TRIPS Intermediate Language (TIL) Manual

Aaron Smith Jon Gibson Jim Burrill Katherine Coons
Robert McDonald Doug Burger Stephen W. Keckler
Kathryn S. McKinley

October 2, 2006 - Version A.04

Tech Report TR-05-20
Department of Computer Sciences
The University of Texas at Austin

This document specifies the TRIPS Intermediate Language (TIL) for the TRIPS architecture, a novel, scalable, and low power architecture for future technologies. TIL is a RISC-like intermediate representation for use by humans and compilers that want to write low-level TRIPS code.

Contents

1 Introduction

This manual presents a brief overview of the TRIPS Intermediate Language (TIL). We acknowledge our debt to the Free Software Foundation's GNU assembler (GAS) documentation, both in style and content.

1.1 Why an Intermediate Language?

TRIPS presents a new dataflow architecture based around a 2D array of ALUs. TRIPS Assembly Language (TASL) can be challenging to follow and reason about for those unfamiliar with it. Unlike more traditional ISAs, TASL is dataflow in nature—instructions have targets instead of operands and include scheduling information. TIL presents a more traditional, linear, three-operand RISC style where programmers need not concern themselves with scheduling the instructions on the grid of ALUs. It presents the familiar alegebraic paradigm where `op dest, operand1, operand2` is equivalent to `dest = operand1 op operand2`.

TIL is an intermediate language and any code written in it will need to be translated to TASL before it can be run. The TRIPS toolchain—specifically the TRIPS Scheduler—will make that translation and schedule all instructions on the TRIPS processor.

For those interested in TASL, more information can be found in the *TRIPS Processor Reference Manual*.

1.2 Brief Overview of TIL File Structure

TIL files can contain `.text`, `.data`, `.rdata`, and `.bss` sections which are further described in the "Sections and Relocation" section of this document.

There are key differences between the code layout of TRIPS `.text` sections and those of other architectures. These differences primarily revolve around the concept of TRIPS *program blocks*.

In the TRIPS architecture, program blocks are atomic execution units. Their representation in TIL is a sequence of linear instructions as one might find in the assembly of a more traditional architecture. Each block has a name associated with it and only by branching to that name can a block be entered. TRIPS blocks cannot be entered other than at the beginning and cannot exit until all of their outputs are produced.

Although a TRIPS block contains a sequential set of instructions, there are no hidden or implicit restrictions on execution order of the instructions in a block. Execution order is only constrained by the following:

- **Dataflow dependencies.** For example, instruction A is dependent on an output of instruction B; therefore, B will execute before A.

- **Load/store dependencies.** Loads and stores follow the sequential order in which they appear in TIL unless they are explicitely marked with *load/store IDs* (LSIDs), in which case they are ordered by LSID. See "Instruction" section for more information on LSIDs.
- **Reuse.** Defining a block temporary register more than once in a TRIPS block causes the scheduler to compute dataflow dependencies for that register based on the sequential order of the affected instructions.

More specifics on how blocks are structured can be found in the "Program Blocks" section of this document. Examples TIL files can be found in the Appendix.

2 Syntax

2.1 Whitespace

Whitespace is one or more blanks or tabs, in any order. Whitespace is used to separate symbols, and to make programs easier for people to read. Unless used within character constants, any whitespace means the same as exactly one space.

2.2 Comments

Comments are allowed anywhere in TIL. Anything from the line comment character to the next newline is considered a comment and is ignored. The line comment character is a semicolon (`;`).

```
; this is a comment which takes an entire line
.bbegin block1
  ret  ; this is a comment at the end of a line
.bend
```

2.3 Symbols

A *symbol* is typically the name of a memory location (for data items) or a program block address (for code sections). A symbol may consist of one or more characters chosen from the set of all letters (both upper and lower case), the digits, the dollar sign (`$`), and the underscore (_).

No symbol may begin with a digit. Case is significant. There is no length limit: all characters are significant. Symbols are delimited by characters not in that set.

A *label* is used to define a data symbol for inclusion in the program's symbol table. Syntactically, a label is a symbol followed immediately by a colon (:). No whitespace is allowed between a label and the colon.

Examples:

```
bro phi1$5 ; branch to the code block named phil$5

_V288:
.ascii "malloc failed for K[%d][%d]\n\000"; create a data symbol _V288
```

2.4 Statements

A statement ends at a newline character (\n) or at a semicolon (;). Newlines and semicolons within character constants do not end statements. Empty statements are allowed, and may include whitespace. They are ignored.

A statement may contain either a directive (a symbol whose first character is a dot '.') or a TIL instruction (see the sections on "TIL and Assembler Directives" and "TIL Instructions" for the supported directives and instructions). Whitespace may precede or follow a directive or instruction.

A statement may optionally begin with a label. Whitespace may precede or follow a label.

Examples:

```
__Label0:             ; an empty statement that contains a label
    .int 45           ; an assembler directive
subi $t12, $t8, 17    ; a TIL instruction
```

2.5 Constants

A constant is a number or character, written so that its value is known by inspection, without knowing any context.

Examples:

```
.data
my_bytes:
.byte 74, 0112, 0x4A, 'J'  ; a variety of data formats
my_string:
.asciz "Ring the bell\7"  ; string constant
my_float:
.single 3.14159265         ; 4-byte floating point.
```

3 Program Blocks

Program blocks (or *blocks*) represent the atomic execution unit in TIL. Blocks are sequences of instructions that begin with a *block begin* (`.bbegin`) directive and end with a *block end* (`.bend`) directive. All TRIPS instructions are contained within TIL blocks. It is illegal to place an instruction outside of a block.

Entry into a block is made by branching to the block's specified name (see the "TIL and Assembler Directives" section on the `.bbegin` directive). Branching to a label within a block is not supported. Execution of a block will terminate when *both* a branch instruction is reached *and* all specified outputs have been satisfied (i.e., values have been produced for all general register writes and all stores to memory have completed).

A TRIPS block should be constructed in the following order:

1. `.global` <*blockname*> (Optional; used to make the block visible to other TIL modules.)
2. `.bbegin` <*blockname*> (The <*blockname*> follows the syntactical rules for symbols as described above.)
3. `read` instructions (Names all general registers used in the block.)
4. Block instructions (Includes non-read/write instructions which reference only block temporaries and memory addresses.)
5. `write` instructions (Names all general registers defined in the block.)
6. `.bend`

Example:

```
.global calc2_$1
.bbegin calc2_$1
  read  $t1, $g3
  addi  $t2, $t1, 1
  bro   calc2_$2
  write $g90, $t2
.bend
```

There are limitations placed on the legal formation of a block that are architecture dependent. These limitations restrict the total number of instructions allowed in a block (the block size), the number of load/store identifiers per block and how many general registers can be read from and written to per block. Specific limitations of the TRIPS prototype can be found in the Appendix.

The TRIPS scheduler and assembler perform extensive checks to verify that blocks are correctly formed and adhere to processor constraints.

3.1 General and Block Temporary Registers

TIL supports two classes of registers: global registers, or *general registers*, and block temporary registers, or simply *block temporaries*.

Class	**Range**	**Syntax**
General Register	0...127	$g<*n*>
Block Temporary Register	0...unlimited	$t<*n*>

The two classes of registers are defined as follows:

- General registers retain their values between blocks and can only be read using a `read` instruction and written using a `write` instruction.
- With the exception of `read` and `write` instructions, all instructions within a block reference block temporaries.
- It is illegal for an instruction to reference a block temporary before the block temporary is defined.
- Register names are case insensitive. The names `$T1` and `$t1` map to the same block temporary register, just as `$G1` and `$g1` map to the same general register.
- Defined block temporaries must be used within a block. That is, *dead code* is not permitted in TIL blocks—the scheduler will issue an error message if it detects dead code.

Example: General registers `$g5` and `$g7` refer to the same general registers in `block1` and in `block2`. Block temporaries `$t1`, `$t2`, and `$t3` refer to different temporary registers in each block.

```
.bbegin block1
  read $t1, $g5
  movi $t2, 0
  add $t3, $t1, $t2
  bro block2
  write $g7, $t3
.bend

.bbegin block2
  read $t1, $g5
  movi $t2, 1
  add $t3, $t1, $t2
  bro block3
  write $g7, $t3
.bend
```

3.2 Predication

The TRIPS ISA provides extensive support for predicated execution. Most instructions can be predicated (see the section on "TIL Instructions" for exceptions). Each predicated instruction has a condition, either *true* or *false*, and a *predicate register* that is compared against the condition at runtime.

A low order bit of "0" in the predicate register signifies *false*; otherwise, the predicate register signifies *true*. If at runtime the value in the predicate register matches the instruction's condition, the instruction will execute; otherwise, the instruction will be treated as a `nop`.

To predicate an instruction, specify the condition by appending `_t` or `_f` to the instruction's opcode followed by a predicate register enclosed in angle brackets (< and >).

Examples:

```
mul_t<$t120> $t87, $t84, $t80
; if $t120 > 0, then $t87 = $t84 * $t80 else nop

addi_f<$t120> $t87, $t79, 8
; if $t120 == 0, then $t87 = $t79 + 8 else nop
```

3.3 Nullification

The TRIPS architecture requires that all block outputs (i.e., stores and general register writes) be produced along all possible predicate paths through a block. If a path does not produce a given output, then that path must produce a corresponding *nullified* output. Since block termination is dependent on all outputs being satisfied, a path that fails to produce a result for any output will prevent the block from terminating.

There are two types of nullification: store nullification and write nullification.

3.3.1 Store Nullification

Every predicate path through a block must produce the same set of store identifiers. A path with a missing store identifier must produce a null for the identifier to signal to the architecture that no value will be received for that identifier. This process is referred to as store nullification.

Example:

```
movi_f<$t120> $t134, 0
null_t<$t120> $t134
sd 8($t135), $t134 S[0] ;store identifier 0 is nullified when $t120 is true
```

There is no difference between nullifying a store's data or address operand or both—as soon as a store receives a null, it is nullified.

3.3.2 Write Nullification

Every predicate path through a block must write to the same set of general registers. If a general register will not be written on a path, it must be nullified to signal to the architecture that no input will be received by the write.

Example:

```
movi_f<$t120> $t137, 0
null_t<$t120> $t137
write $g3, $t137       ;write nullified when $t120 is true
```

4 Sections and Relocation

The assembler creates one section per TIL source file for each of the following types of code and data: text, read-write data, read-only data, and uninitialized data (referred to as *bss*, containing data items that have a name but no value).

During the final link phase, the TRIPS linker merges the various sections of multiple binary modules created by the the TRIPS assembler, into a single *program segment*, with merged `.text` sections, and a single *data segment*, with merged `.data`, `.rdata`, and `.bss` sections.

The executable image is created beginning at virtual address 0x00000000. At load time, the TRIPS runtime system loads or *relocates* this virtual image into physical memory and updates the TRIPS memory mapping hardware to translate the virtual addresses contained in the binary into physical memory addresses of the TRIPS system.

5 Scheduler and Assembler Directives

The following keywords are used by the TRIPS compiler to communicate additional information to the TRIPS scheduler and assembler in order to generate a TRIPS binary:

.align	.bss	.extern	.short
.app-file	.byte	.global	.single
.ascii	.comm	.int	.space
.asciz	.data	.lcomm	.text
.bbegin*	.double	.quad	.weak
.bend*	.equ	.rdata	

An asterisk (*) indicates that the scheduler itself operates on the directive. The other directives are simply passed from the compiler through the scheduler to the assembler. Refer to the *TRIPS Assembly Language Manual* for more information.

5.1 `.align` int

Pad the location counter (in the current section) to a particular storage boundary; int is the alignment required in bytes. For example, '.align 8' advances the location counter until it is a multiple of 8. If the location counter is already a multiple of 8, no change is needed. '.align 1' has no affect.

5.2 `.app-file` string

.app-file specifies that we are about to start a new logical file; string is the new file name. In general, the filename is recognized whether or not it is surrounded by quotes '"'; but if you wish to specify an empty file name, you must give the quotes-"". This statement may go away in future: it is only recognized to be compatible with old programs.

5.3 `.ascii` "string"

.ascii expects zero or more string literals separated by commas. Each string will be put into consecutive addresses with no automatic trailing zero byte.

Example:

```
_V440:
.ascii  "usage:  channel <n> [title]\n\000"
```

The above causes the assembler to create a symbol table entry that is named `_V440` and is associated with a memory chunk that stores the quoted character string.

5.4 `.asciz` "string"

.asciz is just like .ascii, but each string is followed by a zero byte. The 'z' in '.asciz' stands for 'zero'.

5.5 `.bbegin` symbol [xflags]

Begins a new TRIPS block named *symbol.* Within a source file, the symbol must be unique from all other symbols. The name will be governed by the scoping rules used by the compiler for C-style variables.

The optional *xflags* parameter may be used to set and clear certain execution flags which control the behavior of the prototype processor, based on an 8-bit mask.

Example:

```
.global _mutex_trylock ; make this symbol visible to other modules
.bbegin _mutex_trylock 0x4 ; require that this block be synchronized
```

The bitmask `0x6` causes the processor to inhibit the load predictor and require block synchronization for that block.

Refer also to Chapter 2, "Processing Model," and Chapter 3, "Programs," of the *TRIPS Processor Reference Manual* for more information on execution flags.

5.6 `.bend`

Ends the previous TRIPS block defined with .bbegin.

5.7 `.bss`

The bss section is not referenced explicitly in TIL. Any undefined symbol is assumed to be in the bss.

5.8 `.byte` expressions

.byte expects zero or more expressions, separated by commas. Each expression is assembled into the next byte

5.9 `.comm` symbol`,` length

.comm declares a common denoted by symbol. When linking, a common symbol in one object file may be merged with a defined or common symbol of the same name in another object file. If the linker does not see a definition for the symbol-just one or more common symbols-then it will allocate length bytes of uninitialized memory. Length must be an absolute expression. If the linker sees multiple common symbols with the same name, and they do not all have the same size, it will allocate space using the largest size.

5.10 `.data`

.data specifies that the following statements should be added onto the end of the data section. The beginning of the data section is 8-byte aligned. Data subsections are unsupported.

5.11 `.double` fp

.double expects one double-precision floating point number, occupying 8 bytes.

5.12 `.equ` symbol1=symbol2

.equ sets the value of symbol1 equal to symbol2. The compiler uses the .equ directive to supply duplicate names to the same entry point, as required by the Fortran77 runtime environment.

Example:

```
.global mg3xdemo_
.global main
.equ main=mg3xdemo_
```

5.13 `.extern` symbol

.extern is ignored if found in TIL. The linker will try to resolve all undefined symbols as if they were declared as C-style external variables.

5.14 `.global` symbol

.global makes the symbol visible to the linker. If you define a symbol in your partial program, its value is made available to other partial programs that are linked with it. Otherwise, symbol takes its attributes from a symbol of the same name from another file linked into the same program.

5.15 `.int` data

Reserves 4-bytes in the data section with the value "data".

5.16 `.lcomm` symbol, length

Reserve length (an absolute expression) bytes for a local common denoted by symbol. The section and value of symbol are those of the new local common. The addresses are allocated in the bss section, so that at run-time the bytes start off zeroed. Symbol is not declared global (see section .global symbol), so is normally not visible to the linker.

5.17 `.line` line-number

Change the logical line number. line-number must be an absolute expression. The next line has that logical line number. Therefore, any other statements on the current line (after a statement separator character) are reported as on logical line number line-number - 1

5.18 `.quad` data

Reserves 8-bytes in the data section with the value ”data”.

5.19 `.rdata`

Adds the following statements onto the end of the read-only data section. Currently only one read-only data section is supported.

5.20 `.short` data

Reserves 2-bytes in the data section with the value ”data”.

5.21 `.single` fp

This directive reserves 4-bytes in the data section for a single-precision floating point number with the value ”fp”.

5.22 `.space` size`,` fill

This directive emits size bytes, each of value fill. Both size and fill are absolute expressions. If the comma and fill are omitted, fill is assumed to be zero.

5.23 `.text`

Adds the following statements onto the end of the text section. The beginning of the text section is 8-byte aligned. Text subsections are not supported.

5.24 `.weak` symbol

.weak declares the symbol as weak. When a weak symbol is linked with a normal defined symbol, the normal defined symbol is used. If a symbol that is declared weak remains unresolved by the linker during linking, the linker will set the value of the weak symbol to zero.

6 TIL Instructions

6.1 Instruction Formats

The TRIPS ISA includes simple and consistent formats for machine-level instructions. The following table summarizes the corresponding TIL instruction formats and syntax.

Format	**Description**	**Syntax**
G2	General Instruction	opcode[_cond<Tp>] Td, Ta, Tb [N]
G1	General Instruction	opcode[_cond<Tp>] Td, Ta [N]
C1	Constant Instruction	opcode Td, Ta, Imm16 [N]
C0	Constant Instruction	opcode Td, Imm16 [N]
I1	Immediate Instructions	opcode[_cond<Tp>] Td, Ta, Imm9 [N]
L1	Load Instructions	opcode[_cond<Tp>] Td, Imm9(Ta) [LS] [D] [N]
S2	Store Instructions	opcode[_cond<Tp>] Imm9(Ta), Tb [LS] [D] [N]
R1	Read Instructions	opcode Td, Ga
W1	Write Instructions	opcode Gd, Ta
B1	Branch	opcode[_cond<Tp>] Ta [N]
B0	Branch with Offset	opcode[_cond<Tp>] Imm20 [N]
–	Enter	opcode Td, Imm [N]
–	Enter	opcode Td, Symbol [N]

The format names are designated "0", "1", or "2" based on the number of source operands included in the instruction, not including an immediate value. For example, the `ADD` instruction is classified as a *G2* General Instruction, as it consumes two operands to produce its result. No format is given for ENTER instructions as they are pseduo-instructions that only exist in the TIL. For a complete description of instruction formats see, Chapter 8, "Instructions," in the *TRIPS Processor Reference Manual*.

6.2 Instruction Fields

The following lists the fields used in the "Instruction Formats" table above and the "Instruction Summary" table below.

Field	Description
Imm	Any immediate value. Used as the source operand in an enter instruction.
Imm9	A 9-bit immediate value.
Imm16	A 16-bit immediate value.
Imm20	A 20-bit immediate value.
Td	A block temporary register used as the destination operand.
Ta	A block temporary register used as the first source operand.
Tb	A block temporary register used as the second source operand.
Tp	An optional block temporary register used as the predicate operand.
Imm9(Ta)	Load or store with offset. Effective address = Ta + Imm9.
Ga	A general register used as the source operand in a read instruction.
Gd	A general register used as the destination operand in a write instruction.
opcode	An opcode.
_cond	_t or _f appended to an opcode to specify whether the instruction is predicated on true or false.
Symbol	Any legal symbol.
LS	An optional 3-bit load/store identifier specified as L[n] for loads and S[n] for stores.
D	An optional 2-bit data bank alignment specified as D[n].
N	An optional node preplacement specified as N[row, column] or N[row, column, frame].

6.3 Instruction Preplacement

Instructions can be preplaced on a specific execution node by specifying the row, column, and optionally the frame on which to place the instruction. If the frame is not specified, the scheduler will assign a frame number according to normal placement ordering.

Example:

```
add $t7, $t0, $t1 N[0,1]   ; Place this instruction on the execution node
                           ; located at row 0, column 1
```

If a preplacement specifier is found on a TIL instruction that expands to multiple TASL instructions, then the first TASL instruction in the sequence will be preplaced.

6.4 Load/Store Instructions

6.4.1 Load/Store Identifiers

Load/Store identifiers (*LSIDs*) can be appended to load and store instructions in a TRIPS block. If LSIDs are provided for any load or store instructions in a block, they must be provided for *all* load and store instructions in the block. However, an LSID should not be provided for a load prefetch instruction. The toolchain may issue a warning or error message if a prefetch instruction contains a LSIDs.

LSIDs provide the hardware with a strict ordering of loads and stores within a block. If they are not provided, the scheduler will assign them based on the sequential order in which loads and stores occur within a TRIPS block, from top to bottom. LSIDs are numbered beginning at zero.

There is a limit to the number of LSIDs which can be assigned per block. For the TRIPS hardware prototype, a block can use no more than 32 LSIDs. LSIDs can be used more than once in a block so long as they are predicated on disjoint values and will never both execute. Also a load and store can never share the same LSID.

For loads, identifiers are specified in the TIL with L[n] where n is the 3-bit identifier. Similarly, identifiers are specified for stores using S[n].

Example:

```
ld $t7, 64($t12)     L[0]   ; a load with LSID 0
sd_t<$t5> $t9, $t10  S[1]   ; both stores use LSID 1
sd_f<$t5> $t8, $t10  S[1]
```

In the above example, both stores are allowed to use LSID 1 because they are mutually exclusive—it is guaranteed that only one will actually execute. Also in this example, if the load and store refer to the same address, the hardware will strictly order the load before the store based on the provided LSIDs.

6.4.2 Load/Store Data Bank Alignment

TIL programmers can specify data bank alignment *hints*, which the scheduler will consider when mapping load/store instructions onto the TRIPS grid. Data bank alignment hints are supplied through the D specifier in load and store instructions. Data bank alignment hints on load prefetch instructions will be ignored.

Example:

```
ld  $t7, 120($t0) L[4] D[2] ; align this load with data bank 2
```

The D specifier accepts values *0-3*, corresponding to the four data banks of the prototype L1 cache. This capability is particularly useful for fast array accesses in inner loops when the loop bodies have been unrolled multiple times within individual instruction blocks.

Note that the scheduler will attempt to honor data alignment hints but may override them due to other instruction scheduling priorities.

6.5 Instruction Summary

All of the following instructions, except for the ENTER pseudo-instructions, have a matching counterpart in the TRIPS Assembly Language (TASL) and will be converted to the appropriate TASL instruction by the scheduler. Refer also to the *TRIPS Assembly Language (TASL) Manual* and the *TRIPS Processor Reference Manual.*

Each instruction, except for READ and WRITE instructions, may take an optional node preplacement directive that is not show as part of the syntax.

ENTER instructions are pseudo-instructions and may expand into more than one instruction to generate their specified immediate or symbolic value. This expansion should be taken into account when determining the number of instructions in a TRIPS block.

Opcode	**Description**	**Syntax**
ADD	Add	<Tp> Td, Ta, Tb
ADDI	Add Immediate	<Tp> Ta, Tb, Imm9
AND	Bitwise And	<Tp> Td, Ta, Tb
ANDI	Bitwise And Immediate	<Tp> Td, Ta, Imm9
APP	Append Constant	Td, Ta, Imm16
BR	Branch	<Tp> Ta
BRO	Branch with Offset	<Tp> Imm20
CALL	Call	<Tp> Ta
CALLO	Call with Offset	<Tp> Imm20
DIVS	Divide Signed	<Tp> Td, Ta, Tb
DIVSI	Divide Signed Immediate	<Tp> Td, Ta, Imm9
DIVU	Divide Unsigned	<Tp> Td, Ta, Tb
DIVUI	Divide Unsigned Immediate	<Tp> Td, Ta, Imm9
ENTER	Generate Immediate Value	Td, Imm
ENTERA	Generate Data Address	Td, Symbol
ENTERB	Generate Block Address	Td, Symbol
EXTSB	Extend Signed Byte	<Tp> Td, Ta
EXTSH	Extend Signed Halfword	<Tp> Td, Ta
EXTSW	Extend Signed Word	<Tp> Td, Ta
EXTUB	Extend Unsigned Byte	<Tp> Td, Ta
EXTUH	Extend Unsigned Halfword	<Tp> Td, Ta
EXTUW	Extend Unsigned Word	<Tp> Td, Ta
FADD	FP Add	<Tp> Td, Ta, Tb
FDIV	FP Divide	<Tp> Td, Ta, Tb
FDTOI	Convert Double FP to Integer	<Tp> Td, Ta
FDTOS	Convert Double FP to Single FP	<Tp> Td, Ta

FEQ	FP Test EQ	<Tp> Td, Ta, Tb
FITOD	Convert Integer to Double FP	<Tp> Td, Ta
FGE	FP Test GE	<Tp> Td, Ta, Tb
FGT	FP Test GT	<Tp> Td, Ta, Tb
FLE	FP Test LE	<Tp> Td, Ta, Tb
FLT	FP Test LT	<Tp> Td, Ta, Tb
FMUL	FP Multiply	<Tp> Td, Ta, Tb
FNE	FP Test NE	<Tp> Td, Ta, Tb
FSTOD	Convert Single FP to Double FP	<Tp> Td, Ta
FSUB	FP Subtract	<Tp> Td, Ta, Tb
GENS	Generate Signed Constant	Td, Imm16
GENU	Generate Unsigned Constant	Td, Imm16
LB	Load Byte	<Tp> Td, (Ta)Imm9 [LS] [D]
LBS	Load Byte Signed	<Tp> Td, (Ta)Imm9 [LS] [D]
LD	Load Doubleword	<Tp> Td, (Ta)Imm9 [LS] [D]
LH	Load Halfword	<Tp> Td, (Ta)Imm9 [LS] [D]
LHS	Load Halfword Signed	<Tp> Td, (Ta)Imm9 [LS] [D]
LOCK	Load and Lock	<Tp> Td, (Ta)Imm9
LPF	Load Prefetch	<Tp> Td, (Ta)Imm9
LW	Load Word	<Tp> Td, (Ta)Imm9 [LS] [D]
LWS	Load Word Signed	<Tp> Td, (Ta)Imm9 [LS] [D]
MFPC	Move from PC	<Tp> Td
MOV	Move	<Tp> Td, Ta
MOVI	Move Immediate	<Tp> Td, Imm9
MOV3	Move to 3 Targets	<Tp> Td, Ta
MOV4	Move to 4 Targets	<Tp> Td, Ta
MUL	Multiply	<Tp> Td, Ta, Tb
MULI	Multiply Immediate	<Tp> Td, Ta, Imm9
NOP	No Operation	
NULL	Nullify Output	<Tp> Td
OR	Bitwise OR	<Tp> Td, Ta, Tb
ORI	Bitwise OR Immediate	<Tp> Td, Ta, Imm9
READ	Read General Register	Td, Ga
RET	Return	<Tp> Ta
SB	Store Byte	<Tp> (Ta)Imm9, Tb [LS] [D]
SCALL	System Call	<Tp>
SD	Store Doubleword	<Tp> (Ta)Imm9, Tb [LS] [D]
SH	Store Halfword	<Tp> (Ta)Imm9, Tb [LS] [D]
SLL	Shift Left Logical	<Tp> Td, Ta, Tb
SLLI	Shift Left Logical Immediate	<Tp> Td, Ta, Imm9
SRA	Shift Right Arithmetic	<Tp> Td, Ta, Tb
SRAI	Shift Right Arithmetic Immediate	<Tp> Td, Ta, Imm9
SRL	Shift Right Logical	<Tp> Td, Ta, Tb
SRLI	Shift Right Logical Immediate	<Tp> Td, Ta, Imm9

SUB	Subtract	<Tp> Td, Ta, Tb
SUBI	Subtract Immediate	<Tp> Td, Ta, Imm9
SW	Store Word	<Tp> (Ta)Imm9, Tb [LS] [D]
TEQ	Test EQ	<Tp> Td, Ta, Tb
TEQI	Test EQ Immediate	<Tp> Td, Ta, Imm9
TLE	Test LE	<Tp> Td, Ta, Tb
TLEI	Test LE Immediate	<Tp> Td, Ta, Imm9
TLEU	Test LE Unsigned	<Tp> Td, Ta, Tb
TLEUI	Test LE Unsigned Immediate	<Tp> Td, Ta, Imm9
TLT	Test LT	<Tp> Td, Ta, Tb
TLTI	Test LT Immediate	<Tp> Td, Ta, Imm9
TLTU	Test LT Unsigned	<Tp> Td, Ta, Tb
TLTUI	Test LT Unsigned Immediate	<Tp> Td, Ta, Imm9
WRITE	Write General Register	Gd, Ta
XOR	Bitwise XOR	<Tp> Td, Ta, Tb
XORI	Bitwise XOR Immediate	<Tp> Td, Ta, Imm9

APPENDIX

A TRIPS Hardware Prototype

The TRIPS hardware prototype has the following restrictions on the legal formation of TRIPS program blocks:

- The TRIPS prototype has 128 general purpose registers (numbered from 0-127) that are divided into four banks of 32 registers.
- Each TRIPS block can perform 32 reads and 32 writes of general purpose registers per block. These reads and writes are further restricted to 8 per bank per block. Calculating which bank a register is in involves a simple formula: *bank = register number % 4*.
- The prototype allows 128 instructions per block. This number is in addition to the read and write instructions.
- A maximum of 32 load/store identifiers (LSIDs) can be used per block.

This table further summarizes these restrictions.

Processor Resource	**Limit**
Number of general (global) registers	*128*
Number of register banks	*4*
Number of general register reads per block	*32*
Number of general register writes per block	*32*
Number of reads per bank per block	*8*
Number of writes per bank per block	*8*
Number of load/store identifiers (LSIDs) per block	*32*
Number of non-read/write instructions per block	*128*

When estimating the number of TASL instructions that the scheduler will generate for a given TRIPS block, it is important to account for two sources of code expansion:

- ENTER instructions are pseudo-instructions that will be expanded to the required number of instructions needed to generate their immediate or symbolic value. For example, the scheduler will expand an “Generate Block Address” instruction, such as `enterb $t6, Perl_scalarvoid$5`, into two TASL instructions, in order to generate a 32-bit address for the specified program block.
- Each instruction has a fixed number of targets. Therefore, the use of a temporary register by more than the number of available targets will cause fanout in a TRIPS block. The scheduler will insert MOV instructions to forward the register value to additional instructions.

B EBNF Grammar

Note: This section should not be considered fully accurate or complete.

The TRIPS Intermediate Language is specified in this section in a relaxed EBNF notation. Everything in **bold** is to be taken literally. Whitespace between names or literals that are not **bolded** are included for readability. Extra whitespace is allowed between non-terminals or terminals EXCEPT the following:

- variable names and ':'
- '0x' at the beginning of a hexadecimal number
- '-' before a negative number
- '.' in decimal numbers
- 'byte-' and integer in variable declaration

At least one whitespace is required after opcodes and after directives if the directive takes any arguments. An exception to this is that no space is required between a predicated opcode and its predicate register.

comment	→	;string-character*
module	→	**.app-file** [source-name]{info}*
info	→	data-section | text-section | global-directive
data-section	→	bss-section| idata-section | rdata-section
text-section	→	**.text**{ block|text-directive }*
text-directive	→	global-directive| equ-directive
global-directive	→	**.global** name
equ-directive	→	**.equ** name=name
align-directive	→	**.align** pos-integer
space-directive	→	**.space** pos-integer
bss-section	→	**.comm** variable-name,length| **.lcomm** variable-name,length| align-directive
idata-section	→	**.data** { variables| global-directive| align-directive| space-directive }*
rdata-section	→	**.rdata** { variables| global-directive| align-directive| space-directive }*
variables	→	[variable-name**:**] numerical-variable | [variable-name**:**] string-variable | [variable-name**:**] floatingpoint-variable | variable-name**:**
block	→	**.bbegin** block-name{instruction*} **.bend**
instruction	→	operation | operation predicate register{,register}*
register	→	real-register | virtual-register

real-register	→	**$g**pos-integer
virtual-register	→	**$t**pos-integer
predicate	→	_**t**< register >| _**f** < register >
value	→	immediate | symbol
source-name	→	first-letter-character { name-character }*
block-name	→	first-letter-character { name-character }*
variable-name	→	first-letter-character { name-character }*
numerical-variable	→	numerical-typeimmediate
numerical-type	→	**.quad** | **.int** | **.short** | **.byte** | **.byte-**pos-integer
immediate	→	integer | hex-integer
floatingpoint-variable	→	floatingpoint-type floatingpoint-number
floatingpoint-type	→	**.single** | **.double**
floatingpoint-number	→	integer [fraction] [exponent]
fraction	→	.pos-integer
exponent	→	**E** integer
string-variable	→	string-type string
string-type	→	**.ascii** | **.asciz**
string	→	"string-character*"
integer	→	[**-**]pos-integer
pos-integer	→	int-digit+
hex-integer	→	**0x**hex-digit+
int-digit	→	0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
hex-digit	→	int-digit | a | b | c | d | e | f | A | B | C | D | E | F
first-letter-character	→	lc-letter| uc-letter| _
name-character	→	symbol-character | $
string-character	→	symbol-character | punct-character | escape-character
symbol-character	→	lc-letter | uc-letter | int-digit | _
escape-character	→	_| ⁀| ¨ |
whitespace	→	space | tab

C Example Code

You can easily generate TIL code by means of command line switches to `tcc`, the TRIPS compiler script.

Examples:

```
# Compile 'example.c' to 'example.til' and then stop
#
```

```
% tcc -til example.c

# Compile, schedule, assemble, and link, leaving the intermediate
# files as 'example.til', 'example.s', and 'example.o'.
#
% tcc -save-temps example.c -o example
```

Refer to the *TRIPS Application Binary Interface (ABI) Manual* for information on register usage, function calling conventions, and runtime services provided by the TRIPS compiler and runtime system.

C.1 C Source Code

Note the following:

- The compiler will define `answer` as an uninitialized global variable.
- The compiler allocates 8 bytes for both `long` and `long long` integers. We use `long long` in this example for portability.
- The exit code of this example program will depend on the number of arguments passed to it through the command line.

```
/*
 * example.c
*/
unsigned long long answer;

unsigned long long ftn(unsigned long long x)
{
  return (x + 64) + (x + 128) + (x + 16000) +
    (x + 0xffffffffL) + (x + 0xffffffffffffLL);
}

int iftest(float x, float y)
{
  if (x > y)
    return 1;
  else
    return -1;
}

int main(int argc, char *argv[], char * envp[])
{
  answer = ftn((unsigned long long)42);
  printf("Hello, Galaxy! Answer = 0x%Lx.\n", answer);
  return argc + 1;
}
```

C.2 Resulting TIL Code

Note the following:

- Variable `answer` will be placed in the bss by the linker due to the `.comm` directive.
- The compiler expresses immediate values in `enter` pseudo-instructions using decimal notation.
- The compiler uses predicated `mov` instructions in the `iftest` block to evaluate the `if` condition.
- The compiler uses general register g2 to maintain return addresses. The "callee" is responsible for saving this register value passed to it by the caller. This convention and others are described in the *TRIPS Application Binary Interface (ABI) Manual.*

```
.app-file "example.c"
; BSS
.global answer
.comm answer, 8, 8

.data
.align 8
_V6:
.ascii "Hello, Galaxy! Answer = 0x%Lx.\n\000"

.text
.global ftn
.bbegin ftn
        read    $t0, $g2
        read    $t1, $g3
        enter   $t2, 16000
        add     $t3, $t1, $t2
        enter   $t4, 4294967295
        add     $t5, $t1, $t4
        add     $t6, $t3, $t5
        enter   $t7, 281474976710655
        add     $t8, $t1, $t7
        addi    $t9, $t1, 64
        addi    $t10, $t1, 128
        add     $t11, $t9, $t10
        add     $t12, $t8, $t11
        add     $t13, $t6, $t12
        ret     $t0
        write   $g3, $t13
.bend

.global main
;VARIABLE "argc" size:4 $g12
.bbegin main
        read    $t0, $g1
        read    $t1, $g2
        read    $t2, $g3
        read    $t3, $g12
        addi    $t4, $t0, -96
        sd      -88($t0), $t1 S[0]
        sd      -8($t0), $t3 S[1]
        extsw   $t5, $t2
        movi    $t6, 42
        enterb  $t7, main$1
        callo   ftn
        write   $g1, $t4
        write   $g2, $t7
        write   $g3, $t6
        write   $g12, $t5
.bend
.bbegin main$1
        read    $t0, $g3
        entera  $t1, answer
        sd      ($t1), $t0 S[0]
        entera  $t2, _V6
        enterb  $t3, main$2
        callo   printf
        write   $g2, $t3
        write   $g3, $t2
        write   $g4, $t0
.bend
.bbegin main$2
        read    $t0, $g1
        read    $t1, $g12
        addi    $t2, $t1, 1
        extsw   $t3, $t2
        ld      $t4, 8($t0) L[0]
        addi    $t5, $t0, 96
        ld      $t6, 88($t0) L[1]
        ret     $t4
        write   $g1, $t5
        write   $g2, $t4
        write   $g3, $t3
        write   $g12, $t6
.bend
```

```
.global iftest
.bbegin iftest
        read    $t0, $g2
        read    $t1, $g3
        read    $t2, $g4
        fgt     $t3, $t1, $t2
        movi_t<$t3>   $t4, 1
        mov_t<$t3>    $t5, $t4
        movi_f<$t3>   $t6, -1
        mov_f<$t3>    $t5, $t6
        extsw   $t7, $t5
        ret     $t0
        write   $g3, $t7
.bend
```

C.3 Partial TIL code from 171.swim SPEC benchmark

```
.app-file "swim.f"
; BSS - all declarations with no specified section
;       are placed in the BSS section.
    .global _BLNK__
    .comm _BLNK__, 199609200, 8
    .global cons_
    .comm cons_, 120, 8

.data
.align 8
_s143$$3913:
    .ascii  " SPEC benchmark 171.swim"
.align 8
_s149$$3914:
    .ascii  "SWIM7"
.align 8
_s150$$3915:
    .ascii  "UNKNOWN"

.rdata
.align 8
_FMT390_$$3909:
    .ascii  "(\' NUMBER OF POINTS IN THE X DIRECTION\',I8/\' NUMBER OF POINT"
    .ascii  "S IN THE Y DIRECTION\',I8/\' GRID SPACING IN THE X DIRECTION "
    .ascii  "  \',F8.0/\' GRID SPACING IN THE Y DIRECTION    \',F8.0/\' TIME "
    .ascii  "STEP                          \',F8.0/\' TIME FILTER PARAMETER"
    .ascii  "              \',F8.3/\' NUMBER OF ITERATIONS              \',"
    .ascii  "I8)\000"
```

```
.align 8
_FMT350_$$3910:
    .ascii  "(/\' CYCLE NUMBER\',I5,\' MODEL TIME IN  HOURS\',F6.2)\000"
.align 8
_FMT360_$$3911:
    .ascii  "(/\' DIAGONAL ELEMENTS OF U \',//(8E15.7))\000"
.align 8
_FMT366_$$3912:
    .ascii  "(/,\' Pcheck = \',E12.4,/,\' Ucheck = \',E12.4,/,\' Vcheck =\',E1"
    .ascii  "2.4,/)\000"

.text
.global calc2_
.bbegin calc2_
        read    $t0, $g1
        mov     $t1, $t0
        addi    $t2, $t0, -32
        sd      -32($t0), $t1
        movi    $t3, 0
        sd      -16($t0), $t3
        entera  $t4, cons_
        ld      $t5, 8($t4)
        enter   $t6, 4620693217682128896
        fdiv    $t7, $t5, $t6
        ld      $t8, 16($t4)
        fdiv    $t9, $t5, $t8
        ld      $t10, 24($t4)
        fdiv    $t11, $t5, $t10
        lws     $t12, 60($t4)
        extsw   $t13, $t12
        tgei    $t14, $t13, 1
        bro_t<$t14>     calc2_$1
        bro_f<$t14>     calc2_$8
        write   $g74, $t11
        write   $g75, $t9
        write   $g1, $t2
        write   $g76, $t7
        write   $g92, $t13
.bend
.bbegin calc2_$1
        movi    $t0, 1
        extsw   $t1, $t0
        bro     calc2_$2
        write   $g90, $t1
.bend
```