Debug Support on the ColdFire Architecture

William Hohl, Joe Circello, Klaus Riedel

Motorola, Inc.
High Performance Embedded Systems
6501 William Cannon Drive West
Austin, Texas 78735

Motorola, Inc.
Phoenix Design Center
432 North 44th St., Suite 200
Phoenix, Arizona 85008

ABSTRACT

Debug support on the ColdFire architecture is designed to be well-suited to embedded systems, and provides both real-time and background debugging techniques. A dedicated port for real-time information and a serial port allow users to read and write to memory, address and data registers, set up complex, multi-level breakpoints and trace execution paths of instructions. The debug module also includes new features such as concurrent debug and core operations and programmable real-time trace support visible on a parallel output port.

1. INTRODUCTION

There are a number of methods in use today among embedded microprocessors for system emulation and debug support, such as OnCE™ and COP™. The desire to examine register contents, memory and on-chip peripherals in a non-intrusive fashion has provided an impetus for developing a debug architecture that can access all of these system resources but that does not tie up any user resources, such as a bus or the system configuration registers. The difficulties in examining resources in an embedded environment stem from the inaccessibility of the data and/or address bus. System configurations such as the one shown in Fig. 1, where processor status bits and all data and address busses are available, allow logic analyzers and emulators to gain sufficient visibility to the internal operations of the core and do not require any additional debugging mechanism.

As is the case with many embedded cores, however, the processor might be connected to a number of internal devices, such as DMAs, A/D converters or parallel ports, and the bus may not be visible to the external debugger. As the block diagram in Fig. 2 shows, custom logic may be present with internal memory, which may make emulation difficult because of proprietary design considerations. This scheme would require an invasive technique in order to determine register and memory contents.

In the development of a new debug unit for the ColdFire family of embedded processors, real-time applications were considered heavily in the design. Limitations from static debug monitors arise from having to stop the application from executing to read internal kernel resources or change the state of a task [1]. Engine controllers, disk drive controllers and other time-critical applications may not be able to withstand the suspension of the operating system because of physical limitations, e.g., a disk drive head being damaged. Although systems may not be able to completely halt to evaluate register conditions or modify memory, some may be able to tolerate small intrusions in their instruction streams, small enough to allow an interrupt routine to save register values and other essential variables to memory. Additionally, being able to trace the dynamic execution path of the instruction stream provides developers with the resources necessary to monitor and pinpoint any programming problems.

The debug architecture of the ColdFire processors attempts to address these issues of real-time debug functionality across a wide range of possible microprocessor implementations through the use of separate I/O connections. In this manner, a consistent interface between the external development system and the ColdFire microprocessor is maintained, regardless of the type, protocol or presence of an external bus. Real-time operating systems benefit from the introduction of a parallel output port for monitoring the dynamic execution path of instructions. In addition to providing support for real-time debug functions, the ColdFire debug unit is designed to provide a migration path for current emulator packages. By supporting a proper subset of the 683xx family background debug mode (BDM) instructions, emulators can reuse their existing interface.

In keeping with the design methodology of the ColdFire family [2], the debug module is a fully synchronous, edge-triggered design. Since the hardware is completely synthesized, the scalable design can be targeted toward higher frequencies with process improvements. The logic associated with debugging operations occupies approximately twenty-five percent of the core's die area.

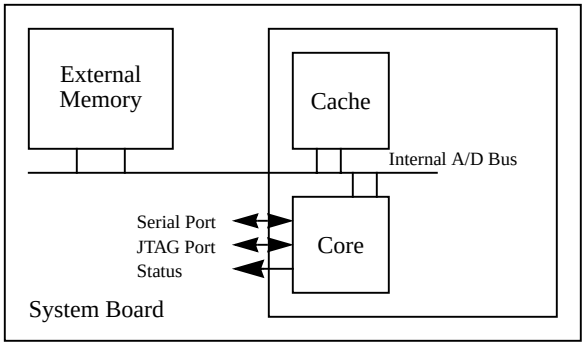

Fig. 1 Board configuration with bus visibility

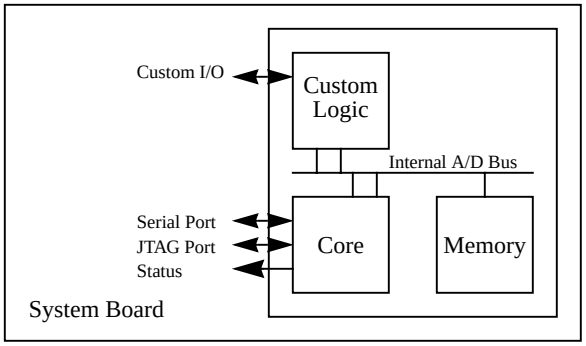

Fig. 2 Board configuration without bus visibility

2. ARCHITECTURE

The emergence of the PowerPC architecture for desktop and high-performance embedded applications has created an opportunity for the 68000 family to refocus entirely on cost-sensitive, deeply embedded systems. The ColdFire architecture is the result of this development and represents a new approach targeted specifically for the emerging class of advanced consumer electronics applications.

Within the domain of cost-driven embedded systems, there are several salient requirements. First, the processor core must be small to permit cost-effective integration of on-chip memories and other system modules and peripherals. Second, a high-density instruction set can minimize memory requirements. In many designs, the cost of the memory system exceeds the microprocessor cost, so this factor can significantly impact overall system cost. The ColdFire processor architecture addresses these requirements through a variable-length instruction set to maximize code density implemented in a RISC-based approach to provide a very efficient silicon design.

The ColdFire architecture uses a synthesis-driven, tools-based design approach. This methodology allows for the simple addition of optional hardware modules to provide custom functions, and provides design independence

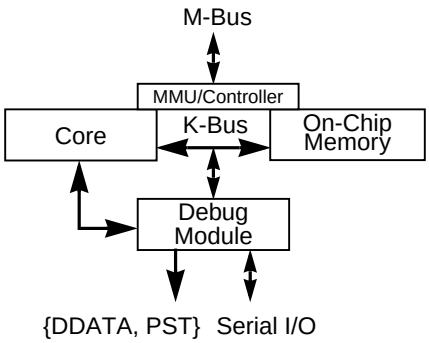

Fig. 3 Processor/Memory complex with Debug Module

across different process technologies that target a range of operating frequencies and voltages.

The ColdFire processor core is implemented with two, independent and decoupled pipelines: a 2-stage instruction fetch pipeline for prefetching instructions and a 2-stage operand execution pipeline to perform the actual instruction execution. A 12-byte instruction buffer serves as the FIFO queue between the two pipelines and provides the decoupling mechanism. The operand execution pipeline is based on the traditional RISC compute engine structure with a dual read-ported register file feeding the ALU. Register-to-register instructions are executed in a single pipeline cycle utilizing hardwired control of the processor's resources. For instructions reading operands from memory, each stage of the operand execution pipeline is used twice: first, for operand address generation and second, for the actual execution of the instruction. The resulting execution of these “embedded-load” instructions provides good performance while minimizing the processor's implementation size.

Thus, the ColdFire processor architecture provides near-68040 levels of performance at a given frequency in a core size smaller than the original 68000 design [2].

The ColdFire design provides a series of hierarchical internal buses, each providing the appropriate bandwidth based on location within the architecture. The position of the debug unit in relation to the core is shown in Fig. 3. The debug unit connects to the K-Bus, a high-speed, single-cycle bus connecting the processor core to internal memories (e.g., cache, RAM, etc.). Other peripheral devices, like a DMA controller, timer or serial communication interface, are connected to the processor through the M-Bus (master bus).

A simplified block diagram of the debug unit is shown in Fig. 4. The ColdFire processor provides real time trace functionality through a parallel output port that delivers encoded

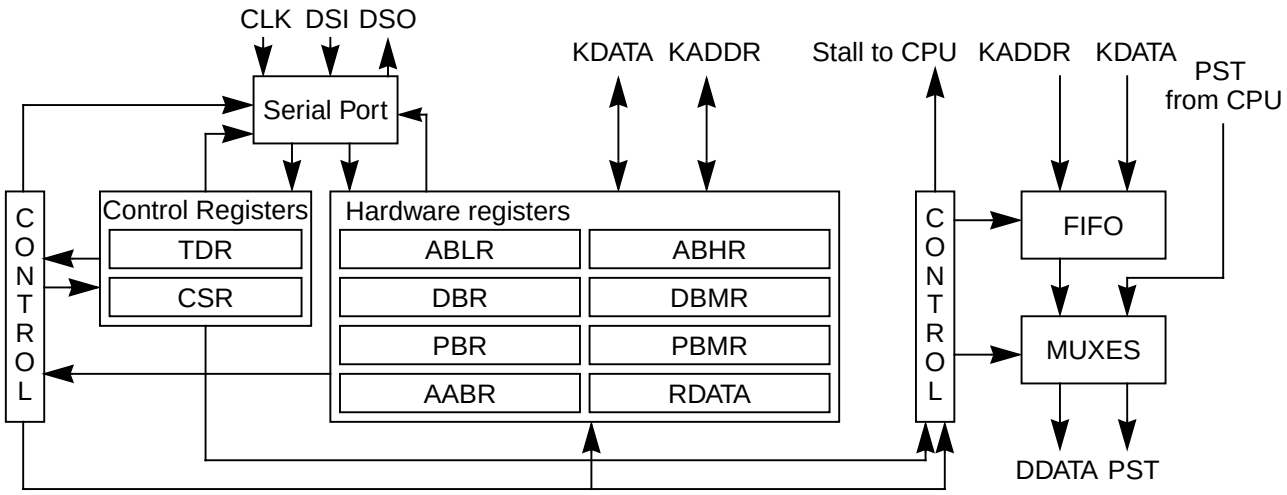

Fig.4 ColdFire debug module block diagram, showing real-time and background debug datapaths.

processor status and data. This port is partitioned into two 4-bit sections: one nibble displays the execution status of the core (PST), while the other displays operand data (DDATA). The nibble-wide DDATA port is connected to the K-Bus by a two entry FIFO buffer. The FIFO buffer consists of two 32-bit storage registers that capture K-Bus information - target addresses and operand data - and displays it on the DDATA port continuously. The execution speed of the processor is affected only when both storage elements have valid data waiting to be dumped onto the DDATA port. When this occurs, the processor is stalled until one storage element becomes available.

The remaining datapath is for background debug and real-time debug operations. It contains two control registers: the configuration status register (CSR) and the trigger definition register (TDR). The CSR defines the operating configuration for the processor and memory subsystem, as well as reflecting the status of the breakpoint logic. The TDR configures the breakpoint logic and defines the type of action taken in response to hardware breakpoints. The available breakpoint and comparison registers are the upper and lower address breakpoint registers (ABHR, ABLR), the attributes register (AABR) which contains a mask in the upper eight bits, the data breakpoint register (DBR) and its mask (DBRM), and the program counter breakpoint register (PBR) and its mask (PBRM). One additional register (RDATA) resides in the debug unit to hold incoming data from the data bus during certain instructions like reads and core-initiated writes to the debug module.

In order to minimize transistor count as much as possible, a number of debug resources are shared between real-time support and background debug mode. While in BDM, the breakpoint registers are used to hold address and operand information to be driven on the bus during reads and writes. These same registers also hold breakpoint configurations during real-time debug operations. The use and operation of these registers is described in more detail in sections 4 and 5.

Enhanced Instruction Set. Several new instructions were added to the existing 68000 family instruction set [3] to accommodate the debug unit: HALT, PULSE, WDDATA and WDEBUG. The core can execute the HALT instruction to suspend processor activity and allow for background debugging. Before halting operation, all previous instructions and bus cycles are completed in the core. The processor then resumes execution once it receives a GO command from the debug unit.

The PULSE command creates a unique processor status, which is useful for generating a trigger to external logic during debug or performance characterization.

To capture data and display it on the DDATA port, the WDDATA instruction fetches the operand defined in the effective address, then places the appropriate number of nibbles on the DDATA output pins, independent of any debug configuration.

Finally, the core can load all internal debug registers with the WDEBUG instruction. This instruction forces memory reads on the bus which are monitored by the debug unit. The two consecutive longwords which are fetched contain a 48-bit debug command: the first 16-bits are the debug opcode, and the last 32-bits are the immediate operand to be loaded into the destination debug control register (DRc). To make the debug unit aware that these operands contain debug reg-

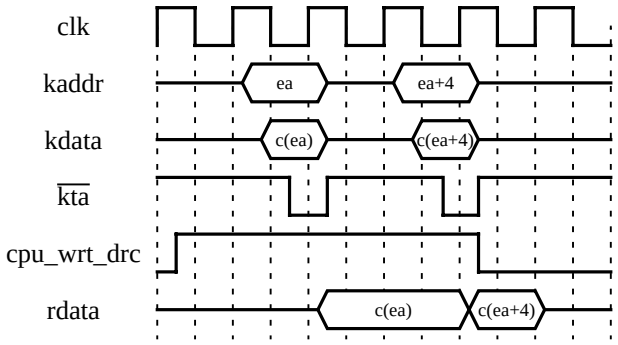

Fig. 5 Core-initiated writes to debug registers

ister information, the core asserts *cpu_wrt_drc* in the debug module. Once the bus access receives a transfer acknowledge ($\overline{\text{kta}}$), the instruction is loaded into the RDATA register, then sent to the serial block. The debug unit then interprets this instruction as if it were a background debug command, routing the first half of the data into the appropriate debug register. The second half of the data follows in the second longword that is fetched. The process is illustrated in Fig. 5. Since there are no hardware interlocks to prevent the core and the debug unit from accessing the registers at the same time, it is left to the user to ensure that simultaneous accesses do not occur.

3. REAL-TIME TRACE SUPPORT

When connected to an external development system, the PST/DDATA port provides a useful debug utility in tracking the dynamic execution path of instructions. Since most branch instructions in the ColdFire instruction set specify PC-relative or absolute addressing, the PST indication of a "taken branch" can be used with an external program image to track the dynamic path. However, other change-of-flow operations cannot be tracked using only this approach. These difficult cases use some form of variant addressing, where a program-visible register or memory location is used in the calculation of the target instruction address. Examples of this type of opcode include the JMP instruction through a "jump table", subroutine return, and all exceptions. Change-of-flow operations using variant addressing can easily be followed with the unique synchronization between the PST lines displaying a taken branch, and the DDATA pins displaying the target address. Two types of data can be optionally traced and identified: branch target addresses and operand data. Control bits in the CSR define the number of bytes of target address to display, as well as which type of operand data to capture (read, write, or no data at all). Tables 1 and 2 show the possible configurations.

Tracing the flow of captured target addresses is done in the following manner. The PST lines identify that a taken branch was executed with an encoding of $5. The lines then

Table 1: CSR Configuration for Operand Data Capture

CSR[12:11]	Data Captured and displayed on DDATA port
00	no operand data
01	all M-Bus write data
10	all M-Bus read data
11	all M-Bus read and write data

Table 2: CSR Configuration for Target Address Capture

CSR[9:8]	Number of branch target address bytes to be displayed on DDATA port
00	no branch target address
01	lower 2 bytes of the target address
10	lower 3 bytes of the target address
11	entire 4 byte target address

display a marker, signaling that the target address is to be displayed on the DDATA pins. This marker also identifies the number of bytes that will be displayed. Marker values of $8, $9, $A, $B are used to quantify the contents of the DDATA port. Possible PST markers and their definitions are given in Table 3. On the subsequent cycle, the target address is available, and it is then displayed on the DDATA pins, least significant nibble to most significant nibble. The DDATA port can be configured to display the lower 16 bits, 24 bits or the entire 32 bits of the target address. Depending on the state of the FIFO, the marker may or may not be displayed on the cycle immediately following the taken branch encoding on the PST port (PST = $5). Tracing the flow of captured operand data is identical to the situation for target addresses, except the marker is displayed after the PST value signalling that the given instruction has started execution.

Fig. 6 shows the execution of an indirect JMP instruction with the lower 16 bits of the target address being displayed on the DDATA output. In this diagram, the indirect JMP branches to address "target". Note the processor internally forms the PST marker ($9) one cycle before the address begins to appear on the DDATA port, which is then displayed for four consecutive clocks, starting with the least-significant nibble. The processor continues execution, unaffected by the DDATA bus activity.

One additional concept which is very important to the real-time trace functionality is the notion of strict synchroniza-

tion between the executed instruction stream, as defined by the PST signals, and the availability of the captured data on the DDATA signals. There is always a strict ordering such that DDATA can be easily associated with the appropriate instruction. A sample of code showing the PST/DDATA output is given in Fig. 7, where the CSR is configured to display two bytes of branch target address, all operands generate M-bus accesses, and both read and write operands are captured. Although the figure shows no overlap between non-zero PST and DDATA values, in reality, overlap does exist.

4. REAL-TIME DEBUG FUNCTIONALITY

Real-time debug is applicable to systems that cannot be completely halted to evaluate register or memory conditions, but that can tolerate a small intrusion into real-time operations. For these cases, the ColdFire architecture provides real-time debug capability by allowing the user to program a wide range of breakpoint configurations and the type of response that the debug unit provides. These options can be configured by loading the various control registers through either the serial port or the use of WDEBUG instructions directly from the core.

The ColdFire debug unit includes hardware breakpoint registers to define single and double-level triggers, with a programmable trigger response. Essentially, the breakpoint mechanism operates in the following manner:

- Single-level trigger
  if (condition == breakpoint register)
  then Trigger
- Double-level trigger
  if (condition1 == breakpoint register1)
  then if (condition2 == breakpoint register2)
  then Trigger

Both the single- and double-level triggers can be programmed using a combination of address, data and program counter values.

Table 3: PST Marker Definitions for DDATA Transfers

PST[3:0]	Marker Definition
1000	Begin 1-byte transfer on DDATA port
1001	Begin 2-byte transfer on DDATA port
1010	Begin 3-byte transfer on DDATA port
1011	Begin 4-byte transfer on DDATA port

Address values can be assigned to trigger on exactly one value, within a range, or outside of a range, where the high and low values are stored in ABHR and ABLR. Attributes such as transfer type and operand size may be factored into the comparison.

Data breakpoints can be assigned to trigger on any one of the four bytes within a long word, on a particular word in a long word, or the entire 32-bit value. There is an additional bit that allows the logical sense of the data breakpoint comparators to be inverted. A trigger based on the occurrence of a data value not being equal to the programmed value is also possible. To completely define the data breakpoint, a 32-bit mask register is combined with the DBR value, where each bit of the mask register affects the corresponding bit of the breakpoint register. If a bit is set in the mask, then the corresponding bit in the comparison is ignored. The data breakpoint can be used for both aligned and misaligned operand references.

Program counter breakpoints are allowed for regions outside of a programmed value or for the value itself, and a 32-bit mask register is used in conjunction with the breakpoint register. Since the actual program counter lies within the

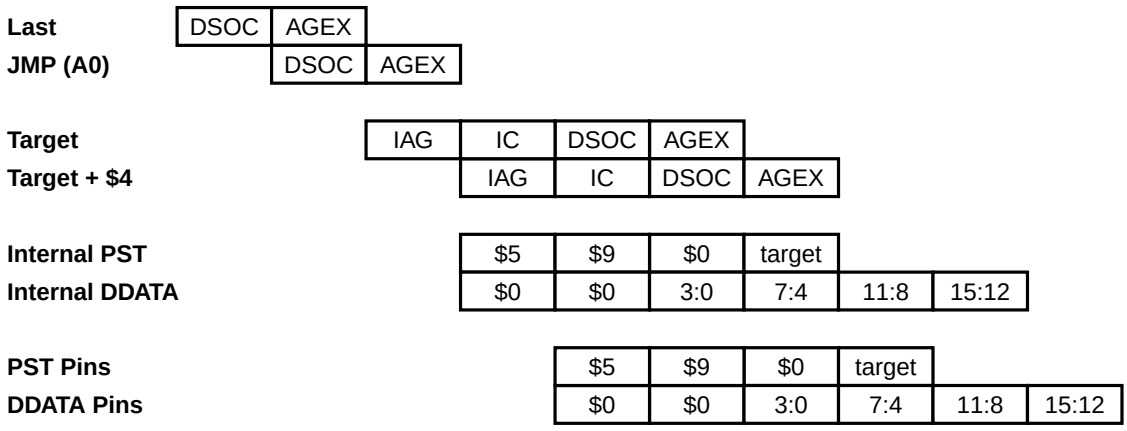

Fig. 6 Pipeline timing example with DDATA output

Inst Address	Instruction	
00001316	movq	#1, d0
00001318	mov.b	d0, (-4,a6)
0000131c	pea	(-68,a6)
00001320	lea	Func2, a0
00001326	jsr	(a0)
0000115c	mov.l	d7, -(a7)

PST	DDATA	Description
$1	--	Begin inst @ 1316
$1	--	Begin inst @ 1318
$8	--	Write operand for inst @ 1318
0	[3: 0]	Bits from write operand for inst @ 1318
0	[7: 4]	Bits from write operand for inst @ 1318
$1	--	Begin inst @ 131c
$b	--	Write operand for inst @ 131c
0	[3: 0]	Bits from write operand for inst @ 131c
0	[7: 4]	Bits from write operand for inst @ 131c
0	[11: 8]	Bits from write operand for inst @ 131c
0	[15:12]	Bits from write operand for inst @ 131c
0	[19:16]	Bits from write operand for inst @ 131c
0	[23:20]	Bits from write operand for inst @ 131c
0	[27:24]	Bits from write operand for inst @ 131c
0	[31:28]	Bits from write operand for inst @ 131c
$1	--	Begin inst @ 1320
$5	--	Begin taken-branch inst @ 1326
$9	--	Target address for inst @ 1326
0	$c	Bits from target address (115c)
0	$5	Bits from target address (115c)
0	$1	Bits from target address (115c)
0	$1	Bits from target address (115c)
$b	--	Write operand for inst @ 1326
0	$8	Bits from return address for inst @ 1326
0	$2	Bits from return address for inst @ 1326
0	$3	Bits from return address for inst @ 1326
0	$1	Bits from return address for inst @ 1326
0	$0	Bits from return address for inst @ 1326
0	$0	Bits from return address for inst @ 1326
0	$0	Bits from return address for inst @ 1326
0	$0	Bits from return address for inst @ 1326
$1	--	Begin inst @ 115c

Fig. 7 Sample code showing PST/DDATA synchronization

ColdFire core, the core does the comparison of the breakpoint register with the program counter of the instruction in the Decode & Select/Operand Cycle (DSOC) stage of the Operand Execution Pipeline [2]. After factoring in the mask register, if a breakpoint occurs, the processor immediately suspends execution until the defined trigger response can take place. The trigger always occurs before the given instruction is executed to allow a precise interrupt for these program counter breakpoints. This contrasts with the address and/or data breakpoint triggers, which are imprecise.

For these triggers, the processor core may have executed several additional instructions before the trigger is recognized. This requirement was strongly encouraged by third-party emulator vendors.

Table 4: Breakpoint Status Field Definition

CSR[31:28]	Breakpoint Status
0000	no breakpoints enabled
0001	waiting for level 1 breakpoint
0010	level 1 breakpoint triggered
0101	waiting for level 2 breakpoint
0110	level 2 breakpoint triggered

All of the breakpoints can be combined to create complex or filtered triggers [4] that can greatly simplify real-time debugging. As an example of a double-level trigger, a first-level breakpoint can be set if a user-mode operand word write of any value other than $1234 occurs to address $3000. A second-level breakpoint and trigger fires if this is followed by the execution of the program counter equal to $8030. The current state of the breakpoints can be read at any time from a four-bit field in the CSR register. Table 4 shows all possible values for the field which provides read-only status information.

The trigger definition register contains two fields for configuring and enabling the two levels of breakpoints and one field for defining the debug unit's response once the breakpoint has been encountered. Three types of trigger responses are provided, and by their very nature, should interfere as little as possible with normal operations. The first type of response, and the least intrusive, is to show the trigger on the DDATA output port and cause no internal response. When the bus is not busy displaying captured data, the DDATA lines will always show a trigger even if other options are chosen.

The trigger can also generate a debug interrupt to the processor. This interrupt is considered higher in priority than all other interrupts and causes the processor to vector to a unique exception handler while executing in emulator mode. As an example, this handler may save the state of all of the program-visible registers, as well as the current context, into a reserved memory area. Once the processor exits this debug interrupt service routine, the memory area can be read using the serial port and BDM commands. While executing in emulator mode, the processor ignores all I/O interrupt requests and all memory references may be mapped into a special "alternate" space under control of a bit in the CSR. By default, these references are simply mapped into

supervisor instruction and data spaces. Emulator mode is exited when a Return from Exception (RTE) instruction is performed. Note that emulator entry and exit are signalled on the PST outputs.

The third and final response to a trigger is to enter background debug mode and halt the processor. The halted condition is also displayed on the PST lines.

To help support real-time debug functions, the ColdFire architecture allows simultaneous operation of the processor core and the debug module. Background debug commands which reference memory must first request the bus from the core in order to perform the access. Included in this concurrent operation is an internal bus arbitration scheme which effectively schedules bus cycles for the debug module by stalling the processor's instruction fetch pipeline and then waiting until all operand requests have been serviced before granting the bus to the debug module. The debug module completes one bus transaction before releasing the bus back to the processor. Registers within the debug module can be loaded without interfering with core operations; however, address and data registers within the core are not accessible while the processor is running.

5. BACKGROUND DEBUG FUNCTIONALITY

The ColdFire processors support most of the BDM functionality found on the 683xx family of parts, plus new instructions to access the debug module. This mode of gathering information is primarily intended for background operations, since the core is halted while the commands are issued and executed.

Background debug mode can be entered by a variety of methods. The first source of entry is the $\overline{\text{BKPT}}$ pin which is sampled by the processor once during the execution of each instruction. If there is a pending halt condition at the sample time, then the processor suspends execution and enters the halted state. The HALT opcode, which is considered a supervisor instruction in the ColdFire instruction set, can also force the processor to suspend execution. If a catastrophic fault-on-fault condition occurs, such as a double bus fault, then background debug mode is automatically entered. The last way to enter BDM is through the use of the hardware breakpoint registers. The hardware breakpoints that were discussed in Section 4 can be configured to generate a pending halt condition in a manner similar to the assertion of the $\overline{\text{BKPT}}$ signal. Regardless of the method used to enter BDM, the fact that the core is halted is reflected on the processor status lines and the method by which BDM was entered is available in the CSR.

Once BDM is entered, commands are issued to the debug unit via a full-duplex serial interface consisting of an input pin (DSI), an output pin (DSO) and a system clock (DSCLK). The external development system serves as the master of the serial communication channel and is responsible for the generation of DSCLK. The operating range of the serial channel is DC to 1/2 the frequency of the processor clock. For ColdFire parts, two modes of clocking are allowed - free-running or pulsed. A free-running DSCLK allows commands to be shifted into the part at any point; the development system, though, must be able to correctly interpret NOT READY responses from the debug unit while it is still processing a command. For the latter clocking scheme, a command is entered, then the DSCLK is disabled for a certain period while the command (e.g., a read or write operation) is performed. Once the clock is enabled again, the response can be shifted out.

For all ColdFire debug operations, the commands consist of a variable number of 17-bit packets of information, one bit for status/control and the remaining 16 bits for the command/address/data. Typically, a read operation will include one packet defining the operation, followed by two packets defining the address. Write operations include one packet defining the operation, two packets for the address and one or two packets containing the operand data. Table 5 lists the available commands. It was decided in the early stages of the design that the entire 683xx command set was not needed. Unlike the original 683xx debug routines that exist as microcode, the new module sits apart from the core. Implementing a CALL function is more difficult, since it is involved in fundamental core operations. Third-party developers also felt that this function was unnecessary. Since the debug module has the exact same addressability as the processor itself, the commands to cause a RESET in any module are available. Our "simple is better" philosophy dictated that the RESET command be eliminated from the instruction set. Two of the original instructions were modified, and two new instructions were added to the set.

Memory reads and writes are supported through the READ, WRITE, DUMP, and FILL commands. Byte, word and longword sizes are permitted, and all addresses are automatically justified to natural boundaries - words on 0-modulo-2 addresses and longwords on 0-modulo-4 addresses. To illustrate the debug's operational sequence, consider the case of a read memory command. The first part of the instruction is serially shifted into the serial port where it is decoded. The ABLR register is then loaded with the second and third opwords of the instruction. Since the debug unit and the core can both be masters of the internal K-bus, a request is made to the core for its control. Once the debug unit has been granted the bus, it completes its access and relinquishes the bus back to the core. The data which was read from memory

Table 5: BDM Command Summary

Mnemonic	Description
READ	Read the sized data at the memory location specified by the longword address.
WRITE	Write the operand-data to the memory location specified by the longword address.
DUMP	Used in conjunction with the READ command to dump large blocks of memory.
FILL	Used in conjunction with the WRITE command to fill large blocks of memory.
GO	The pipeline is flushed and refilled before resuming instruction execution at the current PC.
NOP	NOP performs no operation and may be used as a null command.
WCREG	Write the operand data to the system control register.
RCREG	Read the system control register.
WDREG	Write the operand data to the Debug Module register.
RDREG	Read the Debug Module register.

is then loaded into the RDATA register, transferred to the serial register, and then serially shifted out.

Core address and data registers are accessed through the RAREG and RDREG commands respectively. When performing reads and writes to registers which reside in the CPU, either address registers or data registers, the debug unit actually uses the internal K-bus for these operations. A write command to register D0, for example, converts the BDM opcode into an address, and generates a special "CPU-space" reference on the K-bus. The MMU/Controller module decodes this special access, and signals the processor core of the required operation. The desired operand data is driven onto the data portion of the K-bus by the debug unit, captured by the processor and eventually written to the D0 register after a fixed-length sequence.

Two existing commands were modified to access processor control registers - RCREG and WCREG. Since there is a large number of registers associated with the ColdFire architecture, the BDM commands use a new set of operation encodings to reference them. The second and third words of the command effectively form a 32-bit address used by the debug module to generate a special bus cycle to access the targeted control register. The low-order 12 bits of the address used to access the register are identical to the encoding used by the processor's MOVEC (Move Control Register) instruction.

To load the debug unit's control and breakpoint registers from the processor, the new WDREG instruction is executed. Since the core and serial port can both write to debug registers, once a breakpoint is configured by the emulator, the core could potentially overwrite the setup. Therefore, a mechanism was added so that once the debug registers are configured, a bit in the configuration status register could be set to inhibit any processor-initiated writes to the debug breakpoint registers. This bit can only be modified through commands from the external development system. At this time, only one debug register can be read through the serial port, the CSR register, and the instruction used for reading it is RDREG.

Some detail should be given concerning the use of BDM during reset, since the ColdFire operation differs slightly from other embedded processors. A common situation is one where a system is normally booted from an external ROM. During development it may be desirable to couple the embedded core with on-chip RAM so that an image of the code can be loaded before reset, allowing the developer to easily modify code during testing. The debug architecture allows an external breakpoint to halt the processor *before* reset exception processing has begun. When the external reset pin is asserted, all control registers in the microprocessor are initialized. After the pin has been negated, there is a window of time where the core remains idle before starting the reset exception processing. During this time, the external development system may assert a BKPT to the processor to configure the system. The processor samples for the breakpoint, and if asserted, suspends the reset exception processing. At this point, the processor signals a halt status on the PST output pins, and the external development system may download memory and configure any hardware registers. Once this is completed, the core is restarted using the GO command. The processor's response to this command is dependent on the BDM operations performed while halted. If the core's program counter was *not* modified, then the suspended reset exception processing simply continues. However, if the program counter was loaded, then the normal reset exception processing is bypassed and the processor passes control to the instruction address contained in the PC.

6. CORE OPERATIONS

As noted previously, the occurrence of a breakpoint trigger can be programmed to generate a special debug interrupt. This interrupt is processed at a higher priority than all other interrupts and places the processor in a special emulator

mode of operation. In addition to this method of entry, there are several other ways to enter this mode. Using control bits in the CSR, the processor can be forced to initiate reset exception processing in emulator mode. Additionally, another CSR bit allows the occurrence of a trace mode to force entry into emulator mode. This feature can be useful in developing a single-step debugger in conjunction with an external development system. In all cases, the processor remains in emulator mode until the execution of an RTE instruction.

Finally, the CSR provides a number of additional bits for controlling processor core operation. One bit provides for a single-step mode where the processor executes one instruction and then halts automatically, while another bit enables a non-pipelined mode operation. With this mode enabled, the processor essentially executes a single instruction at a time with no pipeline overlap.

7. CONCLUSIONS

Proper support for debug functionality was a notable goal of the ColdFire design, since it is viewed as critical for widespread adoption of this family of microprocessors. We have attempted to define an architecture that addresses the need for real-time trace and debug, preserves the existing 683xx family BDM functions and provides a consistent interface between the microprocessor and the external development system across a wide variety of potential configurations. The ColdFire architecture maintains compatibility with the large base of embedded development toolsets and takes advantage of the knowledge base of engineers and programmers throughout the world. With over 100 million 68K family microprocessors shipped, the ColdFire architecture provides an exciting new direction to allow Motorola to continue its leadership role as the premier provider of 32-bit embedded solutions.

8. REFERENCES

[1] E. Kuzara, “Real-Time Debugging of Embedded Operating Systems,” in *Proc. of Second Annual Embedded Sys. Conf. East*, April 1994, pp. 241-250.
[2] J. Circello, “ColdFire: A Hot Processor Architecture”, *Byte*, vol. 20, no. 5, pp. 173-174, May, 1995.
[3] *M68000 Family Programmer’s Reference Manual*, Motorola, Phoenix, AZ, 1992.
[4] J. Liband and T. Blakeslee, “Techniques for Real-Time Debugging,” in *Proc. of Fifth Annual Embedded Sys. Conf.*, Oct. 1993, pp. 121-138.

William Hohl is a systems architect with Motorola’s High Performance Embedded Systems Division. He received the BSEE and MSEE degrees from Texas A&M University, and specialized in digital speech processing and digital systems. He previously worked on the 68040 microprocessor and was lead designer of the debug unit for the ColdFire architecture.

Joe Circello is a microprocessor architect for Motorola’s High Performance Embedded Systems Division. With 20 years of experience in mainframes to microprocessors, he is a veteran designer specializing in pipeline organization and performance analysis. While at Motorola, he was pipeline architect for the 68060 and has developed the ColdFire architecture.

Klaus Riedel is a system designer with Motorola’s High Performance Embedded Systems Division. He received a BSEE degree from the University of Texas at Austin. He has previously worked on Motorola’s first ColdFire part - the MCF5102 - and designed the real-time functionality logic on the debug unit.