MARS

Registers
Instruction Set
Directives
Syscalls

ΗΥ 134: ΕΙΣΑΓΩΓΗ ΣΤΗΝ ΟΡΓΑΝΩΣΗ ΚΑΙ ΣΧΕΔΙΑΣΗ Η/Υ Ι

Registers

MIPS has 32 integer registers.

The hardware architecture specifies that:

- General purpose register $0 always returns a value of 0.
- General purpose register $31 is used as the link register for jump and link instructions.
- HI and LO are used to access the multiplier/divider results, accessed by the mfhi (move from high) and mflo commands.

These are the only hardware restrictions on the usage of the general purpose registers.

The various MIPS tool-chains implement specific calling conventions that further restrict how the registers are used. These calling conventions are totally maintained by the tool-chain software and are not required by the hardware.

Register	Number	Usage
zero	0	Constant 0
at	1	Reserved for assembler
v0	2	Used for return values from function calls.
v1	3	
a0	4	Used to pass arguments to procedures and functions.
a1	5	
a2	6	
a3	7	
t0	8	Temporary (Caller-saved, need not be saved by called procedure)
t1	9	
t2	10	
t3	11	
t4	12	
t5	13	
t6	14	
t7	15	
s0	16	Saved temporary (Callee-saved, called procedure must save and restore)
s1	17	
s2	18	
s3	19	
s4	20	
s5	21	
s6	22	
s7	23	
t8	24	Temporary (Caller-saved, need not be saved by called procedure)
t9	25	

k0	26	Reserved for OS kernel
k1	27	
gp	28	Pointer to global area
sp	29	Stack pointer
fp	30	Frame pointer
ra	31	Return address for function calls.

MIPS has 32 floating-point registers ($f0 – $f31). Two registers are paired for double precision numbers. Odd numbered registers cannot be used for arithmetic or branching, just as part of a double precision register pair.

Instruction Set

The following list provides a description of basic MIPS instructions. For more information on pseudo-instructions available in MARS, refer to MARS help section.

Operand Key for Example Instructions	
label, target	any textual label
$t1, $t2, $t3	any integer register
$f2, $f4, $f6	even-numbered floating point register
$f0, $f1, $f2	any floating point register
$8	any Coprocessor 0 register
1	condition flag (0 to 7)
10	unsigned 5-bit integer (0 to 31)
-100	signed 16-bit integer (-32768 to 32767)
100000	signed 32-bit integer (-2147483648 to 2147483647)
Load & Store addressing mode	
-100($t2)	sign-extended 16-bit integer added to contents of $t2

Basic Instructions	
`abs.d $f2,$f4`	Floating point absolute value double precision : Set $f2 to absolute value of $f4, double precision
`abs.s $f0,$f1`	Floating point absolute value single precision : Set $f0 to absolute value of $f1, single precision
`add $t1,$t2,$t3`	Addition with overflow : set $t1 to ($t2 plus $t3)
`add.d $f2,$f4,$f6`	Floating point addition double precision : Set $f2 to double-precision floating point value of $f4 plus $f6
`add.s $f0,$f1,$f3`	Floating point addition single precision : Set $f0 to single-precision floating point value of $f1 plus $f3
`addi $t1,$t2,-100`	Addition immediate with overflow : set $t1 to ($t2 plus signed 16-bit immediate)
`addiu $t1,$t2,-100`	Addition immediate unsigned without overflow : set $t1 to ($t2 plus signed 16-bit immediate), no overflow
`addu $t1,$t2,$t3`	Addition unsigned without overflow : set $t1 to ($t2 plus $t3), no overflow
`and $t1,$t2,$t3`	Bitwise AND : Set $t1 to bitwise AND of $t2 and $t3
`andi $t1,$t2,100`	Bitwise AND immediate : Set $t1 to bitwise AND of $t2 and zero-extended 16-bit immediate
`bc1f 1,label`	Branch if specified FP condition flag false (BC1F, not BCLF) : If Coprocessor 1 condition flag specified by immediate is false (zero) then branch to statement at label's address
`bc1f label`	Branch if FP condition flag 0 false (BC1F, not BCLF) : If Coprocessor 1 condition flag 0 is false (zero) then branch to statement at label's address

`bc1t 1,label`	Branch if specified FP condition flag true (BC1T, not BCLT) : If Coprocessor 1 condition flag specified by immediate is true (one) then branch to statement at label's address
`bc1t label`	Branch if FP condition flag 0 true (BC1T, not BCLT) : If Coprocessor 1 condition flag 0 is true (one) then branch to statement at label's address
`beq $t1,$t2,label`	Branch if equal : Branch to statement at label's address if $t1 and $t2 are equal
`bgez $t1,label`	Branch if greater than or equal to zero : Branch to statement at label's address if $t1 is greater than or equal to zero
`bgezal $t1,label`	Branch if greater than or equal to zero and link : If $t1 is greater than or equal to zero, then set $ra to the Program Counter and branch to statement at label's address
`bgtz $t1,label`	Branch if greater than zero : Branch to statement at label's address if $t1 is greater than zero
`blez $t1,label`	Branch if less than or equal to zero : Branch to statement at label's address if $t1 is less than or equal to zero
`bltz $t1,label`	Branch if less than zero : Branch to statement at label's address if $t1 is less than zero
`bltzal $t1,label`	Branch if less than zero and link : If $t1 is less than or equal to zero, then set $ra to the Program Counter and branch to statement at label's address
`bne $t1,$t2,label`	Branch if not equal : Branch to statement at label's address if $t1 and $t2 are not equal
`break`	Break execution : Terminate program execution with exception
`break 100`	Break execution with code : Terminate program execution with specified exception code
`c.eq.d $f2,$f4`	Compare equal double precision : If $f2 is equal to $f4 (double-precision), set Coprocessor 1 condition flag 0 true else set it false
`c.eq.d 1,$f2,$f4`	Compare equal double precision : If $f2 is equal to $f4 (double-precision), set Coprocessor 1 condition flag specified by immediate to true else set it to false
`c.eq.s $f0,$f1`	Compare equal single precision : If $f0 is equal to $f1, set Coprocessor 1 condition flag 0 true else set it false
`c.eq.s 1,$f0,$f1`	Compare equal single precision : If $f0 is equal to $f1, set Coprocessor 1 condition flag specied by immediate to true else set it to false
`c.le.d $f2,$f4`	Compare less or equal double precision : If $f2 is less than or equal to $f4 (double-precision), set Coprocessor 1 condition flag 0 true else set it false
`c.le.d 1,$f2,$f4`	Compare less or equal double precision : If $f2 is less than or equal to $f4 (double-precision), set Coprocessor 1 condition flag specfied by immediate true else set it false

`c.le.s $f0,$f1`	Compare less or equal single precision : If $f0 is less than or equal to $f1, set Coprocessor 1 condition flag 0 true else set it false
`c.le.s 1,$f0,$f1`	Compare less or equal single precision : If $f0 is less than or equal to $f1, set Coprocessor 1 condition flag specified by immediate to true else set it to false
`c.lt.d $f2,$f4`	Compare less than double precision : If $f2 is less than $f4 (double-precision), set Coprocessor 1 condition flag 0 true else set it false
`c.lt.d 1,$f2,$f4`	Compare less than double precision : If $f2 is less than $f4 (double-precision), set Coprocessor 1 condition flag specified by immediate to true else set it to false
`c.lt.s $f0,$f1`	Compare less than single precision : If $f0 is less than $f1, set Coprocessor 1 condition flag 0 true else set it false
`c.lt.s 1,$f0,$f1`	Compare less than single precision : If $f0 is less than $f1, set Coprocessor 1 condition flag specified by immediate to true else set it to false
`ceil.w.d $f1,$f2`	Ceiling double precision to word : Set $f1 to 32-bit integer ceiling of double-precision float in $f2
`ceil.w.s $f0,$f1`	Ceiling single precision to word : Set $f0 to 32-bit integer ceiling of single-precision float in $f1
`clo $t1,$t2`	Count number of leading ones : Set $t1 to the count of leading one bits in $t2 starting at most significant bit position
`clz $t1,$t2`	Count number of leading zeroes : Set $t1 to the count of leading zero bits in $t2 starting at most significant bit position
`cvt.d.s $f2,$f1`	Convert from single precision to double precision : Set $f2 to double precision equivalent of single precision value in $f1
`cvt.d.w $f2,$f1`	Convert from word to double precision : Set $f2 to double precision equivalent of 32-bit integer value in $f1
`cvt.s.d $f1,$f2`	Convert from double precision to single precision : Set $f1 to single precision equivalent of double precision value in $f2
`cvt.s.w $f0,$f1`	Convert from word to single precision : Set $f0 to single precision equivalent of 32-bit integer value in $f2
`cvt.w.d $f1,$f2`	Convert from double precision to word : Set $f1 to 32-bit integer equivalent of double precision value in $f2
`cvt.w.s $f0,$f1`	Convert from single precision to word : Set $f0 to 32-bit integer equivalent of single precision value in $f1
`div $t1,$t2`	Division with overflow : Divide $t1 by $t2 then set LO to quotient and HI to remainder (use mfhi to access HI, mflo to access LO)
`div.d $f2,$f4,$f6`	Floating point division double precision : Set $f2 to double-precision floating point value of $f4 divided by $f6
`div.s $f0,$f1,$f3`	Floating point division single precision : Set $f0 to single-precision floating point value of $f1 divided by $f3

`divu $t1,$t2`	Division unsigned without overflow : Divide unsigned $t1 by $t2 then set LO to quotient and HI to remainder (use mfhi to access HI, mflo to access LO)
`eret`	Exception return : Set Program Counter to Coprocessor 0 EPC register value, set Coprocessor Status register bit 1 (exception level) to zero
`floor.w.d $f1,$f2`	Floor double precision to word : Set $f1 to 32-bit integer floor of double-precision float in $f2
`floor.w.s $f0,$f1`	Floor single precision to word : Set $f0 to 32-bit integer floor of single-precision float in $f1
`j target`	Jump unconditionally : Jump to statement at target address
`jal target`	Jump and link : Set $ra to Program Counter (return address) then jump to statement at target address
`jalr $t1`	Jump and link register : Set $ra to Program Counter (return address) then jump to statement whose address is in $t1
`jalr $t1,$t2`	Jump and link register : Set $t1 to Program Counter (return address) then jump to statement whose address is in $t2
`jr $t1`	Jump register unconditionally : Jump to statement whose address is in $t1
`lb $t1,-100($t2)`	Load byte : Set $t1 to sign-extended 8-bit value from effective memory byte address
`lbu $t1,-100($t2)`	Load byte unsigned : Set $t1 to zero-extended 8-bit value from effective memory byte address
`ldc1 $f2,-100($t2)`	Load double word Coprocessor 1 (FPU)) : Set $f2 to 64-bit value from effective memory doubleword address
`lh $t1,-100($t2)`	Load halfword : Set $t1 to sign-extended 16-bit value from effective memory halfword address
`lhu $t1,-100($t2)`	Load halfword unsigned : Set $t1 to zero-extended 16-bit value from effective memory halfword address
`ll $t1,-100($t2)`	Load linked : Paired with Store Conditional (sc) to perform atomic read-modify-write. Treated as equivalent to Load Word (lw) because MARS does not simulate multiple processors.
`lui $t1,100`	Load upper immediate : Set high-order 16 bits of $t1 to 16-bit immediate and low-order 16 bits to 0
`lw $t1,-100($t2)`	Load word : Set $t1 to contents of effective memory word address
`lwc1 $f1,-100($t2)`	Load word into Coprocessor 1 (FPU) : Set $f1 to 32-bit value from effective memory word address
`lwl $t1,-100($t2)`	Load word left : Load from 1 to 4 bytes left-justified into $t1, starting with effective memory byte address and continuing through the low-order byte of its word
`lwr $t1,-100($t2)`	Load word right : Load from 1 to 4 bytes right-justified into $t1, starting with effective memory byte address and continuing through the high-order byte of its word

`madd $t1,$t2`	Multiply add : Multiply $t1 by $t2 then increment HI by high-order 32 bits of product, increment LO by low-order 32 bits of product (use mfhi to access HI, mflo to access LO)
`maddu $t1,$t2`	Multiply add unsigned : Multiply $t1 by $t2 then increment HI by high-order 32 bits of product, increment LO by low-order 32 bits of product, unsigned (use mfhi to access HI, mflo to access LO)
`mfc0 $t1,$8`	Move from Coprocessor 0 : Set $t1 to the value stored in Coprocessor 0 register $8
`mfc1 $t1,$f1`	Move from Coprocessor 1 (FPU) : Set $t1 to value in Coprocessor 1 register $f1
`mfhi $t1`	Move from HI register : Set $t1 to contents of HI (see multiply and divide operations)
`mflo $t1`	Move from LO register : Set $t1 to contents of LO (see multiply and divide operations)
`mov.d $f2,$f4`	Move floating point double precision : Set double precision $f2 to double precision value in $f4
`mov.s $f0,$f1`	Move floating point single precision : Set single precision $f0 to single precision value in $f1
`movf $t1,$t2`	Move if FP condition flag 0 false : Set $t1 to $t2 if FPU (Coprocessor 1) condition flag 0 is false (zero)
`movf $t1,$t2,1`	Move if specified FP condition flag false : Set $t1 to $t2 if FPU (Coprocessor 1) condition flag specified by the immediate is false (zero)
`movf.d $f2,$f4`	Move floating point double precision : If condition flag 0 false, set double precision $f2 to double precision value in $f4
`movf.d $f2,$f4,1`	Move floating point double precision : If condition flag specified by immediate is false, set double precision $f2 to double precision value in $f4
`movf.s $f0,$f1`	Move floating point single precision : If condition flag 0 is false, set single precision $f0 to single precision value in $f1
`movf.s $f0,$f1,1`	Move floating point single precision : If condition flag specified by immediate is false, set single precision $f0 to single precision value in $f1e
`movn $t1,$t2,$t3`	Move conditional not zero : Set $t1 to $t2 if $t3 is not zero
`movn.d $f2,$f4,$t3`	Move floating point double precision : If $t3 is not zero, set double precision $f2 to double precision value in $f4
`movn.s $f0,$f1,$t3`	Move floating point single precision : If $t3 is not zero, set single precision $f0 to single precision value in $f1
`movt $t1,$t2`	Move if FP condition flag 0 true : Set $t1 to $t2 if FPU (Coprocessor 1) condition flag 0 is true (one)
`movt $t1,$t2,1`	Move if specfied FP condition flag true : Set $t1 to $t2 if FPU (Coprocessor 1) condition flag specified by the immediate is true (one)
`movt.d $f2,$f4`	Move floating point double precision : If condition flag 0 true, set double precision $f2 to double precision value in $f4

movt.d $f2,$f4,1	Move floating point double precision : If condition flag specified by immediate is true, set double precision $f2 to double precision value in $f4e
movt.s $f0,$f1	Move floating point single precision : If condition flag 0 is true, set single precision $f0 to single precision value in $f1e
movt.s $f0,$f1,1	Move floating point single precision : If condition flag specified by immediate is true, set single precision $f0 to single precision value in $f1e
movz $t1,$t2,$t3	Move conditional zero : Set $t1 to $t2 if $t3 is zero
movz.d $f2,$f4,$t3	Move floating point double precision : If $t3 is zero, set double precision $f2 to double precision value in $f4
movz.s $f0,$f1,$t3	Move floating point single precision : If $t3 is zero, set single precision $f0 to single precision value in $f1
msub $t1,$t2	Multiply subtract : Multiply $t1 by $t2 then decrement HI by high-order 32 bits of product, decrement LO by low-order 32 bits of product (use mfhi to access HI, mflo to access LO)
msubu $t1,$t2	Multiply subtract unsigned : Multiply $t1 by $t2 then decrement HI by high-order 32 bits of product, decrement LO by low-order 32 bits of product, unsigned (use mfhi to access HI, mflo to access LO)
mtc0 $t1,$8	Move to Coprocessor 0 : Set Coprocessor 0 register $8 to value stored in $t1
mtc1 $t1,$f1	Move to Coprocessor 1 (FPU) : Set Coprocessor 1 register $f1 to value in $t1
mthi $t1	Move to HI register : Set HI to contents of $t1 (see multiply and divide operations)
mtlo $t1	Move to LO register : Set LO to contents of $t1 (see multiply and divide operations)
mul $t1,$t2,$t3	Multiplication without overflow : Set HI to high-order 32 bits, LO and $t1 to low-order 32 bits of the product of $t1 and $t2 (use mfhi to access HI, mflo to access LO)
mul.d $f2,$f4,$f6	Floating point multiplication double precision : Set $f2 to double-precision floating point value of $f4 times $f6
mul.s $f0,$f1,$f3	Floating point multiplication single precision : Set $f0 to single-precision floating point value of $f1 times $f3
mult $t1,$t2	Multiplication : Set hi to high-order 32 bits, lo to low-order 32 bits of the product of $t1 and $t2 (use mfhi to access hi, mflo to access lo)
multu $t1,$t2	Multiplication unsigned : Set HI to high-order 32 bits, LO to low-order 32 bits of the product of unsigned $t1 and $t2 (use mfhi to access HI, mflo to access LO)
neg.d $f2,$f4	Floating point negate double precision : Set double precision $f2 to negation of double precision value in $f4
neg.s $f0,$f1	Floating point negate single precision : Set single precision $f0 to negation of single precision value in $f1
nop	Null operation : machine code is all zeroes

`nor $t1,$t2,$t3`	Bitwise NOR : Set $t1 to bitwise NOR of $t2 and $t3
`or $t1,$t2,$t3`	Bitwise OR : Set $t1 to bitwise OR of $t2 and $t3
`ori $t1,$t2,100`	Bitwise OR immediate : Set $t1 to bitwise OR of $t2 and zero-extended 16-bit immediate
`round.w.d $f1,$f2`	Round double precision to word : Set $f1 to 32-bit integer round of double-precision float in $f2
`round.w.s $f0,$f1`	Round single precision to word : Set $f0 to 32-bit integer round of single-precision float in $f1
`sb $t1,-100($t2)`	Store byte : Store the low-order 8 bits of $t1 into the effective memory byte address
`sc $t1,-100($t2)`	Store conditional : Paired with Load Linked (ll) to perform atomic read-modify-write. Stores $t1 value into effective address, then sets $t1 to 1 for success. Always succeeds because MARS does not simulate multiple processors.
`sdc1 $f2,-100($t2)`	Store double word from Coprocessor 1 (FPU)) : Store 64 bit value in $f2 to effective memory doubleword address
`sh $t1,-100($t2)`	Store halfword : Store the low-order 16 bits of $t1 into the effective memory halfword address
`sll $t1,$t2,10`	Shift left logical : Set $t1 to result of shifting $t2 left by number of bits specified by immediate
`sllv $t1,$t2,$t3`	Shift left logical variable : Set $t1 to result of shifting $t2 left by number of bits specified by value in low-order 5 bits of $t3
`slt $t1,$t2,$t3`	Set less than : If $t2 is less than $t3, then set $t1 to 1 else set $t1 to 0
`slti $t1,$t2,-100`	Set less than immediate : If $t2 is less than sign-extended 16-bit immediate, then set $t1 to 1 else set $t1 to 0
`sltiu $t1,$t2,-100`	Set less than immediate unsigned : If $t2 is less than sign-extended 16-bit immediate using unsigned comparison, then set $t1 to 1 else set $t1 to 0
`sltu $t1,$t2,$t3`	Set less than unsigned : If $t2 is less than $t3 using unsigned comparison, then set $t1 to 1 else set $t1 to 0
`sqrt.d $f2,$f4`	Square root double precision : Set $f2 to double-precision floating point square root of $f4
`sqrt.s $f0,$f1`	Square root single precision : Set $f0 to single-precision floating point square root of $f1
`sra $t1,$t2,10`	Shift right arithmetic : Set $t1 to result of sign-extended shifting $t2 right by number of bits specified by immediate
`srav $t1,$t2,$t3`	Shift right arithmetic variable : Set $t1 to result of sign-extended shifting $t2 right by number of bits specified by value in low-order 5 bits of $t3
`srl $t1,$t2,10`	Shift right logical : Set $t1 to result of shifting $t2 right by number of bits specified by immediate
`srlv $t1,$t2,$t3`	Shift right logical variable : Set $t1 to result of shifting $t2 right by number of bits specified by value in low-order 5 bits of $t3
`sub $t1,$t2,$t3`	Subtraction with overflow : set $t1 to ($t2 minus $t3)

`sub.d $f2,$f4,$f6`	Floating point subtraction double precision : Set $f2 to double-precision floating point value of $f4 minus $f6
`sub.s $f0,$f1,$f3`	Floating point subtraction single precision : Set $f0 to single-precision floating point value of $f1 minus $f3
`subu $t1,$t2,$t3`	Subtraction unsigned without overflow : set $t1 to ($t2 minus $t3), no overflow
`sw $t1,-100($t2)`	Store word : Store contents of $t1 into effective memory word address
`swc1 $f1,-100($t2)`	Store word from Coprocessor 1 (FPU) : Store 32 bit value in $f1 to effective memory word address
`swl $t1,-100($t2)`	Store word left : Store high-order 1 to 4 bytes of $t1 into memory, starting with effective byte address and continuing through the low-order byte of its word
`swr $t1,-100($t2)`	Store word right : Store low-order 1 to 4 bytes of $t1 into memory, starting with high-order byte of word containing effective byte address and continuing through that byte address
`syscall`	Issue a system call : Execute the system call specified by value in $v0
`teq $t1,$t2`	Trap if equal : Trap if $t1 is equal to $t2
`teqi $t1,-100`	Trap if equal to immediate : Trap if $t1 is equal to sign-extended 16 bit immediate
`tge $t1,$t2`	Trap if greater or equal : Trap if $t1 is greater than or equal to $t2
`tgei $t1,-100`	Trap if greater than or equal to immediate : Trap if $t1 greater than or equal to sign-extended 16 bit immediate
`tgeiu $t1,-100`	Trap if greater or equal to immediate unsigned : Trap if $t1 greater than or equal to sign-extended 16 bit immediate, unsigned comparison
`tgeu $t1,$t2`	Trap if greater or equal unsigned : Trap if $t1 is greater than or equal to $t2 using unsigned comparison
`tlt $t1,$t2`	Trap if less than: Trap if $t1 less than $t2
`tlti $t1,-100`	Trap if less than immediate : Trap if $t1 less than sign-extended 16-bit immediate
`tltiu $t1,-100`	Trap if less than immediate unsigned : Trap if $t1 less than sign-extended 16-bit immediate, unsigned comparison
`tltu $t1,$t2`	Trap if less than unsigned : Trap if $t1 less than $t2, unsigned comparison
`tne $t1,$t2`	Trap if not equal : Trap if $t1 is not equal to $t2
`tnei $t1,-100`	Trap if not equal to immediate : Trap if $t1 is not equal to sign-extended 16 bit immediate
`trunc.w.d $f1,$f2`	Truncate double precision to word : Set $f1 to 32-bit integer truncation of double-precision float in $f2
`trunc.w.s $f0,$f1`	Truncate single precision to word : Set $f0 to 32-bit integer truncation of single-precision float in $f1
`xor $t1,$t2,$t3`	Bitwise XOR (exclusive OR) : Set $t1 to bitwise XOR of $t2 and $t3
`xori $t1,$t2,100`	Bitwise XOR immediate : Set $t1 to bitwise XOR of $t2 and zero-extended 16-bit immediate

Directives

The following list provides a description of all directives available in MARS.

Directives	
.align	Align next data item on specified byte boundary (0=byte, 1=half, 2=word, 3=double)
.ascii	Store the string in the Data segment but do not add null terminator
.asciiz	Store the string in the Data segment and add null terminator
.byte	Store the listed value(s) as 8 bit bytes
.data	Subsequent items stored in Data segment at next available address
.double	Store the listed value(s) as double precision floating point
.extern	Declare the listed label and byte length to be a global data field
.float	Store the listed value(s) as single precision floating point
.globl	Declare the listed label(s) as global to enable referencing from other files
.half	Store the listed value(s) as 16 bit halfwords on halfword boundary
.kdata	Subsequent items stored in Kernel Data segment at next available address
.ktext	Subsequent items (instructions) stored in Kernel Text segment at next available address
.set	Set assembler variables. Currently ignored but included for SPIM compatibility
.space	Reserve the next specified number of bytes in Data segment
.text	Subsequent items (instructions) stored in Text segment at next available address
.word	Store the listed value(s) as 32 bit words on word boundary

Syscall Functions

A number of system services, mainly for input and output, are available for use by your MIPS program. They are described in the table below.

How to use SYSCALL system services

Step 1. Load the service number in register $v0.
Step 2. Load argument values, if any, in $a0, $a1, $a2, or $f12 as specified.
Step 3. Issue the SYSCALL instruction.
Step 4. Retrieve return values, if any, from result registers as specified.

Example: display the value stored in $t0 on the console

```
li  $v0, 1                # service 1 is print integer
add $a0, $t0, $zero       # load desired value into argument register
                          # $a0, using pseudo-op
syscall
```

Service	Code in $v0	Arguments	Result
print integer	1	$a0 = integer to print	
print float	2	$f12 = float to print	
print double	3	$f12 = double to print	
print string	4	$a0 = address of null-terminated string to print	
read integer	5		$v0 contains integer read
read float	6		$f0 contains float read
read double	7		$f0 contains double read
read string	8	$a0 = address of input buffer $a1 = maximum number of characters to read	*See note below table*
sbrk (allocate heap memory)	9	$a0 = number of bytes to allocate	$v0 contains address of allocated memory

exit (terminate execution)	10		
print character	11	$a0 = character to print	*See note below table*
read character	12		$v0 contains character read
open file	13	$a0 = address of null-terminated string containing filename $a1 = flags $a2 = mode	$v0 contains file descriptor (negative if error). *See note below table*
read from file	14	$a0 = file descriptor $a1 = address of input buffer $a2 = maximum number of characters to read	$v0 contains number of characters read (0 if end-of-file, negative if error). *See note below table*
write to file	15	$a0 = file descriptor $a1 = address of output buffer $a2 = number of characters to write	$v0 contains number of characters written (negative if error). *See note below table*
close file	16	$a0 = file descriptor	
exit2 (terminate with value)	17	$a0 = termination result	*See note below table*

Services 1 through 17 are compatible with the SPIM simulator, other than Open File (13) as described in the Notes below the table. Services 30 and higher are exclusive to MARS.

time (system time)	30		$a0 = low order 32 bits of system time $a1 = high order 32 bits of system time. *See note below table*
MIDI out	31	$a0 = pitch (0-127) $a1 = duration in milliseconds $a2 = instrument (0-127) $a3 = volume (0-127)	Generate tone and return immediately. *See MARS help*
sleep	32	$a0 = the length of time to sleep in milliseconds.	Causes the MARS Java thread to sleep for (at least) the specified number of milliseconds. This timing will not be precise, as the Java implementation will add some overhead.

MIDI out synchronous	33	$a0 = pitch (0-127) $a1 = duration in milliseconds $a2 = instrument (0-127) $a3 = volume (0-127)	Generate tone and return upon tone completion. *See MARS help*
print integer in hexadecimal	34	$a0 = integer to print	
print integer in binary	35	$a0 = integer to print	
print integer as unsigned	36	$a0 = integer to print	
(not used)	37-39		
set seed	40	$a0 = i.d. of pseudorandom number generator (any int). $a1 = seed for corresponding pseudorandom number generator.	No values are returned. Sets the seed of the corresponding underlying Java pseudorandom number generator (java.util.Random). *See note below table*
random int	41	$a0 = i.d. of pseudorandom number generator (any int).	$a0 contains the next pseudorandom, uniformly distributed int value from this random number generator's sequence. *See note below table*
random int range	42	$a0 = i.d. of pseudorandom number generator (any int). $a1 = upper bound of range of returned values.	$a0 contains pseudorandom, uniformly distributed int value in the range 0 = [int] [upper bound], drawn from this random number generator's sequence. *See note below table*
random float	43	$a0 = i.d. of pseudorandom number generator (any int).	$f0 contains the next pseudorandom, uniformly distributed float value in the range 0.0 = f 1.0 from this random number generator's sequence. *See note below table*
random double	44	$a0 = i.d. of pseudorandom number generator (any int).	$f0 contains the next pseudorandom, uniformly distributed double value in the range 0.0 = f 1.0 from this random number generator's sequence. *See note below table*
(not used)	45-49		

ConfirmDialog	50	$a0 = address of null-terminated string that is the message to user	$a0 contains value of user-chosen option 0: Yes 1: No 2: Cancel
InputDialogInt	51	$a0 = address of null-terminated string that is the message to user	$a0 contains int read $a1 contains status value 0: OK status -1: input data cannot be correctly parsed -2: Cancel was chosen -3: OK was chosen but no data had been input into field
InputDialogFloat	52	$a0 = address of null-terminated string that is the message to user	$f0 contains float read $a1 contains status value 0: OK status -1: input data cannot be correctly parsed -2: Cancel was chosen -3: OK was chosen but no data had been input into field
InputDialogDouble	53	$a0 = address of null-terminated string that is the message to user	$f0 contains double read $a1 contains status value 0: OK status -1: input data cannot be correctly parsed -2: Cancel was chosen -3: OK was chosen but no data had been input into field
InputDialogString	54	$a0 = address of null-terminated string that is the message to user $a1 = address of input buffer $a2 = maximum number of characters to read	*See Service 8 note below table* $a1 contains status value 0: OK status. Buffer contains the input string. -2: Cancel was chosen. No change to buffer. -3: OK was chosen but no data had been input into field. No change to buffer. -4: length of the input string exceeded the specified maximum. Buffer contains the maximum allowable input string plus a terminating null.
MessageDialog	55	$a0 = address of null-terminated string that is the message to user $a1 = the type of message to be displayed: 0: error message, indicated by Error icon	N/A

		1: information message, indicated by Information icon 2: warning message, indicated by Warning icon 3: question message, indicated by Question icon other: plain message (no icon displayed)	
MessageDialogInt	56	$a0 = address of null-terminated string that is an information-type message to user $a1 = int value to display in string form after the first string	N/A
MessageDialogFloat	57	$a0 = address of null-terminated string that is an information-type message to user $f12 = float value to display in string form after the first string	N/A
MessageDialogDouble	58	$a0 = address of null-terminated string that is an information-type message to user $f12 = double value to display in string form after the first string	N/A
MessageDialogString	59	$a0 = address of null-terminated string that is an information-type message to user $a1 = address of null-terminated string to display after the first string	N/A

NOTES: Services numbered 30 and higher are not provided by SPIM
Service 8 - Follows semantics of UNIX 'fgets'. For specified length n, string can be no longer than n-1. If it is less than that, adds newline to end. In either case, then pads with null byte.
Service 11 - Prints ASCII character corresponding to contents of low-order byte.
Service 13 - MARS implements three flag values: 0 for read-only, 1 for write-only with create, and 9 for write-only with create and append. It ignores mode. The returned file descriptor will be negative if the operation failed. The underlying file I/O implementation uses java.io.FileInputStream.read() to read and java.io.FileOutputStream.write() to write. MARS maintains file descriptors internally and allocates them starting with 0.
Services 13,14,15 - In MARS 3.7, the result register was changed to $v0 for SPIM compatibility. It was previously $a0 as erroneously printed in Appendix B of *Computer Organization and Design,*.
Service 17 - If the MIPS program is run under control of the MARS graphical interface (GUI), the exit code in $a0 is ignored.
Service 30 - System time comes from java.util.Date.getTime() as milliseconds since 1 January 1970.
Services 40-44 use underlying Java pseudorandom number generators provided by the java.util.Random class. Each stream (identified by $a0 contents) is modeled by a different Random object. There are no default seed values, so use the Set Seed service (40) if replicated random sequences are desired.

Example of File I/O

The sample MIPS program below will open a new file for writing, write text to it from a memory buffer, then close it. The file will be created in the directory in which MARS was run.

```
# Sample MIPS program that writes to a new file.
#   by Kenneth Vollmar and Pete Sanderson
        .data
fout:   .asciiz "testout.txt"      # filename for output
buffer: .asciiz "The quick brown fox jumps over the lazy dog."
        .text
###############################################################
# Open (for writing) a file that does not exist
li   $v0, 13       # system call for open file
la   $a0, fout     # output file name
li   $a1, 1        # Open for writing (flags are 0: read, 1: write)
li   $a2, 0        # mode is ignored
syscall            # open a file (file descriptor returned in $v0)
move $s6, $v0      # save the file descriptor
###############################################################
# Write to file just opened
li   $v0, 15       # system call for write to file
move $a0, $s6      # file descriptor
la   $a1, buffer   # address of buffer from which to write
li   $a2, 44       # hardcoded buffer length
syscall            # write to file
###############################################################
# Close the file
li   $v0, 16       # system call for close file
move $a0, $s6      # file descriptor to close
syscall            # close file
```