Lecture 7: MARS, Computer Arithmetic

• Today’s topics:
  ▪ MARS intro
  ▪ Numerical representations
  ▪ Addition and subtraction
MARS Intro

- Directives, labels, global pointers, system calls
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
    syscall # address as the argument; la is the instruction
            # to load the address of the operand (str)
    syscall # SPIM 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 # SPIM 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

.text
.globl main
main:
    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

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

• The binary number

\[ 01011000 \ 00010101 \ 00101110 \ 11100111 \]

represents the quantity
\[ 0 \times 2^{31} + 1 \times 2^{30} + 0 \times 2^{29} + \ldots + 1 \times 2^0 \]

• 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
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
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

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

\[
\begin{array}{c}
0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{\text{two}} = 0_{\text{ten}} \\
0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{\text{two}} = 1_{\text{ten}} \\
\vdots \\
0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{\text{two}} = 2^{31}-1 \\
1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{\text{two}} = -2^{31} \\
1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{\text{two}} = -(2^{31} - 1) \\
1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{\text{two}} = -(2^{31} - 2) \\
\vdots \\
1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{\text{two}} = -2 \\
1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{\text{two}} = -1 \\
\end{array}
\]

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
\[
x_{31} \cdot 2^{31} + x_{30} \cdot 2^{30} + x_{29} \cdot 2^{29} + \ldots + x_1 \cdot 2^1 + x_0 \cdot 2^0
\]
2’s Complement

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

\[
\begin{align*}
0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{\text{two}} &= 0_{\text{ten}} \\
0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{\text{two}} &= 1_{\text{ten}} \\
&\vdots \\
0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{\text{two}} &= 2^{31}-1
\end{align*}
\]

\[
\begin{align*}
1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{\text{two}} &= -2^{31} \\
1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{\text{two}} &= -(2^{31} - 1) \\
1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{\text{two}} &= -(2^{31} - 2) \\
&\vdots \\
1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{\text{two}} &= -2 \\
1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{\text{two}} &= -1
\end{align*}
\]

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 negating and adding 1 yields the number 5
Signed / Unsigned

- The hardware recognizes two formats:

  unsigned (corresponding to the C declaration \texttt{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 \texttt{signed int} or just \texttt{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
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:

\[
\text{slt} \quad \text{t0}, \text{t1}, \text{zero}
\]

and $t1$ contains the 32-bit number $1111\ 01\ldots\ 01$

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 \text{slt} or \text{sltu}

\[
\text{slt} \quad \text{t0, t1, zero} \quad \text{stores} \quad 1 \quad \text{in t0}
\]
\[
\text{sltu} \quad \text{t0, t1, zero} \quad \text{stores} \quad 0 \quad \text{in t0}
\]
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 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
Addition and Subtraction

- Addition is similar to decimal arithmetic

- For subtraction, simply add the negative number – hence, subtract A-B involves negating B’s bits, adding 1 and A

Source: H&P textbook
Overflows

• For an unsigned number, overflow happens when the last carry (1) cannot be accommodated

• For a signed number, overflow happens when the most significant bit is not the same as every bit to its left
  ▪ when the sum of two positive numbers is a negative result
  ▪ when the sum of two negative numbers is a positive result
  ▪ The sum of a positive and negative number will never overflow

• MIPS allows `addu` and `subu` instructions that work with unsigned integers and never flag an overflow – to detect the overflow, other instructions will have to be executed
Title

• Bullet