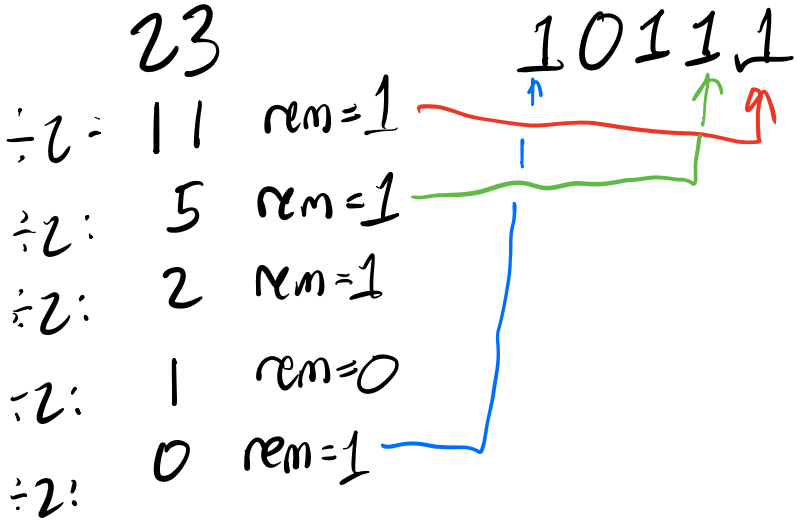
Quick tricks for converting between decimal and binary.

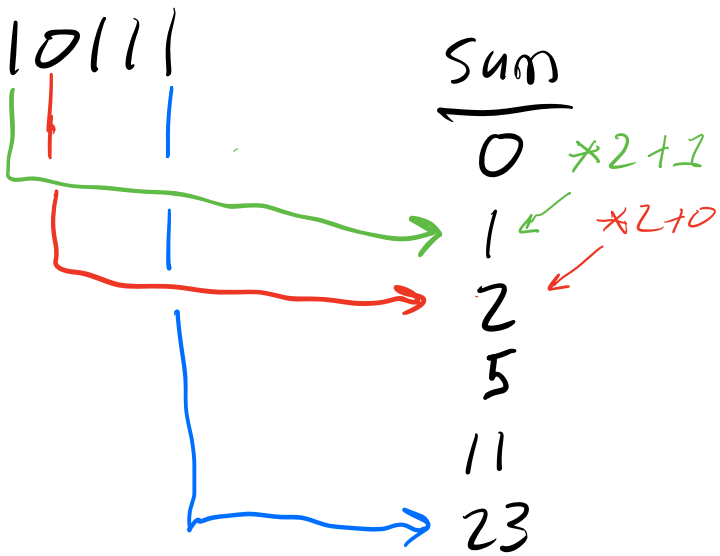
① Decimal to binary

- repeatedly divide the decimal number by 2, writing the remainder each time (0 or 1) into the result, moving right to left.



② Binary to decimal

Start with sum=0 and move left to right in the binary number, each time doubling the sum and adding the value of the current binary digit (0 or 1)



Back to assembl language

Accessing Data using Addressing Modes

- "Immediate"

```
mov $23, %rax
```

↑ numeric literal = 23

- "Register"

```
add %rax, %rcx
```

- "Absolute Address"
  - for accessing memory
    - retrieving data from memory, "fetch"
    - sending data to memory, "store"

The address is specified as a number

```
movq 1000, %rdx
```

↑ moves the contents at address 1000 into %rdx
- moves 64 bits (8 bytes) starting at address 1000.

You will never write a number as an absolute address.

- the assembler will let you use labels (names) for global variables.

- "Pointer"
  - a register contains an address, and that address is dereferenced.

```
addq %rbx, (%rax)
```

this is the place in memory that %rax points to.

The parentheses are super important!

totally different!

```
add %rbx, %rax
add %rbx, (%rax)
```

The analogy in C:

```
x = x + 3;
*x = *x + 3;
```

} totally different!

"Pointer + offset"

- Accesses memory at some offset from where a register points to.

```
Mov 12(%rsi), %rbx
```

specifies an address that is %rsi + 12.

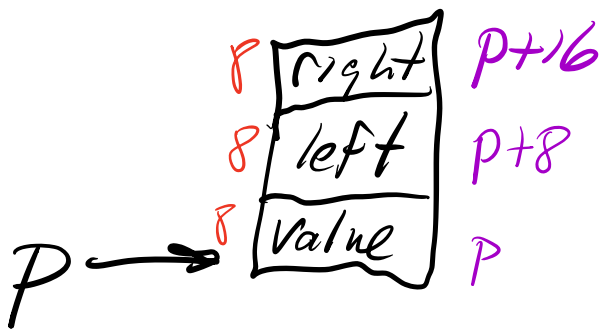
So, if %rsi contains 1000, the memory location accessed is 1000 + 12 = 1012.

This is useful for structs in C.

```
typedef struct node {
    long value;
    struct node *left;
    struct node *right;
} NODE;
```

↑ all 8 bytes

```
NODE *p = malloc(sizeof(NODE));
```



Assuming p is in the %rcx register:

(%rcx) or 0(%rcx)
gives us p → value

8(%rcx) gives us p → left

16(%rcx) gives us p → right

- the offset can also be negative

"Indexed Addressing"

subq %rdx, (%rsi, %rdi, 4)

the address used here is:

%rsi + (%rdi * 4)

If %rsi contains 1000 and %rdi contains 30, then the address is 1000+(30*4)= 1120.

- So, the value in %rdx will be subtracted from whatever is in the 64 bits starting at address 1120.

Useful for arrays:

(%r8, %r5x, 8)

- %r8: points to start of the array
- %r5x: contains the index into the array (must be a 64-bit register)
- 8: size of each element of the array.

Example - adding up the elements of an array.

- Suppose %rdi points to the start of the array.
- Suppose that %rsi contains the size (number of elements) of the array
- Suppose that each element is an "int" (4 bytes)

4-byte register

"int" (4 bytes)

```
        movl $0, %eax   // sum = 0
        mov $0, %rcx    // i = 0
LOOP_TOP:
        cmp %rsi, %rcx  // compare i to size
        jge OUT         // jump if i >= size
        addl (%rdi, %rcx, 4), %eax
                        // sum += a[i];
        incq %rcx       // i++
        jmp LOOP_TOP
OUT:
        // at this point, %eax contains
        // the sum of the elements of the
        // array.
```

size

i

"Indexed + offset"

24(%rcx, %rdi, 8)

address computed is

24 + %rcx + (%rdi * 8)

Useful when a field of a struct is an array.

24(%rcx, %rdi, 8)

- offset of the array in the struct
- start of the struct
- index into the array
- size of each element of the array.