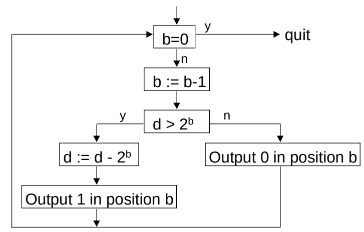
Arithmetic

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

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

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

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}

Subtraction

- The well-known rule that subtraction is equivalent to addition of a negated operand is important in computing

  a - b ≡ a + (-b)

- In the arithmetic-logic unit of a computer, there is no subtraction circuitry per se, just negation

	010 . . . 0	a
	110 . . . 1	$\bar{b}$
+	000 . . . 1	1

Consequences of Signed Fields

A field of b bits can represent 2^b configurations

- If the field is used for unsigned numbers …

  Range: 0 to 2^b-1

- If the field is used for signed numbers …

  Range: -2^{b-1} to 2^{b-1}-1

- If the field is used so that some bits are always the same, then do not represent them

  Range of byte addresses for instructions:

  0 to 2^{b+2} since least significant bits are 00

$$\underbrace{\boxed{000\ .\ .\ .\ 0}}_{b}$$

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 complement
  - 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

```
  010 . . . 0
+ 010 . . . 0
  -----------
  100 . . . 0
```

Overflow is not always bad, but it must be reportable

Overflow is impossible when ...

- *adding numbers with opposite signs because the result is numerically between the operands*
- *subtracting numbers with like signs, since the "addition" rule applies once B is negated*

Conditions Causing Overflow

- For add operations (add and substract), the overflow can be detected by the sign of the operands and the result

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

```
  010 . . . 0
+ 010 . . . 0
  -----------
  100 . . . 0
```

Notice that the subtraction rule is simply the Addition rule applied when subtraction is negation of the second operand followed by addition