

Negative Numbers

32 bits can only represent 2^{32} numbers – if we wish to also represent negative numbers, we can represent 2^{31} positive numbers (incl zero) and 2^{31} negative numbers

$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{\text{two}} = 0_{\text{ten}}$$

$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{\text{two}} = 1_{\text{ten}}$$

...

$$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{\text{two}} = 2^{31}-1$$

$$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{\text{two}} = -2^{31}$$

$$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{\text{two}} = -(2^{31} - 1)$$

$$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{\text{two}} = -(2^{31} - 2)$$

...

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{\text{two}} = -2$$

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{\text{two}} = -1$$

2's Complement

0000 0000 0000 0000 0000 0000 0000 0000_{two} = 0_{ten}

0000 0000 0000 0000 0000 0000 0000 0001_{two} = 1_{ten}

...

0111 1111 1111 1111 1111 1111 1111 1111_{two} = $2^{31}-1$

1000 0000 0000 0000 0000 0000 0000 0000_{two} = -2^{31}

1000 0000 0000 0000 0000 0000 0000 0001_{two} = $-(2^{31} - 1)$

1000 0000 0000 0000 0000 0000 0000 0010_{two} = $-(2^{31} - 2)$

...

1111 1111 1111 1111 1111 1111 1111 1110_{two} = -2

1111 1111 1111 1111 1111 1111 1111 1111_{two} = -1

Why is this representation favorable?

Consider the sum of 1 and -2 we get -1

Consider the sum of 2 and -1 we get +1

This format can directly undergo addition without any conversions!

Each number represents the quantity

$$x_{31} -2^{31} + x_{30} 2^{30} + x_{29} 2^{29} + \dots + x_1 2^1 + x_0 2^0$$

2's Complement

0000 0000 0000 0000 0000 0000 0000 0000_{two} = 0_{ten}

0000 0000 0000 0000 0000 0000 0000 0001_{two} = 1_{ten}

...

0111 1111 1111 1111 1111 1111 1111 1111_{two} = $2^{31}-1$

1000 0000 0000 0000 0000 0000 0000 0000_{two} = -2^{31}

1000 0000 0000 0000 0000 0000 0000 0001_{two} = $-(2^{31} - 1)$

1000 0000 0000 0000 0000 0000 0000 0010_{two} = $-(2^{31} - 2)$

...

1111 1111 1111 1111 1111 1111 1111 1110_{two} = -2

1111 1111 1111 1111 1111 1111 1111 1111_{two} = -1

Note that the sum of a number x and its inverted representation x' always equals a string of 1s (-1).

$$x + x' = -1$$

$x' + 1 = -x$... hence, can compute the negative of a number by

$-x = x' + 1$ inverting all bits and adding 1

Similarly, the sum of x and $-x$ gives us all zeroes, with a carry of 1

In reality, $x + (-x) = 2^n$... hence the name 2's complement

Example

- Compute the 32-bit 2's complement representations for the following decimal numbers:
5, -5, -6

Example

- Compute the 32-bit 2's complement representations for the following decimal numbers:

5, -5, -6

5: 0000 0000 0000 0000 0000 0000 0000 0101

-5: 1111 1111 1111 1111 1111 1111 1111 1011

-6: 1111 1111 1111 1111 1111 1111 1111 1010

Given -5, verify that negating and adding 1 yields the number 5

Signed / Unsigned

- The hardware recognizes two formats:

unsigned (corresponding to the C declaration `unsigned int`)

-- all numbers are positive, a 1 in the most significant bit just means it is a really large number

signed (C declaration is `signed int` or just `int`)

-- numbers can be +/- , a 1 in the MSB means the number is negative

This distinction enables us to represent twice as many numbers when we're sure that we don't need negatives

MIPS Instructions

Consider a comparison instruction:

```
slt $t0, $t1, $zero
```

and \$t1 contains the 32-bit number 1111 01...01

What gets stored in \$t0?

MIPS Instructions

Consider a comparison instruction:

```
slt $t0, $t1, $zero
```

and \$t1 contains the 32-bit number 1111 01...01

What gets stored in \$t0?

The result depends on whether \$t1 is a signed or unsigned number – the compiler/programmer must track this and accordingly use either `slt` or `sltu`

```
slt $t0, $t1, $zero    stores 1 in $t0
```

```
sltu $t0, $t1, $zero   stores 0 in $t0
```

Sign Extension

- Occasionally, 16-bit signed numbers must be converted into 32-bit signed numbers – for example, when doing an add with an immediate operand
- The conversion is simple: take the most significant bit and use it to fill up the additional bits on the left – known as sign extension

So 2_{10} goes from 0000 0000 0000 0010 to
0000 0000 0000 0000 0000 0000 0000 0010

and -2_{10} goes from 1111 1111 1111 1110 to
1111 1111 1111 1111 1111 1111 1111 1110

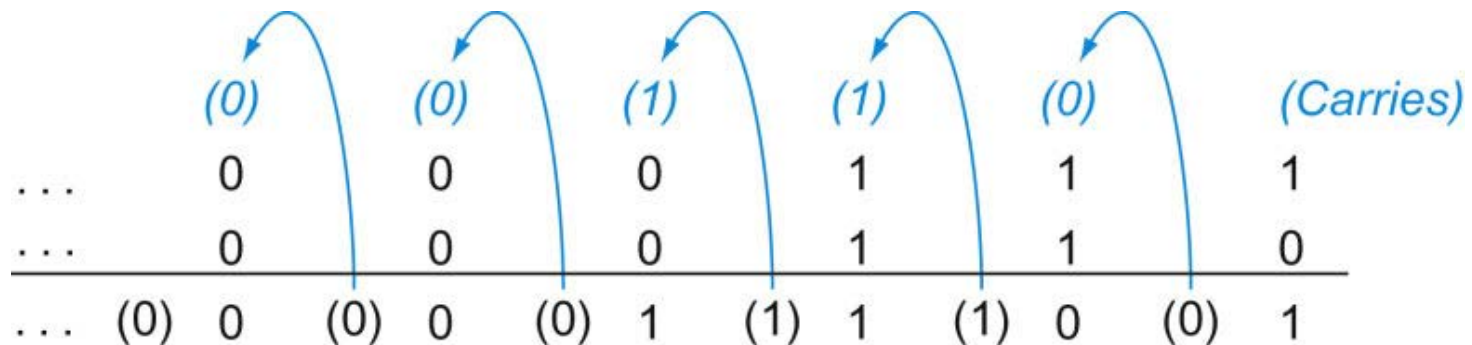
Alternative Representations

- The following two (intuitive) representations were discarded because they required additional conversion steps before arithmetic could be performed on the numbers
 - sign-and-magnitude: the most significant bit represents +/- and the remaining bits express the magnitude
 - one's complement: $-x$ is represented by inverting all the bits of x

Both representations above suffer from two zeroes

Addition and Subtraction

- Addition is similar to decimal arithmetic
- For subtraction, simply add the negative number – hence, subtract $A-B$ involves negating B 's bits, adding 1 and A



Source: H&P textbook

Overflows

- For an unsigned number, overflow happens when the last carry (1) cannot be accommodated
- For a signed number, overflow happens when the most significant bit is not the same as every bit to its left
 - when the sum of two positive numbers is a negative result
 - when the sum of two negative numbers is a positive result
 - The sum of a positive and negative number will never overflow
- MIPS allows `addu` and `subu` instructions that work with unsigned integers and never flag an overflow – to detect the overflow, other instructions will have to be executed