GCC for MMIX

the ABI

Hans-Peter Nilsson
hp@bitrange.com

<URL:http://bitrange.com/mmix/mmixfest-2001/>

This document is a webbification of some slides and notes presented at the MMIXfest 2001, October 6 in Munich, Germany, hosted by the Computer Science and Mathematics Department, Munich University of Applied Sciences.

You are encouraged to report errors in the original document or opinions about it to hp@bitrange.com.

Contents

Location of the original

The original of this document is located at http://bitrange.com/mmix/mmixfest-2001/mmixabi.html.

Summary of changes

(No changes since the original webbification on 2001-10-10.)

Abstract

I'll present an ABI based on Knuth's documents. I want your opinions about some details where I'm undecided. I'll also sketch a few details of an alternative ABI.

Goals for the GCC MMIX port

Performance of programs for the MMIX architecture relies on keeping as many variables as possible in registers, to minimize memory accesses by other means than through the register stack and instruction fetches. So first hand, registers should be used rather than a traditional memory stack.

Another goal is for the assembly code to be usable as input for further modification by a human. The generated code should as far as possible be compatible with mmixal and conventions in Knuth's texts. There are currently (2001-10-10) some non-conformance issues, which will be fixed wherever possible.

Function call conventions as implied by Knuth's texts

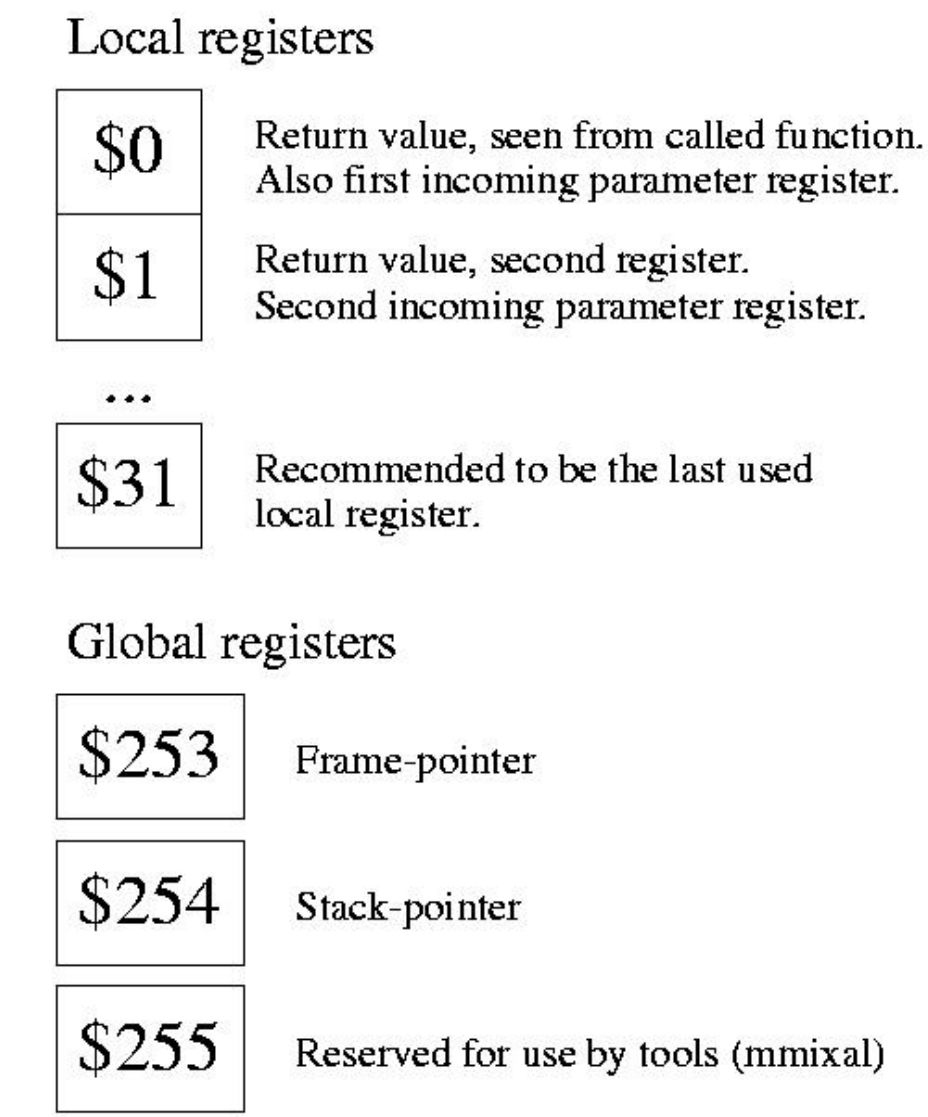
```
long zz = 0;

void x (void)
{
 zz = fn (3, 2, 1);
}

long fn (long a, long b, long c)
{
 return a + b * c;
}
```

```
zz   OCTA 0

x    GET $0,rJ
     SETL $2,#3
     SETL $3,#2
     SETL $4,#1
     PUSHJ $1,fn
     PUT rJ,$0
     GETA $0,zz
     STOU $1,$0,0
     POP 0,0

fn   MULU $1,$1,$2
     ADDU $0,$0,$1
     POP 1,0
```

Recommended MMIX function call convention,
as derived from Knuth's texts.

It is described (in [ref 1]) that parameters are usually passed in registers $0 and up, as seen by the called function. Further, the return address is passed in rJ, using PUSHJ and PUSHGO for function calls and POP for returning ([ref 3]). Return values are put in $0 and up, as seen by the returning (called) function.

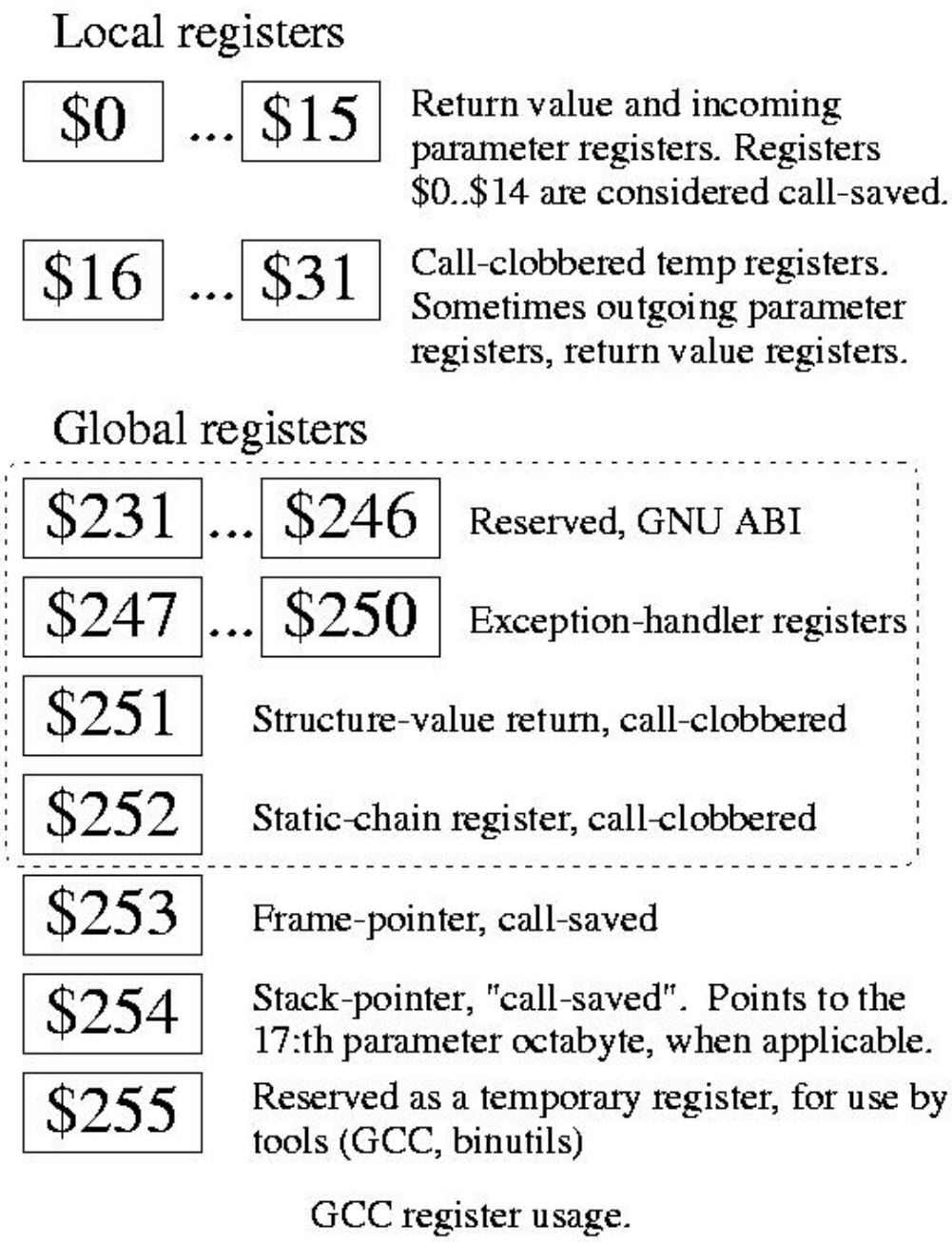
Register usage as implied by Knuth's texts



Recommended MMIX register usage,
derived from Knuth's texts.

In [ref 2], it is mentioned that $255 is used by mmixal for instruction expansion under control of the -x option. This happens when loading out-of-range addresses, making register $255 unsuitable for literal use in assembly programs. At the same place, it is suggested that functions that need a memory stack can use $254 as a stack pointer and when necessary, $253 as a frame-pointer.

Register usage by the GCC port



GCC register usage.

The number of local registers (the value of special register rG), is defined to never be lower than 32. It is also recommended (in [ref 1]) that a program should not assume more than 32 available local registers. This number is higher than most human-written C-functions will use anyway. Therefore, the number of incoming or outgoing registers used for parameters cannot reasonably be higher than 32. In an attempt to keep things simple (and to work around restrictions in GCC), it seemed best to split that number even for incoming and outgoing parameters; the number of registers used for parameters is 16.

Hopefully, this should also minimize the number of register moves necessary to get incoming registers "out-of-the-way" when calling other functions.

Please note that register $16 being an outgoing parameter register is just a first approximation. I'll explain briefly the GCC-related issues later. The 17:th parameter and beyond are passed on the stack, with $254 (the stack-pointer) pointing to the first parameter in the called function.

Just as mmixal does, the GNU binutils uses register $255 for its own purposes when loading addresses. GCC also uses it for temporary purposes in some cases, but only for instructions that the assembler and linker will not expand.

The stack pointer $254 is set up in the code between Main and main, and must have the same value when a function returns as when it was entered.

If register $253, the suggested frame-pointer, is used in a function, it must likewise be saved so it has the same value at function exit as at function entry.

The GCC implementation reserves a few registers.

There's a GCC extension for nested functions, that for some codes circumstances need to pass a context pointer; a pointer to the local variables of the enclosing function when a pointer to a nested function is passed to another function[1]. That pointer is passed in register $252.

Register $251 is used for the structure return pointer. Registers $247..$250 are used for C++ exception handling, and registers $231.. $246 are reserved for the GNU ABI. There is no need for a function to preserve these register values, but it must be prepared that a called function may change them.

This is just the current (2001-10-10) work-in-progress mapping. These allocations are not final, and some of the uses can certainly share the same register. For the imaginary typical MMIX programmer these allocations should not matter, as they do not collide with GREG-allocated registers (unless the program is short on registers).

Type layout

C/C++	MMIX
char	byte
short int	wyde
int	tetra
long int	octa
long long	octa
float	"single float": 32-bit IEEE
double	64-bit IEEE
long double	64-bit IEEE

MMIX fundamental C and C++ type mapping

MMIX can handle 8, 16, 32 and 64-bit data. A reasonable mapping of C types exposes all these sizes so e.g. a C programmer wanting to write for MMIX can do it simply by using standard types[2]. The fundamental address unit type in C is char (sizeof char is always 1), so BYTE maps naturally to it. The C "char" type is by default signed, because this is the case for most gcc ports and therefore would presumably lead to the fewest compatibility problems. The rest of the normal C integer types are "short int", "int" and "long int". They naturally map to wyde, tetrabyte and octabyte. Incidentally, this is the same mapping as the alpha gcc port uses. The gcc type "long long"[3] has to map to octabyte, otherwise cross-compilation from a 32-bit host is not possible. This is a gcc restriction, but might not matter too much, since on a "long long" is usually also 64 bits, this being natural for existing 32-bit hosts.

The normal C floating point types are float, double and long double. MMIX has 64-bit IEEE floating point quantities and limited operations on 32-bit quantities, "short floats". They naturally map to double and float, with long double an alias for double.

Memory layout

Each of the fundamental types must be mapped to an address at least a multiple of its size, so called "naturally aligned". To be theoretically addressable with a GETA, individual variables are aligned to tetrabyte alignment.

```
struct x {
  char a;
  int b;
  long c;
  char d;
} x1 = {                LOC @+(8-@)&7
  1,            x1      BYTE 1
                        LOC @+3
  2,                    TETRA 2
  3,                    OCTA 3
  4                     BYTE 4
};                      LOC @+7

    sizeof (struct x) == 24
```

MMIX GCC structure layout example

Similarly for structures, the structure layout has members naturally aligned, with padding inserted where necessary. The alignment of a structure is that of its largest contained type. The size is a multiple of that type, with padding inserted at the end of the structure.

Parameter passing in the GCC port

If a parameter fits in a register, it is passed by-value, with integer types extended to 64 bits by the caller. Otherwise, it is passed by reference (as a pointer to the original), and the called function has to copy it, in case it needs a local modifiable copy.

The same goes for parameters passed on the stack; integer types are promoted, passed as 64 bits, with larger-than-64-bit types passed by reference.

Returning function values in the GCC port

Scalar values are returned in $0 as seen by the calling function. We usually don't have to worry about the register hole.

There is an exception: complex values together larger than 64 bits are returned in $0 and $1 as seen by the calling function. For these multi-register return values, the called function compensates for the hole, so the calling function sees the register with $0 being the natural first part; like the order allocated for registers.

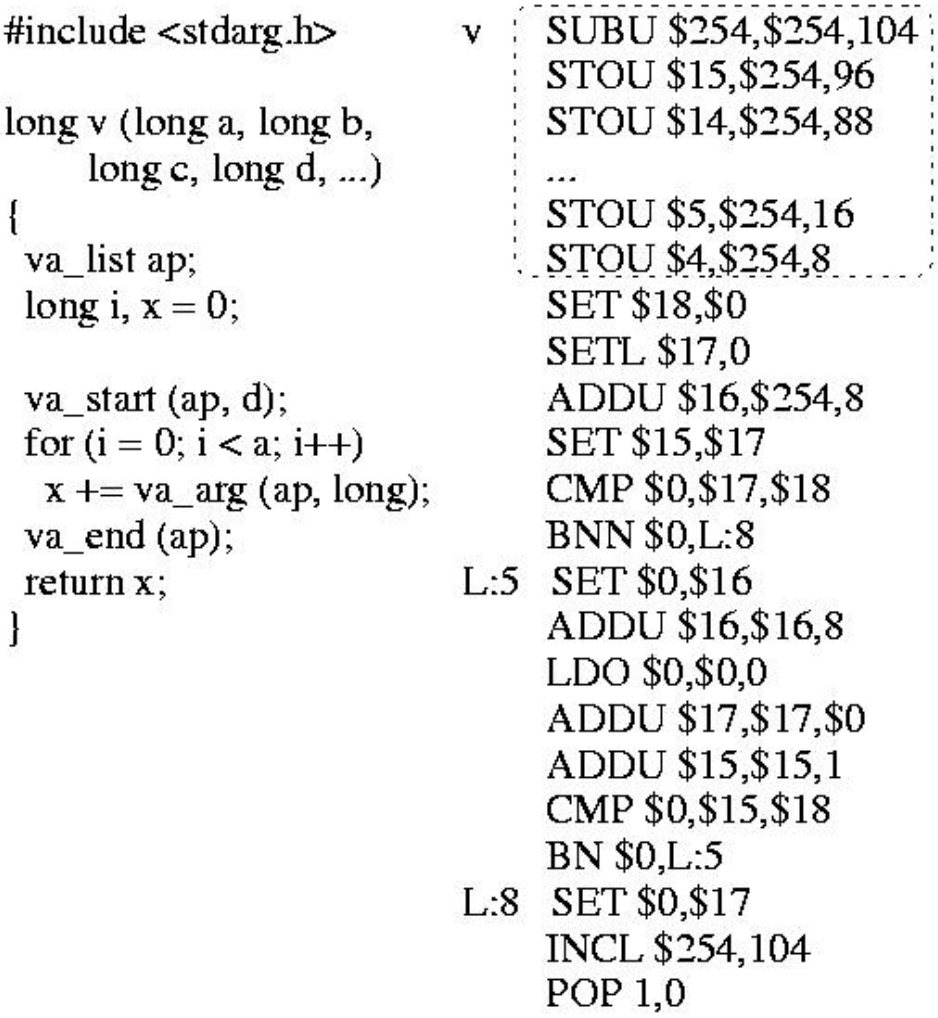
```
struct s { char a; char b; };

struct s
sf (struct s ps)
{
 ps.a++;
 return ps;
}
```

```
sf   SUBU $254,$254,16
     SET $1,$254
     STWU $0,$1,0
     LDB $2,$1,0
     ADDU $0,$2,1
     STBU $0,$1,0
     STBU $0,$251,0
     LDB $1,$1,1
     STBU $1,$251,1
     SET $0,$251
     INCL $254,16
     POP 1,0
```

A function returning a structure through the structure-return register, $251.

Structures are returned specially. The caller passes a pointer to an area where the structure contents is to be stored by the called function, regardless of the structure size. The structure-return pointer is passed in register $251.

Functions with a variable number of parameters



```
#include <stdarg.h>

long v (long a, long b,
        long c, long d, ...)
{
  va_list ap;
  long i, x = 0;

  va_start (ap, d);
  for (i = 0; i < a; i++)
    x += va_arg (ap, long);
  va_end (ap);
  return x;
}
```

```
v   SUBU $254,$254,104
    STOU $15,$254,96
    STOU $14,$254,88
    ...
    STOU $5,$254,16
    STOU $4,$254,8
    SET $18,$0
    SETL $17,0
    ADDU $16,$254,8
    SET $15,$17
    CMP $0,$17,$18
    BNN $0,L:8
L:5 SET $0,$16
    ADDU $16,$16,8
    LDO $0,$0,0
    ADDU $17,$17,$0
    ADDU $15,$15,1
    CMP $0,$15,$18
    BN $0,L:5
L:8 SET $0,$17
    INCL $254,104
    POP 1,0
```

Example function with a variable number of arguments.

While it's certainly possible to implement special calling conventions for functions declared to take a variable number of parameters, like printf, there is no need to. The called stdarg function will arrange to map parameter registers to be accessible as a va_list (usually an array). The called function will have to push the parameters that were passed in registers onto the stack before processing them. There's an implicit assumption that stdarg functions don't have to be optimal in terms of speed. Also, this yields simple, easy-to-understand code.

How to get addresses into registers

Using GREG to get an address base

```
        GREG @
buffer1 BYTE 0,0,0,0,0,0,0,0
buffer2 BYTE 0,0,0,0,0,0,0,0
        ...
Main    LDA $0,buffer2
! (or individual GREG:s)
```

The constant-pool approach

```
buffer1 BYTE 0,0,0,0,0,0,0,0
buffer2 BYTE 0,0,0,0,0,0,0,0
            ...
pool    OCTA buffer1
        OCTA buffer2
            ...
Main    GETA $0,pool+8
        LDOU $0,$0,0
```

The GETA approach

```
buffer1 BYTE 0,0,0,0,0,0,0,0
buffer2 BYTE 0,0,0,0,0,0,0,0
        ...
Main    GETA $0,buffer2
        ! May expand to:
        ! SETL $0,buffer2&65535
        ! INCML $0,(buffer2>>16)&65535
        ! INCMH $0,(buffer2>>32)&65535
        ! INCH $0,(buffer2>>48)
```

Different approaches to getting an address into a register.

When accessing memory, the address has to get into a register somehow. A common convention is to allocate a register with GREG at the start of a number of memory variables, and use a base-plus-offset expression to translate the address into a register plus offset. That method is restricted to about two hundred such variables (or areas of variables, 256 bytes long) per program.

The ARM, Hitachi SH and others use "constant pools", where each far-away address is kept nearby the code; *its* address loaded with a GETA equivalent, then using a LDOU to load the real address. On the other hand, these memory accesses appear out of sequence compared to instruction fetches and the "real" memory accesses, a reason to avoid that solution, at least when modeling real hardware. Also, that would mean a GETA and LDOU with the address taking up an octabyte: disregarding possible sharing of addresses, this equals a four-instruction sequence, at least in size.

I chose to use GETA for address loading, leaving it to the assembler and linker to expand it when necessary. This is counter to the goal of using the same conventions as for manually written assembly code. Therefore I'll implement the

GREG/base-plus-offset approach as an option, default for standard applications. Anyway, it seems good to at least *optionally* be lean on GREG allocations, using the GETA approach.

Assembly label conventions

Some ports prepend symbols with underscores to disambiguate register names, but there's no need for that with MMIX. Some use dollar signs ($) or dots (.) to protect compiler-generated symbols; the MMIX port uses colon (:) at strategic places. (See the static chain example.)

GCC register allocation restrictions

A few properties and limitations of gcc affect the MMIX port badly.

```
extern long i;
long g (void);
long g2 (long);
long h (long);

long f (long p)
{
 long y = g();
 long x = h(y + p);
 i = x+y;
 return i;
}

long f2 (long p)
{
 long y = g2(i);
 long x = h(y + p);
 i = x+y;
 return i;
}
```

```
f    SET $15,$15
     GET $2,rJ
     PUSHJ $15,g
     SET $1,$15
     ADDU $0,$1,$0
     SET $16,$0
     PUSHJ $15,h
     PUT rJ,$2
     SET $0,$15
     ADDU $0,$0,$1
     GET $1,i
     STOU $0,$1,0
     POP 1,0

f2   GET $3,rJ
     GETA $2,i
     LDO $16,$2,0
     PUSHJ $15,g2
     SET $1,$15
     ADDU $0,$1,$0
     SET $16,$0
     PUSHJ $15,h
     PUT rJ,$3
     SET $0,$15
     ADDU $0,$0,$1
     STOU $0,$2,0
     POP 1,0
```

Example of fixed-set register allocation,
along related forcing of L=15.

As a rule, GCC considers the set of registers that can hold parameters and the set of return-value registers constant. Two other important sets, are the set of call-saved registers and call-clobbered registers. While GCC is written for these sets being constant, they should be chosen dynamically for best MMIX code (even per-call within a function), to e.g. avoid setting L higher than necessary. Unfortunately the possibility of making these properties dynamic, or one a function of the other, is limited at best. For the GCC MMIX port, these properties *are* currently (2001-10-10) constant: registers $0..$14 are call-saved, and $16..$31 are call-clobbered.

Changing these parts of gcc would be a major rewrite.

It is reasonably simple, that at a final pass "rename" registers, for example to close an unused gap between the end of the call-saved registers and the outgoing parameter registers. There's hope that the end result will be reasonably near an optimal register allocation, like the one a skilled human would do.

GCC global registers

After mentioning the register allocation limitations, I think I should cheer you up by mentioning that GCC has some provisions for allocating global variables in registers. It doesn't do it automatically, but it should be reasonably simple to interface global register definitions with GREG allocations. I have however not investigated this thoroughly.

An alternative ABI

GNU ABI difference:

$231 ... $246 Parameters and return value

```
long f2 (long p)
{
 long y = g2(i);
 long x = h(y + p);
 i = x+y;
 return i;
}
```

```
f2  GET $3,rJ
    GETA $2,i
    LDO $16,$2,0
    PUSHJ $15,g2
    SET $1,$15
    ADDU $0,$1,$0
    SET $16,$0
    PUSHJ $15,h
    PUT rJ,$3
    SET $0,$15
    ADDU $0,$0,$1
    STOU $0,$2,0
    POP 1,0
```

```
f2  GET $3,rJ
    SET $0,$231
    GETA $2,i
    LDO $231,$2,0
    PUSHJ $4,g2
    SET $1,$231
    ADDU $0,$1,$0
    SET $231,$0
    PUSHJ $4,h
    PUT rJ,$3
    ADDU $231,$231,$1
    STOU $231,$2,0
    POP 0,0
```

Example compiled for the MMIXware ABI along one compiled for the GNU ABI.

To simplify development of the port, I started with a traditional ABI, one where parameters are passed in call-clobbered global registers (same for called and calling function) and where the return value is returned in primarily the first parameter registers. I call this the GNU ABI.

At the moment I don't have any performance figures. Though in my experience, the number of register moves are kept to a minimum if incoming and outgoing parameter register and the primary return value register are the same. On the other hand MMIX instructions being three-operand lessens the importance of that.

Request for comments

- Structures passed and returned by value (in multiple registers)?
- ... with a size threshold?
- Parameters passed extended to octabyte?
- Is mmixal compatibility of interest?
- Thoughts about the GNU ABI?
- Your favorite issue here.

Call for opinions

Should structures be passed by-value in multiple registers?

If so, there should preferably be a size-threshold. What should be the structure-size threshold for that?

Who should extend passed integers, caller or called function? The 64-bit extension of passed integer types is mainly for ease of use; the programmer who interfaces assembly code to compiler-generated code does not have to remember the

exact type passed, "it's an integer, extend or truncate it and pass it in a register" is sufficient for most use. I haven't showed you actual examples of this, because there are some related bugs that cause the code to be quite ugly. That's why I've used "long" in most examples.

Is it useful that the generated code is as mmixal-compatible as possible, even if it results in non-optimal code?

What are your thoughts about the GNU ABI?

References

The sources for these references can be found at http://www-cs-staff.Stanford.EDU/~knuth/mmix-news.html and (probably, I have not checked) in the MMIX book.

1. mmix-doc.ps, section 29, page 23
2. mmixal-intro.ps, section 18, page 8
3. mmix-doc.ps, section 18, page 13

Footnotes

1. The static chain register is necessary when a function pointer to the nested function is passed to another function, the nested function accessing variables in the enclosed function. The passed function pointer for the nested function is actually points to another piece of code, a *trampoline*, where the static chain register is loaded. The trampoline then jumps to the nested function, as can be seen in the figure below:

```
extern void f0 (void (*)(void));
long f1 (void)
{
 long i = 0;

 void f2 (void)
 {
  i++;
 }

 f0 (f2);

 return i;
}
```

```
f1    SUBU $254,$254,56       f2::0 SUBU $252,$252,8
      GET $2,rJ                     LDO $0,$252,0
      SET $0,$254                   ADDU $0,$0,1
      GETA $1,LTRAMP:0              STOU $0,$252,0
      <copy trampoline template     POP 0,0
       to the stack>
      <setup the trampoline>  LTRAMP:0      GETA $255,1F
      ADDU $0,$254,40               LDOU $252,$255,0
      STCO 0,$0,0                   LDOU $255,$255,8
      SET $16,$254                  GO $255,$255,0
      PUSHJ $15,f0              1H  OCTA 0  ! Static chain.
      PUT rJ,$2                     OCTA 0  ! Address of the nested f2::0.
      LDO $0,$0,0
      INCL $254,56
      POP 1,0
```

A nested function using the static-chain register, $252.

2. In the "new" C99 standard, there can be names for types of specific sizes, like int64_t, accessible through the <stdint.h> header file. Mapping ordinary C types to specific sizes may therefore not strictly be necessary.

3. The type *long long* and its unsigned variant are standard types in the C99 standard.

GNU Free Documentation License

Version 1.1, March 2000

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all

copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- **A.** Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- **B.** List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).
- **C.** State on the Title page the name of the publisher of the Modified Version, as the publisher.
- **D.** Preserve all the copyright notices of the Document.
- **E.** Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- **F.** Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- **G.** Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- **H.** Include an unaltered copy of this License.
- **I.** Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- **J.** Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.