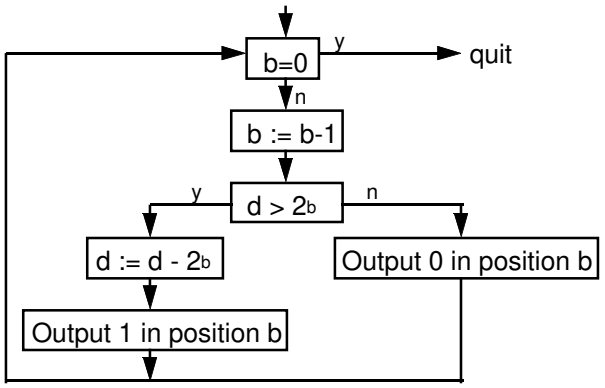
Arithmetic

Computers do not store numbers or letters, per se. They only store bit sequences. The bit sequences can be interpreted as representing integers or floating pint numbers. Arithmetic is accomplished by the direct hardware implementation of arithmetic algorithms

Converting Decimal to b-Bit Binary

- Let d be decimal number, less than 2^b-1



1998 in 12-bit

d		2^b
1998		2048:0
1998	-	1024:1
974	-	512:1
462	-	256:1
206	-	128:1
78	-	64:1
14		32:0
14		16:0
14	-	8:1
6	-	4:1
2	-	2:1
0		1:0

Representation

- Terminology
  - *least significant bit* (lsb): least magnitude bit.
  - *most significant bit* (msb): greatest magnitude bit.
  - *unsigned integers*: k bit sequence representing 0 to 2^k-1
  - *two's complement*: number representation for signed integers.

Unsigned Binary	Decimal
0000 0000 0000 0000	0
0000 0000 0000 0001	1
0000 0000 0000 0010	2
. . .	
1111 1111 1111 1101	65533
1111 1111 1111 1110	65534
1111 1111 1111 1111	65535

Unsigned gets a larger range at the expense of no negative representation

2s Complement Binary	Decimal
0000 0000 0000 0000	0
0000 0000 0000 0001	1
0000 0000 0000 0010	2
. . .	
0111 1111 1111 1111	32767
1000 0000 0000 0000	-32768
1000 0000 0000 0001	-32767
1000 0000 0000 0010	-32766
. . .	
1111 1111 1111 1101	-3
1111 1111 1111 1110	-2
1111 1111 1111 1111	-1

2s Complement is “unbalanced” since it has 1 more negative number than positive numbers. Why?

Converting 2s Complement

A positive number is its unsigned equivalent, provided the msb is 0.

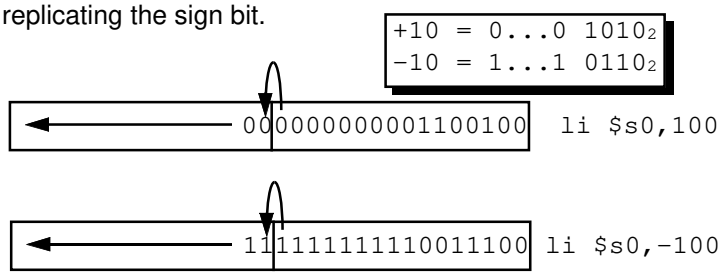
Finding -x given x: Complement all bits & add 1

In n-bit 2s complement, $x + -x = 2^n$

	$0000\ 0000\ 0000\ 1010_2$	10_{10}
Compl.:	1111 1111 1111 0101	
Add 1:	+ 1	
	$1111\ 1111\ 1111\ 0110_2$	-10_{10}
Compl.:	0000 0000 0000 1001	
Add 1:	+ 1	
	$0000\ 0000\ 0000\ 1010_2$	10_{10}

What's Your Sign?

- The standard `add`, `sub`, `addi`, etc. instructions use signed 2s complement values
  - The immediate field is *sign-extended* to make a 32 bit value.
  - A 2s complement value represented with *x* bits can be converted to the same value in a *y* bit representation *y* > *x*, by replicating the sign bit.



Sign extend is also applied when loading bytes and halfwords, i.e. lb and lh

Unsigned Instructions

MIPS provides a set of operations to perform unsigned arithmetic:

```
- addu $8, $9, $10    # add unsigned
- subu $8, $9, $10    # find difference
- addiu  $8, $9,10    # add unsigned immediate
- multu  $8, $9, $10 # multiply unsigned
- divu   $8, $9, $10 # divide unsigned
- lbu    $8, 0($9)    # load byte unsigned
- lhu    $8, 0($9)    # load halfword unsigened
```

- Notice that the "u" modifier follows the opcode

The most common application of unsigned arithmetic is for address calculations

Representations

- A bit sequence is neither signed nor unsigned, integer or floating point, character or pixel ... its just a bit sequence -- the key is how the bits are interpreted
- The interpretation nearly always matters, especially in comparisons :
  - 10110 < 00110 is true since -10 < 6 as 2s compl.
  - 10110 > 00110 is true since 22 > 6 as unsigned
- MIPS has additional comparison operators:
  - `sltu $8, $9, $10 #set less than unsigned`
  - `sltiu $8, $9, 10 #set less than immed. unsign`

Why are there no unsigned variants for `beq, bne, bgtz, blez, sh, sb`, etc.?

Facts of Finite Representation

When combining two numbers produces a result larger than can be represented in the available space, an overflow occurs

Overflow is not always bad, but it must be reportable

Overflow is impossible when ...

- adding numbers with opposite signs because the result is in between
- subtracting numbers with like signs, since the “addition” rule applies once B is negated

Conditions causing overflow:

Operation	Op A	Op B	Overflow
A+B	≥0	≥0	<0
A+B	<0	<0	≥0
A-B	≥0	<0	<0
A-B	<0	≥0	≥0