CS 367: Homework 4 - x86-64 Assembly

Due Friday, 10/14 by 11:59pm.

Here is the download to get you started: h4handout.tar

In this assignment, we will write some assembly functions by hand in separate files, and link them in with a regular old `.c` file. We will hand-code the assembly files, and utilize a `Makefile` to create the executable by generating object files and combining them.

- It may be beneficial to write pseudocode for the algorithms first, and then try to translate them into assembly. Thinking in assembly is probably not second-nature enough for most of us.
- you probably could benefit from using `gdb` when your code doesn't run as expected. Here is our GDB starter guide , which is also going to be useful for our next project (the "Binary Bomb").
- There is a main program that helps you run tests, from the command line. Example:

```
demo$ ./hwk4 mul_20  3
60
```

- As always, grading will occur on `zeus`. If you disregard this advice and work elsewhere, plan on finishing a few days early so you have a chance to ask questions about your predicament in office hours or on piazza…

Assembly Functions

We will hand craft individual assembly files for each of the following functions. Descriptions and examples are below for each.

file name	function prototype	description
`mul_20.s`	`long mul_20(long)`	multiplies the argument by 20 and returns the result. Must use shifting in your solution.
`longlog.s`	`long longlog(long)`	returns the log of the argument (truncated to a whole number).
`collatz.s`	`long collatz(long)`	returns the length of the collatz sequence beginning with the argument's (positive) value.
`prime.s`	`long prime(long)`	given positive long int, returns 0x1 when it is prime and 0x0 when it is not.
`caller.s`	`long caller(long,long)`	adds its arguments together, feeds result to collatz, adds one, and returns that result.
`sum_primes.s`	`long sum_primes(long, long[])`	given an array length and the array, returns sum of all primes in the array.

`mul_20`

As a warm up, we perform a simple calculation: multiply the given value by 20 and return it. Use the following template to get started. You are not allowed to directly call multiplication, use the other tricks we've learned to multiply by a constant.

NOTE, you need to work on `zeus` for this to be sure to work!

file: `mul_20.s`

```
        .text
.global mul_20
        .type   mul_20,@function
mul_20:

        # YOUR CODE HERE

        ret

        .size   mul_20, .-mul_20
```

Each assembly file has a template you can fill out (found in our provided files).

`longlog`

Given a long argument `n`, calculate `longlog(n)`, the whole number $\log_2$ value.

Examples:

n	longlog(n)
1	0
2	1
3	1
4	2
5	2
15	3
16	4
17	4

`collatz`

Given a positive long int argument `n`, calculate the length of the collatz sequence starting with `n`.

- all sequences end with the value `1`.
- when a value is even, the next value is half of this value.
- when a value is odd, the next value is triple this value plus 1.

Examples:

n	collatz(n)	collatz sequence
1	1	1.
5	6	5,16,8,4,2,1.
7	17	7,22,11,34,17,52,26,13,40,20,10,5,16,8,4,2,1.

n	collatz(n)	collatz sequence
42	9	42,21,64,32,16,8,4,2,1.
32	6	32,16,8,4,2,1.

`prime`

Given a positive long argument `n`, determine if it is prime. You will want to inspect the `idiv` instruction as part of your solution (read the *Hints* section below). Return `0x0` when it is composite, and `0x1` when it is prime.

Examples:

n	prime(n)
1	0
2	1
3	1
4	0
5	1
1117	1
11117	1

`caller`

This last one is a bit fun - we will have to manually call another function. Accept two arguments `x` and `y`. Add them together, feed the result to collatz, add one to that answer, and return that result.

Examples

x, y	caller(x,y)	(collatz(x+y))
2,3	7	collatz(5)==6
5,7	11	collatz(12)==10
7,5	11	collatz(12)==10
100,232	114	collatz(332)==113
99999,1	130	collatz(100000)==129

EXTRA CREDIT: `sum_primes`

Accept a `long n` (the length of an array), and `long[] xs` (the array). Add up all the prime numbers in the array. Return this sum.

Be careful in how you call this in the main program - we will give the length argument, `n`, *before* we give any of the values in the list. For instance, a length-three array containing 100, 207, and 300:

```
prompt$ ./hwk4 sum_primes 5    1  2  3  4  5
10
prompt$ ./hwk4 sum_primes 3    3  4  5
8
prompt$ ./hwk4 sum_primes 3   10 20 30
0
prompt$
```

Hints

`idiv` instruction

Division needs a numerator and a denominator; we can then obtain a quotient and remainder. The behavior of `idiv` (signed division) is a bit quirky, but simple enough to use once we understand it.

- Example using 64-bit registers
  - idiv divides a 128-bit value by a 64-bit value.
  - the numerator is the concatenation of two registers, written out as `rdx:rax`. This creates a 128-bit value. However, we often don't need the full space, and will be tasked with zeroing out `rdx` in order to get a meaningful denominator value.
    - instead of manually trying to zero-out `rdx`, we should use the `cqto` command, which sign-extends `rax` to `rdx:rax`. ("convert quad to oct" with those hardcoded registers.) This will generally look like:

```
# %rax holds our numerator
cqto
divq   <denominator>
# now answer (quotient) is in %rax, and remainder is in %rdx.
```

  - the divisor is the given argument:

```
idivq   <numerator>
```

  - the answer is given as a quotient and remainder. The quotient is found in `rax`, and the remainder is in `rdx`.

Main Program

We need a driver program to use all these beautifully handcrafted assembly functions. Provided is a program named `hwk4.c` that accepts commandline arguments: first, the name of the function to call; next, the arguments to that function. (arrays are handled a bit carefully).

All calls to main will use the command line arguments to call the appropriate function and then print the answer out, and quit. If we don't receive valid command line arguments, it prints an error message and quits.

Makefile

Lastly, we provided a `Makefile` that correctly generates the whole program. We must generate it each step of the way, so look below for all stages.

Steps in the `Makefile`

- generate `hwk4.s` (use the `-S` flag).
- generate `hwk4.o` (use the `-c` flag).
- generate `.o` files for each handcrafted `.s` file: use the `-c` flag with gcc.
- combine all `.o` files into the executable named `hwk4`, exhibiting the functionality described for your Main Program. `gcc` helpfully runs the `ld` linker when you feed it `.o` files, just don't ask it to stop early (via `-S` or `-c`). We just use `gcc` here.
- we leave optimizations off during all compilations (`gcc -O0`)
- target the C99 standard (`gcc -std=c99`)
- we generate 64-bit code for `zeus`. This is actually the default behavior. (you could use the `-m32` flag for `gcc` to generate 32-bit code instead, but that's not this assignment's target).

Turning It In

Place all the above files in a folder with the name `netID_h4` (use your own GMU netID, such as `msnyde14_h4` or `gmason76_h4`). Compress that folder as a `.tar` file, and upload it on BlackBoard.

Grading

Grades will be dependent upon correct calculations of each function, as tested through `hwk4.c` or other automated means. We will also inspect them manually for a few style requirements - correct register saving behaviors, commenting your code, etc.

task	points
mul_{20}.s	15%
longlog.s	20%
collatz.s	20%
prime.s	20%
caller.s	15%
style req's	10%
TOTAL	100%

- +5 possible for extra credit.
- if you have no comments, you might get heavily penalized for this assignment. Assembly is hard enough to read without comments, so you need to comment what variables are in what register and so on.