Tutorial: Harshad Numbers

In this tutorial, you will incrementally build a short MIPS program to introduce you to the following topics: *Arithmetic and Logical Expressions, Shifting, Number Notations (binary, hex, decimal), Conditionals and Loops, and Memory Accesses (load and store)*.

It will also show you how to incrementally develop, test, and optimize this program, using the MIPS ISA Simulator (MiSaSiM). It highlights the following features of MiSaSiM: *execution trace, navigation, register history, decimal/hexadecimal display, performance metrics (instruction count, register count).*

This tutorial assumes that you have read the Assembly Programming chapter under Readings on the ECE2035 website, and that you have installed MiSaSiM. Please refer to the User's Guide to make best use of MiSaSiM's features.

Introduction

What makes the numbers on the left side of the equations below interesting? (Hint: what's special about the first factor on the right side, relative to the number on the left side?)

$$42 = 6 * 7$$
$$133 = 7 * 19$$
$$216 = 9 * 24$$
$$2020 = 4 * 505$$
$$88263 = 27 * 3269$$

These are examples of Harshad numbers in base 10. The word “Harshad” in Sanskrit means “giver of joy.” Why? Because the number is divisible by the sum of its digits (e.g., 8+8+2+6+3 = 27).

In base 2, the Harshad numbers form the sequence:

1, 2, 4, 6, 8, 10, 12, 16, 18, 20, 21, 24, 32, 34, 36, …

For example, 21 = 10101 in binary. The sum of its digits is 3, which is a factor of 21.

You might be wondering whether your favorite number is a Harshad number.[1]

Let's write a MIPS program to determine whether a given positive number x is a Harshad number in base 2 (i.e., a 2-Harshad number). The basic idea is shown in Figure 1.

1 Sorry to dash your hopes, but the number 2035 is not a giver of joy in any of the common number bases.

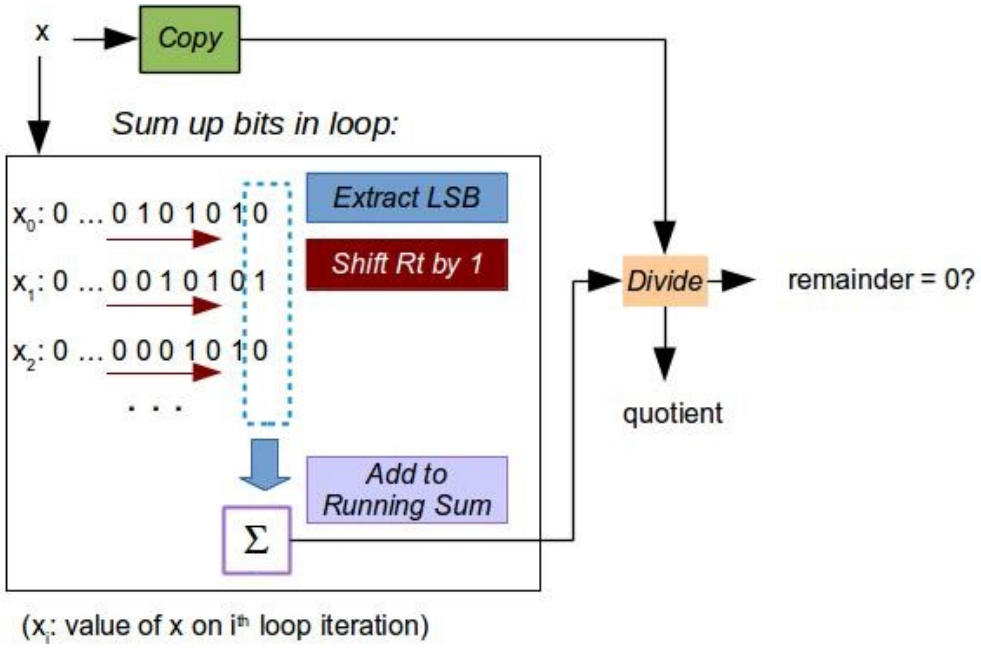


Figure 1: Basic algorithm: 1) save a copy, of x for use in step 3, 2) loop through the individual bits of x, keeping a running sum of each, and 3) finally divide the original value of x by the sum and check whether the remainder is 0. During each iteration of the loop in step 2, it extracts the least significant bit of x, adds it to the running sum, and shifts x right by one bit.

Input x as immediate value

To get started, we'll put a positive integer x in the register $1 using add immediate unsigned.

```
addiu $1, $0, 42  # set $1 to 42 (input to the program)
```

Since $0 always holds the value 0, this is a common way to initialize a register to a constant value. The constant is called an “immediate” because it is available immediately in the encoding of the instruction; no register or memory lookup is needed. (Note that “#” indicates the start of a comment.)

In this instruction, the immediate value is given as a decimal number. The assembler encodes this as a binary string of 32 bits (the word size of our datapath).

$1:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0

Looping through the bits of x

Since looping through the bits of x (in $1) requires iteratively modifying $1 by shifting it by 1 bit to the right each time, we will save a copy of x in $11 to use in the division later.

```
addiu $11, $1, 0   # save a copy of x in $11
```

To loop through the bits of x, we can write a loop that runs 32 times (assuming a word size of 32), with $2 as the loop counter (initialized to 0) and $3 the upper limit (32). The loop checks whether the counter has reached the limit yet (exit test). If so, it stops looping; otherwise it

executes the loop body, including incrementing the counter (update).

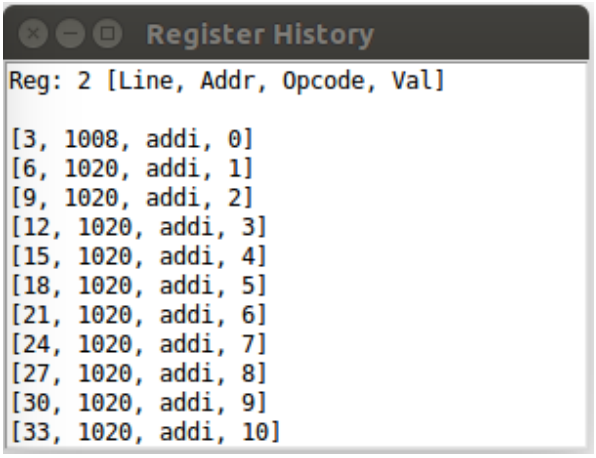
```
Joy2:   addiu $1, $0, 42          # set $1 to 42 (input to program)
        addiu $11, $1, 0          # save a copy of x in $11
        addi  $2, $0, 0           # initialize loop counter
        addi  $3, $0, 32          # set upper limit
Loop:   beq   $2, $3, Exit        # if counter reaches upper limit, exit loop
        …                         # loop body
        addi  $2, $2, 1           # update the loop counter
        j     Loop                # loop back
Exit:   jr    $31                 # return to caller
```

This code annotates some instructions with labels (e.g., “Loop” and “Exit”). These are used by the branching and jump instructions (BEQ and J) to indicate which instruction should be executed next. Branches are conditional jumps. In our example, BEQ will cause the program execution to jump to the instruction labeled “Exit” *if registers $2 and $3 hold equal values;* otherwise, the instruction following the BEQ will be executed. The jump instruction (J) unconditionally causes execution to jump back to the instruction labeled “Loop.”

[As an aside: You may be curious about what the instruction labeled “Exit” is doing. Here is a quick explanation: There is a label on the first instruction, “Joy2,” which can be used to call this program by jumping to it from the caller using a special jump instruction, called jal. This instruction keeps track of where to return: an instruction location in the caller, which it stores in register $31. Then, when the “Joy2” program finishes, it can jump to that location, using another type of jump instruction, called jr for “jump register.” This jumps to the address in $31 to return to the caller of the “Joy2” program.]

Load the code written so far into MiSaSiM [harshad-0.asm] to see what it does. When you hit the Execute or Forward button, an execution trace of the code will appear in the lower left pane. You can navigate through the code using the direction buttons and view the state of the registers at each point in the trace. Stepping through the execution trace, you can see that the loop is executed 32 times.

Click the register $2 to get a trace of its values which are consecutive integers from 0 to 32.



This trace has an entry for each time that register $2 is changed. Each entry tells where the changed happened: which line number in the dynamic trace and which instruction address. It also gives the operator of the instruction creating the new value as well as the new value itself.

In the bottom left pane, there are performance metrics, which show the number of *static instructions* (how many lines of code are in the text of your program – counting instructions only), the *dynamic instruction count* (how many instructions are in the execution trace), the number of registers used, and the number of words of memory used for data (in static space and in stack space).

```
static I= 8, dynamic I= 102, reg data= 4, static data= 0, stack data= 0
Jump: 32.4% Arith: 35.3% Branch: 32.4%
```

Challenge: Can you modify the loop to run 32 times, but with less than 102 dynamic instructions? Hint: The other branch instruction (BNE – branch if not equal) may be helpful.

Extract and sum up bits of x

Now, let's add code to the loop body that extracts the least significant bit (LSB) of x.

```
Joy2:   addiu $1, $0, 42          # set $1 to 42 (input to program)
        addiu $11, $1, 0          # save a copy of x in $11
        addi  $2, $0, 0           # initialize loop counter
        addi  $3, $0, 32          # set upper limit
Loop:   beq   $2, $3, Exit        # if counter reaches upper limit, exit loop
                                  # loop body:
        andi  $4, $1, 1           # extract LSB of x
        addi  $2, $2, 1           # update the loop counter
        j     Loop                # loop back
Exit:   jr    $31                 # return to caller
```

The andi instruction performs a bitwise logical AND of its source operands. The immediate "1" masks off all but the last bit of $1. For example, if $1 has the value shown below and is ANDed with 1, $4 will be given the value below (i.e., `andi $4, $1, 1`).

$1:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1	1	1
																	bitwise AND															
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
$4:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

Next, we'll create a running sum in register $5 by initializing it outside the loop to 0 and adding $4 to its current value on each loop iteration.

```
Joy2:  addiu $1, $0, 42          # set $1 to 42 (input to program)
       addiu $11, $1, 0          # save a copy of x in $11
       addi  $2, $0, 0           # initialize loop counter
       addi  $3, $0, 32          # set upper limit
       addi  $5, $0, 0           # initialize running sum to 0
Loop:  beq   $2, $3, Exit        # if counter reaches upper limit, exit loop
                                 # loop body:
       andi  $4, $1, 1           # extract LSB of x
       add   $5, $5, $4          # add it to the running sum
       addi  $2, $2, 1           # update the loop counter
       j     Loop                # loop back
Exit:  jr    $31                 # return to caller
```

Finally, we'll shift x right by 1 bit using shift right arithmetic (sra). (Note: we chose the arithmetic shift because $1 is holding a value that represents a number. However, since x is assumed to be positive, the sign that is preserved by sra is always 0. So, shift right logical would work as well in this case.)

```
Joy2:  addiu $1, $0, 42          # set $1 to 42 (input to program)
       addiu $11, $1, 0          # save a copy of x in $11
       addi  $2, $0, 0           # initialize loop counter
       addi  $3, $0, 32          # set upper limit
       addi  $5, $0, 0           # initialize running sum to 0
Loop:  beq   $2, $3, Exit        # if counter reaches upper limit, exit loop
                                 # loop body:
       andi  $4, $1, 1           # extract LSB of x
       add   $5, $5, $4          # add it to the running sum
       sra   $1, $1, 1           # shift x right by 1 bit
       addi  $2, $2, 1           # update the loop counter
       j     Loop                # loop back
Exit:  jr    $31                 # return to caller
```

Load this code into MiSaSiM [harshad-1.asm] and look at the register history for:

- $1: verify that it is taking on x, shifted by 1 on each iteration (note that this is the same as dividing x by 2 each time and keeping only the quotient whole number part);
- $4: verify that it is the individual bits of x from least significant to most significant.
- $5: check that it is the running sum we expect.

Notice that after the 6th iteration of the loop, none of these registers' values change. The program is doing a lot of wasted work after that! Its dynamic instruction count is 199.

Challenge: Can you rewrite the program to be more efficient? Hint: can we exit the loop earlier and not rely on the counter to control the loop? Try to get the dynamic count down to 35 or less.

(Here's one solution [harshad-2.asm] and another that uses the trick from the first challenge [harshad-3.asm].)

Check remainder of x/sum

Now, outside of the loop, we can add the final step shown in Figure 1: dividing the original value of x by the running sum and checking for a "0" remainder. MIPS has a division instruction (div) that divides a dividend (first operand) by a divisor (second operand) to get both a quotient and a remainder. It places the quotient and remainder in a special 64-bit HiLo register: the quotient is in the lower 32 bits (Lo) and the remainder is in the upper 32 bits (Hi). The Hi and Lo parts of this register are not directly accessible as source operands to any instruction in the MIPS ISA. Instead, the instructions `mfhi` and `mflo` transfer the Hi and Lo parts to a given register.

For example, we would like to divide the original value of x (in $11) by the running sum ($5). The fragment of code highlighted below does the division and then puts the remainder in $6 and the quotient in $7. (This is using the code from harshad-3.asm.)

```
Joy2:  addiu $1, $0, 42          # set $1 to 42 (input to program)
       addiu $11, $1, 0          # save a copy of x in $11
       addi  $5, $0, 0           # initialize running sum to 0
Loop:  andi  $4, $1, 1           # extract LSB of x
       add   $5, $5, $4          # add it to the running sum
       sra   $1, $1, 1           # shift x right by 1 bit
       bne   $1, $0, Loop        # if x still has non-zero bits, loop
Exit:  div   $11, $5             # divide original value of x by sum
       mfhi  $6                  # transfer the remainder to $6
       mflo  $7                  # transfer the quotient to $7

       ...                       # what goes here?

       jr    $31                 # return to caller
```

If the remainder is 0, x is a 2-Harshad number. Fill in the ". . ." with instructions that put 1 in $8 if x is a 2-Harshad number, otherwise put 0 in $8.

(One solution is given in [harshad-4.asm]. Note there is no need to use the `mflo` instruction to save the quotient in this code, so it is omitted in harshad-4.asm.)

Hexadecimal Immediates

We can test our program by providing alternative initializations of $1 in the first instruction (replacing 42). MiSaSiM also supports expressing immediates as hexadecimal numbers, indicated by preceding them with "0x". For example,

```
addiu $1, $0, 0x2A       # set $1 to 0x2A = 42₁₀ (input to the program)
```

places the same binary string in $1 as shown above, representing the decimal number 42.

Accessing Memory

Input data can also be placed in memory and loaded into a program. Memory can be treated as an array of data values, indexed by addresses. To read in a data value, an address must be provided.

In the MIPS ISA, memory is "byte-addressed" which means each byte of data is accessible through a unique address. The addresses of consecutive bytes are consecutive integers. MIPS load byte (lb/lbu) and store byte (sb/sbu) instructions allow a byte to be read/written, given an address. For example, `lbu $3, 2003($0)` reads in the byte of data that is found at memory location 2003 and places it in the lower 8 bits (least significant byte) of $3. The memory address is specified in a load/store instruction using a register (e.g., $0) and an immediate value (e.g., 2003): the register value is added to the immediate value to create the address. More details are given in Load/Store Byte Instructions.

When we are working with integers, we need to load and store data that is 4 bytes (32 bits) long. The address used to access the word – using a load word (lw) or store word (sw) instruction – is the address of the least significant byte of the word. For example, suppose we have the fragment of memory shown here:

Address	Data
2072:	42
2076:	378
2080:	8245
2084:	917

The word addressed by the address 2076 has 4 bytes in consecutive locations from 2076 to 2079. It is represented by this binary string:

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1	1	1	0	1	0
Byte 3								Byte 2								Byte 1								Byte 0							
Address: 2079								Address: 2078								Address: 2077								Address: 2076							

This is using the "little endian" byte encoding convention, which means that the least significant byte is stored in the smallest address (which is also the word address). The "big endian" convention stores the most significant byte in the smallest address. Some ISAs use one convention, some use the other. MIPS allows you to select which one. MiSaSiM supports the little endian convention for MIPS.

To load in the integer 378 located starting at address 2076, we can use the load word (lw) instruction, which adds the immediate value (2076) to the value of $0 and uses this as the address to look up in memory, placing the data value in $3:

```
lw $3, 2076($0)
```

The following fragments produce the same result of looking up the memory location 2076 and placing the value into $3.

```
addi $4, $0, 2000 # $4: 2000
lw $3, 76($4)     # $3: Memory(2000+76)
```

```
addi $4, $0, 2080 # $4: 2080
lw $3, -4($4)     # $3: Memory(2080-4)
```

Similarly, to write a word of data to memory, we can use the store word (sw) instruction:

```
sw $8, 2080($0)
```

Like lw, this instruction adds the immediate value (2080) to the second source register's value (0) to form the memory address. It then writes the value held by the first source register ($8) to this memory location.

Using Mnemonic Labels

Writing programs using absolute addresses (such as 2080) can be error-prone and it makes the program difficult to change (e.g., changing the size or locations of data requires changes to all the memory accesses). Instead, we'd like to allocate memory space and annotate it with mnemonic labels that can stand for addresses specifying where data is located. This allows the code and data to be relocatable and easy to change and maintain. We've already seen the use of labels when using branches and jumps – in the case of these control instructions, the labels refer to instruction addresses. Now, we are going to use labels to indicate data addresses.

To allocate words of data in a MIPS program, we use the *assembler directives* **`.data`** and **`.text`** to indicate what should be placed in memory as data versus as code. Each is placed on a single line before the region being defined. Here is an example:

```
.data
XLoc:  .word  42
ALoc:  .alloc 1

.text
Joy2:  lw    $1, XLoc($0)        # read input to program from memory into $1
       addiu $11, $1, 0          # save a copy of x in $11
       addi  $5, $0, 0           # initialize running sum to 0
Loop:  andi  $4, $1, 1           # extract LSB of x
       add   $5, $5, $4          # add it to the running sum
       sra   $1, $1, 1           # shift x right by 1 bit
       bne   $1, $0, Loop        # if x still has non-zero bits, loop
Exit:  div   $11, $5             # divide original value of x by sum
       mfhi  $6                  # transfer the remainder to $6
       beq   $6, $0, Yes         # is remainder = 0? (sum is a factor?)
       addi  $8, $0, 0           # if not, $8 = 0
       j     End                 # and jump to the End to store result (0)
Yes:   addi  $8, $0, 1           # if so, x is a Harshad: $8 = 1
End:     . . .                   # store result ($8) at memory location ALoc
       jr    $31                 # return to caller
```

Within the **.text** region, the code for the program is given. Within the **`.data`** region, data is allocated. In this region in our example, the assember directives **`.word`** and **`.alloc`** are also used. One word of data is allocated and initialized to 42 (using **`.word`**). The address of this data word is labeled **`XLoc`**. Another data word is allocated, but not initialized (using **`.alloc`** whose argument is a single integer indicating the number of words to allocate); its address is labeled **`ALoc`**. (Constraints on the use of labels and directives in MiSaSiM are given here.)

A label can be used anywhere that an immediate is used. The address that the label stands for is used as the immediate value when the instruction is executed. For example, to load the value stored at **`XLoc`** into register $1, we can use either of the following fragments:

`lw $1, XLoc($0)`	`addi $9, $0, XLoc` `lw $1, 0($9)`

The **`.alloc`** assembler directive is used to allocate space that is not initialized, usually because the program will compute some results that will be written to that space before that memory location is read. Try filling in the ". . ." in the box in **`Joy2`** above with a **`sw`** instruction that stores the result $8 to memory location labeled **`ALoc`**. (See [harshad-5.asm] for one answer.)