Lecture 8: Number Crunching

Today’s topics:

- MARS wrap-up
- RISC vs. CISC
- Numerical representations
- Signed/Unsigned
Example Print Routine

.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
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
    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
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
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

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 $2^{32}$ numbers between 0 and $2^{32}-1$

... this is known as the unsigned representation as we’re assuming that numbers are always positive
• 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
Negative Numbers

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

\[
\begin{align*}
0000 0000 0000 0000 0000 0000 0000 0000_{\text{two}} &= 0_{\text{ten}} \\
0000 0000 0000 0000 0000 0000 0000 0001_{\text{two}} &= 1_{\text{ten}} \\
\text{...} \\
0111 1111 1111 1111 1111 1111 1111 1111_{\text{two}} &= 2^{31}-1 \\
1000 0000 0000 0000 0000 0000 0000 0000_{\text{two}} &= -2^{31} \\
1000 0000 0000 0000 0000 0000 0000 0001_{\text{two}} &= -(2^{31} - 1) \\
1000 0000 0000 0000 0000 0000 0000 0010_{\text{two}} &= -(2^{31} - 2) \\
\text{...} \\
1111 1111 1111 1111 1111 1111 1111 1110_{\text{two}} &= -2 \\
1111 1111 1111 1111 1111 1111 1111 1111_{\text{two}} &= -1
\end{align*}
\]
## 2’s Complement

<table>
<thead>
<tr>
<th>Binary</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 0000 0000 0000 0000 0000 0000 0000 _two = 0_{ten}</td>
<td></td>
</tr>
<tr>
<td>0000 0000 0000 0000 0000 0000 0000 0001 _two = 1_{ten}</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0111 1111 1111 1111 1111 1111 1111 1111 _two = 2^{31} - 1</td>
<td></td>
</tr>
<tr>
<td>1000 0000 0000 0000 0000 0000 0000 0000 _two = -2^{31}</td>
<td></td>
</tr>
<tr>
<td>1000 0000 0000 0000 0000 0000 0000 0001 _two = -(2^{31} - 1)</td>
<td></td>
</tr>
<tr>
<td>1000 0000 0000 0000 0000 0000 0000 0010 _two = -(2^{31} - 2)</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>1111 1111 1111 1111 1111 1111 1111 1110 _two = -2</td>
<td></td>
</tr>
<tr>
<td>1111 1111 1111 1111 1111 1111 1111 1111 _two = -1</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Binary</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 0000 0000 0000 0000 0000 0000 0000&lt;sub&gt;two&lt;/sub&gt; = 0&lt;sub&gt;ten&lt;/sub&gt;</td>
<td></td>
</tr>
<tr>
<td>0000 0000 0000 0000 0000 0000 0000 0001&lt;sub&gt;two&lt;/sub&gt; = 1&lt;sub&gt;ten&lt;/sub&gt;</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0111 1111 1111 1111 1111 1111 1111 1111&lt;sub&gt;two&lt;/sub&gt; = 2&lt;sup&gt;31&lt;/sup&gt;-1</td>
<td></td>
</tr>
<tr>
<td>1000 0000 0000 0000 0000 0000 0000 0000&lt;sub&gt;two&lt;/sub&gt; = -2&lt;sup&gt;31&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>1000 0000 0000 0000 0000 0000 0000 0000 0001&lt;sub&gt;two&lt;/sub&gt; = -(2&lt;sup&gt;31&lt;/sup&gt; – 1)</td>
<td></td>
</tr>
<tr>
<td>1000 0000 0000 0000 0000 0000 0000 0000 0010&lt;sub&gt;two&lt;/sub&gt; = -(2&lt;sup&gt;31&lt;/sup&gt; – 2)</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>1111 1111 1111 1111 1111 1111 1111 1110&lt;sub&gt;two&lt;/sub&gt; = -2</td>
<td></td>
</tr>
<tr>
<td>1111 1111 1111 1111 1111 1111 1111 1111&lt;sub&gt;two&lt;/sub&gt; = -1</td>
<td></td>
</tr>
</tbody>
</table>

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 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

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

  - 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
Signed / Unsigned

- 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:
\[ \text{slt } \$t0, \$t1, \$zero \]
and \$t1 contains the 32-bit number \[1111\ 01\ldots01\]

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

• 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 $2_{10}$ 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