CSE 101: Introduction to Computational and Algorithmic Thinking

Stony Brook University

Lab Assignment #12

Spring 2018

Assignment Due: April 27, 2018 by 11:59 pm

Assignment Objectives

This lab will review the basics of bitwise operations.

Getting Started

Visit Piazza and download the "bare bones" file `lab12.py` onto your computer. Open `lab12.py` in PyCharm and fill in the following information at the top:

1. your first and last name as they appear in Blackboard
2. your Net ID (e.g., jsmith)
3. your Stony Brook ID # (e.g., 111999999)
4. the course number (CSE 101)
5. the assignment name and number (Lab #12)

Submit your final `lab12.py` file to Blackboard by the due date and time. Late work will not be graded. Code that crashes and cannot be graded will earn no credit.

Part I: Extracting Bits (10 points)

Write a function `extract_bits_3_9()`, which takes a single integer argument named `num`. The function extracts bits 3 through 9, returning them as an integer. Remember that the rightmost bit is bit #0.

As an example, suppose `num` is 1,521,342. In binary, this value is:

`101110011011010111110`

Let's highlight bits 3 through 9, inclusive:

`10111001101`**`1010111`**`110`

The function needs to perform various masking and/or shifting operations as needed so that the function will return:

`1010111`

The above number written in decimal is 87, which is what the function would ultimately return.

Remember that all data in a computer is stored in binary format, so it is not necessary for you to tell the computer to explicitly convert between base 10 and base 2. All integers are represented in binary, but will be printed in decimal to the screen.

You might find a base conversion website helpful while completing this assignment. Many such websites can be found with a quick Google search.

Examples:

The function arguments in the examples below are expressed in binary (denoted by the leading `0b`), making it easier to understand the returned result.

Function Call	**Return Value**
`extract_bits_3_9(0b1110011000100110`**`1010010`**`000)`	`0b1010010`
`extract_bits_3_9(0b1011110001000111110111100111)`	`0b111100`
`extract_bits_3_9(0b101001110111110010110011000)`	`0b110011`
`extract_bits_3_9(0b1100101111111100001111100)`	`0b1111`
`extract_bits_3_9(0b101100101001110100011010)`	`0b100011`
`extract_bits_3_9(0b10010100111101)`	`0b100111`
`extract_bits_3_9(0b11011000001)`	`0b1011000`
`extract_bits_3_9(0b1001100100000)`	`0b1100100`
`extract_bits_3_9(0b1000011111011)`	`0b11111`
`extract_bits_3_9(0b1111011111100)`	`0b1011111`

Part II: Setting Bits (10 points)

Write a function `set_bits()`, which takes two arguments, in this order:

1. `num`: a positive integer
2. `bits_to_set`: a list of non-negative integers (i.e., positive integers and possibly zero). Each integer is guaranteed to be less than the number of bits in the binary representation of `num`.

The function iterates over the `bits_to_set` list, treating each value in the list as a bit position in the binary representation of `num`. The function then sets a 1 at each such bit position in `num`.

To make this a little easier to understand, suppose that `num` is `1101101001101` and `bits_to_set` is `[0, 2, 4, 7]`. This means we want to set the bits at positions 0, 2, 4 and 7 in `num` to 1. Therefore,

```
1101101001100
     *  * * *
```

is changed to:

```
1101111011101
     *  * * *
```

where the asterisks indicate the four bit positions we are attempting to set.

Hint: you will need to use both shifting and masking operations to solve this problem. Consider: suppose we started with the number `1` in binary and shifted it to the left by 5 positions. We would then have `100000`. Presumably this mask would somehow help us to set the 5^{th} bit of some number to 1.

Examples:

The function arguments in the examples below are expressed in binary (denoted by the leading `0b`), making it easier to understand the returned result.

Function Call	**Return Value**
set_bits(0b1**1**100**1**100**0**1010**0**, [0, 9, 13, 5])	0b1**1**100**1**100**1**1010**1**
set_bits(0b11011000000101, [10, 4, 7])	0b11011010010101
set_bits(0b10000100101100100, [0, 3, 5, 6, 15, 16])	0b11000100101101101
set_bits(0b10011100100110011, [0, 1])	0b10011100100110011
set_bits(0b1010011110110011, [10, 13])	0b1010011110110011
set_bits(0b1001011010010, [0, 10, 11, 5])	0b1111011110011
set_bits(0b1011100010011010, [9, 6])	0b1011101011011010
set_bits(0b1111000100100, [3])	0b1111000101100
set_bits(0b101010110000000, [2])	0b101010110000100
set_bits(0b1110111001001111, [2, 7, 12, 14, 15])	0b1111111011001111

Part III: Decode a Machine Instruction (20 points)

The Fictional KIPS Computer

For this part you will implement a function that decodes *machine language* instructions of a fictional computer we will call the *KIPS*. You should read through the Unit 9 lecture notes before starting this part. KIPS supports the following instructions that the CPU can execute:

- `add dest, op1, op2`: Performs an addition. `dest` is the desination register where the sum of registers `op1` and `op2` will be saved. (`dest = op1 + op2`)
- `sub dest, op1, op2`: Performs a subtraction. `dest` is the desination register where the difference of registers `op1` and `op2` will be saved. (`dest = op1 - op2`)
- `mul dest, op1, op2`: Performs a multiplication. `dest` is the desination register where the product of registers `op1` and `op2` will be saved. (`dest = op1 * op2`)
- `div dest, op1, op2`: Performs an integer division. `dest` is the desination register where the quotient of registers `op1` and `op2` will be saved. (`dest = op1 // op2`)
- `li dest, immediate`: Stores the value `immediate` in register `dest`. (i.e.,`dest = immediate`) Recall from lecture that an *immediate* value is simply a constant.

As with the real MIPS computer, the KIPS computer has 10 registers numbered `$t0`, `$t1`, ..., `$t9`. In the above list of instructions, `dest`, `op1` and `op2` are registers. `immediate` is a positive integer (or zero).

Instruction Formats

Every KIPS instruction is exactly 32 bits in length. An *arithmetical* instruction's bits are divided up as follows.

Note that bit #0 is the rightmost bit:

Opcode	**Destination Register**	**First Operand**	**Second Operand**
Bits 24–31	Bits 16–23	Bits 8–15	Bits 0–7

Let's look at an example instruction to understand this format a little better:

```
00000011000010010000010000000010
```

First we divide it into its constituent parts, each of which is one byte, notice:

```
00000011 00001001 00000100 00000010
```

This instruction can be expressed more concisely in hexadecimal:

```
03090402
```

The leftmost byte, corresponding with bits 24–31, equals 3_{10}; the next byte (bits 16–23) equals 9_{10}; the next byte (bits 8–15) equals 4_{10}; and the rightmost byte (bits 0–7) equals 2_{10}.

The leftmost byte gives the *opcode*, which is a number in the range 0-4 that indicates which instruction we are executing:

Opcode	**Instruction**
0	addition
1	subtraction
2	multiplication
3	division
4	load immediate

The number 3 in our sample instruction indicates we are performing a division operation.

The rightmost three bytes, in order from left to right, tell us the destination register (where the result will be saved) and the two registers where we can find the operands. In this example, the destination register is `$t9`, and the two operands are `$t4` and `$t2`. Thus, our instruction:

```
00000011 00001001 00000100 00000010
```

really means this:

```
$t9 = $t4 // $t2
```

An `li` instruction has only three parts, instead of four:

Opcode	**Destination Register**	**Immediate Value**
Bits 24–31	Bits 16–23	Bits 0–15

The following example instruction:

```
00000101 00000110 0000100010011011
```

really means this:

```
li $t6, 2203
```

because $0000100010011011_2 = 2203_{10}$.

Write a function `decode_instruction()` that takes a single integer argument, `inst`, that represents a KIPS machine instruction. The function divides the instruction into its constituent parts and returns them in a tuple. In the case of an arithmetical operation, the tuple returned must have this format:

```
(opcode, dest, op1, op2)
```

and for an `li` instruction it will have this format:

```
(opcode, dest, immediate)
```

All values in the returned tuple are integers.

You will need to use bitwise operations, including shifting and masking, to extract the parts of the instruction.

As an example of bitswise operations, let's suppose for some reason you wanted to extract bits 21–27 of an integer stored in variable `x`. Keep in mind that the bits are numbered 0–31, with bit #0 on the right. One way to do this would be as follows:

```
mask = 0b00001111111000000000000000000000   # 0b indicates a binary value
x = x & mask
x = x >> 21
```

Use this as a model to extact the three or four parts of an instruction (as appropriate) and return the required tuple.

If any of the values `opcode`, `dest`, `op1` or `op2` is invalid, the function should return the tuple `(-1, -1, -1, -1)`. Using the information above, you should be to figure out what are the valid ranges of these values.

Examples:

The function arguments in the examples below are expressed in hexadecimal (denoted by the leading `0x`), making it easier to see the values of the instruction.

Function Call	**Return Value**
`decode_instruction(0x01050000)`	`(1, 5, 0, 0)`
`decode_instruction(0x00040407)`	`(0, 4, 4, 7)`
`decode_instruction(0x01010207)`	`(1, 1, 2, 7)`
`decode_instruction(0x00090400)`	`(0, 9, 4, 0)`
`decode_instruction(0x00090005)`	`(0, 9, 0, 5)`
`decode_instruction(0x01050600)`	`(1, 5, 6, 0)`
`decode_instruction(0x04060614)`	`(4, 6, 1556)`
`decode_instruction(0x02000c05)`	`(-1, -1, -1, -1)`
`decode_instruction(0x07020403)`	`(-1, -1, -1, -1)`
`decode_instruction(0x04110752)`	`(-1, -1, -1, -1)`

How to Submit Your Work for Grading

To submit your `.py` file for grading:

1. Login to Blackboard and locate the course account for CSE 101.
2. Click on "Assignments" in the left-hand menu and find the link for this assignment.
3. Click on the link for this assignment.
4. Click the "Browse My Computer" button and locate the `.py` file you wish to submit. Submit only that one `.py` file.
5. Click the "Submit" button to submit your work for grading.

Oops, I messed up and I need to resubmit a file!

No worries! Just follow the above directions again. We will grade only your last submission.