MIPS Hello World

```
# PROGRAM: Hello, World!

  .data            # Data declaration section

out_string:    .asciiz   "\nHello, World!\n"

  .text            # Assembly language instructions

main:              # Start of code section
  li     $v0, 4            # system call code for printing string = 4
  la     $a0, out_string   # load address of string to be printed into $a0
  syscall                  # call operating system to perform operation in $v0
                           #     syscall takes its arguments from $a0, $a1, ...
```

This illustrates the basic structure of an assembly language program.

- data segment and text segment
- use of label for data object (which is a zero-terminated ASCII string)
- use of registers
- invocation of a system call

MIPS Register Names

MIPS assemblers support standard symbolic names for the general-purpose registers:

`$zero`	stores value 0; cannot be modified
`$v0-1`	used for system calls and procedure return values
`$a0-3`	used for passing arguments to procedures
`$t0-9`	used for local storage; caller saves
`$s0-7`	used for local storage; procedure saves
`$sp`	stack pointer
`$fp`	frame pointer; primarily used during stack manipulations
`$ra`	used to store return address in procedure call
`$gp`	pointer to area storing global data (data segment)
`$at`	reserved for use by the assembler
`$k0-1`	reserved for use by OS kernel

MIPS Arithmetic Instructions

All arithmetic and logical instructions have 3 operands

Operand order is fixed (destination first):

```
<opcode>   <dest>, <leftop>, <rightop>
```

Example:

C code: `a = b + c;`

MIPS code: `add $s0, $s3, $s2`

"The natural number of operands for an operation like addition is three…requiring every instruction to have exactly three operands, no more and no less, conforms to the philosophy of keeping the hardware simple"

Basic MIPS Arithmetic Instructions

Here are the most basic arithmetic instructions:

```
add     $rd,$rs,$rt        Addition with overflow
                           GPR[rd] <-- GPR[rs] + GPR[rt]
div     $rs,$rt            Division with overflow
                           $lo <-- GPR[rs]/GPR[rt]
                           $hi <-- GPR[rs]%GPR[rt]
mul     $rd,$rs,$rt        Multiplication without overflow
                           GPR[rd] <-- (GPR[rs]*GPR[rt])[31:0]
sub     $rd,$rs,$rt        Subtraction with overflow
                           GPR[rd] <-- GPR[rs] - GPR[rt]
```

Instructions "with overflow" will generate an runtime exception if the computed result is too large to be stored correctly in 32 bits.

There are also versions of some of these that essentially ignore overflow, like `addu`.

Limitations and Trade-offs

Design Principle: simplicity favors regularity.

Of course this complicates some things...

C code:

```
a = b + c + d;
```

MIPS pseudo-code:

```
add $s0, $s1, $s2
add $s0, $s0, $s3
```

Operands must be registers (or immediates), only 32 registers are provided

Each register contains 32 bits

Design Principle: smaller is faster.

Why?

Immediates

In MIPS assembly, *immediates* are literal constants.

Many instructions allow immediates to be used as parameters.

```
addi    $t0, $t1, 42  # note the opcode
li      $t0, 42       # actually a pseudo-instruction
```

Note that immediates cannot be used with all MIPS assembly instructions; refer to your MIPS reference card.

Immediates may also be expressed in hexadecimal: `0x2A`

MIPS Logical Instructions

Logical instructions also have three operands and the same format as the arithmetic instructions:

```
<opcode>   <dest>, <leftop>, <rightop>
```

Examples:

```
and    $s0, $s1, $s2   # bitwise AND
andi   $s0, $s1, 42
or     $s0, $s1, $s2   # bitwise OR
ori    $s0, $s1, 42
nor    $s0, $s1, $s2   # bitwise NOR (i.e., NOT OR)
sll    $s0, $s1, 10    # logical shift left
srl    $s0, $s1, 10    # logical shift right
```

MIPS Load and Store Instructions

Transfer data between memory and registers

Example:

C code: `A[12] = h + A[8];`

MIPS code:

```
lw     $t0, 32($s3)   # $t0 <-- Mem[$s3+32]
add    $t0, $s2, $t0
sw     $t0, 48($s3)   # Mem[$s3+48] <-- $t0
```

Can refer to registers by name (e.g., `$s2`, `$t2`) instead of number

Load command specifies destination first: `opcode <dest>, <address>`

Store command specifies destination last: `opcode <dest>, <address>`

Remember arithmetic operands are registers or immediates, not memory!

Can't write: `add    48($s3), $s2, 32($s3)`

Addressing Modes

In *register* mode the address is simply the value in a register:

```
        lw    $t0, ($s3)
```

In *immediate* mode the address is simply an immediate value in the instruction:

```
        lw    $t0, 0    # almost always a bad idea
```

In *base + register* mode the address is the sum of an immediate and the value in a register:

```
        lw    $t0, 100($s3)
```

There are also various *label* modes:

```
        lw    $t0, absval
        lw    $t0, absval + 100
        lw    $t0, absval + 100($s3)
```

Unconditional Branch Instructions

MIPS unconditional branch instructions:

```
j     Label       # PC = Label
b     Label       # PC = Label
jr    $ra         # PC = $ra
```

These are useful for building loops and conditional control structures.

Conditional Branch Instructions

Decision making instructions

- alter the control flow,
- i.e., change the "next" instruction to be executed

MIPS conditional branch instructions:

```
bne   $t0, $t1, <label>  # branch on not-equal
                         # PC += 4 + Label if
                         #    $t0 != $t1
beq   $t0, $t1, <label>  # branch on equal
```

Labels are strings of alphanumeric characters, underscores and periods, not beginning with a digit. They are declared by placing them at the beginning of a line, followed by a colon character.

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

```
        bne     $s0, $s1, Miss
        add     $s3, $s0, $s1
Miss:   ....
```

Conditional Control Structure

```
if ( i < j )
   goto A;
else
   goto B;
```

```
# $s3 == i, $s4 == j
      slt   $t1, $s3, $s4
      beq   $zero, $t1, B
A:    # code...
      b     C
B:    # code...
C:
```