MIPS Hello World

```
# Hello, World!
      .data     ## Data declaration section
## String to be printed:
out_string:    .asciiz   "\nHello, World!\n"

      .text     ## Assembly language instructions go in text segment
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
                             #   specified in $v0
                             # syscall takes its arguments from $a0, $a1, ...

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

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; calling procedure saves these
`$s0-7`	used for local storage; called procedure saves these

And for the reserved registers:

`$sp`	stack pointer
`$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)    # use value in $s3 as address
```

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)   # address is $s3 + 100
```

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:
```

for Loop Example

```c
int Sum = 0;
for (int i = 1; i <= N; ++i) {
   Sum = Sum + i;
}
```

```
# $s0 == Sum, $s1 == N, $t0 == i
      move  $s0, $zero      # register assignment
      lw    $s1, N          # assume global symbol
      li    $t0, 1          # literal assignment
loop: beq   $t0, $s1, done  # loop test
      add   $s0, $s0, $t0   # Sum = Sum + i
      addi  $t0, $t0, 1     # ++i
      b     loop            # restart loop
done:
```

Policy of Use Conventions

MIPS programmers are expected to conform to the following conventions when using the 29 available 32-bit registers:

Name	Register number	Usage
`$zero`	0	the constant value 0
`$v0-$v1`	2-3	values for results and expression evaluation
`$a0-$a3`	4-7	arguments
`$t0-$t7`	8-15	temporaries
`$s0-$s7`	16-23	saved
`$t8-$t9`	24-25	more temporaries
`$gp`	28	global pointer
`$sp`	29	stack pointer
`$fp`	30	frame pointer
`$ra`	31	return address

Register 1 (`$at`) is reserved for the assembler, 26-27 (`$k0`, `$k1`) for operating system.

Registers 28-31 (`$gp`, `$sp`, `$fp`, `$ra`) are reserved for special uses, not user variables.

Pseudo-Instructions

You may have noticed something is odd about a number of the MIPS instructions that have been covered so far. For example:

```
li    $t0, 0xFFFFFFFF
```

Now, logically there's nothing wrong with wanting to place a 32-bit value into one of the registers.

But there's certainly no way the instruction above could be translated into a 32-bit machine instruction, since the immediate value alone would require 32 bits.

This is an example of a *pseudo-instruction*. A MIPS assembler, or SPIM, may be designed to support such extensions that make it easier to write complex programs.

In effect, the assembler supports an *extended MIPS architecture* that is more sophisticated than the actual MIPS architecture of the underlying hardware.

Of course, the assembler must be able to translate every pseudo-instruction into a sequence of valid MIPS assembly instructions.

Data

Basic fact: at the machine language level there are no explicit data types, only contents of memory locations. The concept of type is present only implicitly in how data is used.

declaration: reserving space in memory, or deciding that a certain data item will reside in a certain register.

Directives are used to reserve or initialize memory:

```
.data                   # mark beginning of a data segment
.asciiz   "a string"    # declare and initialize a string
.byte     13, 14, -3    # store values in successive bytes
.space    16            # alloc 16 bytes of space
.word     13, 14, -3    # store values in successive words
```

A complete listing of MIPS/MARS directives can be found in the MARS help feature.

Arrays

First step is to reserve sufficient space for the array.

Array elements are accessed via their addresses in memory, which is convenient if you've given the `.space` directive a suitable label.

```
        .data
list:   .word   2, 3, 5, 7, 11, 13, 17, 19, 23, 29
size:   .word   10
. . .
        la      $t1, list     # get array address
        li      $t2, 0        # set loop counter
print_loop:
        beq     $t2, $t3, print_loop_end  # check for array end

        lw      $a0, ($t1)    # print value at the array pointer
        li      $v0, 1
        syscall

        addi    $t2, $t2, 1   # advance loop counter
        addi    $t1, $t1, 4   # advance array pointer
        j       print_loop    # repeat the loop
print_loop_end:
```

Array Example

This is part of the palindrome example from the course website:

```
        .data
string_space: .space 1024
...
# prior to the loop, $t1 is set to the address of the first
# char in string_space, and $t2 is set to the last one
test_loop:
    bge     $t1, $t2, is_palin    # if lower pointer >= upper
                                      #  pointer, yes

    lb      $t3, ($t1)            # grab the char at lower ptr
    lb      $t4, ($t2)            # grab the char at upper ptr
    bne     $t3, $t4, not_palin   # if different, it's not

    addu    $t1, $t1, 1           # advance lower ptr
    subu    $t2, $t2, 1           # advance upper ptr
    j       test_loop             # repeat the loop
...
```

Procedure Support

From previous study of high-level languages, we know the basic issues:

- declaration: header, body, local variables
- call and return
- parameters of various types, with or without type checking, and a return value
- nesting and recursion

At the machine language level, there is generally little if any explicit support for procedures. This is especially true for RISC architectures.

There are, however, many conventions at the assembly language level.

Procedure Call and Return

Calling a procedure requires transferring execution to a different part of the code… in other words, a branch or jump operation:

```
      jal   <address>   # $ra = PC + 4
                        # PC = <address>
```

MIPS reserves register `$31`, aka `$ra`, to store the return address.

The called procedure must place the return value (if any) somewhere from which the caller can retrieve it. The convention is that registers `$v0` and `$v1` can be used to hold the return value. We will discuss what to do if the return value exceeds 4 bytes later…

Returning from the procedure requires transferring execution to the return address the `jal` instruction placed in `$ra`:

```
      jr   $ra          # PC = $ra
```

Passing Parameters

In most cases, passing parameters is straightforward, following the MIPS convention:

```
$a0      # 1st parameter
$a1      # 2nd parameter
$a2      # 3rd parameter
$a3      # 4th parameter
```

The called procedure can then access the parameters by following the same convention.

What if a parameter needs to be passed by reference? Simply place the address of the relevant data object in the appropriate register, and design the called procedure to treat that register value accordingly.

What if a parameter is smaller than a word? Clever register manipulation in the callee.

What if there are more than four parameters? We'll discuss that later…

Procedure Example 1

Let's implement a MIPS procedure to get a single integer input value from the user and return it:

```
get_integer:
# Prompt the user to enter an integer value.  Read and return
# it.  It takes no parameters.

        li $v0, 4         # system call code for printing a
                          #    string = 4
        la $a0, prompt    # address of string is argument 0 to
                          #    print_string
        syscall           # call operating system to perform
                          #   print operation

        li $v0, 5         # get ready to read in integers
        syscall           # system waits for input, puts the
                          #   value in $v0

        jr $ra
```

Since this doesn't use any registers that it needs to save, there's no involvement with the run-time stack.

Procedure Call Example 1

Since the procedure does not take any parameters, the call is simple. The return value will, by convention, have been placed in `$v0`.

```
. . .
       .data   # Data declaration section
       prompt: .asciiz "Enter an integer value\n"

       .text

main:                             # Start of code section
       jal     get_integer        # Call procedure
       move    $s0, $v0           # Put returned value in "save" reg
. . .
```

Procedure Example 2

Let's design a procedure to take some integer parameters and compute a value from them and return that value. Say the expression to be computed is:

$$(a + b) - (c + d)$$

Then the caller needs to pass four arguments to the procedure; the default argument registers are sufficient for this.

The procedure only returns a single one-word value, so the default register is enough.

The procedure will to use at least two registers to store temporary values while computing the expression, and a third register to hold the final result.

The procedure will use `$t0` and `$t1` for the temporaries, and `$s0` for the result.

(This could be done with fewer registers, but it's more illustrative this way.)

Procedure Example 2

This one's a little more interesting:

```
proc_example:
        addi    $sp, $sp, -4    # adjust stack pointer to make
                                #   room for 1 item
        sw      $s0, 0($sp)     # save the value that was in
                                #   $s0 when the call occurred

        add     $t0, $a0, $a1   # $t0 = g + h
        add     $t1, $a2, $a3   # $t1 = i + j
        sub     $s0, $t0, $t1   # $s0 = (g + h) - (i + j)

        move    $v0, $s0        # put return value into $v0

        lw      $s0, 0($sp)     # restore value of $s0
        addi    $sp, $sp, 4     # restore the stack pointer

        jr      $ra             # jump back to the return
                                #   address
```

Procedure Call Example 2

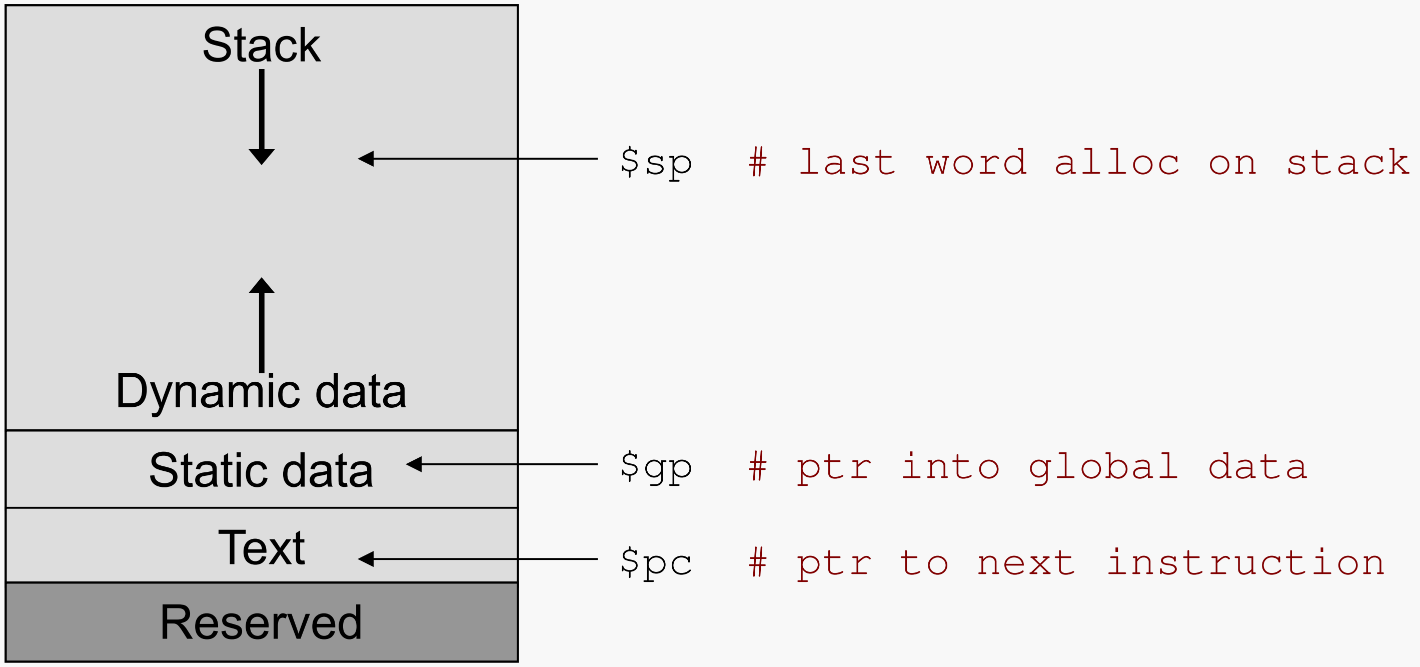
This time we must place the parameters (presumably obtained by calling the procedure `get_integer`), into the default argument registers.

```
. . .
        move    $a0, $s0        # position the parameters
        move    $a1, $s1
        move    $a2, $s2
        move    $a3, $s3
        jal     proc_example    # make the call

        move    $a0, $v0        # return value will be in $v0
        li      $v0, 1          # system call code for print_int
        syscall                 # print it
. . .
```

MIPS Memory Organization

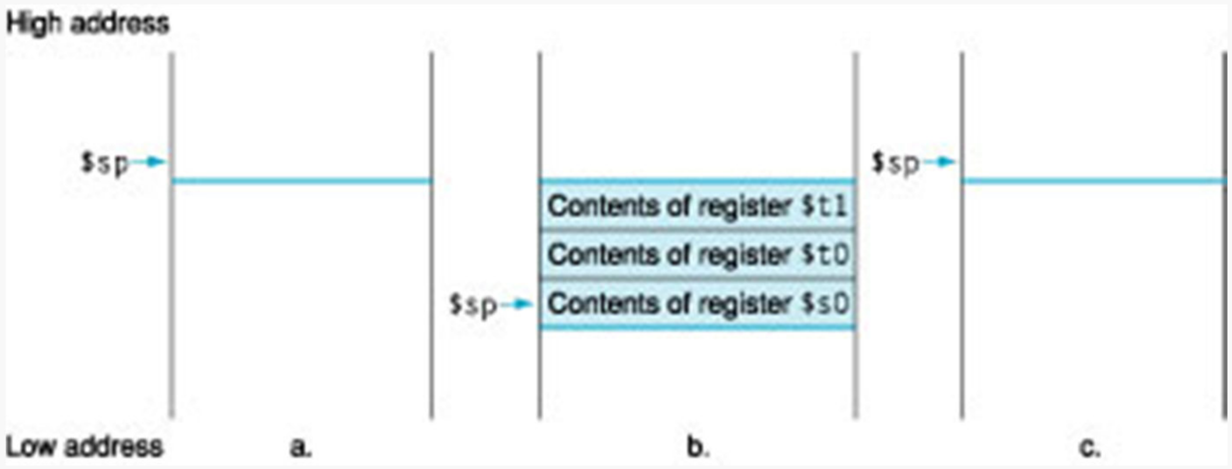
In addition to memory for static data and the program text (machine code), MIPS provides space for the run-time stack (data local to procedures, etc.) and for dynamically-allocated data:


Stack
Dynamic data
Static data
Text
Reserved
$sp # last word alloc on stack
$gp # ptr into global data
$pc # ptr to next instruction

Dynamic data is accessed via pointers held by the program being executed, with addresses returned by the memory allocator in the underlying operating system.

The System Stack

MIPS provides a special register, `$sp`, which holds the address of the most recently allocated word on a stack that user programs can employ to hold various values:



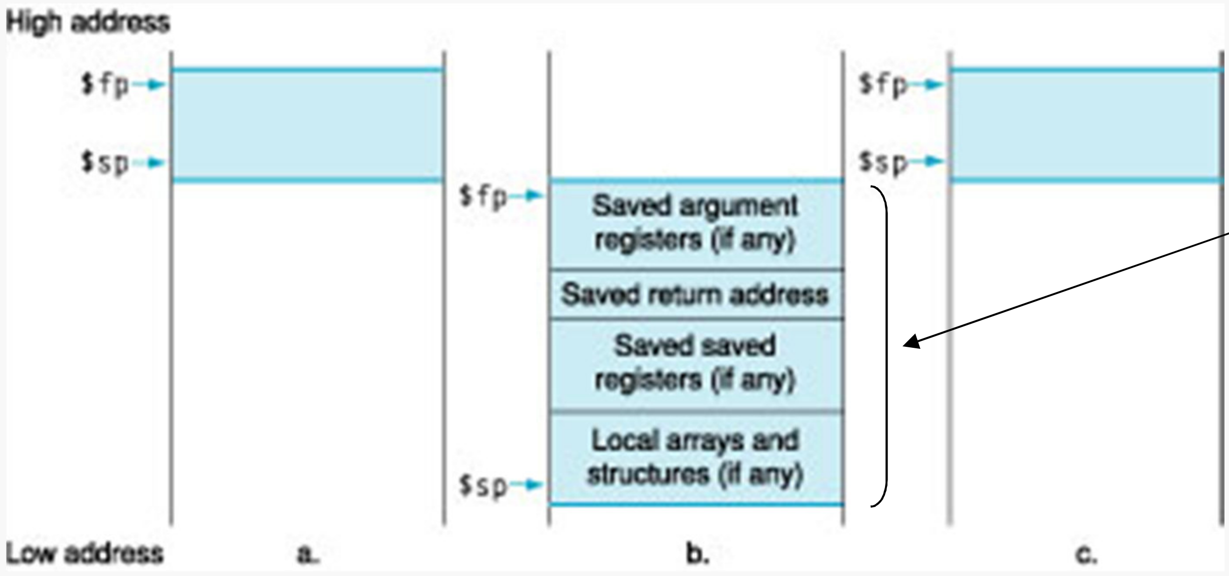
Note that the *run-time stack* is "upside-down". That is, `$sp`, decreases when a value is added to the stack and increases when a value is removed.

So, you decrement the stack pointer by 4 when pushing a new value onto the stack and increment it by 4 when popping a value off of the stack.

Using the System Stack

MIPS programs use the runtime stack to hold:

- "extra" parameters to be passed to a called procedure
- register values that need to be preserved during the execution of a called procedure and restored after the return
- saved procedure return address, if necessary
- local arrays and structures, if any


High address
$fp
$sp
$fp
Saved argument registers (if any)
Saved return address
Saved saved registers (if any)
Local arrays and structures (if any)
$sp
$fp
$sp
Low address
a.
b.
c.

activation record or *stack frame* for called procedure

System Stack Conventions

By convention, the caller will use:

- registers `$s0` - `$s7` for values it expects to be preserved across any procedure calls it makes
- registers `$t0` - `$t9` for values it does not expect to be preserved

It is the responsibility of the called procedure to make sure that if it uses any of the registers `$s0` - `$s7` it backs them up on the system stack first, and restores them before returning.

Obviously, the called procedure also takes responsibility to:

- allocate any needed space on the stack for local data
- place the return value onto the stack

In some situations, it is useful for the caller to also maintain the value that `$sp` held when the call was made, called the *frame pointer*. The register `$fp` would be used for this purpose.