Project 1: MIPS programming with MARS

In this project, you will write a MIPS assembly program and verify that it runs correctly with the simulator MARS (http://courses.missouristate.edu/KenVollmar/mars/). This program consists of two parts (A) and (B) shown in next page, and will be used for subsequent projects.

Restrictions: you are only allowed to use the following listed instructions.

Instr name	example	functionality
add	add $1,$2,$3	$1 = $2 + $3
add immediate	addi $1,$2,100	$1 = $2 + 100
add (unsigned)	addu $1,$2,$3	$1 = $2 + $3 no overflow
subtract	sub $1,$2,$3	$1 = $2 - $3
set if less than	slt $1, $2, $3	if ($2 < $3), $1 = 1 else $1 = 0
set if less than immediate	slti $1, $2, 100	if ($2 < 100), $1 = 1 else $1 = 0
set if less than (unsigned)	sltu $1, $2, $3	assuming unsigned: if ($2 < $3), $1 = 1 else $1 = 0
load upper immediate	lui $1,100	$1 = 100 x 2^{16}
and	and $1,$2,$3	$1 = $2 & $3
and immediate	andi $1,$2,100	$1 = $2 & 100
or	or $1,$2,$3	$1 = $2 | $3
or immediate	ori $1,$2,100	$1 = $2 | 100
xor	xor $1, $2, $3	$1 = $2 ⊕ $3
xor immediate	xori $1, $2, 255	$1 = $2 ⊕ 255
shift left logical	sll $1,$2, 5	$1 = $2 << 5 (logical)
shift right logical	srl $1,$2, 5	$1 = $2 >> 5 (logical)
load word	lw $1, 1000($2)	$1 = memory [$2+1000]
store word	sw $1, 1000($2)	memory [$2+1000] = $1
branch if equal	beq $1,$2,target	if ($1 = $2), branch to target
branch if not equal	bne $1,$2,target	if ($1 ≠ $2), branch to target
jump	j target	(unconditionally) jump to target

- **Specifically, multiplication and division instructions are NOT allowed.**
- **For registers, you are only allowed to use `$0, $8 - $23`. If you are running out of registers to plan your programming, you can use memory locations from `[0x2000, 0x3000)`**
- **Any other instructions or registers need to get approval from the instructor.**

A) "PRPG with Middle-Square"

A Pseudo-Random Pattern Generator (PRPG) can produce (from a seed S_0) a sequence of seemingly random numbers (or, as binary patterns), S_1, S_2, S_3, …. In fact, the sequence will be entirely predictable and repeatable (thus "pseudo-random"), and here we'll explore a strategy proposed by John Von Neumann called "middle-square method":

a) given S_0 which is a 16-bit number, compute the square of S_0^2, which is a 32-bit number
b) derive S_1 by dropping the middle 16 bits of S_0^2, thus S_1 is a 16-bit number again

…continue from S_i to derive S_{i+1} by repeating the above process.

Specifics:

i. Your program must begin with this line of code: **`addi $8, $0, ____,`** so we can use this (and therefore $8) to initialize S_0 to any number.
ii. Your program must produce S_0 , S_1 , S_2 , S_3 , S_4 **...** , S_{15} and write them into memory address `0x2010, 0x2014, 0x2018, 0x201C`, …

B) Randomness assessment

How good is a sequence of pseudo-random patterns? There are many ways to assess the "randomness", but here we will focus on the following:

- **Hamming Weight**: a "good" random pattern should have roughly equal 1's and 0's
- **Hamming Distance:** a "good" pair of random patterns should have of about half the bits being different.

Specifics:

i. For S_0 to S_{15}, your program should calculate the "average Hamming Weight" of these 16 patterns, and store the result to memory location `0x2000`.
ii. For S_0 to S_{15}, your program should calculate the "average Neighbor Hamming Distance", where the average of 16 Hamming Distances are measured: between all the neighboring S_i and S_{i+1} , plus between S_{15} and S_0, and store the result to memory location `0x2004`
iii. For S_0 to S_{15}, your program should calculate the "average Pairwise Hamming Distance", where the Hamming Distances of all possible pairs among the 16 patterns are measured, and the average of them is stored back into memory location `0x2008.`

Note: Here, we only need an approximation of the average with integer precision, so you should not need to go for fractional numbers.

Levels of completion for the program part:

1) (80% grade) Your code should achieve part A) and part B) i).
2) (100% grade) Your code should achieve part A) and part B) i), ii).
3) (10% extra credit) will be given if your code achieves B) iii)

Overall Grade:

You will be graded based on the report quality (40%) and functionality of your program (60%).

1. (40% grade) Answer the following questions in your report:
   a) Which level does your program achieve? How many hours have you spend in total on this project? What was the most difficult (or time-consuming) part of this project?
   b) List out all the instructions and registers that your program uses. If given a chance to start over, how would you have done differently?
   c) How does your program compute the square of a number without using the multiplication instruction? How does your program compute the Hamming Weight of a number? How does your program compute the average without using the division instruction?
   d) Overall, do you think this version of middle-square method is a good PRPG mechanism? Why? What can you propose to make a better one?

2. (60% grade) Program functionality
   - We will test your program with various cases.
   - In addition, you should provide screenshots of your program's results, and total instruction count (see MARS menu -> Tools -> Instruction Statistics: Total), for the following cases:
     a) S0 = 3
     b) S0 = 35
     c) S0 = 167
     d) S0 = 397
     e) S0 = 2019
     f) S0 = 65533

(extra credit will be given if your code achieves a very low dynamical instruction count)

Your submission (on Bb) must include two files:

- One netid_366_p1_report.PDF file, including the following:
  i) Answers & screenshots required from above.
  ii) your assembly code (yes, paste your code in the pdf report too so we can read through your code – missing it will result in -5 points.)
- One netid_366_p1.asm which is your MIPS assembly code (so we can run your code on MARS – missing it will result in -5 points.).