Lecture 8: Number Crunching

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

.data

str: .ascii	z "the	answer	is	"
-------------	--------	--------	----	---

.text

li	\$v0, 4	# load immediate; 4 is the code for print_string		
la \$a0, str		# the print_string syscall expects the string		
		# address as the argument; la is the instruction		
		# to load the address of the operand (str)		
sys	call	# 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>		
sys	call	# MARS will now invoke syscall-1		

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

	.data	
	str1: .asciiz "Enter 2 numbers:"	
.text	str2: .asciiz "The sum is "	
li \$v0 <i>,</i> 4		
la \$a0, str1		
syscall		
li \$v0, 5		
syscall		
add \$t0, \$v0, \$zero		
li \$v0, 5		
syscall		
add \$t1, \$v0, \$zero		
li \$v0,4		
la \$a0, str2		
syscall		
, li \$v0,1		
add \$a0, \$t1, \$t0		
syscall		4

- 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

Two major formats for transferring values between registers and memory

Memory: low address 45 7b 87 7f high address

Little-endian register: the first byte read goes in the low end of the register Register: 7f 87 7b 45 Most-significant bit / Least-significant bit (x86)

Big-endian register: the first byte read goes in the big end of the register Register: 45 7b 87 7f Most-significant bit / Least-significant bit (MIPS, IBM)

Binary Representation

• The binary number

01011000 00010101 0010110 11100111Most significant bitLeast significant bit

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 $(2^{30} > 1 \text{ billion})$
 - In ASCII: 10 characters, 8 bits per char = 80 bits

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

 $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ two = 0_{ten}$ $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = 1_{ten}$

 $\begin{array}{l} 1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ _{two} = -2^{31} \\ 1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001\ _{two} = -(2^{31}-1) \\ 1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010\ _{two} = -(2^{31}-2) \end{array}$

1111 1111 1111 1111 1111 1111 1110_{two} = -2 1111 1111 1111 1111 1111 1111 1111_{two} = -1

2's Complement

```
\begin{array}{l} 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = 0_{ten} \\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = 1_{ten} \\ ... \\ 0111\ 1111\ 1111\ 1111\ 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\ two = -(2^{31} - 1) \\ 1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ two = -(2^{31} - 1) \\ 1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ two = -(2^{31} - 2) \\ ... \\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ two = -2 \\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ two = -1 \end{array}
```

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

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

2's Complement

 $\begin{array}{l} 0000 \ 00000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0$

 $\begin{array}{l} 1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ _{two} = -2^{31} \\ 1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001\ _{two} = -(2^{31}-1) \\ 1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010\ _{two} = -(2^{31}-2) \\ \end{array}$

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

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

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

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

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

• 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

Consider a comparison instruction: slt \$t0, \$t1, \$zero and \$t1 contains the 32-bit number 1111 01...01

What gets stored in \$t0?

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 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 2₁₀ goes from 0000 0000 0000 0010 to 0000 0000 0000 0000 0000 0000 0010

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