WIKIBOOKS

MIPS Assembly/Print Version

MIPS Assembly/Cover

The MIPS microprocessor paradigm was created in 1981 from work done by J. L. Hennessy at Stanford University. Since that time, the MIPS paradigm has been so influential, that nearly every modern-day processor family makes some use of the concepts derived from that original research. This book will discuss the MIPS architecture and (perhaps most importantly) MIPS assembly programming.

This book is in an early stage of development. Any contributions would be helpful and appreciated.

The MIPS microprocessor paradigm was created in 1981 from work done by J. L. Hennessy at Stanford University. Since that time, the MIPS paradigm has been so influential that nearly every modern-day processor family makes some use of the concepts derived from that original research. This book will discuss the MIPS architecture and (perhaps more importantly) MIPS assembly programming.

Table of Contents

Resources and Licensing

Please add {{alphabetical}} only to book title pages.

Introduction

This page is going to serve as a general foreword about this book.

What This Book is About

This book is going to discuss the MIPS assembly language. This book will cover not only the straightforward facet of this subject (how to program MIPS), but will also look deeper, and discuss MIPS from a very low level. In this way, this book should be useful both for people just learning to program MIPS, and also to people who are looking to do advanced tasks in MIPS (such as write a MIPS assembler program, or construct a low-level OS kernel in MIPS). This book will not talk about the specifics of MIPS hardware, however.

Who This Book is For

This book is designed to be a reference for all people who are interested in MIPS. This book starts off with the basics behind the language, and discusses the various operations in such a manner that beginners will be able to get a handle of MIPS programming. However, this book also contains a number of advanced sections for experienced programmers who are looking at doing advanced projects with the MIPS architecture.

How This Book is Organized

This book is organized in such a fashion that the most simple material is presented first, and the most advanced material is saved towards the end. The first section of the book is reserved for historical and interesting information about MIPS, and listings of real-world MIPS implementations. The second section is going to go into the MIPS assembly language, talking about each instruction individually, and explaining how to use them. The third section is going to talk about the topics of programming, assembling, and emulating MIPS code. Finally, the fourth section is going to talk about advanced topics, such as the internal machine code of MIPS machines, the nature of pseudoinstructions, some advanced system instructions, and handling exceptions.

Where to Go From Here

This book will serve as a complete reference to programming MIPS. While there are no Wikimedia resources specifically designed to follow this text, the reader may benefit from reading about higher-level languages, or assemblers, or another advanced programming topic.

For more information about how to design MIPS and other types of microprocessor systems, see Microprocessor Design.

Section 1: Introduction to MIPS

MIPS Architecture

MIPS History

The MIPS architecture

MIPS is a register based architecture, meaning the CPU uses registers to perform operations on. There are other types of processors out there as well, such as stack-based processors and accumulator-based processors.

Registers are memory just like RAM, except registers are much smaller than RAM, and are much faster. In MIPS the CPU can only do operations on registers, and special immediate values.

MIPS processors have 32 general purpose registers, but some of these are reserved. A fair number of registers however are available for your use. For example, one of these registers, the *program counter*, contains the memory address of the next instruction to be executed. As the processor executes the instruction, the program counter is incremented, and the next memory address is fetched, executed, and so on.

Why MIPS?

The MIPS architecture is a Reduced Instruction Set Computer (RISC). As a RISC architecture, it doesn't assign individual instructions to complex, logically intensive tasks. This is in contrast to complex instruction set computer (CISC) architectures like the DEC VAX, which had an instruction to multiply polynomials and another to perform a cyclic redundancy check (CRC), often used in TCP/IP. At the time, it was thought that implementing such instructions in hardware would result in performance increase for programs that used them, even if it resulted in highly complex processor design. MIPS and other RISC architectures were based on the philosophy that, among other things, by only implementing a small core (only a few dozen instructions, instead of several hundred) of the most common instructions, architects could simplify the design and speed up the majority of common instructions so much that the cost of implementing complex programs as multiple instructions would be hidden.

Much has been written on the RISC versus CISC debate,[1][2][3][4] so for our purposes we shall focus on the consequences of the MIPS design choices:

- All MIPS instructions are 32 bits long.
  - This makes hardware for accessing and decoding instructions straightforward.
  - This also means there are a finite number of instructions.
- All MIPS instructions belong to one of three instruction formats. This makes decoding instructions simple for both humans and hardware. Since the instruction format is regular, it doesn't take much work to learn most of the MIPS instruction set.

MIPS Philosophies

- Simplicity favors regularity
- Good design demands good compromise

- Smaller is faster
- Make the common tasks the fastest

MIPS Processors

This page is going to talk about some of the specific MIPS implementations, and MIPS relatives.

Most MIPS implementations use the classic 5 stage pipeline: instruction fetch, instruction decode/register fetch, execute, memory access, and register write back.

Further reading

- The Microchip PIC32 microcontrollers are based on a MIPS32 M4K Core.
- The MIPS architecture is one of several popular architectures in embedded systems.
- The PlayStation uses a MIPS processor. Several Wikibooks discuss programming the PlayStation: Emulation/PlayStation 1, Linux Guide/PlayStation 3, PSP Programming, Map This!.

MIPS Details

Registers

MIPS has 32 general-purpose registers and another 32 floating-point registers. Registers all begin with a dollar-symbol ($). The floating point registers are named $f0, $f1, ..., $f31. The general-purpose registers have both names and numbers, and are listed below. When programming in MIPS assembly, it is usually best to use the register names.

Number	Name	Comments
$0	$zero	Always zero
$1	$at	Reserved for assembler
$2, $3	$v0, $v1	First and second return values, respectively
$4, ..., $7	$a0, ..., $a3	First four arguments to functions
$8, ..., $15	$t0, ..., $t7	Temporary registers
$16, ..., $23	$s0, ..., $s7	Saved registers
$24, $25	$t8, $t9	More temporary registers
$26, $27	$k0, $k1	Reserved for kernel (operating system)
$28	$gp	Global pointer
$29	$sp	Stack pointer
$30	$fp	Frame pointer
$31	$ra	Return address

In general, there are many registers that can be used in your programs: the ten **temporary registers** and the eight **saved registers**, and the arg $a and return-value $v registers. Temporary registers are general-purpose registers that can be used for arithmetic and other instructions freely (call clobbered), while saved registers must keep their value across function calls. (If you want to use one, you have to save it on procedure entry, and restore at procedure exit). The easiest way to handle the call-preserved $s0..7 registers is to not touch them at all.

Temporary register names all start with a $t. For instance, there are $t0, $t1 ... $t9. this means there are 10 temporary registers that can be used without worrying about saving and restoring their contents. The saved registers are named $s0 to $s7.

The **zero register**, is named $zero ($0), and is a static register: it always contains the value zero. This register may not be used as the target of a store operation, because its value is hardwired in, and cannot be changed by the program.

There are also several registers to which the programmer does not have direct access with most instructions. Among these are the Program Counter (PC), which stores the address of the instruction executing (read by JAL to calculate a return address, written by jumps and branches), and the "hi" and "lo" registers, which are used in multiplication and division, which have results longer than 32 bits (multiplication may result in a 64-bit product and division results in a quotient and remainder). There are special instructions to move data to and from the hi and lo registers.

Instruction Formats

There are 3 instruction formats: R Instructions, I Instructions, and J Instructions.

R Instructions

R Instructions take three arguments: two source registers (**rt** and **rs**), and a destination register (**rd**). R instructions are written using the following format:

instruction rd, rs, rt

where each one stands for as follow:

rd	Destination register specifier
rs	Source register specifier
rt	Source/Destination register specifier

For example,

```
add $t0, $t1, $t2
```

Adds the values of $t1 and $t2 and stores the result in $t0.

When assembled into machine code, an R instruction is represented as follows:

opcode	**rs**	**rt**	**rd**	**shamt**	**func**
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

For R-format instructions, the **opcode**, or "operation code" is always zero. **rs**, **rt**, and **rd** correspond to the two source and one destination registers, respectively. **shamt** is used in shift instructions instead of **rt** to make the hardware simpler. In assembly, to shift the value in $t4 two bits to the left and place the result in $t5:

```
sll $t5, $t4, 2
```

Since the opcode is zero for all R-format instructions, **func** specifies to the hardware exactly which R-format instruction to execute. The add example above would be encoded as follows:

```
 opcode rs    rt     rd   shamt funct
 000000 01001 01010 01000 00000 100000
```

Since it is an R-format instruction, the first six bits (opcode) are 0. The next 5 bits correspond to rs, which in this example is $t1. From the table above, we find that $t1 is $9, which in binary is 01001. Likewise, the next five bits encode $t2 = $10 = 01010. The destination is $t0 = $8 = 01000. We are not performing a shift, so shamt is 00000. Finally, since the func for the add instruction is 100000.

For the shift example above, the opcode field is again 0 since this is an R format instruction. The rs field is unused in shifts, so we leave the next five bits at 0. The rt field is $t4 = $12 = 01100. The rd field is $t5 = $13 = 01101. The shift amount, shamt, is 2 = 00010. Finally, the func field for sll is 000000. Thus, the encoding for **sll $t5, $t4, 2** is:

```
opcode rs    rt     rd   shamt  funct
000000 00000 01100 01101 00010  000000
```

I Instructions

I instructions take two register arguments and a 16-bit "immediate" value. An immediate is a value that is stored as part of the instruction instead of in memory. This makes accessing constants much faster than if we had to put constants in memory and then load them (hence the name). I-format instructions, like R-format instructions, specify the target register (**rt**) first. Next comes one source register (**rs**) and finally the immediate value.

instruction rt, rs, imm

For example, let's say that we want to add the value 5 to the register $t1, and store the result in $t0:

```
addi $t0, $t1, 5
```

Or compare-and-branch to a nearby label (range is 16-bit signed displacement, left shifted by 2). The assembler calculates `(target - branch_insn_address + 4) >> 2` as the immediate:

```
beq $t0, $zero,  t0_equals_zero_branch_target
```

I-format instructions are represented in machine code as follows:

opcode	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

The **opcode** specifies which operation is requested. **rs** and **rt** are five bits each, as before, and in the same positions as the R-format instructions. The **imm** field holds the immediate value. Depending on the instruction, the immediate constant may either be sign-extended or zero-extended. If a 32-bit immediate is needed, a special instruction, **lui** ("load upper immediate") exists for loading an immediate into the upper 16 bits of a register. That register can then be logically ORed with another 16-bit immediate to store the final value in that register. That value can then be used in normal R-format instructions. The following sequence of instructions stores the bit pattern 0101 0101 0101 ... into register $t0:

```
lui $t0, 0x5555
ori $t0, $t0, 0x5555
```

Typically, the assembler will automatically split 32-bit constants in this way so the programmer doesn't have to worry about it if you write li $t0, 0x5555555. Some assemblers (like MARS with extended pseudo-instructions mode enabled) even support large immediates for addi / addiu / xori / etc. this way.

The addi example from above would be encoded as follows. The addi instruction has an opcode of 001000. The source register, $t1, is number 9, or 01001 in binary. The target register, $t0, is number 8, or 01000 in binary. Five is 101 in binary, so **addi $t0, $t1, 5** in machine code is:

```
opcode   rs    rt            imm
001000 01001 01000 0000 0000 0000 0101
```

addi and addiu sign-extend their 16-bit immediate to 32 bits (so can represent signed 2's complement values from -32768 .. 32767, i.e. unsigned values of 0..0x7fff and 0xffff8000 .. 0xffffffff)

Bitwise boolean logical instructions like andi, ori, and xori zero-extend the immediate (so can use values from 0..65535)

Other instructions that use the I format include load/store (address = register + signed 16-bit displacement), compare-immediate such as slti $t1, $t0, 1234 (producing a 0 or 1 in a register), lui for loading an immediate to the upper 16 bits of a register, and branch (but not jump) instructions.

J Instructions

J instructions are used to transfer program flow to a given hardcoded absolute address within the 256MB region around the PC register. J instructions are almost always written with labels: the assembler and linker will convert the label into a numerical value. A J instruction takes only one argument: the address to jump to.

(Truly PC-relative control transfers can be done instead with 'b label' that are usable in position-independent code. MIPS branch instructions are I-format instructions with a 16-bit relative displacement, left-shifted by 2, unlike jumps.)

instruction addr

There are two J-format instructions: **j** and **jal**. The latter will be discussed later. The j ("jump") instruction tells the processor to immediately skip to the instruction addressed by **addr**. To example, to jump to a **label1**:

```
j label1
```

A J-format instruction is encoded as

opcode	addr
6 bits	26 bits

On MIPS32 machines, addresses are 32-bits wide, so 26 bits may not be enough to specify which instruction to jump to. Fortunately, since all instructions are 32-bits (four bytes) wide, we can assume that all instructions start at a byte address that's divisible by 4 (we are in fact guaranteed this by the loader). In binary, a number that is divisible by 4 ends with two zeros (just like a number that's divisible by 100 in decimal always ends in two zeros). Therefore, we can allow the assembler to leave out the last two zeros and have the hardware reinsert them. This effectively makes the address field 28 bits. The final four bits will be borrowed from the address of the next instruction, so we cannot let a program straddle a 256MB boundary, because a jump across the boundary would require a change in the 4 uppermost bits. (The PC while a J instruction is processed has already been updated to point to the next instruction, the branch-delay slot.)

In the example above, if label1 specified an instruction as address 120, or 1111000 in binary, we can encode the jump example above in machine code as follows. The opcode for j is 2, or 10 in binary, and we must chop off the last two bits of the jump address, leaving it at 11110. Thus, the machine code for **j 120** is:

```
opcode |---------------addr-----------|
000010 0000 0000 0000 0000 0000 0111 10
```

Section 2: MIPS Instructions

MIPS Instructions

Writing assembly code

An assembly language program has a few common features. These are:

- labels
- sections
- directives
- comments
- commands

Comments

In MIPS assembly, comments are denoted with a number sign ("#"). Everything after that sign on the same line is a comment and is skipped by the assembler's lexer.

Labels

A *label* is something to make your life simple. When you reference a piece of your program, instead of having to count lines, you can just give it a name. You use this in loops, jumps, and variable names.

Labels don't appear in your final code, they're only there for convenience, one of the few perks you'll get from the typical MIPS assembler. It also makes life easy for the assembler, because it can now easily go around relocating and linking code. Don't worry if you don't know what those are, that'll come later.

Labels in MIPS assembly, actually in most variants of ASM as well, are created by writing

```
name:
```

where *name* is a name you use to refer to this label. Note that, you can't create a label with the same name as a MIPS instruction.

Labels are the same as their C cousins, the targets of goto operators.

Sections

A **section** is a way of dividing an assembly language program up into the instructions and the data logically. Since your assembly program is loaded into memory, all the processor needs to know is the start of execution (known as the *entry point*) and merely just increments the instruction pointer and continues along. If data is interspersed in the instruction code in memory, the processor needs to know what is data and what is instructions - which serves for a much more complicated processor. For simplicity, the data is set off in a different area than the instructions. Sections are used for this purpose.

There are two sections you need to be mindful of in MIPS programming: the text and data sections. The text section holds your assembly program, and the data section holds your data. One designates a text section by starting with `.text` and the start of a data section by `.data`.

Directives

The processor assembler (or your emulator) understands a few special commands that perform specific tasks to make your life easier: for example creating memory space for you to store data in variables.

If you wish to create a variable for a number in your assembly program, you may want to:

1. switch to the data section
2. create a label to create a memory reference for your variable
3. give it an appropriate name
4. allocate some memory for it
5. align it properly in memory.
6. switch back to the text section

From what we have learned, we can write the MIPS assembly code up to step 3:

```
   .data
variable_name:
```

But now how can we allocate some memory? We use the `.space` directive. An integer takes up 4 bytes, usually, so we write

```
   .data
variable_name:
   .space 4
```

We need to *align* the data in memory now (we'll explain this below), so we use the `.align` directive

```
    .data
variable_name:
    .space 4
    .align 2
    .text
```

And we've switched back to the text section.

Memory locations in MIPS assembler need to be *aligned* - that is, that memory locations must begin on the correct location otherwise the processor will crash. 4-byte integers must be aligned every 4 bytes, and so on. The `.align n` directive aligns the data to a memory cell fit for 2^n bytes. Therefore, for 32-bit (4-byte) integers we use `.align 2` for 2^2 or 4 bytes. Aligns the next variable or instruction on a byte that is a multiple of number. To align the space allocated, the 'align n' should come before the 'space x' declaration. For example, here is a sample for allocating 6 words (6 x 4 = 24 bytes) in memory and aligning them at word boundaries

```
    .data
    .align 2
my_var_name:
    .space 24
```

Arithmetic Instructions

Register Arithmetic Instructions

Instruction:	**add**	type:	**R Type**

This instruction adds the two operands together, and stores the result in the destination register. Negative numbers are handled automatically using two's complement notation, so different instructions do not need to be used for signed and unsigned numbers.

Instruction:	**sub**	type:	**R Type**

The sub instruction subtracts the second source operand from the first source operand, and stores the result in the destination operand. In pseudo-code, the operation performs the following:

```
rd := rs - rt
```

Both **add** and **sub** trap if overflow occurs. However, some programming systems, like C, ignore integer overflow, so to improve performance "unsigned" versions of the instructions don't trap on overflow.

Instruction:	**addu**	type:	**R Type**
Instruction:	**subu**	type:	**R Type**

Multiplication and Division

The multiply and divide operations are slightly different from other operations. Even if they are R-type operations, they only take 2 operands. The result is stored in a special 64-bit result register. We will talk about the result register after this section.

Instruction:	**mult**	type:	**R Type**

This operation multiplies the two operands together, and stores the result in rd. Multiplication operations must differentiate between signed and unsigned quantities, because the simplicity of Two's Complement Notation does not carry over to multiplication. The **mult** instruction multiplies and sign extends signed numbers.

The result of multiplying 2 32-bit numbers is a 64-bit result. We will discuss the 64-bit results below.

Instruction:	**multu**	type:	**R Type**

The **multu** instruction multiplies the two operands together, and stores the result in rd. This instruction is for unsigned numbers only, and does not sign extend a negative result. This operation also creates a 64-bit result.

Instruction:	**div**	type:	**R Type**

The div instruction divides the first argument by the second argument. The quotient is stored in the lowest 32-bits of the result register. The remainder is stored in the highest 32-bits of the result register. Like multiplication, division requires a differentiation between signed and unsigned numbers. This operation uses signed numbers.

Instruction:	**divu**	type:	**R Type**

Like the div instruction, this operation divides the first operand by the second operand. The quotient is stored in the lowest 32-bits of the result, and the remainder is stored in the highest 32-bits of the result. This operand divides unsigned numbers, and will not sign-extend the result.

64-Bit Results

The 64-bit result register is broken into two 32-bit segments: **HI** and **LO**. We can interface with these registers using the mfhi and mflo operations, respectively.

Instruction:	**mfhi**	type:	**R Type**

Takes only 1 operand. This instruction moves the high-32 bits of the result register into the target register.

Instruction:	**mflo**	type:	**R Type**

Also takes only 1 operand. Moves the value from the LO part of the result register into the specified register.

If the upper (most significant) 32 bits of a product are unimportant to computation, programmers may save a step by using instructions that discard the upper 32 bits.

Instruction:	**mul**	type:	**R Type**

There is no unsigned version of the **mul** instruction. The **mul** instruction may also clobber the existing values in HI and LO.

Register Logic Instructions

These operations perform bit-wise logical operations on their operands.

Instruction:	**and**	type:	**R Type**

Performs a bitwise AND operation on the two operands, and stores the result in rd.

Instruction:	**or**	type:	**R Type**

Performs a bitwise OR operation on the two operands, and stores the result in rd.

Instruction:	**nor**	type:	**R Type**

Performs a bitwise NOR operation on the two operands, and stores the result in rd.

Instruction:	**xor**	type:	**R Type**

Performs a bitwise XOR operation on the two operands, and stores the result in rd.

Immediate Arithmetic Instructions

These instructions sign-extend the 16-bit immediate value to 32-bits and performs the same operation as the instruction without the trailing "i".

Instruction:	**addi**	type:	**I Type**
Instruction:	**addiu**	type:	**I Type**

To subtract, use a negative immediate.

Immediate Logic Instructions

All logical functions zero-extend the immediate.

Instruction:	**andi**	type:	**I Type**

Takes the bitwise AND of rs with the immediate and stores the result in rt.

Instruction:	**ori**	type:	**I Type**

Takes the bitwise OR of rs with the immediate and stores the result in rt.

Instruction:	**xori**	type:	**I Type**

Takes the bitwise XOR of rs with a the immediate and stores the result in rt.

Shift instructions

Instruction:	**sll**	type:	**R Type**

Logical shift left: rd ← rt << shamt. Fills bits from right with zeros.

Instruction:	**srl**	type:	**R Type**

Logical shift right: rd ← rt >> shamt. Fills bits from left with zeros.

Instruction:	**sra**	type:	**R Type**

Arithmetic shift right. If rt is negative, the leading bits are filled in with ones instead of zeros: rd ← rt >>

shamt.

Because not all shift amounts are known in advance, MIPS defines versions of these instructions that shift by the amount in the rs register. The behavior is otherwise identical.

Instruction:	**sllv**	type:	**R Type**
Instruction:	**srlv**	type:	**R Type**
Instruction:	**srav**	type:	**R Type**

Control Flow Instructions

Jump Instruction

The jump instructions load a new value into the PC register, which stores the value of the instruction being executed. This causes the next instruction read from memory to be retrieved from a new location.

Instruction: **j** type: **J Type**

The **j** instruction loads an immediate value into the PC register. This immediate value is either a numeric offset or a label (and the assembler converts the label into an offset).

Instruction: **jr** type: **R Type**

The **jr** instruction loads the PC register with a value stored in a register. As such, the jr instruction can be called as such:

```
jr $t0
```

assuming the target jump location is located in $t0.

Jump and Link

Jump and Link instructions are similar to the jump instructions, except that they store the address of the next instruction (the one immediately after the jump) in the return address ($ra; $31) register. This allows a subroutine to return to the main body routine after completion.

Instruction: **jal** type: **J Type**

Like the **j** instruction, except that the return address is loaded into the $ra register.

Instruction: **jalr** type: **R Type**

The same as the **jr** instruction, except that the return address is loaded into a specified register (or $ra if not specified)

Example

Let's say that we have a subroutine that starts with the label MySub. We can call the subroutine using the following line:

```
jal MySub
...
```

And we can define MySub as follows to return to the main body of the parent routine:

```
jr $ra
```

Branch Instructions

Instead of using rt as a destination operand, rs and rt are both used as source operands and the immediate is sign extended and added to the PC to calculate the address of the instruction to jump to if the branch is taken.

Instruction:	**beq**	type:	**I Type**

Branch if rs and rt are equal. If rs = rt, PC ← PC + 4 + imm.

Instruction:	**bne**	type:	**I Type**

Branch if rs and rt are not equal. If rs ≠ rt, PC ← PC + 4 + imm.

Instruction:	**bgez**	type:	**I Type**

Branch if rs is greater than or equal to zero. If rs ≥ 0, PC ← PC + 4 + imm.

Instruction:	**blez**	type:	**I Type**

Branch if rs is less than or equal to zero. If rs ≤ 0, PC ← PC + 4 + imm.

Instruction:	**bgtz**	type:	**I Type**

Branch if rs is greater than zero. If rs > 0, PC ← PC + 4 + imm.

Instruction:	**bltz**	type:	**I Type**

Branch if rs is less than zero. If rs < 0, PC ← PC + 4 + imm.

Set Instructions

These instructions set rd to 1 if their condition is true. They can be used in combination with **beq** and **bne** and $zero to branch based on the comparison of two registers.

Instruction:	**slt**	type:	**R Type**

If rs < rt, rd ← 1, else 0.

Instruction:	**slti**	type:	**I Type**

If rs < imm, rd ← 1, else 0. The immediate is sign extended.

Instruction:	**sltu**	type:	**R Type**

If rs < rt, rd ← 1, else 0. Treat rs and rt as unsigned integers.

Instruction:	**sltiu**	type:	**I Type**

If rs < imm, rd ← 1, else 0. The immediate is sign extended, but both rs and the extended immediate are treated as *unsigned* integers. The sign extension allows the immediate to represent both very large and very small unsigned integers.

Memory Instructions

Load and store instructions use a special syntax:

```
instr rt, imm(rs)
```

The memory address used for the load or store is rs + imm. The immediate is sign-extended.

Load Instructions

Instruction:	**lbu**	type:	**I Type**

Loads a byte and does not sign-extend the value.

Instruction:	**lhu**	type:	**I Type**

Loads a halfword, or two bytes, and does not sign-extend the value. The halfword must be aligned (i.e., it must start at an even address).

Instruction:	**lw**	type:	**I Type**

Loads a word (four-bytes) from memory. The word must be aligned (i.e., the last two bits of the address must be zero).

Store Instructions

Instruction:	**sb**	type:	**I Type**

Stores the least significant (rightmost) byte of rt to memory.

Instruction:	**sh**	type:	**I Type**

Stores the least significant (rightmost) halfword of rt to memory.

Instruction:	**sw**	type:	**I Type**

Stores the contents of rt in memory.

Floating Point Instructions

Arithmetic

Integer implementation of floating-point addition

```
#Initialize variables
add $s0,$t0,$zero #first integer value
add $s1,$t1,$zero #second integer value
add $s2,$zero,$zero #initialize sum variable to 0
add $t3,$zero,$zero #initialize SUM OF SIGNIFICANDS value to 0

#get EXPONENT from values
sll $s5,$s0,1 #getting the exponent value
srl $s5,$s5,24 #$s5 = first value EXPONENT

sll $s6,$s1,1 #getting the exponent value
srl $s6,$s6,24 #$s6 = second value EXPONENT

#get SIGN from values
srl $s3,$s0,31 #$s3 = first value SIGN
srl $s4,$s1,31 #$s4 = second value SIGN

#get FRACTION from values
sll $s7,$s0,9
srl $s7,$s0,9 #$s7 = first value FRACTION
sll $t8,$s1,9
srl $t8,$s1,9 #$t8 = second value FRACTION

#compare the exponents of the two numbers
compareExp: ######################

beq $s5,$s6, addSig
blt $s5,$s6, shift1 #if first < second, go to shift1
blt $s6,$s5, shift2 #if second < first, go to shift2
j compareExp

shift1: #shift the smaller number to the right
srl $s7,$s7,1 #shift to the right 1
addi $s5,$s5,1
j compareExp
```

Compare

Load

Store

Branch

Further Reading

- Floating Point

Section 4: Programming MIPS

MIPS Assemblers

This page is going to list some assemblers for programming in MIPS, and will discuss how to program using MIPS assemblers.

/edit by Till Fischer, University of Ulm (Germany)

Heres a link to a Download of the MARS, an emulation Assembler Program by the Missouri State University, we use that one in our Studies in order to learn Assembler: http://courses.missouristate.edu/KenVollmar/MARS/

MIPS Emulation

This page is going to talk about using MIPS emulators to test MIPS code on a non-MIPS computer.

A common MIPS Emulator is Spim which is fully cross platform. Another one is MIPS Assembler and Simulator (http://xavier.perseguers.ch/programmation/mips-assembler.html), which has been successfully deployed in and used in a few high schools. QEMU (http://wiki.qemu.org/Main_Page) is another common emulator environment for MIPS. Yet another Emulator is MARS (http://courses.missouristate.edu/KenVollmar/MARS/) a cross-platform Java MIPS IDE.

Open Virtual Platforms (OVP) http://www.OVPworld.org (http://www.OVPworld.org) includes the freely available simulator OVPsim, a library of models of processors, peripherals and platforms, and APIs which enable users to develop their own models. The models in the library are open source, written in C, and include the MIPS 4K, 24K and 34K cores. These models are created and maintained by Imperas http://www.imperas.com (http://www.imperas.com) and in partnership with MIPS Technologies have been tested and assigned the MIPS-Verified(tm) mark. Sample MIPS-based platforms include both bare metal environments and platforms for booting unmodified Linux binary images. These platforms/emulators are available as source or binaries and are fast, free, and easy to use. OVPsim is developed and maintained by Imperas and is very fast (100s of million instructions per second), and built to handle multicore architectures. To download the MIPS OVPsim simulators/emulators visit http://www.OVPworld.org/mips (http://www.OVPworld.org/mips).

Subroutines

This page is going to talk about using subroutine structures in MIPS Assembly. Also, this page will talk about some of the common methods by which higher-level language constructs and subroutines are translated into MIPS assembly code.

Subroutine Mechanics

In MIPS, the general purpose registers are typically taken to be the function parameters and the function variables, as needed. Previous values of the general purpose registers are stored on the stack. At the end of a subroutine, the values of all registers stored in this manner are restored from the stack.

MIPS uses the stack to preserve these registers. The stack pointer points to the bottom of the stack, but the frame pointer points to the top of the local frame. Offsetting from the stack or frame pointers can retrieve the various saved values and function parameters, as needed.

Stem and Leaf

In the function hierarchy, a function that calls other functions is known as a **stem** function. A function which calls no other functions is known as a **leaf** function. Stem functions must save the value of **$ra** on the stack, so that its content is preserved across the other function calls, and so the *called function* can return to the *calling function*, or its parent function.

It is important to note that a leaf function does not need to preserve the value of the **$ra** register because it does not call any other child function and hence its return address is preserved in **$ra**.

Stack Frame

A function may set up a stack frame, with the **$sp** register pointing to the bottom of the stack, and the **$fp** register pointing to the top of the stack at the start of the subroutine. In this manner, **$fp** will be pointing to the function's input arguments (if any are on the stack), and **$sp** will be pointing to the local variables for the function.

The value of **$fp** needs to be preserved across function calls, and stem functions need to save the value of **$fp** before calling a child function, and restore it from the stack after calling the child function.

The value of **$sp** does not need to be stored on the stack, but functions need to take care to return **$sp** to the value it had at the beginning of the function, before the function returns. This means that at the beginning of a subroutine, values can be pushed onto the stack, and at the end of the subroutine, all those values need to be popped right back off the stack. If this is not done correctly, it will destroy the stack frame of the parent function, and possibly cause the computer to crash.

Saved Registers

There are a number of registers that must be preserved across a subroutine call. This means that if the subroutine wants to use those registers, it must save the previous values onto the stack (or some other place), and then reload those values at the end of the function. The **$tx** registers are all temporary registers and do not need to be saved. Likewise, the **$ax** registers are all function arguments, and do not need to be saved. Thr **$v0** and **$v1** registers are both function return values, and do not need to be saved.

The following registers do need to be preserved across a function call, and should be saved by the function if they are going to be used in the function:

$sx
: The "Saved Temporary" registers

MIPS and C Linkage

further reading

- Embedded Systems/Mixed C and Assembly Programming

Programming Style

This page is going to talk about programming style and code formatting techniques in MIPS Assembly. Programming style is an arbitrary construct that is dependent on the individual programmer, but this page will show some common styles anyway.

Good, consistent programming style for assembly languages in general is arguably more important than for higher-level languages, as you don't have the luxury of abstracting even the simplest of operations away. For that reason, comments are even more important than in higher-level languages! The necessity of dealing with the nitty-gritty means your program will grow in lines of code very quickly, and someone's going to have to read it. Why not make their life easier?

There are several styles and conventions one may adhere to. Doesn't matter too much which one you use, but be consistent!

Section 3: Advanced MIPS

Instruction Formats

This page describes the implementation details of the MIPS instruction formats.

R Instructions

R instructions are used when all the data values used by the instruction are located in registers.

All R-type instructions have the following format:

```
OP rd, rs, rt
```

Where "OP" is the mnemonic for the particular instruction. *rs*, and *rt* are the source registers, and *rd* is the destination register. As an example, the **add** mnemonic can be used as:

```
add $s1, $s2, $s3
```

Where the values in *$s2* and *$s3* are added together, and the result is stored in *$s1*. In the main narrative of this book, the operands will be denoted by these names.

R Format

Converting an R mnemonic into the equivalent binary machine code is performed in the following way:

opcode	rs	rt	rd	shift (shamt)	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

opcode
: The opcode is the machinecode representation of the instruction mnemonic. Several related instructions can have the same opcode. The opcode field is 6 bits long (bit 26 to bit 31).

rs, rt, rd
: The numeric representations of the source registers and the destination register. These numbers correspond to the $X representation of a register, such as $0 or $31. Each of these fields is 5 bits long. (25 to 21, 20 to 16, and 15 to 11, respectively). Interestingly, rather than rs and rt being named r1 and r2 (for source register 1 and 2), the registers were named "rs" and "rt" for register source, register target and register destination.

Shift (shamt)
: Used with the shift and rotate instructions, this is the amount by which the source operand *rs* is rotated/shifted. This field is 5 bits long (6 to 10).

Funct
: For instructions that share an opcode, the **funct** parameter contains the necessary control codes to differentiate the different instructions. 6 bits long (0 to 5). Example: Opcode 0x00 accesses the ALU, and the funct selects which ALU

function to use.

Function Codes

Because several functions can have the same opcode, R-Type instructions need a function (Func) code to identify what exactly is being done - for example, 0x00 refers to an ALU operation and 0x20 refers to ADDing specifically.

Shift Values

I Instructions

I instructions are used when the instruction must operate on an immediate value and a register value. Immediate values may be a maximum of 16 bits long. Larger numbers may not be manipulated by immediate instructions.

I instructions are called in the following way:

```
OP rt, IMM(rs)
```

However, **beq** and **bne** instructions are called in the following way:

```
OP  rs, rt, IMM
```

Where *rt* is the target register, *rs* is the source register, and *IMM* is the immediate value. The immediate value can be up to 16 bits long. For instance, the **addi** instruction can be called as:

```
addi $s1, $s2, 100
```

Where the value of $s2 plus 100 is stored in $s1.

I Format

I instructions are converted into machine code words in the following format:

opcode	rs	rt	IMM
6 bits	5 bits	5 bits	16 bits

Opcode
The 6-bit opcode of the instruction. In I instructions, all mnemonics have a one-to-one correspondence with the underlying opcodes. This is because there is no **funct** parameter to differentiate instructions with an identical opcode. 6 bits (26 to 31)

rs, rt

The source and target register operands, respectively. 5 bits each (21 to 25 and 16 to 20, respectively).[5]

IMM

The 16 bit immediate value. 16 bits (0 to 15). This value is usually used as the offset value in various instructions, and depending on the instruction, may be expressed in two's complement.

J Instructions

J instructions are used when a jump needs to be performed. The J instruction has the most space for an immediate value, because addresses are large numbers.

J instructions are called in the following way:

```
OP LABEL
```

Where *OP* is the mnemonic for the particular jump instruction, and *LABEL* is the target address to jump to.

J Format

J instructions have the following machine-code format:

Opcode	Pseudo-Address
6 bits	26 bits

Opcode

The 6 bit opcode corresponding to the particular jump command. (26 to 31).

Address

A 26-bit shortened address of the destination. (0 to 25). The full 32-bit destination address is formed by concatenating the highest 4 bits of the PC (the address of the instruction following the jump), the 26-bit pseudo-address, and 2 zero bits (since instructions are always aligned on a 32-bit word).

FR Instructions

FR instructions are similar to the R instructions described above, except they are reserved for use with floating-point numbers:

Opcode	fmt	ft	fs	fd	funct

FI Instructions

FI instructions are similar to the I instructions described above, except they are reserved for use with floating-point numbers:

Opcode	fmt	ft	Imm

Opcodes

The following table contains a listing of MIPS instructions and the corresponding opcodes. Opcode and funct numbers are all listed in hexadecimal.

Mnemonic	Meaning	Type	Opcode	Funct
add	Add	R	0x00	0x20
addi	Add Immediate	I	0x08	NA
addiu	Add Unsigned Immediate	I	0x09	NA
addu	Add Unsigned	R	0x00	0x21
and	Bitwise AND	R	0x00	0x24
andi	Bitwise AND Immediate	I	0x0C	NA
beq	Branch if Equal	I	0x04	NA
blez	Branch if Less Than or Equal to Zero	I	0x06	NA
bne	Branch if Not Equal	I	0x05	NA
bgtz	Branch on Greater Than Zero	I	0x07	NA
div	Divide	R	0x00	0x1A
divu	Unsigned Divide	R	0x00	0x1B
j	Jump to Address	J	0x02	NA
jal	Jump and Link	J	0x03	NA
jalr	Jump and Link Register	R	0x00	0x09
jr	Jump to Address in Register	R	0x00	0x08
lb	Load Byte	I	0x20	NA
lbu	Load Byte Unsigned	I	0x24	NA
lhu	Load Halfword Unsigned	I	0x25	NA
lui	Load Upper Immediate	I	0x0F	NA
lw	Load Word	I	0x23	NA
mfhi	Move from HI Register	R	0x00	0x10
mthi	Move to HI Register	R	0x00	0x11
mflo	Move from LO Register	R	0x00	0x12
mtlo	Move to LO Register	R	0x00	0x13
mfc0	Move from Coprocessor 0	R	0x10	NA
mult	Multiply	R	0x00	0x18
multu	Unsigned Multiply	R	0x00	0x19
nor	Bitwise NOR (NOT-OR)	R	0x00	0x27
xor	Bitwise XOR (Exclusive-OR)	R	0x00	0x26
or	Bitwise OR	R	0x00	0x25
ori	Bitwise OR Immediate	I	0x0D	NA
sb	Store Byte	I	0x28	NA

Mnemonic	Meaning	Type	Opcode	Funct
`sh`	Store Halfword	I	0x29	NA
`slt`	Set to 1 if Less Than	R	0x00	0x2A
`slti`	Set to 1 if Less Than Immediate	I	0x0A	NA
`sltiu`	Set to 1 if Less Than Unsigned Immediate	I	0x0B	NA
`sltu`	Set to 1 if Less Than Unsigned	R	0x00	0x2B
`sll`	Logical Shift Left	R	0x00	0x00
`srl`	Logical Shift Right (0-extended)	R	0x00	0x02
`sra`	Arithmetic Shift Right (sign-extended)	R	0x00	0x03
`sub`	Subtract	R	0x00	0x22
`subu`	Unsigned Subtract	R	0x00	0x23
`sw`	Store Word	I	0x2B	NA

System Instructions

This page is going to discuss some of the more advanced MIPS instructions that might not be used in every-day programming tasks.

Instruction:	**syscall**	type:	**R Type**

syscall allows you to call upon the basic system functions. To use syscall, first set $v0 with the code of the function you want to call, then use syscall. The exact codes available may depend on the specific system used, but the following are examples of common system calls.

code	call	arguments	results
1	print integer	$a0 = integer to print	
2	print float	$f12 = float to print	
3	print double	$f12 = float to print	
4	print string	$a0 = address of beginning of string	
5	read integer		integer stored in $v0
6	read float		float stored in $f0
7	read double		double stored in $f0
8	read string	$a0 = pointer to buffer, $a1 = length of buffer	string stored in buffer
9	sbrk (allocate memory buffer)	$a0 = size needed	$v0 = address of buffer
10	exit		
11	print character	$a0 = character to print	

```
Example: printing the number 12
li $a0, 12→;loads the number we want printed, 12 in this case, into the first argument register
li $v0, 1→;stores the code for the print integer call into $v0
syscall→;Executes system call
```

Instruction:	**break**	type:	**R Type**

Instruction:	**sync**	type:	**R Type**

Instruction:	**cache**	type:	**R Type**

Instruction:	**pref**	type:	**R Type**

Pseudoinstructions

The MIPS instruction set is very small, so to do more complicated tasks we need to employ assembler macros called **pseudoinstructions**.

List of Pseudoinstructions

The following is a list of the standard MIPS instructions that are implemented as pseudoinstructions:

- abs
- blt
- bgt
- ble
- neg
- negu
- not
- bge
- li
- la
- move
- sge
- sgt
- add

Branch Pseudoinstructions

Branch if less than (blt)

The **blt** instruction compares 2 registers, treating them as signed integers, and takes a branch if one register is less than another.

```
blt $8, $9, label
```

translates to

```
slt $1, $8, $9
bne $1, $0, label
```

Other Pseudoinstructions

Load Immediate (li)

The **li** pseudo instruction loads an immediate value into a register.

```
li $8, 0x3BF20
```

translates to

```
lui $at, 0x0003
ori $8, $at, 0xBF20
```

Absolute Value (abs)

The absolute value pseudo instruction loads the absolute value contained in one register into another register.

```
abs $1, $2
```

translates to

```
addu $1, $2, $0
bgez $2, 8 (offset=8 → skip 'sub' instruction)
sub $1, $0, $2
```

Move (move)

The move pseudo instruction moves the contents of the second register operand into the first register operand.

```
move $1, $2
```

translates to

```
add $1, $2, $0
```

Load Address (la)

```
la $a0,address
```

translates to

```
  lui $at, 4097 (0x1001 → upper 16 bits of $at).
  ori $a0,$at,disp
```

where the immediate ("disp") is the number of bytes between the first data location (always 0x 1001 0000) and the address of the first byte in the string.

Register File

Registers

MIPS has 32 general-purpose registers and another 32 floating-point registers. Registers all begin with a dollar-symbol ($). The floating point registers are named $f0, $f1, ..., $f31. The general-purpose registers have both names and numbers, and are listed below. When programming in MIPS assembly, it is usually best to use the register names.

Number	Name	Comments
$0	$zero, $r0	Always zero
$1	$at	Reserved for assembler
$2, $3	$v0, $v1	First and second return values, respectively
$4, ..., $7	$a0, ..., $a3	First four arguments to functions
$8, ..., $15	$t0, ..., $t7	Temporary registers
$16, ..., $23	$s0, ..., $s7	Saved registers
$24, $25	$t8, $t9	More temporary registers
$26, $27	$k0, $k1	Reserved for kernel (operating system)
$28	$gp	Global pointer
$29	$sp	Stack pointer
$30	$fp	Frame pointer
$31	$ra	Return address

Zero Register

The zero register ($zero or $0) always contains a value of 0. It is built into the hardware and therefore cannot be modified.

$at Register

The $at (Assembler Temporary) register is used for temporary values within pseudo commands. It is not preserved across function calls. For example, with the (slt $at, $a0, $s2) command, $at is set to one if $a0 is less than $s2, otherwise it is set to zero.

$v Registers

The $v Registers are used for returning values from functions. They are not preserved across function calls.

Argument Registers

The $a registers are used for passing arguments to functions. They are not preserved across function calls.

Temporaries

The temporary registers are used by the assembler or assembly language programmer to store intermediate values. They are not preserved across function calls.

Saved Temporaries

Saved Temporary registers are used to store longer lasting values. They are preserved across function calls.

$k Registers

The k registers are reserved for use by the OS kernel. They may change randomly at any time as they are used by interrupt handlers.

Pointer Registers

- **Global Pointer** ($gp) - Usually stores a pointer to the global data area (such that it can be accessed with memory offset addressing).
- **Stack Pointer** ($sp) - Used to store the value of the stack pointer.
- **Frame Pointer** ($fp) - Used to store the value of the frame pointer.
- **Return Address** ($ra) - Stores the return address (the location in the program that a function needs to return to).

All Pointer Registers are preserved accross function calls.

Exceptions

This page is going to talk about exception handling. We will also talk about what exceptions are, and what causes them.

Exception Control Registers

Exception Codes

Handling Exceptions

Resources and Licensing

Resources

Wikimedia Resources

- wikipedia:assembler
- Programming:C
- C++
- Operating System Design
- Embedded Systems
- Reverse Engineering
- Microprocessor Design
- PSP Programming -- the PSP runs MIPS assembly language.

External Resources

- Reverse Engineering for Beginners
- A programmed Introduction to MIPS Assembly (http://chortle.ccsu.edu/AssemblyTutorial/index.html)
- OVPsim Virtual Platform Simulator (http://www.ovpworld2.org/mips)
- A Software MIPS Simulator (http://xavier.perseguers.ch/programmation/mips-assembler.html)
- SPIM: MIPS Simulator (http://www.cs.wisc.edu/~larus/spim.html)
- Patterson and Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, 3rd Edition. Elsevier, 2005. ISBN 1558606941 Invalid ISBN
- Paul, Richard P. *SPARC Architecture, Assembly Language Programming, and C*, 2nd Edition, Prentice Hall, 2000. ISBN 0130255963
- http://www.cs.tau.ac.il/~afek/MipsInstructionSetReference.pdf (the MIPS32 instruction set)

Licensing

The text of this book is released under the following license:

2. VERBATIM COPYING

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

1. John Mashey on RISC/CISC (http://userpages.umbc.edu/~vijay/mashey.on.risc.html)
2. RISC vs. CISC (http://www-cs-faculty.stanford.edu/~eroberts/courses/soco/projects/2000-01/risc/risccisc/)
3. Patterson, D. A. & Ditzel, D. R. (1980). The Case for the Reduced Instruction Set Computer. *SIGARCH Computer Architecture News,* 8(6), 25-33.
4. Clark, D. W. & Strecker W. D. (1980). Comments on 'The case for the reduced instruction set computer' by Patterson and Ditzel. *SIGARCH Computer Architecture News,* 8(6), 34-38.
5. Lin, Charles (2003-03-27). "Instruction Format". Archived from the original on

2018-01-01. https://web.archive.org/web/20180101004911if_/http://www.cs.umd.edu/class/spring2003/cmsc311/Notes/Mips/format.html. Retrieved 2019-11-12.