How to Design the Instructions?

- Operations
  - Arithmetic
  - Logical
  - => Datapath
- Operands
  - => Datapath
- Control flow
  - Decision control
    - branch
    - If-then-else
    - Loop
    - Case or switch
  - Procedures calls
    - Call and return
  - => Control

```c
int add5 (int a)
{
    int tmp = a + 5;
    return tmp;
}

void main ()
{
    int a = 7;
    int c;
    if (a == 7)
        c = add5(a);
}  
```
Decision Making Instructions (Section 2.6)

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

Branch Classifications

- Two basic types of branches
  - **Unconditional**: Always jump to the specified address
  - **Conditional**: Jump to the specified address if some condition is true; otherwise, continue with the next instruction

- **Destination** addresses can be specified in the same way as other operands (combination of registers, immediate constants, and memory locations), depending on what is supported in the ISA
Conditional Branch Instructions

beq register1, register2, L1  #branch equal
bne register1, register2, L1  #branch if not equal

Ex:

    if (i==j) goto L1;
    f = g+h;
L1:  f = f-i;

assume f, g, h, i, j, stored in $s0...$s4

beq  $s3, $s4, L1
add  $s0, $s1, $s2
L1: sub  $s0, $s1, $s3

Explicit address
Calculated by the assembler
Unconditional Branch Instructions and MIPS Control for if-then-else

• MIPS unconditional branch instructions:
  \( j \) \ label

• Example:

  \[
  \text{if } (i==j) \quad \textbf{bne} \quad \$s4, \quad \$s5, \quad \textbf{Else} \\
  f=g+h; \quad \textbf{add} \quad \$s3, \quad \$s4, \quad \$s5 \\
  \textbf{else} \quad \textbf{j} \quad \textbf{Lab2} \\
  f=g-h; \quad \textbf{Else:sub} \quad \$s3, \quad \$s4, \quad \$s5 \\
  \textbf{Exit:...}
  \]
Unconditional Branch Instructions and MIPS Control for if-then-else

- MIPS unconditional branch instructions:
  \[ j \ label \]

- Example:

  \[
  \begin{align*}
  \text{if } (i==j) & \quad \text{beq } $s4, $s5, \text{Lab1} \\
  f=g+h; & \quad \text{add } $s3, $s4, $s5 \\
  \text{else} & \quad j \text{ Lab2} \\
  h=i-j; & \quad \text{Lab1:sub } $s3, $s4, $s5 \\
  \text{Lab2:} & \ldots
  \end{align*}
  \]
set-on-less-than in MIPS

• We have: beq, bne, what about **Branch-if-less-than**?
• New instruction:

```assembly
if $s1 < $s2 then
    $t0 = 1
else
    $t0 = 0

slt $t0, $s1, $s2
```

• Can use this instruction with **beq/bne** to build "**blt** $s1, $s2, Label"
  – blt => slt + bne/beq
  – can now build general control structures
  – Q. why not “**blt**” in **MIPS**?
    • Simplicity

• Note that the assembler needs a register to do this,
  – there are **policy of use conventions for registers**
• **Constant operands** are popular in comparisons
  – $zero always has 0
  – Other value: immediate version, **slti**
    – `slti $t0, $s2, 10` # $t0 = 1 if $s2 < 10
MIPS approach for ==, !=, <, <=, >, >=

- Combine `slt`, `slti`, `beq`, `bne` and `$zero` to create all relative conditions
Observation on Branches

- Most conditional branches go a short and constant distance
- Fancy addressing modes not often used
- No use for auto-increment/decrement

- So in keeping with the RISC philosophy of simplicity, MIPS has only a few basic branch types.
INFO: Complete MIPS Branch Types

• Conditional branch:
  – beq/bne reg1, reg2, addr
  – If reg1 =/≠ reg2, jump to PC + addr (PC-relative)

• Register jump:
  – jr reg
  – Fetch address from specified register, and jump to it

• Unconditional branch:
  – j addr
  – Always jump to PC: addr (use “pseudodirect” addressing)
INFO: Branch Instructions Example

- **Conditional branches**
  - `beq R1, R2, L1` # if $R1 = R2$ go to $L1$
  - `bne R1, R2, L1` # if $R1 \neq R2$ go to $L1$
  - These are I-type instructions

- **Unconditional branches**
  - `jr R8`  # Jump based on register 8

- **Test if < 0**
  - `slt R1, R16, R17`  # $R1$ gets 1 if $R16 < R17$
  - `bne R1, 0, less`  # branch to less if $R1 =\not= 0$
Generating Branch Targets in MIPS

4 PC-relative addressing

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>Address</th>
</tr>
</thead>
</table>

PC + Memory

5 Pseudodirect addressing

<table>
<thead>
<tr>
<th>op</th>
<th>Address</th>
</tr>
</thead>
</table>

PC : Memory

(From Patterson and Hennessy, p. 152; COPYRIGHT 1988 MORGAN KAUFMANN PUBLISHERS, INC. ALL RIGHTS RESERVED)
Compiling Other Control Statements

• Loops:
  – for, while: test before loop body; jump past loop body if false
  – Do: test condition at end of loop body; jump to beginning if true

• switch: (called “case” statements in some other languages)
  – Build a table of addresses
  – Use jr (or equiv. In non-MIPS processor)
  – Be sure to check for default and unused cases!
Decision for Iterating a Computation: Loop

while (save[i] == k)
    i += 1;

Assume i, k use register $s3, $s5, base of array “save” is in $s6

# index $t1 = 4 * i
Loop: sll $t1, $s3, 2
# add index to base
add $t1, $t1, $s6
# load array value
lw $t0, 0($t1)
# test if save[i] == k
bne $t0, $s5, Exit
# i = i + 1
add $s3, $s3, 1
j Loop;

Exit:
Switch Compilation Example

Compile the following:

```
switch (k) {
    case 0: f = f + 1; break;
    case 1: f = f - 2; break;
    case 3: f = -f; break;
}
```

Note the gap (case 2);

(Assume k in r13)

```
#switch test
slti $14, $13, 0 # set r14 if r13 < 0
bne $14, $0, Exit # Go to Exit if k < 0
slti $14, $13, 4 # set r14 if k < 4
beq $14, $0, Exit # Go to Exit if k = 4
add $14, $13, $13 # r14 = 2*k
add $14, $14, $14 # r14 = 4*k
lw $14, 1000 ($14) # Base of table at 1000
jr $14 # Jump to the address table
```

#Switch body

```
L0: addi $8, $8, 1 # add 1 to r8 (f)
    j Exit # jump to Exit (break)
L1: subi $8, $8, 2 # subtract 2 from r8
    j Exit # Another break
L3: sub $8, $0, $8 # f = 0 - f
    j Exit # Another break
```

<table>
<thead>
<tr>
<th></th>
<th>address of L0</th>
<th>address of L1</th>
<th>address of Exit</th>
<th>address of L3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1004</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1008</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1012</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
INFO: Assembly Language vs. Machine Language

• Assembly provides convenient **symbolic representation**
  – much easier than writing down numbers
  – e.g., destination first

• Machine language is the underlying reality
  – e.g., destination is no longer first

• Assembly can provide 'pseudoinstructions'
  – e.g., “move $t0, $t1” exists only in Assembly
  – would be implemented using “add $t0,$t1,$zero”

• When considering performance you should **count real instructions**
### MIPS operands

<table>
<thead>
<tr>
<th>Name</th>
<th>Example</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>32 registers</td>
<td>$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra</td>
<td>Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register $zero always equals 0. $gp (28) is the global pointer, $sp(29) is the stack pointer, $fp (30) is the frame pointer, and $ra (31) is the return address.</td>
</tr>
<tr>
<td>$2^{30}$ memory words</td>
<td>Memory [0], Memory [4], Memory [42949672920]</td>
<td>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 register, such as those saved on procedure calls.</td>
</tr>
</tbody>
</table>

### MIPS assembly language

<table>
<thead>
<tr>
<th>Category</th>
<th>Instruction</th>
<th>Example</th>
<th>Meaning</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>add</td>
<td>add $s1, $s2, $s3</td>
<td>$s1 = $s2 + $s3</td>
<td>Three operands; data in registers</td>
</tr>
<tr>
<td></td>
<td>subtract</td>
<td>sub $s1, $s2, $s3</td>
<td>$s1 = $s2 - $s3</td>
<td>Three operands; data in registers</td>
</tr>
<tr>
<td>Data transfer</td>
<td>load word</td>
<td>lw $s1,100 ($s2)</td>
<td>$s1 = Memory [$s2 + 100]</td>
<td>Data from memory to register</td>
</tr>
<tr>
<td></td>
<td>store word</td>
<td>sw $s1,100 ($s2)</td>
<td>Memory [$s2 + 100] = $s1</td>
<td>Data from register to memory</td>
</tr>
<tr>
<td>Conditional branch</td>
<td>branch on equal</td>
<td>beq $s1, $s2, L</td>
<td>if ($s1 == $s2) go to L</td>
<td>Equal test and branch</td>
</tr>
<tr>
<td></td>
<td>branch on not equal</td>
<td>bne $s1, $s2, L</td>
<td>if ($s1 != $s2) go to L</td>
<td>Not equal test and branch</td>
</tr>
<tr>
<td></td>
<td>set on less than</td>
<td>slt $s1, $s2, $s3</td>
<td>if ($s2 &lt; $s3) $s1 = 1; else $s1 = 0</td>
<td>Compare less than: for beq, bne</td>
</tr>
<tr>
<td>Unconditional jump</td>
<td>jump</td>
<td>j 2500</td>
<td>go to 10000</td>
<td>Jump to target address</td>
</tr>
<tr>
<td></td>
<td>jump register</td>
<td>jr $ra</td>
<td>go to $ra</td>
<td>For switch, procedure return</td>
</tr>
<tr>
<td></td>
<td>jump and link</td>
<td>jal 2500</td>
<td>$ra = PC + 4; go to 1000</td>
<td>For procedure call</td>
</tr>
</tbody>
</table>
Procedures (Section 2.7)

• Six steps in the execution of a procedure
  – **Place parameters** in a place where the procedure can access them
  – **Transfer control** to the procedure
  – Acquire the storage resources needed for the procedure
  – Perform the desired task
  – **Place the result** value in a place where the calling program can access it
  – **Return control** to the point of origin
Function Calls in the MIPS

- Function calls an essential feature of programming languages
  - The program calls a function to perform some task
  - When the function is done, the CPU continues where it left off in the calling program

- But how do we know where we left off?
Calling a Function in the MIPS

• Use the *jal* (“jump and link”) instruction
• *jal addr* just *j addr* except
  – The “return address” (PC) + 4 placed in $ra (R31)
  – This is the address of the *next instruction* after the jal
  – Use jr $ra to return
Instructions Supporting Procedure Calls (2.7)

- **Parameter passing**
  - $a0 \sim $a3 are used for these
  - Q. what if parameters exceed four?
  - Spilling registers, place parameters in stack, $sp$ (R29)

- **Transfer control: Jump and link**
  - `jal` procedure address
  - note: return address is stored in $ra$ (R31)

- **Return value**
  - $v0 \sim v1$ for return values

  - Q. What if returns results exceed two?
  - Saving return address on stack
    - $sp$ (R29) is used as stack pointer

- **Return**
  - `jr` $ra$
int leaf_example (int g, int h, int i, int j)
{
    int f;
    f = (g+h) - (i+j);
    return f;
}
Assume g, h, i, j use $a0,..$a3, f uses $s0
Refer p. 11

leaf_example:

#push old values into stack to avoid damage
addi $sp, $sp, -12;
sw $t1, 8($sp)
sw $t0, 4($sp)
sw $s0, 0($sp)

#functional body
add $t0, $s1, $s2
add $t1, $s3, $s4
sub $s0, $t0, $t1

#return value, copy f to return registers
add $v0, $s0, $zero

#pop old values from stack
lw $s0, 0($sp)
lw $t0, 4($sp)
lw $t1, 8($sp)
addi $sp, $sp, 12
jr $ra
Improve the Example

• Problem in previous example
  – A lot of saving and restoring temporary registers

• How to avoid it in MIPS registers convention
  – Temporary registers, $t0..$t9
    • Value won’t be preserved in the procedure call
  – Saved registers, $s0..$s7
    • Value must be preserved
    • If used, these must be saved and stored
Difficulties with Function Calls

• This example works OK. But what if:
  – The function F calls another function?
  – The caller had something important in regs R6 and/or R7?
  – The called function calls itself, (nested procedure)?
    • Register conflict

• Solution
  – Each version of a function should have its own copies of variables
  – These are arranged in a stack, as a pile of frames.
Nested Procedures

• Problems:
  – Register conflicts

• Solutions:
  – Push all the other register that must be preserved onto the stack
  – Procedure
    • The caller pushed any argument register $a0-$a3 or temporary registers $t0..$t9 that are needed after the call
    • The callee push the return address $ra and any saved registers $s0..$s7 used by the callee
    • Stack push and store
Stack Examples for Nested Functional Calls

- Assume function A calls B, which calls C. Function C calls itself once:

<table>
<thead>
<tr>
<th>A’s vars</th>
<th>B’s vars</th>
<th>C’s vars</th>
</tr>
</thead>
<tbody>
<tr>
<td>start A</td>
<td>A calls B</td>
<td>B calls C</td>
</tr>
<tr>
<td></td>
<td>A’s vars</td>
<td>C’s vars</td>
</tr>
<tr>
<td></td>
<td></td>
<td>C’s vars</td>
</tr>
<tr>
<td></td>
<td></td>
<td>C’s vars</td>
</tr>
<tr>
<td></td>
<td></td>
<td>C’s vars</td>
</tr>
</tbody>
</table>
Examples for Nested Functional Calls

```c
int factorial (int n) {
    if (n < 1) return 1;
    else return (n * factorial (n - 1));
}
```
Parameter \( n \) => $a0

**factorial:**

*push old values into stack to avoid damage*

```assembly
addi $sp, $sp, -8;
sw $ra, 4($sp)
sw $a0, 0($sp)
```

*functional body*

```assembly
slti $t0, $a0, 1    # test if \( n < 1 \)
beq $t0, $zero, L1 # if \( n \geq 1 \), go to L1
```

*return 1*

```assembly
addi $v0, $zero, 1  # return 1
addi $sp, $sp, 8    #pop 2 items off stack
jr $ra
```
#another return
L1:
    addi $a0, $a0, -1  # N >= 1, new factorial (n-1)
    jal factorial
#pop values to restore
    lw $a0, 0($sp)
    lw $ra, 4($sp)
    addi $sp, $sp, 8
    mul $v0, $a0, $v0   #return n * factorial(n-1)
    jr $ra
INFO: Parameter Passing

• **Stack**  
  – Ideal data structure for spilling registers

• **Caller save**. The calling procedure (caller) is responsible for saving and restoring any registers that must be *preserved across the call*. The called procedure (callee) can then modify any register without constraint.

• **Callee save**. The callee is responsible for saving and restoring any registers that it might use. The caller uses registers without worrying about restoring them after a call.
Stack Frames

If a function needs more memory and/or may call others, it uses a stack frame, which holds:

- **Automatic** variables (non-static variables declared within function)
- **Arguments** to the function (just another type of local variable)
- The “return address” (since $ra overwritten by call)
- **Saved** registers from caller ($s0-$s7) if you need to use them
- “Spill” registers, including $t0-$t9 when calling others
## Layout of a Stack Frame

<table>
<thead>
<tr>
<th>$fp$ →</th>
<th>Argument 5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Argument 6</td>
</tr>
<tr>
<td></td>
<td>⋮</td>
</tr>
<tr>
<td></td>
<td>Saved registers</td>
</tr>
<tr>
<td></td>
<td>Local variables</td>
</tr>
<tr>
<td>$sp$ →</td>
<td>Higher memory addresses</td>
</tr>
</tbody>
</table>

Stack grows

Lower memory addresses
Allocating Space for New Data on the Stack
Details of Stack for Procedure Calls (1)

Procedure frame
the segment of stack containing a procedure’s
saved registers and local variables
Calling a Non-Leaf Function (Caller)
- Put arguments to the function in $a0-$a3
- Save contents of $t0-9 if they will be needed later
- If more than 4 args, push them onto stack
- jal (or jral) to beginning of the function code
Details of Stack for Procedure Calls (3)

Calling a Non-Leaf Function (Callee)

• Push current fp onto stack
• Move fp to top of frame (just below old sp)
• Set sp to (fp – frame size)
  – Frame size is the same for every call of the same function
  – Known at compile-time
• Use displacement addressing to get at local variables
  – Save $s0-$s7 (whichever you need to reuse) and $ra in frame
  – Save $a0-$a3 to frame if needed (e.g., calling another function)
Returning from Non-Leaf Function (Callee)

- Put return values (if any) in $v0 and $v1
- Restore $s0-$s7 (whichever were saved) and $ra from frame
- Restore sp to just above current fp
- Restore old fp from stack frame
- Jump to $ra (jr)

- Caller can get return args in $v0 and $v1, if any
Register Conventions in the MIPS

<table>
<thead>
<tr>
<th>Names</th>
<th>Regs</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>0</td>
<td>Constant 0</td>
</tr>
<tr>
<td>-</td>
<td>1</td>
<td>(Reserved for assembler)</td>
</tr>
<tr>
<td>$v0-$v1</td>
<td>2-3</td>
<td>Return values (NOT Preserved across the calls)</td>
</tr>
<tr>
<td>$a0-$a3</td>
<td>4-7</td>
<td>Args to functions (NOT Preserved across the calls)</td>
</tr>
<tr>
<td>$t0-$t9</td>
<td>8-15, 24-25</td>
<td>Temporaries (NOT Preserved across the calls)</td>
</tr>
<tr>
<td>$s0-$s7</td>
<td>16-23</td>
<td>Saved values (Preserved across the calls)</td>
</tr>
<tr>
<td>-</td>
<td>26-27</td>
<td>(Reserved for OS kernel)</td>
</tr>
<tr>
<td>$gp</td>
<td>28</td>
<td>Global pointer to global data</td>
</tr>
<tr>
<td>$sp</td>
<td>29</td>
<td>Stack pointer (Preserved across the calls)</td>
</tr>
<tr>
<td>$fp</td>
<td>30</td>
<td>Frame pointer (Preserved across the calls)</td>
</tr>
<tr>
<td>$ra</td>
<td>31</td>
<td>Return address (Preserved across the calls)</td>
</tr>
</tbody>
</table>
Other Storage: Global Variables

• In C/C++, "global variables" are
  – Variables declared outside of any functions
  – Static variables (inside or outside a function)
  – Static data members of a class (C++)

• Properties:
  – Only one copy of each (unlike automatic variables)
  – Initialization allowed (set value before main () starts)
  – All in one region of memory, accessed through $gp (r28)
Other Storage: Dynamic Storage (Heap)

• In C/C++, the “heap” contains
  – Blocks of memory allocated by malloc() etc.
  – Objects created using the new keyword (C++)
  – Properties:

• Stored in a big chunk of memory between globals and stack
  – Controlled by the programming language’s library (e.g., libc)
  – Can be grown if needed
  – No dedicated reg. Like $gp; everything goes through pointers
Typical Layout of Program

$sp \rightarrow 7fff \ ffff$

hex

$gp \rightarrow 1000 \ 8000$

hex

1000 8000

hex

cp \rightarrow 0040 \ 0000

hex

stack

Dynamic date

Static data

Text

Reserved
What an Executable Program Looks Like

• When you execute a program, it is in the form of an “executable”

• The executable contains everything you need to run your program
  – Every function used, starting with main() – the “text segment”
  – Values of all initialized global variables – the “data segment”
  – Information about uninitialized globals

• Every function and every global variable has an absolute address in memory
Executing an Executable

• When you execute a program, the loader:
  – Allocates space for your program (details vary by OS)
  – Copies the text and data segments of the executable to memory
  – Jumps to a known starting address (specified in the executable)

• Once the executable starts running at that starting address, it
  – Initializes regs such as $gp and $sp; initializes heap (if used)
  – Sets uninitialized globals to 0 (if the language requires this)
  – Sets up command line args into data structure (e.g., argc/argv)
  – Does jal to start of main () function
MIPS operands

<table>
<thead>
<tr>
<th>Name</th>
<th>Example</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>32 registers</td>
<td>$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra</td>
<td></td>
</tr>
<tr>
<td>2^{30} memory words</td>
<td>Memory [0], Memory [4],..., Memory [42949672920]</td>
<td>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.</td>
</tr>
</tbody>
</table>

MIPS assembly language

<table>
<thead>
<tr>
<th>Category</th>
<th>Instruction</th>
<th>Example</th>
<th>Meaning</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>add</td>
<td>add $s1, $s2, $s3</td>
<td>$s1 = $s2 + $s3</td>
<td>Three operands; data in registers</td>
</tr>
<tr>
<td></td>
<td>subtract</td>
<td>sub $s1, $s2, $s3</td>
<td>$s1 = $s2 - $s3</td>
<td>Three operands; data in registers</td>
</tr>
<tr>
<td>Data transfer</td>
<td>load word</td>
<td>lw $s1,100 ($s2)</td>
<td>$s1 = Memory [$s2 + 100]</td>
<td>Data from memory to register</td>
</tr>
<tr>
<td></td>
<td>store word</td>
<td>sw $s1,100 ($s2)</td>
<td>Memory [$s2 + 100] = $s1</td>
<td>Data from register to memory</td>
</tr>
<tr>
<td>Conditional branch</td>
<td>branch on equal</td>
<td>beq $s1, $s2, L</td>
<td>if ($s1 == $s2) go to L</td>
<td>Equal test and branch</td>
</tr>
<tr>
<td></td>
<td>branch on not equal</td>
<td>bne $s1, $s2, L</td>
<td>if ($s1 != $s2) go to L</td>
<td>Not equal test and branch</td>
</tr>
<tr>
<td></td>
<td>set on less than</td>
<td>slt $s1, $s2, $s3</td>
<td>if ($s2 &lt; $s3) $s1 = 1; else $s1 = 0</td>
<td>Compare less than: for beq, bne</td>
</tr>
<tr>
<td>Unconditional jump</td>
<td>jump</td>
<td>j 2500</td>
<td>go to 10000</td>
<td>jump to target address</td>
</tr>
<tr>
<td></td>
<td>jump register</td>
<td>jr $sra</td>
<td>go to $sra</td>
<td>For switch, procedure return</td>
</tr>
<tr>
<td></td>
<td>jump and link</td>
<td>jal 2500</td>
<td>$sra = PC + 4; go to 1000</td>
<td>For procedure call</td>
</tr>
</tbody>
</table>
INFO: **MIPS Registers**

- 32 regs with \( R0 = 0 \)
- Reserved registers: \( R1, R26, R27 \).
- Special usage:
  - \( R28 \): pointer to global area
  - \( R29 \): stack pointer
  - \( R30 \): frame pointer
  - \( R31 \): return address
INFO: Standard Register Conventions

- The 32 integer registers in the MIPS are “general-purpose” – any can be used as an operand or result of an arithmetic operation.

- But *making* different pieces of software work together is *easier* if certain conventions are followed concerning which registers are to be used for what purposes.

- These conventions are usually suggested by the vendor and supported by the compilers.
### INFO: MIPS Registers and Usage Convention

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

Register 1 ($at) reserved for assembler, 26-27 for operating system
## INFO: MIPS Registers and Usage Convention

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