Lecture 8: Number Crunching

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

syscall $v0 type
$a0 argument

Example Print Routine

directives

pseudo instrs like la
labels & using them
to manage mem addr

```
.data
  str:    .asciiz   "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)
  syscall                   #  MARS will now invoke syscall-4
  li      $v0, 1            #  syscall-1 corresponds to print_int
  li      $a0, 5            #  print_int expects the integer as its argument
  syscall                   #  MARS will now invoke syscall-1
```

load addr of

To put "5" in $a0, we can also do:

```
.data
  myint:  .word 5
.text
  la  $t0, myint
  lw  $a0, 0($t0)
```

Example

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

Example

```
.data
  str1:  .asciiz  "Enter 2 numbers:"
  str2:  .asciiz  "The sum is "
```

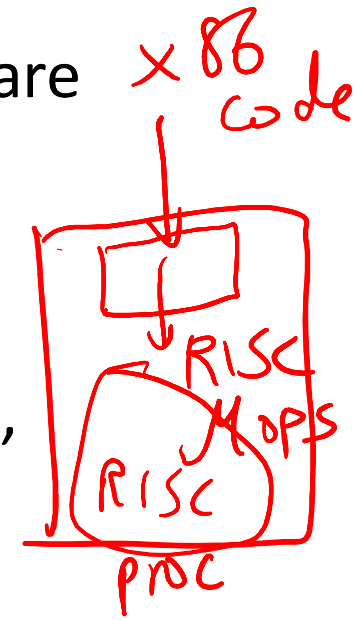
```
.text
  li   $v0, 4
  la  $a0, str1        print str1
  syscall
  li   $v0, 5          read int       places the return value
  syscall
  add  $t0, $v0, $zero   mov $v0 -> $t0     in $v0
  li   $v0, 5
  syscall              read int
  add  $t1, $v0, $zero   mov $v0 -> $t1
  li   $v0, 4
  la  $a0, str2        } print str
  syscall
  li    $v0, 1
  add  $a0, $t1, $t0   } print int
  syscall
```

IA-32 Instruction Set

RISC vs CISC

(MIPS) Reduced — complex (x86)

- 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



Endian-ness

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 00101110 11100111

Most significant bit

Least significant bit

unsigned

represents the quantity

$0 \times 2^{31} + 1 \times 2^{30} + 0 \times 2^{29} + \ldots + 1 \times 2^{0}$

$0 \rightarrow 2^{32} - 1$

- A 32-bit word can represent 2^{32} numbers between 0 and 2^{32}-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 ($2^{30} > 1$ billion)

  In ASCII: 10 characters, 8 bits per char = 80 bits

Negative Numbers ~~#~~ Signed

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

2's complement

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

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

…

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

sign + magnitude repr

1000 0000 0000 0000 0000 0000 0000 0000_{two} = -2^{31} — -0

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

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

…

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

$2^{32}-1$ — 1111 1111 1111 1111 1111 1111 1111 1111_{two} = -1 — $-2^{31}-1$

2's Complement

dec 2 0010
dec 9 1001
dec 11 ← 1011

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = 0_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = 1_{ten}$
…
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = 2^{31}-1$

$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = -2^{31}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = -(2^{31} - 1)$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = -(2^{31} - 2)$
…
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = -2$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = -1$

3 + 9 = 12

1 1
0011
1001
12 = 1100

00….001
11….110
11….111

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

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

2's Complement

$x = 7 = 000111$

$-x = -7 = x' + 1 = 111000 + 1$

$= 111 \ldots 111001$

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = 0_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = 1_{ten}$
...
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = 2^{31}-1$

$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = -2^{31}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = -(2^{31} - 1)$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = -(2^{31} - 2)$
...
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = -2$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = -1$

$x = 1101$

$x' = 0010$

$x + x' = 1111$

$x + x' = -1$

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

−5 = 5 ⇒ flip its bits, add 1

00···0101

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

flip 11···1010

+1 11···1011

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 inverting and adding 1 yields the number 5 default

32b reg

Signed / Unsigned

$-2^{31} \longleftrightarrow +2^{31}$ signed

- The hardware recognizes two formats:

-2 billion +2 billion

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

32b unsigned reg

$0 \longrightarrow 2^{32}$

$0 \longrightarrow 4$ billion

signed (C declaration is signed int or just int)
-- numbers can be +/- , a 1 in the MSB means the number
is negative

64 b regs

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

$-2^{63} \longleftrightarrow +2^{63}$

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

0 4 2 1
+ 0 0 3 6

addi $t0, $t1, -43

16b
const

- Occasionally, 16-bit signed numbers must be converted into 32-bit signed numbers – for example, when doing an add with an immediate operand

1111111 1
000000 0 ......

16b

- 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

11 ......

$t1

32b

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