ColdFire Instruction Set Simulator Software Development Tools, V1.4

Original Release Date: 11 Jun 1999; Revised: 11 May 2005

http://eps.sps.mot.com/~cf/products/cores/system/CFxInstSim_Dbg.pdf

Revision History

Version Number	Revision Date	Effective Date	Author	Description of Changes
V1.1	11 Jun 1999	11 Jun 1999	Joe Circello, Dave Cote, Jim Lamerand	Initial release of proposed enhancements for ColdFire SW Dev Tools
V1.2	17 Jun 1999	17 Jun 1999	Joe Circello	Added +brcnt to br command and added four new commands: db, ex, he, lb
V1.3	29 Mar 2004	29 Mar 2004	Joe Circello	Simplification of document to reflect implemented capabilities in CFxInstSim
V1.4	11 May 2005	11 May 2005	Joe Circello	Copied the V1.3 document into the Freescale templates, added Figure 1-1, the web reference to CFPRM in Section 1.2, and made minor corrections to the `br` and `re` descriptions

Table of Contents

Section 1 Introduction

Section 2 CFxInstSim Debugger Capabilities

Section 1 Introduction

As the promise of “system-on-a-chip” and platform-based designs advance, the time-to-market pressures of most developments almost dictate that the hardware and software components are developed concurrently. In an effort to facilitate this type of co-design, this document details the rudimentary debugger support provided by the ColdFire instruction set simulator (CFxInstSim, CFxISS). Internally, this program is known as *asim*.

It is very important to note the scope of the CFxInstSim development tools is limited to ColdFire cores with attached local memories and external memory subsystems. Stated differently, the peripherals typically included in integrated platform designs are **not** included in this development tool.

1.1 CFxInstSim Overview

The origin of CFxInstSim was to provide a high-speed, C language software model of the ColdFire instruction set architecture (ISA), specifically for use in the design verification process. For all ColdFire developments, the CFxInstSim program serves as the “golden standard” and defines the expected results on an instruction-by-instruction basis. In its simplest form, *CFxInstSim models the behavior of a ColdFire core connected to a memory*. Specifically, a “hex” memory image is loaded into the program, and CFxInstSim then executes the code, a single instruction at a time, beginning with reset exception processing and continuing until a **HALT** instruction is executed. In its most common use, the CFxInstSim program provides the expected results for design verification. This typically includes the register programming model at the completion of each instruction, and the final memory image.

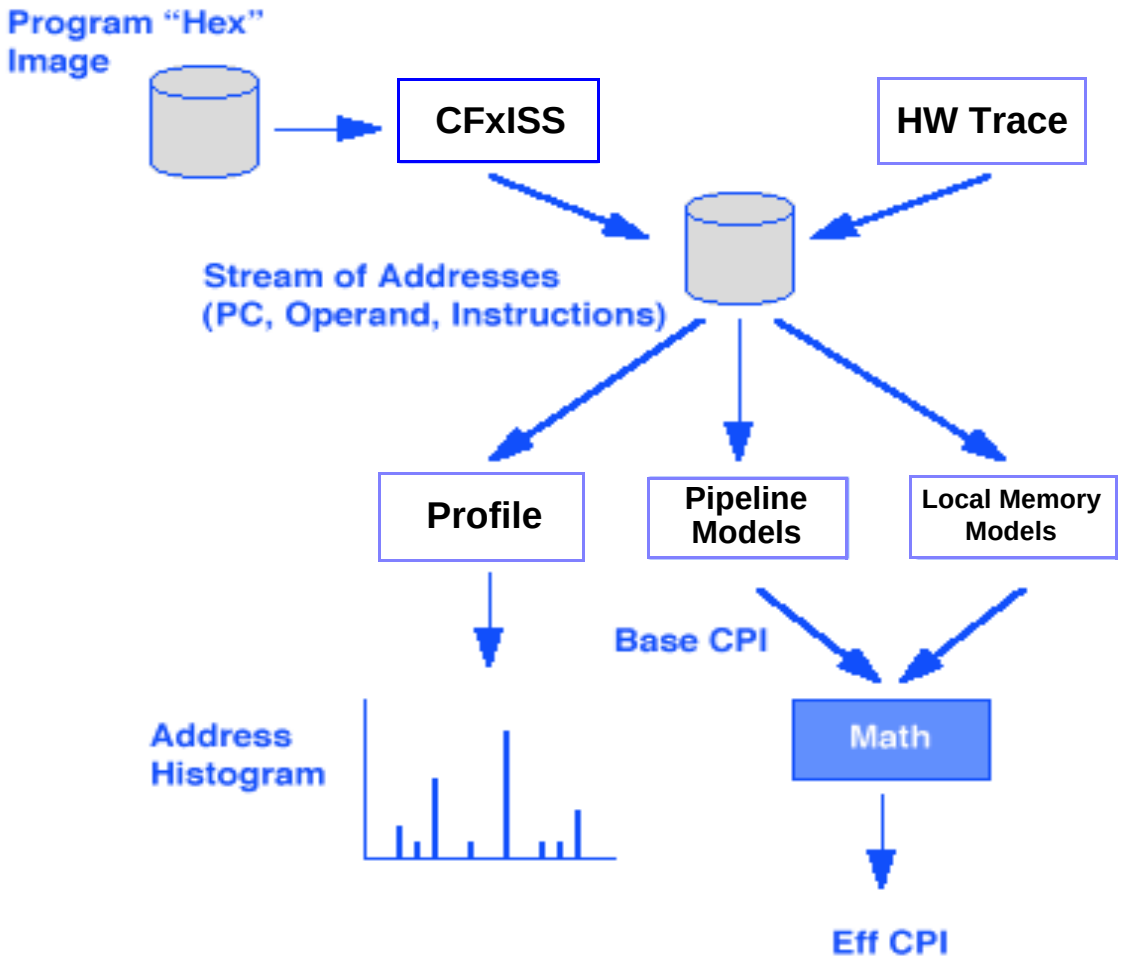
Since it serves as the foundation for the ColdFire verification methodology, any and all changes to the ISA are reflected in the CFxInstSim program. As a result, the program always supports the most current specification of the instruction set architecture. As an example, as the Enhanced Multiply-Accumulate (EMAC) module was being designed, the CFxInstSim program was updated to support this new functionality. Stated differently, the *CFxInstSim program always provides a known-good implementation of the ISA* before the corresponding hardware design is available. This contrasts sharply with 3rd-party development tools, which generally lag the availability of standard-product devices, often by many months. Additionally, tool support for customer-specific designs from this developer community may not even be possible.

As the CFxInstSim program is used for all ColdFire core developments, it is completely unaware of any underlying hardware microarchitecture. This means, by definition, that the CFxInstSim program has no notion of machine cycles, or time. The “smallest” operation is the execution of a single machine instruction. Note this basic approach means that certain asynchronous events, most notably interrupts and external memory error terminations, simply cannot be handled by CFxInstSim. This fundamental limitation to CFxInstSim does not impact the design verification process, but the complication of these types of events and how they are handled is beyond the scope of this document. Suffice it to say, these types of asynchronous events are exercised and tested thoroughly as part of the standard ColdFire verification process.

Additionally, the CFxInstSim program is typically run in a “batch” environment, where the beginning memory image is the input, and the program simulates the execution of every instruction outputting the required programming model information and final memory image. There is no notion of any type of interactive environment while operating in this mode. The performance of CFxInstSim is roughly measured in the 100,000 - 200,000 instructions per second, when running on a Sparc20-class workstation.

In addition to its use as the golden standard for verification purposes, CFxInstSim has been enhanced to provide input data to a large collection of ColdFire performance analysis tools. This set of tools includes memory address profilers, dynamic opcode analyzers, and a variety of “architectural models” for exploring microarchitectural trade-offs, local memory effects, etc. For this type of study, programs are “executed” using CFxInstSim with the appropriate outputs enabled, and the resulting data piped into the given architectural model. **Figure 1-1** depicts this performance analysis process used to calculate the Effective Cycles per Instruction (CPI) metric.

Figure 1-1 ColdFire Performance Analysis



1.2 CFxInstSim Summary

CFxInstSim is:

- C language model of the ColdFire ISA
- Primarily used for design verification
- Executes a memory image, instruction-by-instruction
- Starts with reset exception and continues until a HALT instruction
- Standard outputs are the register programming model after each instruction and final memory image
- Optional outputs for ColdFire performance analysis tools
- Independent of underlying hardware microarchitecture, and therefore has no notion of time
- Cannot support asynchronous events like interrupts and memory error terminations
- Typically operates in a batch mode
- 100K - 200K instructions per second performance

The ColdFire instruction set architecture is detailed in the Programmer's Reference Manual (PRM), available on the Freescale public web site at:

`http://www.freescale.com/files/dsp/doc/ref_manual/CFPRM.pdf`

Section 2 CFxInstSim Debugger Capabilities

As detailed in the previous section, CFxInstSim is a C language implementation of the ColdFire Instruction Set Architecture. In an effort to provide a basic set of debugger capabilities to this program, a subset of the functionality supported by the standard **68K/ColdFire dBUG** package is included. *It should be noted the information provided in the CFxInstSim Unix "man" page supercedes the documentation provided in this paper*. This document is intended to provide a more detailed definition of the debugger support provided in the ISS.

The dBUG package is a firmware program included in all 68K/ColdFire Family evaluation boards. The firmware provides a self-contained command line environment for monitor and debug capabilities. This paper details the subset of dBUG commands implemented in the CFxInstSim program. The set of dBUG commands implemented in the CFxInstSim program are:

Table 2-1 dBUG Commands Supported by CFxInstSim

Command Mnemonic	Description
bf	Block Fill
bm	Block Move
br	Breakpoint
db	Delete Breakpoint
di	Disassemble
ex	Execute Command File
go	Execute
he	Command Syntax Help
lb	List Breakpoints
md	Memory Display
mm	Memory Modify
qu	Quit
rd	Register Display
re	System Reset
rm	Register Modify
sb	Set Number Base
st	Step Over Subroutine
tr	Trace
ve	Show Current Version

This rudimentary debugger functionality is built upon a set of 32 breakpoints and the basic run control needed to support these types of operations. The breakpoints are organized as 31 user-defined values plus a temporary breakpoint used in conjunction with specific dBUG commands.

The CFxInstSim program is executed from the system command line with the starting memory *hex* image passed as an argument.

The user interface to dBUG is the command line. A number of features have been implemented to create an easy and intuitive command line interface. The command line prompt is "`dBUG>` ". Any valid command can be entered from this prompt. All commands begin with a 2-character identifier and are not case-sensitive.

The following paragraphs define the specific dBUG commands implemented in the CFxInstSim program. The descriptive material contained here makes liberal use of the existing documentation, particularly *M5206EC3 User's Manual, Revision 1.3*. In these descriptions, optional fields are enclosed with braces {}.

2.1 bf Block of Memory Fill

Syntax: `bf{.b|.w|.l}  begin end data`

The `bf` command fills a contiguous block of memory starting with address `begin`, stopping immediately before address `end` with the value `data`. The width field `{.b|.w|.l}` is an optional modifier that defines the size of the data being written, with the default being word (16-bit size).

The values for addresses `begin` and `end` are absolute hexadecimal values. The value for `data` is a number converted from the user-defined radix, with the default being hexadecimal.

Examples:

To fill a memory block starting at address 0x1000 and ending at address 0x4000 with a value of 0x1234, the command is:

```
bf   1000 4000 1234
```

To fill a block of memory starting at 0x2000 and ending at 0x3000 with a byte value of 0xAB, the command is:

```
bf.b 2000 3000 ab
```

2.2 bm Block of Memory Move

Syntax: `bm begin end dest`

The `bm` command moves a contiguous block of memory starting with address `begin`, stopping immediately before address `end` to the new address `dest`. The `bm` command copies memory as a series of bytes and does not alter the original block.

The values for addresses `begin`, `end` and `dest` are absolute hexadecimal values.

Example:

To copy a memory block starting at address 0x1000 and ending at address 0x2000 to location 0x8000, the command is:

```
bm   1000 2000 8000
```

2.3 br Breakpoint

Syntax: `br{.b|.w|.l|.x} addr {-r|-w} {data} {+brcnt}`

The CFxInstSim implementation of dBUG includes 32 breakpoints (31 user-defined plus one temporary value used in conjunction with specific commands), which can be instruction addresses (PC values) or operand addresses with an optional data value and reference type specifier included. An optional breakpoint count allows `brcnt` occurrences of the breakpoint to be *ignored* before triggering. The value for address `addr` is an absolute hexadecimal value.

If a PC breakpoint is desired, the command syntax is simply: `br addr`. When a PC breakpoint triggers, control is returned to the dBUG prompt *before* the breakpointed instruction has executed.

If an operand address breakpoint is desired, then the reference size specifier *must* be included. These specifiers are defined as:

`.b`	Byte-sized (8-bit) reference
`.w`	Word-sized (16-bit) reference
`.l`	Longword-sized (32-bit) reference
`.x`	*Any size* reference

Additionally, an optional reference type specifier may be included. These specifiers are:

`-r`	Breakpoint only if read reference
`-w`	Breakpoint only if write reference

Thus, the resulting command syntax for an operand breakpoint is:

```
br.{b|w|l|x} addr {-r|-w}
```

Finally, an operand breakpoint can be further qualified to include a specific data value.

When an operand breakpoint triggers, control is returned to the dBUG prompt at the next instruction boundary, i.e., immediately *after* the instruction has completed execution.

The operand address used in the breakpoint compare function is always the *starting* reference address and size, independent of whether the address is aligned or misaligned. Addresses which are 0-modulo-*size* values are said to be aligned, while all other references are misaligned. All ColdFire cores and the CFxInstSim program “decompose” a misaligned operand reference into a series of smaller, aligned accesses. For purposes of a misaligned breakpoint definition, the starting address and size of *any* of the misaligned accesses may be specified. For this class of operand accesses, the ColdFire processor cores and CFxInstSim behave in a same manner.

An optional count field (`+brcnt`) can be applied to any type of breakpoint. If included, this decimal number defines the number of times the breakpoint is ignored before actually triggering

and returning control to the dBUG prompt. The default is 0.

If the user attempts to set a 32nd breakpoint, an error message is returned.

Examples:

To enable a PC breakpoint at address 0x1000, the command is:

```
br   1000
```

To enable a PC breakpoint on the tenth execution of address 0x1000, the command is:

```
br   1000 +9
```

To enable an operand read (any size) breakpoint at address 0x2000, the command is:

```
br.x 2000 -r
```

To enable an operand breakpoint at address 0x3000 for any-sized, any-type data reference, the command is:

```
br.x 3000
```

To enable an operand byte read breakpoint at address 0x2000 with a data value of 0xAB, the command is:

```
br.b 2000 -r ab
```

To enable an operand longword write breakpoint at address 0x3000 with a data value of 0x12345678, the command is:

```
br.l 3000 -w 12345678
```

Finally, consider a misaligned longword operand reference to address 0x3001. The processor converts this reference into 3 smaller, aligned accesses: a byte access at 0x3001, a word access at 0x3002 and a byte access at 0x3004. To enable an operand breakpoint on the fifth misaligned longword read at address 0x3001, *any* of the following commands can be used:

```
br.b 3001 -r +4
br.w 3002 -r +4
br.b 3004 -r +4
```

In general, the effects of operand misalignment can be ignored if the starting byte address of the operand is always used in conjunction with the .x size specifier. For this example:

```
br.x 3001 -r +4
```

2.4 db Delete Breakpoints

Syntax: **`db number | "all"`**

The `db` command deletes a single breakpoint condition, specified by the decimal value, `number`, or, if the argument is `all`, the complete set of breakpoints, including the temporary value.

The association between a breakpoint condition and the corresponding number can be displayed using the `lb` command (list breakpoints). The temporary breakpoint register is always #0.

Examples:

To delete breakpoint #2, the command is:

```
db   2
```

To delete all breakpoints, the command is:

```
db   all
```

2.5 di Disassemble

Syntax: di {addr}

The `di` command disassembles the next 16 instructions of the target code starting at address `addr`. The value for address `addr` is an absolute hexadecimal value. If the optional `addr` value is not specified, the `di` command uses the current value of the program counter as the starting address.

This command can be repeated, displaying the *next* 16 instructions, by simply pressing the **RETURN** key.

Example:

To disassemble code starting at address 0x1000, the command is:

```
di   1000
```

2.6 ex Execute Command File

Syntax: **`ex filename`**

The `ex` command causes the CFxInstSim program to read the file defined by `filename`, and execute the dBUG commands contained in that file.

Example:

To execute the series of commands in the file named `setup`, the command is:

```
ex   setup
```

2.7 go Execute

Syntax: go {addr}

The `go` command executes the target code starting at address `addr`. The value for address `addr` is an absolute hexadecimal value. If the optional `addr` value is not specified, the `go` command begins execution at the value currently defined by the program counter.

When the go command is executed, all user-defined breakpoints are enabled as the application is executed. Command line control is returned to the dBUG prompt when a breakpoint is triggered, or the code encounters a halt condition, either from the execution of a **HALT** instruction, or the occurrence of the catastrophic fault-on-fault error condition. In all cases, the exit cause is reported before the dBUG prompt is displayed.

This command can be repeated by simply pressing the **RETURN** key.

Examples:

To execute code starting at the current program counter, the command is:

```
go
```

To execute code starting at address 0x1000, the command is:

```
go   1000
```

2.8 he Command Syntax Help

Syntax: **`he`**

The `he` command displays a one-line syntax summary for every dBUG command supported by CFxInstSim.

Example:

To display the one-line syntax summary, the command is:

```
he
```

2.9 lb List Breakpoints

Syntax: `lb`

The `lb` command displays a list of all active breakpoints. The temporary breakpoint always is defined as breakpoint #0, while the 31 user-defined breakpoints are numbers #1-31. This command can be used in conjunction with the `db` command to delete specific breakpoint conditions.

Example:

To display all active breakpoints, the command is:

```
lb
```

2.10 md Memory Display

Syntax: **`md{.b|.w|.l} {begin} {end}`**

The `md` command displays a contiguous block of memory starting with address `begin`, and stopping with address `end`. The width field `{.b|.w|.l}` is an optional modifier that defines the size of the data being displayed, with the default being word (16-bit size).

The values for addresses `begin` and `end` are absolute hexadecimal values. If no beginning address is specified, the `md` command uses the last displayed address as the starting point. If no ending address is specified, the `md` command displays 128 bytes of data.

This command can be repeated by simply pressing the **RETURN** key. It continues with the last displayed address.

Examples:

To display memory at address 0x1000, the command is:

```
md   1000
```

To display a range of longwords from address 0x2000 to address 0x3000, the command is:

```
md.l 2000 3000
```

2.11 mm Memory Modify

Syntax: `mm{.b|.w|.l} addr {data}`

The `mm` command modifies memory at address `addr`. The width field `{.b|.w|.l}` is an optional modifier that defines the size of the data being modified, with the default being word (16-bit size).

The value for address `addr` is an absolute hexadecimal value. The value for `data` is a number converted from the user-defined radix, with the default being hexadecimal.

If the `data` value is included in the command line, the `mm` command immediately writes the 8-, 16- or 32-bit value into the given memory location. If the optional data value is *not* provided in the command line, the `mm` command enters a loop. This interactive loop obtains a data value from the user, writes it into the memory address, increments the address by the data size and repeats. The loop terminates when the user input is a ".", i.e., a period.

Examples:

To write a byte into memory at address 0x1000, the command is:

```
mm.b 1000 ab
```

To write a longword at address 0x2000, the command is:

```
mm.l 2000 12345678
```

To interactively modify memory beginning at address 0x3000 with longword data values, the command is:

```
mm.l 3000
```

2.12 qu Quit

Syntax: **`qu`**

The `qu` command terminates the execution of CFxInstSim.

Examples:

To exit the simulator, the command is:

```
qu
```

2.13 rd Register Display

Syntax: `rd {core}`

The `rd` command displays the ColdFire core's register set. The default display includes the processor's basic registers including the program counter, the general-purpose An/Dn registers, the {E}MAC registers along with the 16-bit Status Register. If the optional `core` specifier is included, the processor's registers plus the various memory configuration registers, including the Vector Base Register (VBR), the Cache Control Register (CACR), the Access Control Registers (ACR*) and all local-memory base address registers (RAMBAR*, ROMBAR*), are displayed.

Examples:

To display the entire register set of the ColdFire core, the command is:

```
rd   core
```

To display the processor's basic register set, the command is:

```
rd
```

2.14 re System Reset

Syntax: `re`

The `re` command performs a “system reset” on the CFxInstSim program. The program loads the stack pointer (A7) with the contents of memory address 0x0 and loads the program counter (PC) with the contents of memory address 0x4. Control is then passed back to the dBUG prompt. The system memory image is not affected.

Example:

To reset the simulator, the command is:

```
re
```

2.15 rm Register Modify

Syntax: **`rm register data`**

The `rm` command modifies the contents of a register from the ColdFire core's register set. The allowable values of the `register` specifier are:

```
pc, d[0-7], a[0-7], sr
acc, macsr, mask                                  (if MAC)
acc[0-3], accext01, accext23, macsr, mask         (if EMAC)
vbr, cacr, acr[0-3], rambar[0-1], rombar[0-1]
```

The value for `data` is a longword number converted from the user-defined radix, with the default being hexadecimal.

Example:

To modify the contents of the d0 register to a value of 0x12345678, the command is:

```
rm   d0 12345678
```

2.16 sb Set Data Radix (Base)

Syntax: `sb base`

The `sb` command allows the user to define the setting of any user-configurable options. The single option of this type is the default radix used for converting `data` values associated with the `bf`, `br`, `mm`, and `rm` commands.

The default is hexadecimal (base 16), and the other choices are binary (base 2), octal (base 8), and decimal (base10). Thus, the set of allowable values for `base` are the decimal values [2,8,10,16].

Example:

To configure the simulator to use decimal representation for data, the command is:

```
sb   10
```

2.17 st Step Over

Syntax: `st`

The `st` command allows the user to “step over” the execution of an entire subroutine rather than trace every instruction within the function. The `st` command sets a temporary breakpoint on the next instruction (past the instruction currently defined by the program counter), and is typically used when breakpointed on a **JSR** or **BSR** instruction. For this case, a `go` command causes the target function to be completely executed before breakpointing on the instruction following the subroutine call.

The `st` command can be used while breakpointed on instructions other than the subroutine calls, but special care must be used as the temporary breakpoint may never be encountered.

Example:

To step over the execution of a subroutine, the command is:

```
st
```

2.18 tr Trace

Syntax: `tr {number}`

The `tr` command allows the user to single-step through the application code. If `{number}` is provided, then "`number`" instructions are executed before control is returned to the dBUG prompt. If `{number}` is not provided, a single instruction is executed. The value of `number` is a decimal specifier.

Examples:

To trace one instruction at the program counter, the command is:

```
tr
```

To trace twenty instructions from the program counter, the command is:

```
tr   20
```

2.19 ve Show Program Version

Syntax: **`ve`**

The `ve` command displays the current source code version of the CFxInstSim program. It represents the specific version number of the simulator which is maintained under change control as part of a configuration management system.

Examples:

To display the version number of CFxInstSim, the command is:

```
ve
```