Team2-3

TIPS Assembly Language Programmer's Guide

TIPS is an acronym for either Thousands of Instructions Per Second, or Trillions of Instructions Per Second, depending on how optimistic we are at the moment.

Instruction Set Architecture

- Sixteen 16-bit temporary/general purpose registers called $t0 through $t7 and $p0 through $p5
- Thirty-two 16-bit special purpose storage registers called $s0 through $s27
- 16-bit Data Bus and 16-bit Address Bus
- A word is defined as 16 bits
- Load/Store Memory Access Architecture
- Byte Addressable Memory
- Every instruction is condition-coded.

Register Usage Conventions

Special Purpose Registers

- **PC ($s31)** – Stores the address of the next instruction.
- **BA ($s30)** – Base Address, all memory accesses will be relative to this address.
- **DU ($s29)** – The upper 16 bits of the 32-bit display register.
- **DL ($s28)** – The lower 16 bits of the 32-bit display register.
- **Flags ($s27)** – The flags register. Several of these bits are designated for the condition codes, as well as other flags relating to processor state. This is Read-Only, except for the Interrupt Enable flag, which may be written by a program.
- **Interrupt Vector Table ($s3-$s10)** – These registers store pointers to the interrupt handler for each of the interrupts. For convenience these may be referred to as $IVT0 through $IVT7.
- Registers $s11 through $s26 are reserved as a stack. The processor does not prevent writing directly to these registers, but this should be done with the utmost care as the processor uses the stack for its own purposes as well.
- Registers $s0 through $s2 are reserved for future uses, and thus should not be used by programs.

Conventions

Arguments to functions by convention are passed through the registers $p0 through $p5. However, if more arguments are needed or the programmer would rather have more temporary registers available, arguments may be placed on the stack. The called function then places the return value or values in these registers as well, or may choose to return values by placing them on top of the stack.
The values in registers $s0 through $s6 should be preserved by procedures. A convenient place to preserve values during the procedure is on the stack.

Registers $t0 through $t7 are referred to as the Lower General Purpose or Temporary registers, while $p0 through $p5 are called the Upper General Purpose or Parameter registers. S-Type and I-Type instructions can only access the Lower General Purpose Registers.

The layout of the flags register is shown in Table 0.1. Bit 0 is the Positive flag, and is set to 1 when the result of the previous instruction was positive. Bit 1 is the Zero flag, and is set when the previous result was zero. Bit 2 is the Negative flag, which is set when the previous instruction yielded a negative result. The 8 interrupt bits signal to the processor whether each of the interrupts has occurred. Bit 7 indicates whether the processor will respond to interrupts or not. This is initially 0, but the operating should set it to 1 as soon as it has populated the Interrupt Vector Table, which should be done as soon as possible.

Table 0.1 Flags Register

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Interrupt Mask								Interrupt Enable	Stack				N	Z	P

Exceptions and I/O

Exceptions

This document does not differentiate between exceptions and interrupts. The two terms may be used interchangeably.

This processor uses an Interrupt Vector Table stored in memory to handle interrupts. The processor supports up to eight interrupts. Five of these may be connected to an external device, while three of them are generated internally by the processor. The different interrupts a described in Table 0.1.

Table 0.1 Exception Descriptions

Number	Name	Description
0	External	These are connected to an external device, so their meaning is determined by the device.
1		
2		
3		
4		
5	Stack Error	Indicates a stack overflow or underflow has occurred. Whether it is an overflow or underflow can be determined by examining the flags.
6	Out of Range	Indicates there was either an arithmetic overflow or underflow. Which it was can be determined based on the condition codes.
7	Reserved	Reserved for future expansion.

When an exception occurs, the processor pushes the current PC value onto the stack, and then consults the Interrupt Vector Table to determine which procedure to call as the interrupt handler. Before calling the interrupt handler, the processor clears the Interrupt Enable flag, which prevents another interrupt from interrupting the current handler. The flag representing that that interrupt occurred is also cleared. The interrupt handler must re-enable interrupts before returning. For the sake of program responsiveness, an interrupt handler should also re-enable interrupts as soon as possible.

I/O

This processor makes use of memory mapped I/O. There are 16 ports which can be either input or output. Each port is 16 bits wide, and the ports may be accessed at the block of memory from 0x0000 through 0x001F. Each port is aligned on a word boundary. For the purposes of this project, port 0x0000 is the input port, and port 0x0002 is the lower part of the Display Output port, while 0x0004 is the upper part of the Display Output port.

Conditionals and Branching

This architecture is unique in that every instruction except for the two I-Type instructions can execute conditionally. This is done because quite often conditional branches skip over one or two instructions, and this way the extra instructions are not needed. This also leads to some rather unique ways of coding things. Condition codes are encoded as three bits in each instruction. The processor determines whether to execute that instruction by doing a logical AND of the condition code and the N,Z,P flags. If that result is non-zero, the function executes. Condition codes are specified by adding a suffix to an instruction. If there is no suffix, the assembler assumes the instruction should run unconditionally and sets all three bits of the condition code to 1. If a code is specified, the assembler sets the bits for the in the condition code. As an example, the instruction Add.nz would run only if the previous instruction set either the Negative or Zero flags, while Xor.pn would run only if the previous instruction produced a positive or negative result, that is, not zero. This processor does not have a specific jump instruction, although one may be implemented as a pseudoinstruction. Instead, jumping is achieved by directly modifying the program counter. Conditional jumps are done with condition codes. For example, to Jump if equal, a program could XOR numbers together and then only adjust the program counter if the result was 0.

Instruction Descriptions

Data Movement

Lm rd, roffset (R, 0x00)

Loads the word stored at memory BA+roffset into rd.

Sm rs, roffset (R, 0x01)

Stores the value in rs to BA+roffset.

Ls rd, special (S, 0x18)

Loads the value in special register special into rd.

Ss rs, special (S, 0x19)

Stores the value in rs into special register special.

Lui rd, imm8 (I, 0x1E)

Loads the 8 bit value imm8 into the upper 8 bits of rd.

Lli rd, imm8 (I, 0x1F)
Loads the 8 bit value imm8 into the lower 8 bits of rd

Push rs (R, 0x0A)
Pushes the value of rs onto the stack

Pop rd (R, 0x0B)
Pops the top value of the stack and stores it into rd.

Pushs special (S, 0x1B)
Puts the value in the special register designated by special onto the top of the stack.

Pops special (S, 0x1C)
Stores the value on top of the stack into special.

Arithmetic and Logic

Adds special, rs (S, 0x1A)
Special = special + rs

Add rd, rs (R, 0x02)
Rd = rd + rs

Sub rd, rs (R, 0x03)
Rd = rd - rs

Or rd, rs (R, 0x04)
rd is set to a bitwise or of rd and rs.

And rd, rs (R, 0x05)
rd is set to a bitwise and of rd and rs

Not rd, rs (R, 0x06)
rd is set to a bitwise not of rs

Xor rd, rs (R, 0x07)
rd is set to the bitwise exclusive or of rd and rs

Sl rd, ramt (R, 0x08)
Shifts rs left by the number stored in ramt

Sli rd, imm (R, 0x0A)
Shifts rs by the immediate value imm. This is takes an immediate value, but is an R-Type instruction. The immediate value is encoded into the rs field of the instruction.

Move rd, rs (R, 0x09)

rd = rs.

Machine Language

Instruction Format

Type	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R-Type	Opcode					rd				Rs				N	Z	P
S-Type	1	1	Instruction			lrd				Special				N	Z	P
I-Type	1	1	1	1	Ins	lrd				Immediate						

R-Type

These are instructions that use the general purpose registers as operands. The opcode field is not allowed to have the first two bits 1, or the first four be 1, because this is how the different types of instructions are distinguished. Below is a table with several sample encoded instructions.

Table 0.1R-Type Instruction Encoding

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Instruction	Opcode					Rd				Rs				N	Z	P
Add.pz $t0, $t1	0	0	0	1	0	0	0	0	0	0	0	0	1	0	1	1
Sl $p4, $t2	0	0	1	0	0	1	0	1	1	0	0	1	0	1	1	1
Sli $t7, 5	0	1	0	1	0	0	1	1	1	0	1	0	1	1	1	1

S-Type

These instructions are for manipulating the special registers. They can only use the lower general purpose registers as source, but can access any of the 32 special registers, although accessing those designated as stack registers directly is not recommend. The table again shows some example instructions.

Table 0.2 S-Type Instruction Encoding

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Instruction	Opcode					Rd					Rs			N	Z	P
Adds $s0, $t1	1	1	0	1	0	0	0	0	0	0	0	0	1	1	1	1
Pops $s0	1	1	1	0	0	0	0	0	0	0	0	0	0	1	1	1
Ls $t1, $s0	1	1	0	0	0	0	0	0	0	0	0	0	1	1	1	1

I-Type

The I-Type instructions create immediate values for storage in registers. They perform varied tasks, but they all require immediate values specified by the programmer. The immediate values can only be loaded into the bottom 8 general purpose registers, and can only load 8-bit values. The table shows the two immediate instructions as examples.

Table 0.3 I-Type Instruction Encoding

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Instruction	Opcode					Rd			Immediate Value							
Lui $t0, 10	1	1	1	1	0	0	0	0	0	0	0	0	1	0	1	0
Lli $t1, 15	1	1	1	1	1	0	0	1	0	0	0	0	1	1	1	1