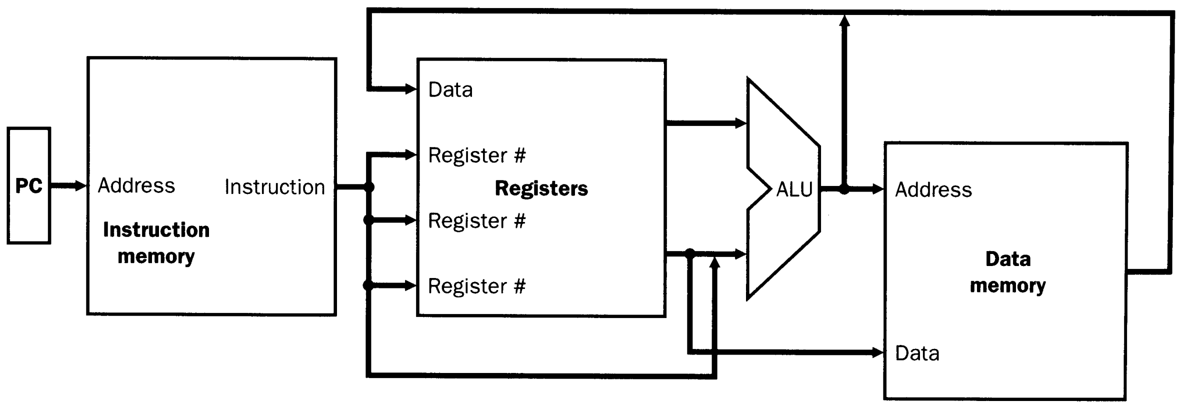
# MODULE 4

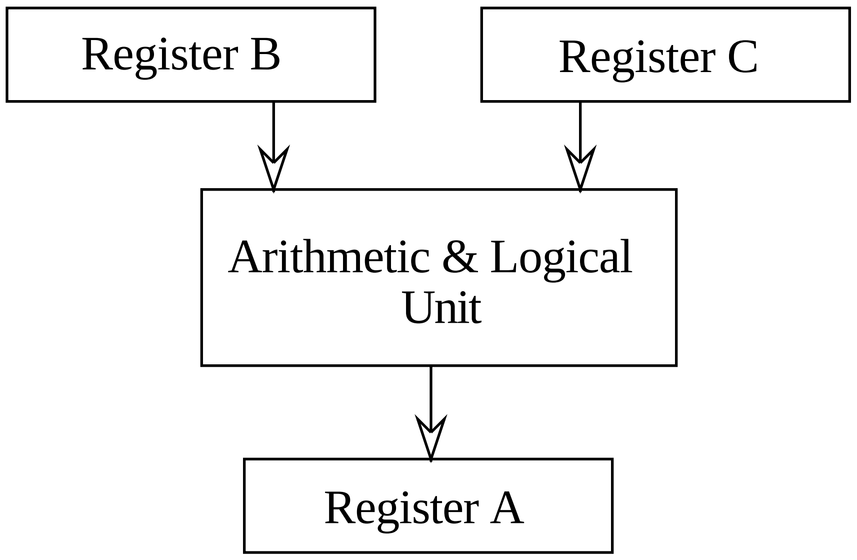
# INSTRUCTIONS: LANGUAGE OF THE MACHINE

## ARCHITECTURE MODEL



The basic instruction set of a computer is comprised of sequences of REGISTER TRANSFERS.

Example: Add A, B, C # A <-- B + C



This is the point at which we begin our investigation.

## OPERATIONS

The most common operations are arithmetic instructions. The MIPS instructions to add two numbers have the form:

```
    add     a, b, c         # a = b + c
    add     a, a, d         # a = a + d
    add     a, a, e         # a = a + e
```

where a, b, c, d, e are where variables are stored. It takes three instructions to add four variables.

A segment of C code:

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

may be translated into:

```
    add     a, b, c         # a = b + c
    sub     d, a, e         # d = a - e
```

This one:

```
f = (g + h) - (i + j);
```

into:

```
    add     t0, g, h        # temp. t0 = g + h
    add     t1, i, j        # temp. t1 = i + j
    sub     f, t0, t1       # f = t0 - t1
```

## REGISTER OPERANDS

The CPU has a limited number of locations called *registers* to store variables (made of SRAM). The MIPS CPU has 32 registers noted $0, ..., $31 which each hold 32 bits, or*word* s of data (4 bytes).

A computer has a much greater storage called the main memory, or just memory (made of DRAM), but which is slower.

The compiler associates the variables of a program to registers, attempting to store the most commonly used in so-called register variables.

```
f = (g + h) - (i + j);
```

Will be compiled into:

```
    add     $8,  $17,  $18     # temp. $8 = g + h
    add     $9,  $19,  $20     # temp. $9 = i + j
    sub     $16,  $8,  $9      # f = $8 - $9
```

where variables are assigned to registers by the compiler.

## MEMORY OPERANDS

Data structures such as arrays are stored in the memory since only a few elements can fit in the registers at any moment in time. To access a word in the memory, the CPU supplies an *address*. The memory is really a large single dimensional array with addresses starting at 0 and up to capacity.

Take the C statement:

```
A[i] = h + A[i];
```

The variable I is called an index (a selector).

The sequence of machine instructions (in assembly code) could be:

```
lw   $8,  Astart($19)        # load A[i] into $8
add $8,  $18,  $8            # add h to $8
sw   $8,  Astart($19)        # store back into A[i]
```

The address of the data in memory is calculated as the sum of Astart (address of first element of array) with the content of register $19 which holds the index i.

**Note on addressing:** The memory is structured as an array of bytes (numbered from 0 to $2^{32}$). Since we load and store words, the word addresses differ by 4. This method of addressing has an effect on the index i. It is represented in register $19 as i x 4.

## SO FAR...

The MIPS computer can be summarized as follows:

STORAGE:

32 registers, \$0, ..., \$31, Fast locations for word data
$2^{30}$ memory words, numbered 0, 4, ..., $2^{30}$-1.

INSTRUCTIONS

| assembly code | meaning | type |
|---|---|---|
| add \$1, \$2, \$3 | \$1=\$2+\$3 | operands: 3 registers |
| sub \$1, \$2, \$3 | \$1=\$2-\$3 | operands: 3 registers |
| lw \$1, 100(\$2) | \$1=Mem[\$2+100] | op’nds: 1 reg., 1 mem. |
| sw \$1, 100(\$2) | Mem[\$2+100]=\$1 | op’nds: 1 reg., 1 mem. |

We have seen only four instructions and two types of operands.

The MIPS CPU includes other types of operands, for example, there is provision for transferring *half-words* (16 bits) of data as well as byte in a single instruction.

The MIPS CPU also includes other operations.

For now, we will live with this simplified view.

## REPRESENTING INSTRUCTIONS

Numbers and machine instructions are both respresented using binary numbers. There is no way to tell them apart other than where they are placed in memory.

For example, a set of 32 bits represents the instruction add according to the following *format*:

| | | | | | |
|---|---|---|---|---|---|
| 0 | 17 | 18 | 8 | 0 | 32 |

In binary:

| | | | | | |
|---|---|---|---|---|---|
| 000000 | 10001 | 10010 | 01000 | 00000 | 100000 |

To make it simpler to discuss, we assign these *fields* symbolic names:

| 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits |
|---|---|---|---|---|---|
| op | rs | rt | rd | shamt | funct |

The meaning of these fields is conventional.

| | |
|---|---|
| • op: | operation of instruction |
| • rs: | the first register source operand |
| • rt: | the second register source operand |
| • rd: | the register destination operand (result) |
| • shamt: | shift amount (see later) |
| • funct: | function, variant of operation specified in op |

## MIPS INSTRUCTION FORMATS

The load and store instructions require a different format from instructions operating on registers.

| 6 bits | 5 bits | 5 bits | 16 bits |
|---|---|---|---|
| op | rs | rt | address |

So now, we can complete the translation from a high level language statement, to assembly code, to machine code:

```
A[i] = h + A[i];
```

Assembly code:

```
lw  $8,  Astart($19)    # load A[i] into $8
add $8,  $18,  $8       # add h to $8
sw  $8,  Astart($19)    # store back into A[i]
```

Machine code:

| | | | | | |
|---|---|---|---|---|---|
| 35 | 19 | 8 | 1200 | | |
| 0 | 18 | 8 | 8 | 0 | 32 |
| 43 | 19 | 8 | 1200 | | |

In binary:

| | | | | | |
|---|---|---|---|---|---|
| 100011 | 10011 | 01000 | 0000 0100 1011 0000 | | |
| 000000 | 10010 | 01000 | 01000 | 0000 | 100000 |
| 101011 | 10011 | 01000 | 0000 0100 1011 0000 | | |

## ALTERING CONTROL FLOW

The control flow of a program is altered by the use of branch and jump instructions.  Let's consider branches first.

```
beq        $1, $2, Label
```

This instruction compares the values stored in a pair of registers, and *depending* on equality branches to a location in memory. This location is specified by Label.

```
bne        $1, $2, Label
```

"Branch if not equal" branches if the two values are different.

Consider the following sequence of "C" code:

```
if (i == j)
     f = g - h;
else
     f = 0;
```

In assembly code:

```
          bne       $19, $20, Else
          sub       $16, $17, $18
          j         Exit
Else:     add       $16, $0, $0
Exit:
```

## ALTERING CONTROL FLOW cont.

```
        bne     $19, $20, Else
        sub     $16, $17, $18
        j       Exit
Else:   add     $16, $0, $0
Exit:
```

bne compares i and j stored in $19 and $20. If i ≠ j, it branches to Else:

If i = j, the add gets executed and g - h is calculated. Assuming these numbers are in $17 and $18, the result is left in $16.

Following this operation, is a jump instruction j. It diverts unconditionally the *flow of control* to the label Exit:.

Final point, at label Else:, the register $0 is special. It is hardwired to the value 0. The net effect of add is to copy the value 0 to register $16.

*Psuedo-instructions* are constructs defined by the assembler for purposes of clarity. For example:

```
move    $8, $18
```

is translated by the assembler into

```
add     $8, $0, $18
```

Thus, move $16, $0 in the program segment would correspond to the add statement above.

## LOOPS

Consider the following snippet of “C” code:

```
while (save[i] == k)
        i = i + j;
```

We now have all that is needed for translation into assembly code.

```
Loop:      mul        $9, $19, $10
           lw         $8, SaveAddr($9)
           bne        $8, $21, Exit
           add        $19, $19, $20
           j          Loop
```

For this code to work we have to assume that:

- We have a multiplication instruction (more on that later)
- i is stored in $19
- j is stored in $20
- k is stored in $21
- $10 has the value 4

Later, we will see that this is somewhat awkward, but for now it serves the purpose.

## slt **INSTRUCTION**

So far, we have instructions:

- add, sub (three register operands),
- lw, sw (one register operand, one memory op'nd),
- bne, beq (two register operands, one branch address).
- j (one branch address)

One technique to provide for statements like if (i < j) is to introduce one more instruction, "set lower than":

```
slt     $1,  $16,  $17
```

Destination register $1 is set to value 1 if the value in $16 is strictly smaller than that in $17 and set to 0 otherwise.

Pairing slt and bne (or beq) allows the compiler to generate all six comparison cases (== , != , <= , >= , < , >). For example:

```
if (i < j)
    a = 0;
```

will be translated into:

```
slt     $1,  $16,  $17
bne     $1, $0, Exit
```

The other cases should be worked out as an exercise.

## ONE MORE: jr

A jump instruction with the jump address specified in a register, “jump register”.

```
jr    $1          # Jump can span entire address space.
```

It is useful for jumping at addresses:
- which result from a calculation (jump table)
- which were previously stored

We now have:

| | | Format |
|---|---|---|
| • add, sub | (three register operands) | **R** |
| • lw, sw | (one reg. operand, one mem. op’nd) | **I** |
| • bne, beq | (two reg. operands, one branch addr.) | **I** |
| • slt | (three register operands) | **R** |
| • j | (one branch address) | **J** |
| • jr | (one register operand) | **R** |

These all fit into just three formats

| | 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits |
|---|---|---|---|---|---|---|
| **R** | op | rs | rt | rd | shamt | funct |

| | 6 bits | 5 bits | 5 bits | 16 bits |
|---|---|---|---|---|
| **I** | op | rs | rt | address |

| | 6 bits | 26 bits |
|---|---|---|
| **J** | op | address |

## PROCEDURES

Procedures (subroutines) allow structuring of programs by “calling” a code sequence, passing “parameters” (and “returning” values):

```
main{}
{
        ...;
        swap(a, 100);
        ...;
}

swap(int k[], int k)
{       int temp;

        temp = v[k];
        v[k] = v[k+1];
        v[k+1] = temp;
}
```

We need an instruction to save the return address. The “jump-and-link” instruction is like the “jump” instruction:

```
jal   ProcedureAddress
```

but it saves the return address, *the next in sequence*, in register $31. The “Program Counter” or PC, (more later), always holds this address. Its content gets copied into $31.

```
        ...
        jal    swap        # call swap
   ---->...

        swap:...           # enter swap
             ...
        jr     $31         # return from swap
```

## IMMEDIATE OPERANDS

To deal with statements of the form

```
a = 4;      b = c + 1;       ++i;
```

we require a mechanism for incorporating constants as part of the instruction. This is referred to as *immediate* addressing.

```
addi $2, $0, 4              # a = 4
addi $3, $4, 1              # b = c + 1
addi $29, $29, 4            # ++I (assuming I pointer)

slti  $8, $18, 10           # $8 = 1 if $18 < 10
```

Most instructions can incorporate immediate addressing as part of the source operand, but not the destination (why?).

To fit within the I format, constants are limited to 16 bits.

| | 6 bits | 5 bits | 5 bits | 16 bits |
|---|---|---|---|---|
| I | op | rs | rt | address |

Larger constants must be handled in 2 passes, e.g.
load $8 with the constant 0x0007A120:

```
lui   $8, 0x7              # upper half of $8 gets 0111
addi $8, $0, 0xA120        # lower gets 1010 0001 0010 0000
                          # (n.b. lui clears lower 16 bits)
```

## STACKS

A stack is one of the most important data structures in computer engineering. Unlike an array, where access to items is arbitrary, a stack stores and retrieves data in a given order: *last-in-first-out*.

- An item is said to be “pushed” onto a stack (placed on top) or “popped” off the stack (removed from top).

- Stacks are maintained using a “stack pointer”, i.e. an address kept at a fixed location (e.g. register $29). Assume that the items to be stacked are words.

```
                ... here the value of $1 is 10
        addi  $29, $29, -4
<-------sw    $1, 0($29)                  # push $1
|
|       ... any code sequence changing $1
|               ... here the value of $1 is 11
|       addi  $29, $29, -4
|   <---sw    $1, 0($29)                  # push $1
|   |
|   |   ... any code sequence changing $1
|   |
|   --->lw    $1, 0($29)                  # pop $1
|       addi  $29, $29, 4
|               ... here the value of $1 is back to 11
|
|       ... any code sequence changing $1
|
------->lw    $1, 0($29)                  # pop $1
        addi  $29, $29, 4
                ... here the value of $1 is back to 10
                ...stack pointer is also back to its original value
```

## STACKS cont.

As just seen, stacks allow the values of storage locations (most often registers) to be saved and restored any number of times, as long as the sequence of pushed and pops is symmetrical.

A procedure may call another procedure. A stack allows the succession of return addresses in ($31) to be kept in an orderly fashion. The most common method is to insert pairs of push/pop operations around the procedure call.

Example:

```
                …
1000            jal   foo1
1004            …

foo1: …
                …
2000            addi  $29, $29,- 4
2004            sw    $31, 0($29)
2008            jal   foo2
2012            lw    $31, 0($29)
2016            addi  $29,$29, 4
                …
2050            jr    $31

foo2: …
                …
3000            jr    $31
```

This can be done at any level of nesting.

Initial state of stack $29 = 9000

| | |
|---|---|
| 9000 | |

In body of first procedure, link register $31 PUSHED onto stack. Stack Pointer $29 = 8996

| | |
|---|---|
| 9000 | |
| 8999 | EC |
| 8998 | 03 |
| 8997 | 00 |
| 8996 | 00 |

Before returning, the stack is POPPED by incrementing the stack pointer.

Each procedure must ensure that the stack pointer is correctly restored before returning.

Stack Pointer $29 = 9000

| | |
|---|---|
| 9000 | |

## STACKS cont.

Other uses for the stack:

### 1. Parameter passing

In passing parameters, the first few first registers ($4 to $7) are conventionally used for this. If the capacity of four registers is not sufficient, the arguments are "pushed" on the stack before calling the procedure.

### 2. Saving registers across calls

In the course of a computation, registers contain the values of variables. These values might be lost when calling a procedure using the same registers. They need to be preserved: the stack is used for that.

### 3. Spill registers

When the compiler runs out of registers, it can either:
- allocate temporary variables on the stack
- or push the values in registers on the stack to make room, restoring them later.

This process is known as *register allocation*. How well it is done is often the hallmark of the quality of a compiler.

### 4. Provide for temporary storage

```
foo()
{       array[1000];            /* dynamic variables go on the stack */
        …
                                          }
```

## ADDRESSING IN BRANCHES AND JUMPS

Jump instruction:

```
        j       10000
```

is assembled into

|   | 6 bits | 26 bits |
|---|---|---|
| **J** | 2 | 10000 |

The 26 bit number used to represent the jump address is indeed very large (67,108,863, 64M). It is sufficient for most programs. However

```
bne         $1, $0, Exit
```

Is assembled into

|   | 6 bits | 5 bits | 5 bits | 16 bits |
|---|---|---|---|---|
| **I** | 5 | 1 | 0 | Exit |

The 16 bit number (65535, 64K) is clearly too small for today's standards.

Branch instructions use the *PC-relative* addressing mode. The program counter always contains the address of the next instruction in sequence. The idea is to add the 16 bit address to the value of the PC to get the target address. This way, a branch can "reach" within $2^{16}$ addresses relative to itself. This is efficient because the greatest majority of branches occur around if's and loops which span a small amount of code.

## SUMMARY OF ADDRESSING MODES

So far we have:

- *Register addressing*: the operand is a register (formats R and I)

- *Base addressing* (or displacement addressing): the operand is a memory location whose address is the sum of a register and an address (an offset) in the instruction.
  (I format)

- *Immediate addressing*: the operand is a constant within the instruction itself.
  (I format)

- *PC-relative addressing*: the branch target address is the sum of the PC plus an address (an offset) in the instruction.
  (I format)

- *Absolute addressing*: the jump target address is found within the instruction itself.
  (J format)

## A COMPLETE EXAMPLE

```
int  v[10000];

sort  (int  v[],  int  n)
{
        int i, j;

        for (i = 0; i < n; i = i + 1) {
                for (j = i - 1; j >= 0 && v [ j ] > v[ j + 1]; j = j -1) {
                        swap(v,  j);
                }
        }
}


swap  (int  v[],  int  k)
{
        int    temp;

        temp  =  v[k];
        v[k]  =  v[k+1];
        v[k+1]  =  temp;
}
```

## Sorting Example

```
#-------------------------------------------------------------
#       Procedure Name:         swap(int v[], int k)
#
#       Description:            Exch. the contents of v[k]
#                               and v[k+1]
#
#       Register Allocation:    $4:     pointer to v[0]
#                               $5:     k
#                               $2:     base register for
#                                       array accesses
#                               $15:    scratch
#                               $16:    scratch
#-------------------------------------------------------------

        .text

#-------------------------------------------------------------
#       Save context of caller
#-------------------------------------------------------------

swap:   addi    $29, $29, -12   # Allocate space on stack
        sw      $2, 0($29)      # Save $2 onto stack
        sw      $15, 4($29)     # Save $15 onto stack
        sw      $16, 8($29)     # Save $16 onto stack

#-------------------------------------------------------------
#       Main procedure body
#-------------------------------------------------------------

        sll     $2, $5, 2       # Turn index k into array
                                # offset
        add     $2, $4, $2      # + off. to base. $2 points
                                # to v[k].
        lw      $15, 0($2)      # temp1 ($15) = v[k]
        lw      $16, 4($2)      # temp2 ($16) = v[k+1]
        sw      $16, 0($2)      # v[k] <-- v[k+1]
        sw      $15, 4($2)      # v[k+1] <-- v[k]

#-------------------------------------------------------------
#       Restore the context of the caller
#-------------------------------------------------------------

        lw      $2, 0($29)      # Restore $2
        lw      $15, 4($29)     # Restore $15
        lw      $16, 8($29)     # Restore $16
        addi    $29, $29, 12    # Restore the stack pointer
```

## Sorting Example: cont.

```
#-------------------------------------------------------------
#       Execute return
#-------------------------------------------------------------

        jr      $31             # Return to calling routine


#-------------------------------------------------------------
#       Procedure Name:         sort(int v[], int n)
#
#       Description:            Sorts the contents of array
#                               v[] in ascending order using
#                               bubblesort (highly
#                               inefficient!).
#
#       Register Allocation:    $4:        pointer to v[0]
#                               $5:        n
#                               $17:       loop index j
#                               $19:       loop index i
#                               $21-$23:   scratch registers
#                               $31:       linkage register
#-------------------------------------------------------------

#-------------------------------------------------------------
#       Save context of caller
#
#       n.b.    See note below regarding saving $5 on
#               function call.
#-------------------------------------------------------------

sort:   addi    $29, $29, -24   # Space on stack for 6 reg.
        sw      $17, 0($29)     # Save $17 - j loop index
        sw      $19, 4($29)     # Save $19 - i loop index
        sw      $21, 8($29)     # Save $21 - scratch
        sw      $22, 12($29)    # Save $22 - scratch
        sw      $23, 16($29)    # Save $23 - scratch
        sw      $31, 20($29)    # Save $31 - linkage register

#-------------------------------------------------------------
#       Main procedure body
#       Set up outer loop:  for (i=0; i<n; i++)
#-------------------------------------------------------------

        li      $19, 0          # $19 <-- loop index i, i=0;
for1tst:slt     $21, $19, $5    # i<n ?
        beq     $21, $0, exitol # No, take exit if outer loop
                                # complete (i>=n)
```

## Sorting Example: cont.

```
#-------------------------------------------------------------
#       Inner loop:  for (j=i-1; j>=0 && v[j]>v[j+1]; j--)
#-------------------------------------------------------------

        addi    $17, $19, -1    # $17 <-- loop index j, j=i-1
for2tst:slti    $21, $17, 0     # j<0 ?
        bne     $21, $0, exitil # Yes, exit inner loop
        sll     $21, $17, 2     # Turn j into array offset
        add     $21, $4, $21    # $21 <-- pointer to v[j]
        lw      $22, 0($21)     # $22 <-- v[j]
        lw      $23, 4($21)     # $23 <-- v[j+1]
        slt     $21, $23, $22   # v[j+1] < v[j] ?
        beq     $21, $0, exitil # No, exit inner loop

#-------------------------------------------------------------
#       Procedure call swap[v,j];
#
#    n.b. $5 is overwritten on function call.  Rather than
#         save on stack (time consuming), we save it in a
#         temporary register and restore it immediately
#         afterwards.
#-------------------------------------------------------------

        move    $21, $5         # Need to change $5 for
                                # function call
        move    $5, $17         # $4 <-- ptr v[j]; $5 <-- j
        jal     swap
        move    $5, $21         # Restore $5.

#-------------------------------------------------------------
#       Bottom of inner loop
#-------------------------------------------------------------

        addi    $17, $17, -1    # Decrement loop counter j
        b       for2tst         # Back to top of loop

#-------------------------------------------------------------
#       Bottom of outer loop
#-------------------------------------------------------------

exitil: addi    $19, $19, 1     # Increment loop counter i
        b       for1tst         # Back to top of loop
```

## Sorting Example: cont.

```
#-------------------------------------------------------------
#       Restore context of the caller
#-------------------------------------------------------------

exitol: lw      $17, 0($29)     # Restore $17
        lw      $19, 4($29)     # Restore $19
        lw      $21, 8($29)     # Restore $21
        lw      $22, 12($29)    # Restore $22
        lw      $23, 16($29)    # Restore $23
        lw      $31, 20($29)    # Restore linkage register
        addi    $29, $29, 24    # Restore stack pointer

#-------------------------------------------------------------
#       Execute return
#-------------------------------------------------------------

        jr      $31             # Return to calling routine



#-------------------------------------------------------------
# Test Program
#
# Test the sorting program on an array of 10 numbers.  Start
# off by printing out the list (demonstration of SPIM's
# built-in system calls), sort it, and print out the sorted
# result.
#-------------------------------------------------------------

        .globl  main

#-------------------------------------------------------------
# Start off by printing a short banner and the unsorted list.
#-------------------------------------------------------------

main:   la      $16, TstArray   # $16 <-- array to be sorted
        li      $18, 10         # $17 <-- loop counter for
                                # print
        la      $19, String1    # $19 <-- string to be
                                # printed
        move    $4, $19         # Point to the string
        li      $2, 4           # Code for print string
        syscall                 # Header for unsorted array
```

## Sorting Example: cont.

```
#------------------------------------------------------------
# Printing loop - use the syscall (1) function to print array
# elements.
#------------------------------------------------------------

main10: lw      $4, 0($16)      # Get current array element
        li      $2, 1           # Code for print integer
        syscall                 # Execute call
        la      $4, crlf        # String for carriage return
                                # + line feed
        li      $2, 4           # Code for print string
        syscall                 # Execute call
        addi    $16, $16, 4     # Point to the next array
                                # element
        addi    $18, $18, -1    # Decrement loop counter
        bne     $18, $0, main10 # Loop until array printed

#------------------------------------------------------------
#       Skip a line between unsorted and sorted output
#------------------------------------------------------------

        la      $4, crlf        # String for carriage return
                                # + line feed
        li      $2, 4           # Code for print string
        syscall                 # Execute call

#------------------------------------------------------------
# Next we use the sorting routine to put the array in order.
#------------------------------------------------------------

        la      $4, TstArray    # Set up call to sort
        li      $5, 10          # $4 <-- v[0]; $5 <-- size
        jal     sort

#------------------------------------------------------------
# I really should have set up the print code as a function,
# but I'll be lazy and simply cut-and-past the code.
#------------------------------------------------------------

        la      $16, TstArray   # $16 <-- array to be sorted
        li      $18, 10         # $17 <-- loop counter for
                                # print
        la      $19, String2    # $19 <-- string to be
                                # printed
        move    $4, $19         # Point to the string
        li      $2, 4           # Code for print string
        syscall                 # Header for sorted array
```

## Sorting Example: cont.

```
main20: lw      $4, 0($16)      # Get current array element
        li      $2, 1           # Code for print integer
        syscall                 # Execute call
        la      $4, crlf        # String for carriage return
                                # + line feed
        li      $2, 4           # Code for print string
        syscall                 # Execute call
        addi    $16, $16, 4     # Point to the next array
                                # element
        addi    $18, $18, -1    # Decrement loop counter
        bne     $18, $0, main20 # Loop until array printed

#-------------------------------------------------------------
#       Finally we exit by doing the appropriate syscall.
#-------------------------------------------------------------

        li      $2, 10          # Get exit code
        syscall                 # And we're out of here

#-------------------------------------------------------------
#       All the data goes here (data segment)
#-------------------------------------------------------------

        .data

String1: .asciiz "Unsorted array:\n\n"

String2: .asciiz "Sorted array:\n\n"

crlf:    .asciiz "\n"

         .align 2

TstArray: .word 5 299 4 -36 1101 2 25 8000 21 99
```

## Passing Parameters on the Stack

foo (int a, int b, int c);

Could translate into the following assembly code:

```
#------------------------------------------
#     Assume that a, b, c are in $15, $16, $17 respectively
#------------------------------------------

      addi  $29, $29, -12         # Allocate space on stack
      sw    $17, 0($29)           # Convention is to store args
      sw    $16, 4($29)           # in reverse order.
      sw    $15, 8($29)
      jal   foo
      addi  $29, $29, 12          # Must restore $sp!
      etc...
```

Inside foo arguments could be accessed as follows:

```
#------------------------------------------
#     $30 = $fp is used as a frame pointer
#------------------------------------------

foo:  addi  $29, $29, -NN         # Allocate NN bytes for saving
      sw    $15, 0($29)           # registers used by foo.
      etc...

      addi  $fp, $29, NN          # Set $fp to start of args
      lw    $15, 8($fp)           # $15 <-- argument a
      lw    $16, 4($fp)           # $16 <-- argument b
      lw    $17, 0($fp)           # $17 <-- argument c
      etc...
```

## ARRAYS VERSUS POINTERS

```
clear1(int  array[],  int  size)
{
        int  i;
        for (i = 0; i < size; i = i + 1)
                array[i] = 0;
}

        move $2,  $0           # i = 0
Loop:   sll   $14, $2, 2       # $14 = I * 4  (muli $14, $2, 4)
        add   $3, $4, $14      # $3 = &array[i]
        sw    $0, 0($3)        # array[i]= 0
        addi  $2, $2, 1        # i = i +1
        slt   $1, $2, $5       # $1 = (i < size)
        bne   $1, $0, Loop     # if () goto Loop


clear2(int  *array,  int  size)
{
        int  *p;
        for(p = &array[0]; p < &array[size]; p = p + 1)
                *p = 0;
}

        move $2, $4            # p = &array[0]
        sll   $14, $5, 2       # $14 = size * 4 (muli $14, $5, 4)
        add   $3, $4, $14      # $3 = &array[size]
Loop:   sw    $0, 0($2)        # Memory[p] = 0
        addi  $2, $2, 4        # p = p + 4
        slt   $1, $2, $3       # $1 = p < &array[size]
        bne   $1, $0, Loop     # if () goto Loop
```

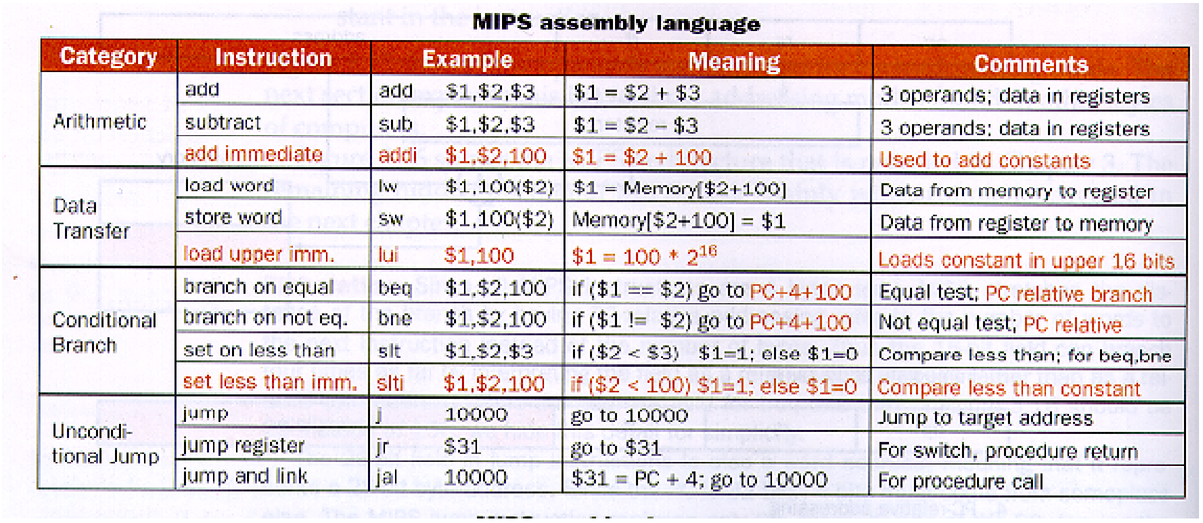
The pointer version is more efficient (from 6 down to 4 instructions). Lesson: identity of concept between indices and pointers. Modern compilers take advantage of this.

## SUMMARY

All constructs found in high level languages:

- expressions with variables and constants;
- control statements: if’s, loops;
- procedures;
- data structures (arrays, stacks, pointers);

can be translated using a small set of machine instructions which fit in just three formats.

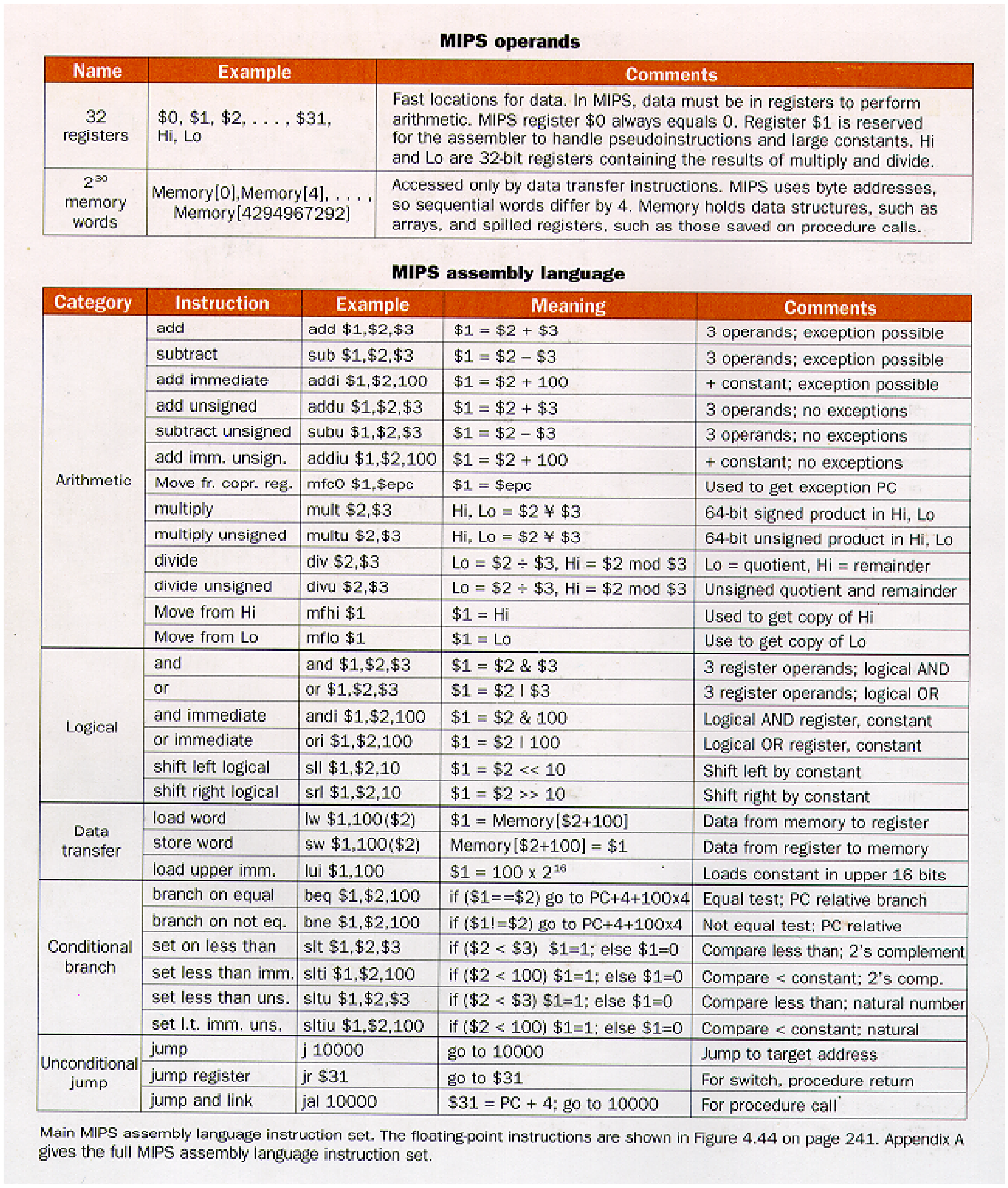


**MIPS assembly language**

| Category | Instruction | Example | Meaning | Comments |
|---|---|---|---|---|
| Arithmetic | add | add $1,$2,$3 | $1 = $2 + $3 | 3 operands; data in registers |
| | subtract | sub $1,$2,$3 | $1 = $2 – $3 | 3 operands; data in registers |
| | add immediate | addi $1,$2,100 | $1 = $2 + 100 | Used to add constants |
| Data Transfer | load word | lw $1,100($2) | $1 = Memory[$2+100] | Data from memory to register |
| | store word | sw $1,100($2) | Memory[$2+100] = $1 | Data from register to memory |
| | load upper imm. | lui $1,100 | $1 = 100 * 2^16 | Loads constant in upper 16 bits |
| Conditional Branch | branch on equal | beq $1,$2,100 | if ($1 == $2) go to PC+4+100 | Equal test; PC relative branch |
| | branch on not eq. | bne $1,$2,100 | if ($1 != $2) go to PC+4+100 | Not equal test; PC relative |
| | set on less than | slt $1,$2,$3 | if ($2 < $3) $1=1; else $1=0 | Compare less than; for beq,bne |
| | set less than imm. | slti $1,$2,100 | if ($2 < 100) $1=1; else $1=0 | Compare less than constant |
| Unconditional Jump | jump | j 10000 | go to 10000 | Jump to target address |
| | jump register | jr $31 | go to $31 | For switch, procedure return |
| | jump and link | jal 10000 | $31 = PC + 4; go to 10000 | For procedure call |

This achievement did not come about overnight, but is the result of 4 decades of technical evolution.  Most modern RISC CPU’s have an instruction set which resembles that of MIPS.  Once this one is understood, the others can be understood by differences (M88000, SPARC, ALPHA, I860, RS/6000, HPSpectrum, PowerPC). Some other have only part of the features of RISC style (Pentium).

# MIPS Assembly Language (Short Form)



**MIPS operands**

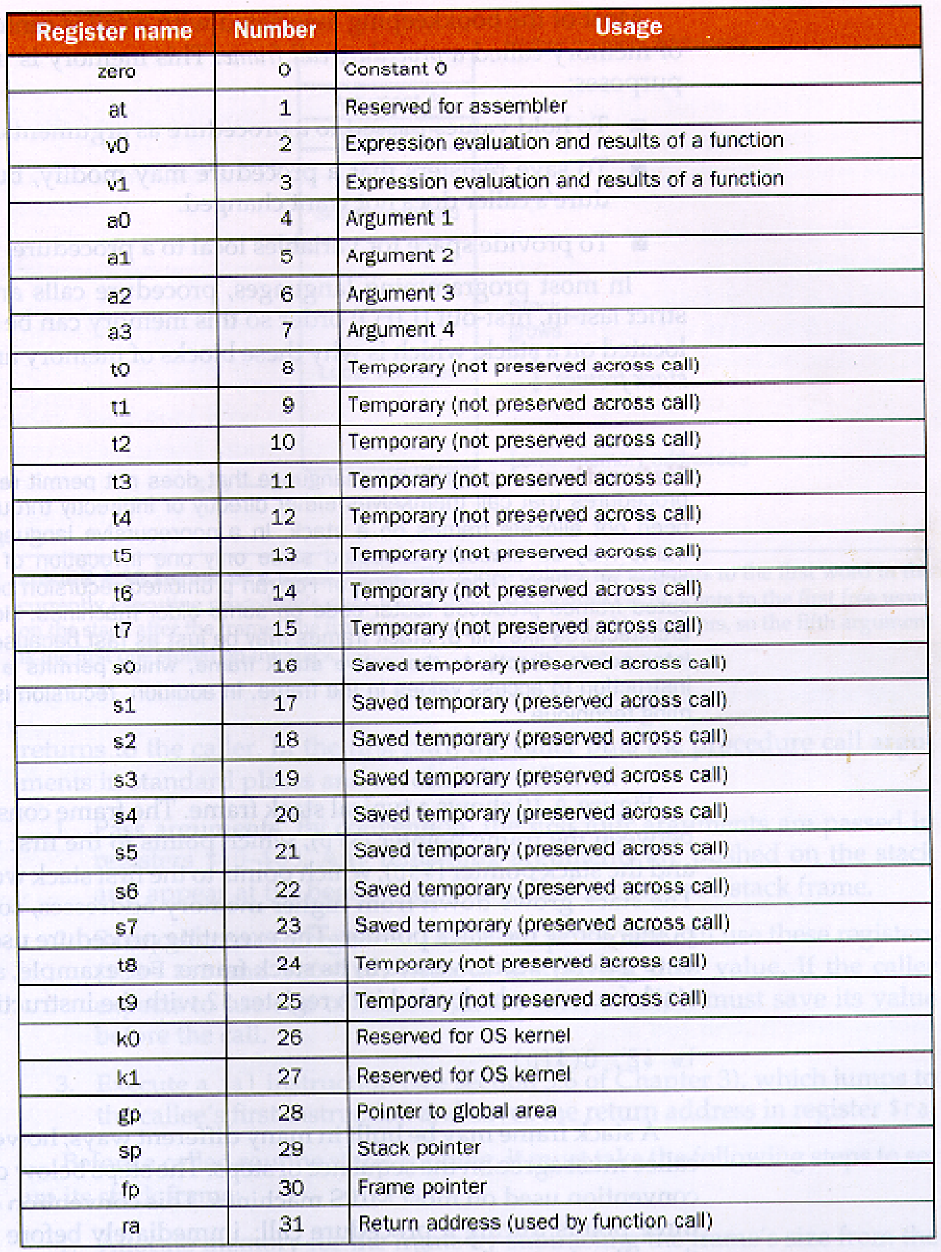
| Name | Example | Comments |
|---|---|---|
| 32 registers | $0, $1, $2, . . . , $31, Hi, Lo | Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register $0 always equals 0. Register $1 is reserved for the assembler to handle pseudoinstructions and large constants. Hi and Lo are 32-bit registers containing the results of multiply and divide. |
| $2^{30}$ memory words | Memory[0],Memory[4], . . . , Memory[4294967292] | Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls. |

**MIPS assembly language**

| Category | Instruction | Example | Meaning | Comments |
|---|---|---|---|---|
| Arithmetic | add | add $1,$2,$3 | $1 = $2 + $3 | 3 operands; exception possible |
| | subtract | sub $1,$2,$3 | $1 = $2 – $3 | 3 operands; exception possible |
| | add immediate | addi $1,$2,100 | $1 = $2 + 100 | + constant; exception possible |
| | add unsigned | addu $1,$2,$3 | $1 = $2 + $3 | 3 operands; no exceptions |
| | subtract unsigned | subu $1,$2,$3 | $1 = $2 – $3 | 3 operands; no exceptions |
| | add imm. unsign. | addiu $1,$2,100 | $1 = $2 + 100 | + constant; no exceptions |
| | Move fr. copr. reg. | mfc0 $1,$epc | $1 = $epc | Used to get exception PC |
| | multiply | mult $2,$3 | Hi, Lo = $2 ¥ $3 | 64-bit signed product in Hi, Lo |
| | multiply unsigned | multu $2,$3 | Hi, Lo = $2 ¥ $3 | 64-bit unsigned product in Hi, Lo |
| | divide | div $2,$3 | Lo = $2 ÷ $3, Hi = $2 mod $3 | Lo = quotient, Hi = remainder |
| | divide unsigned | divu $2,$3 | Lo = $2 ÷ $3, Hi = $2 mod $3 | Unsigned quotient and remainder |
| | Move from Hi | mfhi $1 | $1 = Hi | Used to get copy of Hi |
| | Move from Lo | mflo $1 | $1 = Lo | Use to get copy of Lo |
| Logical | and | and $1,$2,$3 | $1 = $2 & $3 | 3 register operands; logical AND |
| | or | or $1,$2,$3 | $1 = $2 \| $3 | 3 register operands; logical OR |
| | and immediate | andi $1,$2,100 | $1 = $2 & 100 | Logical AND register, constant |
| | or immediate | ori $1,$2,100 | $1 = $2 \| 100 | Logical OR register, constant |
| | shift left logical | sll $1,$2,10 | $1 = $2 << 10 | Shift left by constant |
| | shift right logical | srl $1,$2,10 | $1 = $2 >> 10 | Shift right by constant |
| Data transfer | load word | lw $1,100($2) | $1 = Memory[$2+100] | Data from memory to register |
| | store word | sw $1,100($2) | Memory[$2+100] = $1 | Data from register to memory |
| | load upper imm. | lui $1,100 | $1 = 100 x $2^{16}$ | Loads constant in upper 16 bits |
| Conditional branch | branch on equal | beq $1,$2,100 | if ($1==$2) go to PC+4+100x4 | Equal test; PC relative branch |
| | branch on not eq. | bne $1,$2,100 | if ($1!=$2) go to PC+4+100x4 | Not equal test; PC relative |
| | set on less than | slt $1,$2,$3 | if ($2 < $3) $1=1; else $1=0 | Compare less than; 2's complement |
| | set less than imm. | slti $1,$2,100 | if ($2 < 100) $1=1; else $1=0 | Compare < constant; 2's comp. |
| | set less than uns. | sltu $1,$2,$3 | if ($2 < $3) $1=1; else $1=0 | Compare less than; natural number |
| | set l.t. imm. uns. | sltiu $1,$2,100 | if ($2 < 100) $1=1; else $1=0 | Compare < constant; natural |
| Unconditional jump | jump | j 10000 | go to 10000 | Jump to target address |
| | jump register | jr $31 | go to $31 | For switch, procedure return |
| | jump and link | jal 10000 | $31 = PC + 4; go to 10000 | For procedure call |

Main MIPS assembly language instruction set. The floating-point instructions are shown in Figure 4.44 on page 241. Appendix A gives the full MIPS assembly language instruction set.

# MIPS Register Allocation Convention



| Register name | Number | Usage |
|---|---|---|
| zero | 0 | Constant 0 |
| at | 1 | Reserved for assembler |
| v0 | 2 | Expression evaluation and results of a function |
| v1 | 3 | Expression evaluation and results of a function |
| a0 | 4 | Argument 1 |
| a1 | 5 | Argument 2 |
| a2 | 6 | Argument 3 |
| a3 | 7 | Argument 4 |
| t0 | 8 | Temporary (not preserved across call) |
| t1 | 9 | Temporary (not preserved across call) |
| t2 | 10 | Temporary (not preserved across call) |
| t3 | 11 | Temporary (not preserved across call) |
| t4 | 12 | Temporary (not preserved across call) |
| t5 | 13 | Temporary (not preserved across call) |
| t6 | 14 | Temporary (not preserved across call) |
| t7 | 15 | Temporary (not preserved across call) |
| s0 | 16 | Saved temporary (preserved across call) |
| s1 | 17 | Saved temporary (preserved across call) |
| s2 | 18 | Saved temporary (preserved across call) |
| s3 | 19 | Saved temporary (preserved across call) |
| s4 | 20 | Saved temporary (preserved across call) |
| s5 | 21 | Saved temporary (preserved across call) |
| s6 | 22 | Saved temporary (preserved across call) |
| s7 | 23 | Saved temporary (preserved across call) |
| t8 | 24 | Temporary (not preserved across call) |
| t9 | 25 | Temporary (not preserved across call) |
| k0 | 26 | Reserved for OS kernel |
| k1 | 27 | Reserved for OS kernel |
| gp | 28 | Pointer to global area |
| sp | 29 | Stack pointer |
| fp | 30 | Frame pointer |
| ra | 31 | Return address (used by function call) |

**FIGURE A.9 MIPS registers and usage convention.**