MIPS quick-reference (see the MIPS Green Sheet for more)

Instruction	Syntax	Example
`add`	`add  dest, src0, src1`	`add  $s0, $s1, $s2`
`sub`	`sub  dest, src0, src1`	`sub  $s0, $s1, $s2`
`addi`	`addi dest, src0, immediate`	`addi $s0, $s1, 12`
`lw`	`lw   dest, offset(base addr)`	`lw   $t0, 4($s0)`
`sw`	`sw   src,  offset(base addr)`	`sw   $t0, 4($s0)`
`bne`	`bne  src0, src1, branchAddr`	`bne  $t0, $t1, notEq`
`beq`	`beq  src0, src1, branchAddr`	`beq  $t0, $t1, Eq`
`j`	`j    jumpAddr`	`j    jumpWhenDone`

C	MIPS

`// $s0 -> a, $s1 -> b`
`// $s2 -> c, $s3 -> z`
 	
`int a=4, b=5, c=6, z;`
`z = a+b+c+10;`
	addi \$s0, \$0, 4
	addi \$s1, \$0, 5
	addi \$s2, \$0, 6
	add \$s3, \$s0, \$s1
	add \$s3, \$s3, \$s2
	addi \$s3, \$s3, 10

`// $s0 -> int *p = intArr;`
`// $s1 -> a`
`p[0] = 0;`
`int a = 2;`
`p[1] = a;`
`p[a] = a;`
	sw \$0, 0(\$s0)
	addi \$s1, \$0, 2
	sw \$s1, 4(\$s0)
	sll \$t0, \$s1, 2
	add \$t0, \$t0, \$s0
	sw \$s1, 0(\$t0)

`// $s0 -> a, $s1 -> b`
`int a = 5, b = 10;`
`if (a + a == b) {`
`    a = 0;`
`} else {`
`    b = a - 1;`
`}`
	`addiu $s0, $0, 5`
	`addiu $s1, $0, 10`
	`add $t0, $s0, $s0`
	`bne $t0, $s1, else`
	`add $s0, $0, $0`
	`j exit`
	`else: addiu $s1, $s0, -1`
	`exit: # done!`

`/*What does this do?`
`  (Not C, in English) */`
Returns 2^30, or 2^N where N is the immediate on line 3
	`      addi $s0, $0,  0`
	`      addi $s1, $0,  1`
	`      addi $t0, $0,  30`
	`loop: beq  $s0, $t0, done`
	`      add  $s1, $s1, $s1`
	`      addi $s0, $s0, 1`
	`      j    loop`
	`done: # done!`

`int sum(int n) {`
`    return n ? n + sum(n - 1) : 0;`
`}`
`// use recursion in your MIPS!`
	sum: #for simplicity, we assume that n is >= 0
	addi \$sp, \$sp, -8 # allocate space on stack
	sw \$ra, 0(\$sp) # store the return address
	sw \$a0, 4(\$sp) # store the argument
	slti \$t0, \$a0, 1 # check if n > 0
	beq \$t0, \$0, recurse # n > 0 case
	add \$v0, \$0, \$0 # start return value to 0
	addi \$sp, \$sp, 8 # pop 2 items off stack
	jr \$ra # return to caller

	recurse:
	addi $a0, $a0, -1 # calculate n-1
	jal sum # recursively call sum(n-1)
	lw $ra, 0($sp) # restore saved return address
	
	lw $a0, 4($sp) #restore saved argument
	addi $sp, $sp, 8 #pop 2 items off stack
	add $v0, $a0, $v0 #calculate n + sum(n-1)
	jr $ra #return to caller

Implement `streq`, which sets `$v0` to true if its two character pointer arguments (`$a0` and `$a1`) point to equal strings (and false otherwise), in MIPS

First, let's write this in C:

```
int streq(char *s1, char *s2) {
     do {
          if(*s1 != *s2) {
               return 0;
          }
     } while(*s1++ && *s2++);
     return 1;
}
```

A straightforward translation into MIPS yields:

```
streq:
       lb $t0, 0($a0)              # get the character from s1
       lb $t1, 0($a1)              # get the character from s2
       bne $t0, $t1, notequal      # if they aren't equal, goto
                                   # notequal
       beq $t0, $0, equal          # if they are equal but zero,
                                   # we've checked the whole string;
       addi $a0, $a0, 1            # increment the pointer s1
       addi $a1, $a1, 1            # increment the pointer s2
       j streq                     # loop by jumping to the top
notequal: addi $v0, $0, 0
       jr $ra
equal: addi $v0, $0, 1
       jr $ra
```

What are the instructions to branch on each of the following conditions?

`$s0 < $s1`

```
slt $t0 $s0 $s1
bne $t0 $0 Lbl
```

`$s0 <= $s1`

```
slt $t0 $s1 $s0
beq $t0 $0 Lbl
```

`$s0 > 1`

```
addi $t1 $0 1
slt $t0 $t1 $s0
bne $t0 $0 Lbl
$s0 >= 1
slti $t0 $s0 1
beq $t0 $0 Lbl
```

What are the 3 meanings unsigned can have in MIPS?

lbu – Don't sign extend the loaded byte into the register
addu/addiu/subu – Do not warn on overflow.
sltu/sltiu – Perform unsigned comparison

What is the distinction between zero extension and sign extension? When do we use each?

We use extension when moving from a data type containing M bits to a data type containing N bits where N > M. Our extension operation (zero or sign) determines how we fill in the empty space on the left hand side. With zero extension, we simply fill the leftmost N-M bits with zeroes. With sign extension, we set each of the leftmost N-M bits to the value of the (M-1)th bit (assuming we begin counting from zero) of the original value. Thus, we are effectively copying the sign of the original value into all of the "extra" space. If we consider a number in two's complement, it is easy to see that performing sign extension preserves both the magnitude and sign of the number.