MIPS Assembly Language

1 Simple Example Program

```
################################################################
#   Description:
#       Simple example program
################################################################

################################################################
#   Main program
################################################################

# Variables for main
        .data
greeting: .asciiz "Hello, world!\n"

# Main body
        .text
main:
        li      $v0, 4
        la      $a0, greeting
        syscall

        # Return to calling program
        jr      $ra
```

2 Memory segments

We must mark each part of the program as text(text segment or called code segment) or data(variable segment) reside in the same memory while the program is running

```
   .kdata
      #variables for kernel
   .ktext
      # text for kernel

   .data
      # Variables for main

   .text
      # Main body
      # Your code start here
```

The **.text** segments can contain only instructions, while the **.data** segments contain only variable definitions.

3 Instruction Format

An MIPS instruction has the following structure:

```
label:  opcode/directive    operand, operand, operand   # comment
```

3.1 Label

- The label portion of a statement must begin in column 1 and must end with a ':'. The ':' is not part of the label. It only serves to visually distinguish a new label definition from other program elements.
- Each label represents a memory address in assembly language. It could be the address of data or the address of an instruction (i.e. labels can appear in both .text and .data sections).

- A label represents the address of the instruction or data element that immediately follows it, whether it follows on the same line or a subsequent line.
- Labels defined in a .data section are like variable names and follow the same naming rules. They must begin with a letter or underscore, and can contains letters, underscores, and digits. Variable names cannot be keywords.
- Labels defined in a .text section represent the address of an instruction, and are used as arguments by jump, branch, and return instructions to "go to" that instruction.

3.2 Comments

- A comment is anything start from a “#” to the end of the line.

```
Example: # this is a comment
```

- Block comments are simply formed from multiple line comments

```
Example:
#################################################################
# This is a block comment
#################################################################
```

4 Directives

- Directives do not represent machine instructions, hence they are not executed at run-time. They direct the assembler to do something while translating the program to machine language, such as allocate space for a variable and give it an initial value, which it will have when the program begins executing.
- Directives can be distinguished from instructions by the fact that they begin with a ’.’.
- Directives must be indented (they cannot begin in column 1).
- Data allocation directives are somewhat like variable definitions in high-level languages, but are not quite as meaningful. They usually, but not necessarily, follow a newly defined label (variable) and must be followed by one or more initial values.

Directive	Descriptions
.data <addr>	The following data items should be stored in the data segment. If the optional argument addr is present, the items are stored beginning at address addr.
.text <addr>	The next items are put in the user text segment. If the optional argument addr is present, the items are stored beginning at address addr.
.kdata <addr>	The following data items should be stored in the kernel data segment. If the optional argument addr is present, the items are stored beginning at address addr.
.ktext <addr>	The next items are put in the kernel text segment. If the optional argument addr is present, the items are stored beginning at address addr.
.word	32-bit integer
.half	16-bit integer
.byte	08-bit integer
.float	floating point IEEE 754 single precision
.double	floating point IEEE 754 double precision
.space	Uninitialized memory block (byte)
.ascii	Store the string in memory, but do not null-terminate it.
.asciiz	Store the string in memory and null-terminate it.
.align	Align the next datum on a 2n byte boundary.

4.1 Directives example

```
    .data
age:    .word 30      # 32-bit word initialized with decimal
class:  .half 0x10    # 16-bit word initialized with hex
grade:  .byte 08      # 8-bit word initialized with octal
```

```
        .align  2     # Aligns next element to multiple of 2^2

height: .word   70
gpa:    .float  3.65    # 32-bit floating point

# Array of 6 integers
ArrayInt:   .word   1, 2, 3, 54, 24, 120

# Initialization same value 123 for array of 5 integers.
ArrayInt: .word 123:5

# Uninitialized space of 1024 byte buffer
ArrayBytr:  .space  1024    # 1 KB buffer

# String within null-terminate at the end
Greeting:  .asciiz "Hello, world!"

# String without null-terminate at the end
String:   .ascii  "This string is without a null byte"
.text
```

5 syscall instruction

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.

5.1 terminate execution

```
li $v0, 10   #terminate execution
syscall
```

5.2 print integer

```
addi $a0, $0, 100   # Assign an integer to a0
li   $v0, 1         # Print integer a0
syscall
```

5.3 print float

```
  mov.s $f12, $f3   # Move contents of register $f3 to register $f12
  li $v0, 2         # Print float number
  syscall
```

5.4 print string

reference to 1

5.5 read integer

```
li  $v0, 5                 # Get integer mode,
                           # $v0 contains integer read
syscall
add $a0, $0, $v0           # Move integer from $v0 to register $a0
```

5.6 read float

```
li  $v0, 6                 # Get float number
syscall                    # $f0 contains float read
```

5.7 print character

```
addi $a0, $0, ’A’    # Display character ’A’
li   $v0, 11         # print char
syscall
```

5.8 read character

```
li  $v0, 12           # Get character
                      # $v0 contains character read
syscall
add $a0, $0, $v0      # Move character to register
```

5.9 read string

```
.data
strIn: .space 100    #create 100 bytes space

.text
main:
   la   $a0, strIn   # Get address to store string
   addi $a1, $0, 10  # Input 10 characters into string (includes end of string)
   li   $v0, 8       # Get string mode
   syscall
```