TRIPS Assembler and Assembly Language Specification

TRIPS Internal Memo – 03

September 24, 2003 - Version 2.1D

1 Introduction

The TRIPS Assembler (tas) is a standalone C application whose job is to consume TRIPS Assembly Language or *TASL* files and to produce TRIPS Object Format or *TOFF* files. The TRIPS assembler is derived from the GNU family of software tools.

This document is intended for TRIPS compiler writers and others who wish to understand the translation of source assembly language to TRIPS binary format.

When the TRIPS compiler finishes generating hyperblocks and writes out the TRIPS Intermediate Language (*TIL*), it will in turn invoke the TRIPS instruction scheduler (tsch), which will embed all necessary scheduling information in the TASL file. Consequently, the role of the assembler is simply to translate the assembly language statements and assembler directives in the TASL file directly into TOFF binary format.

This document is intended for compiler writers, hardware designers, and others needing to write and test assembly language programs for TRIPS simulators and execution environments.

1.1 Links in the Chain

The TRIPS toolchain comprises several components, all available on i86 Linux platforms:

Tool	Description	Input Files	Related Documentation
tcc	**The Scale java-based C compiler and wrapper**	**C/FORTRAN (*.c, *.f77).** Support for C++ and Java is deferred.	*TRIPS Processor Architecture Manual* *Scale Website* *TRIPS Application Binary Interface Manual*
tsch	**TRIPS Scheduler and wrapper**	**TRIPS Intermediate Language Files (*.til)**	*TRIPS Intermediate Language Version Two (TILT) Specification* *TRIPS Scheduler Manual*
tas	**TRIPS Assembler**	**TRIPS Assembly Language Files (*.s)**	*TRIPS Assembly Language and Assembler Specification (this document)* *TRIPS Processor Architecture Manual*
tld	**TRIPS Linker**	**TRIPS Object Files and Static Libraries (*.o, *.a)**	*TOFF Object File Format Specification*

Table 1: TRIPS Toolchain Components

1.2 Accessory Components

A variety of other components and utilities are available to test and supplement the toolchain. Here is a summary:

- The TRIPS timing simulator (tsim) provides cycle-accurate execution of TRIPS binaries.
- The TRIPS functional emulator (tem) provides simple instruction-level emulation of TRIPS binaries. Refer to the *TRIPS Functional Emulator User Guide* and *The Binary File Descriptor Library (libbfd)* documentation for more information.
- The TIL emulator (tile) loads and interprets TRIPS Intermediate Language code to ensure compiler output correctness.
- GNU utilities, such as nm and readelf, provide additional support for code development and debugging.
- The software code metrics tool (dynamic.pl) helps to validate the quality and performance of the code produced by the TRIPS toolchain. Refer to

http://www.cs.utexas.edu/users/cart/trips/internal/testing/metrics/index.html for more information.

1.3 Related Documentation

In addition to the documentation listed in Table 1 above, the following are useful for understanding the operation of the TRIPS assembler:

- Version 2.10 of the online manual, *Using as* (http://sources.redhat.com/binutils/docs-2.10/as.html). This Red Hat version supersedes the 2.9 version maintained at the Free Software Foundation (http://www.gnu.org/manual/gas-2.9.1).
- Version 2.12.90 of the as man page, from Version 2.12.90 of the GNU `binutils` source package, under `gas/doc/as.1`.
- Other binutils man pages, such as `readelf(1)`, `nm(1)`, and `objdump(1)`, available on most Linux and GNU installations.
- The GNU assembler internals document, under `gas/doc/internals.texi`, accessible on the TRIPS internal Web site at:

  http://www.cs.utexas.edu/users/cart/trips/internal/infra_tools/TASM/internals.pdf
- *A Design Space Exploration of Grid Processor Architectures*, available at http://www.cs.utexas.edu/users/cart/trips/internal/arch/Micro34/micro01-public.pdf .
- The Scale Compiler Group user documentation, available at: http://www-ali.cs.umass.edu/Scale.
- *TRIPS Project Definition of Terms*, available at http://www.cs.utexas.edu/users/cart/trips/internal/program_UT/definitions.pdf.

Using as, listed above, will be frequently cited in this specification.

1.4 Bug Reporting

Please note that the TRIPS toolchain in general and the TRIPS assembler in particular are works in progress. If you find a discrepancy between this specification and the operation of the assembler, a defect in assembler operation, or incomplete or incorrect documentation, please file a bug against the "Assembler (tas)" toolchain component in the TRIPS Bugzilla bug tracking system:

http://lockhart.csres.utexas.edu/bugs/

1.5 Overview

The TRIPS assembler produces code for a VLIW-like machine with data flow characteristics. As opposed to parsing traditional *operand* syntax of the form:

```
opcode <target> <operand1> <operand2>
```

—the TRIPS assembler expects *target* syntax of the form:

```
<source_element> <opcode> <target1> <target2>
```

—in which the `<source_element>` represents an architecture register or execution node that operates upon one or two implicit *input operands* using the specified `<opcode>` to

produce a single *output value* to forward to one or more target execution nodes, to an architecture register, or to the memory system.

Example:

```
N[5] add N[10,1] N[63,0]
```

This example causes execution node 5 to add its two (unspecified) input operands and to forward the resulting value to two target execution nodes, numbers 10 and 63. The second *operand slot* parameter (1 or 0) causes the result to be placed in the right slot or left slot of the targets, respectively.

1.5.1 Source Elements

Whether it's a general register or an execution node, each instruction requires a *source element*. Otherwise, there is no agent to execute the opcode and produce an output value, whether it's an add, load, or other type of instruction. Syntactically, a source element appears as the first item in a TRIPS instruction.

Note that general registers can appear as the source for two special types of instructions only—the `read` and `write` instructions—by which the contents of that general register are inserted into one or more target nodes within the grid or output to a general register, respectively.

Refer to section 4, "TRIPS-Dependent Features" for more information on node specification and TASL syntax.

Because a grid processor by definition is a multi-dimensional piece of hardware, the TRIPS toolchain knows how to map the logical names of general registers (e.g., `R[28]`) and of execution nodes (e.g., `N[67]`) onto physical 2D and 3D locations. In terms of the software interface, however, both general register and execution node names are *flat*, ranging by default from 0..127 for both.

1.5.2 Sequence Numbers

Although the TRIPS hardware can execute its instructions in parallel and out-of-order, the need arises on occasion to execute those same instructions serially and in order. Hence, TASL syntax supports sequence numbers associated with instructions. Sequence numbers can range from *1* to the total available of instruction slots per block; that is, the upper limit is *grid_width * grid_height * number_of_physical_frames* per processor "slot".

Sequence numbers are enclosed by angle-brackets (<>) immediately after the source node name.

Examples:

```
N[12]-<003> genu %low(target_block) N[19]; sequence number is 3
N[15]-<006> or  N[23,0]; sequence number is 6
```

Notes on sequence numbers:

- Sequence numbers occur on a per-block basis. They can appear in any order within the block but must specify the total number of instructions from 1 to *num_instructions* without duplicates or gaps within a block. The assembler will flag duplicates and missing numbers.

- All instruction blocks include sequence numbers, whether they are supplied

explicitly in the code or implicitly by the assembler. If supplied implicitly, numbering begins with 1 at the first instruction in the block and increments from first to last (top to bottom) in program order.

- By default, the assembler generates a sequence number "chunk" for each block in a module. To generate code without sequence number chunks, choose either or both:
    - Invoke the assembler with the `-s` command line switch. **Example:** `tas -o file.o -s file.s`
    - Include the `.sequence_numbers off` assembler pseudo-op at the beginning of a source file.
- Leading zeros are optional in the sequence number. E.g., both `<3>` and `<003>` specify the same sequence number.
- Read and write instructions do not require sequence instruction numbers. In serializing the execution of read instructions, the hardware will process the instructions in the order found on the disk, which is the same order as that within a block of assembly source code. The assembler will note but ignore any sequence number associated with a `read` or `write` instruction.

1.5.3 Assembly Time

In assembling instructions, tas builds a machine word from an individual instruction in the assembly language file, one machine word per line of instruction, using the binary encodings specified in the ISA Manual. This machine word may be a 32- or a 64-bit quantity, depending on whether the assembler is set to produce *prototype* or *extended format* object code. After verifying that the source node is a legitimate specifier and after performing a variety of range- and error-checking, tas inserts this binary encoded word into a virtual block structure according to its row, column, and frame coordinates. When a block of source code ends due to the `.bend` directive, tas writes the entire block to disk with the words arranged by column within a given frame and then by row.

It may be obvious by now that by the time tas receives its assembly source file, things are pretty tightly bound to a particular machine configuration—the number of rows and columns in a grid, the frame depth, etc. Although a number of configuration parameters are available through the command line and configuration file, tas won't produce meaningful object code for a given processor unless the scheduler hands the assembler meaningful source code for that same processor.

2 Tool Invocation and Command Line Options

The TRIPS assembler is invoked as a Linux-based command line tool. Its command line syntax is derived from gas, the GNU assembler:

tas [*option…*] [*asmfile…*], where the various *options* are described below.

Multiple source file names can be included on the command line and the assembler will process them one at a time. If no source file is specified, then the assembler reads from standard input. By default, the assembler names the output file `t.out`, even though at that point, the file hasn't been linked nor its references fully resolved.

2.1 GNU Options

Command line options derived from the GNU assembler include the following:

`-a[lmns=file]` Turn on print listings, in any of a variety of ways. **Examples**:

`-al` include assembly
`-am` include macro expansions
`-as` include symbols
`=file` set the name of the listing file

You may combine these options; for example, use `-as=symbolf` for assembly listing of symbols directed to file `symbolf`. The `=file` option, if used, must be the last one. By itself, `-a` defaults to `-als`.

Note that the assembly listing will print out the assembled instructions (that is, the hexadecimal codes) in somewhat random fashion, because the assembly listing occurs on an instruction by instruction basis, whereas the assembly process itself occurs on a row by row basis, with multiple instructions per row.

`--defsym` *`sym=value`*

Define the symbol *sym* to be *value* before assembling the input file. *value* must be an integer constant. As usual, a leading `0x` indicates a hexadecimal value, and a leading `0` indicates an octal value.

Example:

```
./tas --defsym SP=0xFFFF0 set.s ; sets the value of
                                ; SP to hexadecimal 0xFFFF0
```

`-I` *`dir`*

Add directory *dir* to the search list for `.include` directives.

-o *`filename`*

Name the output file *`filename`*.

There is always one object file output when you run . By default, it has the name `t.out`. Whatever the object file is called, tas first overwrites any existing file of the same name.

`-L` or `--keep-locals`

Preserve the local symbols from a module in the symbol table of the final executable.

`--statistics`

The `--statistics` switch causes tas to output information about the number and type of sections and relocations created and assembly time.

`--strip-local-absolute`

Remove local absolute symbols from the outgoing symbol table.

`-v`

Print the `as` version at the beginning of the assembly process.

`--version`

Print the `as` version and exit.

`-W` Suppress warning messages. Usually, not a good idea.

`-Z` Generate an object file even after errors. Usually, not a good idea.

Refer also to the online manual, *Using as*, for other options.

2.2 TRIPS-Specific Command Line Options

The following options generate binary code that is targeted for a particular TRIPS processor configuration:

`-D` Turn debugging information so that the assembler will output for each block (a) the header information, (b) the grid instruction format, and (c) the sequence number chunk, if one is output.

`-E` Turn on extended debugging, so that block names and miscellaneous other information are output during assembly.

`-m <RowxColumn>` Assemble for a given TRIPS grid configuration, where the Row and Column parameters must be a power of two, ranging from 2 to 32.

Examples:

`-m2x2` Assemble for a 2x2 grid

`-m4x4` Assemble for a 4x4 grid

`-m4x16` Assemble for an 8x8 grid.

Default is 4x4.

Note that the grid configuration doesn't necessarily have to be square.

`-p <num>` Assemble for a given number of P-frames per A-frame in the target processor, where `<num>`, or the number of P-frames/A-frame, can range from 1 to 256. The default is 8.

`-s` Do not include sequence number information. By default, tas includes sequence number chunks for a given source file in the output file.

The assembler will do the math to ensure consistent hardware parameters and flag mismatches and out-of-range conditions.

Note: The TRIPS linker (tld) will expect all `.o` files that are input for a given link job to be assembled for the same hardware configuration. For example, it is not possible to link an object module assembled for an 8x8x2 row/column/frame configuration with an object module assembled for a 4x4x8 hardware configuration.

2.3 The TRIPS Toolchain Configuration File

The TRIPS toolchain configuration file is an ASCII text file that specifies individual TRIPS hardware configuration parameters. Each tool in the toolchain consults this file for hardware descriptions. Its location is determined by the `TC_CONFIGPATH` environment variable, which is set by default to these locations which are searched in order:

- The current working directory (`.`)
- The user's `$HOME` directory
- `/projects/trips/toolchain/current/etc/trips`
- `/projects/trips/toolchain/stable/etc/trips`

The assembler consults the file for the value of certain key variables, whose defaults are expressed in C syntax:

```
#define GRID_HEIGHT            4
#define GRID_WIDTH             4
#define PFRAMES_PER_AFRAME     8
#define ISA_VERSION           20
#define TSCH_VERSION           2
#define INSN_SEQUENCE_NUMBERS  1
```

Note that command line options take precedence over the values in this file. For example, if

`tcconfig.txt` specifies the value of `INSN_SEQUENCE_NUMBERS` is 1 (or "true") but the user specifies `tas -s` on the command line, the assembler will be configured *not* to output sequence numbers.

2.4 Using the Assembler

The TRIPS assembler is built nightly on weekdays and can be found in the `/projects/trips/toolchain/builds/trelease/latest_isa_proto` disk directory at UTCS. In particular, when the build is successful, the following executables are copied to this release directory:

```
ar  nm  objcopy  objdump  ranlib  readelf  size  strings  strip  tas  tld
```

GNU systems typically include `man` pages for these commands, including `as(1)` and `ld(1)` for the tas and tld commands, respectively.

Also under the `trelease` directory, there is a small sample of TASL files:

`/projects/trips/toolchain/builds/trelease/latest_isa_proto/testsuite/*.s`

When running the assembler, you may find the `-D` debug option helpful in observing assembler code output. Otherwise, if all goes well, tas performs its work silently.

Another useful option is `-a` to create an assembly listing on standard output at the end of the run.

Example:

```
% ./tas -a test1.s
```

Use the `tas -help` command line option to view TRIPS-specific extensions to the standard GNU assembler options.

3 Assembly Language Syntax

This section describes the syntax of the TRIPS Assembly Language. Much of this material is lifted directly from the GNU documentation.

3.1 Comments

The tas line comment character is a semi-colon (`;`). Note that you can use a semi-colon on a line by itself or in-line, following an instruction. C-Style comments are also allowed.

Example:

```
/*
  Include arbitrarily long
        multi-line
  comments in this fashion.
*/
```

3.2 Placement Within a Source Line

A comment, instruction, or label can begin anywhere on a new line—whether in column 0 or column <N>, preceded by tabs and spaces. For example, the parser treats these source statements identically:

```
string1:
                  string1:
                                        string1:
```

A source line ends with a newline character. Line wrapping is disallowed.

3.3 Symbols and Labels

A *symbol* is one or more characters chosen from the set of all letters (both upper and lower case), digits, and the three characters underscore, dot, and dollar sign (`_.$`). No symbol may begin with a digit. Case is significant. There is no length limit: all characters are significant.

A symbol can be given an arbitrary value by writing the symbol name, followed by an equals sign (=), followed by an expression (see section Expressions).

Examples:

```
all_ones=0xFFFFFFFF

.set all_ones= 0xFFFFFFFF
```

Note the difference between a *symbol* and a *label*. A *symbol* is the name of a numeric value, such as a memory address, whether in a text, data, bss, or absolute section of a binary file. A symbol can be used to reference either code or data. A *label* is more specific and *always* represents a memory address, such as the address of the first character in a string.

For text, data, and bss sections, the value of a symbol changes as the linker changes section base addresses during linking. By definition, absolute symbols' values do not change during linking.

Example:

```
p3 = 0xbbbb ; the value of p3 will stay fixed regardless
            ; any relocation that may occur
```

Use the `objdump -t <obj_file>` command to examine the symbol table of an object module or executable.

Note that within a text segment (`.text`) code addresses are available only at block beginnings. That is, `.bbegin` assembler directive sets the address of the whole block to the first code word following the directive. This is the address that the hardware uses to fetch and execute instruction blocks.

3.4 Expressions

An *expression* specifies an address or a numeric value. White space may precede and follow an expression. The result of an expression must be an absolute number or else an offset into a particular section.

3.4.1 Arguments

Arguments can be symbols, numbers, or *subexpressions* that are built up out of the first two items. A subexpression consists of a left parenthesis (`(`) followed by an integer expression, followed by a right parenthesis (`)`); or else a prefix operator followed by an argument.

Example:

```
.set soffset= (_start + 0x1000)
```

The statement causes symbol `soffset` to be set to the absolute value of the subexpression consisting of `_start` plus hexadecimal 1000.

3.4.2 Operators

Operators are arithmetic functions, like + or %. Prefix operators are followed by an argument. Infix operators appear between their arguments. Operators may be preceded and followed by white space.

3.4.3 Prefix Operator

`tas` offers the following *prefix operators*. They each take just one argument:.

- *Negation*. Two's complement negation.
~ *Complementation*. Bitwise *not*.

Example:

```
p5 = ~0xFF
```

3.4.4 Infix Operators

Infix operators take two arguments, one on either side. Operators have precedence, but operations with equal precedence are performed left to right. Apart from + or -, both arguments must be absolute, and the result is absolute.

Highest Precedence

`*` Multiplication.
`/` Division. Integer division with truncation: same as the C operator `/`
`%` Remainder or modulus operator.
`<<` Shift Left. Same as the C operator <<.
`>>` Shift Right. Same as the C operator >>.

Intermediate precedence

`|` Bitwise Inclusive Or.
`&` Bitwise And.
`^` Bitwise Exclusive Or.
`!` Bitwise Or Not.

Lowest Precedence

`+` Addition.
`-` Subtraction.

3.5 Assembler Directives

GNU offers *dozens* of assembler pseudo-operators or *directives* to control the processing of source files, everything from `.eject,` which forces a page break during an assembly listing, to `.byte`, which sets the current location in the data section to a given 8-bit value.

3.5.1 Useful GNU Assembler Directives

Following is a sampling of the more useful TRIPS assembler directives:

Directive	Description	Example

`.align` *num_bytes*	Align the next data item to the alignment specified by *num_bytes*.	`.align 8`
		`_tstring:`
		`.ascii "hello"`

`.ascii` (`.asciz`)	Define an ASCII null-terminated string.	`.asciz "Hello world!"`
`.data`	Begin a program's writable data section.	`.data`
`.equ` *symbol= expression*	Set the value of symbol to expression, same as `.set` below.	`.equ fibon=main`
`.fill` *repeat, size, value*	Fill up data section with *repeat* copies of *size* bytes with *value* contents. (Be careful of unexpected behavior. Refer to the as man page for more information.)	`.fill 20, 8, 0`
`.global`	Make a symbol visible to the linker.	`.global A` `.bbegin A`
`.lcomm` *symbol, length, align*	Reserve the *length* number of bytes for a given *symbol* in the program's BSS space and align it on *align* bytes.	`.lcomm a_b, 400000, 4`

`.rdata`	Begin a program's read-only data section.	`.rdata`
		`.align 8`
		`my_array:`
		`.int 0x1`
		`.int 0x2`

`.set` *symbol= expression*	Set the value of a memory location to an absolute value.	`.set EXIT_ID=32`

`.text`	Begin a program's text (or code) section.	`.text`
		`.global main`
		`.bbegin main`

Table 2: Useful Assembler Directives

Refer also to the "Data Types" subsection following for the key assembler directives that are used to allocate space for basic TRIPS data types.

Here are additional notes for the above directives:

`.global` – By default, tas creates symbols local to the source module and hidden from other modules. This scoping includes block names. To make a symbol visible to other modules, include the `.global` directive before or after the label for that symbol.

`.lcomm` – The name "local common" doesn't mean much, but the directive provides an important mechanism for allocating space for a symbol without having to declare what goes

in that space. Such variables are collected in the program's BSS section for uninitialized data. With no `align` parameter specified, symbols are aligned on 8-byte boundaries.

`.rdata` – Use an `.rdata` directive when you want to initialize memory values at program load time and don't want them to change for the duration of the program.

`.set` – Unlike off-the-shelf GNU assemblers, tas uses an equals sign (=) in the .set directive. The `symbol` will acquire the value of the `expression` as an absolute value that is fixed for program duration. You can use `.set` (or equivalently, an `.equ` directive) to ensure that two distinct labels have the same value. In FORTRAN, for example, a program needs both the program's entry point *and* the application label to reference the same memory location (the start of the beginning program block), and the `.set` directive allows you to equate the two symbol names.

You can use the `.text`, `.data`, `.rdata`, and `.lcomm` directives in any order within a given source module and include them multiple times. The assembler will coalesce the output code into single, unified sections.

Note that the `.extern` keyword appears in the "Other Directives" table below for compatibility, but it is ignored: tas treats all undefined symbols as external. The linker will automatically attempt to resolve undefined symbols at link time.

3.5.2 Other GNU Assembler Directives

Following is a listing of numerous GNU directives, some simple, some complex, and most not needed by the TRIPS toolchain.

GNU Directive	GNU Directive
.balign[wl] abs-expr, abs-expr, abs-expr	.org new-lc , fill
.comm symbol , length	.p2align[wl] abs-expr, abs-expr, abs-expr
.desc symbol, abs-expression	.print "string"
.eject	.psize lines, columns
.end	.purgem name
.equiv symbol, expression	.rept count
.extern	.sbttl "subheading"
.err	.section name [, subsection]
.fail expression	.skip size, fill
.file string	.sleb128 expressions
.fill repeat, size, value	.space size, fill
.func name[,label], .endfunc	.stabd, .stabn, .stabs

GNU Directive	GNU Directive
.if absolute_expression, .else, .elseif, .endif	.string "str"
.include "file"	.struct expression
.irp symbol,values.	.symver name, name2@nodename
.irpc symbol,values	.text subsection
.list, .nolist	.title "heading"
.macro name args... , .exitm	.uleb128 expressions
.octa bignums	

Table 3: Other Assembler Directives

Refer to the "Related Documentation" section for a pointer to more information.

4 TRIPS-Dependent Features

Most of the syntactical features in the preceding section, "Assembly Language Syntax," pertain to the whole family of GNU-based assemblers. Following are TRIPS-dependent features.

4.1 Program and Block Structure

As described in the *TRIPS Object File Format (TOFF) Reference*, the TRIPS assembler and linker produce ELF object files that consist of header, text, and data sections. The "Assembler Directives" subsection above indicates how tas supports the `.text`, `.data`, `.rdata`, and `.lcomm` directives to create and organize TRIPS program sections. Within a `.text` section, a TRIPS program consists of zero or more grid instruction blocks, delimited by `.bbegin/.bend` pairs.

Grid instruction blocks are themselves organized into the following instruction groupings.

- **General register read preamble:** The preamble consists of specialized `read` instructions that inject values from the architecture registers into the grid of execution nodes.
- **Grid instructions:** This group makes up the main body of the block and consists of conventional operators for ALU and memory operations that execute in out-of-order fashion.
- **General register write epilogue:** The epilogue consists of specialized `write` instructions that commit values from the execution nodes onto the architecture registers.

4.2 TRIPS-Specific Assembler Directives

To support the TRIPS execution model, the following TRIPS-specific pseudo-ops supplement the standard GNU assembler directives:

Directive	Description	Usage
`.bbegin <block_name>`	Block begin	Following this directive, assembly statements are translated into machine words within a TRIPS instruction block.
`.sequence_numbers [on | off ]`	Turn sequence numbers `on` or `off`.	If used, this directive must appear at the beginning of the source file. It instructs tas to emit (`on`) or not to emit (`off`) a sequence number chunk after each grid instruction block. Default is `on`.
`.bend [<block_name>]`	Block end	Paired with a predecessor `.bbegin` directive, `.bend` marks the end of one complete instruction block.

Table 4: TRIPS-Specific Assembler Directives

The `.bbegin` directive causes the assembler to allocate memory space for a block header chunk, a grid instruction block, and a sequence number table. From then, `read` and `write` instructions are parsed and collected in 32-bit word quantities to form the complete 2D header chunk in memory. Other grid instructions are parsed and collected into 32-bit word quantities to form the 3D grid instruction block. When it encounters a `.bend` directive, the assembler flushes the current header chunk, the 3D instruction block, and the (optional) sequence number table in translated binary form to the output file.

If you don't specify a `<block_name>`, the assembler will issue a warning and default the name to "`unspecified`". If you don't include a `.bend` directive before starting a new block, the assembler will issue a warning, but close the block off and flush the instructions before beginning a new block.

You don't have to include a `<block_name>` in the `.bend` directive, but if you do, it should match the `<block_name>` given in the preceding `.bbegin` statement. Otherwise, a warning message is issued and the assembler uses the original `<block_name>`.

The `.sequence_numbers` directive is provided as a convenience only. You can instead use the command line `-s` option or the `INSN_SEQUENCE_NUMBERS` field in the toolchain configuration file to control the generation of sequence number chunks. A `.sequence_numbers` directive must appear before any `.bbegin` directive in a file. If present, it will override the setting of the `INSN_SEQUENCE_NUMBERS` field in the toolchain configuration file and will itself be overridden by a `-s` command option switch.

Note: The TRIPS linker (tld) will expect all object files that are input for a given link job to be assembled for the same sequence number configuration. It is not possible to link an object module assembled with sequence numbers with another module that doesn't include sequence numbers.

4.3 Data Types

The TRIPS assembler supports six primary scalar data types as listed below:

Bit size	TAS Directive	ISA Usage
8	`.byte`	*byte*
16	`.short`	*halfword*
32	`.int`	*word*
64	`.quad`	*doubleword*
32	`.single`	*single-precision floating-point*
64	`.double`	*double precision floating-point*

Table 5: Basic Data Types

Note that four other GNU data directives `.long`, `.hword`, `.word`, and `.float` are specifically omitted to avoid confusion with their application in other processor families. Their use is not supported in TASL source code.

Examples:

```
.data
.align 8            ; move to an 8-byte boundary in the data section,
july: .int 0x601d   ; reserve 4 bytes to hold 0x601d, and name that
                    ; location "july"

august: .int 0xf01d   ; reserve 4 bytes for the "august" symbol, and
                      ; initialize it to 0xf01d.  A second align is
                      ; unnecessary here, as the two ints will fit
                      ; into one 8-byte TRIPS word.

.lcomm summer, 8, 4 ; move to a 4-byte boundary in the BSS section
                    ; and reserve 8 bytes for storage location "summer".
                    ; The third parameter *must* be included in the .lcomm
```

When aligning and initializing a global variable, the Compiler backend should first output the alignment information and then the global information.

Example:

```
.align 4
.global coffee
coffee: .int 0x600d
```

Refer to the *TRIPS Application Binary Interface Manual* for toolchain conventions used in supporting aggregate data types, such as `structs` and `unions`, and for packing and aligning data types.

4.4 Byte Ordering

Like other popular microprocessors, TRIPS uses a Big-Endian organization for arranging bytes within a machine word so that the address of a variable is the address of its most significant byte. For example, when aligned on 8-byte boundaries, TRIPS integer data types will be located in memory as follows:

```
a: .byte  0x1
0x0100000000000000

b: .short  0x3
0x0003000000000000

c: .int    0x7
0x0000000700000000

d: .quad    0xf
0x000000000000000f
```

Note that the TRIPS ISA provides a number of instructions (`LB`, `SB`, `LH`, `SH`, etc.) to make loading and storing partial words fast and convenient.

4.5 TRIPS Assembler Syntax

As specified in the *TRIPS Processor Architecture Manual,* the TRIPS architecture supports both *grid instruction formats* and *general register formats*. The following subsections describe, first, the grid instruction formats that make up the basic execution "chunks" and, second, the general register instruction formats that make up the preamble and epilogue portions of the complete instruction blocks.

4.5.1 Grid Instruction Formats

Use the following syntax to specify an instruction that executes on one of the grid execution nodes:

ISA Insn Class	TASL Syntax
G	N[*source_node*] *opcode* N[*target, slot*]
G	N[*source_node*] *opcode* N[*target, slot*] N[*target, slot*]
I	N[*source_node*] *opcode immediate_value* N[*target, slot*]
L	N[*source_node*] *load_opcode* I[*LSID*] *immediate_value* N[*target, slot*]
L	N[*source_node*] *load_opcode* I[*LSID*] N[*target, slot*]

ISA Insn Class	TASL Syntax
S	N[*source_node*] *store_opcode* I[*LSID*] *immediate_value*
S	N[*source_node*] *store_opcode* I[*LSID*]
B	N[*source_node*] *branch_opcode* I[*EXIT_ID*] *branch_offset*
B	N[*source_node*] *branch_opcode* I[*EXIT_ID*]
C	N[*source_node*] *constant_opcode big_immediate_value* N[*target*, *slot*]

Table 6: Grid Instruction Formats

An execution node specifier `N` and instruction identifier `I` can be entered in uppercase or lowercase letters. The spacing between fields can include an arbitrary number of blanks and tabs. Within fields (e.g., *target* and *immediate_value*), whitespace is not allowed.

Within a grid instruction, a register *write queue* target may instead be specified instead of an *execution node* target. The syntax for a write queue target is simply W[*general_register*] with no *slot* included.

Example:

```
N[23] fmul W[30] N[30,1]
```

This syntax instructs source node 23 to multiply its two implicit floating point operands and write the floating point result to general register 30 in the write queue and also forward the result to the right operand slot of execution node 30. Refer also to the "General Register Instruction Formats" subsection following.

Note that all of these instructions may include an optional <*sequence_number*>, enclosed in angle brackets (<>) after the N[*source_node*] and before the *opcode*.

Example:

```
N[23] <003> fmul W[30] N[30,1]; indicates that this is the third
                               ; instruction in program order
```

See also the "Sequence Numbers" subsection in the "Overview" section above.

4.5.1.1 Field Descriptions

In the following descriptions, we assume a 4x4x8 hardware configuration and characteristics of the hardware prototype.

Name	Description
big_immediate_value	An expression that evaluates to a signed or unsigned value known at link time. See the "Expressions" and "Bit Extraction Field Operators" subsections for more information. On the

Name	Description
	prototype, immediate values must be contained within 16 bits.
branch_offset	An expression that specifies an offset from the start of the currently executing block, in the positive or negative direction, to the start of a block header "chunk". As specified in the prototype ISA manual, this field encodes a 20-bit signed value, that "is always treated as a chunk offset, rather than a byte offset."
branch_opcode	An instruction that changes the flow of control from the currently executing block to the beginning of the same or another block. On the prototype, can be `BR, BRO, CALL, CALLO, RET`, or `SCALL`.
constant_opcode	An instruction that contains an embedded constant value. On the prototype, can be `GENS, GENU, APP`, or `NOP`.
I[EXIT_ID]	The *EXIT_ID* is an expression that evaluates to a small positive expression unique to all the branch (type B) instructions within the same block. On the prototype, values may range from 0..7.
I[LSID]	The *LSID* is an expression that evaluates to a small positive expression unique to all the loads and stores within the same block. On the prototype, values may range from 0..31.
immediate_value	An expression that evaluates to a signed or unsigned value known at link time. See the "Expressions" and "Bit Extraction Field Operators" subsections for more information. On the prototype, immediate values must be contained within 9 bits.
load_opcode	An instruction that loads a value from a given memory address to a given operand slot on an execution node. On the prototype, opcodes can be `LB, LH, LW,` or `LD`.
N[source_node]	One of the execution nodes on the grid. For the prototype, the *source_node* name may range from `0..127`.
N[target, slot]	The `target` for the result of the operation, ranging on the prototype from 0..127. The `slot` can be `0` or `1` to specify the left and right operand slot, respectively, or `p` to specify the predicate slot of the target execution node.
opcode	One of the opcodes belonging to the General instruction class (e.g., `add`) or Immediate instruction class (e.g., `addi`).
store_opcode	An instruction that stores a value generated by an execution node to a given memory address. On the prototype, can be `SB, SH, SW`, or `SD`.

Table 7: Fields Within Grid Instructions

4.5.1.2 Expressing Predication

All TRIPS instructions except those in the Constant class (type C) can be predicated upon a *true* or *false*[1] condition.

Use a `_t` or `_f` suffix to indicate that an instruction requires a predicate value. **Example:**

```
sb_t
```

This instruction behaves as follows: if the incoming predicate value is true, store the low-order byte from input operand 2 in the memory address specified by input operand 1.

All TRIPS grid instructions that generate a target (class G, I, L, and C) can output a value to the predicate slot of another execution node. Common usage occurs in integer and floating point test instructions (class G), in which the firing node sends the comparison result to the predicate slot of a target execution node.

Use the `N[`*`target`*`, p]` syntax, with an uppercase or lowercase `p`, to specify a predicate output target.

Example:

```
teq N[18,p]
```

This instruction behaves as follows: if input operand 1 equals input operand 2, send a "1" to the predicate slot of execution node 18; else, send a "0".

Note that there's no implied relationship between a *predicated* instruction (one that requires a true or false predicate value) and a *predicating* instruction (one that produces a predicate operand for another execution node).

4.5.1.3 Other Grid Instruction Notes

- The `N[source_node]` specifier must appear first on the instruction line.
- The second component of a target execution node—for example, the `1` in `N[20,1]` specifies the operand slot for the result of the operation. Use a zero (`0`) to specify the left-hand slot, a one (`1`) to specify the right-hand operand slot, and an uppercase or lowercase `p` to specify the predicate slot. The `, slot` parameter is optional; it defaults to `0` (left-hand operand slot).
- The maximum number of instructions that can be scheduled in a given block is determined by the presumed hardware configuration. No more than *N* grid instructions can be scheduled in a block, where *N* is *number_of_rows* * *number_of_columns* * *number_of_frames*. On the prototype, the maximum number of grid instructions is 4x4x8 or 128 instructions.
- The *`LSID`* load/store identifier is optional. If no `I[`*`LSID`*`]` field appears in a load/store instruction, the assembler will generate an identifier based on the location of the instruction within the assembly code. Default identifiers are numbered

[1] A *true* predicate value has its least significant bit set to `1`. A *false* predicate value has its least significant bit cleared.

starting at `0`, from top to bottom of the grid instruction block. It's not a good idea to provide load/store identifiers for some load/store instructions within a block and not for others.

- Similarly, the *`EXIT_ID`* branch identifier is optional. If no `I[`*`EXIT_ID`*`]` field appears in a B-type instruction, the assembler will generate an identifier based on the location of the instruction within the assembly code. Default identifiers are numbered starting at `0`, from top to bottom of the grid instruction block.

- The scheduler can but doesn't have to output `nop` instructions. The assembler will fill them in automatically for unused execution nodes.

- A number of architecture registers are reserved for stack pointer, etc. Refer to the *TRIPS Application Binary Interface Manual* for information on which registers are used for what purpose.

- If the assembler notices a discrepancy between the grid configuration and the schedule--for example, if the instruction would be mapped to an already scheduled execution node--then the assembler will note the error and not generate an object file.

4.5.1.4 Immediate Values

On the prototype, you can specify either a 9-bit, 16-bit, or a 20-bit value for encoding in an immediate (I), constant (C), and branching (B) instruction, respectively. You can use decimal notation (e.g., `42`) or hexadecimal notation (e.g., `0x601d`) for these immediate values or any *legal GNU assembler expression*. Refer to the "Expressions" subsection above for more information about GNU expression syntax.

The *bit extraction field operators* described below also allow you to capture "pieces" of data values and addresses in the bit fields of constant instructions.

4.5.1.5 Bit Extraction Field Operators

The TRIPS assembler offers the following bit extraction operators (sometimes called "percent operators") as follows:

Operator Syntax	Extracted Bit Fields
`%hi(`*`name`*`)`	63-48
`%mid(`*`name`*`)`	47-32
`%lo(`*`name`*`)`	31-16
`%bottom(`*`name`*`)`	15-0

Table 8 Assembler Percent Operators for Bit Extration

Used with the `GENU`, `GENS`, and `APP` instruction, these field operators enable your assembly code to extract values from their symbolic names. For the prototype, these operators extract 16-bits at a time, either as unsigned values (`GENU`) or signed values (`GENS`).

Example:

```
N[3] gens %hi(big_symbolic_value) N[4,1]
```

This instruction extracts the 16 most significant bits from a 64-bit `big_symbolic_value`, sign-extends them as necessary, and outputs the sign-extended value to the right operand slot of execution node `4`.

Example:

```
N[3] app %lo(moderate_symbolic_value) N[4,1]
```

This APPEND CONSTANT instruction captures bits 31-16 of the value specified by `moderate_symbolic_value`, with no sign-extension, ORs them with the left-shifted value of its left input operand and outputs them to the right operand slot of execution node `4`.

Refer to the "Assembly Language Sample" following to see how the percent operators can be used in combination with C-type instructions to generate memory addresses efficiently.

4.5.1.6 Address Offsets

To support the CALL WITH OFFSET and CALL WITH BRANCH instructions, the assembler offers the following syntax:

```
bro target_block_name

callo target_block_name
```

--in which *`target_block_name`* denotes the name of a grid instruction block, either internal or external to the current source module.

The assembler will mark such instructions as “fix-ups” and send their relocatable information to the linker, which will perform the following tasks during its final pass:

- Determine the ultimate resolved address of the *`target_block_name`*.
- Compute that address in terms of TRIPS “chunks” such that the least significant 6 bits are zeroed and right-shifted to fit the constant field width field of the `bro` or `callo` instruction.
- Re-insert the fixed instruction into the block of grid instructions.

If the computed address exceeds the reach of the constant width field (a signed 20 bits in the prototype), the linker will issue an error message and exit. Hence, the compiler must use relative offsets conservatively, to ensure that they are within the span of the current instruction block.

4.5.2 General Register Instruction Formats

Each instruction block can begin with a general register read preamble and end with a general register write preamble. General register instructions use register specifiers as follows:

TASL Syntax	Description
`R[read_queue_id]`	One of the registers in read queue whose value will be injected into the instruction grid.
`W[write_queue_id]`	One of the registers in the write queue whose value will be output from an execution node.
`G[architectural_register_id]`	One of the general registers whose value will persist across instruction blocks

Table 9: Syntax for Accessing TRIPS General Registers

The prototype supports a maximum of 32 reads per block through read queue registers `R[0]..R[31]`. The prototype supports a maximum of 32 writes per block onto write queue registers `W[0]..W[31]`. Overall, there are 128 architecture registers whose values persist across blocks, named `G[0]..G[127]`.

4.5.2.1 General Register Access

Programs access the TRIPS architecture registers, also referred to as *general registers*, through one level of indirection:

- There is one *read queue* and one *write queue* associated with each block of instructions, enabling the instructions in that block to read from and write to the general register banks. Read queues and write queues are disjoint structures. For the prototype, both queue types support 32 values.
- It is through the *read preamble* that general register values are injected into the grid. It is through the *write epilogue* that grid outputs are committed to the general registers.
- The general registers are partitioned into register *banks*. Each entry in a read queue or a write queue is aligned with one of the partitioned register banks.
- Reads and writes are interleaved across the general registers banks. On the prototype there are eight (8) entries for each of the four (4) general register banks.
- A read from a general register moves through the register read queue which is vertically aligned with the corresponding column in the grid.
- A write from an execution node to a general register moves through the register write queue in the column that is vertically aligned with the corresponding column in the grid.
- From a read queue, a register value can reach any execution node on the grid.
- From an execution node, a value can be output to any of the write queues.

General Register Read Example:

```
R[0] read G[124] N[15,1] N[25,0]
```

Appearing in the general register read preamble, this instruction means, "Read the value of general register 124 through register read queue entry 0 into operand 1 of execution node 15 and operand 0 of execution node 0."

General Register Write Examples:

```
N[12] add W[3]
```

This instruction, appearing in the body of grid instructions, means, "Output the result of the `add` to register write queue entry 3."

```
W[3] write G[127]
```

Appearing in the general register write epilogue, this instruction means, "Write the value in register write queue entry 3 to general register 127."

Preamble and Epilogue Example:

In this simple example, the code for the `Example` block is divided into three parts:

```
;;;;;;;;;;;;; Example instruction block
.bbegin Example
;;;;;;;;;; read preamble
R[0]  read G[124] N[15,0] ; read from read queue
R[13] read G[33]  N[15,1]
```

```
;;;;;;;;;; main instruction body
N[15] teq N[3,p]
N[3]  mov_t W[2] ; output a value to the write queue
;;;;;;;;;; write epilogue
W[2]  write G[12] ; commit the value to general register
.bend
```

Refer also to the "Program and Block Structure" subsection above.

4.5.2.2 General Register Alignment

To support the register access model in the TRIPS ISA, the assembler requires that reads from the general registers and writes to the general register align with the register banks as follows:

Read/Write Queue Entries	General Register Allowed Identifiers
0..7	G[0], G[4], G[8]…G[124]
8..15	G[1], G[5], G[9]…G[125]
16..23	G[2], G[6], G[10]…G[126]
24..31	G[3], G[7], G[11]…G[127]

Table 10: Aligning Read/Write Queues With General Registers

The assembler for the prototype checks read/write alignment according to the following formula:

```
read_or_write_queue_entry / 8
```

⇧

must equal

⇩

```
general_register_identifier mod 4
```

If this condition doesn't hold for all instructions in the read preamble and write epilogue, one or more alignment error messages is printed.

4.6 Reserved Symbols

Currently, there are no TRIPS-specific reserved symbols. To access such entities as the contents of the Processor Control Register, the programmer will access the corresponding memory-mapped address. For C system programmers, there will be a `sysparams.h` file that defines such information in this fashion:

```
#define PCR         0x1eadfee1
```

4.7 Assembly Language Sample

Following is a short sample of TRIPS assembly language, part of the `crt0.s` start-up code to initialize the application context.

```
.set SP=0x7fffc000
.text
.bbegin _start
; set up the stack pointer SP so that its address is in
; R[1]

N[8] genu %bottom(SP)   N[13,0]
N[7] app  %lo(SP)       N[14,0]
N[6] app  %mid(SP)      N[15,0] ; takes care of 48 bits
N[1] nop ; just for the heck of it

.bend
```

Note: This sample will be enlarged as we gain more experience in programming the instruction set.

5 Assembler Operation

As may be expected, the TRIPS assembler derives much of its behavior from the GNU specification.

It may be helpful to run comparisons between the TRIPS assembler and other assemblers in the GNU family. Doing so is easy:

1. Use the `gcc -S` switch to compile a C source file but leave the assembly output in a `*.s` file.

2. Run the as command with a variety of switches to note its behavior and output.

5.1 Assembly Phases

The assembler operates in two phases:

1. Parse the input file, build the symbol table, and check program semantics.

2. Generate the TOFF file as input to the linker.

5.2 Assembler Error Handling

If this were a perfect world, the TRIPS assembler would never need to issue warning or error message when assembling compiler output. The error handling used by tas follows the guidelines from GNU.

5.2.1 Minor Messages

A *minor message* is used for which the error recovery action is almost certainly correct. In this case, tas prints a message and then assembly continues as though no error occurred.

Example:

```
Warning: You never officially ended the block named `Example'.
We will do it for you.
```

5.2.2 Warning Messages

A *warning message* is used when we have an error from which we have a plausible error recovery, e.g., masking the top bits of a constant that is longer than will fit in the destination. In this case we will continue to assemble the source, although we may have made a bad assumption, and we will produce an object file and return normal exit status (i.e., no error).

5.2.3 Error Messages

An *error message* is used to mark errors that result in what we presume to be a useless object file. Say, we ignored something that might have been vital. If we see any of these, assembly will continue to the end of the source, no object file will be produced, and we will terminate with error status. The option, `-Z`, forces tas to produce an object file anyway, but it still exits with error status. The assumption here is that you don't want this object file.

Example:

```
Error: Your source register R[19] and general register G[5]
don't align.
```

5.2.4 Fatal Messages

A *fatal message* appears when tas is quite confused and continuing the assembly is pointless. In this case tas exits immediately with error status.

You shouldn't see fatal messages. If you do, please use Bugzilla to file a report against.

5.2.5 Command Line Switches

All tas warnings and error messages are directed to standard error output. There are two command switches to change the default behavior.

Suppress Warnings: `-W`

If you use this option, no warnings are issued. This option only affects the warning messages: it does not change any particular of how `as` assembles your file. Errors, which stop the assembly, are still reported.

Generate Object File in Spite of Errors: `-Z`

After an error message, the assembler normally produces no output. If for some reason you are interested in object file output even after `tas` gives an error message on your program, use the `-Z` option. If there are any errors, tas continues anyways, and writes an object file after a final warning message of the form `*n* errors, *m* warnings, generating bad object file.'

6 Other Tools

The objdump utility provides a useful disassembly capability for TRIPS binaries. Some useful options include:

```
objdump –t <file> # show the symbol table.  Also, readelf –s <file>
objdump –d <file> # disassemble the file
```

The readelf utility enables you to examine a module's header:

```
readelf –h <file>
```

The nm utility provides a list of symbols from a given object file.

Examples:

```
nm –size-sort <file> # sort by symbol size
nm –n <file> # sort numerically
```

7 Glossary

This glossary is intended to supplement the definitions in the *TRIPS Project Definition of Terms.*

Term	Description
execution node	A functional unit in the grid that performs ALU-type operations on its input operands and produces a single output value to forward to other execution nodes, to the register write queue, or to the memory system.
extended ISA	The instruction set supporting the research TRIPS architecture; as such it supports a variety of grid and general register configurations and bit encodings.
general register	One of the architectural registers in a partitioned register file that is visible to the software whose values persist across blocks. There are 128 general registers in the prototype ISA, 32 of whom may be output by any one execution block.
node register	One of the temporary registers that servers as an input to one of the execution nodes and whose value is lost when a new block is mapped onto the grid.
operand slot	A reservation station or predicate queue maintained on an execution node that accepts input operand values (left operand slot, right operand slot) or predicate value (predicate slot).
prototype ISA	The instruction set specific to the TRIPS hardware prototype, limited to a 4x4x8 grid configuration, 4x8 general register read and write queues, and a 4 x 32 general register set.
read preamble	A group of general register read instructions that (optionally) appears at the beginning of a grid instruction block.
target syntax	Used to express data flow execution, in which the output targets are written explicitly but the input operands are implied.
write epilogue	A group of general register write instructions that (optionally) appears at the end of a grid instruction block.

Table 11: Commonly Used Terms

8 Appendix: Deltas From the January, 2003 Assembler Specification

The updates to this specification are motivated by the requirement to support the Revised TRIPS ISA. Major changes occasioned by the Revised ISA are listed below.

Original ISA	Revised ISA	Differences

`R[0]..R[127]`	`G[0],G[4],…G[124]`
	`G[1],G[5],…G[125]`
	`G[2],G[6],…G[126]`
	`G[3],G[7],…G[127]`
		Partitioned nature of the register banks now visible in the ISA.

`N[15]-<006> or N[23,0]`	`N[15] or N[23,0]`
	
	`N[15]<006> or N[23,0]`
		Sequence numbers are no longer required in a block; however, they can be supplied.

`add`	`add_t`	All instructions except for C-type instructions can be predicated.
`rpt`	`mov, movi, write`	There are syntactical and semantic differences between instructions that move execution node values, read general registers, and copy immediate values.

`add N[0] N[1] N[2]`
 	
`add 0x64 N[1] N[2]`
	`add N[0] N[1]`
	
	`addi 0x64 N[1]`
		General and immediate instructions are limited to a maximum of 2 and 1 targets, respectively.

T2,T3,I1,I2,B1,B2, X0	G, I, L, S, C.	The revised instruction formats no longer include an explicit format field, but the general and immediate instructions include an extended opcode field.
`R[0]- move N[3]`	`N[2] MFPC N[3]`	Reading the program counter (formerly maintained in general register 0) now requires an explicit "Move From PC" grid instruction.
`R[3]- move`	`R[3] read`	General register syntax uses the `read` instruction instead of the `move` instruction and no longer includes a hyphen (-) after the register specifier.

Table 12: Revised Assembler Deltas