Embedded RISC Processor Selection February 1993 Benchmark

Advanced Micro Devices

by Daniel Mann

When selecting a processor for a new embedded product design, there are a range of factors to be taken into consideration. A RISC processor is typically selected when performance is of concern. For this reason processors are often compared with each other when performing synthetic benchmarks.

However, such benchmark comparisons are not sufficient for resolving the processor selection problem. Certainly compiler optimization technology is important, and most of the major RISC processors have available highly optimized compilers, but other factors can weigh more heavily when dealing with embedded product development.

This brief note compares aspects of the underlying 29K™ Family processor architecture relevant to real-time systems; such as operating system support and interrupt response time. Additionally, support tools such as embedded debuggers and monitors which can greatly effect product development times are described.

The topics discussed are more valuable to engineers than the widely touted synthetic benchmark results and enable an engineer to better judge the performance of a processor in a real-time application.

INTERRUPT RESPONSE TIME

The Am29000™ processor is free of microcode which can greatly influence a systems interrupt architecture. Typically, when an interrupt occurs on a CISC type processor, a context frame is saved on a memory stack. A 29K Family user is not constrained by built-in microcode and is free to construct a scheme which reduces overheads and better suits real-time system requirements.

Table 1 shows interrupt response times. The 29K Family of processors can commence a *Freeze Mode* interrupt handler surprisingly quickly. Very few processor cycles elapse before the processor is prepared to commence interrupt processing.

Table 1. Interrupt Response Time

CPU	Am29000 µp	80960CA
Speed	33 MHz	33 MHz
Minimum Latency	0.21 µs	0.9 µs
Worst-case Latency	0.48 µs♦	4.02 µs

♦ 3-cycle memory system assumed

TASK CONTEXT SWITCH

A task context switch occurs when the operating system decides that the currently running task is no longer the highest priority task. When this occurs the processor registers reflecting the current-task context are updated to reflect the in-coming task context. The out-going task context is saved to memory.

Context switching is an operating system overhead which can burden real-time system performance. Processors such as the MC68020 should be able to perform a relatively fast context switch due to their small number of registers.

However, using techniques such as only restoring single activation records and burst-mode memories, the 29K Family is able to perform fast context switching. (Activation record refers to the group of registers allocated to an individual procedure). Table 2 presents synchronous context switch times. Synchronous context switching is what happens in a preemtive operating system (OS) when one task makes an operating system call and the OS decides to let another task run instead. The times shown are for the JMI C EXECUTIVE real-time operating system.

Table 2. Synchronous Context Switch Times

CPU	Am29000 µp	80960CA	MC68020
Speed	33 MHz	33 MHz	25 MHz
Time	6 µs	7 µs	17 µs

Publication# **17819** Rev. **A** Amendment **/0**
Issue Date: **March 1993**

SYSTEM CALL SUPPORT

System calls are trusted library functions that execute with Supervisor mode permissions to access resources not normally available to the user. System calls can be frequently used by real-time systems.

The 29K Family register stack and access protection support enable a reduction in the system call overhead. Table 3 presents a sample of system call times for the JMI C EXECUTIVE real-time operating system.

Table 3. System Call Times

CPU	Am29000 µp	80960CA	MC68020
Speed	33 MHz	33 MHz	25 MHz
write()	4.5 µs	5 µs	29 µs
write to queue	12 µs	19 µs	91 µs
read from queue	12 µs	19 µs	87 µs

PROCEDURE CALLS

The instruction set of a RISC processor is optimized to support compiler generated code for a high level language such as C. The efficiency with which a processor supports procedure calls and returns is crucial in determining overall C language performance.

The C function shown below produces the Fibonacci number sequence. Though *fib()* cannot be considered representative of a typical procedure, it does highlight a number of issues. The function is recursive, and therefore extensively tests the efficiency of the procedure call and return mechanism. Additionally, the method by which processor registers are allocated for procedure use is revealed.

```
int fib(n)
int n;
{ int    result;
  if (n <= 2) return 1;
  result = fib(n–1) + fib(n–2);
  return result;
}
```

Table 4 shows execution times for the *fib()* function with five levels of nested procedure calls.

Table 4. Procedure Call-Stack Overhead

CPU	Am29000µp	80960CA
Speed	33 MHz	33 MHz
fib(1)	0.30 µs	0.61 µs
fib(5)	3.21 µs	6.67 µs

The 80960CA allocates fixed groups of registers (windows) for each new procedure call's use. The Am29000 processor allocates dynamically sized groups of registers (activation records). Both processors have a limited amount of on-chip cache (or registers) for fast access to procedure data. The efficient allocation of the cache resource is important in maintaining performance with deeply nested procedures.

Table 5 indicates the number of words consumed by *fib(1)* and *fib(5)* procedure calls. By allowing the register allocation to be determined at compile-time, rather than having hardwired fixed window sizes, the need to flush and restore registers to external memory is reduced.

Table 5. Procedure Call-Stack Usage

CPU	Am29000 µp	80960CA
fib(1)	4 words	20 words
fib(5)	16 words	100 words

BURST-MODE MEMORY ACCESS

An important activity for many real-time systems is moving blocks of data within memory. The C library routine memcpy(dest, src, length) can be used to perform this operation. In practice, data movement is frequently required by assembly-level interrupt handlers that cannot make use of support library routines. However, the memcpy() routine serves the purpose of demonstrating the underlying processor's data moving characteristics.

The code sequence below was extracted from the memcpy() code for the 80960CA processor. A basic loop performs a load-store sequence; each trip round the loop transfers four bytes of data.

```
L29:
  ld     (g1),g4     #read source
  st     g4,(g0)     #write
  addo   g0,4,g0     #dest. pointer
  lda    4(g1),g1    #src. pointer
  subo   4,g2,g2     #length count
  cmpobl 3,g2,L29    #test for end
```

The following code is the basic loop for memcpy()when executing on an Am29000 processor. The processor makes use of *burst-mode* to transfer blocks of 16 words (each word = 4 bytes). Each trip round the loop transfers 64 bytes of data.

```
  srl    lr4,lr4,2   ;block of 16
  sub    lr4,lr4,2
L6:
  mtsrim cr,15       ;read 16
  loadm  0,0,gr96,lr3 ; words
  mtsrim cr,15       ;write 16
```

```
   storem 0,0,gr96,lr2 ; words
   add    lr2,lr2,16*4 ;advance pointer
   jmpfdec lr4,L6      ;test for end
    add   lr3,lr3,16*4 ;advance pointer
```

Burst-mode is used with load- and store-multiple instructions (LOADM and STOREM above). These instructions differ from regular load and store instructions in that consecutively addressed data is transferred between processor registers and off-chip memory.

In the memcpy() example, the LOADM instruction is used to read 16 words of data into registers. When the memory system is operating in burst-mode, address values are not supplied for each memory access. New address values are generated when a burst-mode access commences. Once initiated, the memory system supplies consecutive data values until the the burst is complete.

The 80960CA instruction loop may be shorter (six instructions) compared to the Am29000 loop (seven instructions), but the 80960CA transfers four bytes per loop iteration compared to the Am29000 processor's 64 bytes. Certainly the LOADM and STOREM instructions are multicycle but their efficiency when accessing data blocks is much higher than the back-to-back load and store operations used by the 80960CA code.

MICROCONTROLLERS INCORPORATING SYSTEM INTERFACE LOGIC

The 29K Family includes three-bus Harvard processors, two-bus processors with simplified memory system interfaces, and microcontrollers. Many of the family members are pin compatible, and User mode code is binary compatible throughout the family.

The microcontroller members of the family, such as the Am29200™ microcontroller, include DRAM, ROM, I/O and other peripheral interface logic on-chip. (See Figure 1) Memory devices can be connected directly to the chip; without the need for any external glue-circuitry.

Microcontroller devices such as the Am29200 microcontroller make the cost of designing with RISC very low.

UNIVERSAL DEBUGGER INTERFACE (UDI)

Code development for embedded processors is generally more costly than development of code of equivalent complexity intended for execution on an engineering workstation. The availability of debug tools and their configurability is an important factor when selecting a processor for an embedded project.

Developers of products containing embedded processors are looking to RISC for future products offering increased capability. The greater performance relative to RISC processor cost should make this possible. The suitability, cost, and productivity of the tools available for code development are likely to be the major factor in deciding the direction ahead when preparing to tool-up for RISC.

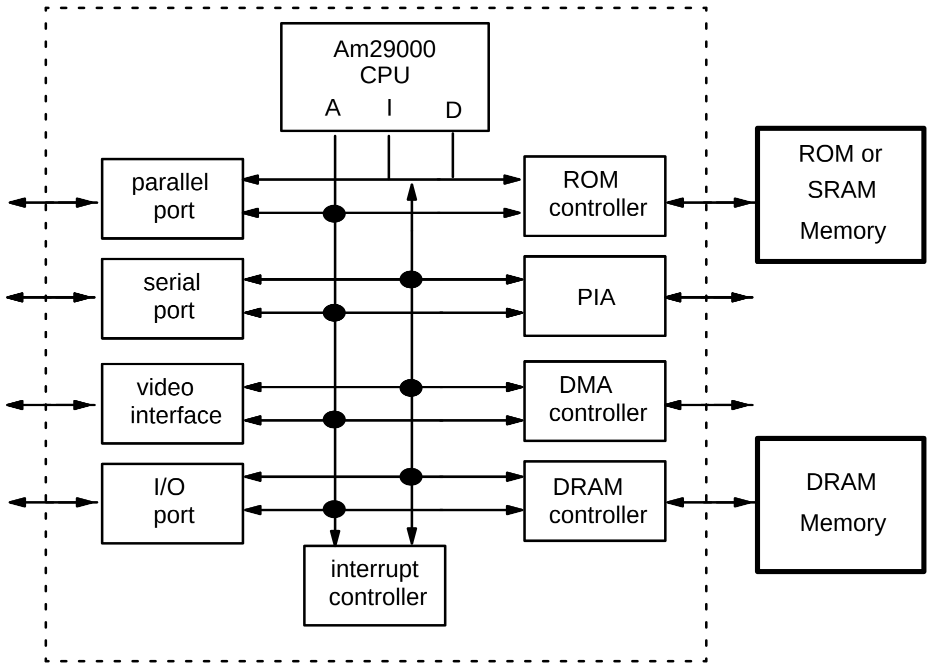

Figure 1. Am29200 Microcontroller Block Diagram

To meet these needs, AMD has developed a Universal Debugger Interface (UDI) standard. Tools which comply with the standard can be freely intermixed. This offers a greater selection of tool configurations. Additionally, it simplifies the debug tool development process, which enables the tool manufacture to bring their product to market sooner.

UDI divides the debugger task into two processes; the Debugger Front End (DFE) and the Target Interface Process (TIP). These two processes communicate via an Inter-Process Communication (IPC) mechanism which complies with the UDI protocol. On UNIX platforms, sockets are used for IPC communication. Figure 2 presents the UDI approach.

EMBEDDED DEBUG MONITOR AND SUPPORT OPERATING SYSTEM

The MiniMON29K™ debug monitor assists in developing software for a 29K Family processor. The monitor is not standalone. It requires the assistance of a support module known as the TIP. The TIP executes on a separate processor and *connects* the monitor to the UDI *backplane.*

The monitor and the TIP typically are connected via an RS232 connection or other hardware specific communication path. In the MiniMON29K monitor, messages flow between the monitor debug core and the TIP and enable the target processor operation to be observed and controlled.

The MiniMON29K monitor is supplied along with a simple operating system known as OS-boot, see the 29K Family target software modules in Figure 3. AMD supplies these modules in source form.

The OS-boot code performs processor initialization and configuration. It also supports a run-time system-call interface called Host Interface (HIF). The 29K Family library routines typically make HIF service requests.

When developing software for a new 29K Family hardware system, it is normally the case that the OS-boot and the MiniMON29K monitor message system are first retargeted to the new hardware. The modular construction of the MiniMON29K monitor and OS-boot simplifies the task of getting application code running on new hardware.

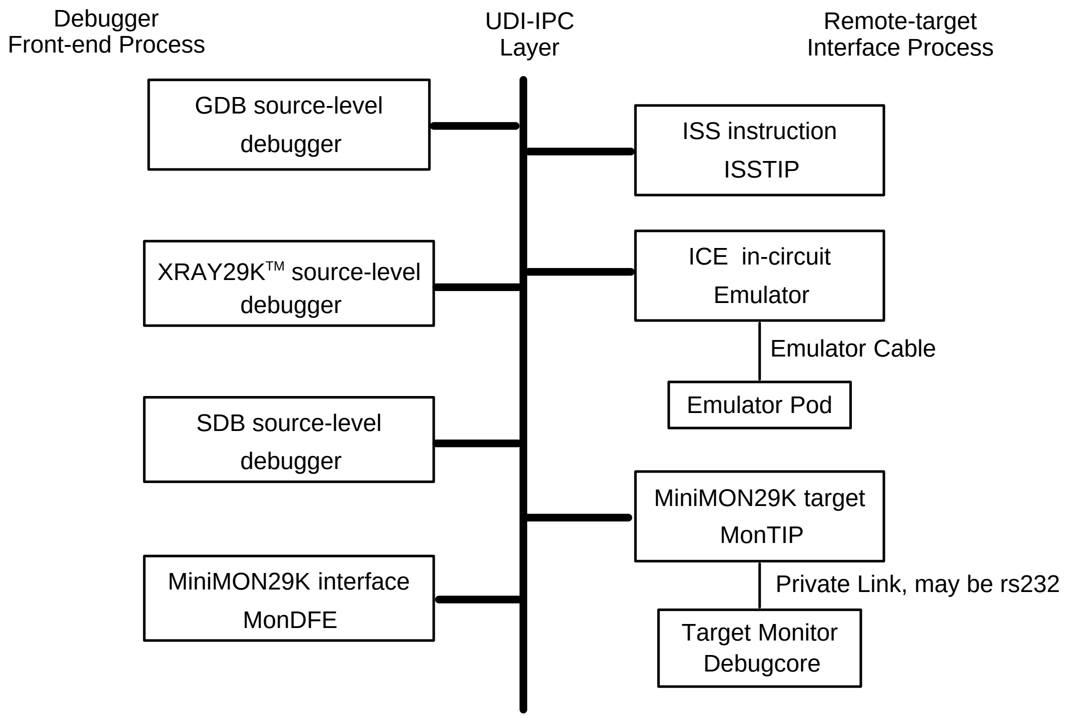

Figure 2. Available Debugging Tools that Conform to UDI Specification

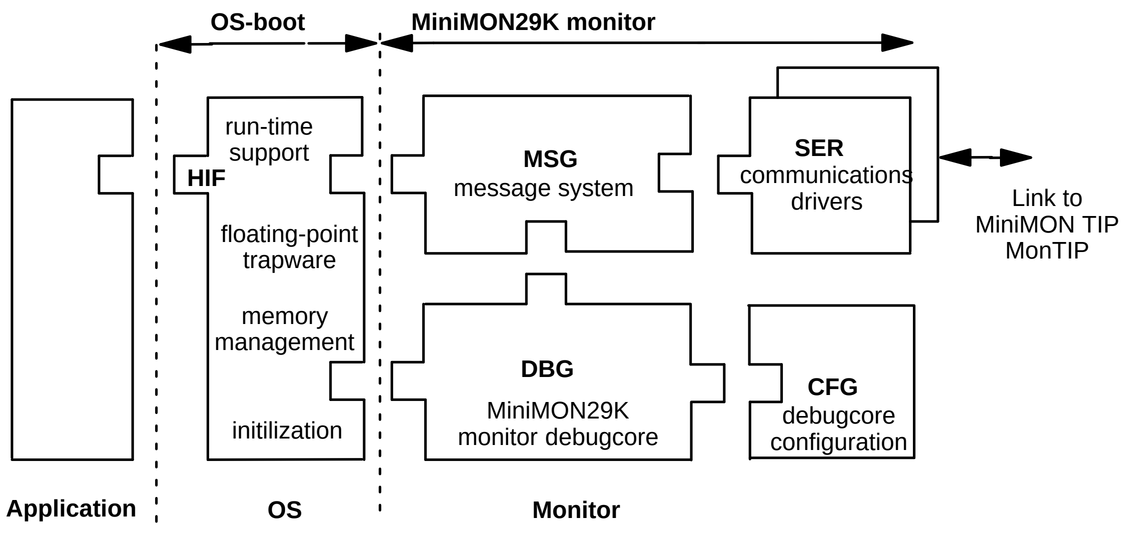

Figure 3. 29K Family Target Software Module Configuration

C++ SOFTWARE DEVELOPMENT

C++ is starting to be more frequently selected for use with medium and large embedded projects. There are no clear criteria or *benchmarks* established for selecting a processor when C++ is the desired programming language.

The 29K Family is particularly suited to C++ code execution for a number of reasons:

- C++ makes frequent calls to object member-functions. This is likely to place heavy demands on the procedure call mechanism of the processor. The dynamically sized register-blocks (activation records) allocated to each procedure by a 29K Family processor will prove efficient at keeping up with the demands of C++ for procedure registers allocation.
- Additionally, the Metaware High C® 29K compiler is able to pass objects in register in preference to memory when supplying object parameters to member functions. Possibly more importantly, the 29K Family High Level Language Calling Convention allows for objects to be returned in registers by procedures. This avoids the need to pass objects via memory which is less efficiently accessed compared to on-chip registers.
- Inline functions are also frequently used by C++ programmers. The large number of registers available to a 29K Family processor enable optimized inline code to make efficient use of registers
- During the introduction of C++ compilers, debuggers have often lagged behind in their ability to offer an effective debug environment. The 29K Family tool-chain is supported with UDI conforming debuggers for both C and C++ code development.