# Program: Hello, World!

.data  # data declaration section; specifies values to be stored
      # in memory and labels whereby the values are accessed
Greeting: .asciiz "\nHello, World!\n"

.text  # Start of code section
main:   # Execution begins at label "main"

li    $v0, 4  # system call code for printing string = 4
la    $a0, Greetings  # load address of string to be printed into $a0
syscall  # call operating system to perform operation;
         # $v0 specifies the system function called;
         # syscall takes $v0 (and opt arguments)

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:

- **$zero** stores value 0; should never be modified
- **$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; primarily used in procedure calls
- **$fp** frame pointer; primarily used during stack manipulations
- **$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; DO NOT USE
- **$k0-1** reserved for use by OS kernel; DO NOT USE
All arithmetic and logical instructions have 3 operands.

Operand order is fixed (destination first):

\[\text{<opcode>} \quad \text{<dest>}, \quad \text{<leftop>}, \quad \text{<rightop>}\]

Example:

C code: \[a = b + c;\]

MIPS code: \[\text{add} \quad \$s0, \quad \$s3, \quad \$s2\]

“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”
Here are the most basic arithmetic instructions:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>$rd,$rs,$rt</td>
<td>Addition with overflow</td>
</tr>
<tr>
<td></td>
<td></td>
<td>GPR[rd] &lt;- GPR[rs] + GPR[rt]</td>
</tr>
<tr>
<td>div</td>
<td>$rs,$rt</td>
<td>Division with overflow</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$lo &lt;- GPR[rs]/GPR[rt]*</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$hi &lt;- GPR[rs] % GPR[rt]</td>
</tr>
<tr>
<td>mul</td>
<td>$rd,$rs,$rt</td>
<td>Multiplication without overflow</td>
</tr>
<tr>
<td></td>
<td></td>
<td>GPR[rd] &lt;- (GPR[rs] * GPR[rt])[31:0]</td>
</tr>
<tr>
<td>sub</td>
<td>$rd,$rs,$rt</td>
<td>Subtraction with overflow</td>
</tr>
<tr>
<td></td>
<td></td>
<td>GPR[rd] &lt;- GPR[rs] - GPR[rt]</td>
</tr>
</tbody>
</table>

Instructions "with overflow" will generate a 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`.

* see `mfhi` and `mflo`
Limitations and Trade-offs

Design Principle: simplicity favors regularity.

Of course this complicates some things...

C code: \[ a = b + c + d; \]

MIPS pseudo-code: 
\[ \text{add} \ $s0, \ $s1, \ $s2 \]
\[ \text{add} \ $s0, \ $s0, \ $s3 \]

Operands must be registers (or immediates), only 29 user registers are provided
Each register contains 32 bits

Design Principle: smaller is faster.

Why?
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: \(0x2A\)
.text
main:

# set coefficients
li    $s0, 10          # c0 == 10
li    $s1,  7          # c1 ==  7
li    $s2,  5          # c2 ==  5
li    $s3, 3           # set x ==  3

# evaluate polynomial
add   $s4, $zero, $s2  # calculate c2
mul   $s4, $s4, $s3    #   c2 * x
add   $s4, $s4, $s1    #   c2 * x + c1
mul   $s4, $s4, $s3    # (c2 * x + c1) * x
add   $s4, $s4, $s0    # (c2 * x + c1) * x + c0
li    $v0, 10          # exit program
syscall
Logical instructions also have three operands and the same format as the arithmetic instructions:

\[
\text{<opcode>} \quad \text{<dest>}, \text{<leftop>}, \text{<rightop>}
\]

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:


MIPS code: 

\[
\text{lw} \quad \text{t0}, \ 32(\text{s3}) \quad \# \ \text{t0} \leftarrow \text{Mem}[\text{s3+32}] \\
\text{add} \quad \text{t0}, \ \text{s2}, \ \text{t0} \\\n\text{sw} \quad \text{t0}, \ 48(\text{s3}) \quad \# \ \text{Mem}[\text{s3+48}] \leftarrow \text{t0}
\]

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

Load command specifies destination first: \[ \text{opcode <dest>, <address>} \]
Store command specifies destination last: \[ \text{opcode <src>, <address>} \]

Remember arithmetic operands are registers or immediates, not memory!

Can’t write: \[ \text{add} \quad 48(\text{s3}), \ \text{s2}, \ 32(\text{s3}) \]}
Example

.data
  c2: .word 5      # coefficients for polynomial
  c1: .word 7
  c0: .word 10
  x:  .word 3      # x value for evaluation
  pofx: .word       # storage location for result

.text
main:

  # load coefficients and x from memory:
  lw $s0, c0            # c0 == 10
  lw $s1, c1            # c1 ==  7
  lw $s2, c2            # c2 ==  5
  li  $s3, x             # x ==  3

  # computations from previous example go here

  sw  $s4, pofx          # save fn value to memory
Additional Load and Store Instructions

There are also load and store instructions that act on data objects smaller than a word:

- `lh` load half-word from specified address
- `lb` load byte from specified address
- `sh` store half-word to specified address
- `sb` store byte to specified address

There are restrictions:

For word-oriented operations, the address must be word-aligned (i.e., a multiple of 4).

For half-word-oriented operations, the address must be half-word-aligned (i.e., a multiple of 2).

Violating the restrictions causes a runtime exception. (See unaligned variants.)
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)
```
MIPS conditional set instructions:

```
slt  $t0, $s0, $s1  # $t0 = 1 if $s0 < $s1
    # $t0 = 0 otherwise
slti $t0, $s0, <imm>  # $t0 = 1 if $s0 < imm
    # $t0 = 0 otherwise
```

These are useful for "remembering" the results of a Boolean comparison for later use.
Unconditional Branch Instructions

MIPS unconditional branch instructions:

\[
\begin{align*}
\text{j} & \quad \text{Label} \quad \# \text{ PC } = \text{Label} \\
\text{b} & \quad \text{Label} \quad \# \text{ PC } = \text{Label} \\
\text{jr} & \quad \$ra \quad \# \text{ PC } = \$ra
\end{align*}
\]

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 alphanumerical 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.
Example

```
# computes N + (N - 1) + ... + 2 + 1
.data
N:    .word 100                # specify limit for summation
Sum:  .word 0                 # space to hold computed sum

.text
main:
    lw $s0, N              # set loop counter to N
    li $s1, 0              # set sum to 0
loop:
    add $s1, $s1, $s0       # update running total
    addi $s0, $s0, -1      # decrement loop counter
    bne $s0, $zero, loop   # repeat until counter reaches 0
    sw $s1, Sum            # save sum to memory
    li $v0, 10             # exit program
    syscall
```