Fonte: http://www.ric.edu/faculty/emcdowell/cs312/mars.txt

INTRODUCTION TO MARS

by Edward McDowell

Mars (MIPS Assembler and Runtime Simulator) is a simple environment for developing MIPS assembly language programs on any platform with a Java virtual machine. Mars was created by Pete Sanderson and Kenneth Vollmar at Missouri State University for teaching elementary assembly language in computer science courses. Mars is free and may be downloaded from http://www.cs.missouristate.edu/~vollmar/MARS/ .

The current version of Mars requires that the Sun Java Development Kit JDK 1.4.2 or higher be installed on your computer. This is available at the Java web page http://java.sun.com .

ENTERING MARS

No installation is required to use Mars, provided that Java is correctly installed on your computer. Download Mars.jar from the web page named above. Double click the icon for Mars.jar and the Mars environment will open.

CREATING A NEW PROGRAM

Select "File:New" from the Mars menu to open a blank editor window. Enter your program. The example below shows the format of a Mars program. The usual cutting and pasting services are available through the "Edit" menu. Select "File:Save As" to save your program to disk. The ".asm" extension is recommended. Once the file has been saved for the first time, you may select "File:Save" to save changes without having to specify the file name.

```
# SAMPLE.ASM: This program prints a greeting.

          .data
#  Define a greeting message.
Message:  .asciiz   "Hello World!\n"

          .text
#  Print the greeting message.
          li        $v0, 4
          la        $a0, Message
          syscall
#  Return to the operating system.
          li        $v0, 10
          syscall
```

OPENING AN EXISTING PROGRAM

To open an existing program, select "File:Open" from the Mars menu. Enter the name of the program file and click the Open button.

ASSEMBLING THE PROGRAM

Once the program has been created and saved to disk it may be assembled (translated into MIPS machine language). Select "Run:Assemble" from the Mars menu. If there are no errors the Execute pane will appear, showing memory and register contents prior to execution. Click the "Edit" tab if you wish to return to the Edit pane.

RUNNING THE PROGRAM

Select "Run:Go" to execute the program. Select "Run:Step" to execute the next machine instruction. The new register and memory contents are displayed in the Execute pane. Select "Run:Stop" to break a looping program. Select "Run:Reset" to reset memory and register contents to their initial state after the program was assembled. Click the "Edit" tab to return to the Edit pane.

CLOSING A PROGRAM

Select "File:Close" to close the current program. Always close the program before removing the program disk or exiting Mars. Exit Mars with "File:Exit".

GETTING THE OUTPUT FROM A RUN

Run your program with "Run:Go" and then select the output with the mouse. Press CTRL/C to copy the output to the clipboard. Click the "Edit" tab to return to the editor window. Move the cursor to the bottom of the editor window and then press CTRL/V to paste the output at the cursor position. Select "File:Print" to print the source file with the program output.

RUNNING MARS FROM THE COMMAND PROMPT

Mars is usually run interactively, by double clicking the Mars.jar icon. It is possible to run Mars at the command prompt to assemble and run a single program. Copy Mars.jar into the folder containing the program you wish to run. The following command should run the program PROGNAME.ASM:

```
java -jar Mars.jar progname.asm
```

MIPS INTEGER REGISTERS (32-BITS)

$zero	$0	constant 0
$at	$1	reserved for assembler
$v0	$2	return value
$v1	$3	return value
$a0	$4	argument
$a1	$5	argument
$a2	$6	argument
$a3	$7	argument
$t0	$8	temporary
$t1	$9	temporary
$t2	$10	temporary
$t3	$11	temporary
$t4	$12	temporary
$t5	$13	temporary
$t6	$14	temporary
$t7	$15	temporary
$s0	$16	saved temporary
$s1	$17	saved temporary
$s2	$18	saved temporary
$s3	$19	saved temporary
$s4	$20	saved temporary
$s5	$21	saved temporary
$s6	$22	saved temporary
$s7	$23	saved temporary
$t8	$24	temporary
$t9	$25	temporary
$k0	$26	reserved for OS
$k1	$27	reserved for OS
$gp	$28	global pointer
$sp	$29	stack pointer
$fp	$30	frame pointer
$ra	$31	return address

MARS SYSTEM CALLS (used with syscall)

Function	$v0	Arguments	Results
Print Integer	1	$a0=value	
Print Single	2	$f12=value	
Print Double	3	$f12=value	
Print String	4	$a0=addr of string	
Read Integer	5		$v0=value
Read Single	6		$f0=value
Read Double	7		$f0=value
Read String	8	$a0=addr of buffer $a1=size of buffer	
Get Memory	9	$a0=amount (bytes)	$v0=address
Exit	10		

MARS ASSEMBLER DIRECTIVES

.align	n	Align to multiple of 2**n.
.ascii	"string"	Store string.
.asciiz	"string"	Store null-terminated string.
.byte	val,val,...	Store byte values.
.data		Store in data segment.
.double	val,val,...	Store double-precision values
.float	val,val,...	Store single-precision values
.globl	sym	Declare global symbol.
.half	val,val,...	Store halfword values.
.space	n	Allocate n bytes of space.
.text		Store in code segment.
.word	val,val,...	Store word values.
.word	val:n	Store val in n words.

MIPS REGISTER INSTRUCTIONS

Register instructions take all operands from CPU registers and do not access memory. These are very efficient.

```
Register instruction format: 000000 sssss ttttt ddddd bbbbb ffffff
    sssss     Rs
    ttttt     Rt
    ddddd     Rd
    bbbbb     bits to shift (00000 except for sll, srl, sra)
    ffffff    function
```

Unused fields are encoded as zero.

```
    add       Rd, Rs, Rt    # func=100000  add
    addu      Rd, Rs, Rt    # func=100001  add unsigned
    sub       Rd, Rs, Rt    # func=100010  subtract
    subu      Rd, Rs, Rt    # func=100011  subtract unsigned
    mult      Rs, Rt        # func=011000  multiply
    multu     Rs, Rt        # func=011001  multiply unsigned
    div       Rs, Rt        # func=011010  divide
    divu      Rs, Rt        # func=011011  divide unsigned

    mfhi      Rd            # func=010000  move from hi
    mflo      Rd            # func=010010  move from lo
    mthi      Rs            # func=010001  move to hi
    mtlo      Rs            # func=010011  move to lo

    and       Rd, Rs, Rt    # func=100100  bitwise and
    or        Rd, Rs, Rt    # func=100101  bitwise or
    xor       Rd, Rs, Rt    # func=100110  bitwise exclusive or
    nor       Rd, Rs, Rt    # func=100111  bitwise nor

    sll       Rd, Rt, bits  # func=000000  shift left logical
    srl       Rd, Rt, bits  # func=000010  shift right logical
    sra       Rd, Rt, bits  # func=000011  shift right arithmetic
    sllv      Rd, Rt, Rs    # func=000100  shift left logical
    srlv      Rd, Rt, Rs    # func=000110  shift right logical
    srav      Rd, Rt, Rs    # func=000111  shift right arithmetic

    nop                     # func=000000  no operation
    syscall                 # func=001100  system call
```

MIPS IMMEDIATE INSTRUCTIONS

Immediate instructions include a 16-bit immediate operand, which is either a constant value or an offset for memory access.

```
Immediate Instruction format:  cccccc sssss ttttt iiiiiiiiiiiiiiii
    cccccc                   operation code
    sssss                    Rs
    ttttt                    Rt
    iiiiiiiiiiiiiiii         16-bit constant or offset
```

Unused fields are encoded as zero.

The following instructions use the immediate operand to represent a constant value.

```
    addi     Rt, Rs, val   # opcode=001000  add immediate
    addiu    Rt, Rs, val   # opcode=001001  add immediate unsigned
    andi     Rt, Rs, val   # opcode=001100  bitwise and immediate
    ori      Rt, Rs, val   # opcode=001101  bitwise or immediate
    xori     Rt, Rs, val   # opcode=001110  bitwise xor immediate
    lui      Rt, val       # opcode=001111  load upper half word
```

Memory operands are coded in the form offset(Rs) where the 16-bit offset is added to the content of Rs to compute the memory address. The memory operand may be a statement label. In this case the assembler will generate two instructions to access the operand, as in the following example:

```
    lw  Rt, label
       lui       $1, address.upper
       lw        Rt, address.lower($1)
```

The following instructions use the immediate operand to represent a memory address.

```
    lb       Rt, memory   # opcode=100000  load byte
    lh       Rt, memory   # opcode=100001  load half word
    lw       Rt, memory   # opcode=100011  load word
    lbu      Rt, memory   # opcode=100100  load byte unsigned
    lhu      Rt, memory   # opcode=100101  load half unsigned
    sb       Rt, memory   # opcode=101000  store byte
    sh       Rt, memory   # opcode=101001  store half word
    sw       Rt, memory   # opcode=101011  store word
```

MIPS JUMP INSTRUCTIONS

There are two jump instructions having a special format. The actual jump address, pppp tttttttttttttttttttttttttt 00, is formed by joining the leftmost 4 bits pppp of the current program counter, the 26-bit jump target tttttttttttttttttttttttttt, and 2 zero bits.

```
Jump instruction format:  cccccc tttttttttttttttttttttttttt
    cccccc                         operation code
    tttttttttttttttttttttttttt     26-bit jump target
```

The jump instructions are:

```
    j        label         # opcode=000010  jump
    jal      label         # opcode=000011  jump and link
```

Register instructions used with branching:

```
    jr       Rs            # func=001000  jump register
    jalr     Rd, Rs        # func=001001  jump and link register
    slt      Rd, Rs, Rt    # func=101010  set on less than
    sltu     Rd, Rs, Rt    # func=101011  set on less than unsigned
```

Immediate instructions used with branching:

The immediate operand is an offset which is shifted left by two bits and then added to the current program counter to get the jump address.

```
    beq      Rs, Rt, label  # opcode=000100  branch on equal
    bne      Rs, Rt, label  # opcode=000101  branch on not equal
    bgtz     Rs, label      # opcode=000111  branch on positive
    blez     Rs, label      # opcode=000110  branch on not positive
    bltz     Rs, label      # opcode=000001  Rt=00000 branch on < 0
    bgez     Rs, label      # opcode=000001  Rt=00001 branch on >= 0
```

The assembler provides macros which combine the instructions described above to perform various conditional jumps:

```
    blt      Rs, Rt, label  # branch on less than
    ble      Rs, Rt, label  # branch on less than or equal
    bgt      Rs, Rt, label  # branch on greater than
    bge      Rs, Rt, label  # branch on greater than or equal
    bltu     Rs, Rt, label  # branch on less than unsigned
    bleu     Rs, Rt, label  # branch on less than or equal unsigned
    bgtu     Rs, Rt, label  # branch on greater than unsigned
    bgeu     Rs, Rt, label  # branch on greater than or equal unsigned
    beqz     Rs, label      # branch on zero
    bnez     Rs, label      # branch on not zero
```

MIPS MACRO INSTRUCTIONS

Macros are pseudo-instructions that are provided by the assembler for common operations that are not part of the MIPS instruction set. These are expanded automatically to a short sequence of instructions that accomplish the desired task.

Some important MIPS macros:

```
    move     Rd, Rs                        # move
       addu     Rd, $0, Rs

    neg      Rd, Rs                        # negate
       sub      Rd, $0, Rs

    not      Rd, Rs                        # bitwise not
       nor      Rd, Rs, $0

    abs      Rd, Rs                        # absolute value
       addu     Rd, $0, Rs
       bgez     Rs, skip
       sub      Rd, $0, Rs
    skip:

    li       Rt, value                     # load immediate
       lui      $1, constant.upper
       ori      Rt, $1, constant.lower

    la       Rt, label                     # load address
       lui      $1, address.upper
       ori      Rt, $1, address.lower

    lw       Rt, label                     # load word at label
       lui      $1, address.upper
       lw       Rt, address.lower($1)

    sw       Rt, label                     # store word at label
       lui      $1, address.upper
       sw       Rt, address.lower($1)

    blt      Rs, Rt, label      # branch on less than
       slt      $1, Rs, Rt
       bne      $1, $0, label

    ble      Rs, Rt, label      # branch on less than or equal
       slt      $1, Rt, Rs
       beq      $1, $0, label

    bgt      Rs, Rt, label      # branch on greater than
       slt      $1, Rt, Rs
       bne      $1, $0, label

    bge      Rs, Rt, label      # branch on greater than or equal
       slt      $1, Rs, Rt
       beq      $1, $0, label
```

MIPS FLOATING-POINT REGISTERS (32-BITS)

Each register may hold a single-precision value or half of a double-precision value. Double-precision values are stored in even-odd register pairs, addressed by the even register. Floating-point numbers are represented in IEEE format.

```
$f0 - $f3    return values
$f4 - $f11   temporary values
$f12 - $f15  arguments
$f16 - $f19  temporary values
$f20 - $f31  saved temporary values
```

MIPS FLOATING-POINT INSTRUCTIONS

```
Memory instruction format:  cccccc sssss ttttt iiiiiiiiiiiiiiii
    cccccc              operation code
    sssss               Rs
    ttttt               Ft
    iiiiiiiiiiiiiiii    16-bit memory offset
```

The Ft operand specifies a floating-point register. The memory operand has the form offset(Rs). If a label is specified the assembler will generate two instructions to access the operand.

```
    lwc1    Ft, memory   # opcode = 110001  load single (also l.s)
    ldc1    Ft, memory   # opcode = 110101  load double (also l.d)
    swc1    Ft, memory   # opcode = 111001  store single (also s.s)
    sdc1    Ft, memory   # opcode = 111101  store double (also s.d)
```

```
Register instruction format:  010001 ppppp ttttt sssss ddddd ffffff
    ppppp    precision:  10000 for single; 10001 for double
    ttttt    Ft
    sssss    Fs
    ddddd    Fd
    ffffff   function
```

Missing fields are zero. This format is different from the register format for integer instructions.

```
    mov.s   Fd, Fs    # func=000110  move single
    mov.d   Fd, Fs    # func=000110  move double
    mtc1    Rt, Fs    # func=000000 prec=00100 Ft=Rt  move Rt to Fs
    mfc1    Rt, Fs    # func=000000 prec=00000 Ft=Rt  move Fs to Rt

    abs.s   Fd, Fs       # func=000101  absolute value single
    abs.d   Fd, Fs       # func=000101  absolute value double
    neg.s   Fd, Fs       # func=000111  negate single
    neg.d   Fd, Fs       # func=000111  negate double
```

```
add.s   Fd, Fs, Ft        # func=000000  add single
add.d   Fd, Fs, Ft        # func=000000  add double
sub.s   Fd, Fs, Ft        # func=000001  subtract single
sub.d   Fd, Fs, Ft        # func=000001  subtract double
mul.s   Fd, Fs, Ft        # func=000010  multiply single
mul.d   Fd, Fs, Ft        # func=000010  multiply double
div.s   Fd, Fs, Ft        # func=000011  divide single
div.d   Fd, Fs, Ft        # func=000011  divide double

cvt.s.d    Fd, Fs  # func=100000 prec=10001  convert d to s
cvt.s.w    Fd, Fs  # func=100000 prec=10100  convert w to s
cvt.d.s    Fd, Fs  # func=100001 prec=10000  convert s to d
cvt.d.w    Fd, Fs  # func=100001 prec=10100  convert w to d
cvt.w.s    Fd, Fs  # func=100100 prec=10000  truncate s to w
cvt.w.d    Fd, Fs  # func=100100 prec=10001  truncate d to w
round.w.s  Fd, Fs  # func=001100 prec=10000  round s to w
round.w.d  Fd, Fs  # func=001100 prec=10001  round d to w

c.eq.s  Fs, Ft     # func=110010  set flag on = single
c.eq.d  Fs, Ft     # func=110010  set flag on = double
c.lt.s  Fs, Ft     # func=111100  set flag on < single
c.lt.d  Fs, Ft     # func=111100  set flag on < double
c.le.s  Fs, Ft     # func=111110  set flag on <= single
c.le.d  Fs, Ft     # func=111110  set flag on <= double
```

```
Branch instruction format:  010001 01000 0000x iiiiiiiiiiiiiiii
    x                  0 for bc1f; 1 for bc1t
    iiiiiiiiiiiiiiii   16-bit offset
```

The floating-point comparison instructions set a status flag according to the outcome of the comparison. The following instructions test the status flag to make a conditional branch. The offset is shifted left by two bits and then added to the current program counter to get the jump address.

```
bc1f    label   #     branch if status flag false
bc1t    label   #     branch if status flag true
```