TRIPS Application Binary Interface (ABI) Manual

Aaron Smith Jim Burrill Robert McDonald
Nicholas Nethercote Bill Yoder Doug Burger
Stephen W. Keckler Kathryn S. McKinley

October 10, 2006 - Version A.06

Tech Report TR-05-22
Department of Computer Sciences
The University of Texas at Austin

This document specifies the TRIPS Application Binary Interface (ABI) Manual for the TRIPS architecture, a novel, scalable, and low power architecture for future technologies.

Contents

1 Overview

This document describes the application binary interface for the TRIPS Grid Processor. The goal of this document is to provide a consistent standard for vendors and researchers to follow. No thought has been given to any other language besides C and FORTRAN. You are encouraged to build upon and expand this document for other languages such as C++ and Java.

For additional information relevant to the Trips Application Binary Interface, please consult the following manuals:

- *TRIPS Processor Reference Manual*
- *TRIPS Intermediate Language (TIL) Manual*
- *TRIPS Assembly Language (TASL) Manual*
- *TRIPS Object File Format (TOFF) Specification*

2 Architectural Description

For a complete architectural description, refer to the *TRIPS Processor Reference Manual.*

2.1 Registers

The TRIPS architecture provides 128 general purpose registers (GPRs). By convention GPRs are named R0 - R127. The architecture makes no distinction between floating point and general purpose registers. The TRIPS architecture does not define any special purpose control registers which are accessible through the instruction set.

2.2 Fundamental Types

Table 1 shows the TRIPS equivalents for ANSI C fundamental types along with their sizes and alignments. Fundamental types are always aligned on natural boundaries. The TRIPS architecture supports 64, 32, 16 and 8-bit load and store operations. All data is in big endian byte order.

For the purposes of this document, we define the following types:

- *doubleword* – A doubleword is 64-bits and the least significant 3-bits of the address of a doubleword in memory are always zero.
- *word* – A word is 32-bits and the least significant 2-bits of the address of a word in memory are always zero.

ANSI C	Size (in bytes)	Alignment (in bytes)
char	1	1
unsigned char	1	1
signed char	1	1
short	2	2
unsigned short	2	2
signed short	2	2
int	4	4
unsigned int	4	4
signed int	4	4
enum	4	4
long	8	8
unsigned long	8	8
signed long	8	8
long long	8	8
unsigned long long	8	8
signed long long	8	8
float	4	4
double	8	8
long double	8	8

Table 1: TRIPS Fundamental Types

- *halfword* – A halfword is 16-bits and the least significant bit of the address of a halfword in memory is always zero.
- *byte* – A byte is 8-bits.

2.3 Compound Types

The alignment requirements for arrays, structures, unions and bit fields are summarized in Table 2.

Arrays are aligned according to the alignment of their individual elements. For example,

```
char ac[10]; /* aligned on 1-byte */
short as[10]; /* aligned on 2-bytes */
float af[10]; /* aligned on 4-bytes */
```

Structures and unions are aligned according to their most restrictive element. Padding should be added to the end of the structure or union to make its size a multiple of the alignment. Fields within structures and unions are aligned according to the field's type with the exception of bit fields. Padding should be added between fields to ensure alignment. For example,

Compound Type	Alignment
Arrays	Same as individual elements
Unions	Most restrictive alignment of members
Structures	Same as unions
Bit fields	Same as individual elements

Table 2: Alignment of Compound Types

```
struct s1 {
  char bc[9];  /* aligned on 1-byte */
  short bs;    /* aligned on 2-bytes */
  int bi;      /* aligned on 4-bytes */
  char bc2[9]; /* aligned on 1-byte */
};
```

The individual elements `bc` and `bc2` are aligned on 1-byte boundaries. The elements `bs` and `bi` are aligned on a 2-byte and 4-byte boundaries respectively. A 1-byte pad will be added between `bc` and `bs` in order to align `bs` on a 2-byte boundary. Since `int` is the most restrictive element of the structure, a 3-byte pad would be added to the end of the structure to align it on a 4-byte boundary.

The maximum size of a bit field is 64-bits. Bit fields cannot be split over a 64-bit boundary. Zero-width bit fields pad to the next 32-bits, regardless of the type of the bit field. No other restriction applies to bit field alignment. However, bit fields impose alignment restrictions on their enclosing structure or union according to the fundamental type of the bit field.

3 Function Calling Conventions

3.1 Register Conventions

Table 3 defines the register conventions for the TRIPS architecture. There is no distinction between floating point and integer values for the purpose of the conventions.

Registers R0, R1 (stack pointer), R2 (return address) and R12–R69 are *callee-save* or *non-volatile*, which means that the compiler preserves their values across function calls. Any function which uses any register in this class must save the value before changing it, and restore it before the function returns.

The remaining registers, R3–R11 and R70–R127, are *caller-save* or *volatile*, which means that they can be overwritten by a called function. The compiler will ensure that any function which uses any register in this class must save the value before calling another function, and restore it after that function returns, if that value is to be reused after the call.

Register R1 (SP) contains the function's stack pointer. It is the responsibility of the function to decrement the stack pointer by the size of its stack frame upon entry in the function

Reg. #	Usage and description	Lifetime
R0	System Call ID for SCALL	Callee-save
R1	Stack Pointer	Callee-save
R2	Return Address Register	Callee-save
R3	Arguments and Return Values	Caller-save
R4	Arguments and Return Values	Caller-save
R5 - R10	Arguments	Caller-save
R11	Reserved for Environment Pointer	Caller-save
R12	Frame Pointer or Local Variable	Callee-save
R13 - R69	Local Variables	Callee-save
R70 - R127	Local Variables	Caller-save

Table 3: Register Conventions

prologue and increment the stack pointer by the size of its stack frame upon exit in the function *epilogue*. To support the debugger, the compiler stores the caller's stack pointer in the link area as a *back chain pointer*, prior to decrementing the stack pointer register (SP) in the prologue.

If a function uses *alloca*, which allocates space for the user on the stack, register R12 (FP) is used to access the function's stack frame while allowing the stack pointer (R1) to be changed by *alloca*. Upon entry to such a function, the address in R1 is first decremented and then this address is copied into R12. Then register R12 is copied back into R1 just before register R1 is incremented on the function's return.

Register R2 contains the function's return address upon entry. It is the responsibility of the function to preserve its return address so that it may return to its caller. If the function calls no other functions, it may do this by keeping its return address in R2. Otherwise, it must save the return address in the link area.

3.2 Stack Frame Layout

Each function has a stack frame on the runtime stack which grows downward from high addresses. Figure 1 shows the stack frame organization. Note that the figure shows low memory addresses at the top and high addresses at the bottom.

From low to high addresses, the stack frame for a function (callee) contains:

- Fixed Size Link Area
- Argument Save Area
- Local Variables
- Register Save Area

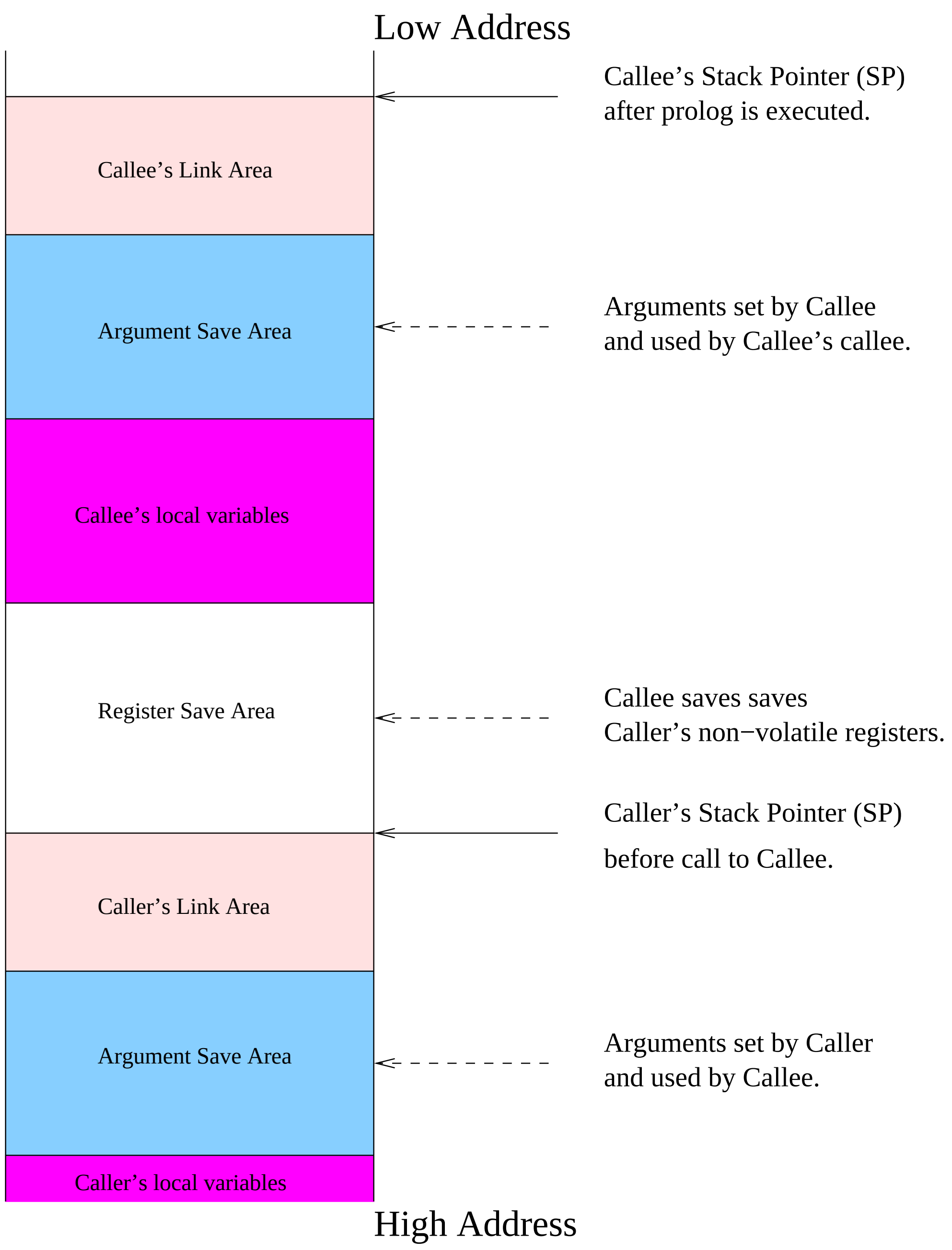
Low Address
Callee's Stack Pointer (SP) after prolog is executed.
Callee's Link Area
Argument Save Area
Arguments set by Callee and used by Callee's callee.
Callee's local variables
Register Save Area
Callee saves saves Caller's non−volatile registers.
Caller's Stack Pointer (SP) before call to Callee.
Caller's Link Area
Argument Save Area
Arguments set by Caller and used by Callee.
Caller's local variables
High Address

Figure 1: Stack Frame Layout

(SP)	Back chain pointer; i.e., stack frame address of caller
$(SP) + 8$	Callee's return address

Figure 2: Link Area After Callee Prologue

3.2.1 Link Area

This fixed size area holds (a) the address of the caller's stack frame and (b) the callee's return address (Figure 2):

- The first doubleword (lowest address in the callee's stack frame) contains the caller's stack pointer value, sometimes called the "back chain". The first stack frame (that is, the stack frame of the _start function) will have a back chain value of 0.
- The second doubleword contains the callee's return address, which is set by the caller before branching to the function. If debugging is not required, this doubleword may be left undefined in order to avoid a store to memory.

  Note: If a function dynamically allocates space on the stack (e.g., *alloca*()), then the allocated space must be between the link area and the argument save area. This means that the link area must be moved when the allocation is performed. The stack pointer register must always point to the link area.

Figure 3 shows the use of back chain pointers to traverse the stack frames.

3.2.2 Argument Save Area

This variable size area is large enough to hold all of the arguments that a routine may pass to any of the routines that it calls as determined by:

- A minimum of `MAX_ARG_REGS` (8) doublewords is usually reserved for the argument save area because the caller can not know if it is calling a routine that uses `va_start`. See section 3.5.
- For a "leaf routine" this area may contain 0 doublewords. When a routine calls a function it places the first `MAX_ARG_REGS` doublewords of arguments in the argument registers (R3 . . . R10). Any additional doublewords of arguments are placed starting in doubleword 8 of the argument save area. Each argument is placed in at least one register or in at least one doubleword in the argument save area. Arguments larger than a doubleword may be split between a register and the argument save area. The least significant 3-bits of the address of any argument in the argument save area are zero.

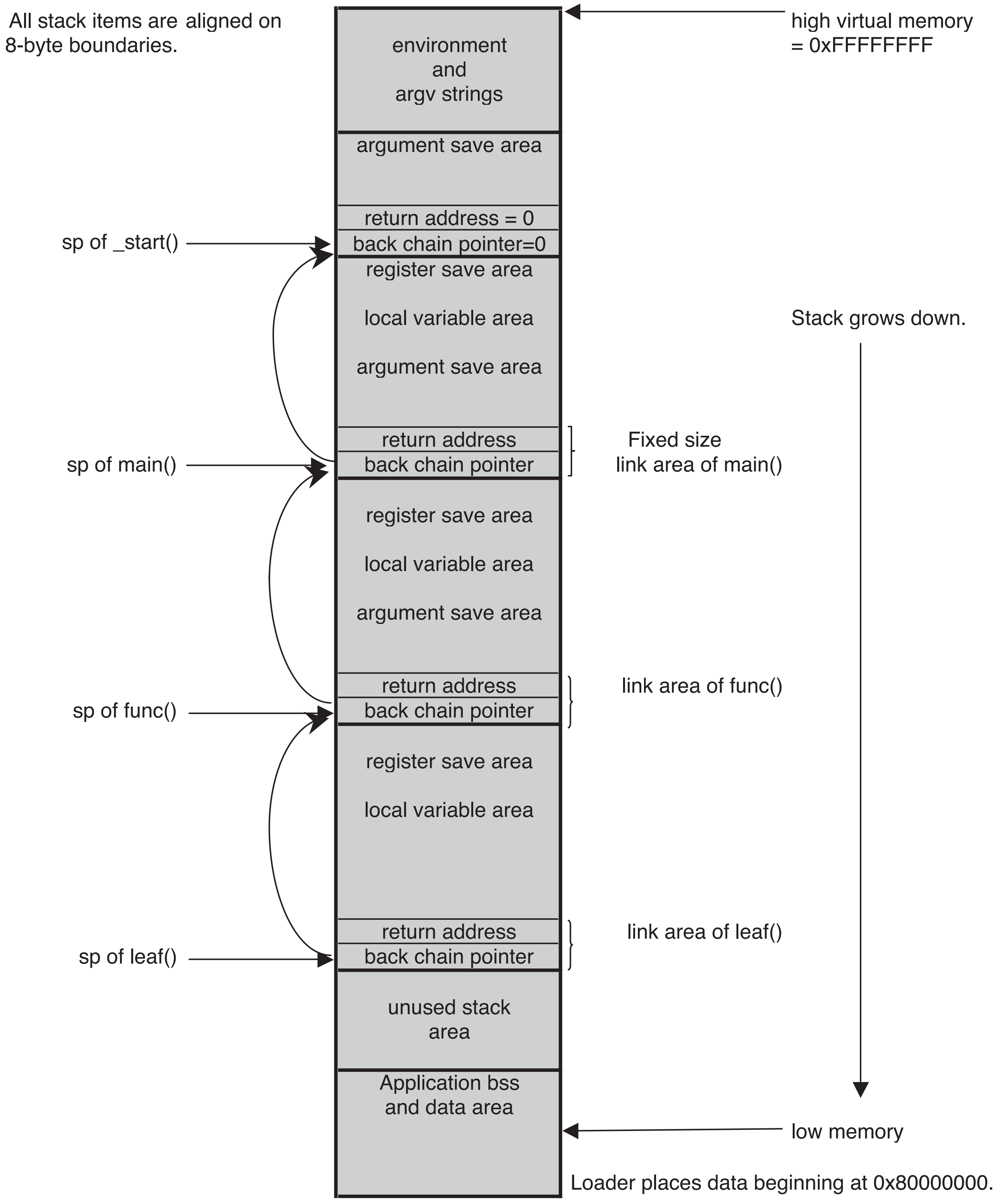

Figure 3: TRIPS Stack Linkages

3.2.3 Local Variables

Any local variables of a callee that must reside in memory are placed in the local variable area. The least significant 3-bits of the address of any variable are always zero. The size of the area may be zero.

3.2.4 Register Save Area

This area holds the contents of any of the callee-save registers that the callee modifies. Registers are saved to increasing addresses. For example, if the callee modifies only the callee-save registers R60 and R62 then the register save area will be 16 bytes. Register R60 will be stored at offset 0 and register R62 will be stored at offset 8 into the register save area. The least significant 3-bits of the address of any register in the register save area are zero. The size of the area may be zero.

3.2.5 Requirements

The following requirements apply to the stack frame:

- The least significant 4-bits of the value in the stack pointer register (SP) shall always be zero.
- The stack pointer shall point to the last word of the current stack frame. Thus, (SP) is the address of the "back chain" word of the link area. The stack shall grow downward, that is, toward lower addresses.
- The stack pointer shall be decremented by the called function in its prologue and restored prior to return.
- Before a function changes the value in any callee-save general register, Rn, it shall save the value in Rn in the register save area.

3.3 Parameter Passing

Both scalar and compound type parameters are passed in registers R3 through R10. Parameters shall be assigned consecutively to registers so that R3 contains the first function parameter. Assuming that the first argument is requires 8 bytes or less, R4 contains the second. This continues until all argument registers are occupied. If there are not enough registers for the entire parameter list then the parameters overflow in consecutive order onto the argument save area of the stack.

Scalars less than 64-bits are right justified within the register. The caller must not assign more than a single scalar argument to a register.

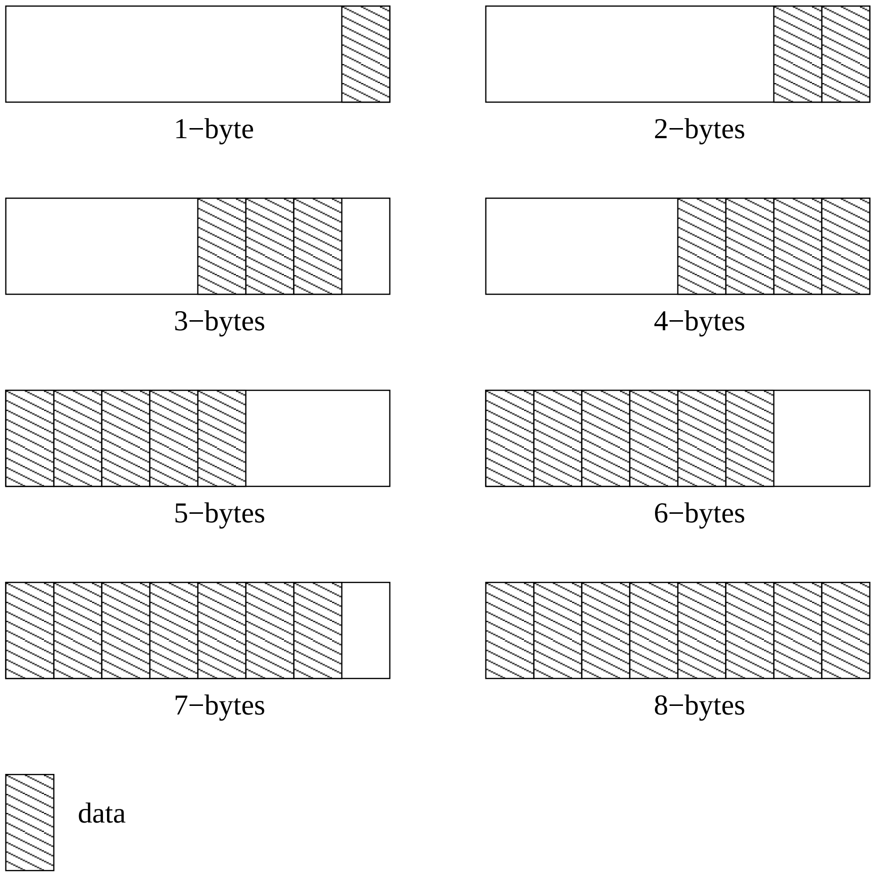

Figure 4: Passing C Structs

Compound types (C structs) larger than 64-bits are packed into consecutive registers. Compound types less than 64-bits are placed within the register in the position that allows a simple store to place them in memory aligned upon a doubleword boundary (see Figure 4).

The argument save area, which is located at a fixed offset of `ARG_SAVE_OFFSET` (24) bytes from the stack pointer, is reserved in each stack frame for use as an argument list. A minimum of `MAX_ARG_REGS` (8) doublewords is reserved if the routine calls another routine. The size of this area must be sufficient to hold the longest argument list being passed by the function which owns the stack frame. Although not all arguments for a particular call are located in storage, consider them to be forming a list in this area, with each argument occupying one or more doublewords.

If more arguments are passed than can be stored in registers, the remaining arguments are stored in the argument save area.

The rules for parameter passing are as follows:

- Each argument is mapped to as many doublewords of the argument save area as are required to hold its value.

  1. Single precision floating point values are mapped to a single doubleword.

2. Double precision floating point values are mapped to a single doubleword.
3. Simple integer types (char, short, int, long, enum) are mapped to a single doubleword. Value shorter than a doubleword are sign or zero extended as necessary.
4. Pointers are mapped to a single doubleword.
5. Aggregates and unions passed by value are mapped to as many doublewords of the argument save area as the value uses in memory.
6. Other scalar values, such as FORTRAN complex numbers, are mapped to the number of doublewords required by their size.

- If the callee has a known prototype, arguments are converted to the type of the corresponding parameter before being mapped into the parameter save area. For example, if a long is used as an argument to a float double parameter, the value is converted to double-precision and mapped to a doubleword in the argument save area.
- The first `MAX_ARG_REGS` (8) doublewords mapped to the argument save area are never stored in the argument save area by the calling function. Instead, these doublewords are passed in registers as described above.
- Argument values beyond the first eight doublewords must be stored in the argument save area following the first eight doublewords. The first eight doublewords in the argument save area are reserved for the initial arguments, even though they are passed in registers.
- General registers are used to pass some values. The first eight doublewords mapped to the argument save area correspond to the register R3 through R10. If the arguments are mapped to fewer than eight doublewords of the argument save area, registers corresponding to those unused doublewords are not used.
- If the callee takes the address of any of its parameters that are passed in registers, then those parameters must be stored by the callee into the argument save area.

Note: if the compilation unit for the caller contains a function prototype, but the callee has a mismatching definition, and if the callee takes the address of any of its parameters, the wrong values may be stored in the first eight doublewords of the argument save area.

3.4 Return Values

Functions shall return values of type float, double, int, long, enum, short, and char, or a pointer to any type, as unsigned or signed integers as appropriate, zero- or sign-extended to 64-bits if necessary, in R3.

Aggregates or unions of any length shall be returned in a storage buffer allocated by the caller. The caller will pass the address of this buffer as a hidden first argument in R3, causing the first explicit argument to be passed in R4. This hidden argument is treated as a normal formal parameter, and corresponds to the first doubleword of the parameter save area.

Functions shall return complex floating point scalar values of size 16-bytes or less in registers R3 (real-part) and R4 (imaginary part).

3.5 Variable Arguments

If the callee uses `va_start` it is the callee's responsibility to store the registers R3 through R10 in the argument save area. The remaining arguments are stored by the caller.

The `va_start` operation causes the address of the specified parameter to be stored in the doubleword allocated for the `va_list` variable. As each argument is accessed by `va_arg` this address is incremented by the proper multiple of 8.

There is no provision in this specification that defines how a "variable argument" function can determine the number of arguments that were passed to it.

4 Runtime Support Functions

4.1 Application Memory Organization

The TRIPS prototype runtime system lays out virtual memory for applications from high virtual addresses to low virtual addresses as follows:

- *environment* – At the "top", or highest address, of application memory is the program environment, which is passed through to the program loader in the `**envp` string array, by the call to the program's `main()` routine.
- *stack* – Beneath the program environment area is the stack, which grows "downward" in 8-byte decrements, toward lower addresses.
- *heap* – The heap, placed on top of the program's text and data segments, grows upward by means of the `brk()` system call.
- *bss* – The unitialized data section, for variables tagged with the `.comm` directive, sets the boundary between the program text and data area and the heap area.
- *initialized data* – The program's read/write initialized data section appears at lower addresses than the `.bss` area.
- *read-only data* – This area is reserved for initialized data that is marked by the compiler with the `.rdata` directive as read-only.
- *program text* – At the lowest program addresses are the code blocks comprising the program's executable section.

Register	Description
R1	The initial stack pointer, aligned to a 16-byte boundary.
R3	Contains *argc*, the number of program arguments.
R4	Contains *argv*, the array of NULL-terminated argument strings.
R5	Contains *envp*, the array of NULL-terminated environment strings.

Table 4: Registers Initialized by the Loader

4.2 Process Initialization

Application behavior at startup on a TRIPS processor is modeled on PowerPC conventions.[1] For an application whose entry point is defined as:

```
int main(int argc, char ** argv, char ** envp)
```

Table 4 lists the contents of registers when the loader returns control to the system software. The contents of other registers are unspecified. It is the responsibility of the application to save those values that will be needed later.

The loader will push the argument count, argument values, and environment strings as the first items on the user-stack, starting at the top of application memory. Next, the loader will push the *addresses* of those strings onto the stack. Hence, R1 will point to the stack address just below the values supplied from the environment and arguments to the program, whose value is a NULL pointer.

4.3 System Calls

System call support on the TRIPS prototype simulators is provided through the SCALL instruction. As defined in the *TRIPS Processor Reference Manual*, when a SCALL instruction is executed, a System Call Exception will occur after the program block with the SCALL commits. The TRIPS prototype simulators provide a runtime exception handler that determines the type of system call and services the request.

To invoke a system call, the identifier for the call is placed in R0. The return address and arguments for the call are passed in R2 and R3–R10 in accordance with the *Function Calling Conventions*, and upon completion, the result code is returned in R3. If the system call was serviced successfully, the value returned in R3 will be 0. Otherwise, R4 will contain the value of errno from the simulator's host environment. Note that if no error has occurred, the value of R4 will be undefined upon return from a system call.

The TRIPS prototype simulators currently provide support-by-proxy for the system services listed in Table 5. These services are defined in the /usr/include/sys/syscalls.h TRIPS system header file.

[1]Zuker, Steve and Karhi, Kari: *System V Application Binary Interface: PowerPC Processor Supplement*, 1995.

Service	Numeric ID
exit	1
read	3
write	4
open	5
close	6
creat	8
unlink	10
time	13
lseek	19
brk	45
gettimeofday	78
stat	106
lstat	107
fstat	108

Table 5: System Call Identifiers

5 Standards Compliance

This section documents any deviation from the relevant standards in use for the TRIPS system. This section discusses only known deviations for which no compliance is planned. All other deviations should be regarded as bugs in the relevant software or hardware.

The relevant standards are

- ANSI^{TM} X3.159-1989 1989 C Programming Language
- ISO/IEC 9899 1999 C Programming Language
- ANSI^{TM} X3.9-1978 Fortran 77 Programming Language
- IEEE 754-1985 and IEEE 854-1987 Floating Point Representation

5.1 C Standards

5.1.1 Calling Conventions

As TRIPS does not support operations on 32-bit IEEE single precision floating point values, single precision floating point values are always passed as double precision arguments to called subroutines. See Section 3.3.2.2 of the ANSI^{TM} X3.159-1989 standard and Sections 6.5.2.2 and 6.9.1 of the ISO/IEC 9899 standard.

5.2 F77 Standards

The TRIPS compiler does not support the “assigned goto” capability as specified in Section 11.3 of the ANSI^{TM} X3.9-1978 standard.

5.3 Floating Point Representation

See the “TRIPS Processor Architecture Manual: Version 1.2: Tech Report TR-05-19 (03/10/05)” for information on this subject.