Lecture 7: MARS, Computer Arithmetic

- Today's topics:
  - MARS intro
  - Numerical representations

Full Example – Sort in C (pg. 133)

```c
void sort (int v[], int n)
{
    int i, j;
    for (i=0; i<n; i+=1) {
        for (j=i-1; j>=0 && v[j] > v[j+1]; j-=1) {
                swap (v,j);
        }
    }
}
```

```c
void swap (int v[], int k)
{
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

- Allocate registers to program variables
- Produce code for the program body
- Preserve registers across procedure invocations

The swap Procedure

- Register allocation: $a0 and $a1 for the two arguments, $t0 for the temp variable – no need for saves and restores as we're not using $s0-$s7 and this is a leaf procedure (won't need to re-use $a0 and $a1)

```
swap:   sll    $t1, $a1, 2
        add    $t1, $a0, $t1
        lw     $t0, 0($t1)
        lw     $t2, 4($t1)
        sw     $t2, 0($t1)
        sw     $t0, 4($t1)
        jr     $ra
```

```
void swap (int v[], int k)
{
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

The sort Procedure

- Register allocation: arguments v and n use $a0 and $a1, i and j use $s0 and $s1; must save $a0 and $a1 before calling the leaf procedure

- The outer for loop looks like this: (note the use of pseudo-instrs)

```
            move   $s0, $zero          # initialize the loop
loopbody1:  bge    $s0, $a1, exit1     # will eventually use slt and beq
            … body of inner loop …
            addi   $s0, $s0, 1
            j      loopbody1
exit1:
```

```
for (i=0; i<n; i+=1) {
  for (j=i-1; j>=0 && v[j] > v[j+1]; j-=1) {
        swap (v,j);
  }
}
```

The sort Procedure

- The inner for loop looks like this:

```
            addi   $s1, $s0, -1          # initialize the loop
loopbody2:  blt    $s1, $zero, exit2     # will eventually use slt and beq
            sll    $t1,  $s1, 2
            add    $t2, $a0, $t1
            lw     $t3, 0($t2)
            lw     $t4, 4($t2)
            bgt    $t3, $t4, exit2
            … body of inner loop …
            addi   $s1, $s1, -1
            j      loopbody2
exit2:
```

```
for (i=0; i<n; i+=1) {
  for (j=i-1; j>=0 && v[j] > v[j+1]; j-=1) {
        swap (v,j);
  }
}
```

Saves and Restores

- Since we repeatedly call “swap” with $a0 and $a1, we begin “sort” by copying its arguments into $s2 and $s3 – must update the rest of the code in “sort” to use $s2 and $s3 instead of $a0 and $a1

- Must save $ra at the start of “sort” because it will get over-written when we call “swap”

- Must also save $s0-$s3 so we don’t overwrite something that belongs to the procedure that called “sort”

Saves and Restores

```
sort:   addi    $sp, $sp, -20
        sw      $ra, 16($sp)
        sw      $s3, 12($sp)
        sw      $s2, 8($sp)
        sw      $s1, 4($sp)
        sw      $s0, 0($sp)
        move    $s2, $a0
        move    $s3, $a1
          …
        move    $a0, $s2        # the inner loop body starts here
        move    $a1, $s1
        jal     swap
          …
exit1:  lw      $s0, 0($sp)
          …
        addi    $sp, $sp, 20
        jr      $ra
```

9 lines of C code → 35 lines of assembly

MARS

- MARS is a simulator that reads in an assembly program and models its behavior on a MIPS processor
- Note that a “MIPS add instruction” will eventually be converted to an add instruction for the host computer’s architecture – this translation happens under the hood
- To simplify the programmer’s task, it accepts pseudo-instructions, large constants, constants in decimal/hex formats, labels, etc.
- The simulator allows us to inspect register/memory values to confirm that our program is behaving correctly

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
                              #  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

```
.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

Most significant bit — Least significant bit

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   ($2^{30} > 1$ 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

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

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

2's Complement

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

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

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

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

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

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

- 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 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

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

- 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

Title

- Bullet