Exercising with MIPS assembly programming

This paper pretends to be of help to the student to study and make the practical exercises of assembler programming in the course of Computer technology. The following points are going to be discussed.

1. Introduction and target
2. Information about the MARS simulator
3. Assembly programming in MIPS
4. Exercises
5. Description of the MIPS processor

1. Introduction and targets

The lab exercises pretend to help the student to gain insight in the concept of Instruction Set Architecture (ISA) of a processor. For this, we focus on the MIPS processor which stands for Microprocessor without Interlocked Pipeline Stages. The practical works in two directions. First given codes are studied; can we read and understand what they are doing? Secondly, the student is given exercises where he/she solves a given problem by developing an assembly code that works for all instances of the problem in the specification

2. MARS: MIPS Assembler and Runtime Simulator

The simulator MARS facilitates exercising with assembler for a MIPS processor without having a real processor available. With MARS we can

- Load and edit assembly MIPS programs
- Run a loaded program step by step
- Monitor during the simulation the content of the registers, the processor and the memory

Downloading and installing the program

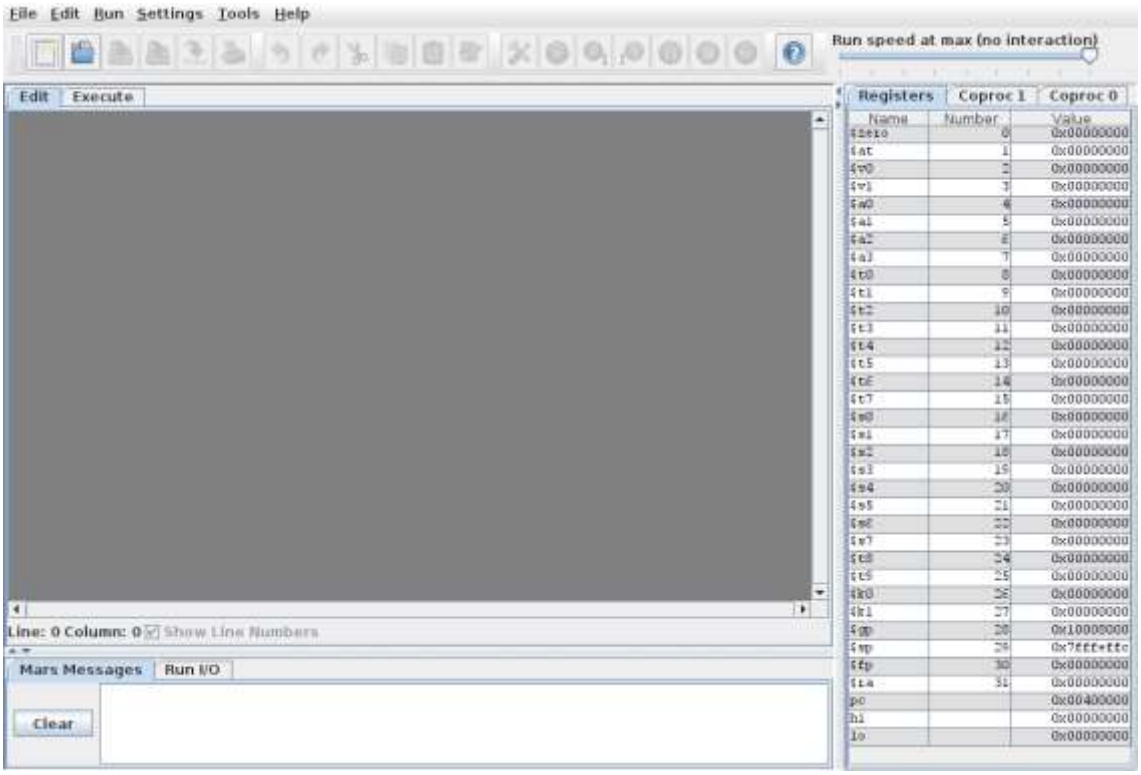
The program uses the JAVA language, so just run the .jar file after downloading from either GUAC or for instance from
http://courses.missouristate.edu/KenVollmar/MARS/index.htm

The program requires having a JAVA virtual machine installed. MARS is an example of an Integrated Development Environment (IDE); many didactic descriptions can be found on the internet. Due to the simulation environment, it aids to learn about the process of designing, assembling and running a program.

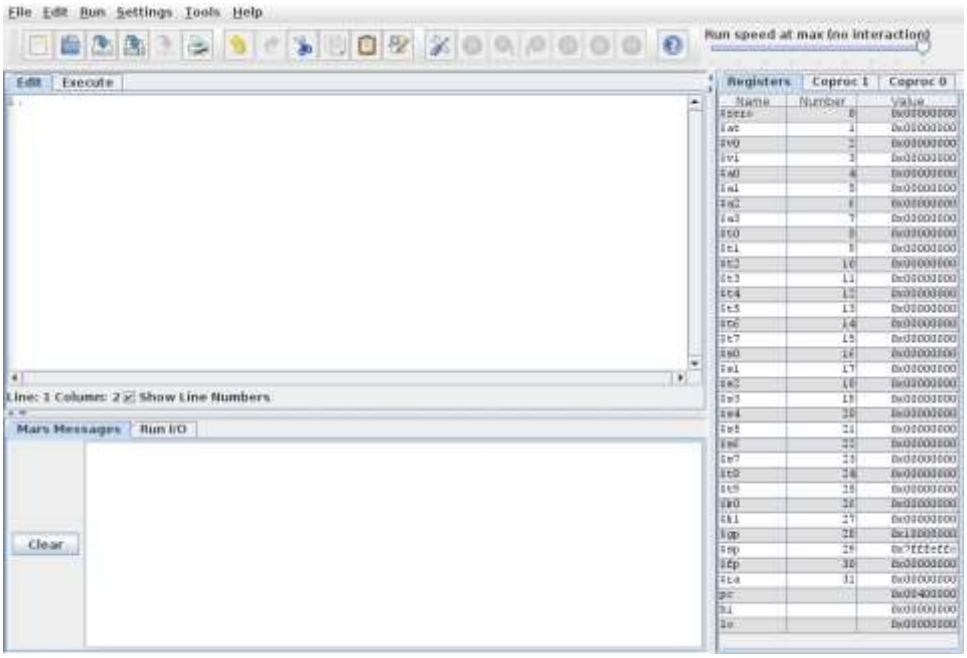
MIT licence

This means that the software has an open source status; it can be used, copied, modified, integrated to other software to adapt it to the needs of the user.

When starting up MARS, only 3 functions are available in the toolbar: **new** (file), **open** (existing assembly file) and **help**.

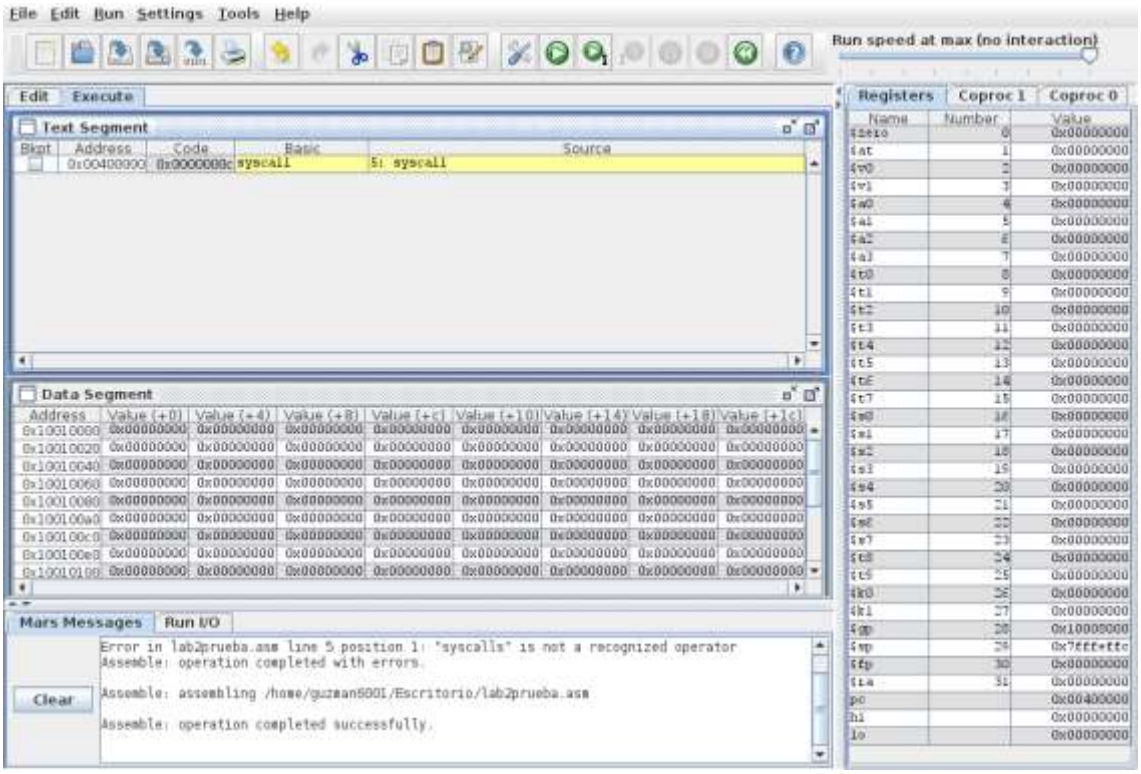


After opening a file, we can access the usual editing functions (copy and paste) and file management (save, save as, etc.) To compile the code, we use "assemble" in the "run" menu, or using the corresponding icons.



Mistakes in the syntax of the code can then be found in the window "Mars Messages". It provides an overview of the lines in the program and the type of errors. When all errors have been repaired, the Edit icon is hidden and we can start running the program by accessing to the "Execute" part of the simulator. It

shows in one window the text part of the code and in another the memory. We can either execute an individual instruction, or run the whole code. We can pause and step back in the instructions.



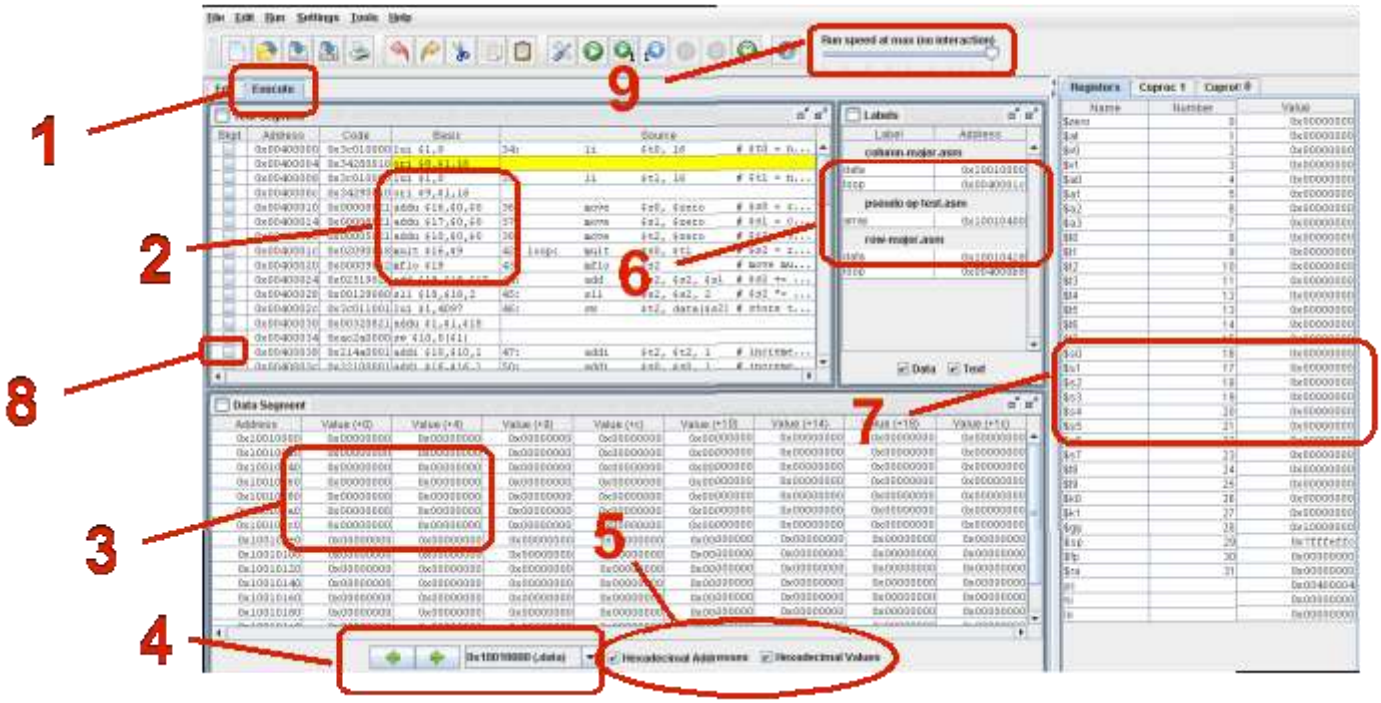
Debugging options

An IDE usually contains several tools for debugging. Besides the step by step execution, MARS also permits reloading the text and data memory.

Consider the main window of the program.

- The info tab tells us whether we are in editing or execution mode.
- The icons provide the usual options. The shaded ones are not available.
- Alternatively, one can use the menu.
- The editor is WYSIWYG (what you see is what you get)

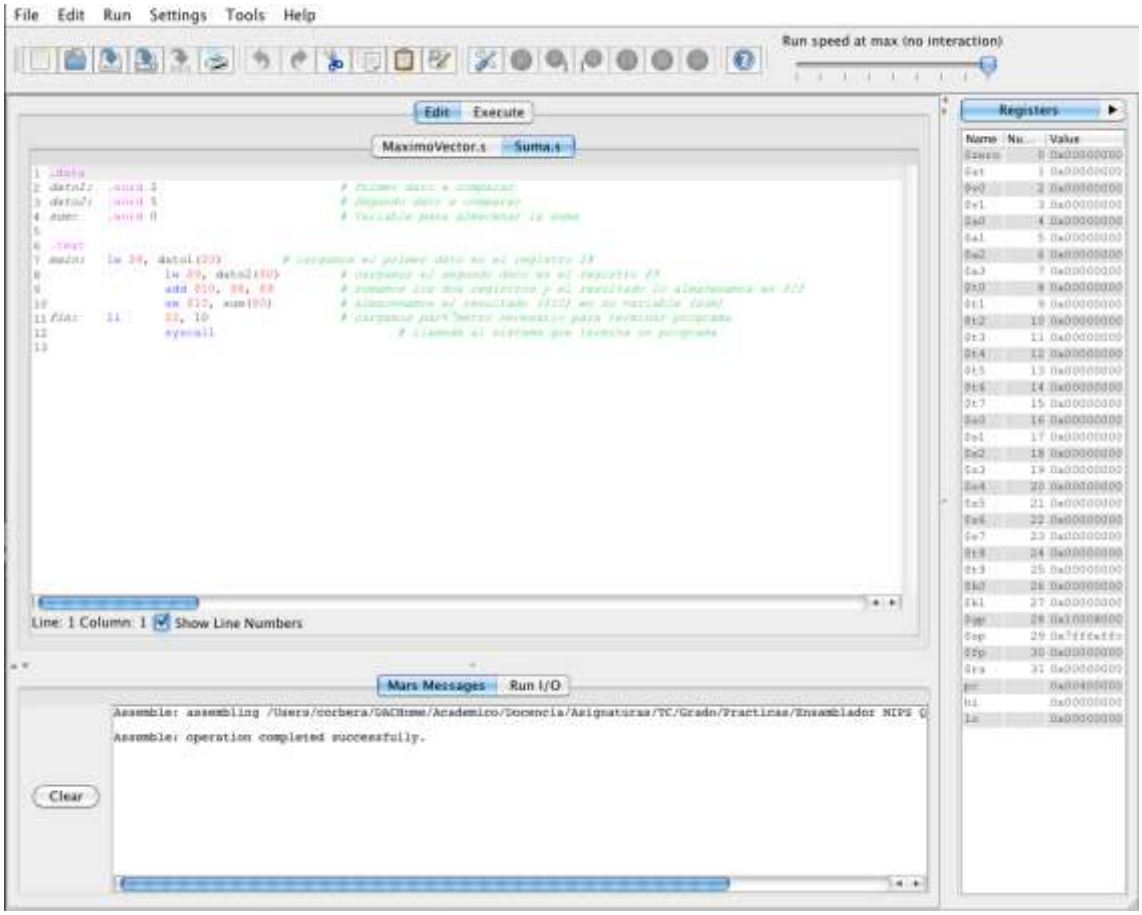
In execution mode we have the following screen.



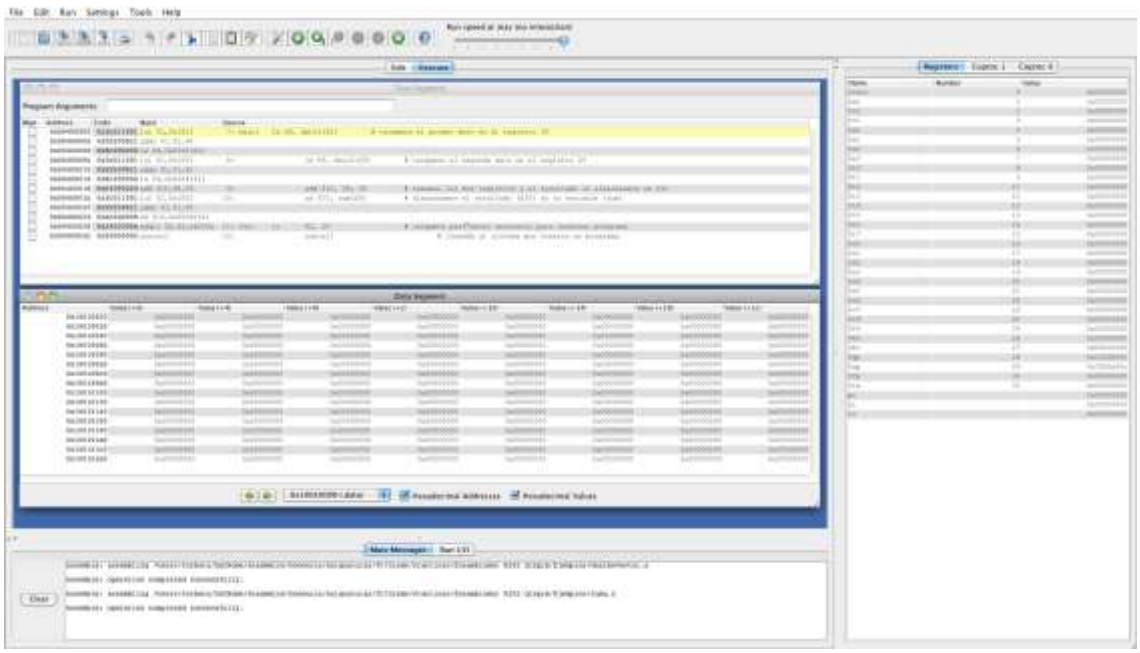
1. The tab shows we are in execution mode
2. Assembly code; address in memory, machine code and line of the source code. Note the latter can differ due to the use of pseudo instructions.
3. Stored memory values. Note, can be edited directly.
4. Window with memory content. Can be checked by arrows and menu.
5. Change to decimal or hexadecimal bases to represent values and addresses.
6. Window to access the addresses of the labels used in the code.
7. Register values can also be edited.
8. Checkbox to activate a breakpoint in an instruction.
9. The run speed can be set to follow each action.

Using the MARS simulator

The source files of the program are simple text files with the extension .s or .asm with assembly MIPS code. They can be edited in the Edit mode of MARS. After opening a source file with the menu or icon  , the screen looks more or less as follows.



The code can be edited in this window. To proceed with the analysis, we can compile the assembly code using the  icon. This leads automatically to the run screen which looks as follows.

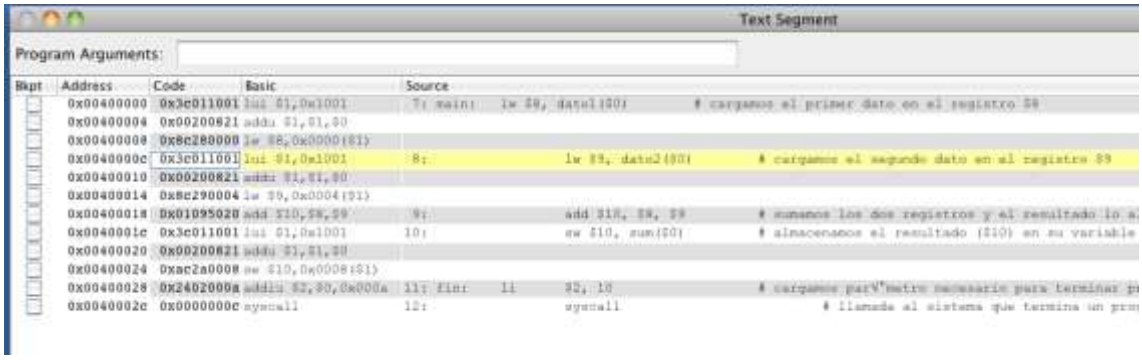


To run the program, several options can be used.

- The  icon runs the program to the end.
- The  icon resets the program and loads the initial values into the simulator. The memory content is determined by the program and the registers are usually clean, i.e. set to zero.
- The  icon executes just one instruction at a time. In this way, we can follow the development of the register and memory contents step by step.

  The instruction currently run is marked in the code. With the  icon, we can step backwards and undo the changes due to the last instruction.

Let us focus on the text segment (code) window in the screen.



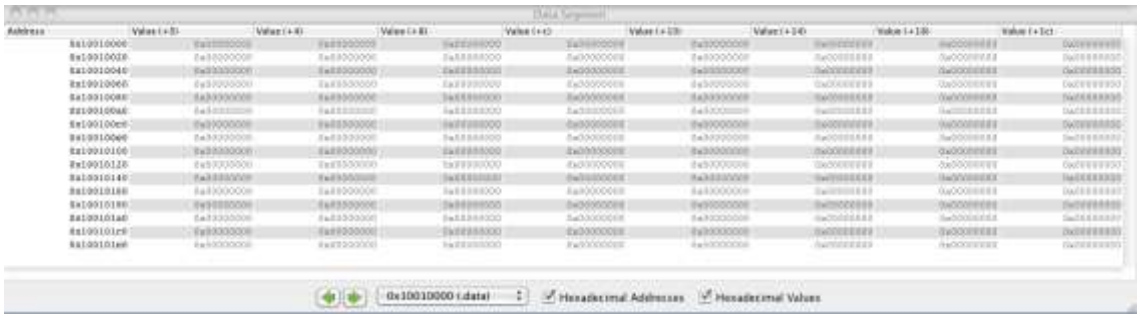
- Column **Bkpt** provides checkboxes to introduce breakpoints.
- Column **Address** gives the memory address where the instruction can be found.
- Column **Code** shows the machine code corresponding to the assembly instruction.
- Column **Basic** shows the assembly instruction of the machine instruction.
- Column **Source** gives the original source code including the comments in any strange human language. Notice that some of the original instructions have been translated into several machine instructions.

Consider the marked instruction, which is the next one to be executed.

- the instruction is stored in memory position $004000C_{hex}$
- the corresponding hexadecimal machine code is $3C011001_{hex}$
- the assembly instruction is: `lui $1, 0x1001`
- the original assembly instruction was: `lw $9, data2($0)`

Why has the original instruction been split into several?

Consider now the **Data Segment** window in the run screen. This also contains the stack and operating system space. However, we focus on the user data declared by .data in the code.



Again we observe several fields ordered into columns.

- The first column shows the base address of the row.
- The other columns give the memory content of a position addressed by the base address added to the displacement value.

For instance, the first address is 10010000_{hex} with content 00000002_{hex}; the displacement is $+0_{hex}$. The next cell shows a content 00000005_{hex}. Its exact address is 10010004_{hex} , which is the base address 10010000_{hex} plus the displacement of $+4_{hex}$.

3. Programming in MIPS assembler

In any programming language, besides using the right instructions, one should follow the corresponding syntax in order to be interpreted correctly by the compiler. A handy feature of assembler is the use of labels and directives that can be interpreted by the compiler.

A **label** identifies an instruction. Its value is the memory position of the instruction.

A **directive** helps to structure the program (`.data, .text`) and to distinguish different types of data (`.byte, .word, .ascii`, ...).

The assembly program consists of a number of variables preceded by the directive `.data` and a number of instructions preceded by the directive `.text`. In the `.data` section (variable declaration), each line consists of

- a label followed by ":"
- a data definition directive
- data
- Possible comments started by "#" Example:

```
.data
mystring:   .ascii            # characters
woord:      .word 50          # requires 4 bytes
vectorW:    .word 2,6,9,1     # 4 positions of each 4 bytes
vectorB:    .byte 3,5,7       # requires 3 bytes in total
block:      .space 30         # 30 bytes without specification
```

The `.text` section contains the program instructions. Each line contains

- An optional label
- a mnemonic to identify the instruction
- operands
- Possible comments started by '#' Example

```
main: lw $8, data1($0)  # loads data1 in register $8
```

For the illustration, consider the following code of a program sum.s that adds two variables (data1 y data2) and stores the resulting sum in another variable.

```
.data
data1:   .word 2   # First data
data2:   .word 5   # Second data
sum:     .word 0   # Variable to store the sum

.text
main: lw $8, data1($0)  # loads first data in register $8
      lw $9, data2($0)  # loads second data in register $9
      add $10, $8, $9   # sums the register contents, result to $10
      sw $10, sum($0)   # stores the result ($10) into sum
end:  li $2, 10         # loads a flag value, necessary for the syscall
      syscall           # calls the system to notify end of the program
```

4. Lab exercises

Exercise 1

Load the program *Sum.s* in the *MARS* simulator and compile. Try to find an answer to the following questions. Yes, you can!

1. The first instruction (labeled `main`), in which memory position can it be found?

`main`	

2. What is the memory position of the following variables?

`data1`	
`data2`	
`sum`	

3. What would be the reason the instruction `lw $9, data2($0)` has been decomposed into three instructions?

4. Sometimes the code uses a lui instruction. For the instruction you observe in the code, give the exact relation between the immediate argument of lui you see and the value that is loaded in the register.

5. Studying the instruction formats at the end of this paper, construct the binary code of instruction `add $10, $8, $9` and translate to hexadecimal.

Binary	Hexadecimal

Compare your code with the one provided by the simulator.

Run the complete program and check the value of variable `sum`.

The help part of the program explains all instructions and pseudo-instructions. Read the part with respect to the command syscall.

6. What is the meaning of the value 10 in register $2 before running syscall?

The fact that a program returns the right result does not prove the correctness of it. A good habit is to at least try several values for the input data and to check whether the outcome is the expected one. For instance, what happens with negative values? Please, try several instances (a difficult word for a set of input values of the program) and check the result.

Exercise 2

The source code *Maximo.s* is written with exotic comments. Load it in the MARS simulator. The target of the code is to write the maximum of two variables *data1* and *data2* into a variable *max*. Study the meaning of the instructions *slt* and *beq.*

1. Is there another way to get to the same result?

Run the program step-by-step and follow the development of memory and registers. Change the data instance such that *data2* is greater than *data1*.

2. What changes in the steps of the algorithm?
3. Change the program such that it looks for the minimum of the two values.

Exercise 3

Consider the program *Vectorsum.s* . Variable *size* represents the number of elements of the vector, *vector* contains its data and *res* the result of the sum.

1. Give for each instruction an interpretation of its contribution to the code.

```
.data
size:    .word 8
vector: .word 2, 4, 6, 8, -2 -4, -6 -7
res:     .word 0
.text
main:  lw $8, size($0)
        la $9, vector
        sub $11, $11, $11
loop:   lw $10, 0($9)
        add $11, $11, $10
        addi $9, $9, 4
        addi $8, $8, -1
        beq $8, $0, exit
        j loop
exit:  sw $11, res($0)
        li $2, 10
        syscall
```

Compile and run the code. Generate various instances of the data and check the outcomes and the steps of the program.

Consider the instructions that steer the loop.

2. Would you implement the loop in the same way? Are there alternatives?

Exercise 4

Copy the former code and rename to *vectormax.s*. Change the code, such that the result to be stored in *res* is the maximum of the elements of *vector*.

Run and check the code with several data instances. Requirements:

- The name of the assembly code file should be exactly "**vectormax.s**"
- The data segment should contain exactly the same data as *Vectorsum.s*:

```
.data
size:   .word 8
vector:      .word 2, 4, 6, 8, -2 -4, -6 -7
res:    .word 0
```

Exercise 5, leaf procedure call

Create a MIPS assembly program to determine the nth Fibonacci number. The concept is that n is the input parameter for a procedure called "**fib**", which is called by the main program. The Fibonacci numbers $f_0, f_1, f_2, f_3, \ldots$ are defined by:

$f_0 = 0$

$f_1 = 1$

$f_n = f_{n-1} + f_{n-2}$ for $n = 2, 3, 4, \ldots$

Requirements:

- The file with the assembly program should be called "**fibonacci.s**"
- The procedure that determines the nth Fibonacci number should be called "**fib**". Practically this means that the procedure starts at a label *fib*.
- The input parameter *n* of "fib" should be taken from register **$4.** The result should be put into **$2**.

Exercise 6, nested procedure linking

To practice with a **recursive** procedure, construct a procedure that determines the sum of squares of natural numbers up to number *n*:

$$f_n = \sum_{i=1}^{n} i^2$$

To make it recursive he idea is to follow the recursión $f_n = f_{n-1} + n^2$ and $f_1 = 1$.

Requirements:

- The file with the assembly program should be called "**cuadrados.s**"
- The procedure that determines the sum up to n squares should be called "**cuad**"
- The input parameter *n* of "cuad" should be taken from register **$4.** The result should be put into **$2**

5. Description of the MIPS processor

32 bits processor:

- Registers of 32 bits
- ALU with 32 bits operands
- Bus width of 32 bits

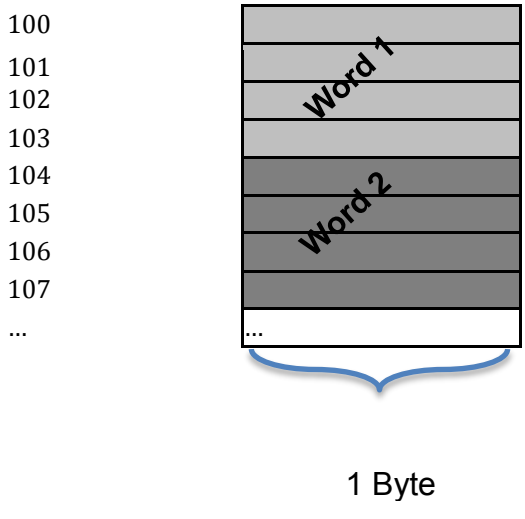
Contains 32 registers at disposal of the programmer.

Instructions of fixed size 32 bits (1 word).

Memory structure:

- Words of 4 bytes (32 bits)
- Addressing on byte level:
  - The words are 32 bits apart, so 4 positions in between two words. If one MIP instruction (one word) is in position 100_{hex} , the next one is in position 104_{hex}.
  - All instruction addresses are multiples of 4. This means that the two least significant bits are 00.
  - The program counter (PC) of 32 bits also contains an address multiple of 4 and is increased (or decreased) by a multiple of 4.
- Memory capacity in bytes: 2^{32}
- Number of words that fit (can be addressed): $2^{32}/4 = 2^{30}$

MEMORY



CUADRO RESUMEN DEL LENGUAJE ENSAMBLADOR BÁSICO DEL MIPS-R2000

CARGA

lw rt,dirección *Carga palabra* **I**

Carga los 32 bits almacenados en la palabra de memoria especificada por dirección en el registro rt.

```
lw $s0, 12($a0)   #  $s0 ← Mem[ 12 + $a0 ]
```

lb rt, dirección *Carga byte y extiende signo* **I**

Carga los 8 bits almacenados en el byte de memoria especificado por dirección en el LSB del registro rt y extiende el signo

```
lb $s0, 12($a0)   #  $s0(7..0) ← Mem[ 12 + $a0 ](1byte)
                  #  $s0(31..8) ← $s0(7)
```

lbu rt, dirección *Carga byte y no extiende signo* **I**

Carga los 8 bits almacenados en el byte de memoria especificado por dirección en el LSB del registro rt sin extender el signo

```
lbu $s0, 12($a0)  #  $s0 ← 0x000000(Mem[ 12 + $a0 ])(1byte)
```

lh rt, dirección *Carga media palabra y ext. signo* **I**

Carga media palabra (16 bits) almacenada en la media palabra de memoria especificada por la dirección en la parte baja del registro rt y extiende el signo

```
lh $s0, 12($a0)   #  $s0 (15..0)← Mem[ 12 + $a0 ](2bytes)
                  #  $s0 (31..16)← $s0(15)
```

lhu rt, dirección *Carga media palabra y no ext. signo* **I**

Carga media palabra (16 bits) almacenada en la media palabra de memoria especificada por la dirección en la parte baja del registro rt y no extiende el signo

```
lhu $s0, 12($a0)  #  $s0← 0x0000Mem[ 12 + $a0 ])(2bytes)
```

la reg, dirección *Carga dirección* **PS**

Carga la dirección calculada en reg

```
la $s0, VAR       #  $s0← dir. asociada a etiqueta VAR
```

lui rt, dato *Carga inmediata superior* **I**

Carga el dato inmediato en los 16 MSB del registro rt

```
lui $s0, 12       #  $s0(31..16) ← 12
                  #  $s0(15..0) ← 0x0000
```

li reg, dato *Carga inmediato* **PS**

Carga el dato inmediato en el registro reg.

```
li $s0, 12        #  $s0 ← 12
```

ARITMÉTICAS

add rd, rs, rt *Suma* **R**

Suma el contenido de los registros rs y rt, considerando el signo. El resultado se almacena en el registro rd

```
add $t0, $a0, $a1        # $t0  ←$a0 + $a1
```

addu rd, rs, rt *Suma sin signo* **R**

Suma el contenido de los registros rs y rt, sin considerar el signo. El resultado se almacena en el registro rd

```
addu $t0, $a0, $a1       # $t0  ←$a0 + $a1
```

sub rd, rs, rt *Resta* **R**

Resta el contenido de los registro rs y rt considerando el signo. El resultado se almacena en el registro rd

```
sub $t0, $a0, $a1        # $t0  ←$a0 - $a1
```

subu rd, rs, rt *Resta sin signo* **R**

Resta el contenido de los registros rs y rt, sin considerar el signo. El resultado se almacena en el registro r.

```
subu $t0, $a0, $a1       # $t0  ←$a0 - $a1
```

addi rt, rs, valor *Suma inmediata* **I**

Suma el contenido del registro rs con el valor inmediato, considerando el signo. El resultado se almacena en el registro rt.

```
addi $t0, $a0, -24       # $t0  ←$a0 + (-24)
```

addiu rt, rs, valor *Suma inmediata sin signo* **I**

Suma el contenido del registro rs con el valor inmediato, sin considerar el signo. El resultado se almacena en el registro rt.

```
addiu $t0, $a0, 24              # $t0  ←$a0 + 24
```

mult rs, rt *Multiplicación* **R**

Multiplica el contenido de los registros rs y rt. Los 32 MSB del resultado se almacenan en el registro HI y los 32 LSB en el registro LO

```
mult $s0, $s1    # $HI  ←($s0 * $s1) (31…16)
                 # $LO←($s0 * $s1) (15…0)
```

div rs, rt *División* **R**

Divide el registro rs por el rt. El cociente se almacena en LO y el resto en HI.

```
div $s0, $s1     # $LO  ← $s0 / $s1
                 # $HI ← $s0 % $s1
```

COMPARACIONES

slt rd, rs, rt *Activa si menor* **R**

Pone el registro rd a 1 si rs es menor que rt y a 0 en caso contrario

```
slt $t0, $a0, $a1        # if ( $a0 < $a1) $t0  ← 1
                         # else $t0  ← 0
```

slti rt, rs, inm *Activa si menor con inmediato* **I**

Pone el registro rt a 1 si rs es menor que el dato inmediato inm y a 0 en caso contrario

```
slti $t0, $a0, -15       # if ( $a0 < -15) $t0  ← 1
                         # else $t0  ← 0
```

seq rdest, rsrc1, rsrc2 *Activa si igual* **PS**

Pone el registro rdest a 1 si rsrc1 es igual que rsrc2 y a 0 en caso contrario

```
seq $t0, $a0, $a2        # if ( $a0 == $a2) $t0  ← 1
                         # else $t0  ← 0
```

sge rdest, rsrc1, rsrc2 *Activa si mayor o igual* **PS**

Pone el registro rdest a 1 si rsrc1 es mayor o igual que rsrc2 y a 0 en caso contrario

```
sge $t0, $a0, $a2        # if ( $a0 >= $a2) $t0  ← 1
                         # else $t0  ← 0
```

sgt rdest, rsrc1, rsrc2 *Activa si mayor* **PS**

Pone el registro rdest a 1 si rsrc1 es mayor que rsrc2 y a 0 en caso contrario

```
sgt $t0, $a0, $a2        # if ( $a0 > $a2) $t0  ← 1
                         # else $t0  ← 0
```

sle rdest, rsrc1, rsrc2 *Activa si menor o igual* **PS**

Pone el registro rdest a 1 si rsrc1 es menor o igual que rsrc2 y a 0 en caso contrario

```
sle $t0, $a0, $a2        # if ( $a0 <= $a2) $t0  ← 1
                         # else $t0  ← 0
```

sne rdest, rsrc1, rsrc2 *Activa si no igual* **PS**

Pone el registro rdest a 1 si rsrc1 es diferente de rsrc2 y a 0 en caso contrario

```
sne $t0, $a0, $a2        # if ( $a0 != $a2) $t0  ← 1
                         # else $t0  ← 0
```

MIPS instructions set in an exotic language

CUADRO RESUMEN DEL LENGUAJE ENSAMBLADOR BÁSICO DEL MIPS-R2000

ALMACENAMIENTO

sw rt, dirección *Almacena palabra* **I**

Almacena el contenido del registro rt en la palabra de memoria indicada por dirección

```
sw $s0, 12($a0)  # Mem[ 12 + $a0 ] ←$s0
```

sb rt, dirección *Almacena byte* **I**

Almacena el LSB del registro en el byte de memoria indicado por dirección

```
sb $s0, 12($a0)  # Mem[ 12 + $a0 ] ←$s0(7..0)
```

sh rt, dirección *Almacena media palabra* **I**

Almacena en los 16 bits de menos peso del registro en la media palabra de memoria indicada por dirección.

```
sh $s0, 12($a0)  # Mem[ 12 + $a0 ] ←$s0(15..0)
```

LÓGICAS

and rd, rs, rt *AND entre registros* **R**

Operación AND bit a bit entre los registros rs y rt. El resultado se almacena en rd

```
and $t0, $a0, $a1          # $t0 ←$a0 & $a1
```

andi rt, rs, inm *AND con inmediato* **I**

Operación AND bit a bit entre el dato inmediato, extendiento ceros, y el registro rs. El resultado se almacena en rt.

```
andi $t0, $a0, 0xA1FF      # $t0 ←$a0 & (0x0000A1FF)
```

or rd, rs, rt *OR entre registros* **R**

Operación OR bit a bit entre los registros rs y rt. El resultado se almacena en rd

```
or $t0, $a0, $a1           # $t0 ←$a0 | $a1
```

ori rt, rs, inm *OR con inmediato* **I**

Operación OR bit a bit entre el dato inmediato, extendiendo ceros, y el registro rs. El resultado se almacena en rt.

```
ori $t0, $a0, 0xA1FF       # $t0 ←$a0 | (0x0000A1FF)
```

MOVIMIENTO ENTRE REGISTROS

mfhi rd *mueve desde HI* **R**

Transfiere el contenido del registro HI al registro rd.

```
mfhi $t0          # $t0 ← HI
```

mflo rd *mueve desde LO* **R**

Transfiere el contenido del registro LO al registro rd.

```
mflo $t1          # $t1 ← LO
```

FORMATO DE LAS INSTRUCCIONES

tipo **R**

inst(6 bits)	rs(5bits)	rt (5bits)	rd(5bits)	shamt(5bits)	co (6bits)

tipo **I**

inst(6bits)	rs(5bits)	rt(5bits)	inm(16 bits)

tipo **J**

inst(6bits)	objetivo (26 bits)

DESPLAZAMIENTO

sll rd, rt, shamt *Desplamiento logico a la izquierda* **R**

Desplaza el registro rt a la izquierda tantos bits como indica shamt

```
sll $t0, $t1, 16     # $t0 ← $t1 << 16
```

srl rd, rt, shamt *Desplazamiento lógico a la derecha* **R**

Desplaza el registro rt a la derecha tantos bits como indica shamt.

```
srl $s0,$t1, 4       # $s0 ← $t1 >> 4
```

sra rd, rt, shamt *Desplaz. aritmético a la derecha* **R**

Desplaza el registro rt a la derecha tantos bits como indica shamt. Los bits MSB toman el mismo valor que el bit de signo de rt. El resultado se almacena en rd

```
sra $s0,$t1, 4       # $s0 ← $t1 >> 4
                     # $s0(31..28) ← $t1(31)
```

SALTOS INCONDICIONALES

j dirección *Salto incondicional* **J**

Salta a la instrucción apuntada por la etiqueta dirección

```
j finbucle           # $pc ← dirección etiqueta finbucle
```

jal dirección *Saltar y enlazar* **J**

Salta a la instrucción apuntada por la etiqueta dirección y almacena la dirección de la instrucción siguiente en $ra

```
jal rutina           # $pc ← dirección etiqueta rutina
                     # $ra ← dirección siguiente instrucción
```

jr rs *Saltar a registro* **R**

Salta a la instrucción apuntada por el contenido del registro rs.

```
jr $ra     # $pc ← $ra
```

SALTOS CONDICIONALES

beq rs, rt, etiqueta *Salto si igual* **I**

Salta a etiqueta si rs es igual a rt

```
beq $t0, $t1, DIR          # if ( $t0=$t1) $pc ← DIR
```

bgez rs, etiqueta *Salto si mayor o igual que cero* **I**

Salta a etiqueta si rs es mayor o igual que 0

```
bgez $t0, SLT              # if ( $t0>=0) $pc ← SLT
```

bgtz rs, etiqueta *Salto si mayor que cero* **I**

Salta a etiqueta si rs es mayor que 0

```
bgtz $t0, SLT              # if ( $t0>0) $pc ← SLT
```

blez rs, etiqueta *Salto si menor o igual que cero* **I**

Salta a etiqueta si rs es menor o igual que 0

```
blez $t1, ETQ              # if ( $t1<=0) $pc ← ETQ
```

bltz rs, etiqueta *Salto si menor que cero* **I**

Salta a etiqueta si rs es menor que 0

```
bltz $t1, ETQ              # if ( $t1<0) $pc ← ETQ
```

bne rs, rt, etiqueta *Salto si distinto* **I**

Salta a etiqueta si rs es diferente de rt

```
bne $t0, $t1, DIR          # if ( $t0<>$t1) $pc ← DIR
```

bge reg1, reg2, etiq *Salto mayor o igual* **PS**

Salta a etiq si reg1 es mayor o igual que reg2

```
bge $t0, $t1, DIR          # if ( $t0>=$t1) $pc ← DIR
```

bgt reg1, reg2, etiq *Salto mayor* **PS**

Salta a etiq si reg1 es mayor que reg2

```
bgt $t0, $t1, DIR          # if ( $t0>$t1) $pc ← DIR
```

ble reg1, reg2, etiq *Salto menor o igual* **PS**

Salta a etiq si reg1 es menor o igual que reg2

```
ble $t0, $t1, DIR          # if ( $t0<=$t1) $pc ← DIR
```

blt reg1, reg2, etiq *Salto menor* **PS**

Salta a etiq si reg1 es menor que reg2

```
blt $t0, $t1, DIR          # if ( $t0<$t1) $pc ← DIR
```

MIPS Instruction Reference (http://www.mrc.uidaho.edu/mrc/people/jff/digital/MIPSir.html)

This is a description of the MIPS instruction set, their meanings, syntax, semantics, and bit encodings. The syntax given for each instruction refers to the assembly language syntax supported by the MIPS assembler. Hyphens in the encoding indicate "don't care" bits which are not considered when an instruction is being decoded.

General purpose registers (GPRs) are indicated with a dollar sign ($). The words SWORD and UWORD refer to 32-bit signed and 32-bit unsigned data types, respectively. The function advance_pc (int) is used in many of the instruction descriptions. This function is defined as follows:

```
void advance_pc (SWORD offset)
{
    PC  =  nPC;
   nPC  += offset;

}
```

The instruction descriptions are given below:

ADD – ***Add (with overflow)***

Description:	Adds two registers and stores the result in a register
Operation:	$d = $s + $t; advance_pc (4);
Syntax:	add $d, $s, $t
Encoding:	`0000 00ss ssst tttt dddd d000 0010 0000`

ADDI -- ***Add immediate (with overflow)***

Description:	Adds a register and a sign-extended immediate value and stores the result in a register
Operation:	$t = $s + imm; advance_pc (4);
Syntax:	addi $t, $s, imm
Encoding:	`0010 00ss ssst tttt iiii iiii iiii iiii`

ADDIU -- ***Add immediate unsigned (no overflow)***

Description:	Adds a register and a sign-extended immediate value and stores the result in a register
Operation:	$t = $s + imm; advance_pc (4);
Syntax:	addiu $t, $s, imm
Encoding:	`0010 01ss ssst tttt iiii iiii iiii iiii`

ADDU -- ***Add unsigned (no overflow)***

Description:	Adds two registers and stores the result in a register
Operation:	$d = $s + $t; advance_pc (4);
Syntax:	addu $d, $s, $t
Encoding:	`0000 00ss ssst tttt dddd d000 0010 0001`

AND -- ***Bitwise and***

Description:	Bitwise ands two registers and stores the result in a register
Operation:	$d = $s & $t; advance_pc (4);
Syntax:	and $d, $s, $t
Encoding:	`0000 00ss ssst tttt dddd d000 0010 0100`

ANDI -- ***Bitwise and immediate***

Description:	Bitwise ands a register and an immediate value and stores the result in a register
Operation:	$t = $s & imm; advance_pc (4);
Syntax:	andi $t, $s, imm
Encoding:	`0011 00ss ssst tttt iiii iiii iiii iiii`

BEQ -- ***Branch on equal***

Description:	Branches if the two registers are equal
Operation:	if $s == $t advance_pc (offset << 2)); these advance_pc (4);
Syntax:	beq $s, $t, offset
Encoding:	`0001 00ss ssst tttt iiii iiii iiii iiii`

BGEZ -- ***Branch on greater than or equal to zero***

Description:	Branches if the register is greater than or equal to zero
Operation:	if $s >= 0 advance_pc (offset << 2)); these advance_pc (4);
Syntax:	bgez $s, offset
Encoding:	`0000 01ss sss0 0001 iiii iiii iiii iiii`

BGEZAL -- ***Branch on greater than or equal to zero and link***

Description:	Branches if the register is greater than or equal to zero and saves the return address in $31
Operation:	if $s >= 0 $31 = PC + 8 (or nPC + 4); advance_pc (offset << 2)); these advance_pc (4);
Syntax:	bgezal $s, offset
Encoding:	`0000 01ss sss1 0001 iiii iiii iiii iiii`

BGTZ -- ***Branch on greater than zero***

Description:	Branches if the register is greater than zero
Operation:	if $s > 0 advance_pc (offset << 2)); these advance_pc (4);
Syntax:	bgtz $s, offset
Encoding:	`0001 11ss sss0 0000 iiii iiii iiii iiii`

BLEZ -- ***Branch on less than or equal to zero***

Description:	Branches if the register is less than or equal to zero
Operation:	if $s <= 0 advance_pc (offset << 2)); these advance_pc (4);
Syntax:	blez $s, offset
Encoding:	`0001 10ss sss0 0000 iiii iiii iiii iiii`

BLTZ -- ***Branch on less than zero***

Description:	Branches if the register is less than zero
Operation:	if $s < 0 advance_pc (offset << 2)); these advance_pc (4);
Syntax:	bltz $s, offset
Encoding:	`0000 01ss sss0 0000 iiii iiii iiii iiii`

BLTZAL -- ***Branch on less than zero and link***

Description:	Branches if the register is less than zero and saves the return address in $31
Operation:	if $s < 0 $31 = PC + 8 (or nPC + 4); advance_pc (offset << 2)); these advance_pc (4);
Syntax:	bltzal $s, offset
Encoding:	`0000 01ss sss1 0000 iiii iiii iiii iiii`

BNE -- *Branch on not equal*

Description:	Branches if the two registers are not equal
Operation:	if $s != $t advance_pc (offset << 2)); these advance_pc (4);
Syntax:	bne $s, $t, offset
Encoding:	`0001 01ss ssst tttt iiii iiii iiii iiii`

DIV -- *Divide*

Description:	Divides $s by $t and stores the quotient in $LO and the remainder in $HI
Operation:	$LO = $s / $t; $HI = $s % $t; advance_pc (4);
Syntax:	div $s, $t
Encoding:	`0000 00ss ssst tttt 0000 0000 0001 1010`

DIVU -- *Divide unsigned*

Description:	Divides $s by $t and stores the quotient in $LO and the remainder in $HI
Operation:	$LO = $s / $t; $HI = $s % $t; advance_pc (4);
Syntax:	divu $s, $t
Encoding:	`0000 00ss ssst tttt 0000 0000 0001 1011`

J -- *Jump*

Description:	Jumps to the calculated address
Operation:	PC = nPC; nPC = (PC & 0xf0000000) | (target << 2);
Syntax:	j target
Encoding:	`0000 10ii iiii iiii iiii iiii iiii iiii`

JAL -- *Jump and link*

Description:	Jumps to the calculated address and stores the return address in $31
Operation:	$31 = PC + 8 (or nPC + 4); PC = nPC; nPC = (PC & 0xf0000000) | (target << 2);
Syntax:	jal target
Encoding:	`0000 11ii iiii iiii iiii iiii iiii iiii`

JR -- *Jump register*

Description:	Jump to the address contained in register $s
Operation:	PC = nPC; nPC = $s;
Syntax:	jr $s
Encoding:	`0000 00ss sss0 0000 0000 0000 0000 1000`

LB -- *Load byte*

Description:	A byte is loaded into a register from the specified address.
Operation:	$t = MEM[$s + offset]; advance_pc (4);
Syntax:	lb $t, offset($s)
Encoding:	`1000 00ss ssst tttt iiii iiii iiii iiii`

LUI -- *Load upper immediate*

Description:	The immediate value is shifted left 16 bits and stored in the register. The lower 16 bits are zeroes.
Operation:	$t = (imm << 16); advance_pc (4);
Syntax:	lui $t, imm
Encoding:	`0011 11-- ---t tttt iiii iiii iiii iiii`

LW -- *Load word*

Description:	A word is loaded into a register from the specified address.
Operation:	$t = MEM[$s + offset]; advance_pc (4);
Syntax:	lw $t, offset($s)
Encoding:	`1000 11ss ssst tttt iiii iiii iiii iiii`

MFHI -- *Move from HI*

Description:	The contents of register HI are moved to the specified register.
Operation:	$d = $HI; advance_pc (4);
Syntax:	mfhi $d
Encoding:	`0000 0000 0000 0000 dddd d000 0001 0000`

MFLO -- *Move from LO*

Description:	The contents of register LO are moved to the specified register.
Operation:	$d = $LO; advance_pc (4);
Syntax:	mflo $d
Encoding:	`0000 0000 0000 0000 dddd d000 0001 0010`

MULT -- *Multiply*

Description:	Multiplies $s by $t and stores the result in $LO.
Operation:	$LO = $s * $t; advance_pc (4);
Syntax:	mult $s, $t
Encoding:	`0000 00ss ssst tttt 0000 0000 0001 1000`

MULTU -- *Multiply unsigned*

Description:	Multiplies $s by $t and stores the result in $LO.
Operation:	$LO = $s * $t; advance_pc (4);
Syntax:	multu $s, $t
Encoding:	`0000 00ss ssst tttt 0000 0000 0001 1001`

NOOP -- *no operation*

Description:	Performs no operation.
Operation:	advance_pc (4);
Syntax:	noop
Encoding:	`0000 0000 0000 0000 0000 0000 0000 0000`

Note: The encoding for a NOOP represents the instruction SLL $0, $0, 0 which has no side effects. In fact, nearly every instruction that has $0 as its destination register will have no side effect and can thus be considered a NOOP instruction.

OR -- *Bitwise or*

Description:	Bitwise logical ors two registers and stores the result in a register
Operation:	$d = $s | $t; advance_pc (4);
Syntax:	or $d, $s, $t
Encoding:	`0000 00ss ssst tttt dddd d000 0010 0101`

ORI -- *Bitwise or immediate*

Description:	Bitwise ors a register and an immediate value and stores the result in a register
Operation:	$t = $s | imm; advance_pc (4);
Syntax:	ori $t, $s, imm
Encoding:	`0011 01ss ssst tttt iiii iiii iiii iiii`

SB -- *Store byte*

Description:	The least significant byte of $t is stored at the specified address.
Operation:	MEM[$s + offset] = (0xff & $t); advance_pc (4);
Syntax:	sb $t, offset($s)
Encoding:	`1010 00ss ssst tttt iiii iiii iiii iiii`

SLL -- *Shift left logical*

Description:	Shifts a register value left by the shift amount listed in the instruction and places the result in a third register. Zeroes are shifted in.
Operation:	$d = $t << h; advance_pc (4);
Syntax:	sll $d, $t, h
Encoding:	`0000 00ss ssst tttt dddd dhhh hh00 0000`

SLLV -- *Shift left logical variable*

Description:	Shifts a register value left by the value in a second register and places the result in a third register. Zeroes are shifted in.
Operation:	$d = $t << $s; advance_pc (4);
Syntax:	sllv $d, $t, $s
Encoding:	`0000 00ss ssst tttt dddd d--- --00 0100`

SLT -- *Set on less than (signed)*

Description:	If $s is less than $t, $d is set to one. It gets zero otherwise.
Operation:	if $s < $t $d = 1; advance_pc (4); these $d = 0; advance_pc (4);
Syntax:	slt $d, $s, $t
Encoding:	`0000 00ss ssst tttt dddd d000 0010 1010`

SLTI -- *Set on less than immediate (signed)*

Description:	If $s is less than immediate, $t is set to one. It gets zero otherwise.
Operation:	if $s < imm $t = 1; advance_pc (4); these $t = 0; advance_pc (4);
Syntax:	slti $t, $s, imm
Encoding:	`0010 10ss ssst tttt iiii iiii iiii iiii`

SLTIU -- *Set on less than immediate unsigned*

Description:	If $s is less than the unsigned immediate, $t is set to one. It gets zero otherwise.
Operation:	if $s < imm $t = 1; advance_pc (4); these $t = 0; advance_pc (4);
Syntax:	sltiu $t, $s, imm
Encoding:	`0010 11ss ssst tttt iiii iiii iiii iiii`

SLTU -- *Set on less than unsigned*

Description:	If $s is less than $t, $d is set to one. It gets zero otherwise.
Operation:	if $s < $t $d = 1; advance_pc (4); these $d = 0; advance_pc (4);
Syntax:	sltu $d, $s, $t
Encoding:	`0000 00ss ssst tttt dddd d000 0010 1011`

SRA -- *Shift right arithmetic*

Description:	Shifts a register value right by the shift amount (shamt) and places the value in the destination register. The sign bit is shifted in.
Operation:	$d = $t >> h; advance_pc (4);
Syntax:	sra $d, $t, h
Encoding:	`0000 00-- ---t tttt dddd dhhh hh00 0011`

SRL -- *Shift right logical*

Description:	Shifts a register value right by the shift amount (shamt) and places the value in the destination register. Zeroes are shifted in.
Operation:	$d = $t >> h; advance_pc (4);
Syntax:	srl $d, $t, h
Encoding:	`0000 00-- ---t tttt dddd dhhh hh00 0010`

SRLV -- *Shift right logical variable*

Description:	Shifts a register value right by the amount specified in $s and places the value in the destination register. Zeroes are shifted in.
Operation:	$d = $t >> $s; advance_pc (4);
Syntax:	srlv $d, $t, $s
Encoding:	`0000 00ss ssst tttt dddd d000 0000 0110`

SUB -- *Subtract*

Description:	Subtracts two registers and stores the result in a register
Operation:	$d = $s - $t; advance_pc (4);
Syntax:	sub $d, $s, $t
Encoding:	`0000 00ss ssst tttt dddd d000 0010 0010`

SUBU -- *Subtract unsigned*

Description:	Subtracts two registers and stores the result in a register
Operation:	$d = $s - $t; advance_pc (4);
Syntax:	subu $d, $s, $t
Encoding:	`0000 00ss ssst tttt dddd d000 0010 0011`

SW -- *Store word*

Description:	The contents of $t is stored at the specified address.
Operation:	MEM[$s + offset] = $t; advance_pc (4);
Syntax:	sw $t, offset($s)
Encoding:	`1010 11ss ssst tttt iiii iiii iiii iiii`

SYSCALL -- *System call*

Description:	Generates a software interrupt.
Operation:	advance_pc (4);
Syntax:	syscall
Encoding:	`0000 00-- ---- ---- ---- ---- --00 1100`

The syscall instruction is described in more detail on the System Calls page.

XOR -- ***Bitwise exclusive or***

Description:	Exclusive ors two registers and stores the result in a register
Operation:	$d = $s ^ $t; advance_pc (4);
Syntax:	xor $d, $s, $t
Encoding:	`0000 00ss ssst tttt dddd d--- --10 0110`

XORI -- ***Bitwise exclusive or immediate***

Description:	Bitwise exclusive ors a register and an immediate value and stores the result in a register
Operation:	$t = $s ^ imm; advance_pc (4);
Syntax:	xori $t, $s, imm
Encoding:	`0011 10ss ssst tttt iiii iiii iiii iiii`