5 CRIS COMPILER SPECIFICS

5.1 CRIS Compiler Options

This document is a portion of the GNU C Compiler documentation, which describes compiler '-m' options for different target processors (as for instance: Using and Porting GNU CC, by Richard M. Stallman, published by Free Software Foundation, Inc. 1998).

These specifications may be subject to changes with future revisions of the CRIS GCC.

The following '-m' options are defined for the CRIS architecture family:

`-mcpu=CPU_MODEL`

`-march=CPU_MODEL`

These options produce code that runs on CPU_MODEL. Values 'etrax4', 'etrax100' and 'vN', where N is in the range from 0...10 are recognized. When 'vN' is specified, N denotes the version-register contents of the targeted CPU model.

`-mtune=CPU_MODEL`

is like '-mcpu=CPU_MODEL' but does not affect the instruction set, only the applicable scheduling parameters.

`-metrax4`

`-mno-etrax4`

Set (unset) '-mcpu=v3' additions to the base instruction set.

`-metrax100`

`-mno-etrax100`

Set (unset) '-mcpu=v8' additions to the base instruction set and 32-bit general alignment.

```
-mconst-align
-mdata-align
-mstack-align
-m16bit
-m32bit
-m8bit
-mno-const-align
-mno-data-align
-mno-stack-align
```

Align constants, data and stack respectively, to 16-bit (two bytes) data boundary by alignment directives, or by rounding up the size of the stack-frame. Only individual variables are affected; the (unaligned) ABI is unaffected. Saying ' –m16bit' is equivalent to all of '–mconst-align2, '–mdata-align', and '–mstack-align'. This is the default when the base ('v0') instruction set is specified. Saying '–m32bit' means rounding them up to a 32-bit data boundary. This is the default for the 'v8' instruction set and up. Specifying '-m8bit' means do not align anything. The 'no-' counterpart disables alignment of that entity.

```
-mmax-stack-frame=SIZE
```

Warn when the stack-frame exceeds SIZE bytes.

```
-mprologue-epilogue
-mno-prologue-epilogue
```

Do (do not) output a prologue and epilogue for any function. For code compiled with the '-mno-prologue-epilogue' option, it is necessary to add a function prologue and epilogue through 'asm' statements.

5.2 CRIS Preprocessor Macros

The GCC port sets the following preprocessor macros:

```
__cris__
__CRIS__
__GNU_CRIS__
```

These three macros are always set to '1'.

```
__arch_X
```

This macro is set to '1' for the options '-mcpu=X' and '-march=X' (where the variable 'X' is the value entered for CPU_MODEL). See section 5.1 *CRIS Compiler Options* for an explanation of these options.

The macro '__tune_X' is set for the option '-mtune=X' in the same way as the macro '__arch_X' is for '-march=X'.

Note: The underlining at the beginning and end of the macros above represents two underline spaces.

5.3 The CRIS ABI

5.3.1 Introduction

This is a description of the CRIS GCC (GNU C Compiler) ABI (Application Binary Interface), the binary-level conventions for the ETRAX 100 processor. An application binary interface defines a system interface for executing compiled programs. Among the conventions that an ABI establishes are register usage, calling conventions, parameter passing, and layout of data.

These specifications may be subject to changes with future revisions of the CRIS GCC ABI.

5.3.2 CRIS GCC fundamental data types

This is how C and C++ data types correspond to CRIS GCC data types, see tableTable 1-4, “Data types supported by CRIS,” on page 17 .

A signed, unsigned, or plain (in C++) **char** is a signed or unsigned byte (or 8-bit integer).

A signed or unsigned **short int** is a signed or unsigned word (or 16-bit integer).

A signed or unsigned **int** and **long** is a signed or unsigned dword (or 32-bit integer).

Pointers to any type are represented as 32-bit integer entities.

Enumerated types in C and C++, **enum**, are represented as integer objects, 32-bit dwords.

The floating point types **float** and **double** are represented as 32-bit IEEE-754 floating point numbers:

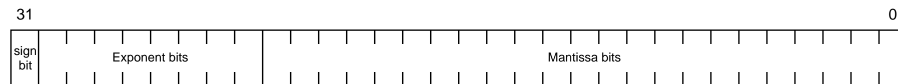

Figure 5-1 32-bit floating point number

The type **long double** is represented as a 64-bit IEEE-754 floating point number, with the lower part of the mantissa in the dword at the lower address.

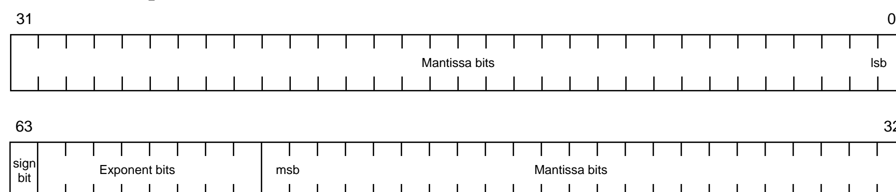

Figure 5-2 64-bit floating point number

5.3.3 CRIS GCC object memory layout

The memory layout of a structure has each member at increasing addresses, without any alignment padding in between members. The size of the structure is, therefore, the sum of each of the sizes of the elements (with the exception of zero bitfields, which align to the next byte boundary).

Example of the structure layout of the CRIS ABI:

```
struct example
{
    char c;               /* 1 Byte,  offset 0 */
    short s;              /* 2 Bytes, offset 1 */
    int i;                /* 4 Bytes, offset 3 */
    long l;               /* 4 Bytes, offset 7 */
    float f;              /* 4 Bytes, offset 11 */
    double d;             /* 4 Bytes, offset 15 */
    long double ld;       /* 8 Bytes, offset 19 */
    char s[6];            /* 6 Bytes, offset 27 */
};
```

The size of the `struct example` is 33 Bytes.

Bitfields span over any byte, word or dword boundaries. A zero-length bitfield aligns to the next byte boundary. The first declared field is in the lowest bits of the lowest address at the starting address.

Compiler options specify whether objects have byte, word or dword alignment. Code must not assume that objects are laid out at stricter alignments than bytes. Compiler options specify the actual alignment. For example, -m8bit specifies that objects are always byte-aligned, while the default is 16-bit alignment. Note that options specifying a processor-version also implicitly control the alignment of objects.

5.3.4 CRIS GCC calling convention

Arguments shorter than or equal to 32 bits are passed by value. Integral types smaller than 32 bits are promoted to the corresponding 32-bit types by the same rules as in ISO C. Larger entities are passed by reference by passing a pointer to a read-only value. This means that the callee has to copy that value if it wants to modify it. The first four parameters (by value or reference) to a function are passed in registers R10..R13, starting with the first parameter in R10. Starting with the fifth parameter, parameters are passed on the stack, starting with offset zero upon entry to the called function (not including any return address).

Return values shorter than or equal to 32 bits are returned in register R10. Structure return values are passed (to the called function) by reference in register R9 to a caller-allocated area. (This may change so that structures less than or equal to 128 bits are returned in registers R10..R13.) The **this** pointer in C++ is passed as an invisible first argument in R10 (i.e. the first argument to a non-static member function ends up in register R11 and so on).

Registers R9..R13, and SRP (except any return values in register R10) are assumed clobbered upon return from the function. Registers R0..R8 must have the same contents upon return from, as before the call to the function.

5.3.5 Stack frame layout

As can be seen below, the stack does not have a static layout except for the order of its components. It may, in fact, be collapsed and empty (not even a return address), having zero offset from the callers stack.

	(higher address) [...]	
Stack pointer value at function entry	Parameter #7	If more than four parameters.
	Parameter #6	
	Parameter #5	
	Parameter #4 (R13)	Only if variable-arguments function, and if the parameters are not named.
	Parameter #3 (R12)	
	Parameter #2 (R11)	
	Parameter #1 (R10)	
	Return address	Only if non-leaf function.
Frame pointer value	Saved fp (R8)	Only if a function needs a frame pointer.
	Local variables and internal temporaries	If the function has any.
	Preserved register values, R0..R7/R8	If modified, and R8 only when no frame pointer is needed.
	Variables of variable size and alloca() storage	If used.
Stack pointer value	Parameter storage when calling for parameter #5 and up (lower address)	If used.

Figure 5-3 Stack frame layout

Very few functions need a frame pointer. When a frame pointer is needed, register 8 is used. The frame pointer value is derived from the stack pointer value at the beginning of the function.

If functions with more than four parameters are called, the memory room for the parameters from the fifth and up is allocated in the stack frame of this function.

For functions with a variable number of parameters the function itself is responsible for storing any necessary portion of registers R10..R13 as indicated in figure 5-3 on page 162. The "va_list" type is a pointer to an array of Dword-sized parameters or (by reference) pointers to parameters.