Отчет по индивидуальному домашнему заданию №3 по дисциплине “Архитектура вычислительных систем”.
Владимиров Дмитрий Андреевич, БПИ218.
Вариант 11.

Условие. Разработать программу, вычисляющую с помощью степенного ряда с точность не хуже 0,05% значение функции 1/(1 – x) для заданного параметра x.

Аннотация. Степенной ряд раскладывается в (1 + x + x^2 + x^3 + …). Ряд сходится при |x| < 1, значит такими и должны быть входные параметры. Проверка на это присутствует. Ограничение на точность в 0,5% понял следующим образом: считать степенной ряд до тех пор, пока разность соседних членов не станет меньше чем 0,0005. При указании имени файла в параметрах командной строки при запуске – параметр считывается из файла, иначе генерируется рандомно.

На 4 балла:

1. Осуществлено более сложнее задание с выводом и вводом через файлы, поэтому с клавиатуры вводится имя файла, а результат выводится в файл output.txt. Два исходных файла на языке C: main.c и res.c
2. Оба файла main.c и res.c скомпилированы без каких-либо оптимизирующих опций в файлы main.s и find_substring.s путем команд > gcc -masm=intel -S main.c -o main.s и > gcc -masm=intel -S res.c -o res.s. В этих файлах в комментариях указана связь между переменными в коде на C с действиями с ними в коде на ассемблере. Далее файлы main.s и res.s докомпилированы в исполняемый файл main при помощи команды > gcc main.s res.c -o main.
3. Файлы main.c и res.c скомпилированы с флагами оптимизации по размеру в файлы main2.s и res2.s путем команд > gcc -masm=intel -fno-asynchronous-unwind-tables -fno-jump-tables -fno-stack-protector -fno-exceptions -S main.c -o main2.s и > gcc -masm=intel -fno-asynchronous-unwind-tables -fno-jump-tables -fno-stack-protector -fno-exceptions -S res.c -o res2.s.
4. Файлы main2.s и res2.s докомпилированы в исполняемый файл main2 при помощи команды > gcc main2.s res2.s -o main2.
5. Исходные файлы на ассемблере и на языки C приложены и соответствуют имена, указанным в отчете выше.
6. Оба файла были протестированы при помощи команд > ./main input<номер теста>.txt и > ./main2 input<номер тесте>.txt и сравнения результатов в файле output.txt. Папка с тестами, на всякий случай, прилагается.

Входные данные	main	main2
0	1.000000	1.000000

0.1	1.111000	1.111000
-0.1	0.909000	0.909000
0.4	1.666230	1.666230
-0.5	0.666992	0.666992
0.9	9.995432	9.995432
0.999	999.500449	999.500449
-0.9999	0.499775	0.499775
0.99994	16658.333380	16658.333380

Видим, что выходные данные не отличаются, а значит обе программы функционируют одинаково.
Были использованы следующие опции компиляции:

- -masm=intel для использования необходимого вида ассемблера.
- -fno-asynchronous-unwind-tables гарантирует точность таблиц раскрутуки только в пределах функции.
- -fno-jump-tables – не генерировать инструкций табличного перехода
- -fno-stack-protector – отключает защиту компилятор
- -fno-exceptions позволяет отключить генерацию таблиц обработки исключений.

На 5 баллов:

1. Функции с передачей данных через параметры присутствуют
2. Локальные переменные используются, при компиляции, как было видно в ассемблерной программе, они отображаются на стек
3. Комментарии по передаче фактических и формальных параметров в функцию и возвращаемого результата добавлены в файлы main.s и res.s к остальным комментариям.
4. Все выше упомянутые файлы, в том числе и файлы с необходимыми комментариями (а именно main.s и res.s) прилагаются

На 6 баллов:
Задача: использовать регистры процессора rbx и r12-r15, а также xmm0-8 везде где можно, то есть минимизировать использование стека.

1. Где было возможно, сохранил переменные в регистры процессора. Файлы с ассемблерным кодом после рефакторинга и комментариями – main_refactored.s и res_refactored.s.
2. Файл res_refactored.s оказался меньше файла res.c, потому что регистры xmm0-8 не выделяются отдельно, а вот файл main_refactored.s оказался больше файла main.s. Несмотря на то, что работа с регистрами уменьшает количество выполняемых операций (а значит, по логике должна и уменьшать размер кода), файл оказался недостаточно большим, чтоб

покрыть те несколько строк с выделением регистров процессора для функции.
3. Программа протестирована на тех же тестовых данных, что и в первый раз. Предварительно программа была скомпилирована при помощи команды > gcc main_refactored.s res_refactored.s -o main_refactored

Входные данные	main	main_refactored
0	1.000000	1.000000
0.1	1.111000	1.111000
-0.1	0.909000	0.909000
0.4	1.666230	1.666230
-0.5	0.666992	0.666992
0.9	9.995432	9.995432
0.999	999.500449	999.500449
-0.9999	0.499775	0.499775
0.99994	16658.333380	16658.333380

На 7 баллов:

1. Программа реализована в виде двух единиц компиляции. (ассемблерные файлы main.s и res.s)
2. Исходные данные лежат в каком-то файле (при указании этого файла, иначе – рандом) и выводятся в файл output.txt. Имя входного файла указывается аргументом командной строки, если файл не удается открыть – выводится надпись “Wrong filename” и программа завершается. => командная строка проверяется на корректное открытие файла. На корректность числа аргументов тоже проверка присутствует (если их нет, то запускается рандомная генерация, если это только файл, то считывается аргумент из файла, иначе же выводится сообщение “You should enter only file name or nothing to generate a random argument” и программа завершается аварийно.
3. Файлы для тестового прогона расположены в отдельной папке tests.
4. Так как программа не менялась с момента критериев на 4 балла, то необходимость тестировать ее на данном этапе и формировать отчеты о расширении отсутствует.

На 8 баллов:

1. При отсутвии входных данных они генерируются рандомно – сгенерированное число показывается в консоли, результат же выводится в файле output.txt
2. Выбрать в командной строке ввод из файла или рандомной генерации возможно. Как я отмечал ранее, если необходимо сгенерировать данные

рандомно, то не нужно передавать ничего, просто запустить исполняемый файл с программой

3. Время работы программы без ввода и вывода замеряется и выводится каждый раз в консоль. Для чистоты эксперимента было приянто решение находить результат 10000 раз, так можно объективнее сравнивать скорости выполнения двух программ.
4. Для тестирования времени работы программы подсчет результата в ней выполняется 10000 раз, буду запускать по 5 раз файл main.s на каждых входных данных и брать среднее значение из них.

Входные данные	Время в миллисекундах
0	0
0.1	0
-0.1	0
0.4	0
-0.5	0
0.9	3
0.999	374
-0.9999	3775
0.99994	6943

На 9 баллов:

1) Для начала скомпилируем ассемблерные файлы с флагом оптимизации -O3 путем команд > gcc -S -O3 main.c -o main_opt.s и > gcc -S -O3 res.c -o res_opt.s. Оба файла по размеру стали меньше чем обычные main.s и find_substring.s без флагов оптимизации.

Далее получим исполняемый файл main_opt путем команды > gcc main_opt.s res_opt.s -o main_opt

Исполняемый файл main_opt также по размеру оказался меньше чем main

Теперь сравним время выполнения программ на тех же входных данных, что и обычно. Данные для первого файла у нас уже есть, осталось только высчитать их для второго.

Входные данные	main (время в мс)	main_opt (время в мс)
0	0	0
0.1	0	0
-0.1	0	0
0.4	0	0

-0.5	0	0
0.9	3	2
0.999	374	290
-0.9999	3775	2921
0.99994	6943	4918

2) Для начала скомпилируем ассемблерные файлы с флагом оптимизации -Os путем команд > gcc -S -Os main.c -o main_optsize.s и > gcc -S -Os res.c -o res_optsize.s. Оба файла по размеру стали ожидаемо меньше чем обычные main.s и res.s без флагов оптимизации.
Далее получим исполняемый файл main_optsize путем команды > gcc main_optsize.s res_optsize.s -o main_optsize
Исполняемый файл main_optsize также по размеру оказался меньше чем main
Теперь сравним время выполнения программ на тех же входных данных, что и обычно. Данные для первого файла у нас уже есть, осталось только высчитать их для второго.

Входные данные	main (время в мс)	main_optsize (время в мс)
0	0	0
0.1	0	0
-0.1	0	0
0.4	0	0
-0.5	0	0
0.9	3	2
0.999	374	293
-0.9999	3775	3009
0.99994	6943	5127

Вывод: Флаг -O3 больше оптимизирует по скорости, но меньше по размеру кода (хотя какая-то оптимизация и происходит), а флаг -Os сильно оптимизирует по размеру кода, но по производительности уступает флагу -O3