CMSC 216 Project #9 Spring 2022

Date due: Tuesday, May 3, 11:59:59 p.m.

1 Introduction

In this project you will be writing three MIPS assembly language programs. Remember that information about MIPS is on the Administrative and resources page on ELMS. We demonstrated running the `QtSpim` graphical simulator in class. If your account is set up right you should be able to run it without having to do anything special or different, but you have to be using an X server to run the graphical simulator.

Your task will be to take some C programs we give you and write assembly programs that are equivalent to them. Read the requirements in Section 3 **carefully** before coding, and again after finishing the project. Failure to follow these requirements will result in losing **significant credit**, so be sure to read them carefully. Ask in advance if you have questions.

There are two assembly homeworks on ELMS: Homework #9 is on basic assembly, and Homework #10 is on functions in assembly. If you do these homeworks and understand the answers, you will have already gotten practice with every assembly language feature that this project requires. If you come to the TAs' office hours to ask any questions about writing assembly, you **must** have already done these homeworks. The TAs will **not** answer any questions about the project unless you can show them that you have first done the assembly homeworks. (Of course if you need help with the homeworks you can ask in office hours.)

Due to the size of the course it is not feasible for us to be able to provide project information or help via email/ELMS messages, so we will be unable to answer such questions. However you are welcome to ask any questions verbally during the TAs' office hours (or during, before, or after discussion section or lecture if time permits). However, you **cannot wait to write projects in a course with this many students**.

1.1 Extra credit

You are optionally able to earn either 3 or 6 extra–credit bonus points on this project. If you make **only one submission** that passes **all the public tests** you will get **3 extra–credit bonus points**. Additionally, if your single submission is made **at least 48 hours before the on–time deadline** you will get **3 additional extra credit bonus points**, for 6 extra credit points total. (If for some reason your program passes all the public tests on Grace, but doesn't work on the submit server when submitted, so you have to submit more than once– **and this is not due to an error on your part**– you can talk with me verbally in office hours about receiving the extra credit despite having more than one submission.)

2 Project description, and programs to be written

The project tarfile contains three C programs `prog1.c`, `prog2.c`, and `prog3.c` that you must translate to MIPS assembly programs. Extract the files from the tarfile using commands similar to those from before:

```
cd ~/216
tar -zxvf ~/216public/project09/project09.tgz
```

For each C program you must write a MIPS assembly language program that works the same way and does the same things, so it will produce the same output as the C program if given the same input (with one exception discussed in Section 2.4). Your MIPS assembly files **must** be named `prog1.s`, `prog2.s`, and `prog3.s`.

Below we briefly explain what the three C programs do, so we can discuss some things about them, but the definitive description of what they do is the programs themselves, available in the project tarfile. Your assembly programs should do things **exactly** the same way as the C programs do.

All of your assembly programs should terminate via the MIPS system call to terminate execution; abnormal termination should never occur (except in the case of an I/O error, which you are not required to handle).

2.1 Digit reverse program (`prog1.c`)

This program reads a single integer from the input, stores it into a global variable `n`, and prints the number whose digits are the reverse of those of `n`. For example, if `n` is 321 the program prints 123. `n` may have any integer value but the function prints -1 in these two error cases:

- We do not have a interpretation for what the reverse digit for of a negative number would be. For example, would the reverse of -123 be 321 or -321 or even 321− (which is not even a valid number)? Since we do not have a good answer we call this an error case and the program just prints -1 if `n` is negative.
- The reverse digit form of zero is just zero. However, we consider it to be an error case if `n` has more than one digit but its rightmost digit is zero, for example the number 3210. Your assembly program could print 0123 for this case but the C program does not do this, so your assembly program should not either. As a result, if `n` is **not zero itself** but the rightmost digit of `n` is zero then -1 ls also just be printed.

This C program will print 216 if given the input 000216, because leading zeros are ignored when reading numbers in C. The MIPS system call for reading an integer also just ignores leading zeros, so your assembly program will have the same effect.

As you are writing your own tests of this program, note that the largest positive `int` on the Grace machines is 2,147,483,647 ($2^{31} - 1$), so the program will not be able to read any number that is larger than this.

This program will be the easiest of the three to write because it does not use any functions other than `main`, and `main` only uses global variables.

2.2 Catalan program (`prog2.c`)

This program reads an integer into a global variable `number`, which it then passes into a parameter `n` of a function `catalan`, which returns the value of the `n`'th Catalan number. Examples of Catalan numbers for a few values of `n` are in the Project #1 assignment. The value returned by the function is stored in another global variable `result` in `main`, which is then printed, followed by a newline. In non–error cases the program's output will be zero or more, but the function returns -1 if its parameter `n` is less than zero.

The program uses `int`s, not `long`s, because we are only using `int`s on the MIPS. We will not test your program with values of `number` that would cause the result to not fit into an `int`, and you can ignore such values in your own testing. (Overflow can happen even for small values of `number`.)

This program (and the next one) will be more work than the first one because they have functions other than `main`, and the functions have parameters, local variables, and return values.

Carefully read and study the PDF lecture examples of functions in assembly. They illustrate everything that you have to know and to use. But you have to understand everything that they are doing, **before** trying to write assembly code yourself. Study them and if there is anything that you don't understand 100% about what they are doing or why, ask in office hours **before** starting to write this project.

Here are a few suggestions for developing the program one part at a time, assuming you are up to speed on the assembly function examples. (Even someone who is familiar with things can still make mistakes coding in assembly, because it's easy to make errors but hard to find them.)

- First just write the `main` function to read a number (which your first program also did, so you should understand how to read an integer now) and just print it afterwards (you should also know how to print an integer).

  You don't really have to print the number after it's read– you can instead see whether the number is being read into a register correctly in the `QtSpim` graphical simulator– but the `main` function has to print something at its end anyway, and it's not that many instructions to print a number.

- Then just write an empty `catalan` function. In other words, try writing a `catalan` function that has no local variables or parameters, and call it from the `main` function between reading and printing the number. Run the program at this point. If it works then you have some indication that you are creating the stack frame for `catalan` correctly and removing it later.
- Then just modify the `catalan` function to return a hardcoded value, like `216` (it won't actually compute Catalan numbers yet, and it won't even have a parameter yet, it will just always return 216). Then modify the `main` function to print the value returned by the `catalan` function after calling it. If this works then it seems that you are able to return a value from an assembly function.
- Then modify both the `main` function and the `catalan` function to add a parameter to `catalan`, and pass the number read in `main` into `catalan`. But have the `catalan` function just always return its parameter (not 216), instead of trying to compute Catalan numbers yet. If this works then it seems you are able to pass a parameter to a function. You could also add a few instructions in the beginning of the `catalan` function to access and print its parameter.

- Then of course add instructions to the `catalan` function to actually compute and return the `n`'th Catalan number.

2.3 Recursive Catalan program (`prog3.c`)

This program turns the function in `prog2.c` into a recursive function. The program's results will be the same in all cases as the previous one; it just uses recursion instead. This program **<u>must use a recursive function</u>**.

Note that most of the steps above don't apply to this program– if you already wrote the second program then you already have a `catalan` function that is being called from the `main` function and that computes and returns a value. You just have to change it to call itself, which may be a little more tricky to wrap your head around than an iterative function, but of course we have provided examples of recursive assembly functions.

2.4 Reading input

All three programs read input, and they just read integer inputs. You may assume that legitimate integer values will be input to all programs, and that the numbers will be small enough to fit into an `int`.

Note that you can't use the mouse to copy and paste input into the input window in the graphical simulator `QtSpim`, you just have to type input manually (pressing return after each number entered if a program reads more than one number). But the programs each only read one number, so entering the input manually should be easy.

3 Requirements

In this project you are a compiler. In particular, you are a compiler that does not do any optimization. There are many different ways that the three C programs could have been written differently yet produce the same results, but you are not going to modify the C programs when you convert them to assembly, because that is not what a compiler does. A compiler simply converts C programs, as the programmer wrote them, to assembly.

What this means is that you **must follow the programs 100% accurately in translating them to assembly.** Your assembly programs **must** use the exact same algorithms as the corresponding C programs. They **must** have instructions that implement the statements that are in the C programs, just converted to assembly. The functions **must** have the same number of returns as the C programs. They **must** have the **exact same local variables and parameters** as the functions in the C programs do. **Every function, local variable, global variable, if statement, loop, etc., which is present in each C program needs to likewise be exactly present in the assembly program that you write for it.** And there should also **not** be anything in your assembly programs that is not in the corresponding C programs.

If you don't follow the programs **exactly** in translating them, you will lose **significant** credit. As an extreme example, it would easily be possible to write a program that had the same effect as the second or third programs, without using a separate function at all. However we will detect this in grading and you would **lose all credit for that part of the project as a result**. (If this caused you to pass fewer than half of the public tests of this project then you would not pass the course, regardless of overall grade.) Here are more specifics and requirements:

- For the second and third programs that use functions, each function **must** use registers beginning with `$t0`. You **cannot** try to "reserve" different registers for use in different functions. First, if you do this, we will deduct significant credit. Second, and more important, compilers do not do this because it **will not work**. Trying to do this for even small programs such as the second and third ones in this project would require more registers than the machine has. So just start using registers with `$t0`, `$t1`, etc., in each function.

- A consequence of the above is that any statement in one of the C programs that has a side effect **must** immediately cause something (the modified variable) to be stored **in memory**. **The semantics of side effects are that they cause memory to be modified, so you <u>cannot</u> just keep variables only in registers** (there are not enough registers to be able to do this).

  Registers temporarily store operands and results of computations, but operands of computations are first fetched (loaded) from memory, and results of computations are stored back in memory right after they are produced.

- Related to the above: when an assembly function makes any function call– which could either be to another function, or even a recursive call to itself– it **cannot assume that any registers have the same values after the function call that they did before the call**.

Of course if assembly code puts a value in a register and uses that register later, and it has not changed the value in that register or made any function calls in the meantime in the meantime, the register will still have the same value. But different functions use the same registers– because there are not enough registers for each function to have its "own" registers– so registers will almost certainly **not** have the same values after a function call as before. So what assembly code **must** do is to **immediately store the result of any statement that has a side effect into memory**– which must be a memory location **in the runtime stack if the side effect is changing a local variable or parameter**– and after any function call, any values that are needed again must be **reloaded into registers from memory** (from the runtime stack, if the values needed are local variables or parameters).

- The second and third programs (that have functions) **must** pass the same arguments as their C main programs pass to their functions, **using the runtime stack**, as shown in class. You **cannot** just use global variables or registers to communicate values from one function (including `main`) to another function. And because the functions in the C programs **only** use local variables and parameters– and do **not** use any global variables– your assembly functions (other than `main`) can **only** use local variables and parameters and **not** use any global variables.

  If you don't faithfully follow the program and store all function parameters in the runtime stack you will lose **significant** credit.

- Your functions in the second and third programs **must** pass their return values back via a register, as illustrated in class (**not** for example using a global variable).

- The functions in the second and third assembly programs **must** have a local variable for every local variable in the corresponding C program. Similar to the previous item, all function local variables **must** be stored in the runtime stack, **not** in the data segment with a label (only global and static variables are stored in the data segment).

  For example, the function in the third program is shown on the left on the next page. The version to the right of it would work exactly the same, but your assembly code **must** implement the version on the left (storing the result of the recursive call into the local variables `temp1` and `temp2`, rather than just directly returning the result produced by recursive calls), because one thing this C program (this part of the project) is testing for is your code being able to store and access local variables in a recursive function.

- As mentioned, the `catalan` functions in the second and third C programs **only** use local variables and parameters; they do **not** use any global variables. **Only** the `main` functions use global variables. Your assembly functions **must** do the same.

If your assembly programs don't follow these things you will lose **significant** credit.

```
static int catalan(int n) {
  int ans, temp;

  ans= -1;

  if (n == 0)
    ans= 1;
  else
    if (n > 0) {
      temp= catalan(n - 1);
      ans= (2 * (2 * n - 1) * temp) /
      (n + 1);
    }

  return ans;
}
```

```
static int catalan(int n) {
  if (n == 0)
    return 1;
  else
    if (n > 0)
      return (2 * (2 * n - 1) *
              catalan(n - 1)) / (n + 1);
    else return -1;
}
```

Of course there are slightly different ways of translating some statements from C to assembly, which would all be correct (just like there are somewhat different ways of writing the C projects in this course that satisfy the requirements and are fine). For example, when writing code for a conditional or loop you can choose to invert the condition or not (of course the code for the subsidiary statements would be in different orders depending upon which way you choose). But the intention is to faithfully translate the programs like a compiler would (albeit an inefficient compiler), using the same algorithms, steps, statements, and variables that the original C programs do.

A Development procedure

A.1 Running your programs and checking your results

The public tests are just text files that have the numbers to be read by your assembly programs. Each assembly program will be run using the `spim` simulator, with input redirected from an input data file. (For debugging you will want to use `QtSpim`, but the tests just use the nongraphical command–line simulator `spim`.) To simplify things, since different tests run different programs, we wrote short, one–line shell scripts to run each public test. To run the first public test just run the script `public1` (as a command), which you can see just executes the command `spim -file prog1.s < public1.inputdata`. The shell scripts then pipe the output of the simulator into the command `tail -n +2`, which ignores or removes the very first line of its input (it prints all of its standard input starting with the second line). This is because the `spim` simulator prints an extra first line when it runs, which is different on the submit server (because the simulator is set up differently there than on Grace). After `tail` is used to ignore that line your program's actual output is all that remains, which can be directly compared with the expected outputs both on Grace and on the submit server.

As before, if no differences exist between your output and the correct output, `diff` will produce no output, and your code passed the test. Because the tests are all shell scripts the `run-tests2` script that you were able to use in an earlier project (which is a slightly modified version of the `run-tests` script that your TA showed in discussion section) can again be used in this project, to automate the process of running all of the tests. Once you have written the three assembly programs just use the single command `run-tests2` to run all of the public tests, and see which ones passed or failed. Note also that if you name your own test input files of the form student*n*.inputdata, and create expected output files with names of the form student*n*.output and write scripts to run your test named like student*n* (where *n* is a one–digit or two–digit number), `run-tests2` will run your tests along with the public tests. (You must give your scripts executable permission using the `chmod` command, for example `chmod u+x student01` for a script named `student01`, in order to be able to run them.)

A.2 Submitting your program

As before, the command `submit` will submit your project. **Before** you submit, however, you must first check to make sure you have passed all the public tests, by running them yourself.

A.3 Grading criteria

Your grade for this project will be based on:

public tests	40 points
secret tests	45 points
programming style	15 points

A.3.1 Style grading

For this project the style guidelines are different, as you are writing in assembly language, not C. However, assembly language is still a programming language, and good style is crucial to use in any language, so you or others can read and understand your code. Please pay close attention to these guidelines:

- Each of your MIPS program files should begin with a comment containing your name, TerpConnect login ID, University ID number, and your section number.
- Each of your MIPS program files should have a **detailed** explanatory comment at the top explaining in high–level terms what that program is doing.
- Your code must be **thoroughly commented** with explanatory comments throughout. Assembly language can easily become unreadable without proper documentation (and it's not that easy to understand even with good explanation), so it is absolutely necessary that you comment your code very well.

Because assembly is much less readable than C you should write descriptive comments **during development**, not wait until you"re finished with coding to comment. Following the logic to trace a bug in assembly can be confusing and your comments will help you to remember what you were doing when you wrote code.

- Reasonable and consistent indentation is required, and you should also line up the operands of instructions neatly. As assembly language is straightforward (syntactically) compared to higher–level languages, it should not be too difficult to manually maintain indentation, however, Emacs has its own idea of what assembly indentation should look like, which might not agree with yours, and it can be annoying when it indents things differently from what you are trying to do. To disable Emacs from trying to indent your program automatically, just use the following as the very first line of your assembly program (the # makes it a comment, so `spim` and `QtSpim` will ignore it):

```
# -*- mode: text -*-
```

- Label names should be descriptive and meaningful.
- Use appropriate vertical whitespace, meaning blank lines between blocks of instructions that are performing different tasks.
- Program lines should be no longer than 80 characters. (Run your Project #2 on your programs to check this!)

B Other requirements, notes, and hints

- Reminder– there is a short handout on ELMS explaining how to use the graphical simulator `QtSpim`, including useful information about things like how to rerun a program, how to reload it after changing it, how to enlarge the default font, how to single–step through a program, etc. Be sure to read it to help with the development process.
- Write and test small pieces of code at a time. This is even more important with assembly than C code.
- Along these lines, for programs that have functions, you could at first write a dummy function that just returns a fixed value like 1 or 0. Although this still requires writing the prologue and epilogue that set up and remove the function's stack frame it will be easier than implementing the function in full, and it will allow you to test the main program to some extent. You can change the value the function returns in checking your main program, then when the main program works, develop the function for real.
- Don't forget to assign the return values of function calls to the local variables `ans` in the second and third programs, which the C programs do, otherwise your functions will probably be returning the wrong values.
- Except as noted in Section 2.4 above, in all cases each of your assembly programs must perform identically and produce the same results as the corresponding C program.

  We emphasize again that your programs **must** do the same things the C programs do, and use the same algorithms. You should strive to implement the programs in assembly as closely to what the C programs are doing as possible. Do not add any extra conditions or checks, or remove any existing ones. And every C statement that has a side effect **must** cause a memory location to be modified (even if you think the program would work without it).

  Remember that assembly instructions are sequential and that when compilers translate code to assembly they do so incrementally, one C statement at a time. So a good approach is to go through a program and think of how each individual line of code **in isolation** would be translated into one or more assembly instructions, almost completely **ignoring** what statements are before it and after it.
- Your programs may use any MIPS instructions supported by `spim`/`QtSpim`, even if they were not explained in class.
- For the programs that have functions you **must** use the conventions we showed in class for the location and order that parameters and local variables are stored in stack frames.
- You will lose **major** credit if your second and third programs don't use functions, or if the `catalan` function in the third program isn't recursive.
- If you just copy the second program `prog2.s` to the third one `prog3.s` it would work and pass all tests. But this will be checked for in grading, it would be considered to be hardcoding and handled as such, and you would receive at minimum a **severe** penalty. (Doing so could also postentially be considered to be cheating, so we would have to involve the Office of Student Conduct to determine that.)

- Recall that the course project grading policies handout on ELMS says that all your projects must work on **at least half of the public tests** (by the end of the semester) in order for you to be eligible to pass the course. The project grading policy has full details.
- If an assembly program uses memory that was never initialized it can produce different results in `spim` vs. `QtSpim`, and it can produce different results when just run in QtSpim versus single–stepped through in `QtSpim`. So even in assembly you must make sure that every variable in memory is given a value somehow before the value in that location is used.
- **Do not use an actual MIPS compiler to generate your MIPS assembly code. If you do, you will not get any credit for the project, so you will not pass the course.** It is easy to identify compiler–generated code. You need to write the assembly programs yourself.
- For this project you will **lose one point** from your final project score for every submission that you make in excess of four submissions, for any reason.

C Academic integrity

Please **carefully read** the academic honesty section of the syllabus. **Any evidence** of impermissible cooperation on programming assignments, use of disallowed materials or resources, posting your project code publicly anywhere, or unauthorized use of computer accounts, **will be submitted** to the Student Honor Council, which could result in an XF for the course, or suspension or expulsion from the University. Be sure you understand what you are and what you are not permitted to do in regards to academic integrity when it comes to programming assignments. These policies apply to all students, and the Student Honor Council does not consider lack of knowledge of the policies to be a defense for violating them. More information is in the course syllabus– please review it now.

The academic integrity requirements also apply to any test data for programming assignments, which must be **your own original work**. Exchanging test data or working together to write test cases is also prohibited.