Lecture 8: Number Crunching

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

HW 3 due
next Wed

slti

beq

Example 1 (pg. 98)

MOTIVATION FOR CALLER-CALLEE CONVENTION

```
int  leaf_example (int g, int h, int i, int j)
{
   int f ;
   f = (g + h) – (i + j);
   return f;
}
```

```
leaf_example:
   addi     $sp,  $sp,  -12
   sw       $t1, 8($sp)
   sw       $t0, 4($sp)
   sw       $s0, 0($sp)
   add      $t0, $a0, $a1
   add      $t1, $a2, $a3
   sub      $s0, $t0, $t1
   add      $v0, $s0, $zero
   lw       $s0, 0($sp)
   lw       $t0, 4($sp)
   lw       $t1, 8($sp)
   addi     $sp, $sp, 12
   jr       $ra
```

Notes:

In this example, the callee took care of saving the registers it needs.

The caller took care of saving its $ra and $a0-$a3.

Could have avoided using the stack altogether.

Example 2 (pg. 101)

① → Reversal
② pseudo instrs
③ Saves / Restores

```
int  fact  (int n)
{
   if (n < 1)  return (1);
       else return (n * fact(n-1));
}
```

if (condt)
then
else

Notes:
The caller saves $a0 and $ra
in its stack space.
Temp register $t0 is never saved.

br (condt) L1
else
L1: then

```
fact:
    slti    $t0, $a0, 1
    beq     $t0, $zero, L1
    addi    $v0, $zero, 1
    jr      $ra
L1:
    addi    $sp, $sp, -8
    sw      $ra, 4($sp)
    sw      $a0, 0($sp)
    addi    $a0, $a0, -1
    jal     fact
    lw      $a0, 0($sp)
    lw      $ra, 4($sp)
    addi    $sp, $sp, 8
    mul     $v0, $a0, $v0
    jr      $ra
```

bge $a0, 1, L1
return 1
ret n x fact
n x fact(n-1)

IA-32 Instruction Set

RISC vs CISC

Reduced (simple)

- Intel's IA-32 instruction set has evolved over 20 years – old features are preserved for software compatibility

Mult

Add

- Numerous complex instructions – complicates hardware design (Complex Instruction Set Computer – CISC)

MAC

CISC

- Instructions have different sizes, operands can be in registers or memory, only 8 general-purpose registers, one of the operands is over-written

μops

- RISC instructions are more amenable to high performance (clock speed and parallelism) – modern Intel processors convert IA-32 instructions into simpler micro-operations

CRISC

Large Constants

addi

16 b

32 b

- Immediate instructions can only specify 16-bit constants
- The lui instruction is used to store a 16-bit constant into the upper 16 bits of a register… combine this with an OR instruction to specify a 32-bit constant
- The destination PC-address in a conditional branch is specified as a 16-bit constant, relative to the current PC
- A jump (j) instruction can specify a 26-bit constant; if more bits are required, the jump-register (jr) instruction is used
- See green sheet!

Endian-ness

Two major formats for transferring values between registers and memory

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

lw $t0

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

$5+6$

$$\begin{array}{r} {}^{1}0101 \\ 0110 \\ \hline 1011 \end{array}$$

10

- The binary number

01011000 00010101 00101110 11100111

Most significant bit

Least significant bit

represents the quantity

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

-2B → +2B

- A 32-bit word can represent 2^{32} numbers between 0 and 2^{32}-1   4B

  … 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?

  NO

- 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

$-2B \longrightarrow +2B$

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

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 ~ 2B

1000 0000 0000 0000 0000 0000 0000 0000_{two} = -2^{31} -1 or -0 -2B
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 -2B -1

2's Complement

Signed
-2B to +2B

00 . . . . . 0 0 1
11 . . . . . 1 1 0

Carry in
$+2^{31}$

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 $1111_{two} = -1$

1 1 . . . . . 1 1
0 0 . . . . . 0 1 0
1 1 . . . . . 1 1 1
1 0 0 . . . . . 0 0 1

-2^{31}

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

$5 = 101$

$-5 = x' + 1$ = start with 5, invert bits, add 1

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

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

$x = 1111\ldots\ldots1$
$x' = 000\ldots\ldots0$
$11\ldots\ldots111$

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

$5 = 000101$

$-5 = x' + 1$

$111010 + 1$

$-5 = 111\cdots111011$

-6

$\boxed{111010}$

1

111011

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

-1

int {-2B to +2B}

signed int

- 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

4B

unsigned int {0 to +4B}

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

MIPS Instructions

int x;

set on less than $t1 < 0

Consider a comparison instruction:

slt $t0, $t1, $zero

and $t1 contains the 32-bit number 1111 01…01

-17

What gets stored in $t0?

$\boxed{\text{\$t0 set to 1}}$

compiler uses slt

lw $t1, addr of x

compiler uses sltu

unsigned int x;

$t0 set to 0

lw $t1, addr of x

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

... 0000101

1 1 1 1 1 1 1 1 1 1 ... 0

- 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 (Dismissed)

- 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