Lecture 8: Number Crunching

- Today's topics:
 - MARS wrap-up
 - RISC vs. CISC
 - Numerical representations
 - Signed/Unsigned

School of Computing Guidelines

Class rosters are provided to the instructor with the student's legal name as well as "Preferred first name" (if previously entered by you in the Student Profile section of your CIS account). While CIS refers to this as merely a preference, I will honor you by referring to you with the name and pronoun that feels best for you in class, on papers, exams, etc. Please advise me of any name or pronoun changes (and please update CIS) so I can help create a learning environment in which you, your name, and your pronoun will be respected.

Cheating policy:

Working with others on assignments is a good way to learn the material and is encouraged. However, there are limits to the degree of cooperation that is permitted. Students may discuss among themselves the meaning of homework problems and possible approaches to solving them. Any written portion of an assignment, however, is to be done strictly on an individual basis. Note the School of Computing's <u>Academic Misconduct Policy</u>. BOTTOM LINE: You may not copy from another student or from any other source, and you may not allow another student to copy your work!! Any violation of the above is considered to be cheating and will result in a reduced or a failing grade in the class. TAs will be on the lookout for solution sets that appear very similar. Also, if your class rank in the assignments is significantly different from your class rank in the exams, only your rank in the exams will count towards your final grade.

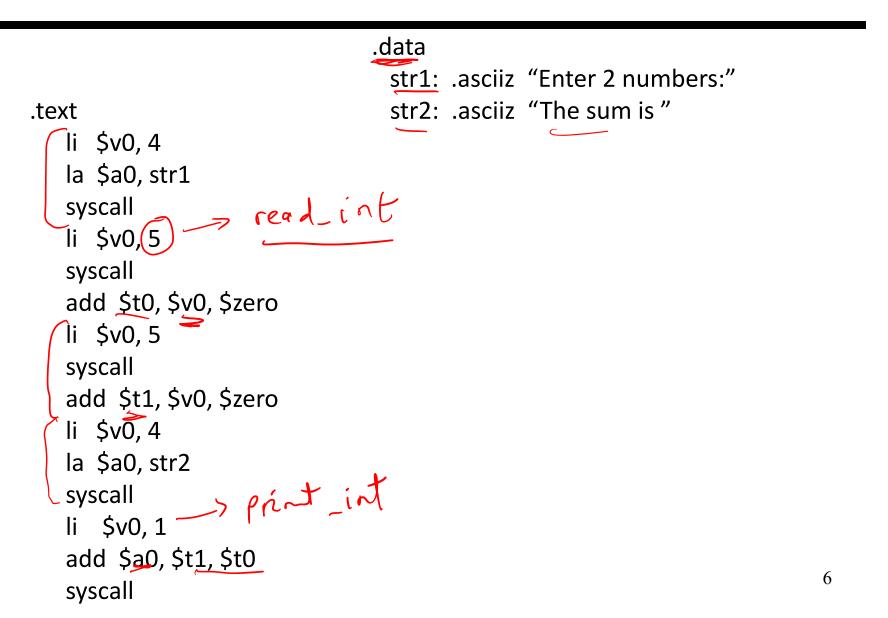
Example Print Routine

.data	ctives & labels => assender is sheltering you from low level details "the answer is" syscalls
li \$v0, 4 la \$a0, str	<pre># load immediate; 4 is the code for print_string # the print_string syscall expects the string # address as the argument; la is the instruction # to load the address of the operand (str)</pre>
syscall	# MARS will now invoke syscall-4
li \$v0,1	<pre># syscall-1 corresponds to print_int</pre>
li \$a0, 5	<pre># print_int expects the integer as its argument</pre>
syscall	# MARS will now invoke syscall-1

Example

• Write an assembly program to prompt the user for two numbers and print the sum of the two numbers

Example

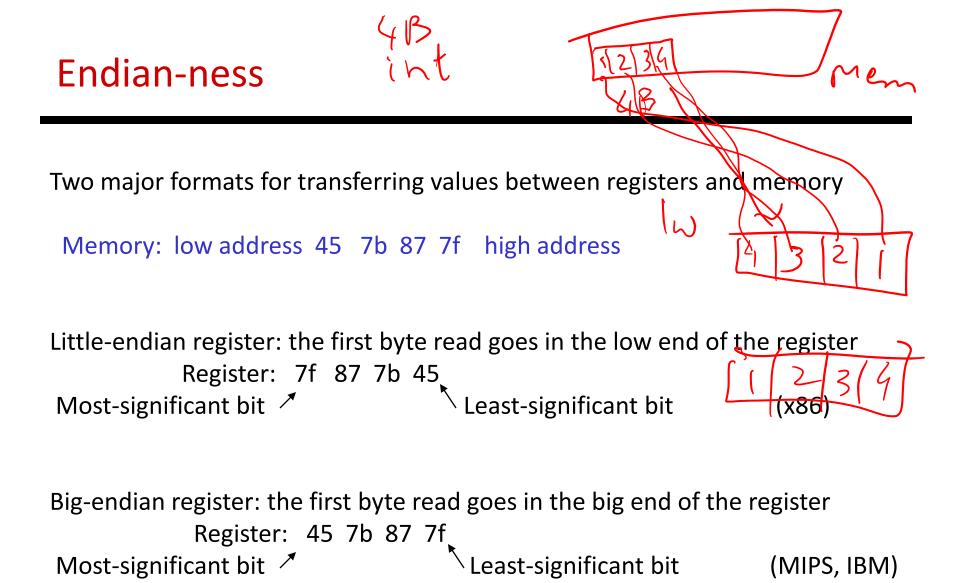


IA-32 Instruction Set

- Intel's IA-32 instruction set has evolved over 20 years old features are preserved for software compatibility
- Numerous complex instructions complicates hardware design (Complex Instruction Set Computer – CISC)
- Instructions have different sizes, operands can be in registers or memory, only 8 general-purpose registers, one of the operands is over-written
- RISC instructions are more amenable to high performance (clock speed and parallelism) – modern Intel processors convert IA-32 instructions into simpler micro-operations



RISC US CISC



Binary Representation The binary number 01011000 00010101 00101110 111001 Least significant bit Most significant bit 1251 represents the quantity $0 \times 2^{31} + 1 \times 2^{30} + 0 \times 2^{29} + ... + 1 \times 2^{0}$

A 32-bit word can represent 2³² numbers between
 0 and 2³²-1

... this is known as the unsigned representation as we're assuming that numbers are always positive

ASCII Vs. Binary

- Does it make more sense to represent a decimal number in ASCII?
- Hardware to implement arithmetic would be difficult
- What are the storage needs? How many bits does it take to represent the decimal number 1,000,000,000 in ASCII and in binary?

ASCII Vs. Binary

- Does it make more sense to represent a decimal number in ASCII?
- Hardware to implement arithmetic would be difficult
- What are the storage needs? How many bits does it take to represent the decimal number 1,000,000,000 in ASCII and in binary?
 In binary: 30 bits (2³⁰ > 1 billion)
 In ASCII: 10 characters, 8 bits per char = 80 bits

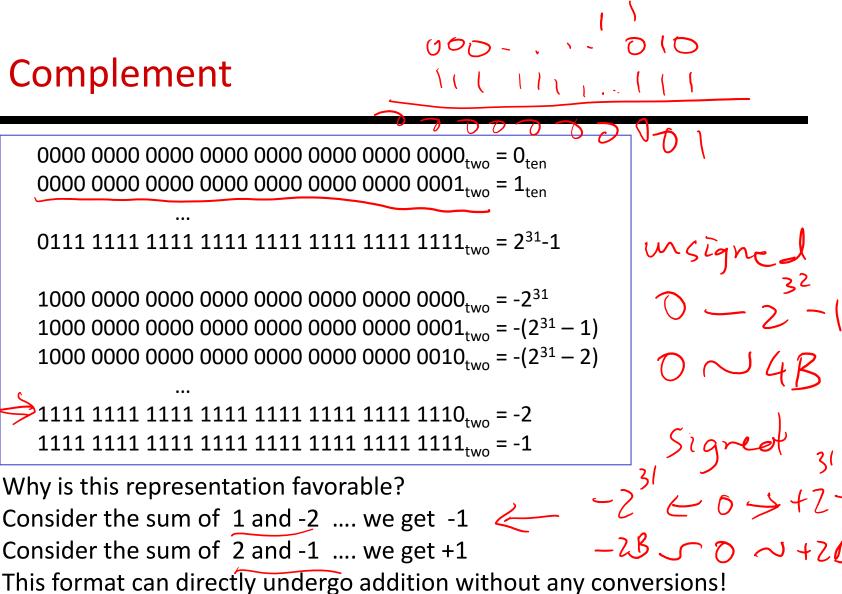


Signed

32 bits can only represent 2³² numbers – if we wish to also represent negative numbers, we can represent 2³¹ positive numbers (incl zero) and 2³¹ negative numbers

 $\begin{array}{c} 0000\ 000\ 00\ 000\ 000\ 000\ 000\ 00\ 000\ 000\ 000\ 000\ 00\ 00\ 000\ 0$

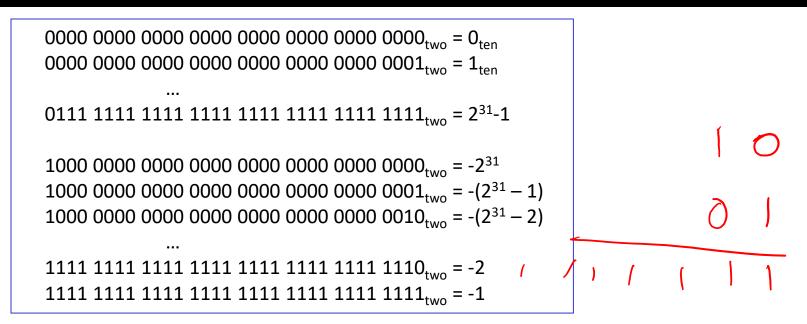
2's Complement



Each number represents the quantity

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

2's Complement



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

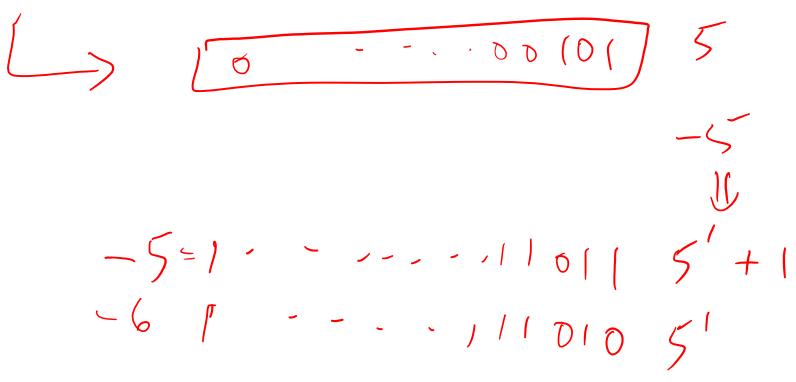
$$x + x' = -1$$
 $x' + 1 = -x$ $-x = x' + 1$... hence, can compute the negative of a number by $-x = x' + 1$ inverting all bits and adding 1

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 $\sim 2 = 2 - 2 + 1$

5, -5, -6

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



Example

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

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

• The hardware recognizes two formats:



unsigned (corresponding to the C declaration <u>unsigned int</u>) -- all numbers are positive, a 1 in the most significant bit just means it is a really large number -2B L D -+24

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

set on less than

Consider a comparison instruction: slt \$t0, \$t1, \$zero and \$t1 contains the 32-bit number 1111 01...01 What gets stored in \$t0? (not less tha) unsigned 3to = 0sltu Signed (less than) \$7.0 = 1 SIL 18

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





• Occasionally, 16-bit signed numbers must be converted into 32-bit signed numbers – for example, when doing an 1 [] 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

and -2₁₀ goes from 1111 1111 1111 1110 to 1111 1111 1111 1111 1111 1110

- 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