CSE 220: Systems Fundamentals I

Stony Brook University

Programming Assignment #2

Spring 2021

Assignment Due: Sunday, Mar 21st, 11:59 PM (EST)

Corrections:

03/05/2021: Fixed typo in last test case of `is_stack_empty`
03/07/2021: Clarified test case examples in `stack_pop`

Learning Outcomes

After completion of this programming project you should be able to:

- Implement non-trivial algorithms that require conditional execution and iteration.
- Create, read, and write one-dimensional arrays of arbitrary length.
- Design and implement functions that implement the MIPS assembly function calling conventions.

Getting Started

Visit the course website and download MarsSpring2021.jar. From Blackboard, download the starter code (**hw2.zip**) accompanying this document.

Inside **hw2.zip** you will find two files called **hw2.asm** and **hw2-funcs.asm**. The file called **hw2.asm** is the main file that is used to define global variables, verify input arguments provided to the program and includes **hw2-funcs.asm**, which contains the main logic. You should not change anything in **hw2.asm**. The file **hw2-funcs.asm** contains several function stubs, which you will need to implement. These stubs contain only `jr $ra` instructions. Your job in this assignment is to implement all the functions as specified below. Do not change the function names since the grading scripts will be looking for functions of the given names. However, you may implement additional helper functions of your own, and add them to **hw2-funcs.asm**.

If you are having difficulty implementing these functions, write out pseudocode or implement the functions in a higher-level language first. Once you understand the algorithm and what steps to perform, then translate the logic into MIPS assembly code. It may seem like this approach takes more time. But once you have the implementation details pinned down in a higher-level language, then translating it to assembly is less error prone. This will help save time and errors.

Be sure to initialize all of your values (e.g., registers) within your functions. Never assume registers or memory will hold any particular values (e.g., zero). MARS initializes all of the registers and bytes of main memory to zeroes. The grading scripts will fill the registers and/or main memory with random values before calling your functions.

IMPORTANT: Do not define a `.data` section in your hw2-funcs.asm file. A submission that contains a `.data` section will most likely not integrate with our grading script. This may lead to you getting no credit.

Important Information about CSE 220 Programming Projects

- Read this document carefully. If you cannot understand something, ask for clarification . The course staff will try and answer those questions proactively. However, questions whose answers are clearly stated in the documents will be given lowest priority by the course staff.
- You must use the Stony Brook version of MARS posted on the course website. Do not use the version of MARS posted on the official MARS website. The Stony Brook version has a reduced instruction set, added tools, and additional system calls you might need to complete the homework assignments.
- When writing assembly code, try to stay consistent with your formatting and to comment as much as possible. This will make your code readable, which will help us grade your code and also assist you better if you get stuck.
- Do not copy or share code. Your submissions will be checked against other submissions from this semester and from previous semesters.
- Submit to Blackboard by the due date and time. Late work will be penalized as described in the course syllabus. Code that crashes and cannot be graded will earn no credit. No changes to your submission will be permitted once the deadline has passed.

How Your CSE 220 Assignments Will Be Graded

Your programming assignments will be graded almost entirely in an automated way. Grading scripts will execute your code with input values (e.g., command-line arguments, function arguments) and will check for expected results (e.g., print-outs, return values, etc.) For this assignment, your program will be generating output and your functions will be returning values that will be checked for exact matches by the grading scripts. It is your responsibility to output/return the expected values.

Some other items you should be aware of:

- Each test case must execute in 20,000 instructions or fewer. Efficiency is an important aspect of programming. This maximum instruction count will be increased in cases where a complicated algorithm might be necessary, or a large data structure must be traversed. To find the instruction count of your code in MARS, go to the **Tools** menu and select **Instruction Statistics**. Press the button marked **Connect to MIPS**. Then assemble and run your code as normal.

  If you are using the command line to assemble and run your code, you can use the following command:

  ```
  $ java -jar /path/to/MarsSpring2021.jar /path/to.asm --argv <args> --noGui -i -q
  ```

  The path separators will be '\' instead of '/' if you are on a Windows machine.

- Any excess output from your program (debugging notes, etc.) will impact grading. Do not leave erroneous print-outs in your code.
- We will provide you with a small set of test cases for each assignment to give you a sense of how your work will be graded. It is your responsibility to test your code thoroughly by creating your own test cases.
- The testing framework we use for grading your work will not be released, but the test cases and expected results used for testing will be released.

Register Conventions

You must follow the register conventions discussed in lecture and recitation. You can also review the register conventions from this illustrative MIPS register conventions guide by Prof. McDonnell. Failure to follow them will result in loss of credit when we grade your work. Here is a brief summary of the register conventions and how your use of them will impact grading:

- It is the callee's responsibility to save any $s registers it overwrites by saving copies of those registers on the stack and restoring them before returning.
- If a function calls a secondary function, the caller must save $ra before calling the callee. In addition, if the caller wants a particular $a, $t or $v register's value to be preserved across the secondary function call, then place a copy of that register in an $s register before making the function call.
- A function which allocates stack space by adjusting $sp must restore $sp to its original value before returning.
- There is no reason to modify the registers $fp and $gp in this assignment. So, try not to change them. However, if a function modifies one or both, the function must restore them before returning to the caller.

Note that you will lose points, if you do not follow register conventions. Grading scripts will check to see your code follows the conventions.

Unit-Testing Functions

To test your implemented functions, you may use the files ending with `_test.asm` in the starter code (**hw2.zip**) provided to you. You should add your own test code to these files before assembling and running them.

Each test file contains initial code to help you hook/call into the functions that you are supposed to implement in *hw2-funcs.asm*. You will need to add code to call the appropriate functions with appropriate arguments. The astute reader may also write a script to automatically run all the test files with the necessary test cases. This is of course not a requirement and should not be attempted unless you are comfortable with scripting. Your focus should be on first implementing the expected functionality. Your submission will be graded using the examples provided in this document and additional test cases. Do not submit your test files. They will be deleted. We will use different test files for grading.

Interpreting Expressions

In this assignment, we will develop a basic interpreter to evaluate `AExp`. A valid `AExp` is a particular kind of arithmetic expression that evaluates to an integer. The only operands in an `AExp` are integers. The only arithmetic operators allowed in an `AExp` are addition (`+`), subtraction (`-`), multiplication (`*`), and integer or floor division (`/`). Parenthesis `(` and `)` are also allowed in an `AExp`. Operators in an `AExp` have a pre-defined precedence. Parenthesis have the highest precedence. This means that an `AExp` within parenthesis will be evaluated first. The multiplication and division operators have a lower precedence than parenthesis but a higher precedence than addition and subtraction. As an example, consider the `AExp` "`(1+2)*3`". This will evaluate to the integer 9 since parenthesis have the highest precedence. Hence, 3 is multiplied with the result of the inner `AExp (1+2)`. However, if the `AExp` was "`1+2*3`", then the result would be 7 because the multiplication operator has higher precedence than the addition operator. All the operators are left associative, that is, when two operators have the same precedence, then the one occurring first from the left will be evaluated first. For example, in the `AExp` "`1+2-3`" `1+2` will be evaluated first and then 3 will be subtracted from its result.

There are many ways of implementing an interpreter for such expressions. One algorithm uses the two-stack technique. Briefly, the rules of the algorithm are as follows:

1. Maintain two stacks -- a value stack and an operator stack.
2. Iterate over the expression, character-by-character.
3. If we encounter a number, then we push the number into the value stack
4. If we encounter an operator (+,-,*,/), then we push it into the operator stack. However, if the operator stack contains operators greater than or equal to the operator we are inspecting, then for every operator, pop the operator, pop twice from the value stack, and apply the operator to the values from the stack. The result should be pushed back into the value stack.
5. If we encounter the open parenthesis character, we push it onto the operator stack
6. If we encounter the close parenthesis character, we pop from the operator stack until we pop the open parenthesis. Every time we pop from the operator stack, we pop two values from the value stack and apply the operator. The result should be pushed back into the value stack every time.
7. If we encounter an unexpected character, then we should display an error message. See `BadToken` label in **hw2.asm** for the message content.

Note that a stack is an array like data structure with two main operations. The elements in a stack are stored in a last-in-first-out (LIFO) fashion. It has two major operations, push and pop. The push operation is used to store an element at the top of the stack and push the existing elements down by 1. The pop operation is used to retrieve and remove the top element from the stack.

We will implement this interpreter by implementing the following functions:

Part1: Verify Digits

```
is_digit(char c)
```

This function takes a character as argument and returns 1 if the character is a digit [0-9]; and 0 otherwise.

Returns:

- 0 or 1 in `$v0`

Sample Test Cases:

- `is_digit('1')  -> 1`
- `is_digit('x')  -> 0`

Part2: Verify Operators

```
valid_ops(char c)
```

This function takes a character as argument and returns 1 if the character is a binary operator {+,-,*,/}; 0 otherwise.

Returns:

- 0 or 1 in `$v0`

Sample Test Cases:

- `valid_ops('+')  -> 1`
- `valid_ops('^')  -> 0`

Part3: Operator Precedence

```
op_precedence(char c)
```

This function takes a valid operator (+,-,*,/) as an argument and returns an integer associated with the precedence of the operator. The integer associated with a valid operator should conform to the precedence level of an operator as defined in the section *Interpreting Expressions*. Higher integer values mean higher precedence. Equal values mean similar precedence.

If the operator provided as argument is an invalid operator, then print any error message.

Returns:

- precedence in `$v0`

Sample Test Cases:

- `op_precedence('+')   -> n1`
- `op_precedence('*')   -> n2`
  `n1 < n2, where n1 and n2 are integers`
- `op_precedence('^')   -> Any error message`

Part4: Binary Operations

```
apply_bop(int v1, char op, int v2)
```

This function takes two (positive or negative) integers v1 and v2 and a valid operator as arguments. It returns the integer value that results from applying the operator to the values v1 and v2. You should assume that for every valid operation, the first operand is v1 and the second operand is v2. Operators '+' and '-' behave like arithmetic operators addition and subtraction. For multiplication, assume that the result will be the lower 32 bits of the product. The upper 32 bits should be ignored. The operator '/' indicates *floor division*. The result of floor division is the smallest integer obtained by dividing two numbers. For example, floor dividing 3 by 2 is 1.

Similarly, floor dividing -1 by 2 is -1. Floor division by 0 should result in an error message in the label `ApplyOpError` (see **hw2.asm**).

Returns:

- Result of binary operation in `$v0`

Sample Test Cases:

- `apply_bop(1,'+',2)  -> 3`
- `apply_bop(3,'-',5)  -> -2`
- `apply_bop(1,'*',6)  -> 6`
- `apply_bop(2,'/',3)  -> 0`

Part5: **Stack Push Operation**

```
stack_push(int x, int tp, int* addr)
```

This function takes three arguments. The first argument, *x*, denotes the element that needs to be pushed into the stack. The second argument, *tp*, indicates the top of the stack. The third argument, *addr*, denotes the *base address* of a stack. The data type *int** is borrowed from C and has no significance in MIPS. It is used to denote an address in this document and is used strictly for understanding purposes. The function should store element x at the top of the stack. After storing, the function should update the top of the stack and return the new top of the stack back to the caller.

Assumptions:

- The caller should provide the correct tp that can be used to store an element at the top of the stack.
- It is the caller's responsibility to keep track of the top of the stack after the function returns.
- The stack grows by incrementing the top of stack.
- The size of an element in the stack is 4.
- The stack will not be more than 500 elements. If the stack limit is reached, then terminate the program with any error message.

Returns:

- New top of the stack in `$v0`

Sample Test Cases:

- `stack_push(1,0,<stack_base_addr>)`
  `-> $v0=4 and 1 is at tp=0`
- `stack_push(1,4,<stack_base_addr>)`
  `-> $v0=8 and 1 is at tp=4`

Part6: Stack Pop Operation

`stack_pop(int tp, int* addr)`

This function takes two arguments. The first argument, *tp*, indicates the top of the stack. The second argument, *addr*, denotes the *base address* of a stack. The function should return the element at the top of the stack. Additionally, the function should return the top of the stack back to the caller.

An element cannot be popped from an empty stack. Popping an element from an empty stack should lead to program termination with any error message.

Assumptions:

- The caller should provide the correct *tp* that can be used to pop an element at the top of the stack.
- It is the caller's responsibility to keep track of the top of the stack after the function returns.
- The stack shrinks by decrementing the top of the stack.
- The size of each element in the stack is 4.

Returns:

- New top of the stack in `$v0`
- Value at top of the stack in `$v1`

Sample Test Cases:

- `stack_pop(0,<stack_base_addr>)`
  `-> $v0=0, $v1=10; assuming 10 was at tp=0`
- `stack_pop(4,<stack_base_addr>)`
- `-> $v0=4, $v1=20; assuming 20 was at tp=4`

Part7: Stack Peek Operation

`stack_peek(int tp, int* addr)`

This function takes two arguments. The first argument, *tp*, indicates the top of the stack. The second argument, *addr*, denotes the *base address* of a stack. The function should return the element at the top of the stack.

Peeking from an empty stack is not allowed. it should lead to program termination with any error message.

Valid Assumptions:

- The caller should provide the correct *tp* that can be used to read an element at the top of the stack.

Returns:

- Value at top of the stack in `$v0`

Sample Test Cases:

- `stack_peek(0,<stack_base_addr>)`
  `-> $v0=10; assuming 10 was at tp=0`
- `stack_peek(4,<stack_base_addr>)`
- `-> $v0=20; assuming 20 was at tp=4`

Part8: Verify Empty Stack

`is_stack_empty(int tp)`

This function takes one argument, *tp*, which indicates the top of the stack. The function should return 1 if the stack is empty; otherwise, 0.

Valid Assumptions:

- The initial stack pointer is 0.
- The caller should provide the correct `tp`.
- A stack grows by incrementing the top of stack.
- A stack shrinks by decrementing the top of stack.

Returns:

- 0 or 1 in `$v0`

Sample Test Cases:

- `is_stack_empty(10)`
  `-> $v0=0`
- `is_stack_empty(0)`
  `-> $v0=0`
- `is_stack_empty(-4)`
  `-> $v0=1`

Note*: For all stack operations, we are assuming that the memory addresses are word aligned.*

Part9: Evaluate an AExp

`eval(string aexp)`

This function takes an *AExp* expressed as a null-terminated string and prints its integer value after parsing and interpreting it. The function should print the message in `BadToken` (in **hw2.asm**) if it contains one or more invalid characters. The function should print the message in `ParseError` (in **hw2.asm**) if the expression is not a well formed expression as per the rules of

a valid *AExp*. For example, the expression `"2+-3"` contains valid characters but is an ill-formed expression. You can assume that the string will not contain any spaces before, after, or in between characters.

The function takes one argument:

- `aexp`: the starting address of a null-terminated string.

Returns:

- Nothing.

Sample Test Cases:

- `eval("1+2-3")` `-> 0`
- `eval("1+2*3")` `-> 7`
- `eval("(2-3)/5*25")` `-> -25`
- `eval("")` `-> "Ill Formed Expression"`
- `eval("2+-3")` `-> "Ill Formed Expression"`
- `eval(2^3)` `-> "Unrecognized Token"`

How to Submit Your Work for Grading

Create an archive **hw2.zip**. It should contain the following:

1. **hw2.asm**. This should be the same as the one provided with the starter code.
2. **hw2-funcs.asm**. This file will contain all your function implementations. Remember that this file should not have a *data* section.
3. Do not include the test files. They will be deleted before grading if found.

Go to **Assignments->Assignment 2** on **Blackboard** and submit **hw2.zip**.

You can submit multiple times. We will grade your most recent submission before the due date.