NUBI - A Revised ABI for the MIPS® Architecture

Document Number: MD00438
Revision 0.19
October 5, 2005

MIPS Technologies, Inc
1225 Charleston Road
Mountain View, CA 94043-1353

Template: B1.14, Built with tags: 2C MIPS32 ABI

Table of Contents

Figures

Tables

Introduction, scope and goals

The sections of this manual address what a useful ABI should encompass:

- Chapter 2 “Data Organization in a NUBI program” determines how you represent `long int`, `int`, `short`, `char` and pointer types.
- Chapter 3 “Register and calling conventions” describes registers reserved for particular purposes, stack conventions, who saves what, where arguments are found and where values are returned.
- Chapter 5 “Object code formats” describes the basic encompassing standard (ELF, usually) and its architecture-specific options. In particular this includes a set of “relocation types”: a relocation type is a recipe for modifying a program to adapt to the run-time locations of data or function calls. In ELF, each relocation type gets a code number.

  ELF object formats are (essentially) defined using C source code, and this manual references a particular version of source code from the GNU project.

  NUBI will use DWARF 2 debug information. The debug formats are not described here, and you should refer to [DWARF2] or the source code.
- Chapter 6 “Debug conventions”: to find its way through your program, the debugger relies on some conventions about stack arrangements, as well as explicit information passed through DWARF sections in the object file.
- Chapter 7 “Linux position-independent code”: Linux applications and shared libraries come as binaries which permit both code and data to be conveniently relocated in the virtual address space.
- Chapter 8 “Signals, signal frames and the “sigcontext” structure”: the old Unix signal mechanism lets an application choose to catch a signal event, providing a subroutine entry point for a *signal handler* (within the application program) which will be called by the kernel if the signal condition occurs.

  This is more complicated than it might sound. Once the signal handler is running, the kernel simply believes the application is running, and forgets any saved state from the deferred pre-signal activity. Information required to restart the interrupted part of the user program is held on the user-space stack, in a structure called a *signal frame*.

  The signal frame has to hold all the state (machine registers etc) for the interrupted user thread, but also has to fake a calling stack for the signal handler so that when it returns it will invoke the `sigreturn()` system call.

  The signal frame is a compatibility nightmare: it’s affected by the Linux kernel build, the C library, and by the CPUs repertoire of registers. But it’s definitely down to the ABI to define it!
- Chapter 9 “Thread-local storage”: much discussed recently. Multi-threaded applications and libraries alike are commonly requiring language support, allowing programmers to specify “per-thread” variables and have the system implement something convenient and efficient.
- *Instruction set issues* : there have been many variants of the MIPS instruction set in different CPUs prior to the definition of MIPS32 and MIPS64. MIPS Technologies would like to see maximum use of those standards (and in particular release 2 of the MIPS32 and MIPS64 architecture specifications): but we recognize that the wider community includes users of slightly different instruction sets, and NUBI must work there too if it is to become a viable standard. NUBI will recognize a range of base architectures, each a superset of the last.

  As is commonplace with modern architectures with embedded applications, there are also various optional extensions to the instruction set. For NUBI’s purposes we only have to concern ourselves with instructions which might be generated by a compiler (and are available in user mode - privileged instructions are hand-coded with the assembler). So the extensions which matter are the MIPS16e compact ISA, and the MIPS DSP ASE’s set of fractional, SIMD and multiply-accumulate variants.

  Compliant NUBI programs carry in the object file an identifier for the base instruction set plus any extensions which account for all the user-mode instructions in the program.

  The o32 ABI defines a fair number of object-code fields which encode ISA variants: but those fields filled up some time ago and their encoding is a cause of strife. NUBI judiciously extends the range of one field in the ELF header to define the base instruction set, but the use of instruction set extensions is in additional “option” object code sections, modelled on one of SGI’s ELF extensions.

It is our intention to leave a fair amount of background and motivational information in this specification to keep it comprehensible (and thus provide some defense against corruption of the specification when people evolve it without having understood why things were done a particular way).

Where detail would duplicate that already provided in source code - notably in header files which define the “ELF” data structures - we will not recite that source code in this document. We will reference specific public versions of source code when necessary.

1.1 Why does the MIPS Architecture need a new ABI?

An “ABI” (“Application Binary Interface”, though that hardly helps) is a set of rules governing compiled programs which - if followed - make the programs able to be linked together (for calling and to share data) and to be comprehensible to various useful bits of software - that includes debuggers, the Linux kernel, and run-time loaders.

Earlier MIPS ABIs were interpreted as machine-specific extensions to the cross-architecture [SVR4]; but the definition of OS services in a unix-like system now falls to POSIX and (specifically) Linux. This specification does not include the machine-independent parts of SVR4 ABI by reference or otherwise.

The existing MIPS ABIs were evolved substantially by Silicon Graphics Inc (“SGI”) for various versions of their “Irix” OS; they are fairly typical of ABIs for Linux and other sophisticated operating systems. At least to date most MIPS embedded systems have got by using a subset of SGI’s complicated ABI.

ABI History

The MIPS ABI took shape as a set of register usage and calling conventions established from the earliest days of MIPS CPUs. It picked up the “ABI” acronym and a defined binding to object code with the AT&T-inspired “Unix System V” document which is rooted with [SVR4].

That process had coalesced as early as 1990 into much of the “o32” ABI which is widely used today. By about 1994 the ABI was expanded to encompass position-independent code and the ELF object code “syntax”, and there have been no substantive and intentional changes since.

SGI pioneered 64-bit operating systems for MIPS in the early 1990s, and the o32 ABI was quite unsuitable for real 64-bit computing. SGI defined a 64-bit ABI called “n64” suitable for the largest applications; and then - belatedly realizing that n64’s 64-bit pointer and `long` types bloated programs and caused portability problems to many applications which didn’t need them - produced the very similar standard “n32”, which differs primarily in having 32-bit pointers.

From 1995 or so SGI used solely 64-bit-capable MIPS CPUs, so they had no need to revisit a 32-bit ABI. As a result the embedded MIPS world is still stuck on the 20-year-old o32 standard. A series of talks five years ago failed to come up with a replacement.

Meanwhile, the perceived deficiencies of o32 have led to the proliferation of variants and more narrowly-focussed alternatives, to the point where there are now as many as 15 incompatible MIPS ABIs.

It may yet prove the least worst decision for us all to continue to use o32 “forever”: but escaping from o32 could noticeably improve performance and ease various kinds of compatibility. So this is MIPS Technologies’ proposal to do so: but this won’t make sense unless we can take the community with us and end up with fewer ABIs - not just another family to add to the overlong list.

Specific Goals for NUBI

- *Replace multiple existing ABIs*: a new family should be “good enough” for all the applications we can make contact with, and a seed for consolidation on a single standard.
- *Make better use of registers*: o32’s limit of four argument registers causes unnecessary stack shuffling, and programs would run slightly better if we reserved more. Eight argument registers has been tried with success, notably by n64/n32.

  We will define more “saved” registers. Not only is this common for other architectures, but it is evident that the GNU C compiler would quite often generate better code with a few more of these.

- *Add a thread pointer*: a per-thread pointer in a reserved register makes for efficient thread-local storage.
- *Avoid unnecessary trouble with MIPS16e™*: the "MIPS16" compact-code standard[1] uses half-size (16-bit) instructions, and one of the trade-offs made means it only has first-class access to eight general-purpose registers. We want to ensure the ABI's register use does not cause avoidable pain to MIPS16e programs[2].
- *Better position-independent code ("PIC")*: all Linux shared libraries and applications are built PIC, so PIC efficiency matters. The general PIC code sequences for external data access and subroutine linkage are quite slow: we want to permit more optimizations of PIC calling and data referencing sequences.
- *Reduce 32-/64-bit incompatibility*: as systems grow to 64-bits we expect there to be some demand to interlink 32- and 64-bit code. This will never be seamless, but we believe it's worth making it practicable in controlled circumstances.

1.1.1 Introducing NUBI

We've chosen this name, for now. We intend to improve on o32 with something simpler, but which prefers being trouble-free for the MIPS programming community over ground-breaking innovation.

Unfortunately there can't be just one NUBI. We believe there are three basic choices (related to the hardware) which will create NUBI variants, but which it's essential to support:

- Endianness: big- and little-endian programs are wholly incompatible. However, with some care this document covers both. When we need to distinguish them we'll use a suffix "L" or "B" for little/big-endian.
- Use of 64-bit integer instructions: whether the software is restricted to a MIPS32 instruction set ("NUBI32"), or can use the whole of MIPS64 ("NUBI64"). If you only ever use MIPS32 instructions all general-purpose registers might as well be 32-bits wide.

  Even NUBI32 software is assumed to have access to double-precision floating point operations,
- Size of basic C types: we need to recognize a variant of NUBI64 with 64-bit pointers and `long`. I'm going to use NUBI64W ("W" for "wide") to denote this for now. Better suggestions welcomed.

  In the medium term we expect the "narrow" 64-bit ABI which uses 32-bit pointers and `long` type to be more popular.

That leads to the following list of six main variants: NUBI32L, NUBI32B, NUBI64L, NUBI64B, NUBI64WL and NUBI64WB. We'll use "NUBI32" to mean "NUBI32L and NUBI32B", and - at a pinch - NUBI–L to mean "NUBI32L and NUBI64L".

How's NUBI different from o32?

- *More argument registers*: eight instead of four.
- *Argument registers shared with return-value registers*: helps avoid having too many registers with pre-defined roles.
- *Adds a thread pointer*: for efficient TLS.
- *o32 was stack based, NUBI is register-based*: at bottom o32 used a stack-based calling convention, though it's disguised because the first, notional, 4×32-bit locations of the underlying stack argument structure are left unwritten, with the real data passed in four argument registers.

  At the time o32 was introduced C programs were frequently written without "function prototypes" to describe the types of the arguments expected by an external function. Without any prototypes calls to functions with non-standard arguments or (worse) with a variable number of arguments, like `printf()`, were difficult to get right. The underlying stack structure helped; a troubled function could save the four registers onto the stack to obtain a

[1] MIPS16e is the name for the instruction set provided by MIPS32 CPUs in 16-bit-instruction mode, whereas MIPS16 may be used for the name of the mode. MIPS16e adds a few instructions (which noticeably improve code compression) to the instruction set defined for earlier MIPS16 CPUs.

[2] Since the normal 32-bit MIPS register set treats pretty much all registers the same, the MIPS16e constraint should be unproblematic.

completely predictable memory structure for its arguments.

The stack-based structure also made it possible to pass data of derived types (structures, principally) by value. This has advantages: sometimes the data types you think are obscure turn out to be common.

In o32's generation, the critical example of this was the Fortran complex-number data type (which is a pair of floating point values).

However, in NUBI32's time the problem is more likely to be that we'd like to handle `long long` arguments and return values more efficiently. This might be worth a special case (which would cover complex and double-precision floating point numbers too).

- *o32 was irredeemably 32-bit, NUBI makes interworking possible* : a call between a NUBI64 program and a NUBI32 program will always need "gasket" code, but a gasket which is automatically produced would be a useful tool for complicated applications which are migrating from 32-bit to 64-bit, and where not all the components are recompiled together.

Data Organization in a NUBI program

This chapter describes how data should be stored in memory in the run-time image of a NUBI program.

Those of you familiar with traditional ABIs for the MIPS architecture will note that it is entirely compatible with “o32”.

The intent of this specification is to be restate the “o32” conventions, but the description has been completely re-written in the interests of clarity and brevity. If in that process we have unintentionally changed some obscure corner case in o32, we reserve the right to prefer a consistent interpretation of this text over o32.

For the purposes of this document memory is taken as an array of unsigned 8-bit quantities, whose index is the virtual address. For all compliant compilers the memory array corresponds to a C definition `unsigned char []`.

Like all modern computers, MIPS uses 2s-complement representation for signed integers - so in any data size “-1” is represented by binary all-ones. The overwhelming advantage of 2s-complement numbers is that the basic arithmetic operations (add, subtract, multiply, divide) have the same implementation for signed and unsigned data types[3].

C integer data types come in `signed` and `unsigned` versions, which are always the same size and alignment. A declaration without a signed/unsigned keyword is interpreted as signed[4].

In the medium term we expect to see some programs using fixed-point fractional data types, and perhaps even fixed-length vectors of small fixed-point types (which at the hardware level are carried in single registers). The fractional types will be declared by attaching an attribute to an integer declaration. A compiler may act on that declaration to give particular semantics to arithmetic operations: the ABI will only require that such data is managed according to the rules applicable to the underlying integer type.

Vector types could require an extension to NUBI. It’s our goal that we should leave it possible to do that in a way which preserves compatibility for software which doesn’t use vectors.

2.1 Sizes of basic types

Table 2-1 lists fundamental C data types and how they’re implemented for MIPS architecture CPUs. We’ll come back to the `long` and pointer types a bit later - their size changes according to whether you use the NUBI32 or NUBI64 ABI.

Table 2-1 Basic Data types and memory representations

C type	*MIPS asm name*	*size (bytes)*
`_Bool`	**`byte`**	**`1`**
`char`	**`byte`**	**`1`**
`short`	**`half`**	2
`int`	**`word`**	4
`long long`	**`dword`**	8
`float`	**`.single`**	4
`double`	**`.double`**	8
`long double`		16

[3] At least, until the result has greater precision than the operands.

[4] Some old code assumes that `char` is unsigned, and that behavior is available from GCC with a compiler flag. Strictly speaking, such behavior is non-compliant with NUBI.

2.2 Size of "long" and pointer types

Although these vary according to the type, in practice they're always the same as something else... For NUBI64W `long` is implemented just like the `long long` shown above, while for all other NUBI variants `long` is implemented just like an `int`.

It's good portability practice to have a pointer and `long` the same size, and all NUBI variants do so.

2.3 "long double" floating point types

We propose that NUBI follows in the footsteps of SGI's "n64" and "n32" standards and include an extended precision floating point number as a `long double`.

SGI implemented this as a pair of `double` variables whose sum is the number represented; one effectively holds the most significant bits of the number, the other the least significant bits. This cannot represent numbers outside the range of a regular `double`, but provides 50+ bits extra precision and is relatively efficient to compute with using a double-precision FPU. IBM's PPC64 ABI does the same.

We are inclined to use an SGI-compatible definition for NUBI; comments welcome.

2.4 Extended integer and "complex" types

GNU C has syntax for defining longer integer types, and modern C compilers are expected to support complex number types (corresponding to each floating point type). But for the purposes of this ABI such types will be dealt with exactly as if they consisted of a structure containing an array of scalars.

For extended integers the underlying scalar type will be the longest basic integer type which can be put together to form an extended type of the right bit-size.

We do not see any medium-term likelihood that any complex or extended types will get hardware support, and the largest alignment requirement for any object in NUBI shall be to an 8 byte unit.

2.5 Alignment requirements for basic types

Data types can only be directly handled by standard MIPS instructions if they are *naturally aligned*: that is, a 2-byte datum starts at an address which is even (zero modulo 2), a 4-byte datum starts at an address which is zero modulo 4, and an 8-byte datum starts at an address which is zero modulo 8[5].

Consequently, NUBI requires that all the primitive data types be naturally aligned.

2.6 Memory layout of basic types and how it changes with endianness

Table 2-2 shows how each basic type is laid out in our byte-addressed memory; the arrangement is different for big-endian and little-endian software.

In Table 2-2 the bit numbering is reversed between the two endianness layouts, which makes the bitwise depiction of the fields of floating point numbers easier to absorb (and prettier). It's a useful opportunity to give a warning: this neither adds nor subtracts any meaning. Bytes are indivisible 8-bit objects, and bit-numbers (where used in this specification) annotate the arithmetic significance of bits within an 8-, 16-, 32- or 64-bit integer type.

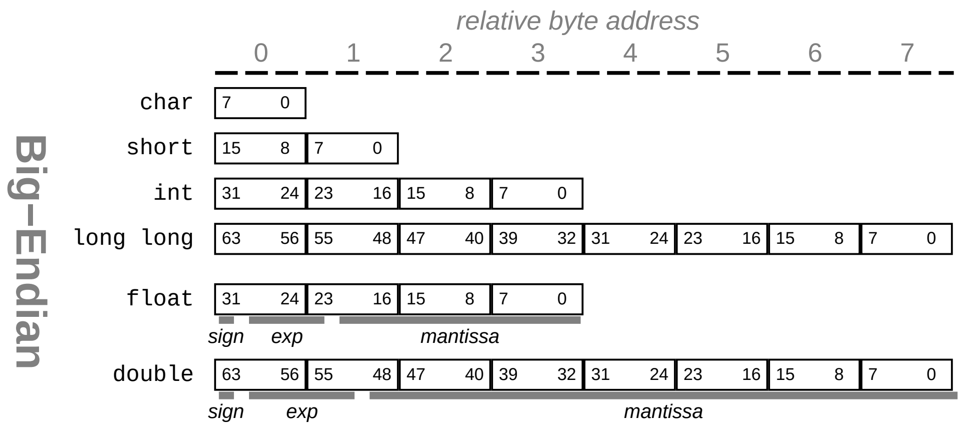
Each of these data types must be naturally aligned, as described above.

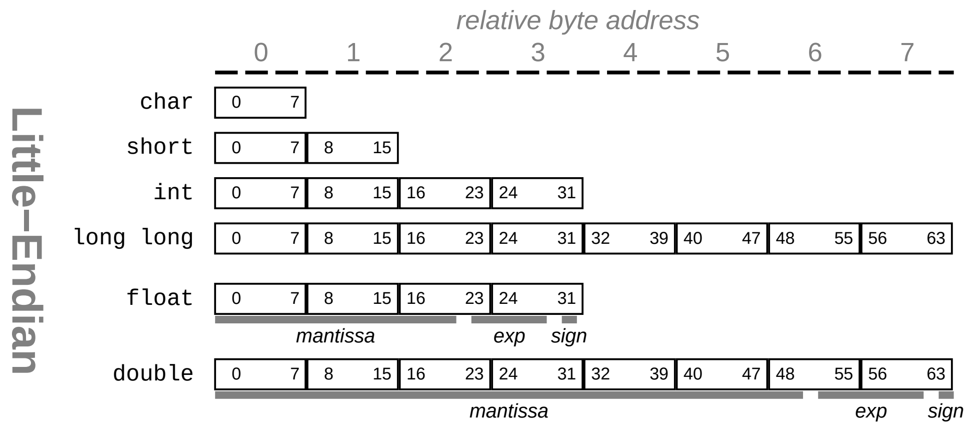
"Endianness" can be a troubling subject. If you are uneasy about it, read it up in [SMR].

[5] The 8-byte alignment is not a hardware requirement for a MIPS32 CPU (32-bit integer registers) with no floating point hardware - which is likely to be the commonest CPU in embedded systems.

But of course the 8-byte alignment is only produced when you define 8-byte types such as `long long` and `double`. We think that making some 32-bit CPUs incompatible isn't worth the small saving in data memory.

Table 2-2 C data types in memory


Big–Endian
relative byte address
0 1 2 3 4 5 6 7
char: 7 0
short: 15 8 | 7 0
int: 31 24 | 23 16 | 15 8 | 7 0
long long: 63 56 | 55 48 | 47 40 | 39 32 | 31 24 | 23 16 | 15 8 | 7 0
float: 31 24 | 23 16 | 15 8 | 7 0
sign exp mantissa
double: 63 56 | 55 48 | 47 40 | 39 32 | 31 24 | 23 16 | 15 8 | 7 0
sign exp mantissa


Little–Endian
relative byte address
0 1 2 3 4 5 6 7
char: 0 7
short: 0 7 | 8 15
int: 0 7 | 8 15 | 16 23 | 24 31
long long: 0 7 | 8 15 | 16 23 | 24 31 | 32 39 | 40 47 | 48 55 | 56 63
float: 0 7 | 8 15 | 16 23 | 24 31
mantissa exp sign
double: 0 7 | 8 15 | 16 23 | 24 31 | 32 39 | 40 47 | 48 55 | 56 63
mantissa exp sign

2.7 Memory layout and alignment of derived types

Derived types are built by concatenating simple types, but inserting unused (“padding”) bytes between items so as to respect the alignment rules[6].

It’s worth giving a couple of examples. Here’s the byte offsets of data items in a `struct mixed`:

```
struct mixed {
    char c;      /* byte 0 */
                 /* bytes 1-7 are ``padding'' */
    double d;    /* bytes 8-15 */
    short s;     /* bytes 16-17 */
};
```

[6] Some compiler systems provide mechanisms to alter the alignment rules for particular data definitions: GNU C supports the data declaration attribute `__attribute((align(x))` and the slightly more ANSI `#pragma pack(x)`. Both work to reduce the maximum padding by loosening alignment requirements down to 4 bytes, 2 bytes or none.

This allows you to model more possible data patterns with C data declarations, and the compiler will generate appropriate code (with some loss of efficiency) to handle the resulting unaligned basic data types. But such declarations are outside the scope of this document.

It's worth stressing that the byte offsets of the fields of constructed data types (*other than those using C bitfields*, see Section 2.7.1 "Bit fields in structures") are unaffected by endianness.

Constructed data types are aligned in memory to the largest alignment boundary required by a data type defined inside them. So a `struct mixed` will start on an 8-byte boundary; and that means that if you build an array of these structures you will need padding between each array element. C compilers provide for this by "tail padding" the structure to make it usable for an array, so `sizeof(struct mixed) == 24` and the structure should really be annotated:

```
struct mixed {
    char c;      /* byte 0 */
                 /* bytes 1-7 are ``padding'' */
    double d;    /* bytes 8-15 */
    short s;     /* bytes 16-17 */
                 /* bytes 18-23 are ``tail padding'' */
};
```

Just to remind you: the size (and consequently the alignment requirement) of pointer and `long` data types is 4 for NUBI32 and NUBI64, but 8 for NUBI64W.

2.7.1 Bit fields in structures

C allows you to define structures which pack several short "bit field" members into one or more locations of a standard integer type. This is a useful feature for emulation, hardware interfacing, and perhaps for defining dense data structures, but is fairly incomplete. Bitfield definitions are nominally CPU-dependent and substantially endianness-dependent.

One can, for example, define a data structure which permits access to the various fields of a MIPS single-precision floating point number:

```
#if BYTE_ORDER == BIG_ENDIAN

struct ifloat {
    unsigned int sign:1;
    unsigned int bexp:8;
    unsigned int mant:23;
};

#else /* little-endian */

struct ifloat {
    unsigned int mant:23;
    unsigned int bexp:8;
    unsigned int sign:1;
};

#endif
```

In this case (as you'd hope and expect) the three fields are packed into one 32-bit `int` storage unit. How do the two cases differ? Well, for both endianness the bitfields are allocated with the first-defined field occupying the lowest byte-addressed part of the `int`. For big-endian, that means the high-order bits are occupied first; for little-endian, it's the low-order bits.

Does this make sense? Certainly some; if you tried to implement bitfields in a less endianness-dependent way, then in the following example `struct fourbytes` would have a different memory layout from `struct fouroctets`- and that doesn't seem reasonable:

```
struct fourbytes {
  signed char a; signed char b; signed char c; signed char d;
}

struct fouroctets {
  int a:8; int b:8; int c:8; int d:8;
}
```

A bitfield can only be packed inside one storage unit of its defined type; if we try to define a structure for a MIPS double-precision floating point number, the mantissa field contains part of two 32-bit `int` storage units and can't be defined in one go. The best we can do in ANSI C is something like this:

```
struct ieee754dp_konst {
    unsigned    sign:1;
    unsigned    bexp:11;
    unsigned    manthi:20;  /* cannot get 52 bits into... */
    unsigned    mantlo:32;  /* .. a regular C bitfield    */
};
```

You're permitted to leave out the name of the field definition, so you don't have to invent names for fields which are just there for padding.

Bitfields of type other than a signed or unsigned `int` (or `_Bool`) are not mandated by most C standards. But many compilers will support bitfields in longer integer types. If we were confident that all the compilers we would ever need support an `unsigned long long` bit field, we could have defined the double-precision floating point structure without having to split the mantissa over two fields.

The full alignment rules for bit-fields are complicated:

- As we said above, a bit-field must reside entirely in its *storage unit* - which is the enclosing, properly aligned location of the bit-field's defined type.

  Thus a bit-field never crosses an alignment boundary of its defined type.
- Bit-fields can share their storage unit with other struct/union members, including members that are not bit-fields (to pack together, of course, the adjacent structure member would have to be of a smaller type).
- Structures inherit their own alignment requirement (recursively) from the alignment requirement of their most demanding field type. Named bit-fields will cause the structure to be aligned at least as well as the type requires.

  Unnamed fields - regardless of their defined type - only force the storage unit or overall structure alignment to that of the smallest integer type which can accommodate that many bits.
- You might want to be able to force subsequent structure members to occupy a new storage unit. In a NUBI-compliant compiler you can do that with an *unnamed zero-width* field. Zero-width fields are otherwise pointless, and named zero-width fields are illegal.

You now know everything you need to map C data declarations to memory the NUBI way.

2.8 Soft-float: floating point values in integer storage

Many CPUs oriented to the embedded market have no floating point hardware. When you compile C code using floating point for such a CPU you can leave it to the run-time system to catch the "coprocessor unusable" exceptions and emulate the floating point operations, but that's horribly inefficient.

Compilers will typically implement a "soft-float" option where floating point data types are implemented as if they were integers of the same size, and floating point operations implemented by a judicious mix of inline code and calls to a somewhat-invisible library.

If you want to be strict, you could see the soft-float option as yet another doubling of NUBI variants. But that would be counterproductive. Fortunately, the difference between regular and soft-float code can be confined to the compiler. So we rule that when you compile soft-float then for all ABI purposes `float` will be treated as an alias for `int`, and `double` an alias for `long long`.

Compilers should provide a flag in object files and in debug records which note the soft-float option; linkers should give a warning when mixing soft- and hard-float modules, but should not treat this as a fatal error which cannot be overriden.

Register and calling conventions

The MIPS architecture makes nearly all the registers the same; only `$0` (which is just an always-zero bit bucket) and `$31` (used implicitly for a return address by the **`jal`** instruction which has no encoding space to pick a different register) are different.

It's a nice paradox that confronted by such an orthogonal architecture, software tools end up laying down some pervasive conventions which almost no software can break. And one of the conventions governs the use of the registers; in principle this is just part of the ABI.

MIPS currently favor a re-work of register conventions which is not backward compatible. It seems to be too constraining to conserve arbitrary assembler code, and relatively easy to automate conversion of assembler source in the face of any change we choose to make.

To refresh your memory on the older ABIs, refer to Section 3.2 "Register usage in legacy (o32, n32, n64) ABIs". But for NUBI we propose to:

- *Combine return-value and argument registers*: this is done in other ABIs and there's no obvious problem with it - so it allows us to increase the number of re-assignable registers.
- *Demote the "assembler temporary" register*: compilers should not generate assembler code which implicitly uses registers, so should generate all their code under a ".noat". This (temporary) register should still be avoided when you're writing assembler code and want to use more complicated addressing modes in store instructions (etc).
- *More saved and less temporary registers*: four ex-temporary registers become "saved".
- *Define small-data-pointer as "saved"*: in particular the `gp` register used as the GOT pointer in PIC code or for "small-data" access in bare-iron code is redefined as "saved" - and in programs which are neither GOT nor use small data, it's free for reuse.

  In this case NUBI follows a change already made for n32/n64.
- *Add a thread pointer*: as widely agreed.

3.1 NUBI register convention

NUBI's register assignments are shown in Table 3-1.

3.1.1 Floating point register convention

MIPS CPUs which have FPU hardware have 32 floating point registers, whose assembler names are `$f0` - `$f31`. Even 32-bit MIPS CPUs support the 64-bit IEEE double-precision format.

The now mostly obsolete 32-bit MIPS I CPUs do arithmetic only in the 16 even-numbered[7] registers `$f0`-`$f30`. NUBI does not officially support them.

All modern MIPS CPUs have 32 64-bit, full double-precision, registers and can do arithmetic with any of them.

Table 3-2 shows NUBI's convention. Where it is harmless to do so, we've remained consistent with n64's naming and register use conventions: but NUBI has 12 temporary, 12 saved and 8 argument registers (n64 had 14, 8, and 8 respectively, with two more dedicated for return values).

Table 3-1 Register roles for NUBI

Reg No	*Hardware constraint*	*NUBI Name*	*Use*	*o32 Name*
`$0`	Always reads zero	`zero`		`zero`
`$1`		`AT/t3`	assembler temporary (additional temp in compiled code)	`AT`
`$2`	MIPS16 accessible	`t0/pf`	PIC-code function call address/temporary	`v0`
`$3`	MIPS16 accessible	`gp/s13†`	global data pointer/GOT pointer/saved register	`v1`
`$4-$7`	MIPS16 accessible	`a0-a3`	arguments/return values	`a0-a3`
`$8-$11`		`a4-a7`	more arguments	`t0-t3`
`$12-$15`		`s0-s3`	saved registers	`t4-t7`
`$16-$17`	MIPS16 accessible	`s4-s5`	saved registers	`s0-s1`
`$18-$23`		`s6-s11`	saved registers	`s2-s7`
`$24`	MIPS16 cond. code	`t1`	temporary	`t8`
`$25`		`t2`	temporary	`t9`
`$26-$27`		`k0-k1`	reserved for interrupt/trap handler	`k0-k1`
`$28`		`tp`	thread pointer	`gp`
`$29`	MIPS16 stack pointer	`sp`	stack pointer	`sp`
`$30`		`s12/fp`	saved register recycled as frame pointer if required	`s8/fp`
`$31`	link for jal	`ra`	Return address for subroutine	`ra`

Table 3-2 Floating point register usage conventions

Reg No	*NUBI Name*	*Use*	*n64 Name*
`$f0`	`ft10`	Temporary (not saved)	`fv0`
`$f1`	`ft0`	Temporary (not saved)	`ft0`
`$f2`	`ft11`	Temporary (not saved)	`fv1`
`$f3-$f11`	`ft1-9`	Temporary (not saved)	`ft1-9`
`$f12-$f19`	`fa0-fa7`	Arguments/return values	`fa0-fa7`
`$f20-$f23`	`fs8-fs11`	Values saved over function call	`ft10-ft13`
`$f24-$f31`	`fs0-fs7`	Values saved over function call	`fs0-fs7`

† Non-position-independent code does not need a GOT pointer, and the use of the ‘‘small data’’ region is optional for bare-iron software. Where neither is specified, this becomes an additional ‘‘saved’’ register.

3.2 Register usage in legacy (o32, n32, n64) ABIs

Table 3-3 shows how registers have been used to date.

Table 3-3 Legacy register conventions in o32, n32, n64

Register Nos	*name*	*use*		
`$0`	`zero`	always zero		
`$1`	`AT`	assembler temporary		
`$2-$3`	`v0-v1`	return value from function		
`$4-$7`	`a0-a3`	function arguments		
	o32		*n32/n64*	
	name	*use*	*name*	*use*
`$8-$11`	`t0-t3`	temporaries	`a4-a7`	more arguments
`$12-$15`	`t4-t7`	temporaries	`t0-t3`	temporaries (value must be assumed lost over subroutine call)
`$24-$25`	`t8-t9`	temporaries	`t8-t9`	temporaries (value must be assumed lost over subroutine call)
`$16-$23`	`s0-s7`	saved registers (value preserved over subroutine call)		
`$26-$27`	`k0-k1`	reserved for interrupt/trap handler		
`$28`	`gp`	global data pointer/GOT pointer		
`$29`	`sp`	stack pointer		
`$30`	`s8/fp`	stack frame pointer if required (additional saved register if not)		
`$31`	`ra`	Return address for subroutine		

3.3 NUBI calling convention

One of the most significant differences between contemporary ABIs is they way in which they pass arguments to functions. All agree that it's desirable to ensure that most arguments are passed in registers rather than on the stack. All - in the last resort, when arguments are very large or very numerous - resort to passing arguments in memory. However, there are two distinct styles of calling convention, with different characteristic advantages and problems.

Stack- and Register-orientated calling conventions

- *Stack-orientated*: all arguments are notionally laid out on the stack. It's almost as if you were just applying the data-storage rules described in the previous chapter, though separate arguments are subject to an additional alignment requirement, to ensure that each starts on a register-sized boundary.

  Then as many bytes of arguments as can be accommodated are sucked up into the registers available for argument passing, in order. Where the whole of a register-sized and aligned chunk contains a floating point value, it is passed in a floating point register.

  Advantages: a stack-orientated mechanism allows you to pass any data type (even a structure passed by value) in registers. While this is not terribly useful in generality there are cases - like software quad-precision, or complex numbers represented as pairs of floating point - which are important in particular application areas.

  There's a subdivision according to whether the nominal stack-space whose data is in registers should exist or not (it does in o32 and the 64-bit PowerPC specification, but does not exist in n64).

  Disadvantages: mapping derived types to the stack is relatively complicated and liable to corner-case incompatibilities. It also requires a lot of copying, which is sometimes unnecessary.

  We want to define sub-ABIs for 32-bit and 64-bit CPUs, and the way the "register size" unit determines the mapping of structure arguments creates troublesome incompatibilities.

- *Register-orientated*: all arguments are reduced to a form which will fit in a register, and passed in registers. When you run out of registers, you notionally put the argument into a register and save it to memory, then stack such register images.

  For structure arguments, the form which fits in a register is a pointer to a structure in memory.

  A naive register-orientated scheme requires that arguments which don't fit in a register be copied by the caller - C arguments passed by value should not be at risk of being modified by the callee. More sophisticated variants - which we'd choose - get the called function to make a copy of the argument, but only if it either writes or shares the address of that argument.

  Register-orientated schemes only allocate stack space when there are more arguments than will fit in registers.

  Advantages: it's simple to parse, for both machines and programmers. Moreover - particularly for the bulk of functions with less arguments than the number of argument registers - it minimizes the differences between the 32- and 64-bit versions.

  Disadvantages: `varargs` can become ugly. In it's purist form, even small derived types (and things like complex numbers are treated like derived types) are passed by an implicit pointer which forces data out of registers. That can cost efficiency in applications where such structure-by-value arguments are ubiquitous.

The best-known legacy MIPS ABIs (o32, n32 and n64) are all stack-orientated. But MIPS Technologies are inclined to make NUBI register-orientated. Your input is particularly welcomed on this issue.

3.4 NUBI calling convention

The NUBI calling convention is register-, not slot-based[8].

- *Arguments* : each argument is represented by a single register-sized value. Each of the first 8 arguments travels in the appropriate `a0-7` register (or `fa0-7` for scalar floating point arguments[9]). If there are more than eight arguments, further ones are formed *as if put in a register* and then saved on the stack into a 64-bit slot[10].

  We use floating point registers for double[11] and float arguments, and integers by value for all integer values which will fit in a register. Simple integer values are as if loaded into the register (ie they occupy the least-significant bits).

  Derived types (structures etc) and non-standard scalar types are passed in a register if and only if their memory-stored image is register-size aligned and fits into a register[12]. Derived types are mapped into a register by a register-size load (that's **`lw`** for NUBI32 and **`ld`** for NUBI64). Note that there is no guarantee that (even in rather simplest structures) a smaller-than-register integer field will end up conveniently positioned in the register.

  All other arguments are passed by reference. The callee must copy the argument if it writes it or takes its address.

  The `long long int` type is bigger than a register in NUBI32, and is then passed by reference. All bigger-than-register derived types are, too.

- *Return value* : an integer value or derived type which fits in a single register is returned in `a0`, while a floating point value which fits in a register is in `fa0`. All other return values are returned via a pointer specified by the caller as an implicit extra argument, passed *first*.

- *GOT pointer* : `gp` will be the GOT pointer for PIC code (which is Linux-compatible). This is not program-constant (different link units must have different GOTs, and different modules may have different GOTs). Functions changing its value should save its value on entry and restore it on return (ie `gp` is treated as "saved", as is already done by n32/n64).

  `tp` will be the thread pointer for thread-local storage. `tp` is expected to be set by the OS/library when the thread starts, and is read-only to normal compiled code. Code which does not use thread-local storage should not use this register at all.

---

[7] In this and in other matters it is influenced by the "EABI" proposals made by a group of GNU C workers some years ago. We've tried to ensure that NUBI is never arbitrarily different from EABI.

[8] This is different from PPC32 or EABI, which were willing to pass 16 arguments in registers if 8 of them happened to be floating point.

[9] Functions with more than 8 arguments are rare enough that we can afford to standardize on the big slots without wasting significant amounts of stack space.

[10] NUBI assumes that a compliant MIPS CPU implements a floating point unit which implements both `double` (IEEE-754 64-bit standard) and `float` (IEEE-754 32-bit standard) data types and operations, implement the MIPS architecture load/store double operations, and have 32 64-bit floating point registers.

Historic MIPS I CPUs had only 16 64-bit registers (formed by pairs of 32-bit registers) and did not support load/store double. NUBI doesn't provide for generating convenient floating point for those CPUs - though it does allow for soft float - see above.

[11] We are aware that pass-by-reference for `long long` - and perhaps for `long double` and `_complex double` too - may sacrifice too much efficiency. We'll do some research to estimate how bad the impact would be, with particular attention to the Linux kernel and libraries.

We will consider an alternative where a derived/extended type which consists of a pair of identical scalars would be passed "as if" it was two arguments, in two registers. If we do that we'd also have to ensure that NUBI64 "skips" an argument register opportunity after an argument which fits in a 64-bit register but requires a pair of 32-bit registers, so subsequent arguments would remain in sync with NUBI32's... this is nasty, and we don't want to do it unless the impact of not doing so is bad.

3.5 NUBI stack frame standards

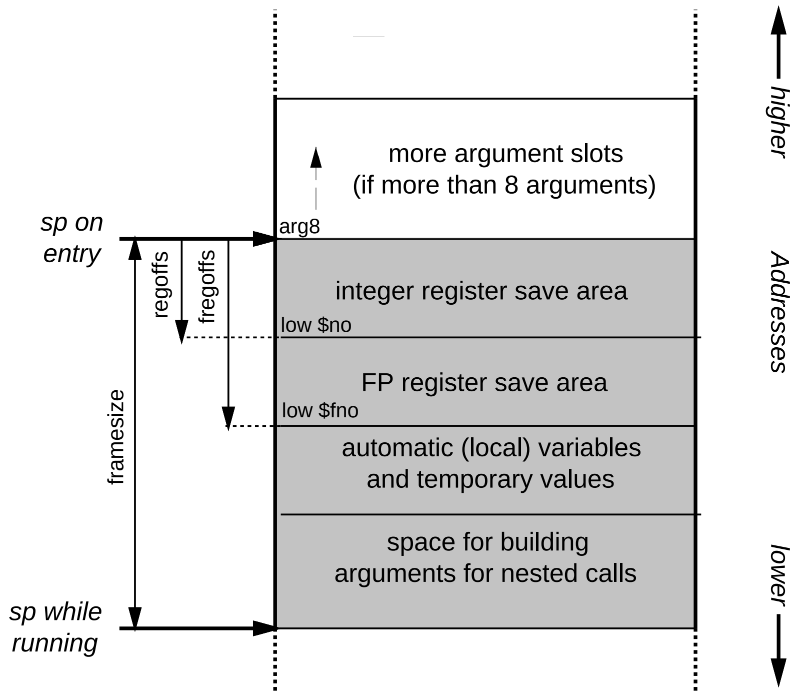
A NUBI function must maintain the stack pointer at an 8-byte-aligned location at or below the lowest stack location used to date by the function, and restore it immediately before return.

A function's stack use includes, of course, any argument slots required by calling a function with more than 8 arguments.

All further rules about stack management are not essential for successful intercalling, but debuggers and other useful software tools assume these conventions[13].

Here's a diagram:

Figure 3-1 Stack layout in NUBI



more argument slots
(if more than 8 arguments)
arg8
sp on entry
regoffs
fregoffs
integer register save area
low $no
FP register save area
low $fno
framesize
automatic (local) variables and temporary values
space for building arguments for nested calls
sp while running
higher
Addresses
lower

Notes on the stackframe diagram Figure 3-1

- *Argument slots*: needed only when there are more than 8 arguments: each is 64 bits (8 bytes) in extent.
- *Integer/FP register save area*: integer saved registers in NUBI32 occupy 32-bit (4 bytes); all other register-save slots are 64-bit (8 bytes). Registers are saved with the lowest-numbered registers (that's *physically* lowest numbered) at the lowest memory positions.
- *framesize, regoffs, fregoffs*: these are the arguments to an assembler directive which will leave information for the debugger about the shape of the stack.

[12] That is, these rules are made compulsory where they don't seem likely to cost anything, to avoid pointless diversity arising by chance.

Code Conventions for function stack management.

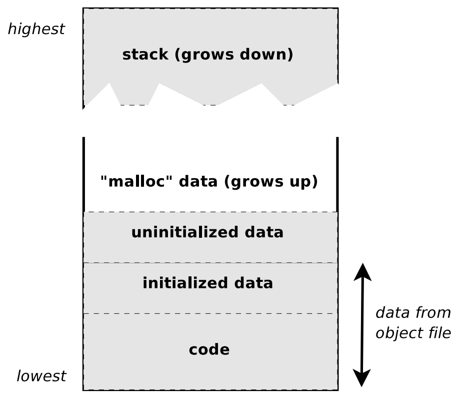
Commonly a NUBI function will adjust the stack pointer on entry by an amount known at compile time, and keep it there until it returns. The adjustment should be made prior to any branch or label at the start of the function.

Any function whose stack depth is unknown at compile time must maintain a frame pointer in the `fp` register.

Programs in memory

Running programs reside in computer memory. The ABI conventions don't determine the memory layout of the program, but the two certainly interact, and some of the practical complications of the ABI are there to allow users to build the memory map their application needs. So it's helpful to draw some pictures: we'll start with the simplest possible one in Figure 4-1.

Figure 4-1 The simplest memory map



Complications ensue if - as is often the case with bare-iron software - you are using "unmapped" memory regions like kseg0/kseg1, and need to fit your software onto the hardware's memory map. Sometimes code and read-only data must go in ROM, but writable data and stack must of course be in RAM. We're not going into that here.

GP-relative data

Loading or storing a piece of data at an address known at link time requires a two-instruction sequence on MIPS (typically an **`addiu`** followed by a **`lw/sw`**.) But a common MIPS trick with bare-iron programs is to try to bring together the most commonly used data in a program (up to 64Kbytes of it), then to reserve the `gp` register as a pointer into the middle of this area. That allows data items to be loaded or stored with a single instruction. It's hard to figure out what is really the most commonly-used data in advance, so we pick all data items up to a specific size. With `gcc` the ``` ``-S8'' ``` option directs all data items whose size is 8 bytes or less to a separate "small data" region. Well, in fact there are two regions, one for initialized data (".sdata") and one for uninitialized (".sbss") - but they are eventually placed next to each other in the memory map.

This trick does not work for Linux applications built of multiple shared objects: it depends on the linker being able to figure out all the offsets from `gp` at link-time, which is not possible with dynamic loading. For this reason the PIC standard recycles the `gp` register to become the global offset table pointer.

Object code formats

Object code is used to represent NUBI binary programs in computer files, and all NUBI object code uses a variant of the ELF format.

ELF was defined as part of the "System V Application Binary Interface" standard [SVR4] and the supplement which described the MIPS ("o32") variant is [SVR4_MIPS]. This specification is compliant with base ELF, though not with o32: it aims to provide a reference which will replace the latter book[14]. Some generic ELF information is repeated here, as an aid to navigation.

Chapter 2 "Data Organization in a NUBI program" of this ABI defines a precise mapping from a C data structure to an array of bytes, so - within the context of the ABI - binary file data contents are unambiguously defined by C structures. That sounds suspiciously circular, but it works.

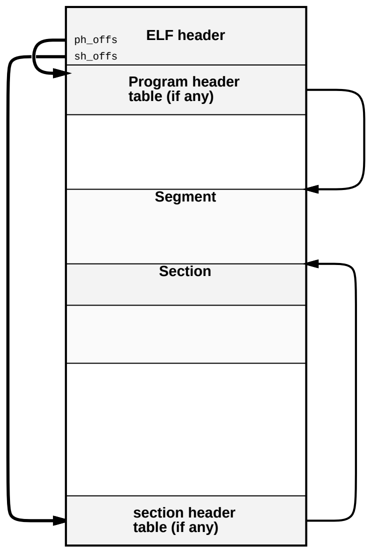
The software we will regard as a reference implementation for NUBI (and whose C header files will be authoritative) will be the GNU "binutils" distribution. The principle header files concerned will be:

File	*What's defined here*
`src/include/elf/external.`	Describes the structure fields of the header and key tables. It's "external" because it's describing the contents of the files, not the internal data structures of the library which manipulates object file data.
`src/include/elf/common.h`	Describes standard values for fields.
`src/include/elf/mips.h`	Field values which are specific to the MIPS architecture

5.1 ELF object file - components

An ELF file might contains several piece as sketched out in Figure 5-1.

Figure 5-1 What's in an ELF file?



[13] The most important audience for this chapter are programmers constructing compliant object code producers and consumers. None of this audience needs to have machine-independent ELF features reprised.

That is, it’s a lump of binary which starts with an *ELF header*; that header tells you the file position of the *Program header table*, which locates and describes a number of chunks called *segments* and the *Section header table*, which locates and describes a number of *sections*. Segments and sections are often made of the same chunks of data: quite often a segment spans a number of different sections.

By original design, segments are for program loaders; sections are for intermediate build tools which write object files. Otherwise they have similar structures and many things which are “sections” early in the build process get packed into segments of the same name and very similar structure in the loadable file. When ELF was created the section/segment distinction was relatively straightforward: but the shared-library system of a modern OS like Linux defers the binding of names to real subroutines to program load time or even run time. As a result the loader does pretty much everything a linker needs to do...

An ELF file starts with its “ELF header”. Typically the program header table follows immediately after, with the section header table at the end, and the section/segment contents in between. However, that’s not required by the standard and should not be assumed by software which reads ELF.

A good way to get used to the contents of ELF files is to dump them out using the Linux (also available on other Unix-like OS’) utility `readelf`.

5.1.1 The ELF file header

An object file starts with an ELF file header. For NUBI32 that's an `Elf32_External_Ehdr` type, whereas for NUBI64 it's an `Elf64_External_Ehdr`. Both are defined in `external.h` like this:

```
typedef struct {
  unsigned char e_ident[16];       /* ELF "magic number" */
  unsigned char e_type[2];         /* Identifies object file type */
  unsigned char e_machine[2];      /* Specifies required architecture */
  unsigned char e_version[4];      /* Identifies object file version */
  unsigned char e_entry[4];        /* Entry point virtual address */
  unsigned char e_phoff[4];        /* Program header table file offset */
  unsigned char e_shoff[4];        /* Section header table file offset */
  unsigned char e_flags[4];        /* Processor-specific flags */
  unsigned char e_ehsize[2];       /* ELF header size in bytes */
  unsigned char e_phentsize[2];    /* Program header table entry size */
  unsigned char e_phnum[2];        /* Program header table entry count */
  unsigned char e_shentsize[2];    /* Section header table entry size */
  unsigned char e_shnum[2];        /* Section header table entry count */
  unsigned char e_shstrndx[2];     /* Section header string table index */
} Elf32_External_Ehdr;

typedef struct {
  unsigned char e_ident[16];       /* ELF "magic number" */
  unsigned char e_type[2];         /* Identifies object file type */
  unsigned char e_machine[2];      /* Specifies required architecture */
  unsigned char e_version[4];      /* Identifies object file version */
  unsigned char e_entry[8];        /* Entry point virtual address */
  unsigned char e_phoff[8];        /* Program header table file offset */
  unsigned char e_shoff[8];        /* Section header table file offset */
  unsigned char e_flags[4];        /* Processor-specific flags */
  unsigned char e_ehsize[2];       /* ELF header size in bytes */
  unsigned char e_phentsize[2];    /* Program header table entry size */
  unsigned char e_phnum[2];        /* Program header table entry count */
  unsigned char e_shentsize[2];    /* Section header table entry size */
  unsigned char e_shnum[2];        /* Section header table entry count */
  unsigned char e_shstrndx[2];     /* Section header string table index */
} Elf64_External_Ehdr;
```

With the exception of `e_ident`, these fields are interpreted as integer values of various sizes. In the original SVR4 documents, the ELF header was defined using integer types - which are unambiguous, because their mapping to byte-addressable memory was defined in Chapter 2 "Data Organization in a NUBI program" above. But the GNU binutils project has found it more convenient to use char arrays, which makes the field sizes explicit. That somewhat obfuscates the structure definitions and the code which interprets ELF fields, but means you can make tools which work reliably in a cross-compilation environment (where the compilation host may have different endianness, basic-type size and alignment requirements).

Field values relevant to NUBI[15] are shown in Table 5-1:

[14] We have omitted a number of historical field values associated with old MIPS ABIs. The accumulation of these values had become part of the problem rather than the solution. We believe that the NUBI object format will provide all the information linkers and loaders need, but some of the information which was formerly (and unsatisfactorily) squeezed into the header will appear in special "option" sections instead.

Table 5-1 ELF file header field values in NUBI

Field	*Legal value*	*Meaning*
`e_ident[0]`	`ELFMAG0 = 0x7F`	"Magic number" which provides informal evidence that this is, indeed, an ELF file.
`e_ident[1]`	`ELFMAG1 = 'E'`	
`e_ident[2]`	`ELFMAG2 = 'L'`	
`e_ident[3]`	`ELFMAG3 = 'F'`	
`e_ident[4]`	`ELFCLASS32 = 1`	32-bit ELF file. This value is critical, since it determines the size and interpretation of other fields, even in the ELF header itself.
	`ELFCLASS64 = 2`	64-bit ELF file
`e_ident[5]`	`ELFDATA2LSB = 1`	Little-endian. This not only flags the endianness of the program encoded here, but the encoding of all the bigger-than-byte fields in the ELF file.
	`ELFDATA2MSB = 2`	Big-endian
`e_ident[6]`	`1`	A version number for the ELF header data structure. Always 1 so far, and unlikely ever to change.
`e_ident[7]`	`ELFOSABI_LINUX = 3`	Indicates a Linux-compatible ABI (or OS using compatible object file)
	`ELFOSABI_STANDALONE = 255`	Use this for code built non-PIC for "bare-iron" or single-address space RTOS.
`e_ident[8]`		"ABI version" used to flag incompatible updates to an evolving ABI. This does not seem to have been used for architecture-dependent information.
`e_type`	`ET_REL = 1`	Relocatable file (typically an intermediate `.o` file). NUBI programs should generate this value, but should not rely on it for much.
	`ET_EXEC = 2`	Executable file
	`ET_DYN = 3`	Shared object (dynamic library) file
	`ET_CORE = 4`	"Core file" synthesized by an OS to capture the state of a program for later debug.
`e_machine`	`EM_MIPS = 8`	Originally narrowly defined for "MIPS I" big-endian. But usage has evolved to mean that this denotes any MIPS family architecture, and NUBI will stay with that tradition.
`e_version`	`EV_CURRENT = 1`	ELF version 1 - there has never been another, and probably never will be.
`e_flags`		Packed field of MIPS-specific information, defined in Figure 5-2 below. NUBI will retain compatibility with the most-used fields, but will encode information about the program encoding in object code sections rather than trying to squeeze it into this overused field.

A note on "e_flags" and program/CPU compatibility

Earlier MIPS ABIs used many more `e_flag` field values than are described here. Some of these fields are no use any more: sometimes that's because all possible values are assigned but more are needed, but sometimes it just isn't clear why any ELF user would want to know.

NUBI object files will contain only the sub-fields shown in Figure 5-2 below.

Figure 5-2 Fields in the ELF "e_flags" entry for MIPS

31 28	27	26 16	15 12	11 10	9	8	7 6	5	4 2	1	0
ARCH	ARCH_ASE_M16	×	ABI	×	FP64	32BIT_MODE	×	ABI2	×	PIC	×

Where:

- `ARCH`: (`e_flags` & `EF_MIPS_ARCH`) - this 4-bit value can be:

ARCH	*Meaning*
`E_MIPS_ARCH_1 = 0`	Program uses MIPS I instructions...
`E_MIPS_ARCH_2 = 1`	MIPS II†
`E_MIPS_ARCH_3 = 2`	MIPS III
`E_MIPS_ARCH_4 = 3`	MIPS IV
`E_MIPS_ARCH_32 = 5`	MIPS32
`E_MIPS_ARCH_64 = 6`	MIPS64
`E_MIPS_ARCH_32R2 = 7`	MIPS32 release 2
`E_MIPS_ARCH_64R2 = 8`	MIPS64 release 2

  Compliant NUBI programs may take any value from 1 through 8. NUBI does not aspire to supporting MIPS I hardware (though MIPS I integer code is unproblematic, the distinct early floating-point dialect will cause trouble).

  The MIPS32 specification includes a description of the CPU's OS-only privileged resources. But the privileged instruction set is outside of the scope of this specification.

  So for the purposes of this specification the following relationships can be guaranteed:

Table 5-2 Relationship between mainstream MIPS architecture revisions

$A \rightarrow B$ - A is a subset of B

$$\begin{tikzcd} A \arrow[d] \\ B \end{tikzcd}$$

A is the 32-bit-register only subset of B

$$\begin{tikzcd} \text{MIPS I} \arrow[r] & \text{MIPS II} \arrow[r] \arrow[d] & {} & \text{MIPS32} \arrow[r] \arrow[d] & \text{MIPS32R2} \arrow[d] \\ & \text{MIPS III} \arrow[r] & \text{MIPS IV} \arrow[r] & \text{MIPS64} \arrow[r] & \text{MIPS64R2} \end{tikzcd}$$

  There was a specification for a "MIPS V" revision of the ISA. But it's really an "instruction set extension" (or "ASE") which adds paired-single floating point SIMD instructions to MIPS IV. A NUBI program's use of ASE instructions is communicated through special object code "option" section entries, as described in Section 5.1.5 "Notes sections for compiler/instruction set options" below.

- `ARCH_ASE_M16`: (`e_flags` & `EF_MIPS_ARCH_ASE_M16`) - 1 if code uses special half-sized instructions (as in MIPS16 and MIPS16e). NUBI programs should set this, but should read and provide authoritative information will be in object code "option" sections.
- `FP64`: (`e_flags` & `EF_MIPS_FP64`) - 1 if code compiled assuming full 64-bit FPU. Always set for NUBI (even NUBI32 insists that FP hardware, if present, be 64-bit).

[15] MIPS II is interpreted strictly as the subset of MIPS III which excludes all 64-bit integer operations.

- ABI: (e_flags & EF_MIPS_ABI) distinguishes between MIPS ABI families, and is extended for NUBI:

EF_MIPS_ABI	*Meaning*
EF_MIPS_ABI_O32 = 1	MIPS o32
EF_MIPS_ABI_O64 = 2	o32 informally extended to programs using 64-bit instructions and registers. Has seen some use for 64-bit Linux kernels.

EF_MIPS_ABI_EABI32 = 3
EF_MIPS_ABI_EABI64 = 4
	1995 consortium "EABI" spec, in 32- and 64-bit flavors. Some use for "bare iron" programming, but they more often stick to o32.

EF_MIPS_ABI_NUBI = 5†
EF_MIPS_ABI_NUBIW = 6†
	NUBI has just two values, the second reserved for the "wide" 64-bit variant. You should use e_ident entries to distinguish the 32-/64-bit and endianness variants.

- 32BIT_MODE: (e_flags & EF_MIPS_32BITMODE) - 1 when code assumes 32-bit registers only. Always set for NUBI32, but NUBI-compliant software should not rely on it.
- ABI2: (e_flags & EF_MIPS_ABI2) - with a 32-bit file (ELFCLASS32), a 1 denotes the n32 ABI. But it's always zero for NUBI.
- PIC: (e_flags & EF_MIPS_PIC) - 1 if file contains PIC ("position-independent") code.

5.1.2 The ELF Program header table

You can find out whether there's a program header table present from the e_phnum field of the ELF header: it will be zero if there's no program header table. Be careful: the integer value is encoded according to the endianness of the ELF file and aligned according to the MIPS rules described above - either or both of those might not be directly compatible with the host you're using to read the file. But if present, the program header table starts at the file offset you can read as the e_phoff field of the ELF header.

† This value is a new one and is subject to change until we get agreement from appropriate maintainers.

The program header is an array of `Elf32_External_Phdr` types, each defined like this:

```
typedef struct {
  unsigned char p_type[4];      /* Identifies program segment type */
  unsigned char p_offset[4];    /* Segment file offset */
  unsigned char p_vaddr[4];     /* Segment virtual address */
  unsigned char p_paddr[4];     /* Segment physical address */
  unsigned char p_filesz[4];    /* Segment size in file */
  unsigned char p_memsz[4];     /* Segment size in memory */
  unsigned char p_flags[4];     /* Segment flags */
  unsigned char p_align[4];     /* Segment alignment, file & memory */
} Elf32_External_Phdr;

typedef struct {
  unsigned char p_type[4];      /* Identifies program segment type */
  unsigned char p_flags[4];     /* Segment flags */
  unsigned char p_offset[8];    /* Segment file offset */
  unsigned char p_vaddr[8];     /* Segment virtual address */
  unsigned char p_paddr[8];     /* Segment physical address */
  unsigned char p_filesz[8];    /* Segment size in file */
  unsigned char p_memsz[8];     /* Segment size in memory */
  unsigned char p_align[8];     /* Segment alignment, file & memory */
} Elf64_External_Phdr;
```

The simplest loadable ELF program can have just one program header entry.

The only fields with encoded values are `p_type`, `p_flags` and `p_align`, and all are described here.

- `p_type`: describes the use and nature of this segment:

p_type	*Meaning*
`PT_NULL = 0`	unused entry
`PT_LOAD = 1`	Loadable program segment
`PT_DYNAMIC = 2`	Information required to glue “dynamic libraries” to this program at run-time.
`PT_INTERP = 3`	Names the program which should be run to “interpret” this program. For dynamically-linked Linux applications this will generally be the dynamic loader `/lib/ld-linux.so`.
`PT_NOTE = 4`	Auxiliary information
`PT_SHLIB = 5`	Reserved, unspecified semantics
`PT_PHDR = 6`	Entry for the header table itself
`PT_TLS = 7`	A per-thread “thread local storage” segment.

`PT_GNU_EH_FRAME`
`PT_GNU_STACK`
`PT_GNU_RELRO`
	GNU tool standard section (frame unwind information)
	Stack flags
	Section will be read-only after relocation

`PT_MIPS_REGINFO = 0x70000000`	obsolete
`PT_MIPS_RTPROC = 0x70000001`	obsolete
`PT_MIPS_OPTIONS = 0x70000002`	type for MIPS “.MIPS.options” sections. NUBI may use those for instruction set and other build variants if it turns out we can’t use an ELF-generic section type such as “PT_NOTE”.

- `p_flags`: has three bit fields (any combination of these values is possible):

`PF_X = 1`	Segment is executable
`PF_W = 2`	Segment is writable
`PF_R = 4`	Segment is readable

- `p_align`: defines the alignment required. It may be zero or 1 (indicating no alignment requirement, the segment may be loaded into memory anyhow), or a positive power of two: so a value of 16 implies that the base of this segment should end up in a memory address which is zero modulo 16.

5.1.3 The ELF Section header table

The section header table is an array of `Elf32_External_Shdr` types, each defined like this:

```
typedef struct {
  unsigned char sh_name[4];         /* Section name, index in string tbl */
  unsigned char sh_type[4];         /* Type of section */
  unsigned char sh_flags[4];        /* Miscellaneous section attributes */
  unsigned char sh_addr[4];         /* Section virtual addr at execution */
  unsigned char sh_offset[4];       /* Section file offset */
  unsigned char sh_size[4];         /* Size of section in bytes */
  unsigned char sh_link[4];         /* Index of another section */
  unsigned char sh_info[4];         /* Additional section information */
  unsigned char sh_addralign[4];    /* Section alignment */
  unsigned char sh_entsize[4];      /* Entry size if section holds table */
} Elf32_External_Shdr;

typedef struct {
  unsigned char sh_name[4];         /* Section name, index in string tbl */
  unsigned char sh_type[4];         /* Type of section */
  unsigned char sh_flags[8];        /* Miscellaneous section attributes */
  unsigned char sh_addr[8];         /* Section virtual addr at execution */
  unsigned char sh_offset[8];       /* Section file offset */
  unsigned char sh_size[8];         /* Size of section in bytes */
  unsigned char sh_link[4];         /* Index of another section */
  unsigned char sh_info[4];         /* Additional section information */
  unsigned char sh_addralign[8];    /* Section alignment */
  unsigned char sh_entsize[8];      /* Entry size if section holds table */
} Elf64_External_Shdr;
```

We'll look particularly at `sh_type` and `sh_flags`: everything else is simpler.

But first a general comment: an object code section has a type value, a section name, and flag bits as described below. In older MIPS ABIs it's often the case that any one out of the three suffices to tell you what the section is. Such redundancy has its costs, and we'll aim for NUBI to use only a small number of architecture-specific section types and flag bits.

For whatever reason, MIPS architecture programmers have invented more architecture-specific ELF section type values than all other architectures put together. We would like to rein this in...

- `sh_type`: one of many section types, summarized in Table 5-3.

Table 5-3 Object code section types ("sh_type" values)

sh_type	*Meaning*
`SHT_NULL = 0`	Unused slot in section header table
`SHT_PROGBITS = 1`	Program specific (private) data
`SHT_SYMTAB = 2`	Link editing symbol table
`SHT_STRTAB = 3`	A string table. Every relocatable file has at least one of these, with at least the names of the sections in it.
`SHT_RELA = 4`	Relocation entries with addends
`SHT_HASH = 5`	A symbol hash table
`SHT_DYNAMIC = 6`	Information for dynamic linking
`SHT_NOTE = 7`	Information that marks file
`SHT_NOBITS = 8`	Section occupies no space in file (eg BSS definition)
`SHT_REL = 9`	Relocation entries without addends, nut used in NUBI
`SHT_SHLIB = 10`	Reserved, unspecified semantics
`SHT_DYNSYM = 11`	Dynamic linking symbol table

`SHT_INIT_ARRAY = 14`
`SHT_FINI_ARRAY = 15`
	Array of pointers to `_init`/`_fini` functions to be called (respectively) immediately after load and just before exiting. Could be used to implement C++ constructors/destructors, for example.

`SHT_PREINIT_ARRAY = 16`	Same as `SHT_INIT_ARRAY` functions, but these ones are called first.
`SHT_GROUP = 17`	Section contains a section group
`SHT_SYMTAB_SHNDX = 18`	Indices for `SHN_XINDEX` entries.
`SHT_GNU_LIBLIST = 0x6ffffff7`	List of prelink dependencies

`SHT_GNU_verdef = 0x6ffffffd`
`SHT_GNU_verneed = 0x6ffffffe`
`SHT_GNU_versym = 0x6fffffff`
	Information used to implement a version compatibility system for shared libraries. The sections list (respectively) versions defined/exported by this file, versions needed by file, and version tags for our symbols.

History (particularly SGI history) has left a vast number of MIPS processor-specific section types. Most architectures get by with one or two: NUBI should follow that example. We're not yet sure which of these are so obsolete that we need not even document them...	
`SHT_MIPS_OPTIONS = 0x7000000d`	SGI-defined field used in n32/n64 for carrying fields about build tools, build options, OS and hardware specialization in this file. NUBI may use it to identify the CPU instruction set features used in the program, if something more generic proves unsuitable. See Section 5.1.5 "Notes sections for compiler/instruction set options".
`SHT_MIPS_DWARF = 0x7000001e`	MIPS DWARF debugging section. Probably relates to SGI-unique DWARF extensions for 64-bit. Not required for NUBI.

`sh_flags` is a bit-field, as shown in Table 5-4.

Table 5-4 Fields in the section header "sh_flags" word

31 11	10	9	8	7	6	5	4	3	2	1	0
See Table 5-5	TLS	GROUP	OS_NONCONFORMING	LINK_ORDER	INFO_LINK	STRINGS	MERGE	×	EXECINSTR	ALLOC	WRITE

And in that field:

- `TLS`: (`sh_flags & SHF_TLS`) - "thread-local storage", a section containing data to be replicated per-thread.
- `GROUP`: (`sh_flags & SHF_GROUP`) - member of a section group.
- `OS_NONCONFORMING`: (`sh_flags & SHF_OS_NONCONFORMING`) - OS-specific processing required.
- `LINK_ORDER`: (`sh_flags & SHF_LINK_ORDER`) - preserve order of sections marked thus while linking.
- `INFO_LINK`: (`sh_flags & SHF_INFO_LINK`) - indicates that `sh_info` holds section header table index.
- `STRINGS`: (`sh_flags & SHF_STRINGS`) - section is full of null-terminated strings.
- `MERGE`: (`sh_flags & SHF_MERGE`) - a data section where identical values may be merged with those in other sections of the same name. Used to avoid duplicating common strings and constants.
- `EXECINSTR`: (`sh_flags & SHF_EXECINSTR`) - contains executable machine instructions.
- `ALLOC`: (`sh_flags & SHF_ALLOC`) - will occupy memory during execution.
- `WRITE`: (`sh_flags & SHF_WRITE`) - will be marked writable for execution.

Table 5-5 Machine/OS-dependent fields in the section headder "sh_flags" word

31	30	29	28	27	26	25	24	23 20	19 11	10 0
STRING	ADDR	MERGE	GPREL	NOSTRIP	LOCAL	NAMES	NODUPES	×	×	See Table 5-4

- `STRING`: (`sh_flags & SHF_MIPS_STRING`) -
- `ADDR`: (`sh_flags & SHF_MIPS_ADDR`) -
- `MERGE`: (`sh_flags & SHF_MIPS_MERGE`) -
- `GPREL`: (`sh_flags & SHF_MIPS_GPREL`) - contains data items which will be in the "global data area" (and accessed at an offset from the `gp` register). See Chapter 4 "Programs in memory".
- `NOSTRIP`: (`sh_flags & SHF_MIPS_NOSTRIP`) - may not be discarded by `strip` utility (which aims to minimize the size of an object file while maintaining its usefulness).
- `LOCAL`: (`sh_flags & SHF_MIPS_LOCAL`) - thread-local (now probably obsoleted by TLS, see above).
- `NAMES`: (`sh_flags & SHF_MIPS_NAMES`) - "linker should generate implicit weak names for this section."
- `NODUPES`: (`sh_flags & SHF_MIPS_NODUPES`) - not used in NUBI.

Then the other non-obvious fields in a section header entry are

- `sh_name`: is an index into the "string table" section which locates a name for this section. To avoid a circular dependency, the string table can be reached without a name lookup: it's entry number `e_shstrndx` in the section header table.
- `sh_link`: is used as a section header index for another section, for entry types which require that.
- `sh_info`: used by specific types as required.
- `sh_entsize`: if the section holds a table of objects defined by structure types, this is the size of each entry.

Section names used in NUBI

ELF in general (and NUBI in particular) allows for arbitrarily-named sections. But some named sections will be important to particular tools, some are commonplace and worth a couple of lines explaining, and some are simply worth avoiding because of their historical connotations.

For now, Table 5-6 is a comprehensive list of all MIPS section names which have been used in o32, n32 and n64.

Table 5-6 Section names used in MIPS ABIs

Name	*Type*	*Default Flags*
.bss	`SHT_NOBITS`	SHF_ALLOC + SHF_WRITE
.comment	`SHT_PROGBITS`	none (by default)
.data	`SHT_PROGBITS`	SHF_ALLOC + SHF_WRITE
.dynamic	`SHT_DYNAMIC`	SHF_ALLOC
.dynstr	`SHT_STRTAB`	SHF_ALLOC
.dynsym	`SHT_DYNSYM`	SHF_ALLOC
.fini	`SHT_PROGBITS`	SHF_ALLOC + SHF_EXECINSTR
.got	`SHT_PROGBITS`	SHF_ALLOC + SHF_WRITE + SHF_MIPS_GPREL
.hash	`SHT_HASH`	SHF_ALLOC
.init	`SHT_PROGBITS`	SHF_ALLOC + SHF_EXECINSTR
.interp	`SHT_PROGBITS`	SHF_ALLOC
.line	`SHT_PROGBITS`	SHF_ALLOC
.note	`SHT_NOTE`	none (by default)
.relname	`SHT_REL`	none (by default), see [ABI32]
.relaname	`SHT_RELA`	none (by default), see [ABI32]
.rodata	`SHT_PROGBITS`	SHF_ALLOC
.strtab	`SHT_STRTAB`	none
.symtab	`SHT_SYMTAB`	none
.text	`SHT_PROGBITS`	SHF_ALLOC + SHF_EXECINSTR
.MIPS.compact_rel	`SHT_MIPS_COMPACT`	none
.conflict	`SHT_MIPS_CONFLICT`	SHF_ALLOC
.MIPS.contentname	`SHT_MIPS_CONTENT`	SHF_ALLOC+ SHF_MIPS_NOSTRIP
.dynamic	`SHT_DYNAMIC`	SHF_ALLOC
.MIPS.eventsname	`SHT_MIPS_EVENTS`	SHF_ALLOC+SHF_MIPS_NOSTRIP
.gptabname	`SHT_MIPS_GPTAB`	none
.MIPS.interfaces	`SHT_MIPS_IFACE`	SHF_ALLOC+SHF_MIPS_NOSTRIP
.MIPS.lbss	`SHT_NOBITS`	SHF_ALLOC + SHF_WRITE
.MIPS.ldata	`SHT_PROGBITS`	
.liblist	`SHT_MIPS_LIBLIST`	SHF_ALLOC
.lit4	`SHT_PROGBITS`	SHF_ALLOC + SHF_MIPS_GPREL
.lit8	`SHT_PROGBITS`	SHF_ALLOC + SHF_MIPS_GPREL
.MIPS.lit16	`SHT_PROGBITS`	SHF_ALLOC + SHF_MIPS_GPREL
.MIPS.options	`SHT_MIPS_OPTIONS`	SHF_ALLOC+SHF_MIPS_NOSTRIP
.msym	`SHT_MIPS_MSYM`	SHF_ALLOC
.rel.dyn	`SHT_REL`	SHF_ALLOC
.reldname	`SHT_RELA`	SHF_ALLOC
.sbss	`SHT_NOBITS`	SHF_ALLOC + SHF_MIPS_GPREL + SHF_WRITE
.sdata	`SHT_PROGBITS`	SHF_ALLOC + SHF_MIPS_GPREL + SHF_WRITE
.srdata	`SHT_PROGBITS`	SHF_ALLOC + SHF_MIPS_GPREL
.MIPS.symlib	`SHT_MIPS_SYMBOL_LIB`	SHF_ALLOC
.MIPS.translate	`SHT_PROGBITS`	SHF_MIPS_NOSTRIP + SHF_ALLOC (non-shared only)
.debug_abbrev	`SHT_MIPS_DWARF`	none (generic DWARF section)

Table 5-6 Section names used in MIPS ABIs

Name	*Type*	*Default Flags*
.debug_aranges	`SHT_MIPS_DWARF`	none (generic DWARF section)
.debug_frame	`SHT_MIPS_DWARF`	SHF_MIPS_NOSTRIP (generic DWARF section)
.debug_funcnames	`SHT_MIPS_DWARF`	none (generic DWARF section)
.debug_info	`SHT_MIPS_DWARF`	none (generic DWARF section)
.debug_line	`SHT_MIPS_DWARF`	none (generic DWARF section)
.debug_loc	`SHT_MIPS_DWARF`	none (generic DWARF section)
.debug_pubnames	`SHT_MIPS_DWARF`	none (generic DWARF section)
.debug_str	`SHT_MIPS_DWARF`	none (generic DWARF section)
.debug_typenames	`SHT_MIPS_DWARF`	none (MIPS DWARF section)
.debug_varnames	`SHT_MIPS_DWARF`	none (MIPS DWARF section)
.debug_weaknames	`SHT_MIPS_DWARF`	none (MIPS DWARF section)

5.1.4 Relocations and relocation types

Linkers coalesce object code sections into larger units, eventually producing a loadable program. The main transformation performed by linkers and loaders on ultimately-loadable data is in adjusting data fields representing addresses of symbols to match the address of the symbol in the final program.

A relocation section is an array of relocation operation structures of type `Elf32_External_Rela`, and looks like this:

```
typedef struct {
  unsigned char r_offset[4];    /* Location at which to apply the action */
  unsigned char r_info[4];      /* index and type of relocation */
  unsigned char r_addend[4];    /* Constant addend used to compute value */
} Elf32_External_Rela;

typedef struct {
  unsigned char r_offset[8];    /* Location at which to apply the action */
  unsigned char r_info[8];      /* index and type of relocation */
  unsigned char r_addend[8];    /* Constant addend used to compute value */
} Elf64_External_Rela;
```

A relocation section may contain multiple relocations, but all for one particular code/data section.

The `r_info` field packs together an index into the symbol table and a relocation type (an opcode, effectively). For NUBI64 both index and relocation type are 32-bit integers; for NUBI32 the index is a 24-bit integer and the type an 8-bit integer. In either case the index occupies the high-numbered bits of the implied integer (so the order of index and type as seen in memory addresses is endianness-dependent)[16].

All MIPS relocation entries have `addend` - a supplementary integer argument - whether it's used or not[17].

The address in the section which wants to be fixed up might appear as a simple aligned pointer; but it might also be a field inside an instruction. Worse, with a RISC instruction set like MIPS, such an address may be effectively split into two or more parts in different fields of separate instructions (not necessarily adjacent). So each relocation type is associated with both a particular sort of calculation to be done, and one or more bitfields in the data selected by `r_offset` which are patched with the computed value.

The relocation entries required for MIPS16 compressed code are especially complex.

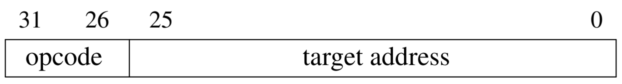
[16] Some of you may remember the n64 relocation record, which could define up to *three* relocation types, implying that up to three operations should be carried out in turn.

[17] ELF permits the use of a variant relocation entry with the addend omitted. NUBI does not use such entries.

Relocation targets are instruction formats

The data items you're relocating might be pointers in data space, which are unproblematic. It gets more interesting when a value which represents an address (so the linker may want to fix up) must be insinuated inside the appropriate bit-field of a binary instruction code. MIPS instruction codes are described in great detail in [MIPS32] or [MIPS64], but it probably helps to give a very quick sketch here. In non-PIC code most function calls use the **`jal`** instruction, whose instruction code is shown in Figure 5-3:

Figure 5-3 Fields of a MIPS "jal" instruction



The "target address" is obtained by truncating the run-time address of the target function to 28 bits (the top four bits of the target address will be copied from the instruction's own location, so the range of **`jal`** is limited to the 2^{28}-byte (512MB) chunk of memory which encloses the instruction). Then you drop the two low bits - MIPS instructions are always 32-bit objects, 4-byte aligned, so the low address bits are redundant.

Other MIPS instructions have signed or unsigned constants in the low 16 bits.

The only other really confusing thing happens when an address must be loaded in two steps, because the largest "immediate" constant available for MIPS instructions is 16 bits. So you can get sequences like:

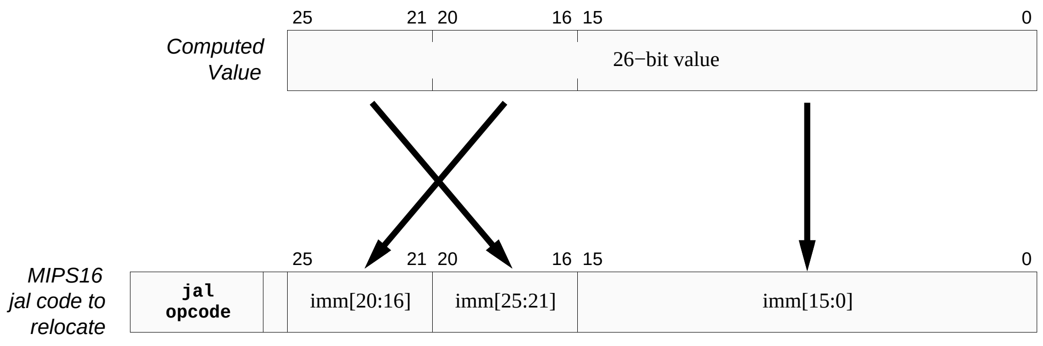
```
lui t1, high_part_of_addr
lw  t1, low_part_of_addr(t1)
```

`lui` is a MIPS instruction which shifts a 16-bit unsigned pointer left by 16 places before loading it into a register. The slight wrinkle here is that you can't just use the 16 high bits for the "high part of address": the **`lw`** instruction considers its 16-bit offset to be signed, so if your 32-bit address value has bit 15 set (about a 50:50 chance, really) the low 16 bits will look like a negative value, and you'll use the wrong address. So when bit 15 of the address value is set, you need to add one to the high part to compensate...

MIPS16 instruction formats are more challenging

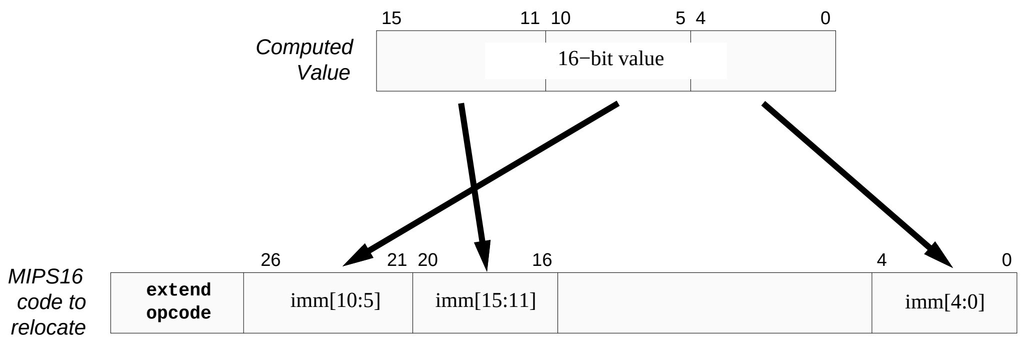
Where things get more difficult is with the compact-code MIPS16 instruction set. Since MIPS16 works by providing a subset instruction set in 16-bit codes, it's not surprising that MIPS16 instructions have some trouble encoding large constants. So MIPS16 includes some 32-bit "extended" instructions, and the linker knows about just two instruction formats: the MIPS16 version of **`jal`** and the "extended immediate" format. In both cases the bits of the value you want to patch in must be rather unpleasantly mangled: see Figure 5-4 for the first case:

Figure 5-4 Mangling address fields to relocate MIPS16 "jal"



And see Figure 5-5 for the second:

Figure 5-5 Mangling address fields to relocate MIPS16 extended-immediate insn



With those to refer to, I hope the relocation calculations will make a bit more sense.

Relocation types table

Let's see how it's going to work. We'll use a C-like notation, with `*r` representing a whole word we're fixing up. In a ghastly abuse of notation, we'll assume that `*r[25:0]` represents the target bitfield consisting of the low-order 26 bits of the word (which happens to be what we'll need to fix up for MIPS **`jal`** function-call instructions), and so on. We could do something more like real C by defining a union of bitfield structures, but then it would be endianness-dependent and perhaps harder to read.

Unless specified otherwise relocations are valid for both external and local symbols.

The pseudocode uses:

- `symbol`: the value of the symbol indexed by the high order bits of `r_info`.
- `addend`: value found in `r_addend`.
- `base_address`: an offset applied to everything in this object file (normally used for dynamic linking);
- `IsLocal`, `IsExternal`: a symbol is "local" when it's defined in the same object file as it's used in, and is marked as such by a flag in the symbol table. Symbols which aren't local are external: the `IsExternal` test sometimes shows that a relocation calculation is to be done differently when the symbol value and the data you're fixing up came from different files;
- `got()`: the GOT entry allocated specifically for whatever is in the parentheses;
- `got_page()`: the GOT entry allocated for something else but which can be used to compute the address for this symbol;
- `got_offset()`: the offset from the GOT entry allocated for the expression to the value of the expression,
- `IsGOT`: this relocation's symbol has its own GOT entry (only in PIC code, and then some variables may piggyback on another GOT entry);
- `gp`: the value of the global pointer after relocation is complete. It's a pointer for use when reading the GOT, but it doesn't necessarily point to the base of the GOT; MIPS load/store instructions have a signed offset, so the maximum range for `gp`-relative accesses is achieved by putting the pointer in the "middle" of the GOT;
- `_gp`: the object files evolving value of the global pointer as it was before relocation - it's stored in the symbol table under the name "_gp";
- `pc`: the address of the place being relocated;
- `scp`: the address of the section containing the symbol referred to by the relocation;

Relocation overflow checks

Sometimes a tool optimistically assumes that a location will be reachable with a 16-bit offset, but while doing relocations further down the line it turns out that it wasn't. This might happen for bare-iron code using the "GP-relative" code trick, or when PIC code overflows the larges manageable GOT size. When a tool calculates the relocation for such code, it should check for overflow and report it as an error. The following relocation types do require an overflow check¨

```
R_MIPS_16
R_MIPS_ADD_IMMEDIATE
R_MIPS_CALL16
R_MIPS_GOT16
R_MIPS_GOT_DISP
R_MIPS_GOT_OFST
R_MIPS_GOT_PAGE
R_MIPS_GPREL16
R_MIPS_LITERAL
R_MIPS_PC16
R_MIPS_REL16
```

Now the list itself:

Table 5-7 Relocation type calculations for NUBI

Relocation Type	*Description and Calculation*
`R_MIPS_32`	Initialize 32-bit pointers. `R_MIPS_REL32` used for the dynamic symbol table. `*r[31:0] = symbol + addend;`
`R_MIPS_REL32`	`*r[31:0] = (symbol + addend - base_address);`
`R_MIPS_26`	Fixing up **`jal`** targets. `*r[25:0]= (symbol + addend) >> 2;`
`R_MIPS_HI16`	Loading an address split into low and high halves, which will be added. `*r[15:0] = (symbol + addend + 0x8000) >> 16;`

`R_MIPS_LO16`	`*r[15:0] = (IsLocal && IsGOT) ?`
	`(got_offset(symbol + addend) - gp):`
	`(symbol + addend);`

`R_MIPS_GPREL32`	Generates a 32-bit offset from GP to a data value. May be used with giant GOT tables. `*r[31:0] = symbol + addend + _gp - gp;`
`R_MIPS_INSERT_A` `R_MIPS_INSERT_B`	*r[31:0] = addend;
`R_MIPS_16`	For an "immediate" form instruction, emitted by the **`%half()`** GAS operator. `*r[15:0] = symbol + (int16_t)addend;`

`R_MIPS_GPREL16`	Used for GP-relative load/stores accessing a GP-relative "small" program data sections (not relevant to PIC code). `R_MIPS_LITERAL` is the same, but used when the target is in one of the `.lit` sections used to hold constants.
	`*r[15:0] = IsExternal ?`
	`(symbol + (int16_t)addend - gp):`
	`(symbol + (int16_t)addend + _gp - gp);`

`R_MIPS_LO16` for GOT local symbols seems to be inconsistent in this respect. We ought to check this isn't a BFD bug.

Table 5-7 Relocation type calculations for NUBI

Relocation Type	*Description and Calculation*
`R_MIPS_LITERAL`	`*r[15:0] = symbol + (int16_t)addend + _gp - gp;`

`R_MIPS_GOT16`	Used to access the GOT in PIC code
	`*r[15:0] = IsExternal ?`
	`(&got(symbol + addend) - gp):`
	`(&got_page(symbol + addend) - gp);`

`R_MIPS_PC16`	Intended to fixup address field in branch instructions with a 16-bit signed offset. Not really useful without a shift-right - likely to be replaced by a different type before being used. `*r[15:0] = symbol + (int16_t)addend - pc;`
`R_MIPS_CALL16`	Like `R_MIPS_GOT16` (though pairing with `R_MIPS_LO16` is not permitted). Used for function calls, to permit lazy binding, where the GOT entry is initialized to the address of a stub. `*r[15:0] = &got(symbol) - gp;`
`R_MIPS_SHIFT5` `R_MIPS_SHIFT6`	Not used for NUBI
`R_MIPS_64`	For 64-bit pointers `*r[63:0] = (uint64_t)(symbol + addend);`
`R_MIPS_GOT_DISP`	Sets the displacement field for a GP-relative load in PIC code which fetches a pointer from the GOT. `*r[15:0] = &got(symbol + addend) - gp;`
`R_MIPS_GOT_PAGE`	used to access the GOT in PIC code, typically with **`lui`**. Absolute, 16-bit, bits `*r[15:0] = &got_page(symbol + addend) - gp;`
`R_MIPS_GOT_OFST`	`*r[15:0] = got_offset(symbol + addend) - gp;`

`R_MIPS_GOT_HI16`	`*r[15:0] =`
	`(&got(symbol + addend) - gp + 0x8000) >> 16;`

`R_MIPS_GOT_LO16`	`*r[15:0] = &got(symbol + addend) - gp;`
`R_MIPS_SUB`	`r[63:0] = (uint64_t)(symbol - addend);`
`R_MIPS_DELETE`	`/* nothing */`

`R_MIPS_HIGHER`
 	
`R_MIPS_HIGHEST`
	`*r[15:0] =`
	`(symbol + addend + 0x80008000) >> 32;`
	`*r[15:0] =`
	`(symbol + addend + 0x800080008000) >> 48)`

`R_MIPS_CALL_HI16`
`R_MIPS_CALL_LO16`
	`*r[15:0] = (&got(symbol) - gp + 0x8000) >> 16);`
	`*r[15:0] = &got(symbol) - gp;`

`R_MIPS_SCN_DISP`	`*r[31:0]= symbol + addend - scp;`

`R_MIPS_REL16`
`R_MIPS_ADD`
`_IMMEDIATE`
	`*(uint16_t *)r |= symbol + addend; /* ?? */`
	`*r[15:0] = symbol + (int16_t)addend;`

`R_MIPS_PJUMP`	`*r[31:0] = replacement;`
`R_MIPS_RELGOT`	`*r[31:0] = symbol + addend - base_address;`
`R_MIPS_JALR`	`*r[31:0] = replacement;`

Table 5-7 Relocation type calculations for NUBI

Relocation Type	*Description and Calculation*

`R_MIPS16_26`	Used to fix up the MIPS16 **`jal`** instruction, fields as shown in Figure 5-4 above.
	Watch out for alignment; double-length MIPS16 instructions don't necessarily start on 4-byte boundaries, so you can't write the code just as shown...
	`int val;`
	`val = symbol + addend;`
	`*r[15:0] = val;`
	`*r[20:16] = val >> 21; /* selects val[25:21] */`
	`*r[25:21] = val >> 16; /* selects val[20:16] */`

`R_MIPS16_GPREL`	Fix up a 16-bit offset in a MIPS16 extended GP-relative load/store. Requires bit-rearrangement as shown in Figure 5-5 above.
	`int val;`
	`val = IsExternal ?`
	`       (symbol + (int16_t)addend - gp + _gp):`
	`       (symbol + (int16_t)addend - gp);`
	*r[4:0] = val;
	r[20:16] = val >> 11; / selects val[15:11] */
	r[26:21] = val >> 5; / selects val[10:5] */

`R_MIPS16_GOT16`	`For planned MIPS16 PIC code, not used yet.`
	`int val;`
	`val = IsExternal ?`
	`     (&got(symbol + addend) - gp):`
	`     (&got_page(symbol + addend) - gp);`
	`*r[4:0] = val;`
	`*r[20:16] = val >> 11; /* selects val[15:11] */`
	`*r[26:21] = val >> 5; /* selects val[10:5] */`

`R_MIPS16_CALL16`	`For planned MIPS16 PIC code, not used yet.`
	`int val;`
	`val = &got_page(symbol) - gp;`
	`*r[4:0] = val;`
	`*r[20:16] = val >> 11;`
	`*r[26:21] = val >> 5;`

`R_MIPS16_HI16`	`int val;`
	`val = (symbol + addend) >> 16);`
	`*r[4:0] = val;`
	`*r[20:16] = val >> 11; /* selects val[15:11] */`
	`*r[26:21] = val >> 5; /* selects val[10:5] */`

`R_MIPS16_LO16`	`int val;`
	`val = IsGOTLocal ?`
	`     (got_offset(symbol + addend) - gp):`
	`     (symbol + addend);`
	`*r[4:0] = val;`
	`*r[20:16] = val >> 11; /* selects val[15:11] */`
	`*r[26:21] = val >> 5; /* selects val[10:5] */`

`R_MIPS_PC32`	`*r[31:0] = symbol + addend - pc;`
`R_MIPS_` `GNU_REL16_S2`	`*r[15:0] = (symbol + addend - pc) >> 2);`

Table 5-7 Relocation type calculations for NUBI

Relocation Type	*Description and Calculation*
`R_MIPS_GNU_VTINHERIT`	`/* nothing */`
`R_MIPS_GNU_VTENTRY`	`/* nothing */`

Weak symbols in PIC/dynamic-loader systems

What do we need to say about how weak symbols work?

5.1.5 Notes sections for compiler/instruction set options

We propose to introduce some "note" sections[18] which will remember some of the characteristics of the code (as known to the compiler, presumably).

We have two uses for this:

- To mark the use of optional instruction set extensions ("ASE"s).

  There are a number of mainstream MIPS instruction sets (each offering backward compatibility to its predecessors), but the use of those is recorded through the `E_MIPS_ARCH` field of the ELF header `e_flags` word, as described in the notes to Figure 5-2 above.

  But a MIPS instruction set may also be extended (more or less orthogonally) with one or more "ASE"s (for "Application-Specific instruction set Extensions"). We propose that these be recorded in these notes sections: more on this below.
- To mark the use of pervasive and likely-to-be-incompatible compile options, such as the `-msoft-float` flag to the compiler. These will be stored as a string; any GNU compiler option which may lead to compatibility problems (and is not selecting an instruction set or ASE) should be stored as the string "gcc-<option>" - so soft-float should be marked by the string "gcc-msoft-float".

MIPS ASEs to be recorded in object files

These are the main extensions requiring support at present:

- *Floating point extension*: adds all the user-level MIPS III 64-bit floating point operations. When combined with MIPS64 or MIPS32 Release 2, this includes all the floating point instructions defined in those specifications.
- *Paired-single floating point*: the intersection of the capabilities of the MIPS Technologies 20Kf core and the instructions defined by MIPS V.
- *MIPS-3D*: a small further extension to the paired-single instruction set promulgated by MIPS Technologies. For the purposes of NUBI, this also adds any MIPS64 paired-single operations which are not in MIPS V.
- *MIPS16e*: the compressed 16-bit opcode choice. To be precise, this is the MIPS16e ASE defined by MIPS Technologies, which was a superset of original MIPS16. As far as we are aware, there are no CPUs which combine our base instruction set with the original, subset, MIPS16.
- *SmartMIPS*: a small instruction set extension for low-end encryption.
- *MIPS DSP ASE*: the recently-announced MIPS technologies extension.
- *MDMX*: a "media-orientated" instruction set running from floating point registers. MDMX was defined many years ago, and is extant in two forms: a subset added to MIPS IV for the NEC/Sandcraft Vr54xx, and MIPS Technologies' specification for an ASE relative to MIPS64, as implemented by Broadcom/Sibyte.

[18] If this turns out to conflict with some other use of note sections, we could fall back to SGI's MIPS-specific "options" sections.

There are already other instruction set extensions defined and in use: but an extension like multi-threading is currently effectively a kernel-only one[19]. These will multiply in future, so we propose to reserve a total of 16 possible ASE locations, which leaves nine spare positions for future ASEs.

Each of these will be marked by a separate value; 0 for no support, and then increasing numbers for any extension to the ASE. If any ASE evolves into a non-backward-compatible form, it should be recorded as if a novel ASE. That gives us:

Table 5-8 ASE numbers and values for NUBI notes sections

ASE no/ASE		*Rev*	*On top of*	*Comments*
0	Floating point instructions	1	MIPS II/III	As defined for MIPS III
		2	MIPS64R1	As defined for MIPS64
		3	MIPS32R2/MIPS64R2	
1	MIPS16e	1	MIPS32/MIPS64	
2	Paired-single floating point	1	MIPS IV	"MIPS V"
		2	MIPS64	See MIPS64 manual
3	MIPS-3D	1	MIPS64/MIPS32R2	
4	SmartMIPS	1	MIPS32	
5	MIPS DSP	1	MIPS32R2/MIPS64R2	
6	MDMX	1	MIPS III	As implemented in NEC Vr54xx
		2	MIPS64/MIPS32R2	As defined by MIPS Technologies

Interpreting instruction set revisions and extensions

- *Compilers*: should offer flags to control the generation of code which select one of the revisions and one or more of the ASEs. Only the most inappropriate combinations should elicit error messages (eg MIPS-3D without floating point).

  However, the compilation system should only generate object code markers for instruction sets which are actually used. If command line options to the compiler say "MIPS32R2" but only original MIPS32 instructions are produced, then the object code should report what is present, not what was requested.

- *Assemblers*: should do what they're told, assembling any non-ambiguous instruction regardless of its instruction set provenance. Assemblers must know about the instruction set groups in order that they can generate the appropriate object sections describing the instruction set revision and the set of extensions exploited. Assemblers may provide command line options which explicitly outlaw certain revisions and extensions: only when provided with such options should they flag an instruction from some outlawed extension/revision as an error.

- *Linkers*: should quietly join up programs, maintaining the revision information at the high-water mark of constituent modules and keeping a superset list of extensions.

- *Operating systems*: an OS should query its own CPU and establish its willingness to handle various revisions and extensions (these may exceed the raw ability of the CPU if the OS is prepared to emulate instructions). This information should be available through some mechanism to user-level programs[20]

  The Linux library system already has - on some architectures - machinery which can be used to select one of a number of possible CPU-dependent sub-libraries, to make sure you have optimally-tuned versions of critical low-level functions.

[19] That's not necessarily true in the long run. MIPS MT defines user-level instructions which can be used - in theory - to create threads without OS intervention. When (and if) there's an OS which exploits that, NUBI would need an additional ASE identifier.

[20] In the case of Linux, probably through the "auxiliary environment" passed to `main()` through the system loader.

The loader may refuse to run programs which exceed the capability of the CPU/OS.

> Implementors wanting to maximize take-up of applications exploiting extensions or up-to-the-minute revisions should consider wrapping up that use in shared libraries provided in several versions, so that the loader of an appropriately-capable CPU can select the best version of the library.

The rest of this chapter will motivate and clarify the object code entities defined in the header files.

Design principles for NUBI's ELF object standards

- *Major Options*: NUBI has big- and little-endian variants for 32- and 64-bit CPUs, and in two data-size variants for 64-bits. The big- and little-endian versions will be noted by different values in the ELF header `ei_data` field[21]. NUBI32 and NUBI64 (which we hope may prove somewhat interlinkable) should share an EM_ value.
- *Software options*: NUBI programs have some build options which are so pervasive that they are appropriately recorded in object code flags. These will include:

  Soft float

  C floating point data types are stored and manipulated in regular integer registers, and floating point calculations are done using "built-in" library functions.

  It would be eminently reasonable to encode "position-independent code" and "uses small data region" in the same way; but those options may already be reliably deduced from other information in the object file.
- *Instruction set variants*: will *not* continue to be recorded in the ELF header flags in the same way.

  In the past options such as MDMX, MIPS-3D, paired-single instructions, MIPS32 release 2, MIPS DSP or MIPS MT (the list is incomplete) have been represented by flags. However, there isn't room to encode all such variants, such an encoding will always be unstable and unreliable, and there's no universally desirable action for either build tools or OS loaders to take in the event of a mismatch.

  In particular a linker should not refuse to link NUBI object files on the grounds of incompatibility, unless the incompatibility affects the operation of the linker itself. Lesser incompatibilities should produce at most an error which can be overridden with a suitable linker command line flag.

  We will encode the instruction set evolution level in an ELF header field (there are enough values spare for this purpose), but the program's use of instruction set extensions ("ASE"s) will be summarized in an ".options" section.

  Run-time systems for NUBI programs may compare the instruction set usage with the known capability of the system and act accordingly. Loaders should never refuse to run a program unless they are quite sure that it won't run on this particular system: for example, if an OS gives the loader no access to the capabilities of the CPU, the loader should run the program (and it may break with an exception if it uses an unsupported instruction).

  Such an approach also provides a good example for customers adding home-made new instructions using CorExtend or coprocessor 2. The options section format should be readily extensible.
- *Link-time code fixups*: it's probably impossible to make TLS efficient without providing the linker with the power to simplify heavyweight linkages where they turn out to be unnecessary. So we should bow to the inevitable. But that means we can exploit the same kind of trick to simplify calls and references via the GOT, when we can tell at link time that the call/reference will be within the link unit.

  Such fixups are only attempted when they can be achieved without growing the code and only when the linker sees "standard form" sequences.

[21] Thanks to Felix Burton of WRS, who pointed out that this should suffice, and we don't need two different EM_ values for endianness.

Perhaps NUBI64W should have a distinct value.

Debug conventions

A debugger program (for NUBI purposes) wants to be able to display the state of a program-under-debug at a source-code level, working from a copy of the source code, object files, and access to the memory space of the program.

A note on DWARF Debug information in the object files

Special object code sections are devoted to debug information (which is frequently optional because it makes the object files bigger and provides information useful for reverse-engineering). You may need to rebuild a program with appropriate debug flags.

The governing standard used by NUBI is DWARF2 (as its name suggests to those who've seen or read ‘‘The Lord of the Rings’’, DWARF and ELF are distinct and independent standards which one hopes will fight together, rather than fight each other). DWARF defines the contents of debug sections in the object file which are otherwise uninterpretable binary goop.

DWARF2 originally suffered from the problem that the standardization process ran into the sand without producing a definitive specification. But it's been rescued by the ‘‘Free Standards Group’’. Find out about it by reading [DWARF2]. DWARF2 is machine- and ABI-independent, so we will not describe it further here.

Our reference implementation will be the DWARF support code in the ‘‘BFD’’ library which underlies all the GNU tools which read or write DWARF2. Any non-GNU implementation should be cross-tested against BFD.

6.1 Stack frame and code conventions for debugger navigation

So long as a compiler generates code conforming to the data organization and calling convention standards of NUBI laid down in previous chapters, it will link with other programs and interwork. When you try to debug it, though, you will be interested in some information which is not yet tied down: in particular, a debugger likes to look back up the stack, interpreting what it sees as a call nesting and using saved data to reconstruct data values seen by calling functions (even that data which resided in registers, and has since been saved by some called function).

In principle, DWARF debug records will record where everything went. But such arbitrary freedom is not particularly valuable, complex debugger information is fragile and likely to go wrong, and such a system breaks down completely for assembler modules. It's more robust to define conventions for how stack frames are laid out, at least to the extent that these conventions are harmless to the efficiency of compiled code.

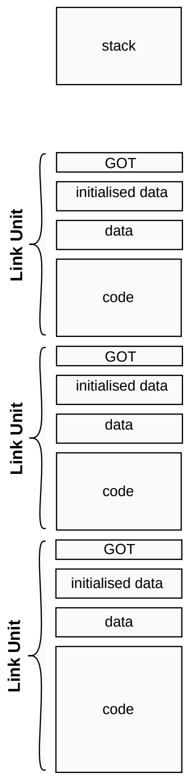
You can see some of those conventions in Figure 3-1 ‘‘Stack layout in NUBI’’.

Linux position-independent code

Linux applications are built using library code which can be shared between multiple processes (running the same or different programs).

A Linux process running an application is reasonably called a *program*, see Figure 7-1.

Figure 7-1 Linux program memory image



The program consists of multiple quasi-independent *link units*[22]. Each link unit is loaded into a number of chunks of memory (“segments”), many of which have initial data provided by chunks of the link unit object file (“sections”). But some chunks of memory - notably, that which holds the stack - are created by the loader.

NUBI must allow program text to be shared with any other active program which incorporates one of the same link units. That other program may have a quite different memory map; so the memory image of NUBI code itself must not depend on any position-dependent alteration of the code by the loader. All code and data references must go on working regardless of where the link unit’s segments end up in memory.

However, the link unit’s segments are brought into memory together, and the relative addresses (“offsets”) of data and instruction points are preserved.

When different link units share variables or function entry points, those have to be computed by the running program. That’s where the GOT (“global offset table”) comes in.

Each link unit has a GOT which occupies its own segment. The GOT contains an entry for each external symbol referred to by any part of the link unit’s code. In the simplest case, the GOT entry has the symbol’s address, as

[22] There’s no consensus on what to call one of the individual binaries making up a dynamically linked program. What we’ve called a “link unit” has been called a “dynamic shared object” (“DSO”), an “object”, or a “module”. But such a thing is quite distinct from a C++ “object”; and the word “module” is already used to mean what is built in one go by the compiler. So in this document I’m going to use the phrase “link unit” to remind you that such a binary is the final thing produced at link time.

known and fixed up by the dynamic linker (who knows every link unit's symbol table).

It's up to the ABI to determine the format of a GOT entry and the various supported linker computations (the ''relocation type'') needed to convert the file-stored version to a usable memory image.

7.1 How link units get into a program

There are three ways:

- *Brought in when the program is loaded*: the base program binary (the ''executable'') has a record which can list off other link units which are to be built in from the start.
- *Loaded on first reference to subroutine*: GOT entries corresponding to function calls to not-yet-loaded link units can be initialized so that the first call indirects to a function in the dynamic linker - a function whose job is to load the missing link unit, fix up the GOT and then return control (carefully) to the newly loaded link unit's function.

  Note that there is currently no corresponding trick which allows a link unit to be loaded on demand as a result of a data reference.
- *Loaded explicitly* : as a result of a call to `dlopen()`. It's worth noting that a dynamically-opened link unit really does behave pretty much exactly like one marked as necessary at link time.

  However, in the event that the same symbol is referenced by different link units[23] it is significant which link unit gets loaded first.

7.2 Global Offset Table (''GOT'') organization

The GOT is an automatically-generated data structure - there is at least one for each link unit, though it's perfectly legal for there to be more than one[24]. Each function knows its own GOT.

Each entry in the GOT is an absolute pointer to a data item[25] or function entry point which is defined (implicitly or explicitly) as external by any function in the link unit which uses this particular GOT. GOT entries depend on the layout of link units within the program's address space, so must be computed by the dynamic loader as link units are loaded.

Because the binary image of the code (and read-only data) of a link unit is really shared in an OS like Linux, we can't do any address-map-dependent fixups on the code when loading it. The GOT, though, is synthesized for each program, and the pointers in it are different for each map which uses the shared link unit.

GOT accesses are frequent and could be lengthy, so in PIC code the compiler/linker synthesize code which keeps the `gp` register pointing to each function's GOT. That's slightly complicated, because the GOT (together with the rest of the link unit) may be located anywhere in program virtual memory.

Code which is using a GOT entry to access data in a different link unit knows *only* the offset in the GOT where it should find the pointer: those offsets are finally resolved when the link unit is linked together, but the pointer stored in the GOT is not fixed until the other link unit is loaded at program load time. So the compiler must generate code which does a double-indirect, loading a pointer to the external data:

[23] Sometimes this might just be an error causing the load to fail, but some symbols are specifically marked as permitting multiple definitions (''first definition loaded wins''), with some safety restrictions.

[24] The use of multiple GOTs is the preferred way to deal with large link units which overflow the 128KB GOT size (if the GOT grows bigger than this, compilers and assemblers have to generate longer code sequences to retrieve pointers from the GOT).

[25] A GOT entry referencing local data can in fact be shared between different local data items so long as their relative locations will be fixed at link-time, and the compiler knows that. This can help avoid overrunning GOT tables and is a recommended optimization.

```
        load:
                lw t1, gp(MYSYMBOL_INDEX)
                lw t1, (t1)

        store:
                lw t1, gp(MYSYMBOL_INDEX)
                sw t2, (t1)

        call:
                lw pf, gp(MYFUNCTION_INDEX)
                jalr pf
```

It's helpful for a remote function to know it's been called by address, since then there is a register which already contains the address of the function (which makes it easier for the function to find its own GOT). This motivates standardizing on one of the temporary registers as the PIC-function call register `pf`. A function prologue invoked through the GOT using the `pf` register can compute GP in a single step[26]:

```
        entrypoint:
                addu gp, pf, GOT - entrypoint
```

7.2.1 The GOT and demand-loading

You could build a shared-library system where the loader pre-assigns space for all required link units and builds all the various GOTs as part of loading a program, before starting to run it.

There's a reason why you don't do that, and a reason why it's not much extra work to avoid it.

So what's wrong with linking everything at program load time? Well, building all the GOTs and chasing all the libraries for a large application can be a big overhead. This makes program start-up slower (and users hate that). There are also often going to be a lot of rarely-used functions in programs, which may be implemented in libraries which need almost never be loaded.

So it's appealing to load a program with most of its shared libraries "missing". It's easy to catch function calls to not-yet-loaded libraries: the dynamic loader (which is itself another shared object) simply points GOT entries at a dummy entry point in the loader itself, where code resides to load the missing object and retry the first access.

No such trick works for data. If a main program and shared library have data in common, then the loader has to set up the GOTs to implement that linkage at load time.

This sounds like extra work: but it turns out there's a reason why programs will grow extra shared objects well after load time. Linux offers a `dlopen()` call which allows a program to bind in a specific named shared library, after which `dlsym()` returns a pointer to a named thing in the library, now available in your address space. Such "computed" library loads are valuable tools for making programs which accommodate themselves to their environment in various ways.

The whole `dlopen()` thing has distinct semantics from the shared libraries automagically acquired from your build process. And the shared-library system is overengineered for it - a `dlopen()` library doesn't need the GOT at all. But it's near enough the same thing that it isn't worth building two distinct mechanisms.

7.3 Conventions to help optimize position-independent code

In the simplest case, every call or static data reference in a PIC program goes through the GOT.

Such references are clearly somewhat expensive, in terms of extra instructions to run. But they turn out to be worse than that. The GOT references pollute the D-cache, increasing miss rate; few CPUs can follow pointer chains at full speed; and - worst of all - the **`jalr`** instruction which is used for all via-the-GOT calls defeats the "branch prediction" logic in all known MIPS CPUs, which probably costs rather more time than the whole visible calling sequence.

[26] It's probably not a single step: the assembler will often need an extra instruction or two to generate a load with a large offset.

So we'd like to do as many calls and references as is possible without the GOT overhead.

Function calls within the link unit need only a PC-relative call instruction, particularly if the caller and the target function share the same GOT pointer. MIPS has a PC-relative call instruction in **`bal`**, though its limited range (±128KB of program) means that there are many link units which are too big for it to be used universally.

Data references within the link unit would apparently need a PC-relative load, but could use link-time-known offsets from the GOT pointer itself. Further, data references within a module can potentially benefit from compile-time-known relationships between different locally-defined data items[27], which would allow a program to use the GOT less often.

In both cases the compiler is too early to apply any optimization robustly. So for better PIC code we need to empower the linker to perform some transformations on the whole link unit. We will only consider transformations which keep the size of the code constant.

Optimizing intra-link-unit data references and calls

Data known to be defined[28] in this link unit must be at a link-time-known offset from the module's GOT pointer in `gp`. So we can optimize:

```
lw rg, GOTNO(gp)        lui rg, %hi(sym-got)
                    →   addui rg, gp, %lo(sym-got)
lw dest, 0(rg)          lw dest, 0(rg)
```

That's got one extra instruction, but will run faster on any plausible CPU. The trouble is that if we want to allow for such a transformation, we need to leave the space on all loads.

Optimizing calls

The same trick is of some value on intra-link-unit calls, but it doesn't affect the biggest inefficiency, that of using a call-by-register which defeats most branch predictors, causing a pipeline stall. Where possible the linker should transform an intra-link-unit call into a **`jal`** instruction, which is predictable.

The other avoidable overhead is recomputing the GOT. When you make a call from the same link unit, the GOT value will often be unchanged. To fix this, functions will have to have two entry points: the first is for where `gp` needs to be recomputed, the auxiliary entry point is where that is not necessary.

[27] MIPS Technologies have such a feature in GCC and are working to port it to GCC4: it's particularly valuable for MIPS16 code.

[28] Some data items (with a "weak" local definitions) may have only a provisional binding to a data item in the link unit, and can't be optimized.

Signals, signal frames and the “sigcontext” structure.

`struct sigcontext` is a machine-dependent data structure which in the Linux OS is accessed both by the OS and application programs (albeit only rather specialized applications: debuggers, profilers, and interpreters using binary code). Most such interactions are mediated by the C runtime library, to decouple application and kernel evolution: but that’s not done in this case.

One could - with some justification - argue that evolving `sigcontext` is a kernel/application interaction, and beyond the scope of some ABI documents. However, the MIPS `sigcontext` associated with the o32/n32/n64 specifications has become very untidy, and is not extensible to support application use of MIPS Technologies’ DSP ASE (which adds three double-size “accumulator” registers and a word-size control register). But the MIPS architecture now also permits the user-level use of up to 128 “co-processor 2” registers and the “CorXtend™” feature makes it fairly easy to add new computational instructions.

A change to `sigcontext` is like a change of ABI in the pragmatic sense that it requires a number of dependent applications to be rebuilt. So it seems virtuous to do both at once.

For the purposes of this section you don’t need to know a whole lot about signals. A signal is an event raised by calling an in-kernel routine, which affects a user-level process. Signals are distinguished by a `signum`: only a small number of values are typically available. Processes may elect to “handle” a signal: to do that you call the `signal()` library function, which takes the address of a function within the process, which will be magically called whenever the associated signal is raised.

If a process doesn’t handle a signal, the event may be ignored or it may terminate the process: which happens depends on a complicated mixture of folklore and configuration which we won’t go into.

But when a signal is handled, normal process execution is suspended while the nominated function is called: a signal is like a user-level interrupt.

Like interrupts, signals can be nested: there’s nothing which privileges a signal handler and thereby prevents another signal being delivered, causing another signal to be taken. Only the signal currently being handled is blocked until the handler returns.

During the signal handler’s execution, the process is just executing in user space. When the signal handler returns, it returns to a fragment of code called a *signal trampoline*[29]. The trampoline ends up calling the very OS-dependent `sigreturn()` system call. The important message here is that the OS kernel keeps relatively little state information about a signal handler: it doesn’t care, for example, whether it ever returns.

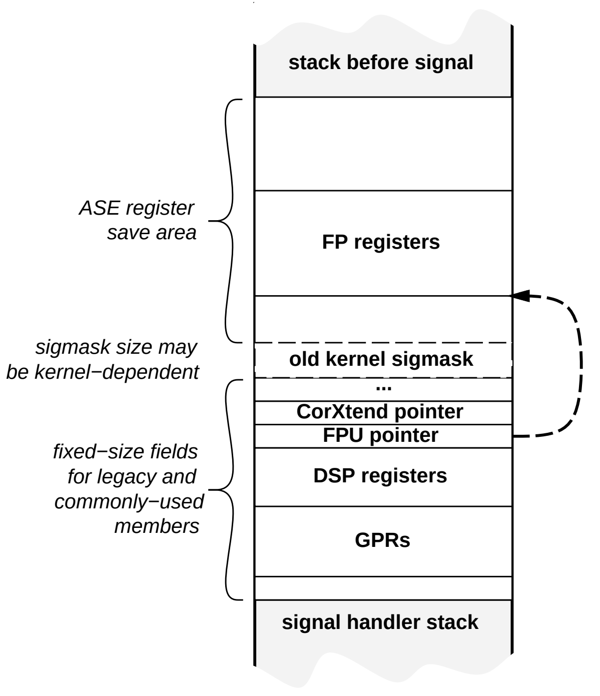
So that raises a question: prior to the signal, the process was cheerfully running in user space. Signals are raised in the kernel, so at the point a signal is raised the target process is typically stopped as a result of some hardware interrupt or other, with all its low-level machine state saved on the process’ kernel stack. But once the kernel has invoked the signal handler, the process will be running in user space and the kernel stack will be empty. Someone has to keep a copy of the low-level machine state of the original interrupted user code: so it’s kept on the process’ user stack, in the form of a `struct sigcontext`. Because it contains all the general purpose register (for example), this structure is architecture-dependent.

If the signal frame was just a magic cookie maintained by the kernel with some help from the C library, it would not need description here. But signals are fabulously useful for debuggers, just-in-time translators, profilers and other pieces of application code which do wonderful things by seeing their own process’ code as data. Such programs need to navigate the signal frame.

[29] In current MIPS and other-architecture practice, the trampoline is often created by the `signal()` code by synthesizing machine instructions onto the stack, but this is now avoidable and it’s probably good to avoid it.

8.1 NUBI signal data structure

Figure 8-1 Signal frame (sigcontext) elements for a MIPS/NUBI program



Notes:

- *No executable code on stack*: the "signal trampoline" code which is invoked when the signal handler returns will be hard-wired into a library - the common approach of synthesizing it onto the stack is now avoidable.
- *Self-documenting*: the size of the ASE register file areas, and the size of the whole sigframe structure, will be written into the signal frame and used by intelligent signal frame navigators.
- *Saved kernel sigmask*: will be located just above the fixed-size section of the structure, so that it can be (as is common) extensible.
- *Saving registers*: The GP registers `$0-31`, DSP registers `$ac0-3` (each a hi/lo pair) and `dspcontrol` will be awarded fixed places in the structure.

  Allowing for all possible registers from other existing ASEs (beginning with the FPU extension) will make for a very large structure, while not future-proofing us against the large register sets which seem quite likely to accompany evolved SIMD instruction sets. So we propose that - starting with the floating point registers of the FPU extension - ASE registers should be moved out of the main structure. Those ASE registers which are saved by the kernel will be saved higher up the stack, and referenced through a pointer in `sigcontext`.

Thread-local storage

Multi-threaded code is interesting. A POSIX standard sets out the ABI by which you can create and manage threads, and the Linux NPTL system is an implementation of the standard. On the face of it, the POSIX threads environment is one in which threads are anonymous: they run, they are often seen as interchangeable, but they have little of their own identity.

NPTL uses kernel threads, in the sense that every POSIX thread is a kernel thread; but much of the system is built in the libraries. For the NPTL code in the libraries which implements the POSIX standard, anonymity is a pain: the system needs to attach data to each thread.

It seems probable that real threaded applications will often want to keep per-thread data. This feature has been defined as a C-level concept of ‘‘thread-local storage’’ or TLS: define data with the appropriate attribute, and each thread will get and address its own unique copy of that data.

NUBI reserves a general purpose register as a thread pointer, `tp`: the compiler generates TLS data sections which the NPTL system replicates as required for each new thread, then sets the pointer in such a way that compiled code gets the right data. The NPTL library functions figure out the value for `tp` for a new thread based on some notion of thread identity acquired from the kernel: from then on the value persists, since no other software is permitted to write `tp`.

This space intentionally left blank... we will adopt a vague definition here, to be detailed when implemented. We expect the implementors to value compatibility with the existing MIPS NPTL/TLS scheme and a representative ‘‘green-field’’ ABI such as the Linux IA64 ABI.

Appendix A: References

Other manuals on ABIs and object code

[DWARF2]:
"DWARF Debugging Information Format". Published in 1993 by UNIX International, Waterview Corporate Center, 20 Waterview Boulevard, Parsippany, NJ 07054, USA. Revision: Version 2.0.0 (July 27th, 1993) Available from the Free Standards Group at:
`ftp://ftp.freestandards.org/pub/dwarf/dwarf-2.0.0.pdf`

[SVR4]:
The "System V Application Binary Interface" Edition 4.1, written by AT&T and the Santa Cruz Operation Inc. Available from `http://www.caldera.com/developers/devspecs/gabi41.pdf`.

[SVR4_MIPS]:
System V Application Binary Interface - MIPS RISC Supplement, 3rd edition. Available from:
`http://www.caldera.com/developers/devspecs/mipsabi.pdf`.

[SGI_ELF64]:
"64-bit ELF Object File Specification" Draft Version 2.5, from Silicon Graphics at:
`http://techpubs.sgi.com/library/manuals/4000/007-4658-001/pdf/007-4658-001.pdf`.

[SGI_N32]:
"MIPSpro N32 ABI Handbook", at
`http://techpubs.sgi.com/library/manuals/2000/007-2816-005/pdf/007-2816-005.pdf` from Silicon Graphics.

MIPS Architecture reference material

[MIPS64]:
"MIPS64® Architecture for Programmers" in multiple volumes. Volumes I (Introduction, MIPS Technologies document number MD00083) and II (Instruction Set, MD00085) are probably the most relevant.

[MIPS32]:
"MIPS32® Architecture for Programmers" in multiple volumes. The "Introduction" is MIPS document number MD00082.

General MIPS reading

[SMR]:
"See MIPS Run", Dominic Sweetman 1999, published by Morgan Kaufmann, ISBN 1–55860–410–3.

Appendix B: Evolving NUBI

Guidelines for early implementors

We will make our best efforts to create an unambiguous definition of a sensible family of ABIs. But as in all intellectual constructs of any complexity, we shall not entirely succeed.

MIPS Technologies will put resources in to ensure that our GNU toolkit and Linux work is upfront using NUBI as we go along. We hope that we'll trip over a fair proportion of any problems ourselves, and save anyone else from being involved.

But again, this is unlikely to be perfect. Early adopters - and we hope some of you will be in there early - will not only find bugs, but may suffer from the specification tweaks to work around the ambiguities and bugs everyone else finds.

Compatibility testing

We need to set up some kind of interworking sandbox, where different evolving tools can be tested against each other. To be filled in later.

Appendix C: 64-/32-bit interworking tricks

This appendix shows (in what is currently a rather hand-waving manner) how 32- and 64-bit variants of NUBI might intercall.

32-/64-bit calling gasket

A 32-bit program using NUBI can directly call a function in a 64-bit program using NUBI64, provided there are no 64-bit integer arguments or return values involved.

A 64-bit program function can't ever call into a 32-bit program directly, because its "saved" registers would likely be corrupted. Instead it must use a gasket which allocates stack space for all the saved registers (and the return address) and arranges for them to be restored on return - a fairly expensive operation.

Where the intercalling function has a 64-bit integer argument or return value, something more magic has to happen:

- *32-bit calling 64-bit*: from the 32-bit side a `long long` argument will have been passed with a pointer, so needs to be dereferenced in place - pretty straightforward.

  A `long long` return value will have been anticipated by providing a pointer to a location to be filled. The gasket will need to copy the 64-bit data returned in the return register `a0` into the appropriate location.

  I think that's really it...

- *64-bit calling 32-bit*: a `long long` argument will be in a register; the gasket will have to create a stack location, copy the data to that place, and replace the register value with a pointer to the stack slot.

  A `long long` return value will be returned to a pointer supplied by the gasket; again, the gasket will need to create a stack location and provide its address as an additional argument before calling the 32-bit function. On return that data must be copied into a register before return to the 64-bit universe.

Argument conversions - of themselves - leave nothing behind which has to be cleaned up when the function returns. But in general 64-bit routines calling 32-bit code will need to save and restore `s`-registers across the 32-bit function call (32-bit code will only preserve the low half of those registers).

Building gaskets

You can get the linker to emit the gasket code, but it's quite complicated and means passing lots of mysterious data to the linker.

It's probably easier to recycle a trick used for the MIPS16 code generator; the compiler which builds code always constructs a gasket for the other-width, but puts it in a separate section which is ignored for simple builds. The compiler is in a better situation to know what needs to be adjusted or saved.

Appendix D: Revision History

Revision	Date	Description
0.01	29th November 2004	Circulated for discussion among interested parties inside MIPS Technologies.
0.02	1st December 2004	Changes in response to MIPS Technologies feedback.
0.10	2nd December 2004	Draft for distribution to discussion group.
0.11	16th December 2004	Revised draft for group in the light of internal and external feedback. Change bars are against v0.10.
0.15	13th September 2005	Substantial extension, working towards a complete draft to support a trial implementation. Change bars are against v0.11
0.16	23rd September 2005	For circulation outside MIPS Technologies. Object code section much improved, and new section on signals.
0.19	5th October 2005	Much work on relocations and other object code details. Released. Change bars are against Revision 0.16.