MIPS Assembly

The MIPS Assembly Language

Assembly: An Introduction

Assembly, the Hardware Aspect

Vocabulary

Components of an Assembly Program

Lexical Category	Example(s)
`Comment`	# Comment
`Assembler Directive`	.data, .asciiz, .global
`Operation Mnemonic`	add, addi, lw, bne
`Register name`	$10, $t2
`Address label (dec)`	help:, length:, loop:
`Address label (use)`	hello, length, loop
`String constant`	"ERRRRRRR"
`Character Constant`	'H'. '?'

Some Conventions Used In This Document

Operand Notation	Description
`$d`	Destination operand, must be int register
`$r`	Source operand, must be int register
`Imm`	Immediate value
`$d/Imm`	Source operand, can be int register or immediate value
`$d_f`	Destination operand, floating-point
`$s_f`	Source operand, floating point
`Mem`	Memory location May be: a variable name, or an indirect reference (memory address, etc.)

Data and Memory in Assembly

Overview

Addressing Data in Memory

The processor controls the execution of instructions through the fetch-decode-execute or the execution cycle:

- Fetch instruction from memory
- Decode or identify the instruction
- Execute the instruction

The processor may access one or more bytes of memory at a time.

Memory transfer between register and memory:

The processor stores data in reverse-byte sequence, i.e. a low-order byte is stored in a low memory address and high-order byte in high memory addressee:

Memory							
High-order	••••••••	••••••••	••••••••	••••••••	••••••••	••••••••	**Low-order**

For example, when brining a value from register to memory:

Register	••••••••	••••••••	••••••••	••••••••	>>>
Memory	••••••••	••••••••	••••••••	••••••••	
	address	add + 1	add + 2	…	

, **and**, when getting the numeric data from memory to register:

Memory	••••••••	••••••••	••••••••	••••••••	>>>
Register	••••••••	••••••••	••••••••	••••••••	

Types of Memory

There are two kinds of memory addresses:

- **Absolute address**: a direct reference of specific location
- **Segment address** (or, **offset**): Starting address of a memory segment with the offset

Data Structures

"Variables"

The fundamental unit of computer storage is a **byte**: It could be

- ON(`1`), or
- OFF(`0`)

Out of which 7 bits are used for data and the last is used for parity.

One Byte							
Actual data (7 bits)							Parity
`(0/1)`	`(0/1)`	`(0/1)`	`(0/1)`	`(0/1)`	`(0/1)`	`(0/1)`	`(0/1)`

A frame thing like this should go by odd parity (odd number of `1`s)

Data Sizes

the processor supports the following data sizes:

`byte`	8-bit Integer
`halfword`	16-bit Integer
`word`	32-bit Integer
`float`	32-bit floating point number
`double`	64-bit floating point number

Encodings

ASCII

ASCII (A*merican* S*tandard* C*ode for* I*nformation* I*nterchange*) has 128 characters, 95 graphic and 3 control characters:

00	nul	**01**	soh	**02**	stx	**03**	etx	**04**	eot	**05**	enq	**06**	ack	**07**	bel
08	bs	**09**	ht	**0a**	nl	**0b**	vt	**0c**	np	**0d**	cr	**0e**	so	**0f**	si
10	dle	**11**	dc1	**12**	dc2	**13**	dc3	**14**	dc4	**15**	nak	**16**	syn	**17**	etb
18	can	**19**	em	**1a**	sub	**1b**	esc	**1c**	fs	**1d**	gs	**1e**	rs	**1f**	us
20	sp	**21**	!	**22**	"	**23**	#	**24**	$	**25**	%	**26**	&	**27**	'
28	(	**29**	)	**2a**	*	**2b**	+	**2c**	,	**2d**	-	**2e**	.	**2f**	/
30	0	**31**	1	**32**	2	**33**	3	**34**	4	**35**	5	**36**	6	**37**	7
38	8	**39**	9	**3a**	:	**3b**	;	**3c**	<	**3d**	=	**3e**	>	**3f**	?
40	@	**41**	A	**42**	B	**43**	C	**44**	D	**45**	E	**46**	F	**47**	G
48	H	**49**	I	**4a**	J	**4b**	K	**4c**	L	**4d**	M	**4e**	N	**4f**	O
50	P	**51**	Q	**52**	R	**53**	S	**54**	T	**55**	U	**56**	V	**57**	W
58	X	**59**	Y	**5a**	Z	**5b**	[	**5c**	\	**5d**	]	**5e**	^	**5f**	_
60	`	**61**	a	**62**	b	**63**	c	**64**	d	**65**	e	**66**	f	**67**	g
68	h	**69**	i	**6a**	j	**6b**	k	**6c**	l	**6d**	m	**6e**	n	**6f**	o
70	p	**71**	q	**72**	r	**73**	s	**74**	t	**75**	u	**76**	v	**77**	w
78	x	**79**	y	**7a**	z	**7b**	{	**7c**	|	**7d**	}	**7e**	~	**7f**	del

Latin-I

Latin-I includes ASCII + 96 more graphic characters.

Unicode

Unicode includes several encodings:

UTF-8, UTF-16, UTF-32

MIPS Overview

Cheat Sheets

Instruction Formats

R-Type

	Instruction Format					
Bit #	31 - 26	25 -21	20 - 16	15 - 11	10 - 6	5 - 0
R-type	op	rs	rt	rd	smt	func
add	0	rs	rt	rd	0	100000
addu	0	rs	rt	rd	0	100001
sub	0	rs	rt	rd	0	100010

subu	0	rs	rt	rd	0	100011
and	0	rs	rt	rd	0	100100
or	0	rs	rt	rd	0	100101
xor	0	rs	rt	rd	0	100110
nor	0	rs	rt	rd	0	100111
sll	0	0	rt	rd	s	0
srl	0	0	rt	rd	s	10
sra	0	0	rt	rd	s	11
sllv	0	rs	rt	rd	0	100
srlv	0	rs	rt	rd	0	110
srav	0	rs	rt	rd	0	111
jr	0	rs	0	0	0	1000
slt	0	rs	rt	rd	0	101010
sltu	0	rs	rt	rd	0	101011

I-Type

I-type	op	rs	rt	immediate
addi	1000	rs	rt	immediate
addiu	1001	rs	rt	immediate
andi	1100	rs	rt	immediate
lui	1111	0	rt	immediate
ori	1101	rs	rt	immediate
xori	1110	rs	rt	immediate
beq	100	rs	rt	immediate
bne	101	rs	rt	immediate
slti	1010	rs	rt	immediate
sltiu	1011	rs	rt	immediate
lw	100011	rs	rt	immediate
sw	101011	rs	rt	immediate

J-Type

J	op		address	
j	000010		address	j 10000
jal	000011		address	jal 10000

Instructions Quick Reference

Arithmetic and Logical Instructions

Instruction	Operation
add $d, $s, $t	$d = $s + $t
addu $d, $s, $t	$d = $s + $t
addi $t, $s, i	$t = $s + SE(i)

`addiu $t, $s, i`	$t = $s + SE(i)
`and $d, $s, $t`	$d = $s & $t
`andi $t, $s, i`	$t = $s & ZE(i)
`div $s, $t`	lo = $s / $t; hi = $s % $t
`divu $s, $t`	lo = $s / $t; hi = $s % $t
`mult $s, $t`	hi:lo = $s * $t
`multu $s, $t`	hi:lo = $s * $t
`nor $d, $s, $t`	$d = ~($s | $t)
`or $d, $s, $t`	$d = $s | $t
`ori $t, $s, i`	$t = $s | ZE(i)
`sll $d, $t, a`	$d = $t << a
`sllv $d, $t, $s`	$d = $t << $s
`sra $d, $t, a`	$d = $t >> a
`srav $d, $t, $s`	$d = $t >> $s
`srl $d, $t, a`	$d = $t >>> a
`srlv $d, $t, $s`	$d = $t >>> $s
`sub $d, $s, $t`	$d = $s - $t
`subu $d, $s, $t`	$d = $s - $t
`xor $d, $s, $t`	$d = $s ^ $t
`xori $d, $s, i`	$d = $s ^ ZE(i)

Constant-Manipulating Instturcions

Instruction	Operation
`lhi $t, i`	HH($t) = i
`llo $t, i`	LH($t) = i

Comparison Instructions

Instruction	Operation
`slt $d, $s, $t`	$d = ($s < $t)
`sltu $d, $s, $t`	$d = ($s < $t)
`slti $t, $s, i`	$t = ($s < SE(i))
`sltiu $t, $s, i`	$t = ($s < SE(i))

Branch Instructions

Instruction	Operation
`beq $s, $t, label`	if ($s == $t) pc += i << 2
`bgtz $s, label`	if ($s > 0) pc += i << 2
`blez $s, label`	if ($s <= 0) pc += i << 2
`bne $s, $t, label`	if ($s != $t) pc += i << 2

Jump Instructions

Instruction	Operation
`j label`	pc += i << 2
`jal label`	$31 = pc; pc += i << 2
`jalr $s`	$31 = pc; pc = $s
`jr $s`	pc = $s

Load Instructions

Instruction	Operation
`lb $t, i($s)`	$t = SE (MEM [$s + i]:1)

`lbu $t, i($s)`	$t = ZE (MEM [$s + i]:1)
`lh $t, i($s)`	$t = SE (MEM [$s + i]:2)
`lhu $t, i($s)`	$t = ZE (MEM [$s + i]:2)
`lw $t, i($s)`	$t = MEM [$s + i]:4

Store Instructions

Instruction	Operation
`sb $t, i($s)`	MEM [$s + i]:1 = LB ($t)
`sh $t, i($s)`	MEM [$s + i]:2 = LH ($t)
`sw $t, i($s)`	MEM [$s + i]:4 = $t

Data Movemenet Instructions

Instruction	Operation
`mfhi $d`	$d = hi
`mflo $d`	$d = lo
`mthi $s`	hi = $s
`mtlo $s`	lo = $s

Exception and Interrupt Instructions

Instruction	Operation
`trap 1`	Print integer value in $4
`trap 5`	Read integer value into $2
`trap 10`	Terminate program execution
`trap 101`	Print ASCII character in $4
`trap 102`	Read ASCII character into $2

Registers

Registers Overview

A CPU **register** is a temporary storage built into the CPU itself (separate from the memory)

MIPS has:

- 32 of 32-bit integer registers ($0 ~ $31), and
- 32 32-bit floating-point registers ($f0 ~ $f31).

Some of the registers are used for special purposes

Registers

Register #	Register Name	Register Description
`0`	`zero`	the value `0`
`1`	`$at`	(`assembler temporary`) reserved by the assembler
`2-3`	`$v0 - $v1`	(`values`) from expression evaluation and function results
`4-7`	`$a0 - $a3`	(`arguments`) First four parameters for subroutine. Not preserved across procedure calls

8-15	$t0 - $t7	(temporaries) Caller saved if needed. Subroutines can use w/out saving. Not preserved across procedure calls
16-23	$s0 - $s7	(saved values) Callee saved. A subroutine using one of these must save original and restore it before exiting. Preserved across procedure calls
24-25	$t8 - $t9	(temporaries) Caller saved if needed. Subroutines can use w/out saving. These are in addition to $t0 - $t7 above. Not preserved across procedure calls.
26-27	$k0 - $k1	reserved for use by the interrupt/trap handler (of OS)
28	$gp	global pointer Points to the middle of the 64K block of memory in the static data segment.
29	$sp	stack pointer Points to last location on the stack.
30	$s8/$fp	saved value / frame pointer Preserved across procedure calls
31	$ra	return address

Violet shading: Preserved

The glocbal pointer is for static data

Reserved Registers

The following reserved registers should not be used in user programs:

Register Name	Register Usage
$k0 - $k1	Reserved for OS
$at	Assembler Temporary
$gp	Global Pointer
$epc	Exception Program Counter

Miscellaneous Registers

In addition to previously listed registers, there are also some miscellaneous registers:

Register Name	Register Usage
$pc	Program Counter
$status, $psw	Status Register
$cause	Exception cause register

$hi	Used for multipication/division
$lo	

, **where**:

- `$pc` Points to the next instruction to be executed, not traditionally accessed by user programs
- `$status`/`$psw` is the process status register, updated by CPU after each instruction
- `$cause`, used in the event of an exception/interruption,

MIPS Instructions - Data Alteration

MIPS Field Types

MIPS Field Types An Introduction

MIPS instructions each occupy 32 bits of space.

Instructions have different formats, while their lengths remain the same.

MIPS instructions fall into those 3 categories:

`R-type`	For **Arithmetic Operations and such**
`I-type`	For **immediate** and **data transfer** instructions
`J-type`	For **Jump**

In memory, they are generally expressed such as:

`R-type, 32 bits`					
`000000`	`10001`	`10010`	`01000`	`00000`	`100000`
(6 bits)	(5 bits)	(5 bits)	(5 bits)	(5 bits)	(6 bits)

MIPS Fields-R

These instructions are identified by an opcode of 0, and differentiated by funct values.

R-Type instructions include `add` and `sub`:

op	rs	rt	rd	`shamt`	funct
(6 bits)	(5 bits)	(5 bits)	(5 bits)	(5 bits)	(6 bits)

, **where**:

- `op`: Basic operation of the instruction, also called the opcode
- `rs`: The first register source operand
- `rt`: The second register source operand
- `rd`: The register destination operand
- `shamt`: Shift amount
- `funct`: Function, specific variant of op

MIPS Fields-I

I-type instructions include `add immediate`, `lw` and `sw`:

In this case, the `Constant or Address` section serves as either the constant as an operand, or the int address offset.

op	rs	rt	Constant or Address
(6 bits)	(5 bits)	(5 bits)	(16 bits)

for example:

`lw $t0,32($s3) # Temporary reg $t0 gets A[8]`			
op	rs	rt	Constant or Address
…	`$s3`	`$t0`	`bin(32)`
…	19	8	`bin(32)`

MIPS Fields-J

op	address
(6 bits)	(26 bits)

Addition and Subtraction

R-Arithmetic Operands, addition and subtraction

`add $d, $s, $t`	`$d = $s + $t`
`addu $d, $s, $t`	`$d = $s + $t`

`addi $t, $s, i`	`$t = $s + SE(i)`
`addiu $t, $s, i`	`$t = $s + SE(i)`

`sub $d, $s, $t`	`$d = $s - $t`
`subu $d, $s, $t`	`$d = $s - $t`

More complicated performances can be executed such as:

```
# f = (g + h) – (i + j);
add $t0,$s1,$s2 # register $t0 contains g + h
add $t1,$s3,$s4 # register $t1 contains i + j
sub $s0,$t0,$t1 # f gets $t0 – $t1, which is (g + h)–(i + j)
```

Multiplication and Division: `mult`, `div`, `etc.`

Multiplication: mult and multu

In multiplication, the register stores the result into two separate 32-bit registers: the higher part in $hi, and the lower part in $lo.

`mult $s, $t`	`hi:lo = $s * $t`
`multu $s, $t`	`hi:lo = $s * $t`

Division: Accessing hi and lo

`div $s, $t`	`lo = $s / $t; hi = $s % $t`
`divu $s, $t`	`lo = $s / $t; hi = $s % $t`

Accessing $hi and $lo registers

`mfhi $d`	$d = hi
`mflo $d`	$d = lo

Application of Data Movements

Result of a multiplication (of two 32-bit numbers) is a 64-bit number, stored in two 32-bit registers, `$hi` and `$lo`:

```
mult $t1, $t2 # hi,lo = $t1 * $t2
mflo $t0 # $t0 = lo
mfhi $t3 # $t3 = hi
```

, **or**, using the *shorthand*,

```
mul $t0, $t1, $t2 # hi,lo = $t1 * $t2; $t0 = lo
```

Bit-Wise Operations

Logical Operations

It would be useful to operate on fields of bits within a word or even on individual bits.

Logical operations	C operators	Java operators	MIPS Instructions
Shift left	<<	<<	sll
Shift right	>>	>>>	srl
Bit-by-bit AND	&	&	and, andi
Bit-by-bit OR	|	|	or, ori
Bit-by-bit NOT	~	~	nor

There are two classes of logical operations:

- shifts: move all the bits in a word to the left or right, filling emptied bits with 0
    - Additionally, effectively

Shift instructions

`sll $d, $t, a`	$d = $t << a
`sllv $d, $t, $s`	$d = $t << $s

`sra $d, $t, a`	$d = $t >> a
`srav $d, $t, $s`	$d = $t >> $s

`srl $d, $t, a`	$d = $t >>> a
`srlv $d, $t, $s`	$d = $t >>> $s

In an **shift** instruction, the `shamt` section is no longer dormant:

op	rs	rt	rd	`shamt`	funct
(6 bits)	(5 bits)	(5 bits)	(5 bits)	<**THIS**^^>	(6 bits)

Bitwise AND/OR

`and $d, $s, $t`	$d = $s & $t
`andi $t, $s, i`	$t = $s & ZE(i)
`or $d, $s, $t`	$d = $s | $t
`ori $t, $s, i`	$t = $s | ZE(i)
`nor $d, $s, $t`	$d = ~($s | $t)
`xor $d, $s, $t`	$d = $s ^ $t
`xori $d, $s, i`	$d = $s ^ ZE(i)

The value of each single digit of the resulted register would depend on the values of the two arguments, such as:

- and: 1 if arguments are both 0

Example of AND
0000 0000 0000 0000 0000 1101 0000 0000
0000 0000 0000 0000 0011 1100 0000 0000
0000 0000 0000 0000 0000 1100 0000 0000

Logical Operations

Branch Instructions

`beq $s, $t, label`	if ($s == $t) pc += i << 2
`bgtz $s, label`	if ($s > 0) pc += i << 2
`blez $s, label`	if ($s <= 0) pc += i << 2
`bne $s, $t, label`	if ($s != $t) pc += i << 2

They imply branching when:

`beq $s, $t, label`	Equal
`bgtz $s, label`	Not equal
`blez $s, label`	Greater than 0
`bne $s, $t, label`	Less or equal to 0

These instructions jump to a specific `label` if certain conditions are met.

, **such as**:

```
# With the following correlation:
```

f	g	h	i	j
$s0	$s1	$s2	$s3	$s4

```
# The equivalent of if (i == j) f = g + h; else f = g – h;:

bne $s3,$s4,Else # go to Else if i ≠ j
add $s0,$s1,$s2 # f = g + h (skipped if i ≠ j)
j Exit # go (jump) to Exit
Else:sub $s0,$s1,$s2 # f = g – h (skipped if i = j)
Exit:
```

Jump Instructions

`j label`	pc += i << 2
`jr $rs`	pc = $s
`jal label`	$31 = pc; pc += i << 2
`jalr $rs`	$31 = pc; pc = $s

`j label`	Jump to `label`
`jr $rs`	Jump to `$rs`
`jal label`	Jump to `label`,
`jalr $rs`	Jump to `$rs`, store next line in $rd

Among them, `jal` and `jalr` perform a super-set of the operations of `j`/`jr`

`jal`/`jalr` and `j`/`hr` can be implemented one in terms of the other:

```
Emulate JAL/JALR with J/JR
Original                Emulated

jal foo                 la $ra, ret_label
                        j foo
                       ret_label:
```

```
Emulate J/JR with JAL/JALR
Original                 Emulated
```

```
j foo                       prolog:
                              addi $sp, $sp, -4
                              sw $ra, ($sp)

                              jal foo

                            epilog:
                              lw $ra, ($sp)
                              addi $sp, $sp, 4
```

Set If Instructions

`slt $d, $s, $t`	$d = ($s < $t)
`sltu $d, $s, $t`	$d = ($s < $t)
`slti $t, $s, i`	$t = ($s < SE(i))
`sltiu $t, $s, i`	$t = ($s < SE(i))

`seq`	Set on Equak
`slt`	Set on Less Than
`sle`	Set no Less than or Equal

For instructions such as seq $t1, $t2, $t3:

Register in place of $t1 would be set to 1 if the following conditions were met, otherwise 0.

Requirements such as:

`seq`	`$t2 == $t3`
`slt`	`$t2 < $t3`
`sle`	`$t2 <= $t3`

`slt` and `slti` can be used in combination with `beq` and `bne`:

```
slt $t0, $s1, $s2 # if ($s1 < $s2)
bne $t0, $zero, L # branch to L
```

, because:

- Hardware for < and >=, … ar slower than = and $\neq$
- Combining with branch involves more work per instruction, requiring a slower clock
- All instructions penalized
- `beq` and `bne` are common case

Set If Instructions: Signed / Unsigned Comparison

Unsigned comparisons compare registers as if they were unsigned integers:

`slt, slti`	Signed comparison
`sltu`	Set on Less Than, unsigned
`sltui`	Set on Less Than Immediate, unsigned

- For example, with

```
$s0 = 1111 1111 1111 1111 1111 1111 1111 1111
$s1 = 0000 0000 0000 0000 0000 0000 0000 0001
```

, **thus**:

`slt $t0, $s0, $s1 # signed`	`–1 < +1 → $t0 = 1`
`sltu $t0, $s0, $s1 # unsigned`	`+4,294,967,295 > +1 → $t0 = 0`

Operands can also be compared with a constant:

```
slti $t0,$s2,10 # $t0 = 1 if $s2 < 10
```

The MIPS architecture doesn't include branch on less than because it is too complicated.

Constant-Manipulating Instructions

Constant-Manipulating Operations

`lhi $t, i`	Load high immediate, load upper half of register with immediate `i`
`llo $t, i`	Load low immediate, load lower half of register with immediate `i`

MIPS Components - Memory and Data

Memory Overview

A Program - the Memory Layout:

High Memory	Stack
	Heap
	Uninitialized Data
	Data
	Text (Code)
Lowest Memory	Reserved

, **where**:

- `Stack` starts in high memory and grows downward
- `Heap` is where dynamically allocated data will be stored
- `Uninitialized data` contains declared variables whose values are yet to be assigned
- `Data` is where the initialized data is stored
- `Text` (code) is where the machine language is stored
- `Reserved` is not available to uer programs

Memory Range

To store a word, four bytes are required (which uses four memory addresses)

Name	Example	Comments
2^{30} memory words	`Memory[0], Memory[4], …, Memory[4294967292]`	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, arrays, and spilled registers.

Addressing Schemes

In addition to associating variables with registers, the compiler allocates data structures like arrays and structures to locations in memory.

The compiler can then place the proper starting address into the data tansfer instructions.

word			
byte	byte	byte	byte

MIPS address individual bytes, thus

Address of a word matches that of one of the 4 bytes within the word

(Hence, address of sequential words differ by 4)

Alignment Restriction, Big and Little Endian

Alignment Restriction: In MIPS, words must start at addresses that are multiples of 4.

Furthermore, addressing schemes are divided into big endian and little endian:

- **Big Endian**: address (word) == address (leftmost address of the word)
    - (MIPS is also in this league)
- **Little Endian**: address (word) == address (rightmost address of the word)

Big Endian
word

byte	byte	byte	byte

Little Engian			
word			
byte	byte	byte	byte

Byte address also affect the array index. To get the proper byte address in the code above, the offset has to be 4 × 8 = 32, so that the `lw` instruction would select the param/4th entry, rather than the paramth entry:

		20	100
		16	10
		12	101
		8	1
		4	110
		0	
		Address	Data
Processor		Memory	

```
# A[12] = h + A[8]
# h=$s2, A=$s3
lw    $t0,32($3)        # Temporary reg $t0 gets A[8] (not 32!)
add   $t0,$s2,$t0       # Temporary reg $t0 gets h + A[8]
lw    $t0,32($3)        # Stores h + A[8] back into A[12], using 48
as offset and register $s3 as the base register
```

Memory IO

Load Instructions

Load byte, load half, load word

`lb $t, i($s)`	$t = SE (MEM [$s + i]:1)
`lbu $t, i($s)`	$t = ZE (MEM [$s + i]:1)
`lh $t, i($s)`	$t = SE (MEM [$s + i]:2)
`lhu $t, i($s)`	$t = ZE (MEM [$s + i]:2)
`lw $t, i($s)`	$t = MEM [$s + i]:4

Store Instructions

`sb $t, i($s)`	MEM [$s + i]:1 = LB ($t)
`sh $t, i($s)`	MEM [$s + i]:2 = LH ($t)
`sw $t, i($s)`	MEM [$s + i]:4 = $t

Applying Loading & Storing Instructions

Taking `lw` and `sw` as an example:

`lw $t, i($s)`	Load word, load stuff from memory into register
`sw $t, i($s)`	Save word, put stuff from register into memory

The size of register (<< *where arithmetic operations occur*) is tiny compared to memory.
Instructions that transfer data between memory and register are data transfer instructions.

To access a word in memory, the instruction must supply the memory address

Compiling an Assignment When an Operand is in Memory

```
# g = h + A[8]

# g at $s1, h at $s2, A at $s3
# The address of this array element is the sum of the base of the
array A, found in register $s3, plus the number to get element 8

lw $t0,8($s3) # Temporary reg $t0 gets A[8]

add $s1,$s2,$t0 # g = h + A[8]
# The constant in a data transfer instruction is called the offset,
and the register added to form the address is called the base
register.
```

Moving Data Between Registers

Accessing hi and lo

`move $d $s`	$d = $s

`mfhi $d`	$d = hi
`mflo $d`	$d = lo

`mthi $s`	hi = $s
`mtlo $s`	lo = $s

Accessing the Stack

Spilling

Spilling is the process of putting less commonly used variables into memory.

Although:

Data is more useful in the register
Register is smaller, thus faster.

Stack In

Sometimes, we might need to store their old values in a stack:

```
addi $sp,$sp,-12 # adjust stack to make room for 3 items
sw $t1, 8($sp) # save register $t1 for use afterwards
sw $t0, 4($sp) # save register $t0 for use afterwards
sw $s0, 0($sp) # save register $s0 for use afterwards
```

Example, what to Do after Stack In

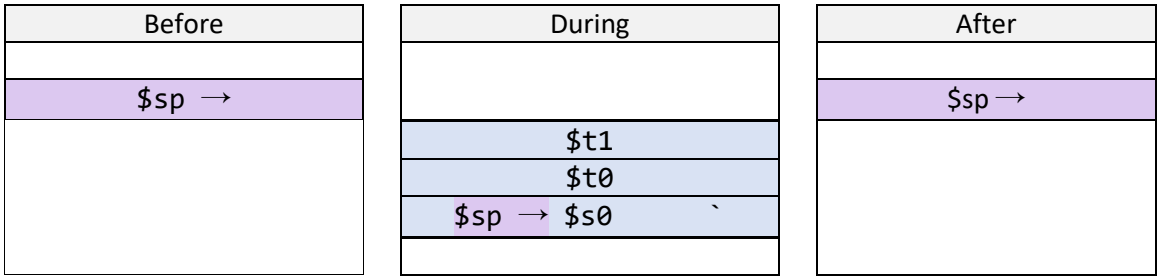
```
add $t0,$a0,$a1 # register $t0 contains g + h
add $t1,$a2,$a3 # register $t1 contains i + j
sub $s0,$t0,$t1 # f = $t0 – $t1, which is (g + h)–(i + j)

add $v0,$s0,$zero # returns f ($v0 = $s0 + 0)
```

Stack Out

```
lw $s0, 0($sp) # restore register $s0 for caller
lw $t0, 4($sp) # restore register $t0 for caller
lw $t1, 8($sp) # restore register $t1 for caller
addi $sp,$sp,12 # adjust stack to delete 3 items
```

Stack Access Visualized

In these steps, stack can be visualized as:

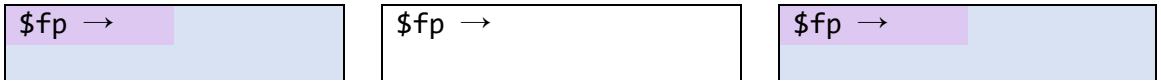


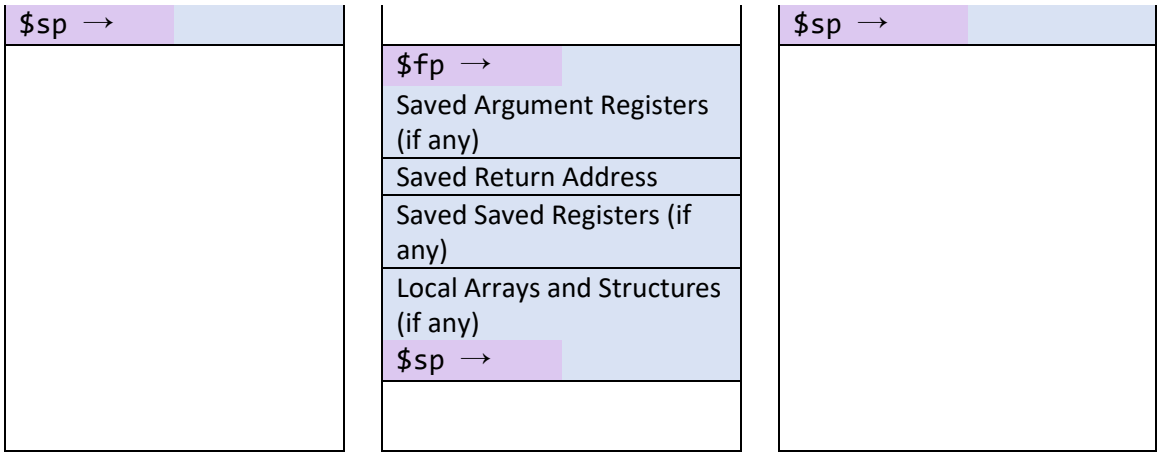
, **where**:

- This shade denotes `$sp`, the stack pointer
- This denotes stored data

Local Data on the Stack

$fp	Frame Pointer	Points to beginning of the stack frame
$sp	Stack Pointer	Points to end of the stack frame





When a frame pointer is used, it is initialized using the address in $sp on a call, and $sp is restored using $fp.

`$fp` is needed to access arguments and local variables in case it is not statically known how much the stack will grow (variable length parameters and dynamic local arrays)

The stack is also used to store variables that are local to the procedure that do not fit in the registers (local arrays and structures). Procedure frame / activation record: The segment of the stack containing a procedure's saved registers and local variables

Some MIPS software use a frame pointer (`$fp`) to point to the first word of the frame of a procedure: Stack pointer might change during the procedure, so references to a local variable in memory might have different offsets depending on where they are in the procedure, making them hard to understand.

Alternatively, a `$fp` offers a stable base register within a procedure for local memory references.

An activation record appears on the stack whether or not an explicit frame pointer is used.

Exception and Interrupt Instruction

`trap 1`	Print integer value in $4
`trap 5`	Read integer value into $2
`trap 10`	Terminate program execution
`trap 101`	Print ASCII character in $4
`trap 102`	Read ASCII character into $2

Procedure Call

Procedure Call Components - Registers

Registers necessary for Procedure Calls

MIPS allocates those registers for procedure calling:

`$a0 - $a3`	Four argument registers in which to pass parameters
`$v0 - $v1`	Two value registers in which to return values
`$ra`	Return address register

Temporary Registers

To avoid saving & restoring a register whose value is never used, MIPS separates 18 of the registers into two groups:

`$t0 - $t9`	10 temporary registers that are not preserved by the callee
`$s0 - $s7`	8 registers that must be preserved on a procedure call (If used, the callee must save and restore them)

Procedure Call Components - Instructions

Concepts

In the execution of a procedure, such as

```
int leaf (int g, h, i, j)
{
    int f;
    f = (g + h) - (i + j);
    return f;
}
```

, **the program** must follow theses six steps:

1. Place parameters in a place where the procedure can access them.
2. Transfer control to the procedure.
3. Acquire the storage resources needed for the procedure.
4. Perform the desired task.
5. Place the result value in a place where the calling program can access it.
6. Return control to the point of origin.

jal: Procedure call, Jump and Return

`jal Label`	

Procedure Call: **Jump and Link**
`jal <Label>`

```
Address of following instruction puts in $ra
Jumps to Label: sets program counter to <Label>
```

Here, "link" means that an address/link is formed to the calling site (stored in $ra) that llows the procedure to return to the proper address

`jal` saves PC+4 in register $ra to link to the following instruction to set up the procedure return.

jr: Procedure Return

`jr $ra`	

Procedure Return: **Jump register**

`jr $ra`
`Copies $ra to program counter $pc`
`Can also be used for computer jumps, for case/switch statements`

What Happens in a Procedure call:

- Caller puts parameters in `$a0 ~ $a3`
- ↓ `jal <Label of the Callee>` to the callee
- ↓ Callee performs the calculations,
- ↓ Callee stores results in `$v0 ~ $v1`
- ✓ `jr $ra` back to the caller

The Parameter Problem

Elaboration

"What if there are more than four parameters? The MIPS convention is to place the extra parameters on the stack just above the frame pointer. The procedure then expects the first four parameters to be in registers $a0 through $a3 and the rest in memory, addressable via the frame pointer."

Procedure Examples

Leaf Procedure Example

```
int leaf (int g, h, i, j)
{
    int f;
    f = (g + h) - (i + j);
    return f;
}
# Arguments g, …, j in $a0, …, $a3
# f in $s0 (hence, need to save $s0 on stack)
# Result in $v0
```

leaf:	

`addi $sp, $sp, -4`	Save $s0 on stack
`sw $s0, 0($sp)`	
`add $t0, $a0, $a1`	Procedure body
`add $t1, $a2, $a3`	
`sub $s0, $t0, $t1`	
`add $v0, $s0, $zero`	Result
`lw $s0, 0($sp)`	Restore $s0
`addi $sp, $sp, 4`	
`jr $ra`	Return

Non-Leaf Procedures

Procedures that call other procedures: caller needs to save on the stack:

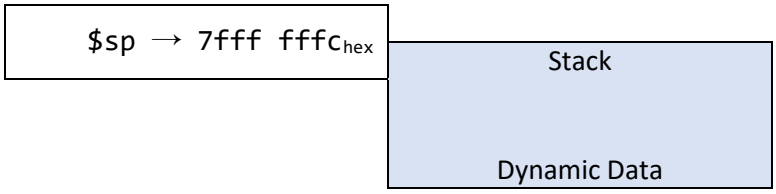
- Its return address
- Any arguments and temporaries needed after the call

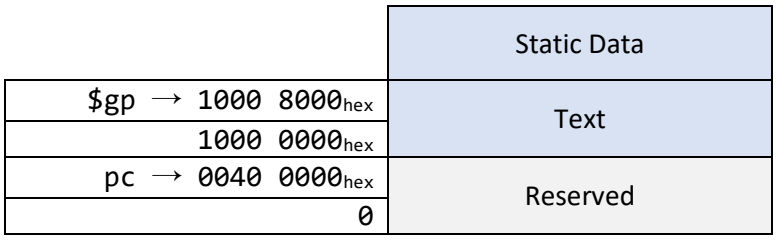
Such as this:

```
int fact (int n)
{
    if (n < 1) return 1;
    else return n * fact(n - 1);
}
```

```
fact:
    addi $sp, $sp, -8 # adjust stack for 2 items
    sw $ra, 4($sp) # save return address
    sw $a0, 0($sp) # save argument
    slti $t0, $a0, 1 # test for n < 1
    beq $t0, $zero, L1
    addi $v0, $zero, 1 # if so, result is 1
    addi $sp, $sp, 8 # pop 2 items from stack
    jr $ra # and return
L1: addi $a0, $a0, -1 # else decrement n
    jal fact # recursive call
    lw $a0, 0($sp) # restore original n
    lw $ra, 4($sp) # and return address
    addi $sp, $sp, 8 # pop 2 items from stack
    mul $v0, $a0, $v0 # multiply to get result
    jr $ra # and return
```

Memory Layout





, **where**:

- Text: Program code
- Static data: global variables e.g. static variables in C
- Dynamic data: heap, e.g. malloc in C, new in Java, object creation in Python…
- Stack: automatic storage

MIPS Components, Meta Information

Labels

Concepts

Labels are cod locations, typically used as:

- To indicate a function/procedure name, or
- as target of a jump

{WARNING}
Note that Labels are case-sensitive, meaning Loop and loop can be two separate things.

When a label appears along on a line, it refers to the following memory location.

main:

The label main: should point to the first instruction of the progrm, such as:

```
# Daniel J. Ellard -- 02/21/94
# add.asm-- A program that computes the sum of 1 and 2,
#     leaving the result in register $t0.
# Registers used:
#     t0 - used to hold the result.
#     t1 - used to hold the constant 1.
main: li $t1, 1 # load 1 into $t1.
      add $t0, $t1, 2 # $t0 = $t1 + 2.
# end of add.asm
```

Size Workarounds

The 32-bit Constant Problem

The Problem Ahead

As a matter of fact…
One immediate operand can be no more than 16 bytes (a halfword) long Thus, to load an 32-bit immediate value (constant) into a register, it needs to be loaded 16 bits at a time

`lui $s Imm`	`Load Upper Immedaite`

`lui` (load upper immediate) specifically sets the ***upper 16*** bits of a constant in a register, allowing a subsequent instruction to specify the ***lower 16*** bits in the constant.

After command `lui $t0, 255:`
Register `$t0` becomes

0000 0000 0000 0000	0000 0000 0000 0000
↓	
0000 0000 1111 1111	0000 0000 0000 0000

Example: Loading a 32-bit constant

For example, when loading a 32-bit constant into register `$s0`:

0000 0000 0011 1101 0000 1001 0000 0000

`lui rt, constant`		
`lui $s0, 61`	0000 0000 0011 1101	0000 0000 0000 0000
`ori $s0, $s0, 2304`	0000 0000 0011 1101	0000 1001 0000 0000

`lui $s0, 61 # 61 decimal =` 0000 0000 0011 1101 binary

0000 0000 0011 1101	0000 0000 0000 0000

`ori $s0, $s0, 2304 # 2304 decimal =` 0000 1001 0000 0000

0000 0000 0011 1101	0000 1001 0000 0000

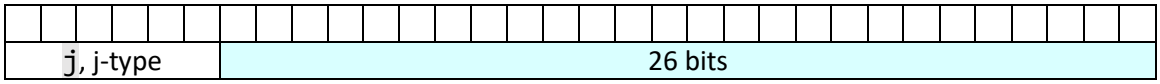
Elaborations
Creating 32-bit constants needs care. The instruction `addi` copies the

leftmost bit of the 16-bit immediate field of the instruction into the upper 16 bits of a word. Logical or immediate from Section 2.5 loads 0s into the upper 16 bits and hence is used by the assembler in conjunction with `lui` to create 32-bit constants.

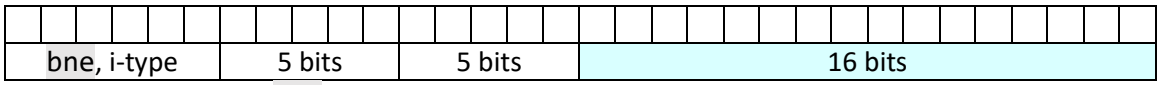
Addressing in Branches and Jumps

The Addressing Size Problem

Although jump instructions (with j format) has 26 bits for a space:



, **however**,



in instructions such as `bne`, there are only 16 bits for the branch address

PC Relative Addressing (< 2^{16} addresses away)

`Program counter = Register + Branch address`

Since most branching actions happen less than 2^{16} addresses away, PC-relative addressing can be used.

Since it's convenient for the hardware to increment the PC early to point to the next instruction, MIPS address is actually relative to the address of the following instruction (PC + 4) as opposed to PC itself.

For example, an while loop:

```
Loop:sll $t1,$s3,2 # Temp reg $t1 = 4 * i
    add $t1,$t1,$s6 # $t1 = address of save[i]
    lw $t0,0($t1) # Temp reg $t0 = save[i]
    bne $t0,$s5, Exit # go to Exit if save[i] ≠ k
    addi $s3,$s3,1 # i = i + 1
    j Loop # go to Loop
Exit:
```

If we assume we place the loop starting at location 80000 in memory, the assembled instruction wold look like:

`Loop: sll`	80000	0	0	19	9	4	0
`add`	80004	0	9	22	9	0	32
`lw`	80008	35	9	8	0		
`bne`	80012	5	8	21	2 (relative addressing)		
`addi`	80016	8	19	19	1		
`j Loop`	80020	20000					
	80024	…					

PC-relative addressing: An addressing regime in which the address is the sum of the program counter (PC) and a constant in the instruction.

Really Far Branching (> 2^{16} addresses away)

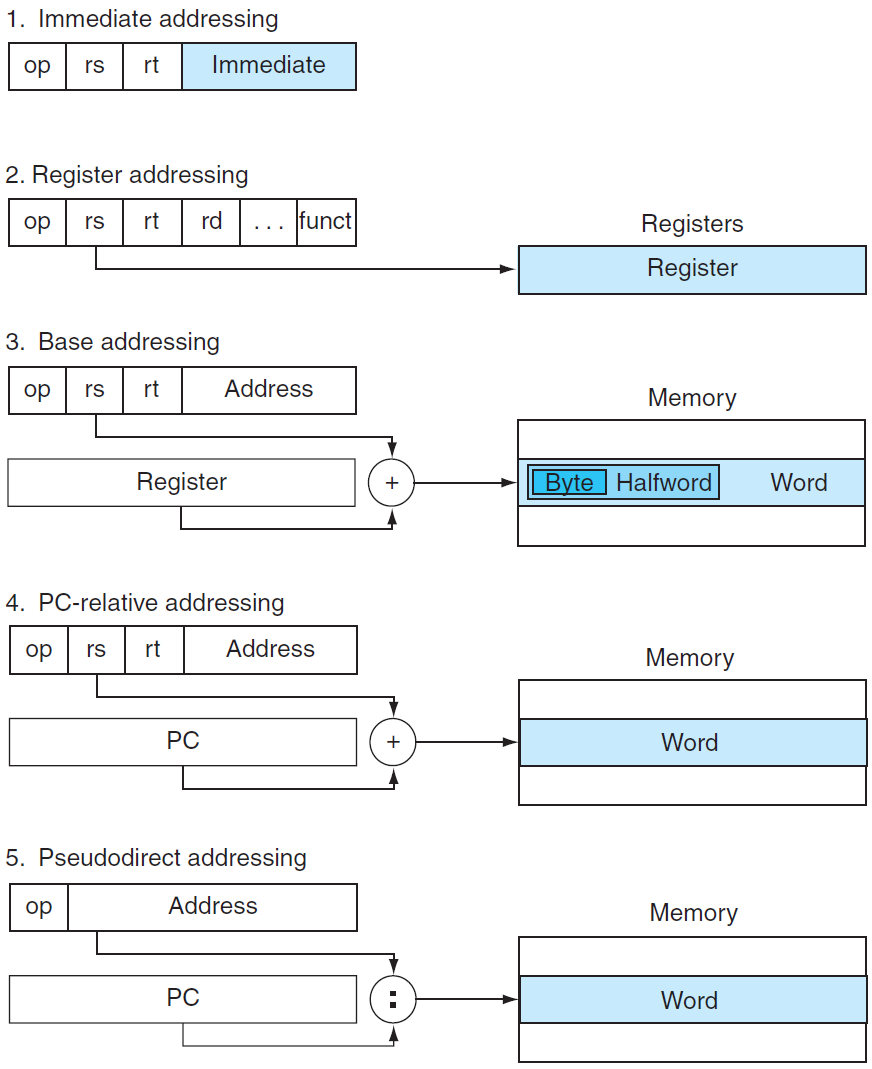
Nearly every conditional branch is to a nearby location, but occasionally it branches far away, farther than can be represented in the 16 bits of the conditional branch instruction. The assembler comes to the rescue just as it did with large addresses or constants: it inserts an unconditional jump to the branch target, and inverts the condition so that the branch decides whether to skip the jump.

```
    beq     $s0, $s1, L1
    bne     $s0, $s1, L2
    j       L1
L2:
```

MIPS Addressing Summary

Addressing Modes

1. Register addressing	Where the operand is a register
2. Base/displacement addressing	Operand is at he memory location whose addressing is the sum of a register and constant in the instruction
3. Immediate addressing	Operand is a constant within he instruction itself
4. PC-relative addressing	Address is the sum of PC and constant in the instruction
5. Pseudo-direct addressing	Jump address is the 26 bits of the instruction concatenated with the upper bits of the PC



Note that a single operation can use more than one addressing mode. Add, for example, uses both immediate (addi) and register (add) addressing.

Implementation of Advanced Utilities

ASCII String Operation (Such as, C)

ASCII String Overview

In this example, we would use the ASCII encoding scheme where each and every character is resented on a byte basis (0 - 7f(DEL))

there are three choices for re[resenting a sring:

(1) The first position of the string is reserved to give the length of a string
(2) (As in a Structure), accompanying variable has length of the string
(3) The last position of a string is indicated by a character used to mark the end of a string

C's approach to strings
C uses option (3) to represent a string, namely puts an 0 (`nul`) at the end of every string

Loading Strings

`lb`	Load byte (halfword)
`sb`	Store byte
`lbu`	Load byte, unsigned

```
lb rt, <offset>(rs) #sign extend to 32 bits
lbu rt, <offset>(rs) #zero extend to 32 bits
sb rt, <offset>(rs) #store rightmost byte
```

Example: A String Copy Procedure

To implement the following algorithm (?):

```
void strcpy (char x[], char y[])
{
   int i;
   i = 0;
   while ((x[i]=y[i])!='\0')
      i += 1;
}
```

Written with the following correspondance:

x	y	i
$a0	$a1	$s0

```
strcpy:

    # (Adjust the stack pointer, save $s0 first)
    addi $sp, $sp, -4 # adjust stack for 1 item
    sw $s0, 0($sp) # save $s0

    # (Initialize i==0)
    add $s0, $zero, $zero # i = 0

    # Beginning of the Loop, form y[i] y adding $s0(i) to $a1(y)
    # (No need to *4, since it's based on byte not word)
L1: add $t1, $s0, $a1 # addr of y[i] in $t1

    # Then, load $t1(the offset)
    lb $t2, 0($t1) # $t2 = y[i]

    # Similarly, acquire address of x[i]
    add $t3, $s0, $a0 # addr of x[i] in $t3
```

```
    # Save y[i] in x[i]
    sb $t2, 0($t3) # x[i] = y[i]

    # Exit the loop if it's last character of the string (str == nul)
    beq $t2, $zero, L2 # exit loop if y[i] == 0

    # Else, increment it and repeat the loop
    addi $s0, $s0, 1 # i = i + 1

    j L1 # Start next iteration of loop
    # Exit condition
L2: lw $s0, 0($sp) # restore saved $s0
    addi $sp, $sp, 4 # pop 1 item from stack
    jr $ra # and return to $ra
```

Unicode String Operations (Such as, Java)

Unicode String overview

In Unicode, a character is represented by 16 bits:

`0x????`

(Two bytes), also called halfwords

Loading Strings

`lh`	Load halfword
`sh`	Store halfword
`lhu`	Load halfword, unsigned

In Java includes a word that gives the length of the string, similar to Java arrays.

Addressing Methods

Branch Addressing

Branch instructions specify opcode two registers target address

`beq rs, rt, L1`

PC-relative addressing (PC already incremented by 4):
`target address = PC + offset × 4`

Jump Addressing

`j` and `jal` targets could be anywhere in the code

(Pseudo) direct addressing
target address = $PC_{31..28}$: address × 4

Target Addressing Example

To implement

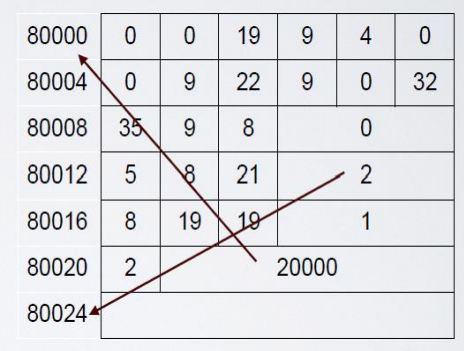
```
while (save[i] == k)
    i += 1;
```

,

With the following correspondence:

i	k	save
$s3	$s5	$s6

```
Loop: sll $t1, $s3, 2
    add $t1, $t1, $s6
    lw $t0, 0($t1)
    bne $t0, $s5, Exit
    addi $s3, $s3, 1
    j Loop
Exit: …
```



Branching Far Away

If branch target is too far to encode with 16-bit offet, assembler rewrites the code:

```
    beq $s0,$s1, L1
     ↓
    bne $s0,$s1, L2
    j L1
L2: …
```

Address Mode Summary

```
addi $s3, $s3, 1
sub $s0, $s1, $s2
lw $t0, 32($s3)
bne $t0, $zero, L
j Exit
```

Execution Structures

if Statements

```
if (i==j)
    f = g+h;
```

```
else
    f = g-h;
```

```
    bne $s3, $s4, Else
    add $s0, $s1, $s2
    j Exit
Else: sub $s0, $s1, $s2
Exit: …
```

for Statements

```
short A[10];
...
for (int i = 0; i < 10; i++) {
    A[i] = i;
}
```

```
short A[10];
...
int i = 0;
while (i < 10) {
    A[i] = i;
    i++
}
```

while statements

```
while (save[i] == k)
    i += 1;
```

```
Loop: sll $t1, $s3, 2
    add $t1, $t1, $s6
    lw $t0, 0($t1)
    bne $t0, $s5, Exit
    addi $s3, $s3, 1
    j Loop
Exit: …
```

End of Program

Syscalls

Concepts

If a program was not terminated at the end, it may:

will blunder on through memory, interpreting whatever it finds as instructions to execute

… **which** is definitely what we'd want to happen.

The `syscall` instruction instructs the program o give control back to the OS., the OS would then look at register $v0 to see what the program asks it to do.

… for example,

```
    li $v0, 10 # syscall code 10 is for exit.
    syscall # make the syscall.
```

Appendix: Pseudo-instructions

Overview

Concepts

In MIPS, some operations can be performed with the help of other instructions. They can be coded in assembly language, and assembler will expand them to rel instructions.

The exact expansion is compiler-defined, bue the result would be similar to:

Data Moves

Assembly syntax	Name	Expansion	Operation in C
`move $t, $s`	move	or $t, $s, $zero	t = s
`clear $t`	clear	or $t, $zero, $zero	t = 0
`li $t, C`	load 16-bit immediate	ori $t, $zero, C_lo	t = C
`li $t, C`	load 32-bit immediate	lui $t, C_hi ori $t, $t, C_lo	t = C
`la $t, A`	load label address	lui $t, A_hi ori $t, $t, A_lo	t = A

Branches

Assembly syntax	Name	Expansion
b C	branch unconditionally	beq $zero, $zero, C
bal C	branch unconditionally and link	bgezal $zero, C
bgt $s, $t, C	branch if greater than	slt $at, $t, $s bne $at, $zero, C
blt $s, $t, C	branch if less than	slt $at, $s, $t bne $at, $zero, C
bge $s, $t, C	branch if greater than or equal	slt $at, $s, $t beq $at, $zero, C

ble $s, $t, C	branch if less than or equal	slt $at, $t, $s beq $at, $zero, C
bgtu $s, $t, C	branch if greater than unsigned	sltu $at, $t, $s bne $at, $zero, C
beqz $s, C	branch if zero	beq $s, $zero, C
beq $t, V, C	branch if equal to immediate	ori $at, $zero, V beq $t, $at, C
bne $t, V, C	branch if not equal to immediate	ori $at, $zero, V bne $t, $at, C

Multiplication / Division

Assembly syntax	Name	Expansion	Operation in C
mul $d, $s, $t	multiplicate and return 32 bits	mult $s, $t mflo $d	d = (s * t) & 0xFFFFFFFF
div $d, $s, $t	quotient	div $s, $t mflo $d	d = s / t
rem $d, $s, $t	remainder	div $s, $t mfhi $d	d = s % t

Jumps

Assembly syntax	Name	Expansion	Operation in C
jalr $s	jump register and link to ra	jalr $s, $ra	ra = PC + 4; goto s;

Logical Operations

Assembly syntax	Name	Expansion	Operation in C
not $t, $s	not	nor $t, $s, $zero	t = ~s

No-operations

Assembly syntax	Name	Expansion	Operation in C
nop	nop	sll $zero, $zero, 0	{}