We’ll be working with the MIPS instruction set architecture (ISA)
- similar to other architectures developed since the 1980's
- almost 100 million MIPS processors manufactured in 2002
- used by NEC, Nintendo, Cisco, Silicon Graphics, Sony, …

**MIPS Hardware**

**General Data Registers**
- 32 general 32-bit registers
- conventions govern the use of the general registers

**Floating-Point Co-processor**
- 32 single-precision or float 32-bit registers dedicated to floating-point operations
- FP registers may also be used as 64-bit (double-precision)
- conventions govern the use of the general registers

**System registers**
- registers used by the hardware/operating system rather than directly by user programs
- program counter (PC) stores the address of the next instruction to be executed
- other registers used for memory access operations, etc.
MIPS General Registers

<table>
<thead>
<tr>
<th>Number</th>
<th>Name</th>
<th>Conventional Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$zero</td>
<td>always stores constant value 0</td>
</tr>
<tr>
<td>1</td>
<td>$at</td>
<td>temporary storage used by assembler</td>
</tr>
<tr>
<td>2-3</td>
<td>$v0, $v1</td>
<td>function results, expression evaluation</td>
</tr>
<tr>
<td>4-7</td>
<td>$a0 - $a3</td>
<td>arguments passed to functions</td>
</tr>
<tr>
<td>8-15</td>
<td>$t0 - $t7</td>
<td>temporary storage for local results</td>
</tr>
<tr>
<td>24-25</td>
<td>$s8, $s9</td>
<td>auto-saved storage for local results</td>
</tr>
<tr>
<td>16-23</td>
<td>$s0 - $s7</td>
<td>auto-saved storage for local results</td>
</tr>
<tr>
<td>26-27</td>
<td>$k0, $k1</td>
<td>reserved for OS kernel use</td>
</tr>
<tr>
<td>28</td>
<td>$gp</td>
<td>address of global data region</td>
</tr>
<tr>
<td>29</td>
<td>$sp</td>
<td>address of run-time stack region</td>
</tr>
<tr>
<td>30</td>
<td>$fp</td>
<td>address of frame on run-time stack</td>
</tr>
<tr>
<td>31</td>
<td>$ra</td>
<td>return address</td>
</tr>
</tbody>
</table>

General Memory Organization

Viewed as a large, single-dimension array, with an address.

A memory address is an index into the array.

"Byte addressing" means that the index points to a byte of memory.
MIPS Memory Organization

Bytes are nice, but most data items use larger "words"

For MIPS, a word is 32 bits or 4 bytes.

<table>
<thead>
<tr>
<th>0</th>
<th>4</th>
<th>8</th>
<th>12</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>32 bits of data</td>
<td>32 bits of data</td>
<td>32 bits of data</td>
<td>32 bits of data</td>
<td>Registers hold 32 bits of data</td>
</tr>
</tbody>
</table>

$2^{32}$ bytes with byte addresses from 0 to $2^{32} - 1$

$2^{30}$ words with byte addresses 0, 4, 8, ... $2^{32} - 4$

Words are aligned, that is, each has an address that is a multiple of 4.

Endian Concepts

In computing, endianness refers to the order in which small units (bytes) are used to form larger units (words).

Consider storing the integer value $0x4A3B2C1D$ in memory:

<table>
<thead>
<tr>
<th>1000</th>
<th>1001</th>
<th>1002</th>
<th>1003</th>
</tr>
</thead>
<tbody>
<tr>
<td>4A</td>
<td>3B</td>
<td>2C</td>
<td>1D</td>
</tr>
<tr>
<td>1000</td>
<td>1001</td>
<td>1002</td>
<td>1003</td>
</tr>
<tr>
<td>4A</td>
<td>3B</td>
<td>2C</td>
<td>1D</td>
</tr>
</tbody>
</table>

MSB stored at the memory location with the lowest address

Little-endian

LSB stored at the memory location with the lowest address

Big-endian

There doesn’t seem to be any significant advantage of either approach.

Intel x86 processor family is little-endian.

Motorola processor family is big-endian.

MIPS implementations can be either big-endian or little-endian.
Detecting Endianness

```c
const int IS_LITTLE_ENDIAN = 0;
const int IS_BIG_ENDIAN = 1;

int machineEndianness()
{
    long int i = 1;                     // >= 32 bits, LSB is 0x01
    const char *p = (const char *) &i;  // access low-address byte
    if (p[0] == 1)                      // see what's there
        return IS_LITTLE_ENDIAN;
    else
        return IS_BIG_ENDIAN;
}
```

Good stuff in C. Do you understand what's going on here?

Why it matters…

```c
#include <stdio.h>
int main (int argc, char* argv[])
{
    /* Our example data structure */
    struct {
        char one[4];
        int two;
        char three[4]; } data;
    /* Fill our structure with data */
    strcpy (data.one, "foo");
    data.two = 0x01234567;
    strcpy (data.three, "bar");
    /* Write it to a file */
    FILE* fp = fopen ("output", "wb");
    if (fp) {
        fwrite (&data, sizeof (data), 1, fp);
        fclose (fp);
    }
    return 0;
}
```

```
66 6f 6f 00 67 45 23 01 62 61 72 00 | foo.gE#.bar.
66 6f 6f 00 01 23 45 67 62 61 72 00 | foo..#Egbar.
```