INTRODUCTION

Most customers using the 29K family in embedded applications program in a high-level language. This is possible now like never before for a few simple reasons. First, RISC technology provides more than enough horsepower to overcome all the usual excuses for not programming in a high-level language. Second, the architecture of the 29K works to reduce the semantic gap between the compiler and the chip. This makes it possible for compilers to produce code of hand-generated quality. Third, the compiler technology available for the 29K family reduces the performance loss of typical high-level languages to a barely noticeable level.

While most, if not all, RISC microprocessor vendors make the same statement, there are many aspects of programming for the 29K family that are unique. The remainder of this paper discusses just a few of them.

INTRODUCTION TO THE 29K FAMILY FOR PROGRAMMERS

The 29K family is a family of binary compatible 32 bit RISC microprocessors. There are presently 5 members of the family, with more due soon. The Am29000 was the first member of the family and executes at speeds of 16 to 33MHz, with average cycles per instructions of a little over 1. The Am29005 is a reduced cost member of the family that removes the 512 byte cache and the memory management unit from the Am29000. It is offered only at 16MHz and only in a plastic package. The third member of the family is the Am29010. The Am29010 is pin compatible with the Am29000 with very high floating point performance on chip and some integer enhancements. It is available at speeds of 20 to 40MHz. The Am29030 and Am29035 are the latest members of the family. They offer 8K and 4K on chip instruction caches respectively and simplified hardware interfaces. The family performance range is from less than 10,000 dhrystones per second to over 80,000 dhrystones per second.
PROGRAMMING THE AM29000 FAMILY

Richard A. Relph
Advanced Micro Devices
Austin, TX

Richard A. Relph is a senior member of the technical staff at Advanced Micro Devices. He works in the Embedded Processor Division defining and arbitrating standards for use with Am29000 family software tools. Richard also assists key customers with special programming requirements.

INTRODUCTION

Most customers using the 29K family in embedded applications program in a high-level language. This is possible now like never before for a few simple reasons. First, RISC technology provides more than enough horsepower to overcome all the usual excuses for not programming in a high-level language. Second, the architecture of the 29K works to reduce the semantic gap between the compiler and the chip. This makes it possible for compilers to produce code of hand-generated quality. Third, the compiler technology available for the 29K family reduces the performance loss of typical high-level languages to a barely noticeable level.

While most, if not all, RISC microprocessor vendors make the same statement, there are many aspects of programming for the 29K family that are unique. The remainder of this paper discusses just a few of them.

INTRODUCTION TO THE 29K FAMILY FOR PROGRAMMERS

The 29K family is a family of binary compatible 32 bit RISC microprocessors. There are presently 5 members of the family, with more due soon. The Am29000 was the first member of the family and executes at speeds of 16 to 33MHz, with average cycles per instructions of a little over 1. The Am29030 is a reduced cost member of the family that removes the 512 byte cache and the memory management unit from the Am29000. It is offered only at 16MHz and only in a plastic package. The third member of the family is the Am2930. The Am29030 is pin compatible with the Am29000 with very high floating point performance on chip and some integer enhancements. It is available at speeds of 20 to 40MHz. The Am29030 and Am29035 are the latest members of the family. They offer 8K and 4K on chip instruction caches respectively and simplified hardware interfaces. The family performance range is from less than 10,000 dhrystoness per second to over 80,000 dhrystoness per second.
The 29K family can be programmed in several languages, from assembly to Ada. Like most RISC microprocessors today, though, the language of choice for programming the 29K family is C. But C for the 29K isn’t just C, it is more. Programming for the Am29K family of 32-bit RISC microprocessors is especially similar to programming for the 32-bit microprocessor you are using now. I say especially because we’ve taken pains to make it so. As anyone who has ported an application of any size from one tool set to another already knows, C can be less than perfectly portable. Despite the many improvements made to C over the years, none have made the language more compatible with existing code. While no compiler can be perfectly compatible with all other compilers, there are things that can be done to ensure that the porting goes quickly and easily with a minimum of recoding necessary.

ANSI vs. "old" C

There are 2 major variants of the C programming language. The first widely used and copied variant was the portable C compiler from Unix. The second is ANSI C. The dominant C compiler (our C compiler from now on) in use for the 29K provides support for both.

Where existing code is being ported, the ability to tune our compiler to match closely the semantics of the old compiler can save weeks chasing subtle problems. These can be caused by things most programmers aren’t even aware of. An example would be value preserving versus unsigned preserving promotion rules.

Signed vs. unsigned chars

An amazing amount of old C code relied on char’s on VAXs being signed. An almost as large body of code from embedded systems is optimized for the unsigned chars preferred by the 68K family. Our processors and our compiler support both equally well.

Prototypes

ANSI C provides a very useful new feature called prototypes. Prototypes enable type-safe compilation and can help detect bugs during compilation rather than during the debug phase. The biggest problem with prototypes is that they seem to require changing every function definition in your entire program. Worse, if you do, the program is no longer compilable by non-ANSI compilers.

Our compiler provides the tools and implements the undefined behavior areas of the ANSI standard to enable an easy transition to the benefits of prototypes without rendering the code un compilable by non-ANSI compilers. This is done via two special compiler features.

To get the benefits of prototypes with the 29K compiler without ANSI-ifying your code, you first "compile" all of the source files with the -Hprint_protos toggle on, redirecting standard out to a file, say "protos.h".

Then, create a file called hc.pro that simply #includes protos.h. hc.pro is a file that our compiler automatically prepends to each of the files it compiles. At this point a compiler will see something like:

```c
double foo( float a, double b, int c );
double foo( a, b, c );
float a;
double b;
int c;
{ }
```

The problem is that ANSI doesn’t define the semantics of this sequence. It falls into the somewhat large region of undefined behavior. But, undefined behavior isn’t illegal, it’s just not mandated by the ANSI standard what should be done. Our compiler chooses to assign a meaning to the program that is equivalent to:

```c
double foo( float a, double b, int c );
double foo( float a, double b, int c );
{ }
```

Most other compilers simply complain that the function definition is incompatible with the earlier function declaration and give up. Our compiler can be configured to warn about the use of this special feature so that if you are trying to produce ANSI conformant code, you can.

`alloca()`

`alloca()` is a very useful construct that is very un-portable, but even harder to model using portable constructs. `alloca()` can’t be implemented practically on some machines, so ANSI couldn’t make it a part of the ANSI standard. That didn’t cause existing code that uses it to disappear. Nor did it eliminate the very good reasons for using it.
C IS THE PRIMARY LANGUAGE

The 29K family can be programmed in several languages, from assembly to Ada. Like most RISC microprocessors today, though, the language of choice for programming the 29K family is C. But C for the 29K isn’t just C, it is more. Programming for the Am29K family of 32-bit RISC microprocessors is especially similar to programming for the 32-bit microprocessor you are using now. I say especially because we've taken pains to make it so. As anyone who has ported an application of any size from one tool set to another already knows, C can be less than perfectly portable. Despite the many improvements made to C over the years, none have made the language more compatible with existing code. While no compiler can be perfectly compatible with all other compilers, there are things that can be done to ensure that the porting goes quickly and easily with a minimum of recoding necessary.

ANSI vs. "old" C

There are 2 major variants of the C programming language. The first widely used and copied variant was the portable C compiler from Unix. The second is ANSI C. The dominant C compiler (our C compiler from now on) in use for the 29K provides support for both.

Where existing code is being ported, the ability to tune our compiler to match closely the semantics of the old compiler can save weeks chasing subtle problems. These can be caused by things most programmers aren’t even aware of. An example would be value preserving versus unsigned preserving promotion rules.

Signed vs. unsigned chars

An amazing amount of old C code relied on char’s on VAXs being signed. An almost as large body of code from embedded systems is optimized for the unsigned chars preferred by the 68K family. Our processors and our compiler support both equally well.

Prototypes

ANSI C provides a very useful new feature called prototypes. Prototypes enable type-safe compilation and can help detect bugs during compilation rather than during the debug phase. The biggest problem with prototypes is that they seem to require changing every function definition in your entire program. Worse, if you do, the program is no longer compilable by non-ANSI compilers.

Our compiler provides the tools and implements the undefined behavior areas of the ANSI standard to enable an easy transition to the benefits of prototypes without rendering the code un compilable by non-ANSI compilers. This is done via two special compiler features.

To get the benefits of prototypes with the 29K compiler without ANSI-ifying your code, you first “compile” all of the source files with the -Hprint_protos toggle on, redirecting standard output to a file, say "protos.h".

Then, create a file called hc.pro that simply #includes protos.h. hc.pro is a file that our compiler automatically prepends to each of the files it compiles. At this point a compiler will see something like:

```c
double foo( float a, double b, int c );
double foo( a, b, c )
float a;
double b;
int c;
{ }
```

The problem is that ANSI doesn’t define the semantics of this sequence. It falls into the somewhat large region of undefined behavior. But, undefined behavior isn’t illegal, it’s just not mandated by the ANSI standard what should be done. Our compiler chooses to assign a meaning to the program that is equivalent to:

```c
double foo( float a, double b, int c );
double foo( float a, double b, int c )
{ }
```

Most other compilers simply complain that the function definition is incompatible with the earlier function declaration and give up. Our compiler can be configured to warn about the use of this special feature so that if you are trying to produce ANSI conformant code, you can.

alloca()

alloca() is a very useful construct that is very un-portable, but even harder to model using portable constructs. alloca() can’t be implemented practically on some machines, so ANSI couldn’t make it a part of the ANSI standard. That didn’t cause existing code that uses it to disappear. Nor did it eliminate the very good reasons for using it.
alloca(), for those of you who haven't seen it, is a very fast memory allocation function. It is usually recognized by the compiler and in-lined and results in allocating memory in under a dozen instructions. Not only is it fast, but memory leakage is impossible. Memory leakage is a result of forgetting to call free() to deallocate a malloc()'d block of memory. Memory leakage is prevented because the allocated memory is actually on the stack, so when a return from the function that called alloca() is executed, the allocated memory is freed right with the memory reserved by the compiler for the automatic variables of the function.

This is what makes the function so hard to model with malloc() and free(). While you can always replace the alloca() with a malloc(), it is not possible to guarantee the free() takes place. Freeing of memory allocated by alloca() occurs not only at the return statements in the function, but also whenever longjmp() is called with a jmpbuf set by a function higher in the call tree. alloca() is not always what you need, but where you must have a fast, leakage-safe memory allocation function, it is very handy.

Existing code that uses alloca() can be difficult to port to a compiler that doesn't support alloca(). Naturally enough, our compiler does support alloca().

Complete warnings

Finding bugs at the first opportunity can go a long way to reducing development times significantly. If a bug can be pointed out during compilation, it will take much, much less time to find than if that bug must be found by debugging. To help in finding code that is possibly the result of unintended programming, our compiler provides warning about such things as variables that may be used before they are assigned to and code that can never be executed. Static functions and all variables that are defined (as opposed to declared) but never referenced are flagged. There are many, many more warnings that the compiler produces to provide the programmer with hints about places where bugs might exist.

Not only does our compiler produce a myriad of helpful warning messages, but they are quite complete. Many compilers simply say that a variable or function definition or declaration conflicts with an earlier declaration or definition. Our compiler tells you exactly where both declarations are in the module by line and character position.

Creating makefiles

make is a wonderful utility for ensuring that the object files that make up a program are up to date with respect to the source files that make them up. The only problem is that keeping the makefile dependency lists up to date is frequently forgotten about when a change is made. Our compiler includes the capability to generate dependency lists for makefiles automatically, including generating dependencies for include files that include other files.

C and assembler interlisting

Desk checking compiler generated code is usually an early step toward convincing yourself that a bug that doesn't occur in other systems running the same source is really your bug or a compiler bug. To make this easier, and to provide a way to see what the compiler is doing with your code, our compiler produces listings that interleave source and assembler. The assembler output includes comments beside the assembly code to show variables that are being referenced. What's more, this file is assembleable so that hand optimizations can be made in context.

29K FAMILY SPECIFIC FEATURES

Our compiler also implements several features designed to make it easy to take advantage of the 29K family. For the most part, these features are in the area of access to instructions and resources that are not adequately modeled in C.

_extract(), _clz(), _cpbyte()

All members of the 29K family include a funnel shifter. The funnel shifter extracts any 32 adjacent bits from a 64-bit value. This is extremely useful in applications, such as laser printers and graphics displays, that need to do bit alignment. C doesn't provide such an operator, so these types of applications usually include code that looks like:

\[
result = (\text{left} \ll n) \mid (\text{right} \gg (32-n))
\]

For the 29K, this can be executed in 2 instructions, 1 that sets \(n\) into a special register, and 1 that computes the result. This would look like:

\[
\text{mscr}( FC, n ); \quad \text{result} = \_\text{extract}( \text{left}, \text{right} );
\]

We could have made \(n\) a parameter of the _extract() function, but many applications need to set \(n\) once and then do many extracts. Note also that \(n, \text{left}, \text{and right}\) can all be arbitrary C expressions.
alloca(), for those of you who haven't seen it, is a very fast memory allocation function. It is usually recognized by the compiler and in-lined and results in allocating memory in under a dozen instructions. Not only is it fast, but memory leakage is impossible. Memory leakage is a result of forgetting to call free() to deallocate a malloc()d block of memory. Memory leakage is prevented because the allocated memory is actually on the stack, so when a return from the function that called alloca() is executed, the allocated memory is freed right with the memory reserved by the compiler for the automatic variables of the function.

This is what makes the function so hard to model with malloc() and free(). While you can always replace the alloca() with a malloc(), it is not possible to guarantee the free() takes place. Freeing of memory allocated by alloca() occurs not only at the return statements in the function, but also whenever longjmp() is called with a jmpbuf set by a function higher in the call tree. alloca() is not always what you need, but where you must have a fast, leakage-safe memory allocation function, it is very handy.

Existing code that uses alloca() can be difficult to port to a compiler that doesn't support alloca(). Naturally enough, our compiler does support alloca().

Complete warnings

Finding bugs at the first opportunity can go a long way to reducing development times significantly. If a bug can be pointed out during compilation, it will take much, much less time to find than if that bug must be found by debugging. To help in finding code that is possibly the result of unintended programming, our compiler provides warning about such things as variables that may be used before they are assigned to and code that can never be executed. Static functions and all variables that are defined (as opposed to declared) but never referenced are flagged. There are many, many more warnings that the compiler produces to provide the programmer with hints about places where bugs might exist.

Not only does our compiler produce a myriad of helpful warning messages, but they are quite complete. Many compilers simply say that a variable or function definition or declaration conflicts with an earlier declaration or definition. Our compiler tells you exactly where both declarations are in the module by line and character position.

Creating makefiles

make is a wonderful utility for ensuring that the object files that make up a program are up to date with respect to the source files that make them up. The only problem is that keeping the makefile dependency lists up to date is frequently forgotten about when a change is made. Our compiler includes the capability to generate dependency lists for makefiles automatically, including generating dependencies for include files that include other files.

C and assembler interlisting

Desk checking compiler generated code is usually an early step toward convincing yourself that a bug that doesn't occur in other systems running the same source is really your bug or a compiler bug. To make this easier, and to provide a way to see what the compiler is doing with your code, our compiler produces listings that interleave source and assembler. The assembler output includes comments beside the assembly code to show variables that are being referenced. What's more, this file is assembleable so that hand optimizations can be made in context.

29K FAMILY SPECIFIC FEATURES

Our compiler also implements several features designed to make it easy to take advantage of the 29K family. For the most part, these features are in the area of access to instructions and resources that are not adequately modeled in C.

_extract(), _clz(), _cpbyte()

All members of the 29K family include a funnel shifter. The funnel shifter extracts any 32 adjacent bits from a 64-bit value. This is extremely useful in applications, such as laser printers and graphics displays, that need to do bit alignment. C doesn't provide such an operator, so these types of applications usually include code that looks like:

result = (left << n) | (right >> (32-n));

For the 29K, this can be executed in 2 instructions, 1 that sets n into a special register and 1 that computes the result. This would look like:

_mtsr( FC, n );
_result = _extract( left, right );

We could have made n a parameter of the _extract() function, but many applications need to set n once and then do many extracts. Note also that n, left, and right can all be arbitrary C expressions.
Similar instructions in the 29K family are the CLZ instruction, that counts the number of leading 0 bits in a value, and the CPBYTE instruction that does 4 8-bit compares in a single cycle. These instructions are similarly accessible at the C level.

```c
__Am29000Regs[]
```

The Am29000 family has a uniquely large register set. There are 192 on-chip general purpose registers. Up to 128 of them may be used as a stack cache to keep most of the data needed by a function in the register file whenever the function is active. The remaining 64 registers are divided into two subsets. 32 of them are used just like registers in other microprocessors like the 68000. 4 of these are reserved for you, the applications programmer. The remaining 28 are managed by the compiler. That is, they are compiler temporaries, return values from functions, and stack management values. The last 32 registers are reserved for the operating system. The 36 registers that the compiler doesn’t use (the 4 reserved for the applications programmer plus the 32 reserved for the operating system) are still accessible from C.

Access to them is via the pseudo-array __Am29000Regs[]. When the compiler sees a reference to __Am29000Regs[112], for example, it treats that as register number 112 (called gr112). To use these registers to hold a global data GlobalVariable item accessed from several functions in a program, this sequence would do it without requiring changes throughout the code.

```c
extern int GlobalVariable;
extern int __Am29000Regs[];
#define GlobalVariable (__Am29000Regs[112])
```

From here on out, whenever the program contains a reference to globalVariable, the preprocessor will translate those into accesses of __Am29000Regs[112] which the compiler will map onto gr112 in the processor.

```c
_mtsr(), _mfsr()
```

Besides the 192 general purpose registers, the Am29000 family contains a few special registers. These registers are used for such things as the 4th operand to certain instructions, maintenance of the user/supervisor boundary, and exception management. The _mtsr() and _mfsr() functions provide access to these registers from C.

**ASSEMBLER ACCESS**

The features and facilities of our C compiler described so far are more than 99% of what applications programmers need. But, occasionally, a programmer may wish to drop down to assembly language. For those rare cases, we have two different ways to include assembly language directly in C source. Naturally, C can call assembler or vice-versa, but that doesn’t always do the trick.

```c
__asm ret_type( arg1 ) { ... }
```

The first way to get to the assembly level from C is by using _asm function types. _asm functions are functions that have return values and arguments, just like C functions, but they get inlined and optimized with the C code that calls them. The body of the function may include code specialized to deal with three different classes of arguments - constants, values in registers, and values in memory. This allows the function to execute different code if some parameters are constants instead of memory based values, for example. Because the compiler makes this determination at compile time and does in-line the assembly code, no performance is lost at run-time making these sorts of decisions.

```c
_asm( "string" )
```

The second way to get at assembler level is to simply inject a call to the _asm() pseudo-function. The function takes as its only parameter a string that the compiler emits into the assembly output unmodified. This could be useful for testing robustness of code by inserting an instruction that forces an interrupt condition.

**SYSTEM SUPPORT FEATURES**

Little vs. big endian

The 29K family and its tools are flexible. While most microprocessors are either little endian (DEC/Intel byte order) or big endian (IBM/Motorola byte order), the 29K family supports both. This allows 29K-based embedded systems that communicate directly with other CPUs to adapt to the byte order of the other CPU.
Similar instructions in the 29K family are the CLZ instruction, that counts the number of leading 0 bits in a value, and the CPBYTE instruction that does 4 8-bit compares in a single cycle. These instructions are similarly accessible at the C level.

`__Am29000Regs[]`

The Am29000 family has a uniquely large register set. There are 192 on-chip general purpose registers. Up to 128 of them may be used as a stack cache to keep most of the data needed by a function in the register file whenever the function is active. The remaining 64 registers are divided into two subsets. 32 of them are used just like registers in other microprocessors like the 68000. 4 of these are reserved for you, the applications programmer. The remaining 28 are managed by the compiler. That is, they are compiler temporaries, return values from functions, and stack management values. The last 32 registers are reserved for the operating system. The 36 registers that the compiler doesn't use (the 4 reserved for the applications programmer plus the 32 reserved for the operating system) are still accessible from C.

Access to them is via the pseudo-array `__Am29000Regs[]`. When the compiler sees a reference to `__Am29000Regs[112]`, for example, it treats that as register number 112 (called gr112). To use these registers to hold a global data `GlobalVariable` item accessed from several functions in a program, this sequence would do it without requiring changes throughout the code.

```c
extern int GlobalVariable;
extern int __Am29000Regs[];
define GlobalVariable (__Am29000Regs[112])
```

From here on out, whenever the program contains a reference to `GlobalVariable`, the preprocessor will translate those into accesses of `__Am29000Regs[112]` which the compiler will map onto gr112 in the processor.

`_mter(), _mfsr()`

Besides the 192 general purpose registers, the Am29000 family contains a few special registers. These registers are used for such things as the 4th operand to certain instructions, maintenance of the user/supervisor boundary, and exception management. The `_mter()` and `_mfsr()` functions provide access to these registers from C.

### ASSEMBLER ACCESS

The features and facilities of our C compiler described so far are more than 99% of what applications programmers need. But, occasionally, a programmer may wish to drop down to assembly language. For those rare cases, we have two different ways to include assembly language directly in C source. Naturally, C can call assembler or vice-versa, but that doesn't always do the trick.

```c
__Am max_type( args ) {}  
```

The first way to get to the assembly level from C is by using `_Am` function types. `_Am` functions are functions that have return values and arguments, just like C functions, but they get inlined and optimized with the C code that calls them. The body of the function may include code specialized to deal with three different classes of arguments - constants, values in registers, and values in memory. This allows the function to execute different code if some parameters are constants instead of memory based values, for example. Because the compiler makes this determination at compile time and does in-line the assembly code, no performance is lost at run-time making these sorts of decisions.

```c
_ASM( "string" )
```

The second way to get at assembler level is to simply inject a call to the `_ASM()` pseudo-function. The function takes as its only parameter a string that the compiler emits into the assembly output unmodified. This could be useful for testing robustness of code by inserting an instruction that forces an interrupt condition.

### SYSTEM SUPPORT FEATURES

**Little vs. big endian**

The 29K family and its tools are flexible. While most microprocessors are either little endian (DEC/Intel byte order) or big endian (IBM/Motorola byte order), the 29K family supports both. This allows 29K-based embedded systems that communicate directly with other CPUs to adapt to the byte order of the other CPU.
As mentioned earlier, up to 128 of the 192 general purpose registers can be used as a stack cache. The design of the calling convention requires that at least 34 registers be used as a stack cache. Using the maximum number will result in the best possible execution of the application. Occasionally, the system software architect may decide that some registers would be better used to reduce interrupt latency by reserving them for interrupt contexts. The compiler must be told how many registers are available for its use as local variables. The -lregs compiler option provides a means for the user to specify how many stack cache registers are available to the compiler.

Signals

Most embedded systems deal with interrupts on a regular part of what they do. The Am29000's RISC architecture provides many different ways to deal with interrupts. One of them is to turn the interrupt into a signal to a foreground task. Our C run-time library and sample systems code includes all the code necessary to register signal handlers and raise signals from interrupt context. We can support signal functions that never return (they might do a longjmp()) and signal functions that always return.

OS access

While many embedded applications don't require the services of a file system, some do and almost all would benefit from having one available during debugging. Our C run-time library and sample systems code includes all the code necessary to register signal handlers and raise signals from interrupt context. We can support signal functions that never return (they might do a longjmp()) and signal functions that always return.

ANSI library

The library shipped with our C compiler is fully ANSI confromant except for the system() function. Source code for those parts written in assembler is provided. The library includes optimized string and memory functions, plus high-performance floating point functions for both single and double precision. The math libraries are different for the Am29050 and other 29K family processors that don't have on-chip floating point.

DEBUGGING TOOLS

The focus of this paper has been getting programs to an executable state and then optimizing the performance of that code by using 29K family special features. But getting code to compile and link for a new embedded system is only part of the battle. Debugging and performance tuning is a frequent problem when using a high-end embedded processor. The 29K family has the broadest support for tools to help in these phases of project development. In-circuit emulators have proven themselves to be more than worth the price in projects where time-to-market is a consideration. At least one emulator provides complete, non-statistical execution (as opposed to fetch) histograms. Two provide code coverage capability. All emulators work with one or more source level debuggers. Many attach to a network directly. There are companies supplying logic analyzer adapters and resident monitors. The Am29030 and Am29035 (and all future 29K family members) include support for JTAG, including the ability to examine and modify all the on-chip and off-chip resources. This enables a new breed of very low cost "emulators" that can be used to download, control execution of, and examine the state of programs being debugged. All via a 5 wire interface that doesn't even require removing the chip from the board. Coupled with a logic analyzer, these JTAG-based emulators provide many (but not all) of the features of full-up in-circuit emulation without a huge investment in 29K specific debug tools.

SUMMARY

The 29K family is as easy to program for as any other processor of any sort. Its primary language is C. For porting existing code from other processors, the C compiler includes many special features to ease that task. Tuning the code to take advantage of the unique characteristics of the 29K family doesn't require learning assembly language. If assembly language is what you want, then you can even include assembly code in your C programs. Debug tools are available from a variety of vendors providing a broad array of cost, feature, and performance options.
As mentioned earlier, up to 128 of the 192 general purpose registers can be used as a stack cache. The design of the calling convention requires that at least 34 registers be used as a stack cache. Using the maximum number will result in the best possible execution of the application. Occasionally, the system software architect may decide that some registers would be better used to reduce interrupt latency by reserving them for interrupt contexts. The compiler must be told how many registers are available for its use as local variables. The -lregs compiler option provides a means for the user to specify how many stack cache registers are available to the compiler.

Signals

Most embedded systems deal with interrupts on a regular part of what they do. The Am29000's RISC architecture provides many different ways to deal with interrupts. One of them is to turn the interrupt into a signal to a foreground task. Our C run-time library and sample systems code includes all the code necessary to register signal handlers and raise signals from interrupt context. We can support signal functions that never return (they might do a longjmp()) and signal functions that always return.

OS access

While many embedded applications don't require the services of a file system, some do and almost all would benefit from having one available during debugging. Our C run-time library and sample systems code includes all the code necessary to register signal handlers and raise signals from interrupt context. We can support signal functions that never return (they might do a longjmp()) and signal functions that always return.

ANSI library

The library shipped with our C compiler is fully ANSI conformant except for the system() function. Source code for those parts written in assembler is provided. The library includes optimized string and memory functions, plus high-performance floating point functions for both single and double precision. The math libraries are different for the Am29050 and other 29K family processors that don't have on-chip floating point.

DEBUGGING TOOLS

The focus of this paper has been getting programs to an executable state and then optimizing the performance of that code by using 29K family special features. But getting code to compile and link for a new embedded system is only part of the battle. Debugging and performance tuning is a frequent problem when using a high-end embedded processor. The 29K family has the broadest support for tools to help in these phases of project development. In-circuit emulators have proven themselves to be more than worth the price in projects where time-to-market is a consideration. At least one emulator provides complete, non-statistical execution (as opposed to fetch) histograms. Two provide code coverage capability. All emulators work with one or more source level debuggers. Many attach to a network directly. There are companies supplying logic analyzer adapters and resident monitors. The Am29030 and Am29035 (and all future 29K family members) include support for JTAG, including the ability to examine and modify all the on-chip and off-chip resources. This enables a new breed of very low cost "emulators" that can be used to download, control execution of, and examine the state of programs being debugged. All via a 5 wire interface that doesn't even require removing the chip from the board. Coupled with a logic analyzer, these JTAG-based emulators provide many (but not all) of the features of full-up in-circuit emulation without a huge investment in 29K specific debug tools.

SUMMARY

The 29K family is as easy to program for as any other processor of any sort. It's primary language is C. For porting existing code from other processors, the C compiler includes many special features to ease that task. Tuning the code to take advantage of the unique characteristics of the 29K family doesn't require learning assembly language. If assembly language is what you want, then you can even include assembly code in your C programs. Debug tools are available from a variety of vendors providing a broad array of cost, feature, and performance options.