Tensilica Datasheet

Tensilica Software Development Toolkit (SDK)

Quickly develop application code

Features

- Cadence® Tensilica® Xtensa® Xplorer™ Integrated Development Environment (IDE) with full graphical user interface (GUI)
- Mature, optimizing Xtensa C/C++ Compiler (XCC)
- Operator overloading support in C for custom data types
- Pipeline-modeling, cycle-accurate instruction set simulator (ISS) with fast TurboXim
- GNU profiler, linker, debugger, assembler, and utilities
- Multi-processor subsystem simulation, debug, profiling, and memory partitioning
- Vectorization Assistant for locating code loops that need restructuring to enable vectorization
- Project management tools
- Performance and energy analysis tools
- Use Mentor Graphics NucleusPLUS, Express Logic's ThreadX, Micrium's uC/OS-II, T-Engines' µT-Kernel, or the Linux operating systems

Benefits

- Easy-to-use Xtensa Xplorer IDE based on familiar Eclipse platform
- Small, high-performance code from 'C' source
  - Compiler offers state-of-the-art inter-procedural and alias analysis
  - Automatic vectorization of operations for Xtensa SIMD processors
  - Automatic Flexible Length Instruction eXtension (FLIX) instruction bundling for multi-issue Xtensa very long instruction word (VLIW) cores
- Detailed pipeline analysis guides optimizations from cycle/pipeline-accurate ISS
- Fast TurboXim simulation for up to 50 million instructions per second
- Vectorization Assistant guides code optimizations for better SIMD performance
- Easily and quickly evaluate multi-processor subsystems
- Familiar GNU-based toolchain

Software Development Tools for Cadence Tensilica DPUs

If you've looked at Tensilica's website or processor product briefs, you know that you can extend Tensilica's Xtensa dataplane processors (DPUs)—adding instruction sets, execution units, and processor I/O interfaces—to match your specific application needs.

By customizing the DPU for a particular application, you can often get significantly lower energy consumption and 10-100X performance increases. This level of performance and efficiency is often essential in the SoC dataplane. By customizing the DPU, you create a core that's uniquely yours, giving you extra protection in today's highly competitive marketplace.

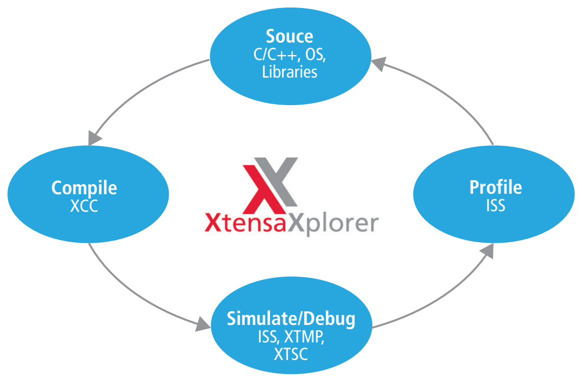


Figure 1: Tensilica's Eclipse-based Xtensa Xplorer IDE serves as the cockpit for custom processor development.

The Xtensa Processor Developer's Toolkit is the integrated design environment that delivers powerful tools to your desktop to guide you through the processor customization process. You'll find that Tensilica has created the most advanced, powerful, and easy-to-use tools for processor customization.

The Processor Developer's Toolkit is required for any design team that is using Tensilica's TIE instructions to modify the processor. If you are using an Xtensa processor with no modification or only changes to configuration options, you do not need the Processor Developer's Toolkit—you'll only need the Software Developer's Toolkit.

Designers get a compiler, linker, assembler, and debugger for their particular processor hardware. As the base instruction set architecture (ISA) is always present, third-party tools can still be used even when the core is customized for a particular application.

A Comprehensive System

Now in their 11th generation, Tensilica's tools for software development are highly refined and provide developers with a complete, comprehensive solution for both system design and software development, as illustrated in Figure 3.

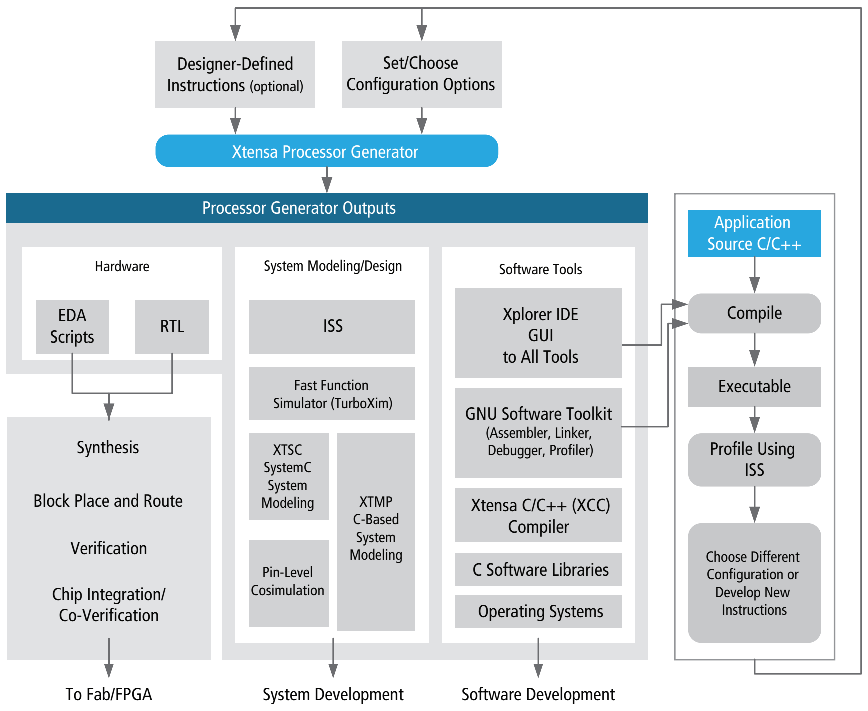
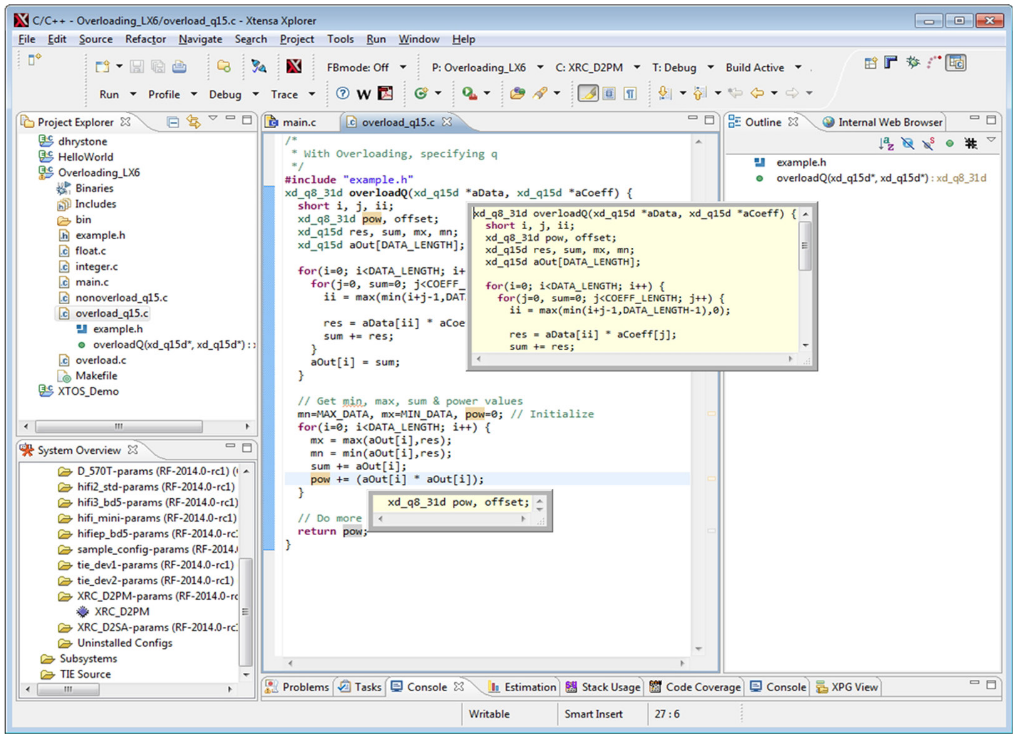

Designer-Defined Instructions (optional)
Set/Choose Configuration Options
Xtensa Processor Generator
Processor Generator Outputs
Hardware
EDA Scripts
RTL
Synthesis
Block Place and Route
Verification
Chip Integration/ Co-Verification
System Modeling/Design
ISS
Fast Function Simulator (TurboXim)
XTSC SystemC System Modeling
XTMP C-Based System Modeling
Pin-Level Cosimulation
Software Tools
Xplorer IDE GUI to All Tools
GNU Software Toolkit (Assembler, Linker, Debugger, Profiler)
Xtensa C/C++ (XCC) Compiler
C Software Libraries
Operating Systems
Application Source C/C++
Compile
Executable
Profile Using ISS
Choose Different Configuration or Develop New Instructions
To Fab/FPGA
System Development
Software Development

Figure 2. Tensilica's proven methodology automates the creation of customized processors and matching software tools.


C/C++ - Overloading_LX6/overload_q15.c - Xtensa Xplorer
File Edit Source Refactor Navigate Search Project Tools Run Window Help
FBmode: Off
P: Overloading_LX6
C: XRC_D2PM
T: Debug
Build Active
Run
Profile
Debug
Trace
Project Explorer
dhrystone
HelloWorld
Overloading_LX6
Binaries
Includes
bin
example.h
float.c
integer.c
main.c
nonoverload_q15.c
overload_q15.c
example.h
overloadQ(xd_q15d*, xd_q15d*) : :
overload.c
Makefile
XTOS_Demo
System Overview
D_570T-params (RF-2014.0-rc1)
hifi2_std-params (RF-2014.0-rc1)
hifi3_bd5-params (RF-2014.0-rc1)
hifi_mini-params (RF-2014.0-rc1)
hifiep_bd5-params (RF-2014.0-rc
sample_config-params (RF-2014.
tie_dev1-params (RF-2014.0-rc1)
tie_dev2-params (RF-2014.0-rc1)
XRC_D2PM-params (RF-2014.0-rc
XRC_D2PM
XRC_D2SA-params (RF-2014.0-rc
Uninstalled Configs
Subsystems
TIE Source
main.c
overload_q15.c
Outline
Internal Web Browser
example.h
overloadQ(xd_q15d*, xd_q15d*) : xd_q8_31d
Problems
Tasks
Console
Estimation
Stack Usage
Code Coverage
Console
XPG View
Writable
Smart Insert
27 : 6

```
/*
 * With Overloading, specifying q
 */
#include "example.h"
xd_q8_31d overloadQ(xd_q15d *aData, xd_q15d *aCoeff) {
  short i, j, ii;
  xd_q8_31d pow, offset;
  xd_q15d res, sum, mx, mn;
  xd_q15d aOut[DATA_LENGTH];

  for(i=0; i<DATA_LENGTH; i+
    for(j=0, sum=0; j<COEFF_
      ii = max(min(i+j-1,DAT

      res = aData[ii] * aCoe
      sum += res;
    }
    aOut[i] = sum;
  }

  // Get min, max, sum & power values
  mn=MAX_DATA, mx=MIN_DATA, pow=0; // Initialize
  for(i=0; i<DATA_LENGTH; i++) {
    mx = max(aOut[i],res);
    mn = min(aOut[i],res);
    sum += aOut[i];
    pow += (aOut[i] * aOut[i]);
  }

  // Do more
  return pow;
}
```

```
xd_q8_31d overloadQ(xd_q15d *aData, xd_q15d *aCoeff) {
  short i, j, ii;
  xd_q8_31d pow, offset;
  xd_q15d res, sum, mx, mn;
  xd_q15d aOut[DATA_LENGTH];

  for(i=0; i<DATA_LENGTH; i++) {
    for(j=0, sum=0; j<COEFF_LENGTH; j++) {
      ii = max(min(i+j-1,DATA_LENGTH-1),0);

      res = aData[ii] * aCoeff[j];
      sum += res;
```

```
xd_q8_31d pow, offset;
```

Figure 3. The editor includes many useful functions to speed up code generation and debugging

Project manager

When you start a new software project or modify an ongoing project, the project manager organizes all related project source files and allows you to create new classes, files, or folders. New projects can be managed using the built-in project-management and version-control mechanisms, which eliminate the need to manually maintain makefiles and provide a clean environment for new project builds. The project manager allows you to set all tool options and flags (build properties) for each build target within each individual project. Optionally, you can create unmanaged projects that allow total user control over build target properties.

Multi-processor subsystems

The Xtensa Xplorer IDE provides multi-processor projects that allow the designer to create a subsystem of heterogeneous cores with shared memory. The memory partitioning for each core and the shared memory area are specified in the GUI to make that task simple.

Simulation of the resultant system is launched from within the IDE and allows the software developer to debug, profile, and partition their code very quickly.

Source code editor

The C-code editor allows you to efficiently create and modify your code using rich editing capabilities. Recognition of language features such as keywords, comments, declarations, and strings are eased through syntax highlighting. Symbol indexing allows fast program navigation including find declaration, find definition, and find type. Other features in the editor that speed up coding include code completion, auto indenting, and quick diff. Block comment/uncomment is useful when debugging or profiling large source files, as is text folding for hiding areas of text that you don't need to view. Other standard views, such as source outline, make target, and problems are also available.

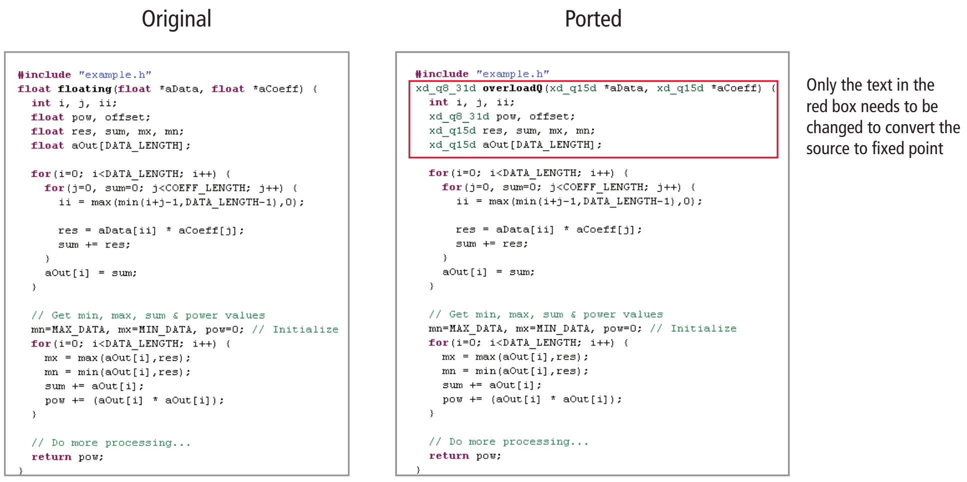
Xtensa compiler toolchain

Tensilica's Xtensa C/C++ compiler is based on the GNU compiler front-end with a highly customized code generation back-end (derived from the Open64 project) targeting the compact 16/24-bit Xtensa ISA. The Xtensa C/C++ compiler also includes support for the TIE language, including intermediate representation and optimization. The Xtensa C/C++ compiler additionally supports Tensilica's FLIX, allowing from 4-byte to 16-byte VLIW instruction bundles of up to 30 simultaneous instructions limited only by opcode availability.

The Xtensa C/C++ compiler employs sophisticated multi-level optimizations such as function inlining, software pipelining, static single assignment (SSA) optimizations, and other code generation techniques to reduce code size. All of these optimizations increase code execution speed and reduce code size. Based on industry-standard benchmarks, the Xtensa C/C++ compiler generates the highest code density when compared to compilers for other 32-bit RISC architectures.

The Xtensa C/C++ compiler provides the advanced optimization techniques known as feedback-directed optimization and interprocedural analysis.

Feedback-directed optimization is a two-step process where code is instrumented on the first pass of compilation and run using a representative input data set to produce a file containing profiling information. On the second pass, this profiling information is used to optimize application code to further reduce branch delays, improve inlining, and minimize the impact of register spills. The Xtensa C/C++ compiler will optimize an application's critical areas for performance while optimizing the remainder of the code for space. The Xtensa C/C++ compiler is also capable of hardware feedback-directed optimization, in which the user's target hardware platform can run the instrumented code to similarly provide application-specific optimization. Hardware feedback-directed optimization is a much faster method and the optimization is performed on the actual target system as opposed to simulated in the ISS.


Original
Ported
Only the text in the red box needs to be changed to convert the source to fixed point

```
#include "example.h"
float floating(float *aData, float *aCoeff) {
  int i, j, ii;
  float pow, offset;
  float res, sum, mx, mn;
  float aOut[DATA_LENGTH];

  for(i=0; i<DATA_LENGTH; i++) {
    for(j=0, sum=0; j<COEFF_LENGTH; j++) {
      ii = max(min(i+j-1,DATA_LENGTH-1),0);

      res = aData[ii] * aCoeff[j];
      sum += res;
    }
    aOut[i] = sum;
  }

  // Get min, max, sum & power values
  mn=MAX_DATA, mx=MIN_DATA, pow=0; // Initialize
  for(i=0; i<DATA_LENGTH; i++) {
    mx = max(aOut[i],res);
    mn = min(aOut[i],res);
    sum += aOut[i];
    pow += (aOut[i] * aOut[i]);
  }

  // Do more processing...
  return pow;
}
```

```
#include "example.h"
xd_q8_31d overloadQ(xd_q15d *aData, xd_q15d *aCoeff) {
  int i, j, ii;
  xd_q8_31d pow, offset;
  xd_q15d res, sum, mx, mn;
  xd_q15d aOut[DATA_LENGTH];

  for(i=0; i<DATA_LENGTH; i++) {
    for(j=0, sum=0; j<COEFF_LENGTH; j++) {
      ii = max(min(i+j-1,DATA_LENGTH-1),0);

      res = aData[ii] * aCoeff[j];
      sum += res;
    }
    aOut[i] = sum;
  }

  // Get min, max, sum & power values
  mn=MAX_DATA, mx=MIN_DATA, pow=0; // Initialize
  for(i=0; i<DATA_LENGTH; i++) {
    mx = max(aOut[i],res);
    mn = min(aOut[i],res);
    sum += aOut[i];
    pow += (aOut[i] * aOut[i]);
  }

  // Do more processing...
  return pow;
}
```

Figure 4. Operator overloading makes porting existing code easier

Interprocedural analysis is an optimization method that looks globally across all associated files of an application at link time. Global optimization is a much more powerful method than optimizing locally within an expression or procedure. Interprocedural analysis examines relationships across function calls, and can perform optimizations that cannot be achieved with a local scope. Interprocedural analysis eliminates unneeded computations, improves function inlining, and performs alias analyses that may not be performed by less sophisticated optimization techniques.

The Xtensa C/C++ compiler supports operator overloading on custom data types in the 'C' programming language (without the overhead that is often associated with it).

Tensilica is well known for its ability to let designers add custom instructions and data types to improve performance. If an application needs to work on 56-bit data, a designer can define a custom 56-bit data type with a single line of code. The designer can also specify what regular 'C' operators, such as '+' and '*', should do when using this data type. The overloading is always done with zero overhead so the resulting binaries are always efficient.

Porting and creating 'C' application code that uses custom data types is easier because standard 'C' operator syntax can be used. This makes the code easier to read and simpler to port via changes in the 'C' header files rather than throughout the source code itself. See Figure 4.

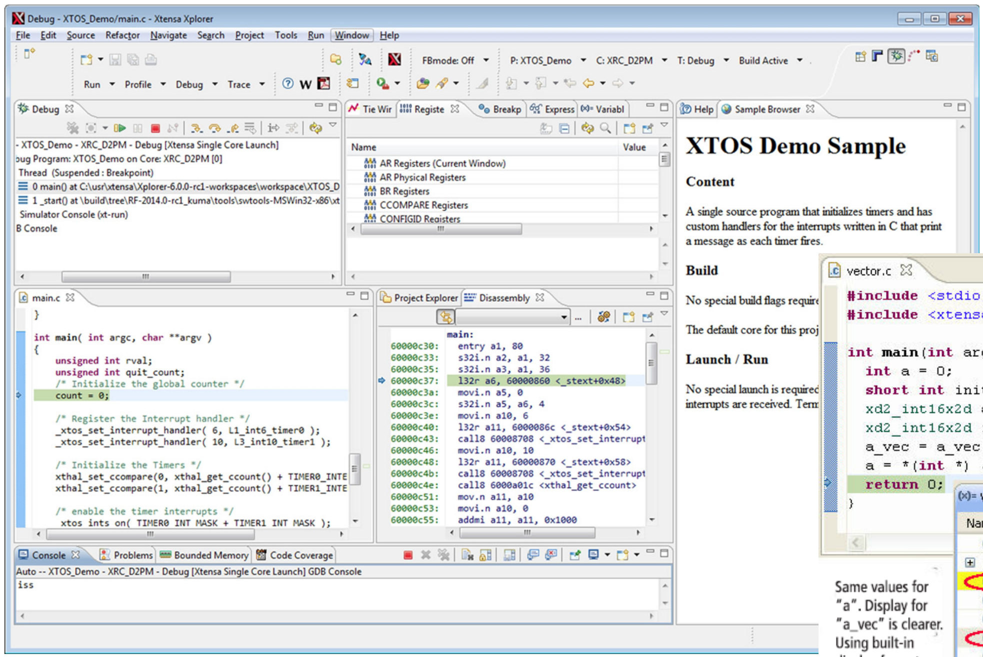
The rest of the software development toolchain is based on standard GNU tools. The compiler front-end remains similar to the preprocessor in the GNU tools, and the flags for the preprocessor remain the same. The assembler and linker also utilize the same flags as the GNU versions of the tools.

Xtensa debugger

The debugger allows you to target either the pipeline-/cycle-accurate ISS or TurboXim when no hardware is available, or external probes to connect with hardware development boards. As shown in Figure 5, the GUI-based debugger allows full system visibility into your project; it controls program execution and provides views to variables, breakpoints, memory, registers, etc. Source and assembly code can be made visible simultaneously while debugging an application, and either code window can be single stepped. The debugger interoperates seamlessly with the other development tools (compiler toolchain, ISS) to allow rapid code development for Xtensa processor systems.

Cores in multi-processor subsystems can be debugged and stepped synchronously or asynchronously with the other cores.

With user-defined data formatting, any data value can be re-formatted to display a more user-friendly representation. This is particularly effective when dealing with non-native 'C' types such as fixed point or vector data or when certain bits represent status. This data can be displayed in the Xtensa Xplorer IDE however you want using familiar print formatting. Datatypes that are defined by Tensilica in its DSP engines have default formatting that will show the user-friendly representations automatically. See Figure 6 for an example.



```
- XTOS_Demo - XRC_D2PM - Debug [Xtensa Single Core Launch]
bug Program: XTOS_Demo on Core: XRC_D2PM [0]
Thread (Suspended : Breakpoint)
0 main() at C:\usr\xtensa\Xplorer-6.0.0-rc1-workspaces\workspace\XTOS_D
1 _start() at \build\tree\RF-2014.0-rc1_kuma\tools\swtools-MSWin32-x86\xt
Simulator Console (xt-run)
B Console
```

AR Registers (Current Window)
AR Physical Registers
BR Registers
CCOMPARE Registers
CONFIGID Registers

XTOS Demo Sample

Content

A single source program that initializes timers and has custom handlers for the interrupts written in C that print a message as each timer fires.

Build

No special build flags requir

The default core for this proj

Launch / Run

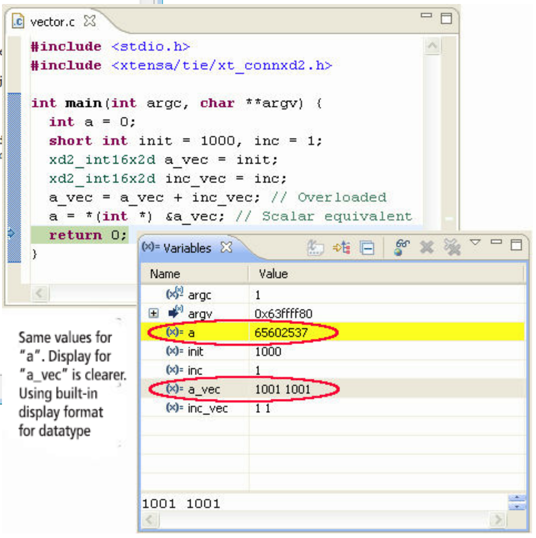
No special launch is required
interrupts are received. Term

```
}

int main( int argc, char **argv )
{
    unsigned int rval;
    unsigned int quit_count;
    /* Initialize the global counter */
    count = 0;

    /* Register the Interrupt handler */
    _xtos_set_interrupt_handler( 6, L1_int6_timer0 );
    _xtos_set_interrupt_handler( 10, L3_int10_timer1 );

    /* Initialize the Timers */
    xthal_set_ccompare(0, xthal_get_ccount() + TIMER0_INTE
    xthal_set_ccompare(1, xthal_get_ccount() + TIMER1_INTE

    /* enable the timer interrupts */
     xtos ints on( TIMER0 INT MASK + TIMER1 INT MASK );
```

```
            main:
60000c30:     entry a1, 80
60000c33:     s32i.n a2, a1, 32
60000c35:     s32i.n a3, a1, 36
60000c37:     l32r a6, 60000860 <_stext+0x48>
60000c3a:     movi.n a5, 0
60000c3c:     s32i.n a5, a6, 4
60000c3e:     movi.n a10, 6
60000c40:     l32r a11, 6000086c <_stext+0x54>
60000c43:     call8 60008708 <_xtos_set_interrupt
60000c46:     movi.n a10, 10
60000c48:     l32r a11, 60000870 <_stext+0x58>
60000c4b:     call8 60008708 <_xtos_set_interrupt
60000c4e:     call8 6000a01c <xthal_get_ccount>
60000c51:     mov.n a11, a10
60000c53:     movi.n a10, 0
60000c55:     addmi a11, a11, 0x1000
```

```
Auto -- XTOS_Demo - XRC_D2PM - Debug [Xtensa Single Core Launch] GDB Console
iss
```

Figure 5. The Xtensa debugger allows full visibility into the system

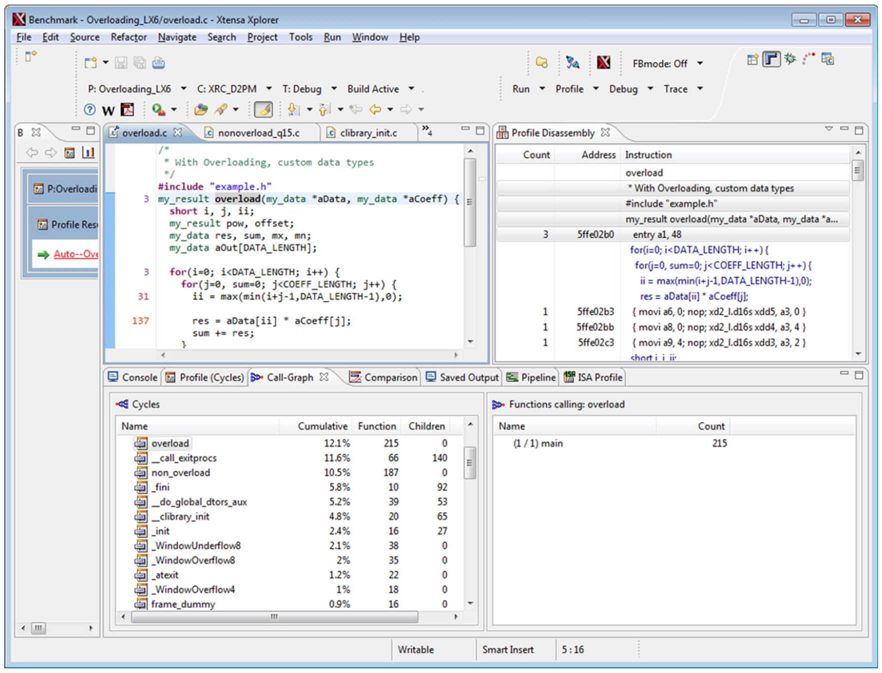

vector.c
Variables
Same values for "a". Display for "a_vec" is clearer. Using built-in display format for datatype

```
#include <stdio.h>
#include <xtensa/tie/xt_connxd2.h>

int main(int argc, char **argv) {
  int a = 0;
  short int init = 1000, inc = 1;
  xd2_int16x2d a_vec = init;
  xd2_int16x2d inc_vec = inc;
  a_vec = a_vec + inc_vec; // Overloaded
  a = *(int *) &a_vec; // Scalar equivalent
  return 0;
}
```

Name	Value
argc	1
argv	0x63ffff80
a	65602537
init	1000
inc	1
a_vec	1001 1001
inc_vec	1 1

```
1001  1001
```

Figure 6. Data can be reformatted the way you want it



```
    /*
     * With Overloading, custom data types
     */
    #include "example.h"
  3 my_result overload(my_data *aData, my_data *aCoeff) {
      short i, j, ii;
      my_result pow, offset;
      my_data res, sum, mx, mn;
      my_data aOut[DATA_LENGTH];

  3   for(i=0; i<DATA_LENGTH; i++) {
        for(j=0, sum=0; j<COEFF_LENGTH; j++) {
 31       ii = max(min(i+j-1,DATA_LENGTH-1),0);

137       res = aData[ii] * aCoeff[j];
          sum += res;
        }
```

Count	Address	Instruction
		overload
		* With Overloading, custom data types
		#include "example.h"
		my_result overload(my_data *aData, my_data *a...
3	5ffe02b0	entry a1, 48
		for(i=0; i<DATA_LENGTH; i++) {
		for(j=0, sum=0; j<COEFF_LENGTH; j++) {
		ii = max(min(i+j-1,DATA_LENGTH-1),0);
		res = aData[ii] * aCoeff[j];
1	5ffe02b3	{ movi a6, 0; nop; xd2_l.d16s xdd5, a3, 0 }
1	5ffe02bb	{ movi a8, 0; nop; xd2_l.d16s xdd4, a3, 4 }
1	5ffe02c3	{ movi a9, 4; nop; xd2_l.d16s xdd3, a3, 2 }
		short i, j, ii;

Cycles

Name	Cumulative	Function	Children
overload	12.1%	215	0
__call_exitprocs	11.6%	66	140
non_overload	10.5%	187	0
_fini	5.8%	10	92
__do_global_dtors_aux	5.2%	39	53
__clibrary_init	4.8%	20	65
_init	2.4%	16	27
_WindowUnderflow8	2.1%	38	0
_WindowOverflow8	2%	35	0
_atexit	1.2%	22	0
_WindowOverflow4	1%	18	0
frame_dummy	0.9%	16	0

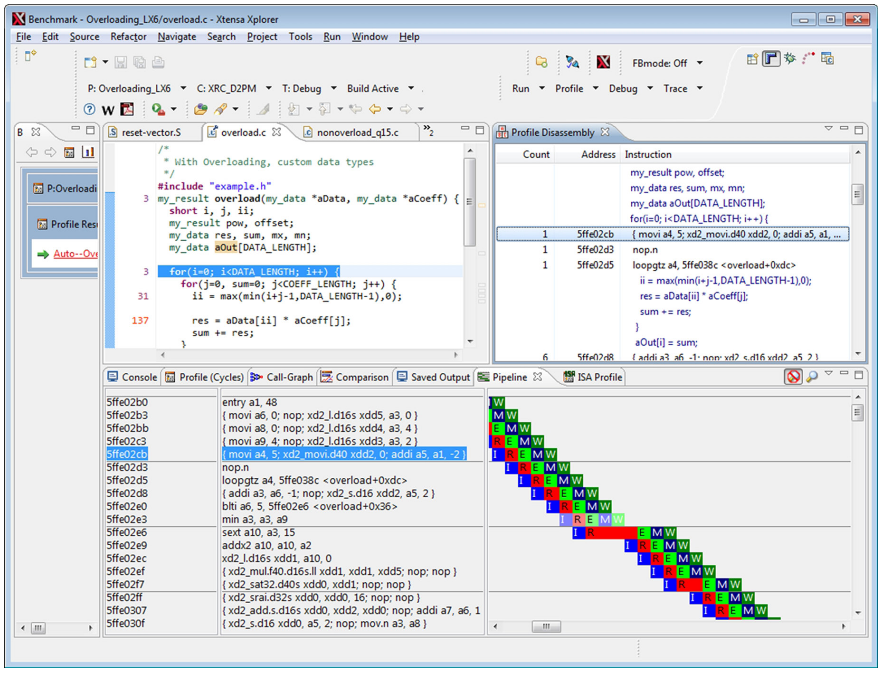
Functions calling: overload

Name	Count
(1 / 1) main	215

Figure 7. The profiling window allows performance metric analysis while optimizing code "hot spots"

Profiling tools

Code profiling is an extremely important tool for optimizing the performance of your application code. The Xtensa Xplorer IDE enables you to view profiling results generated by Tensilica's pipeline-accurate ISS (see Figure 7). Additionally, for much faster and more accurate profiling, you can generate profiling data from hardware instantiated in an FPGA or ASIC. You can track performance data such as instruction execution count, subroutine calls, subroutine total cycles, cache performance, etc. While viewing functions in the profiling view, you can also simultaneously view the assembly code in the disassembly view and the source code in the editor. The call graph view enables you to view the entire application hierarchy's caller and callee



```
    /*
     * With Overloading, custom data types
     */
    #include "example.h"
  3 my_result overload(my_data *aData, my_data *aCoeff) {
      short i, j, ii;
      my_result pow, offset;
      my_data res, sum, mx, mn;
      my_data aOut[DATA_LENGTH];

  3   for(i=0; i<DATA_LENGTH; i++) {
        for(j=0, sum=0; j<COEFF_LENGTH; j++) {
 31       ii = max(min(i+j-1,DATA_LENGTH-1),0);

137       res = aData[ii] * aCoeff[j];
          sum += res;
        }
```

Count	Address	Instruction
		my_result pow, offset;
		my_data res, sum, mx, mn;
		my_data aOut[DATA_LENGTH];
		for(i=0; i<DATA_LENGTH; i++) {
1	5ffe02cb	{ movi a4, 5; xd2_movi.d40 xdd2, 0; addi a5, a1, ...
1	5ffe02d3	nop.n
1	5ffe02d5	loopgtz a4, 5ffe038c <overload+0xdc>
		ii = max(min(i+j-1,DATA_LENGTH-1),0);
		res = aData[ii] * aCoeff[j];
		sum += res;
		}
		aOut[i] = sum;
6	5ffe02d8	{ addi a3, a6, -1; nop; xd2_s.d16 xdd2, a5, 2 }

```
5ffe02b0    entry a1, 48
5ffe02b3    { movi a6, 0; nop; xd2_l.d16s xdd5, a3, 0 }
5ffe02bb    { movi a8, 0; nop; xd2_l.d16s xdd4, a3, 4 }
5ffe02c3    { movi a9, 4; nop; xd2_l.d16s xdd3, a3, 2 }
5ffe02cb    { movi a4, 5; xd2_movi.d40 xdd2, 0; addi a5, a1, -2 }
5ffe02d3    nop.n
5ffe02d5    loopgtz a4, 5ffe038c <overload+0xdc>
5ffe02d8    { addi a3, a6, -1; nop; xd2_s.d16 xdd2, a5, 2 }
5ffe02e0    blti a6, 5, 5ffe02e6 <overload+0x36>
5ffe02e3    min a3, a3, a9
5ffe02e6    sext a10, a3, 15
5ffe02e9    addx2 a10, a10, a2
5ffe02ec    xd2_l.d16s xdd1, a10, 0
5ffe02ef    { xd2_mul.f40.d16s.ll xdd1, xdd1, xdd5; nop; nop }
5ffe02f7    { xd2_sat32.d40s xdd0, xdd1; nop; nop }
5ffe02ff    { xd2_srai.d32s xdd0, xdd0, 16; nop; nop }
5ffe0307    { xd2_add.s.d16s xdd0, xdd2, xdd0; nop; addi a7, a6, 1
5ffe030f    { xd2_s.d16 xdd0, a5, 2; nop; mov.n a3, a8 }
```

Figure 8. The pipeline viewer helps you understand instruction stalls and latency issues

functions. For those inner loop optimizations, the graphical pipeline view (Figure 8) shows any pipeline inefficiencies and bubbles that may be occurring.

Profiling of multi-processor subsystems shows each core side by side for easy load assessment and re-partitioning guidance. See Figure 9.

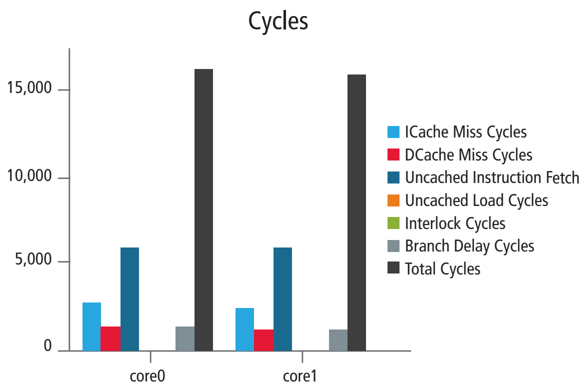


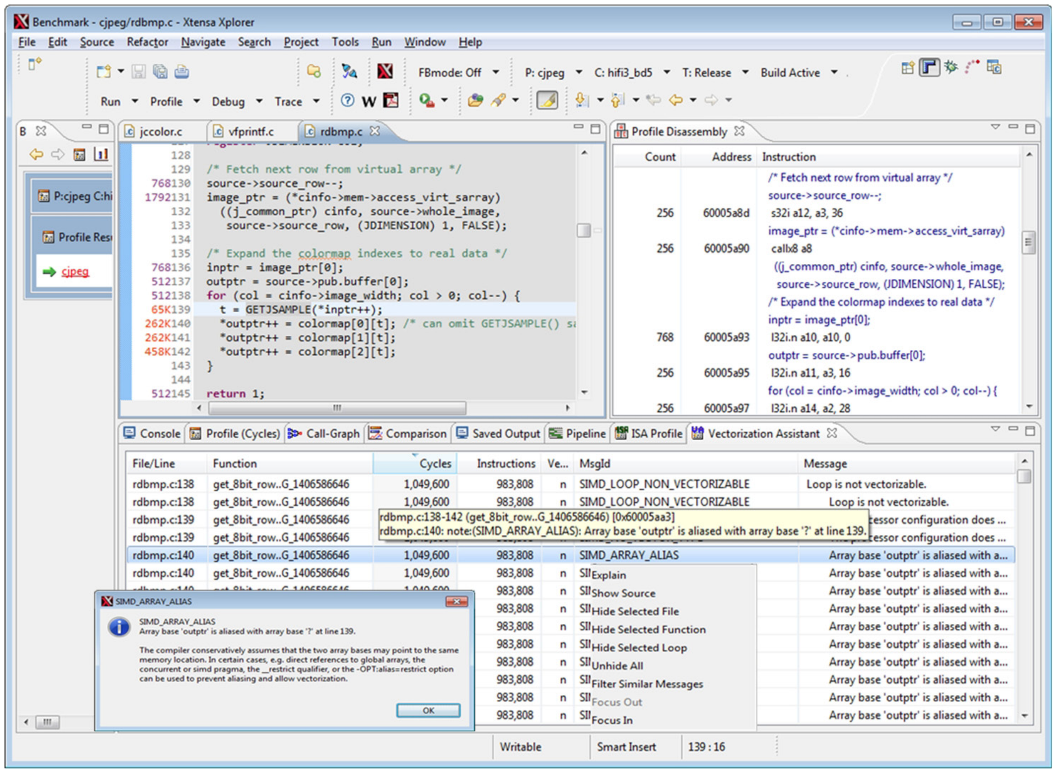
Figure 9. Multi-core profiling

Vectorization Assistant

Vectorization is the process of transforming the flow of your code (from the usual handling of one data item at a time) into a parallel loop that operates on multiple data items at once. The Xtensa compiler is capable of performing this transformation automatically, but you can help it exploit implicit parallelism in your code by eliminating certain patterns of data access that prevent successful vectorization.

Figure 10 shows how the Vectorization Assistant finds and displays loops in your code that could be "vectorized" by the compiler if the source was tweaked. Locating areas in the code that have not been vectorized, but could be, can take a long time looking at profiles, assembler, and pipeline views—then you have the task of doing the optimization to make it vectorize. In a few clicks, the Vectorization Assistant gets you to the loops in your source code that would benefit the most from vectorization.

The list of messages shown is initially sorted by the number of processor cycles used by a given loop, such that the most expensive loops appear first. You can focus the view on a particular file, folder, or project; you can filter out certain classes of messages that are not currently interesting; and you can hide messages that you do not wish to address at the moment.



```
       128
       129  /* Fetch next row from virtual array */
    768130  source->source_row--;
   1792131  image_ptr = (*cinfo->mem->access_virt_sarray)
       132    ((j_common_ptr) cinfo, source->whole_image,
       133     source->source_row, (JDIMENSION) 1, FALSE);
       134
       135  /* Expand the colormap indexes to real data */
    768136  inptr = image_ptr[0];
    512137  outptr = source->pub.buffer[0];
    512138  for (col = cinfo->image_width; col > 0; col--) {
    65K139    t = GETJSAMPLE(*inptr++);
   262K140    *outptr++ = colormap[0][t]; /* can omit GETJSAMPLE() sa
   262K141    *outptr++ = colormap[1][t];
   458K142    *outptr++ = colormap[2][t];
       143  }
       144
    512145  return 1;
```

Count	Address	Instruction
		/* Fetch next row from virtual array */
		source->source_row--;
256	60005a8d	s32i a12, a3, 36
		image_ptr = (*cinfo->mem->access_virt_sarray)
256	60005a90	callx8 a8
		((j_common_ptr) cinfo, source->whole_image,
		source->source_row, (JDIMENSION) 1, FALSE);
		/* Expand the colormap indexes to real data */
		inptr = image_ptr[0];
768	60005a93	l32i.n a10, a10, 0
		outptr = source->pub.buffer[0];
256	60005a95	l32i.n a11, a3, 16
		for (col = cinfo->image_width; col > 0; col--) {
256	60005a97	l32i.n a14, a2, 28

File/Line	Function	Cycles	Instructions	Ve...	MsgId	Message
rdbmp.c:138	get_8bit_row..G_1406586646	1,049,600	983,808	n	SIMD_LOOP_NON_VECTORIZABLE	Loop is not vectorizable.
rdbmp.c:138	get_8bit_row..G_1406586646	1,049,600	983,808	n	SIMD_LOOP_NON_VECTORIZABLE	Loop is not vectorizable.
rdbmp.c:139	get_8bit_row..G_1406586646					essor configuration does ...
rdbmp.c:139	get_8bit_row..G_1406586646					essor configuration does ...
rdbmp.c:140	get_8bit_row..G_1406586646	1,049,600	983,808	n	SIMD_ARRAY_ALIAS	Array base 'outptr' is aliased with a...
rdbmp.c:140	get_8bit_row..G_1406586646	1,049,600	983,808	n	SI	Array base 'outptr' is aliased with a...
			983,808	n	SI	Array base 'outptr' is aliased with a...
			983,808	n	SI	Array base 'outptr' is aliased with a...
			983,808	n	SI	Array base 'outptr' is aliased with a...
			983,808	n	SI	Array base 'outptr' is aliased with a...
			983,808	n	SI	Array base 'outptr' is aliased with a...
			983,808	n	SI	Array base 'outptr' is aliased with a...
			983,808	n	SI	Array base 'outptr' is aliased with a...
			983,808	n	SI	Array base 'outptr' is aliased with a...

```
rdbmp.c:138-142 (get_8bit_row..G_1406586646) [0x60005aa3]
rdbmp.c:140: note:(SIMD_ARRAY_ALIAS): Array base 'outptr' is aliased with array base '?' at line 139.
```

SIMD_ARRAY_ALIAS

SIMD_ARRAY_ALIAS
Array base 'outptr' is aliased with array base '?' at line 139.

The compiler conservatively assumes that the two array bases may point to the same memory location. In certain cases, e.g. direct references to global arrays, the concurrent or simd pragma, the __restrict qualifier, or the -OPT:alias=restrict option can be used to prevent aliasing and allow vectorization.

Figure 10. Vectorization Assistant helps find areas that can be improved

SoC Modeling

Many SoC designs today employ multiple processors. As SoC design becomes more complex, new methods to describe, debug, and profile overall system performance need to be employed. Unfortunately, most software development tools vendors do not provide pre-silicon simulation environments for multi-processor SoCs. Tensilica offers two modeling tools: XTensa Modeling Protocol (XTMP) for modeling in C and XTensa SystemC (XTSC) for modeling in SystemC.

Both tools are powerful additions to Tensilica's software development toolkit. They provide an Application Programming Interface (API) to the ISS, allowing fast and accurate simulation of SoC designs incorporating one or more processor cores. Running up to 10,000 times faster than RTL simulators, the XTMP/XTSC environments are potent tools for software development and SoC design. Both tools give you the ability to rapidly explore SoC partitioning alternatives and hardware/software performance tradeoffs. See Figure 11.

Modeling

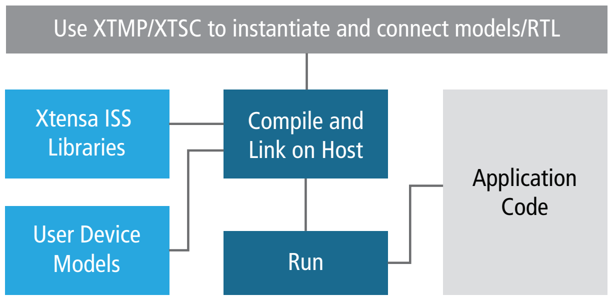
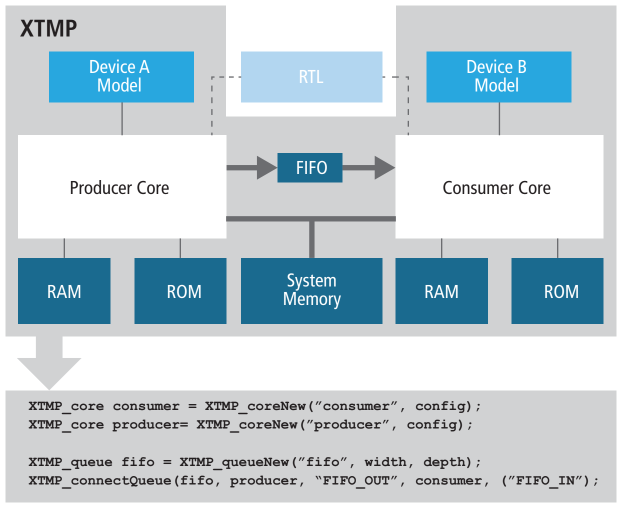


Figure 11. Using the ISS with XTMP or XTSC for modeling

XTMP and XTSC are used for simulating homogeneous or heterogeneous multi-processor design subsystems as well as complex uniprocessor architectures. Use the Xtensa Xplorer IDE's multi-processor project to instantiate multi-processor subsystems (or do it manually) and optionally connect them to custom peripherals and interconnects. You can create, debug, profile, and verify combined SoC and software architectures early in the design process. As the simulator operates at a higher level than HDL simulations, simulation time is cut drastically. See Figures 12 and 13.

XTMP and XTSC are integrated into the Xtensa Xplorer IDE, which automates the creation and development of multi-processor subsystem simulations. For XTMP, simulations are described in standard C code, which you can modify to allow more complex systems and additional simulator control if required. For XTSC, simulations are described in standard SystemC code. In addition, you have full visibility into all aspects of the simulation through the extensive API. Designers can use a single Xtensa Xplorer IDE to debug all simulated cores for additional visibility. The Xtensa Xplorer IDE manages all of these connections for you in its IDE for simplicity and easy viewing of any core.

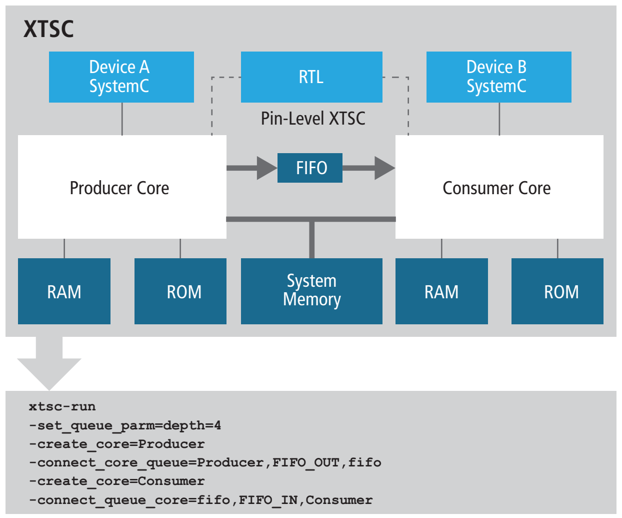
Modeling with XTMP



```
XTMP_core consumer = XTMP_coreNew("consumer", config);
XTMP_core producer= XTMP_coreNew("producer", config);

XTMP_queue fifo = XTMP_queueNew("fifo", width, depth);
XTMP_connectQueue(fifo, producer, "FIFO_OUT", consumer, ("FIFO_IN");
```

Figure 12. XTMP provides a simulation environment using instantiations of multi-processor-capable ISS, memory models, and connectors

Modeling with XTSC



```
xtsc-run
-set_queue_parm=depth=4
-create_core=Producer
-connect_core_queue=Producer,FIFO_OUT,fifo
-create_core=Consumer
-connect_queue_core=fifo,FIFO_IN,Consumer
```

Figure 13. With its pin-level modeling capabilities, XTSC allows co-simulation with Verilog

Modeling of local and system memory

XTMP and XTSC allow memory modeling of both local and system memory. System memory can have programmable latencies specified for different transaction types, allowing an accurate system simulation for analyzing performance tradeoffs. Memory-mapped peripherals may be included in an XTMP/XTSC system simulation, and functions are provided to connect the processor to peripheral devices.

Multi-threaded environment

An XTMP or XTSC simulation runs in a multi-threaded environment, with each processor running in its own thread. Core threads can be run asynchronously or synchronized through events using the attached debugger. Another option is to run all cores in lock-step, cycle-by-cycle mode. If one core stops on a break, all cores stop until it resumes. XTMP and XTSC have many options for implementing, controlling, and displaying results of system simulations deploying multiple cores, memories, and user-defined devices.

Pin-level SystemC modeling with Verilog

Additionally, Tensilica provides a link between its pipeline-accurate, cycle-accurate ISS and the leading Verilog simulators. Designers can now run pin-level SystemC co-simulations of Tensilica DPUs in their native Verilog simulators with pin-level XTSC, as seen in Figure 13.

Relative performance for different modeling levels

The wide range of choices allows customers to trade off speed versus model accuracy and pick the best type of model for the task at hand. Fast functional simulation gives the equivalent of 20-50 MIPS and is accurate to the CPU clock cycle level, while a full Verilog gate-level model may run only 10-100 cycles/sec but provides the accuracy needed to verify detailed timing. The range of modeling options and their estimated relative performance is summarized in Table 1.

Simulation Speed	Modeling Tool	Benefits
20 to 50 MIPS[1]	Standalone ISS in TurboXim mode	Fast functional simulation for rapid application testing
1.5 to 40 MIPS[1,2]	XTMP or XTSC in TurboXim mode	Fast functional simulation for rapid application testing at the system level
800K to 1,600K cycles/second	Standalone ISS in cycle-accurate mode	Software verification and cycle accuracy
600K to 1,000K cycles/second[2]	XTMP in cycle-accurate mode	Multi-core subsystem modeling and cycle accuracy
350K to 600K cycles/second[2]	XTSC in cycle-accurate mode	Multi-core subsystem modeling with SystemC interfaces and cycle accuracy
1K to 4K cycles/second	Verilog RTL simulation	Functional verification, pipeline-cycle accuracy and high visibility and accuracy
10 to 100 cycles/second	Verilog gate-level simulation	Timing verification, pipeline-cycle accuracy and high visibility and accuracy.

1. TurboXim mode simulation speed is an estimate for relatively long-running application programs (1 billion instructions or more)
2. Simulation speed is an estimate for a single Xtensa core in XTMP or XTSC
3. Running on a typical low-cost dual-core workstation

Table 1. Modeling Performance

Summary

The Xtensa Xplorer IDE is a complete GUI-based collection of tools that allows the software developer to create code for systems based on Xtensa processors. From project implementation to code generation to analysis, the Xtensa SDK enables you to achieve fast time-to-market while employing one of the most efficient 32-bit architectures available today. Xtensa processors lower total system costs and help design teams construct extremely high-performance system architectures.