrays must be sized.

Supported: Array[10];

Not Supported: Array[];

Array Accesses

The code in Example 46 shows a case where accesses to an array can limit performance in the final RTL design. In this example there are three accesses to the array `mem[N]` to create a summed result.

```
#include "array_mem_bottleneck.h"

dout_t array_mem_bottleneck(din_t mem[N]) {

  dout_t sum=0;
  int i;

  SUM_LOOP:for(i=2;i<N;++i)
    sum += mem[i] + mem[i-1] + mem[i-2];

  return sum;
}
```

Example 46 Array-Memory Bottleneck

During synthesis the array is implemented as a RAM. If the RAM is specified as a single-port RAM it is impossible to pipeline loop `SUM_LOOP` to process a new loop iteration every clock cycle.

Trying to pipeline `SUM_LOOP` with an initiation interval of 1 will result in the following message (after failing to achieve a throughput of 1, AutoESL automatically relaxes the constraint):

```
@I [SCHED-61] Pipelining loop 'SUM_LOOP'.
@W [SCHED-69] Unable to schedule 'load' operation ('mem_load_1',
array_mem_bottleneck.c:54) on array 'mem' due to limited resources (II = 1).
@W [SCHED-69] Unable to schedule 'load' operation ('mem_load_2',
array_mem_bottleneck.c:54) on array 'mem' due to limited resources (II = 2).
@I [SCHED-61] Pipelining result: Target II: 1, Final II: 3, Depth: 4.
```

The problem here is that the single-port RAM has only a single data port: only 1 read (and 1 write) can be performed in each clock cycle.

- SUM_LOOP Cycle1: read `mem[i]`;
- SUM_LOOP Cycle2: read `mem[i-1]`, sum values;
- SUM_LOOP Cycle3: read `mem[i-2]`, sum values;

A dual-port RAM could be used, but this will only allow two accesses per clock cycle. Three reads are required to calculate the value of `sum` and so three accesses per clock cycle are required in order to pipeline the loop with an new iteration every clock cycle.

Note: Arrays implemented as memory or memory ports, can often become bottlenecks to performance.

The code in Example 46 can be re-written as shown in Example 47 to allow the code to be pipelined with a throughput of 1. Notice, in Example 47 by performing pre-reads and manually pipelining the data accesses there is only one array read specified in each iteration of the loop: this ensures only a single-port RAM is required to achieve the performance.

```
#include "array_mem_perform.h"

dout_t array_mem_perform(din_t mem[N]) {

  din_t tmp0, tmp1, tmp2;
  dout_t sum=0;
  int i;

  tmp0 = mem[0];
  tmp1 = mem[1];
  SUM_LOOP:for (i = 2; i < N; i++) {
      tmp2 = mem[i];
      sum += tmp2 + tmp1 + tmp0;
      tmp0 = tmp1;
      tmp1 = tmp2;
  }

  return sum;
}
```

Example 47 Array-Memory with Performance Access

AutoESL has a number of optimization directives for changing how arrays are implemented and accessed. It is typically the case that directives can be used, and changes to the code are not required. Arrays can be partitioned into blocks or into their individual elements. In some cases, AutoESL will automatically partition arrays into individual elements: this is controllable using the configuration settings for auto-partitioning.

When an array is partitioned into multiple blocks, the single array is implemented as multiple RTL RAM blocks. When partitioned into elements, each element will be implemented as a register in the RTL. In both cases, partitioning generally allows more elements to be accessed in parallel (but it also depends on data dependencies within the code) and can help with performance; the design trade-off is between performance and the number of RAMs or registers required to achieve it.

FIFO accesses

A special care of arrays accesses are when arrays are implemented as FIFOs. This is often the case when `dataflow` optimization is used.

Accesses to a FIFO must be in sequential order starting from location zero. In addition, if an array is read in multiple locations, the code must strictly enforce the order of the FIFO accesses. It is often the case that arrays with multiple fanout cannot be implemented as FIFOs without additional code to enforce the order of the accesses.

Array Initialization

As discussed in Type Qualifiers, although not a requirement, it is highly recommended to specify arrays which are to be implemented as memories with the `static` qualifier. This not only ensures AutoESL will implement the array with a memory in the RTL, it also allows the initialization behavior of `static` types to be used.

In the following code, an array is initialized with a set of values. Each time the function is executed, array `coeff` is assigned these values. After synthesis, each time the design executes the RAM which implements `coeff` will be loaded with these values. For a single-port RAM this would take 8 clock cycles. For an array of 1024, it would of course take 1024 clock cycles, during which time no operations depending on `coeff` could occur.

```
int coeff[8] = {-2, 8, -4, 10, 14, 10, -4, 8, -2};
```

The following code uses the `static` qualifier to define array `coeff`. The array is initialized with the specified values at start of execution. Each time the function is executed however, array `coeff` remembers its values from the previous execution: a `static` array behaves in C code, as a memory does in RTL.

```
static int coeff[8] = {-2, 8, -4, 10, 14, 10, -4, 8, -2};
```

In addition, if the variable has the `static` qualifier, AutoESL will initialize the variable in the RTL design and in the FPGA bitstream: this removes the need for multiple clock cycles to initialize the memory and ensures that initializing large memories is not an operational overhead.

The RTL configuration command can be used to specify if `static` variables return to their initial state after a reset is applied (not the default). If a memory is to be returned to its initial state after a reset operation, this will incur an operational overhead and require multiple cycles to reset the values: each value has to be written into each memory address.

Implementing ROMs

As was shown in Example 29 in the review of `static` and `const` type qualifiers, AutoESL does not require that an array be specified with the `static` qualifier in order to synthesize a memory, or that the `const` qualifier be used in order to infer the memory should be a ROM. AutoESL will perform analysis of the design and seek to create the most optimum hardware.

It is however highly recommended to use the `static` qualifier for arrays which are intended to be memories: as noted in Array Initialization, a `static` type behaves in an almost identical manner as a memory in RTL.

The `const` qualifier is also recommended when arrays are only read, since AutoESL cannot always infer a ROM should be used by analysis of the design. The general rule for the automatic inference of a ROM is that a local, or a local `static` (non-global) array is written to before being read. The following practices in the code can help infer a ROM:

- Initialize the array as early as possible in the function that declares it.
- Group writes together.
- Don't interleave array(ROM) initialization writes with non-initialization code.
- Don't store different values to the same array element (group all writes together in the code).
- Element value computation must not depend on any non-constant (at compile-time) design variable(s), other than the initialization loop counter variable.

If complex assignments are used to initialize a ROM, for example functions from the `math.h` library, placing the array initialization into a separate function will allow a ROM to be inferred. In Example 48, array `sin_table[256]` is inferred as a memory and implemented as a ROM after RTL synthesis.

```
#include "array_ROM_math_init.h"
#include <math.h>

void init_sin_table(din1_t sin_table[256])
{
   int i;
   for (i = 0; i < 256; i++) {
      dint_t real_val = sin(M_PI * (dint_t)(i - 128) / 256.0);
      sin_table[i] = (din1_t)(32768.0 * real_val);
   }
}

dout_t array_ROM_math_init(din1_t inval, din2_t idx)
{
   short sin_table[256];
   init_sin_table(sin_table);
   return (int)inval * (int)sin_table[idx];
}
```

Example 48 ROM Initialization with math.h

Note: Since the result of the `sin()` function results in constant values, no core is required in the RTL design to implement the `sin()` function. The `sin()` function is not one of the cores listed in Table 2 and is not supported for synthesis in C (refer to C++ for Synthesis for using `math.h` functions in C++).

Unsupported C Constructs

While AutoESL has support for a wide range of the C language, there are some constructs which are not synthesizable or result in errors further down the design flow. This section outlines areas where coding changes must be made, if the function is to be synthesized and implemented in an FPGA device.

As a general rule, in order to be synthesized the C function must contain the entire functionality of the design (none of the functionality can be performed by system calls to the operating system), the C constructs must be of a fixed/bounded size and the implementation of those constructs unambiguous.

Top-Level Function Name Limitations

It is possible for a design written in C to use names which are illegal in the Verilog or VHDL Hardware Description Languages (HDLs) used to implement the RTL design.

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

The top-level function and its associated arguments cannot use the Verilog names as listed in Table 3.

Verilog Keywords	A-E	F-N	O-S	T-W
	and	for	or	table
	always	force	output	task
	assign	forever	parameter	tran
	attribute	fork	pmos	tranif0
	begin	function	posedge	tranif1
	buf	highhz0	primitive	time
	bufif0	highhz1	pulldown	tri
	bufif1	if	pullup	triand
	bufif1	initial	pull0	trior
	case	inout	pull1	trireg
	cmos	input	rcmos	tri0
	deassign	integer	reg	tri1
	default	join	release	vectored
	defparam	large	repeat	wait
	disable	medium	rnmos	wand
	else	module	rpmos	weak0
	endattribute	nand	rtran	weak1
	end	negedge	rtranif0	while
	endcase	nor	rtranif1	wire
	endfunction	not	scalared	wor
	endprimitive	notif0	small	
	endmodule	notif1	specify	
	endtable	nmos	specparam	
	endtask		strong0	
	event		strong1	
			supply0	
			supply1	

Table 3 Verilog Keywords

Similarly, the top-level function and its associated arguments cannot use the VHDL keywords shown in Table 4.

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

Table 4 VHDL Keywords

System Calls

System calls cannot be synthesized since they are actions which relate to performing some task upon the operating system in which the C program is running.

AutoESL will automatically ignore commonly used system calls which only display data and have no impact on the execution of the algorithm, such as `printf()` and `fprintf(stdout,)`, however in general calls to the system cannot be synthesized and should be removed from the function prior to synthesis. Other examples of such calls are `getc()`, `time()`, `sleep()` etc. all of which make calls to the operating system.

AutoESL automatically defines the macro __SYNTHESIS__ when synthesis is performed. This allows the __SYNTHESIS__ macro to be used to exclude non-synthesizable code from the design.

Example 49 shows a case where the intermediate results from a sub-function are saved to a file on the hard drive. The macro __SYNTHESIS__ is used to ensure the non-synthesizable files writes are ignored during synthesis.

```
#include "hier_func4.h"

int sumsub_func(din_t *in1, din_t *in2, dint_t *outSum, dint_t *outSub)
{
	*outSum = *in1 + *in2;
	*outSub = *in1 - *in2;
}

int shift_func(dint_t *in1, dint_t *in2, dout_t *outA, dout_t *outB)
{
	*outA = *in1 >> 1;
	*outB = *in2 >> 2;
}

void hier_func4(din_t A, din_t B, dout_t *C, dout_t *D)
{
	dint_t apb, amb;

	sumsub_func(&A,&B,&apb,&amb);
#ifndef __SYNTHESIS__
	FILE *fp1;   // The following code is ignored for synthesis
	char filename[255];
	sprintf(filename,"Out_apb_%03d.dat",apb);
	fp1=fopen(filename,"w");
	fprintf(fp1, "%d \n", apb);
	fclose(fp1);
#endif
	shift_func(&apb,&amb,C,D);
}
```

Example 49 File Writes for Debug

The __SYNTHESIS__ macro is provided as a convenient way to exclude non-synthesizable code, without removing the code itself from the C function. Using such a macro does however mean the C code for simulation and the C code for synthesis are now different.

Note: If the __SYNTHESIS__ macro is used to change the functionality of the C code, it can result in different results between C simulation and C synthesis. Errors in such code are inherently difficult to debug and using the __SYNTHESIS__ macro to create changes in functionality should be avoided.

Dynamic Memory Usage

Any system calls which manage memory allocation within the system, for example `malloc()`, `alloc()` and `free()` are using resources which exist in the memory of the operating system and are created and released during runtime: to be able to synthesize a hardware implementation the design must be fully self-contained, specifying all required resources.

Memory allocation system calls must be removed from the design code prior to synthesis. However, since dynamic memory operations are used to define the functionality of the design, they must be transformed into equivalent bounded representations. Example 50 shows how a design using `malloc()` can be transformed into a synthesizable version.

The code in Example 50 highlights two useful coding style techniques:

- First, the design does not make use of the `__SYNTHESIS__` macro: rather the user defined macro `NO_SYNTH` is used to select between the synthesizable and non-synthesizable versions. This ensures the same exact same code is simulated in C and synthesized in AutoESL.
- Secondly, the pointers in the original design using `malloc()` do not need to be re-written to work with fixed sized elements. Fixed sized resources can be created and the existing pointer can simply be made to point to the fixed sized resource: this technique can prevent manual re-coding of the existing design.

```
#include "malloc_removed.h"
#include <stdlib.h>
//#define NO_SYNTH

dout_t malloc_removed(din_t din[N], dsel_t width) {

#ifdef NO_SYNTH
	long long *out_accum = malloc (sizeof(long long));
	int* array_local = malloc (64 * sizeof(int));
#else
	long long _out_accum;
	long long *out_accum = &_out_accum;
	int _array_local[64];
	int* array_local = &_array_local[0];
#endif
	int i,j;

	LOOP_SHIFT:for (i=0;i<N-1; i++) {
		if (i<width)
			*(array_local+i)=din[i];
		else
			*(array_local+i)=din[i]>>2;
	}

	*out_accum=0;
	LOOP_ACCUM:for (j=0;j<N-1; j++) {
		*out_accum += *(array_local+j);
```

```
      }

      return *out_accum;
}
```

Example 50 Transforming malloc() to Fixed Resources

Since the coding changes here impact the functionality of the design, it is not recommended to use the __SYNTHEESIS__ macro. The recommended approach is to:

1. Add the user defined macro NO_SYNTH to the code and modify the code.
2. Enable macro NO_SYNTH, execute the C simulation and save the results.
3. Disable the macro NO_SYNTH (e.g. comment out, as in Example 50), execute the C simulation to verify the results are identical.
4. Perform synthesis with the user defined macro disabled.

This methodology ensures the updated code is validated with C simulation and the exact same code is then synthesized.

Pointer Limitations

General Pointer Casting

Pointer casting is not supported in the general case but is supported between native C types. Refer to Pointers for details on pointer casting.

Pointer Arrays

Arrays of pointers are supported for synthesis if each pointer points to a scalar or an array of scalars. Arrays of pointers cannot point to additional pointers. Refer to Pointers for details on pointer arrays.

Recursive Functions

Recursive functions cannot be synthesized. This applies to functions which can form endless recursion:

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

Tail recursion, where there are a finite number of function calls, is also not supported:

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

In C++, templates can be used to implement tail recursion. C++ is addressed next.

C++ for Synthesis

This chapter covers aspects of the C++ language as it is used for synthesis using AutoESL. Almost all of the items covered in the chapter on C for Synthesis also relate to coding with C++ (top-level function arguments, pointers, loops, arrays etc.) and the chapter C for Synthesis should have be read before proceeding with this chapter.

Topics in C for Synthesis which do not apply when using C++ are the arbitrary precision types used with C (C++ has its own arbitrary precision types) and the limitations when compiling arbitrary precision types in C: `autocc` is not required for C++ simulation and no Integer Promotion Issues are encountered with C++ arbitrary precision types.

The addition language features of C++, relevant for synthesis, include classes, templates, C++ arbitrary precision types, support for the `math.h` library and standard template libraries are covered in this chapter.

AutoESL expects C++ functions to be named with the standard `g++` file extensions (`.cpp`, `.cxx`, etc). Standard C language functions, appropriately renamed and not using AutoESL C arbitrary precision types `(u)int#`, can be synthesized as C++ designs. There is no requirement to use a C++ object orientated coding style.

C++ Classes

C++ classes are fully supported for synthesis with AutoESL. The top-level for synthesis must be a function: a class cannot be the top-level for synthesis. To synthesize a class member function, the class itself should be instantiated into function: the top-level class should not simply be instantiated into the test bench. Example 51 shows how class `CFIR` (defined in the header file discussed next) is instantiated in the top-level function `cpp_FIR` and used to implement an FIR filter.

```
#include "cpp_FIR.h"

// Top-level function with class instantiated
data_t cpp_FIR(data_t x)
  {
    static CFir<coef_t, data_t, acc_t> fir1;

    cout << fir1;

    return fir1(x);
  }
```

Example 51 C++ FIR Filter

Note: Classes and class member functions cannot be the top-level for synthesis. The class should be instantiated in a top-level function.

Before examining the class used to implement the design in Example 51, it is worth noting AutoESL automatically ignores the standard output stream `cout` during synthesis. When synthesized AutoESL will issue the following warnings:

```
@I [SYNCHK-101] Discarding unsynthesizable system call:
'std::ostream::operator<<' (cpp_FIR.h:108)
@I [SYNCHK-101] Discarding unsynthesizable system call:
'std::ostream::operator<<' (cpp_FIR.h:108)
@I [SYNCHK-101] Discarding unsynthesizable system call: 'std::operator<<
<std::char_traits<char> >' (cpp_FIR.h:110)
@
```

The header file `cpp_FIR.h` is shown below in Example 52 and shows the definition of class `CFir` and its associated member functions. In this example the operator member functions `()` and `<<` are overloaded operators, which are respectively used to execute the main FIR algorithm and used with `cout` to format the data for display during C simulation.

```
#include <fstream>
#include <iostream>
#include <iomanip>
#include <cstdlib>
using namespace std;

#define N 85

typedef int coef_t;
typedef int data_t;
typedef int acc_t;

// Class CFir definition
template<class coef_T, class data_T, class acc_T>
class CFir {
	protected:
		static const coef_T c[N];
		data_T shift_reg[N-1];
	private:
	public:
		data_T operator()(data_T x);
		template<class coef_TT, class data_TT, class acc_TT>
		friend ostream&
		operator<<(ostream& o, const CFir<coef_TT, data_TT, acc_TT> &f);
};

// Load FIR coefficients
template<class coef_T, class data_T, class acc_T>
const coef_T CFir<coef_T, data_T, acc_T>::c[N] = {
	#include "cpp_FIR.inc"
};

// FIR main algorithm
template<class coef_T, class data_T, class acc_T>
data_T CFir<coef_T, data_T, acc_T>::operator()(data_T x) {
        int i;
        acc_t acc = 0;
        data_t m;

        loop: for (i = N-1; i >= 0; i--) {
                if (i == 0) {
                        m = x;
                        shift_reg[0] = x;
                } else {
                        m = shift_reg[i-1];
                        if (i != (N-1))
                        shift_reg[i] = shift_reg[i - 1];
                }
                acc += m * c[i];
        }
        return acc;
}

// Operator for displaying results
template<class coef_T, class data_T, class acc_T>
ostream& operator<<(ostream& o, const CFir<coef_T, data_T, acc_T> &f) {
        for (int i = 0; i < (sizeof(f.shift_reg)/sizeof(data_T)); i++) {
                o << "shift_reg[" << i << "]= " << f.shift_reg[i] << endl;
        }
        o << "______________" << endl;
        return o;
}

data_t cpp_FIR(data_t x);
```

Example 52 C++ Header File Defining Classes

The test bench Example 51 is shown in Example 53 and demonstrates how top-level function `cpp_FIR` is called and validated. This example highlights some of the important attributes of a good test bench for AutoESL synthesis:

- The output results are checked against known good values.
- The test bench returns 0 if the results are confirmed to be correct.

More details on test benches are provided in Productive Test Bench.

```
#include "cpp_FIR.h"

int main() {
        ofstream result;
        data_t output;
        int retval=0;


        // Open a file to save the results
        result.open("result.dat");

        // Apply stimuli, call the top-level function and save the results
        for (int i = 0; i <= 250; i++)
```

```
	{
		output = cpp_FIR(i);

		result << setw(10) << i;
		result << setw(20) << output;
		result << endl;

	}
	result.close();

	// Compare the results file with the golden results
	retval = system("diff --brief -w result.dat result.golden.dat");
	if (retval != 0) {
		printf("Test failed  !!!\n");
		retval=1;
	} else {
		printf("Test passed !\n");
	}

	// Return 0 if the test
	return retval;
}
```

Example 53 C++ Test Bench for cpp_FIR

Constructors, Destructors and Virtual Functions

Class constructors and destructors will be included and synthesized whenever a class object is declared.

Virtual functions, including abstract ones, are supported for synthesis if AutoESL can statically determine the function during elaboration. The following are cases where virtual functions are not supported for synthesis:

- Virtual functions can be defined in a multi-layer inheritance class hierarchy but only with a single inheritance.
- Dynamic polymorphism is only supported if the pointer object can be determined at compile time. For example, such pointers cannot be used in an if-else or loop constructs.
- An STL container cannot be used to contain the pointer of an object and call the polymorphism function. For example:

```
vector<base *> base_ptrs(10);

//Push_back some base ptrs to vector.
for (int i = 0; i < base_ptrs.size(); ++i) {
	//Static elaboration cannot resolve base_ptrs[i] to actual data type.
	base_ptrs[i]->virtual_function();
}
```

- Cases where the base object pointer is a global variable are not supported. For example:

```
Base *base_ptr;

void func()
{
       ……
       base_prt->virtual_function();
       ……
}
```

- The base object pointer cannot be a member variable in a class definition.

```
// Static elaboration cannot bind base object pointer with correct data type.
class A
{
       …..
       Base *base_ptr;
       void set_base(Base *base_ptr);
       void some_func();
       …..
};

void A::set_base(Base *ptr)
{
       this.base_ptr = ptr;
}

void A::some_func()
{
       ….
       base_ptr->virtual_function();
       ….
}
```

- If the base object pointer or reference is in the function parameter list of constructor, AutoESL will not convert it. (The ISO C++ standard has depicted this in section12.7: sometimes the behavior is undefined).

```
class A {
      A(Base *b) {
             b-> virtual _ function ();
       }
};
```

Global Variables and Classes

It is not recommended to use global variables in classes as they can prevent some optimizations from occurring. Example 54, shows a case where a class is used to create the component for a filter (class `polyd_cell` is used as a component which performs shift, multiply and accumulate operations).

```
typedef long long   acc_t;
typedef int  mult_t;
typedef char data_t;
typedef char coef_t;

#define TAPS        3
#define PHASES      4
#define DATA_SAMPLES 256
#define CELL_SAMPLES 12

// Use k on line 73 static int k;

template <typename T0, typename T1, typename T2, typename T3, int N>
class polyd_cell {
private:
public:
	T0 areg;
	T0 breg;
	T2 mreg;
	T1 preg;
	T0 shift[N];
	int k;   //line 73
	T0 shift_output;
	void exec(T1 *pcout, T0 *dataOut, T1 pcin, T3 coeff, T0 data, int col)
	{
	Function_label0:;

	if (col==0) {
		SHIFT:for (k = N-1; k >= 0; --k) {
			if (k > 0)
				shift[k] = shift[k-1];
			else
				shift[k] = data;
		}
	*dataOut = shift_output;
	shift_output = shift[N-1];
	}
	*pcout = (shift[4*col]* coeff) + pcin;

	}
};

// Top-level function with class instantiated
void cpp_class_data (
	acc_t		*dataOut,
	coef_t		coeff1[PHASES][TAPS],
	coef_t		coeff2[PHASES][TAPS],
	data_t  dataIn[DATA_SAMPLES],
	int		row
) {

	acc_t  pcin0 = 0;
	acc_t  pcout0, pcout1;
	data_t dout0, dout1;
	int col;
	static acc_t accum=0;
	static int sample_count = 0;
```

```
       static polyd_cell<data_t, acc_t, mult_t, coef_t, CELL_SAMPLES>
polyd_cell0;
       static polyd_cell<data_t, acc_t, mult_t, coef_t, CELL_SAMPLES>
polyd_cell1;

       COL:for (col = 0; col <= TAPS-1; ++col) {

       polyd_cell0.exec(&pcout0,&dout0,pcin0,coeff1[row][col],dataIn[sample_cou
nt],col);

       polyd_cell1.exec(&pcout1,&dout1,pcout0,coeff2[row][col],dout0,col);


              if ((row==0) && (col==2)) {
                     *dataOut = accum;
                     accum = pcout1;
              } else {
                     accum = pcout1 + accum;
              }

       }
       sample_count++;
}
```

Example 54 C++ Class Data Member used for Loop Index

Within class `polyd_cell` there is a loop `SHIFT` used to shift data. If the loop index `k` used in loop `SHIFT` was removed and replaced with the global index for `k` (shown earlier in the example, but commented `static int k`), AutoESL would be unable to pipeline any loop or function in which class `polyd_cell` was used. AutoESL would issue the following message:

```
@W [XFORM-503] Cannot unroll loop 'SHIFT' in function 'polyd_cell<char, long
long, int, char, 12>::exec' completely: variable loop bound.
```

Using local non-global variables for loop indexing ensures AutoESL can perform all optimizations.

Templates

As earlier examples in this chapter have shown, AutoESL supports the use of templates in C++ for synthesis. Templates are not supported however for the top-level function.

Note: The top-level function cannot be a template.

In addition to the general use of templates shown in Example 52 and Example 54, templates can be used implement a form of recursion, which is not supported in standard C synthesis (Recursive Functions).

Example 55 shows a case where a templatized `struct` is used to implement a tail-recursion Fibonacci algorithm. The key to performing synthesis is that a termination class is used to implement the final call in the recursion, where a template size of one is used.

```
//Tail recursive call
template<data_t N> struct fibon_s {
	template<typename T>
	static T fibon_f(T a, T b) {
		return fibon_s<N-1>::fibon_f(b, (a+b));
	}
};

// Termination condition
template<> struct fibon_s<1> {
	template<typename T>
	static T fibon_f(T a, T b) {
		return b;
	}
};

void cpp_template(data_t a, data_t b, data_t &dout){
  dout = fibon_s<FIB_N>::fibon_f(a,b);
}
```

Example 55 C++ Tail Recursion with Templates

Types

As with the C language types discussed in section Types, AutoESL supports the same standard types in C++ types for synthesis.

Support is also provided for C++ arbitrary precision integers: the C++ arbitrary precision integers are not the same as those used in C and do not have any of the simulation limitations. In addition supported is provided in C++ for arbitrary precision fixed point types.

C++ Arbitrary Precision Integer Types

The native data types in C++ are on 8-bit boundaries (8, 16, 32 and 64 bits). RTL signals and operations however support arbitrary bit-lengths.

AutoESL provides arbitrary precision data types for C++ to allow variables and operations in the C++ code to be specified with any arbitrary bit-widths: 6-bit, 17-bit, 234-bit etc. up to 1024 bits.

Note: 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.

C++ supports use of the arbitrary precision types defined in the SystemC standard: simply include the SystemC header file `systemc.h` and use SystemC data types. More details on SystemC types are provided in the chapter on SystemC.

Arbitrary precision data types have are two primary advantages over the native C++ types:

- Better quality hardware: If for example, a 17-bit multiplier is required, arbitrary precision types can be used to specify that exactly 17-bit are used in the calculation.
  - Without arbitrary precision data types, such a multiplication (17-bit) must be implemented using 32-bit integer data types and result in the multiplication being implemented with multiple DSP48 components.
- Accurate C++ simulation/analysis: Arbitrary precision data types in the C++ code allows the C++ simulation to be performed using accurate bit-widths and for the C++ simulation to validate the functionality (and accuracy) of the algorithm before synthesis.

The arbitrary precision types in C++ have none of the disadvantages of those in C:

- C++ arbitrary types can be compiled with standard C++ compilers (there is no C++ equivalent of `autocc`, as discussed in Validating Arbitrary Precision Types in C).
- C++ arbitrary precision types do not suffer from Integer Promotion Issues.

It is not uncommon for users to a file extension from `.c` to `.cpp` so the file can be compiled as C++, where neither of the above issues are present.

The remainder of this section explains how to use arbitrary precision types. A detailed description of arbitrary precision types is provided in a reference section at the end of this document (C++ Arbitrary Precision Types) and includes:

- Techniques for assigning constant and initialization values to arbitrary precision integers (including values greater than 1024-bit).
- A description of AutoESL helper methods, such as printing, concatenating, bit-slicing and range selection functions.
- A description of operator behavior, including a description of shift operations (a negative shift values, results in a shift in the opposite direction).

Using Arbitrary Precision Types with C++

For the C++ language, the header file `ap_int.h` defines the arbitrary precision integer data types `ap_(u)int<W>`. For example, `ap_int<8>` represents an 8-bit signed integer data type and `ap_uint<234>` represents a 234-bit unsigned integer type.

The `ap_int.h` file is located in the directory `$AUTOESL_ROOT/include`, where `$AUTOESL_ROOT` is the AutoESL installation directory.

The code shown in, is a repeat of the code shown in the earlier example on basic arithmetic (Example 22 and again in Example 35). In this example the data types in the top-level function to be synthesized are specified as `dinA_t`, `dinB_t` etc.

```
#include "cpp_ap_int_arith.h"

void cpp_ap_int_arith(din_A  inA, din_B  inB, din_C  inC, din_D  inD,
                dout_1 *out1, dout_2 *out2, dout_3 *out3, dout_4 *out4
```

```
  ) {

       // Basic arithmetic operations
       *out1 = inA * inB;
       *out2 = inB + inA;
       *out3 = inC / inA;
       *out4 = inD % inA;

}
```

Example 56 Basic Arithmetic Revisited with C++ Types

In this latest update to this example, the C++ arbitrary precision types are used:

- Add header file `ap_int.h` to the source code.
- Change the native C++ types to arbitrary precision types `ap_int<N>` or `ap_uint<N>`, where N is a bit-size from 1 to 1024 (as noted above, this can be extended to 32K-bits is required).

The data types are defined in the header `cpp_ap_int_arith.h` as shown in Example 36. Compared with Example 22, the input data types have simply been reduced to represent the maximum size of the real input data (e.g. 8-bit input `inA` is reduced to 6-bit input). The output types however have been refined to be more accurate, for example, `out2`, the sum of `inA` and `inB`, need only be 13-bit and not 32-bit.

```
#ifndef _CPP_AP_INT_ARITH_H_
#define _CPP_AP_INT_ARITH_H_

#include <stdio.h>
#include "ap_int.h"

#define N 9

// Old data types
//typedef char dinA_t;
//typedef short dinB_t;
//typedef int dinC_t;
//typedef long long dinD_t;
//typedef int dout1_t;
//typedef unsigned int dout2_t;
//typedef int32_t dout3_t;
//typedef int64_t dout4_t;

typedef ap_int<6> dinA_t;
typedef ap_int<12> dinB_t;
typedef ap_int<22> dinC_t;
typedef ap_int<33> dinD_t;

typedef ap_int<18> dout1_t;
typedef ap_uint<13> dout2_t;
typedef ap_int<22> dout3_t;
typedef ap_int<6> dout4_t;
```

```
void cpp_ap_int_arith(dinA_t inA,dinB_t inB,dinC_t inC,dinD_t inD,dout1_t
*out1,dout2_t *out2,dout3_t *out3,dout4_t *out4);

#endif
```

Example 57 Basic Arithmetic with C++ Arbitrary Precision Types

If Example 56 is synthesized it will result in a design which is functionally identical to Example 22 and Example 36: to keep the test bench as similar as possible to Example 36, rather than use the C++ `cout` operator to output the results to a file, the built-in `ap_int` method `.to_int()` is used to convert the `ap_int` results to integer types used with the standard `fprintf` function.

```
fprintf(fp, "%d*%d=%d; %d+%d=%d; %d/%d=%d; %d mod %d=%d;\n",
        inA.to_int(), inB.to_int(), out1.to_int(),
        inB.to_int(), inA.to_int(), out2.to_int(),
        inC.to_int(), inA.to_int(), out3.to_int(),
        inD.to_int(), inA.to_int(), out4.to_int());
```

Note: Section C++ Arbitrary Precision Types provides comprehensive details on the methods, synthesis behavior and all aspects of using the `ap_(u)int<N>` arbitrary precision data types.

C ++ Arbitrary Precision Fixed Point Types

C++ functions can take advantage of the arbitrary precision fixed point types provided with AutoESL. Figure 8 summarizes the basic features of these fixed point types:

- The word can be signed (`ap_fixed`) or unsigned (`ap_ufixed`).
- A word with of any arbitrary size `W` can be defined.
- The number of places above the decimal point `I`, also defines the number of decimal places in the word, `W-I` (represented by `B` in the figure below).
- The type of rounding or quantization (`Q`) can be selected.
- The overflow behavior (`O` and `N`) can be selected.

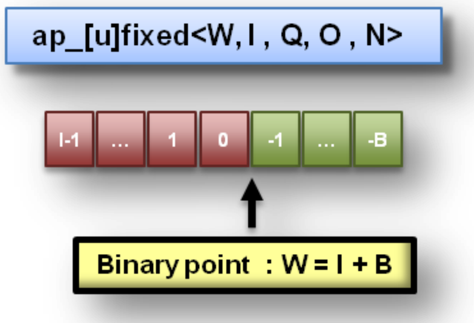


Figure 8 Arbitrary Precision Fixed Point Types

The arbitrary precision fixed point types can be used when header file `ap_fixed.h` is included in the code.

The advantages of using fixed point types are:

- They allow fractional number to be easily represented.
- When variables have a different number of integer and decimal place bits, the alignment of the decimal point is handled automatically.
- There are numerous options to automatically handle how rounding should happen: when there are too few decimal bits to represent the precision of the result.
- There are numerous options to automatically handle how variables should overflow: when the result is greater than the number of integer bits can represent.

These attributes are summarized by examining the code in Example 58. First, the header file `ap_fixed.h` is included. The `ap_fixed` types are then defined via typedef statement:

- A 10-bit input: 8-bit integer value with 2 decimal places.
- A 6-bit input: 3-bit integer value with 3 decimal places.
- A 22-bit variable for the accumulation: 17-bit integer value with 5 decimal places.
- A 36-bit variable for the result: 30-bit integer value with 6 decimal places.

Notice the function contains no code to manage the alignment of the decimal point after operations are performed: that is done automatically.

```
#include "ap_fixed.h"

typedef ap_ufixed<10,8, AP_RND, AP_SAT> din1_t;
typedef ap_fixed<6,3, AP_RND, AP_WRAP> din2_t;
typedef ap_fixed<22,17, AP_TRN, AP_SAT> dint_t;
typedef ap_fixed<36,30> dout_t;

dout_t cpp_ap_fixed(din1_t d_in1, din2_t d_in2) {

       static dint_t sum;
       sum =+ d_in1;
       return sum * d_in2;
}
```

Example 58 AP_Fixed Point Example

The quantization and overflow modes are shown in Table 5 and are described in detail in the in the reference section C++ Arbitrary Precision Fixed Point Types.

Note: Quantization and overflow modes which do more than the default behavior of standard hardware arithmetic (wrap and truncate) will result in operators with more associated hardware: it costs logic (LUTs) to implement the more advanced modes, such as round to minus infinity or saturate symmetrically.

Identifier	Description	
W	Word length in bits	
I	The number of bits used to represent the integer value (the number of bits above the decimal point)	
Q	Quantization mode dictates the behavior when greater precision is generated than can be defined by smallest fractional bit in the variable used to store the result.	
	Mode	Description
	AP_RND	Rounding to plus infinity
	AP_RND_ZERO	Rounding to zero
	AP_RND_MIN_INF	Rounding to minus infinity
	AP_RND_INF	Rounding to infinity
	AP_RND_CONV	Convergent rounding
	AP_TRN	Truncation to minus infinity
	AP_TRN_ZERO	Truncation to zero (default)
O	Overflow mode dictates the behavior when more bits are generated than the variable to store the result contains.	
	Mode	Description
	AP_SAT	Saturation
	AP_SAT_ZERO	Saturation to zero
	AP_SAT_SYM	Symmetrical saturation
	AP_WRAP	Wrap around (default)
	AP_WRAP_SM	Sign magnitude wrap around
N	The number of saturation bits in wrap modes.	

Table 5 Fixed Point Identifier Summary

Using `ap_(u)fixed` types the C++ simulation will be bit-accurate and fast simulation can be used to validate the algorithm and its accuracy. After synthesis, the RTL will exhibit the exact same bit-accurate behavior.

Arbitrary precision fixed point types can be freely assigned literal values in the code, as shown in the test bench (Example 59) used with Example 58, where the values of `in1` and `in2` are declared and assigned constant values.

When assigning literal values involving operators the literal values must first be cast to `ap_(u)fixed` types or the C compiler and AutoESL will interpret the literal as an integer or `float/double` type and may fail to find a suitable operator. For example, in the assignment of `in1 = in1 + din1_t(0.25)` the literal 0.25 is cast an `ap_fixed` type.

```
int main()
  {
	ofstream result;
	din1_t in1 = 0.25;
	din2_t in2 = 2.125;
	dout_t output;
	int retval=0;


	result.open("result.dat");
	// Persistent manipulators
	result << right << fixed << setbase(10) << setprecision(15);

	for (int i = 0; i <= 250; i++)
	{
		output = cpp_ap_fixed(in1,in2);

		result << setw(10) << i;
		result << setw(20) << in1;
		result << setw(20) << in2;
		result << setw(20) << output;
		result << endl;

		in1 = in1 + din1_t(0.25);
		in2 = in2 - din2_t(0.125);
	}
	result.close();

	// Compare the results file with the golden results
	retval = system("diff --brief -w result.dat result.golden.dat");
	if (retval != 0) {
		printf("Test failed  !!!\n");
		retval=1;
	} else {
		printf("Test passed !\n");
	}

	// Return 0 if the test passes
	return retval;
  }
```

Example 59 AP_Fixed Point Test Bench Example

Synthesis of C++ Math Functions

As discussed in the section Floats and Doubles, AutoESL can only synthesize operations in the C language using float or double types, or functions from the `math.h` library, if there is a floating point core available in the target technology to implement the operation. The available floating point cores for each technology are listed in Table 2.

With C++ designs however, AutoESL provides a bit-approximate implementation for the most common functions from the `math.h` or `cmath.h` library.

- A bit-approximate implementation does not provide the exact same accuracy as the standard operation. The accuracy is typically within 1 ULP (unit of last place) over most operating ranges but may be as large as 100 ULP.
- The list of supported functions is provided in Table 6.

Note: As discussed below, it is important to confirm the differences in accuracy between pre-synthesis simulation and post-synthesis simulation to determine if the differences are acceptable.

Functions which will be implemented using a bit-approximate implementation (since no floating point core is available) are listed in Table 6. AutoESL will automatically synthesize these functions. (These are in addition to the functions listed in Table 2 for which floating point cores exist).

Refer to the common usage errors after Table 6 before synthesizing code with math functions.

Function	float	double	Accuracy (ULP)
ceilf	X	NA	Exact
copysignf	X	NA	Exact
fabsf	X	NA	Exact
floorf	X	NA	Exact
logf	X	NA	1 to 5
cosf	X	NA	1 to 100
sinf	X	NA	1 to 100
ceil	X	X	Exact

Function	float	double	Accuracy (ULP)
copysign	X	X	Exact
fabs	X	X	Exact
floor	X	X	Exact
log	X	X	1 to 5 (6 for doubles)
cos	X	-	1 to 100
sin	X	-	1 to 100

Table 6 Math.h Bit-Approximate Supported Functions

The following are common use errors when synthesizing math functions. These are often, but not exclusively, caused by converting C functions to C++ in order to take advantage of synthesis for math functions.

If the C++ `cmath.h` header file is used, the floating point functions (`sinf`, `cosf`, etc) can be used and these will result in 32-bit operations in hardware. The `cmath.h` header file also overloads the standard functions (`sin`, `cos`, etc) so that they can be used for float and double types.

If the C `math.h` library is used, the floating point functions (`sinf`, `cosf`, etc) are required in order to synthesize 32-bit floating point operations. All standard function calls (`sin`, `cos`, etc.) will result in doubles and 64-bit double-precision operations being synthesized.

Note: When converting C functions to C++ in order to take advantage of `math.h` support, ensure the new C++ code compiles correctly before synthesizing with AutoESL.

For example, if `sqrtf()` is used in the code with `math.h` it requires the following code `extern "C" float sqrtf(float);` added to C++ code to support it.

Also, follow the warnings on mixing double and float types, outlined in section Floats and Doubles, to avoid unnecessary hardware caused by type conversion.

Example 60 shows a C++ design using the `sinf`, `cosf` and `sqrtf` functions can which can be synthesized using AutoESL.

```
#include "cpp_math.h"

data_t cpp_math(data_t angle) {
	data_t s = sinf(angle);
	data_t c = cosf(angle);
	return sqrtf(s*s+c*c);
}
```

Example 60 C++ Synthesis of math.h Functions

The header file `cpp_math.h` is shown in Example 61 and shows how the `cmath` library is called (`math.h` is used in C), the standard namespace is used and the data type `data_t` is defined as a `float` type.

```
#include <cmath>
#include <fstream>
#include <iostream>
#include <iomanip>
#include <cstdlib>
using namespace std;

typedef float data_t;

data_t cpp_math(data_t angle);
```

Example 61 C++ Synthesis of math.h Functions header file

Although the test bench shown in Example 62 will validate the results in a pre-synthesis simulation, it will return an error after post-synthesis RTL simulation: this is discussed next.

```
#include "cpp_math.h"

int main() {
	ofstream result;
	data_t angle = 0.01;
	data_t output;
	int retval=0;


	result.open("result.dat");
	// Persistent manipulators
	result << right << fixed << setbase(10) << setprecision(15);

	for (data_t i = 0; i <= 250; i++)
	{
		output = cpp_math(angle);

		result << setw(10) << i;
		result << setw(20) << angle;
		result << setw(20) << output;
		result << endl;

		angle = angle + .1;
	}
	result.close();
```

```
        // Compare the results file with the golden results
        retval = system("diff --brief -w result.dat result.golden.dat");
        if (retval != 0) {
              printf("Test failed  !!!\n");
              retval=1;
        } else {
              printf("Test passed !\n");
        }

        // Return 0 if the test passes
        return retval;
}
```

Example 62 C++ Synthesis of math.h functions Test Bench

Simulation Differences

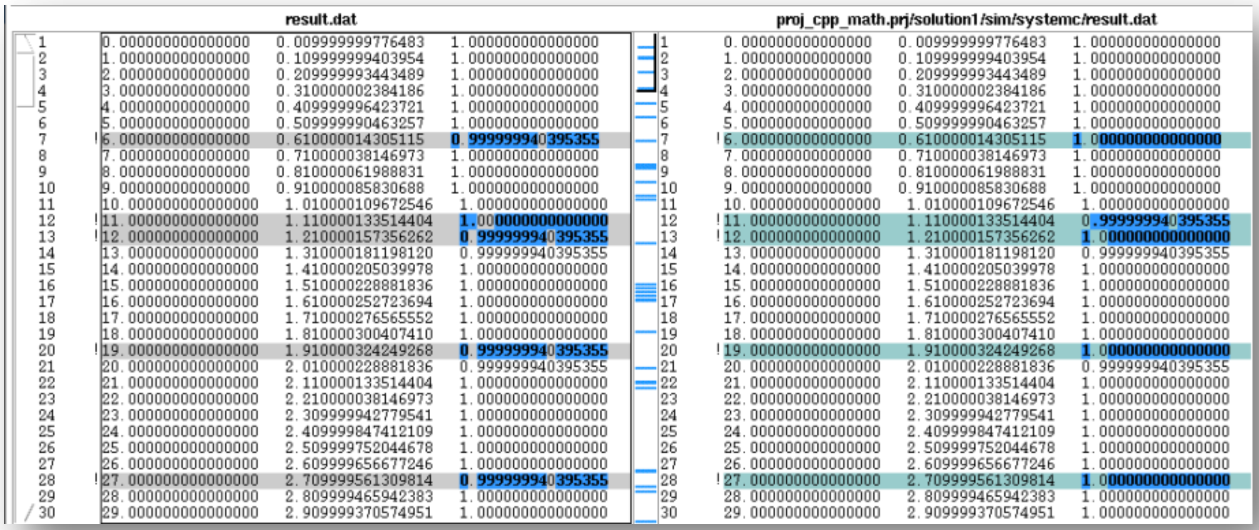
The test bench in Example 62 performs a comparison of the output results, saved to file `result.dat`, with the expected data values in `result.golden.dat`. After synthesis, the RTL implementation of the `math.h` functions will be bit-approximate versions of the functions and the `results.dat` file output by the RTL simulation may therefore contain different results than those expected.

AutoESL executes the RTL simulation in the project sub-directory <SOLUTION>/sim/<HDL>, where SOLUTION is the name of the solution and HDL is the HDL type chosen for RTL simulation. For example, given the following project settings:

- Project is called `proj_cpp_math.prj`
- Solution is `solution1`
- RTL is simulated using `systemc`

the RTL simulation output will be saved in file in `proj_cpp_math.prj/solution1/sim/systemc/results.dat`.

Figure 9 shows a comparison of the pre-synthesis `result.dat` file and the post-synthesis RTL `result.dat` file: the output value is shown in the 3rd column.



result.dat

```
 0. 000000000000000   0. 009999999776483   1. 000000000000000
 1. 000000000000000   0. 109999999403954   1. 000000000000000
 2. 000000000000000   0. 209999993443489   1. 000000000000000
 3. 000000000000000   0. 310000002384186   1. 000000000000000
 4. 000000000000000   0. 409999996423721   1. 000000000000000
 5. 000000000000000   0. 509999990463257   1. 000000000000000
!6. 000000000000000   0. 610000014305115   0. 999999940395355
 7. 000000000000000   0. 710000038146973   1. 000000000000000
 8. 000000000000000   0. 810000061988831   1. 000000000000000
 9. 000000000000000   0. 910000085830688   1. 000000000000000
 10. 000000000000000   1. 010000109672546   1. 000000000000000
!11. 000000000000000   1. 110000133514404   1. 000000000000000
!12. 000000000000000   1. 210000157356262   0. 999999940395355
 13. 000000000000000   1. 310000181198120   0. 999999940395355
 14. 000000000000000   1. 410000205039978   1. 000000000000000
 15. 000000000000000   1. 510000228881836   1. 000000000000000
 16. 000000000000000   1. 610000252723694   1. 000000000000000
 17. 000000000000000   1. 710000276565552   1. 000000000000000
 18. 000000000000000   1. 810000300407410   1. 000000000000000
!19. 000000000000000   1. 910000324249268   0. 999999940395355
 20. 000000000000000   2. 010000228881836   0. 999999940395355
 21. 000000000000000   2. 110000133514404   1. 000000000000000
 22. 000000000000000   2. 210000038146973   1. 000000000000000
 23. 000000000000000   2. 309999942779541   1. 000000000000000
 24. 000000000000000   2. 409999847412109   1. 000000000000000
 25. 000000000000000   2. 509999752044678   1. 000000000000000
 26. 000000000000000   2. 609999656677246   1. 000000000000000
!27. 000000000000000   2. 709999561309814   0. 999999940395355
 28. 000000000000000   2. 809999465942383   1. 000000000000000
 29. 000000000000000   2. 909999370574951   1. 000000000000000
```

proj_cpp_math.prj/solution1/sim/systemc/result.dat

```
 0. 000000000000000   0. 009999999776483   1. 000000000000000
 1. 000000000000000   0. 109999999403954   1. 000000000000000
 2. 000000000000000   0. 209999993443489   1. 000000000000000
 3. 000000000000000   0. 310000002384186   1. 000000000000000
 4. 000000000000000   0. 409999996423721   1. 000000000000000
 5. 000000000000000   0. 509999990463257   1. 000000000000000
!6. 000000000000000   0. 610000014305115   1. 000000000000000
 7. 000000000000000   0. 710000038146973   1. 000000000000000
 8. 000000000000000   0. 810000061988831   1. 000000000000000
 9. 000000000000000   0. 910000085830688   1. 000000000000000
 10. 000000000000000   1. 010000109672546   1. 000000000000000
!11. 000000000000000   1. 110000133514404   0 .999999940395355
!12. 000000000000000   1. 210000157356262   1. 000000000000000
 13. 000000000000000   1. 310000181198120   0. 999999940395355
 14. 000000000000000   1. 410000205039978   1. 000000000000000
 15. 000000000000000   1. 510000228881836   1. 000000000000000
 16. 000000000000000   1. 610000252723694   1. 000000000000000
 17. 000000000000000   1. 710000276565552   1. 000000000000000
 18. 000000000000000   1. 810000300407410   1. 000000000000000
!19. 000000000000000   1. 910000324249268   1. 000000000000000
 20. 000000000000000   2. 010000228881836   0. 999999940395355
 21. 000000000000000   2. 110000133514404   1. 000000000000000
 22. 000000000000000   2. 210000038146973   1. 000000000000000
 23. 000000000000000   2. 309999942779541   1. 000000000000000
 24. 000000000000000   2. 409999847412109   1. 000000000000000
 25. 000000000000000   2. 509999752044678   1. 000000000000000
 26. 000000000000000   2. 609999656677246   1. 000000000000000
!27. 000000000000000   2. 709999561309814   1. 000000000000000
 28. 000000000000000   2. 809999465942383   1. 000000000000000
 29. 000000000000000   2. 909999370574951   1. 000000000000000
```

Figure 9 Pre-Synthesis and Post-Synthesis Simulation Differences

The results of pre-synthesis simulation and post-synthesis simulation differ, in this algorithm and test bench, by fractional amounts. The question is whether these fractional amounts are acceptable in the final RTL implementation.

The recommended flow for handling these differences are:

- Verify the differences in accuracy are acceptable and use a different `result.golden.dat` file when performing pre and post-synthesis simulation.

- Or create a smart test bench which checks the results to ensure they lie within an acceptable error range. This will probably include reading the expected results into the test bench and checking the absolute difference between the actual and expected values.

Unsupported C++ Constructs

The supported C++ constructs which cannot be synthesized are listed in this section and are in addition to those listed in Unsupported C Constructs.

Dynamic Objects

As with restrictions on dynamic memory usage in C, C++ objects which are dynamically created and/or destroyed are not supported for synthesis. This includes dynamic polymorphism and dynamic virtual function calls. The following cannot be synthesized since it create new function at run time.

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

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

foo(a);
```

Standard Template Libraries

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

The solution with STLs is to create a local function with identical functionality which does not exhibit these characteristics of recursion, dynamic memory allocation or the dynamic creation and destruction of objects.

-

SystemC Synthesis

AutoESL provides support for SystemC (IEEE standard 1666), a C++ class library used to model hardware and available at www.systemc.org. AutoESL supports SystemC version 2.1 and SystemC Synthesizable Subset (Draft 1.3).

This chapter provides details on the synthesis of SystemC functions with AutoESL. The information provided here is in addition to the information provided in the earlier chapters, C for Synthesis and C++ for Synthesis, and those chapters should be read to fully understand the basic rules of coding for synthesis

Note: As with C and C++ designs, the top-level function for synthesis must be a function below the top-level for C compilation `sc_main():` the `sc_main()` function cannot be the top-level function for synthesis.

Design Modeling

The top-level for synthesis must be an SC_MODULE. Designs can be synthesized if modeled using the SystemC constructor processes SC_METHOD and SC_CTHREAD or if SC_MODULES are instantiated inside other SC_MODULES.

An SC_ MODULE cannot be defined inside another SC_MODULE (they can be instantiated, as shown later). In cases, like the following where a module is defined inside another:

```
SC_MODULE(nested1)
{
	SC_MODULE(nested2)
	{
		sc_in<int> in0;
		sc_out<int> out0;
		SC_CTOR(nested2)
		{
			SC_METHOD(process);
			sensitive<<in0;
		}
		void process()
		{
			int var =10;
			out0.write(in0.read()+var);
		}
	};

	sc_in<int> in0;
	sc_out<int> out0;
	nested2 nd;
	SC_CTOR(nested1)
	:nd("nested2")
	{
		nd.in0(in0);
```

```
                nd.out0(out0);
        }
};
```

Must be transformed into a version, such as shown next, where the modules are not nested.

```
SC_MODULE(nested2)
{
        sc_in<int> in0;
        sc_out<int> out0;
        SC_CTOR(nested2)
        {
                SC_METHOD(process);
                sensitive<<in0;
        }
        void process()
        {
                int var =10;
                out0.write(in0.read()+var);
        }
};

SC_MODULE(nested1)
{
        sc_in<int> in0;
        sc_out<int> out0;
        nested2 nd;
        SC_CTOR(nested1)
        :nd("nested2")
        {
                nd.in0(in0);
        nd.out0(out0);
        }
};
```

Similarly, an SC_MODULE cannot be derived from another SC_MODULE, as shown in this example:

```
SC_MODULE(BASE)
{
        sc_in<bool> clock; //clock input
        sc_in<bool> reset;
        SC_CTOR(BASE) {}

};

class DUT: public BASE
{
public:
        sc_in<bool> start;
        sc_in<sc_uint<8> > din;
        …
};
```

For the module constructor, it is highly recommended to define it inside the module. Cases like the following,

```
SC_MODULE(dut) {
       sc_in<int> in0;
       sc_out<int>out0;
       SC_HAS_PROCESS(dut);
       dut(sc_module_name nm);
        …
};

dut::dut(sc_module_name nm)
{
       SC_METHOD(process);
       sensitive<<in0;
 }
```

Should be transformed to:

```
SC_MODULE(dut) {
       sc_in<int> in0;
       sc_out<int>out0;

       SC_HAS_PROCESS(dut);
       dut(sc_module_name nm)
       :sc_module(nm)
       {
              SC_METHOD(process);
              sensitive<<in0;
       }
       …
};
```

SC_THREADs are not supported for synthesis.

Using SC_METHOD

Example 63 shows the header file, `sc_combo_method.h`, for a small combinational design modeled using an SC_METHOD to model a half-adder. The top-level design name, `sc_combo_method`, is specified in the SC_MODULE.

```
#include <systemc.h>

SC_MODULE(sc_combo_method){
       //Ports
       sc_in<sc_uint<1> > a,b;
       sc_out<sc_uint<1> > sum,carry;

       //Process Declaration
       void half_adder();

       //Constructor
       SC_CTOR(sc_combo_method){
              //Process Registration
              SC_METHOD(half_adder);
              sensitive<<a<<b;
       }
};
```

Example 63 SystemC Combinational Example Header

The design has two single-bit input ports (`a` and `b`). The SC_METHOD is sensitive to any changes in the state of either input port and executes function `half_adder`. The function `half_adder` is specified in file, `sc_combo_method.cpp`, shown in Example 64 and calculates the value for output port `carry`.

```
#include "sc_combo_method.h"

void sc_combo_method::half_adder(){
  bool s,c;
  s=a.read() ^ b.read();
  c=a.read() & b.read();
  sum.write(s);
  carry.write(c);

#ifndef __SYNTHESIS__
       cout << "Sum is " << a << " ^ " << b << " = " << s << ": " <<
sc_time_stamp() <<endl;
       cout << "Car is " << a << " & " << b << " = " << c << ": " <<
sc_time_stamp() <<endl;
#endif
```

Example 64 SystemC Combinational Example Main Function

Example 64 shows how any `cout` statements used to display values during C simulation can be protected from synthesis using the __SYNTHESIS__ macro (refer to System Calls for more details on this auto-defined macro).

The test bench for the Example 64 is shown in Example 65. This test bench displays a number of important attributes required when using AutoESL.

```
#ifdef __RTL_SIMULATION__
#include "sc_combo_method_rtl_wrap.h"
#define sc_combo_method sc_combo_method_RTL_transactor
#else
#include "sc_combo_method.h"
#endif
#include "tb_init.h"
#include "tb_driver.h"

int sc_main (int argc , char *argv[])
{
sc_report_handler::set_actions("/IEEE_Std_1666/deprecated", SC_DO_NOTHING);
sc_report_handler::set_actions( SC_ID_LOGIC_X_TO_BOOL_, SC_LOG);
sc_report_handler::set_actions( SC_ID_VECTOR_CONTAINS_LOGIC_VALUE_, SC_LOG);
sc_report_handler::set_actions( SC_ID_OBJECT_EXISTS_, SC_LOG);
```

```
      sc_signal<bool>     s_reset;
      sc_signal<sc_uint<1> >    s_a;
      sc_signal<sc_uint<1> >    s_b;
      sc_signal<sc_uint<1> >    s_sum;
      sc_signal<sc_uint<1> >    s_carry;

      // Create a 10ns period clock signal
      sc_clock s_clk("s_clk",10,SC_NS);

      tb_init      U_tb_init("U_tb_init");
      sc_combo_method     U_dut("U_dut");
      tb_driver    U_tb_driver("U_tb_driver");

      // Generate a clock and reset to drive the sim
      U_tb_init.clk(s_clk);
      U_tb_init.reset(s_reset);

      // Connect the DUT
      U_dut.a(s_a);
      U_dut.b(s_b);
      U_dut.sum(s_sum);
      U_dut.carry(s_carry);

      // Drive stimuli from dat* ports
      // Capture results at out* ports
      U_tb_driver.clk(s_clk);
      U_tb_driver.reset(s_reset);
      U_tb_driver.dat_a(s_a);
      U_tb_driver.dat_b(s_b);
      U_tb_driver.out_sum(s_sum);
      U_tb_driver.out_carry(s_carry);

      // Sim for 200
      int end_time = 200;

      cout << "INFO: Simulating " << endl;

      // start simulation
      sc_start(end_time, SC_NS);

      if (U_tb_driver.retval != 0) {
            printf("Test failed  !!!\n");
      } else {
            printf("Test passed !\n");
  }
  return U_tb_driver.retval;
};
```

Example 65 SystemC Combinational Example Test Bench

In order to perform RTL simulation using the `autosim` feature in AutoESL, the test bench must contain the macros shown at the top of Example 65. Given a design with the name DUT, the following must be used, where DUT is replaced with the actual design name.

```
#ifdef __RTL_SIMULATION__
#include "DUT_rtl_wrap.h"
#define DUT DUT_RTL_transactor
#else
#include "DUT.h"  //Original unmodified code
#endif
```

Failure to add this in the test bench where the design header file is included will result in `autosim` RTL simulation failing.

The report handler functions shown in Example 65 should be added to all SystemC test bench files used with AutoESL.

```
sc_report_handler::set_actions("/IEEE_Std_1666/deprecated", SC_DO_NOTHING);
sc_report_handler::set_actions( SC_ID_LOGIC_X_TO_BOOL_, SC_LOG);
sc_report_handler::set_actions( SC_ID_VECTOR_CONTAINS_LOGIC_VALUE_, SC_LOG);
sc_report_handler::set_actions( SC_ID_OBJECT_EXISTS_, SC_LOG);
```

These settings prevent the printing of extraneous messages during RTL simulation. The most important of these messages are the warnings:

```
Warning: (W212) sc_logic value 'X' cannot be converted to bool
```

The adapters placed around the synthesized design will start with unknown (X) values. Not all SystemC types support unknown (X) values. This warning is issued when this occurs but if can be ignored unless the design lacks the appropriate data handshakes (and reads data unknown data).

Finally, the test bench in Example 65 performs checking on the results and returns a value of zero if the results are correct. In this case, the results are verified inside function `tb_driver` but the return value is checked and returned in the top-level test bench.

```
if (U_tb_driver.retval != 0) {
             printf("Test failed  !!!\n");
      } else {
             printf("Test passed !\n");
  }
  return U_tb_driver.retval;
```

Instantiating SC_MODULES

Hierarchical instantiations of SC_MODULEs can be synthesized, as shown in Example 66. In Example 66, the two instances of the half-adder design (`sc_combo_method`) from Example 63 are instantiated to create a full-adder design.

```
#include <systemc.h>
#include "sc_combo_method.h"
```

```
SC_MODULE(sc_hier_inst){
	//Ports
	sc_in<sc_uint<1> > a, b, carry_in;
	sc_out<sc_uint<1> > sum, carry_out;

	//Variables
	sc_signal<sc_uint<1> > carry1, sum_int, carry2;

	//Process Declaration
	void full_adder();

	//Half-Adder Instances
	sc_combo_method     U_1, U_2;

	//Constructor
	SC_CTOR(sc_hier_inst)
	:U_1("U_1")
	,U_2("U_2")
	{
		// Half-adder inst 1
		U_1.a(a);
		U_1.b(b);
		U_1.sum(sum_int);
		U_1.carry(carry1);

		// Half-adder inst 2
		U_2.a(sum_int);
		U_2.b(carry_in);
		U_2.sum(sum);
		U_2.carry(carry2);

		//Process Registration
		SC_METHOD(full_adder);
		sensitive<<carry1<<carry2;
	}
};
```

Example 66 SystemC Hierarchical Example

The function `full_adder` is used to create the logic for the `carry_out` signal, as shown in Example 67.

```
#include "sc_hier_inst.h"

void sc_hier_inst::full_adder(){
  carry_out= carry1.read() | carry2.read();

}
```

Example 67 SystemC full_adder Function

Using SC_CTHREAD

The constructor process SC_CTHREAD is used to model clocked processes (threads) and is the primary way to model sequential designs. Example 68 shows a case which highlights the primary attributes of a sequential design.

- The data has associated handshake signals, allowing it to operate with the same test bench before and after synthesis.
- An SC_CTHREAD sensitive on the clock is used to model when the function is executed.
- The SC_CTHREAD supports reset behavior.

```
#include <systemc.h>

SC_MODULE(sc_sequ_cthread){
	//Ports
	sc_in <bool>  clk;
	sc_in <bool>  reset;
	sc_in <bool>  start;
	sc_in<sc_uint<16> > a;
	sc_in<bool> en;
	sc_out<sc_uint<16> > sum;
	sc_out<bool> vld;

	//Variables
	sc_uint<16> acc;

	//Process Declaration
	void accum();

	//Constructor
	SC_CTOR(sc_sequ_cthread){

		//Process Registration
		SC_CTHREAD(accum,clk.pos());
		reset_signal_is(reset,true);
	}
};
```

Example 68 SystemC SC_CTHREAD Example

Function `accum` is shown in Example 69. The important aspects highlighted by this examples are:

- The core modeling process is an infinite `while()` loop with a `wait()` statement inside it.
- Any initialization of the variables is performed before the infinite `while()` loop: this code is executed when reset is recognized by the SC_CTHREAD.
- The data reads and writes are qualified by handshake protocols.

```
#include "sc_sequ_cthread.h"
```

```
void sc_sequ_cthread::accum(){

	//Initialization
	acc=0;
	sum.write(0);
	vld.write(false);
	wait();

	// Process the data
	while(true) {
		// Wait for start
		while (!start.read()) wait();

		// Read if valid input available
		if (en) {
			acc = acc + a.read();
			sum.write(acc);
			vld.write(true);
		} else {
			vld.write(false);
		}
		wait();
	}

}
```

Example 69 SystemC SC_CTHREAD Function

Synthesis with Multiple Clocks

SystemC, unlike C and C++ synthesis, supports designs with multiple clocks. In a multiple clock design, the functionality associated with each clock must be captured in a SC_CTHREAD.

Example 70 shows a design in which there are two clocks, named `clock` and `clock2`. One is used to activate an SC_CTHREAD executing function `Prc1` and the other used to activate an SC_CTHREAD executing function `Prc2`. After synthesis, all the sequential logic associated with function `Prc1` will be clocked by `clock`, while `clock2` will drive all the sequential logic of function `Prc2`.

```
#include"systemc.h"
#include"tlm.h"
using namespace tlm;

SC_MODULE(sc_multi_clock)
{
	//Ports
	sc_in <bool>  clock;
	sc_in <bool>  clock2;
	sc_in <bool>  reset;
	sc_in <bool>  start;
	sc_out<bool>  done;
	sc_fifo_out<int> dout;
```

```
        sc_fifo_in<int> din;

        //Variables
        int share_mem[100];
        bool write_done;

        //Process Declaration
        void Prc1();
        void Prc2();

        //Constructor
        SC_CTOR(sc_multi_clock)
        {
                //Process Registration
                SC_CTHREAD(Prc1,clock.pos());
                reset_signal_is(reset,true);

                SC_CTHREAD(Prc2,clock2.pos());
                reset_signal_is(reset,true);
        }
};
```

Example 70 SystemC Multiple Clock Design

Top-Level SystemC Ports

The ports in a SystemC design are specified in the source code. The one major difference when using SystemC, as compared to C and C++ functions, is that AutoESL only performs interface synthesis on supported memory interfaces (refer to Arrays on the Interface). All port on the top-level interface must be of types `sc_in_clk`, `sc_in`, `sc_out`, `sc_inout`, `sc_fifo_in`, `sc_fifo_out` or `ap_mem_if`.

With the exception of the supported memory interfaces (`sc_fifo_in`, `sc_fifo_out` and `ap_mem_if`) all handshaking between the design and the test bench must be explicitly modeled in the SystemC function.

Note: AutoESL may add additional clock cycles to a SystemC design if this is required to meet timing. Since the number of clock cycles after synthesis may be different, SystemC designs should handshake all data transfers with the test bench.

Transaction level modeling using TLM 2.0 and event based modeling are not supported for synthesis.

SystemC Interface Synthesis

In general, AutoESL does not perform interface synthesis on SystemC. As mentioned above, it does support interface synthesis for some memory interfaces, namely RAM and FIFO ports.

RAM Port Synthesis

Unlike the synthesis of C and C++, AutoESL does not automatically transform array ports into RTL RAM ports. In the following SystemC code, AutoESL directives must be used to partition the array ports into individual elements or this example code cannot be synthesized:

```
SC_MODULE(dut)
{
	sc_in<T> in0[N];
	sc_out<T>out0[N];

	…
	SC_CTOR(dut)
	{
		…
	}
};
```

The directives which would partition these arrays into individual elements are,

```
set_directive_array_partition dut in0 -type complete
set_directive_array_partition dut out0 -type complete
```

If however N is a large number, this will result in many individual scalar ports on the RTL interface.

Example 71 shows how a RAM interface can be modeled in SystemC simulation and fully synthesized by AutoESL. In Example 71, the arrays are replaced by `ap_mem_if` types which can synthesized into RAM ports.

- To use `ap_mem_port` types, the header file `ap_mem_if.h` from the `include/ap_sysc` directory in the AutoESL installation area must be included.
    - Inside the AutoESL environment, the directory `include/ap_sysc` is automatically included.
- The arrays for `din` and `dout` are replaced by `ap_mem_port` types (The fields are explained after Example 71).

```
#include"systemc.h"
#include "ap_mem_if.h"


SC_MODULE(sc_RAM_port)
{
	//Ports
	sc_in <bool>  clock;
	sc_in <bool>  reset;
	sc_in <bool>  start;
	sc_out<bool>  done;
	//sc_out<int> dout[100];
	//sc_in<int> din[100];
	ap_mem_port<int, int, 100, RAM2P> dout;
	ap_mem_port<int, int, 100, RAM2P> din;

	//Variables
	int share_mem[100];
```

```
        sc_signal<bool> write_done;

        //Process Declaration
        void Prc1();
        void Prc2();

        //Constructor
        SC_CTOR(sc_RAM_port)
        : dout ("dout"),
        din ("din")
        {
                //Process Registration
                SC_CTHREAD(Prc1,clock.pos());
                reset_signal_is(reset,true);

                SC_CTHREAD(Prc2,clock.pos());
                reset_signal_is(reset,true);
        }
};
```

Example 71 SystemC RAM Interface

The format of the `ap_mem_port` type is:

```
ap_mem_port (<data_type>, < address_type>, <number_of_elements>, <Mem_Target>)
```

- The `data_type` is the type used for the stored data elements. In Example 71, these are standard `int` types.
- The `address_type` is the type used for the address bus. This type should have enough data bits to address all elements in the array, or C simulation will fail.
- The `number_of_elements` specifies the number of elements in the array being modeled.
- The `Mem_Target` specifies the memory to which this port will connect and hence determines the IO ports on the final RTL. A list of the available targets are provided in Table 7.

The memory targets described in Table 7 influence both the ports created by synthesis and how the operations are scheduled in the design. For example, a dual-port RAM will result in twice as many IO ports as a single-port RAM and may allow internal operations to be scheduled in parallel: if code constructs, such as loops, and data dependencies allow this.

Target RAM	Description
RAM1P	A single-port RAM.
RAM2P	A dual-port RAM.
RAMT2P	A true dual-port RAM, with support for both read and write on both the input and output side

Target RAM	Description
ROM1P	A single-port ROM.
ROM2P	A dual-port ROM.

Table 7 SystemC ap_mem_port Memory Targets

Once the `ap_mem_port` has been defined on the interface, the variables are simply accessed in the code in the same manner as any other arrays:

```
          dout[i] = share_mem[i] + din[i];
```

The test bench to support Example 71 is shown below in Example 72. The `ap_mem_port` type must be supported by an `ap_mem_chn` type in the test bench. The `ap_mem_chn` type is defined in the header file `ap_mem_if.h` and supports the same fields as `ap_mem_port`.

```
#ifdef __RTL_SIMULATION__
#include "sc_RAM_port_rtl_wrap.h"
#define sc_RAM_port sc_RAM_port_RTL_transactor
#else
#include "sc_RAM_port.h"
#endif
#include "tb_init.h"
#include "tb_driver.h"
#include "ap_mem_if.h"

int sc_main (int argc , char *argv[])
{
 sc_report_handler::set_actions("/IEEE_Std_1666/deprecated", SC_DO_NOTHING);
sc_report_handler::set_actions( SC_ID_LOGIC_X_TO_BOOL_, SC_LOG);
sc_report_handler::set_actions( SC_ID_VECTOR_CONTAINS_LOGIC_VALUE_, SC_LOG);
sc_report_handler::set_actions( SC_ID_OBJECT_EXISTS_, SC_LOG);

      sc_signal<bool>     s_reset;
      sc_signal<bool>     s_start;
      sc_signal<bool>     s_done;
      ap_mem_chn<int,int, 100, RAM2P> dout;
      ap_mem_chn<int,int, 100, RAM2P> din;

      // Create a 10ns period clock signal
      sc_clock s_clk("s_clk",10,SC_NS);

      tb_init      U_tb_init("U_tb_init");
      sc_RAM_port  U_dut("U_dut");
      tb_driver    U_tb_driver("U_tb_driver");

      // Generate a clock and reset to drive the sim
      U_tb_init.clk(s_clk);
      U_tb_init.reset(s_reset);
      U_tb_init.done(s_done);
      U_tb_init.start(s_start);
```

```
        // Connect the DUT
        U_dut.clock(s_clk);
        U_dut.reset(s_reset);
        U_dut.done(s_done);
        U_dut.start(s_start);
        U_dut.dout(dout);
        U_dut.din(din);

        // Drive inputs and Capture outputs
        U_tb_driver.clk(s_clk);
        U_tb_driver.reset(s_reset);
        U_tb_driver.start(s_start);
        U_tb_driver.done(s_done);
        U_tb_driver.dout(dout);
        U_tb_driver.din(din);

        // Sim
        int end_time = 1100;

        cout << "INFO: Simulating " << endl;

        // start simulation
        sc_start(end_time, SC_NS);

        if (U_tb_driver.retval != 0) {
                printf("Test failed  !!!\n");
        } else {
                printf("Test passed !\n");
   }
   return U_tb_driver.retval;
};
```

Example 72 SystemC RAM Interface Test Bench

FIFO Port Synthesis

FIFO ports on top-level interface can be synthesized directly from the standard SystemC `sc_fifo_in` and `sc_fifo_out` ports. An example on using FIFO ports on the interface is provided below (Example 73).

After synthesis each FIFO port will have a data port and associated FIFO control signals (inputs will have empty and read ports; outputs will have full and write ports). An advantage of using FIFO ports is that the handshake required to synchronize data transfers are automatically added in the RTL test bench.

```
#include"systemc.h"
#include"tlm.h"
using namespace tlm;


SC_MODULE(sc_FIFO_port)
{
        //Ports
        sc_in <bool>  clock;
        sc_in <bool>  reset;
        sc_in <bool>  start;
        sc_out<bool>  done;
        sc_fifo_out<int> dout;
        sc_fifo_in<int> din;

        //Variables
        int share_mem[100];
        bool write_done;

        //Process Declaration
        void Prc1();
        void Prc2();

        //Constructor
        SC_CTOR(sc_FIFO_port)
        {
                //Process Registration
                SC_CTHREAD(Prc1,clock.pos());
                reset_signal_is(reset,true);

                SC_CTHREAD(Prc2,clock.pos());
                reset_signal_is(reset,true);
        }
};
```

Example 73 SystemC FIFO Interface

Unsupported SystemC Constructs

Modules and Constructors

As mentioned above, but repeated here for reference, the following are not supported:

- An SC_MODULE cannot be nested inside another SC_MODULE.
- An SC_MODULE cannot be derived from another SC_MODULE.
- SC_THREAD is not supported (the clocked version, SC_CTHREAD is supported).

Instantiating Modules

An SC_MODULE cannot instantiated using `new`. Such code,

```
SC_MODULE(TOP)
{
        sc_in<T> din;
        sc_out<T> dout;

        M1 *t0;

        SC_CTOR(TOP){
                t0 = new M1("t0");
                t0->din(din);
                t0->dout(dout);
    }
}
```

Must be transformed to:

```
SC_MODULE(TOP)
{
       sc_in<T> din;
       sc_out<T> dout;

       M1 t0;

       SC_CTOR(TOP)
       : t0(“t0”)
       {
              t0.din(din);
              t0.dout(dout);
       }
}
```

Module Constructors

Only name parameters can be used with module constructors. The following passing on variable `temp` of type `int` is not allowed.

```
SC_MODULE(dut) {
  sc_in<int> in0;
  sc_out<int>out0;
  int var;
  SC_HAS_PROCESS(dut);
  dut(sc_module_name nm, int temp)
:sc_module(nm),var(temp)
 { …   }
};
```

Functions

Virtual Functions are not supported. The following code cannot be synthesized due to the use of the virtual function.

```
SC_MODULE(DUT)
{
       sc_in<int> in0;
       sc_out<int>out0;

       virtual int foo(int var1)
       {
              return var1+10;
       }

       void process()
       {
              int var=foo(in0.read());
              out0.write(var);
       }
       …
};
```

Top-Level Interface Ports

Reading an `sc_out` port is not supported. The following example is not allowed due to the read on `out0`.

```
SC_MODULE(DUT)
{
  sc_in<T> in0;
  sc_out<T>out0;
  …
  void process()
  {
   int var=in0.read()+out0.read();
   out0.write(var);
  }
};
```

C Arbitrary Precision Types

This chapter provides the details on the Arbitrary Precision (AP) types provided for C language design by AutoESL. The sections in the chapter provide details on the associated functions for C `int#w` 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.

Compiling [u]int#W Types

In order to use the `[u]int#W` types the "ap_cint.h" header file must be included 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()

This section also discusses use of the related functions:

- 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 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                     8198552921648689
printf("llu%40llu\n",c);
// llu                    8198552921648689

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                     443212636115274675
printf("llo%40llo\n",c);
// llo                    443212636115274675

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);       // 0000000000000000000054
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 C/C++ fundamental integer data type, e.g. char, short, int, etc.

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 `ap_[u]int` and C/C++ fundamental 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

[u]int#W::RType [u]int#W::**operator +** ([u]int#W op)

This operator produces the sum of two `ap_[u]int` (or one `ap_[u]int` and a C/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

[u]int#W::RType [u]int#W::**operator -** ([u]int#W 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

[u]int#W::RType [u]int#W::**operator *** ([u]int#W 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

[u]int#W::RType [u]int#W::**operator /** ([u]int#W 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

[u]int#W::RType [u]int#W::**operator %** (*[u]int#W 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.

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

Bitwise OR

[u]int#W::RType [u]int#W::**operator |** (*[u]int#W op*)

Returns the bitwise OR of the two operands.

Bitwise AND

[u]int#W::RType [u]int#W::**operator &** (*[u]int#W op*)

Returns the bitwise AND of the two operands.

Bitwise XOR

[u]int#W::RType [u]int#W::**operator ^** (*[u]int#W op*)

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, i.e. 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

[u]int#W [u]int#W::**operator <<** (*ap_uint<int_W2> op*)

Integer Shift Right

[u]int#W [u]int#W::**operator <<** (*ap_int<int_W2> op*)

Unsigned Integer Shift Left

[u]int#W [u]int#W::**operator >>** (*ap_uint<int_W2> op*)

Integer Shift Left

[u]int#W [u]int#W::**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.

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.

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 [u]int#W::**operator ==** (*[u]int#W op*)

Inequality

bool [u]int#W::**operator !=** (*[u]int#W op*)

Less than

bool [u]int#W::**operator <** (*[u]int#W op*)

Greater than

bool [u]int#W::**operator >** (*[u]int#W op*)

Less than or equal

bool [u]int#W::**operator <=** (*[u]int#W op*)

Greater than or equal

bool [u]int#W::**operator >=** (*[u]int#W op*)

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 `ap_[u]int` type variable(s).

Length

apint_bitwidthof()

```
int           apint_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 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 native C 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]<> Types

In order to use the `ap_[u]fixed<>` classes one must include the "ap_int.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] Variables

There are separate signed and unsigned classes: ap_int<int_W> & 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]fixed<> 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]fixed<>` variables using standard C/C++ integer literals.

However, this method of assigning values to `ap_[u]fixed<>` 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 (e.g. for those LL or ULL suffixes).

In order to allow assignment of values wider than 64-bits, the `ap_[u]fixed<>` 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 (i.e. 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 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]<>` 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]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. 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]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 & 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();      // => "40177334565141662503102O"
printf("%s\n", Val.to_string(16, true).c_str(); // => "-7F0123456789ABCDF0"
```

Expressions Involving ap_[u]<> types

Variables of `ap_[u]<>` 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 & 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 C/C++ fundamental integer data type, e.g. char, short, int, etc.

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 `ap_[u]int` and C/C++ fundamental 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 C/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 that 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 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, i.e. 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, i.e. 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 & 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_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, i.e. 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 check whether *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, I.e. the *i*th bit will become `0` if its original value is `1` and vice versa.

Example of bit set, clear & 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_int:: **sign**()

This member function checks whether the `ap_(u)int` instance is negative, returning true if negative and return false if positive.

Explicit Conversion Methods

To C/C++ "(u)int"

int ap_(u)int::**to_int** ()

unsigned ap_(u)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)int"

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`. Note that 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. The user is 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_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.
    - Step 1: First delete redundant MSBs.
    - Step 2: The new sign bit is the least significant bit of the deleted bits. 0 in this case.
    - Step 3: 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> & 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]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 & 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]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 & 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::RType 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::RType 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::RType 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::RType 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::RType 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: e.g `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 the operand is positive, a shift right 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 the operand is positive, a shift right 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_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_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_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_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_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_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
```