Санкт-Петербургский политехнический университет Петра Великого

Институт компьютерных наук и технологий

Высшая школа интеллектуальных систем и суперкомпьютерных технологий

**Отчёт по лабораторной работе № 4**

Дисциплина: Низкоуровневое программирование

Тема: Раздельная компиляция

Вариант: 4

Выполнил студент гр. 3530901/90002 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Е.К. Борисов

(подпись)

Принял старший преподаватель \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Д.С. Степанов

(подпись)

“ ” 2021 г.

Санкт-Петербург

2021

##### **Постановка задачи**

1. Изучить методические материалы, опубликованные на сайте курса.

2. Установить пакет средств разработки “SiFive GNU Embedded Toolchain” для RISC-V.

3. На языке C разработать функцию, реализующую сортировку массива вставкой. Поместить определение функции в отдельный исходный файл, оформить заголовочный файл. Разработать тестовую программу на языке C.

4. Собрать программу «по шагам». Проанализировать выход препроцессора и компилятора. Проанализировать состав и содержимое секций, таблицы символов, таблицы перемещений и отладочную информацию, содержащуюся в объектных файлах и исполнимом файле.

5. Выделить разработанную функцию в статическую библиотеку. Разработать make-файлы для сборки библиотеки и использующей ее тестовой программы. Проанализировать ход сборки библиотеки и программы, созданные файлы зависимостей.

##### **1.Программа на языке C:**

Алгоритм сортировки вставками - алгоритм, в котором элементы входной последовательности просматриваются по одному, и каждый новый поступивший элемент размещается в подходящее место среди ранее упорядоченных элементов

Предполагается, что первый элемент списка отсортирован. Переходим к следующему элементу, обозначим его х. Если х больше первого, оставляем его на своём месте. Если он меньше, копируем его на вторую позицию, а х устанавливаем, как первый элемент.

Переходя к другим элементам несортированного сегмента, перемещаем более крупные элементы в отсортированном сегменте вверх по списку, пока не встретим элемент меньше x или не дойдём до конца списка. В первом случае x помещается на правильную позицию.

Напишем данный алгоритм на языке C, поместим функцию сортировки в отдельный файл и оформим заголовочный файл.

|  |
| --- |
|  |
| Рис 1.1 Заголовочный файл insertion.h |

В заголовочном файле указана функция сортировки вставкой для дальнейшего использования ее в тестовой программе.

|  |
| --- |
|  |
| Рис 1.2 Файл функции сортировки вставкой insertion.c |

|  |
| --- |
|  |
| Рис 1.3 Файл тестовой программы main.c |

Сделаем компиляцию программы с помощью MinGW-w64 и посмотрим на результат работы программы.

|  |
| --- |
|  |
| Рис. 1.4 Результат работы программы |

Как мы видим сортировка выполнена корректно, переходим к сборке программы “По шагам”

##### **2. Сборка программы “по шагам”**

##### *Препроцессирование*

Используя пакет разработки “SiFive GNU Embedded Toolchain” выполним препроцессирование файлов, используя следующие команды:

|  |
| --- |
| riscv64-unknown-elf-gcc.exe -march=rv64iac -mabi=lp64 -O1 -E main.c -o main.i  riscv64-unknown-elf-gcc.exe -march=rv64iac -mabi=lp64 -O1 -E insertion.c -o insertion.i |

Параметр *-E* указывает на то, что обработка файлов должна происходить только препроцессором. Параметр *-o* отвечает за название результирующего файла, то есть результат препроцессирования содержится в файлах *main.i* и *insertion.i*. В связи с тем, что в файле тестовой программы мы использовали стандартную библиотеку языка C “stdio.h” для вывода на консоль значений массива, результирующий файл препроцессирования имеет много добавочных строк.

Листинг 2.1. Файл main.i (фрагмент)

|  |
| --- |
| # 1 "main.c"  # 1 "<built-in>"  # 1 "<command-line>"  # 1 "main.c"  -------------------------------------  # 4 "insertion.h"  void InsertionSort(int n, unsigned array[]);  # 3 "main.c" 2  int main() {  unsigned array[10] = {1, 5, 4, 6, 2, 3, 7, 9, 8, 10};  printf("Source array:\n");  for (int i = 0; i < 10; ++i){  printf("%d ", array[i]);  }  InsertionSort(10, array);  printf("\nSorted array:\n");  for (int i = 0; i < 10; ++i){  printf("%d ", array[i]);  }  return 0;  } |

Листинг 2.2. Файл insertion.i

|  |
| --- |
| # 1 "insertion.c"  # 1 "<built-in>"  # 1 "<command-line>"  # 1 "insertion.c"  # 1 "insertion.h" 1  void InsertionSort(int n, unsigned array[]);  # 2 "insertion.c" 2  void InsertionSort(int n, unsigned array[]) {  int itemToInsert, location;  for (int i = 1; i < n; i++)  {  itemToInsert = array[i];  location = i - 1;  while(location >= 0 && array[location] > itemToInsert)  {  array[location+1] = array[location];  location = location - 1;  }  array[location+1] = itemToInsert;  }  } |

Появившиеся нестандартные директивы, начинающиеся с символа “#”, используются для передачи информации об исходном тексте из препроцессора в компилятор. Например, последняя директива «# 1 “main.c”» информирует компилятор о том, что следующая строка является

результатом обработки строки 1 исходного файла “main.c”. Также мы видим, что в данных файлах содержится информация из заголовочного файла.

##### *Компиляция*

Компиляция осуществляется следующими командами:

|  |
| --- |
| riscv64-unknown-elf-gcc.exe -march=rv64iac -mabi=lp64 -O1 –S main.i -o main.s  riscv64-unknown-elf-gcc.exe -march=rv64iac -mabi=lp64 -O1 –S insertion.i -o insertion.s |

Листинг 2.3. Файл main.s

|  |
| --- |
| .file "main.c"  .option nopic  .attribute arch, "rv64i2p0\_a2p0\_c2p0"  .attribute unaligned\_access, 0  .attribute stack\_align, 16  .text  .section .rodata.str1.8,"aMS",@progbits,1  .align 3  .LC1:  .string "Source array:"  .align 3  .LC2:  .string "%d "  .align 3  .LC3:  .string "\nSorted array:"  .text  .align 1  .globl main  .type main, @function  main:  addi sp,sp,-96  sd ra,88(sp)  sd s0,80(sp)  sd s1,72(sp)  sd s2,64(sp)  sd s3,56(sp)  lui a5,%hi(.LANCHOR0)  addi a5,a5,%lo(.LANCHOR0)  ld a1,0(a5)  ld a2,8(a5)  ld a3,16(a5)  ld a4,24(a5)  ld a5,32(a5)  sd a1,8(sp)  sd a2,16(sp)  sd a3,24(sp)  sd a4,32(sp)  sd a5,40(sp)  lui a0,%hi(.LC1)  addi a0,a0,%lo(.LC1)  call puts  addi s0,sp,8  addi s2,sp,48  mv s1,s0  lui s3,%hi(.LC2)  .L2:  lw a1,0(s1)  addi a0,s3,%lo(.LC2)  call printf  addi s1,s1,4  bne s1,s2,.L2  addi a1,sp,8  li a0,10  call InsertionSort  lui a0,%hi(.LC3)  addi a0,a0,%lo(.LC3)  call puts  lui s1,%hi(.LC2)  .L3:  lw a1,0(s0)  addi a0,s1,%lo(.LC2)  call printf  addi s0,s0,4  bne s0,s2,.L3  li a0,0  ld ra,88(sp)  ld s0,80(sp)  ld s1,72(sp)  ld s2,64(sp)  ld s3,56(sp)  addi sp,sp,96  jr ra  .size main, .-main  .section .rodata  .align 3  .set .LANCHOR0,. + 0  .LC0:  .word 1  .word 5  .word 4  .word 6  .word 2  .word 3  .word 7  .word 9  .word 8  .word 10  .ident "GCC: (SiFive GCC-Metal 10.2.0-2020.12.8) 10.2.0" |

Листинг 2.4. Файл insertion.s

|  |
| --- |
| .file "insertion.c"  .option nopic  .attribute arch, "rv64i2p0\_a2p0\_c2p0"  .attribute unaligned\_access, 0  .attribute stack\_align, 16  .text  .align 1  .globl InsertionSort  .type InsertionSort, @function  InsertionSort:  li a5,1  ble a0,a5,.L1  mv a7,a1  addiw t1,a0,-1  li a6,0  li a0,-1  j .L6  .L4:  addi a4,a4,1  slli a4,a4,2  add a4,a1,a4  sw a2,0(a4)  addiw a6,a6,1  addi a7,a7,4  beq a6,t1,.L1  .L6:  lw a2,4(a7)  sext.w a4,a6  mv a5,a7  blt a6,zero,.L4  .L3:  lw a3,0(a5)  bgeu a2,a3,.L4  sw a3,4(a5)  addiw a4,a4,-1  addi a5,a5,-4  bne a4,a0,.L3  j .L4  .L1:  ret  .size InsertionSort, .-InsertionSort  .ident "GCC: (SiFive GCC-Metal 10.2.0-2020.12.8) 10.2.0" |

Файл *main.s:*

Красным цветом выделена часть кода, где мы можем увидеть обращение тестовой программы к функции сортировки с помощью псевдоинструкции call.

Фиолетовым цветом показаны участки кода, где мы видим реализацию двух циклов for тестовой программы.

Зеленым цветом помечена часть кода, содержащая наш исходный массив данных.

Файл *insertion.s:*

Красным цветом выделен фрагмент кода, отвечающий за цикл for функции сортировки, мы видим, как значение регистра *a6* увеличивается на 1 при каждой итерации до тех пор, пока оно не будет равно значению регистра *t1*,в котором хранится значение длинны массива.

Зеленым цветом же выделен внутренний цикл while, мы можем увидеть как происходит перестановка элементов массива и условие выхода из цикла.

##### *Ассемблирование*

Ассемблирование осуществляется следующими командами:

|  |
| --- |
| riscv64-unknown-elf-gcc.exe -march=rv64iac -mabi=lp64 -v -c main.s -o main.o  riscv64-unknown-elf-gcc.exe -march=rv64iac -mabi=lp64 -v -c insertion.s -o insertion.o |

На выходе мы получаем два бинарных файла “main.o” и “ insertion.o”. Для их прочтения используем программу из пакета разработки.

Листинг 2.5. Заголовки секций файла main.o

|  |
| --- |
| riscv64-unknown-elf-objdump.exe -h main.o |
| main.o: file format elf64-littleriscv  Sections:  Idx Name Size VMA LMA File off Algn  0 .text 0000009c 0000000000000000 0000000000000000 00000040 2\*\*1  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE  1 .data 00000000 0000000000000000 0000000000000000 000000dc 2\*\*0  CONTENTS, ALLOC, LOAD, DATA  2 .bss 00000000 0000000000000000 0000000000000000 000000dc  ALLOC  3 .rodata.str1.8 00000027 0000000000000000 0000000000000000 000000e0 2\*\*3  CONTENTS, ALLOC, LOAD, READONLY, DATA  4 .rodata 00000028 0000000000000000 0000000000000000 00000108 2\*\*3  CONTENTS, ALLOC, LOAD, READONLY, DATA  5 .comment 00000031 0000000000000000 0000000000000000 00000130 2\*\*0  CONTENTS, READONLY  6 .riscv.attributes 00000026 0000000000000000 0000000000000000 00000161 2\*\*0  CONTENTS, READONLY |

Листинг 2.6. Заголовки секций файла insertion.o

|  |
| --- |
| riscv64-unknown-elf-objdump.exe -h insertion.o |
| insertion.o: file format elf64-littleriscv  Sections:  Idx Name Size VMA LMA File off Algn  0 .text 00000044 0000000000000000 0000000000000000 00000040 2\*\*1  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE  1 .data 00000000 0000000000000000 0000000000000000 00000084 2\*\*0  CONTENTS, ALLOC, LOAD, DATA  2 .bss 00000000 0000000000000000 0000000000000000 00000084 2\*\*0  ALLOC  3 .comment 00000031 0000000000000000 0000000000000000 00000084 2\*\*0  CONTENTS, READONLY  4 .riscv.attributes 00000026 0000000000000000 0000000000000000 000000b5 2\*\*0  CONTENTS, READONLY |

Секции:

*.text* - скомпилированный машинный код;

*.data* - секция инициализированных данных;

*.rodata* - аналог .data для неизменяемых данных;

*.bss* - секция данных, инициализированных нулями;

.comment — информация о версии компилятора;

вывод objdump нам также сообщает, что RISC-V является little-endian архитектурой, произведем декодирование кода, чтобы рассмотреть секцию .text подробнее, c помощью команды:

|  |
| --- |
| riscv64-unknown-elf-objdump -d -M no-aliases -j .text main.o |

Опция “-d” инициирует процесс дизассемблирования, опция “-M no-aliases”

требует использовать в выводе только инструкции системы команд (но не псевдоинструкции ассемблера)

Листинг 2.7. Дизассемблированный файл main.o

|  |
| --- |
| main.o: file format elf64-littleriscv  Disassembly of section .text:  0000000000000000 <main>:  0: 711d c.addi16sp sp,-96  2: ec86 c.sdsp ra,88(sp)  4: e8a2 c.sdsp s0,80(sp)  6: e4a6 c.sdsp s1,72(sp)  8: e0ca c.sdsp s2,64(sp)  a: fc4e c.sdsp s3,56(sp)  c: 000007b7 lui a5,0x0  10: 00078793 addi a5,a5,0 # 0 <main>  14: 638c c.ld a1,0(a5)  16: 6790 c.ld a2,8(a5)  18: 6b94 c.ld a3,16(a5)  1a: 6f98 c.ld a4,24(a5)  1c: 739c c.ld a5,32(a5)  1e: e42e c.sdsp a1,8(sp)  20: e832 c.sdsp a2,16(sp)  22: ec36 c.sdsp a3,24(sp)  24: f03a c.sdsp a4,32(sp)  26: f43e c.sdsp a5,40(sp)  28: 00000537 lui a0,0x0  2c: 00050513 addi a0,a0,0 # 0 <main>  30: 00000097 auipc ra,0x0  34: 000080e7 jalr ra,0(ra) # 30 <main+0x30>  38: 0020 c.addi4spn s0,sp,8  3a: 03010913 addi s2,sp,48  3e: 84a2 c.mv s1,s0  40: 000009b7 lui s3,0x0  0000000000000044 <.L2>:  44: 408c c.lw a1,0(s1)  46: 00098513 addi a0,s3,0 # 0 <main>  4a: 00000097 auipc ra,0x0  4e: 000080e7 jalr ra,0(ra) # 4a <.L2+0x6>  52: 0491 c.addi s1,4  54: ff2498e3 bne s1,s2,44 <.L2>  58: 002c c.addi4spn a1,sp,8  5a: 4529 c.li a0,10  5c: 00000097 auipc ra,0x0  60: 000080e7 jalr ra,0(ra) # 5c <.L2+0x18>  64: 00000537 lui a0,0x0  68: 00050513 addi a0,a0,0 # 0 <main>  6c: 00000097 auipc ra,0x0  70: 000080e7 jalr ra,0(ra) # 6c <.L2+0x28>  74: 000004b7 lui s1,0x0  0000000000000078 <.L3>:  78: 400c c.lw a1,0(s0)  7a: 00048513 addi a0,s1,0 # 0 <main>  7e: 00000097 auipc ra,0x0  82: 000080e7 jalr ra,0(ra) # 7e <.L3+0x6>  86: 0411 c.addi s0,4  88: ff2418e3 bne s0,s2,78 <.L3>  8c: 4501 c.li a0,0  8e: 60e6 c.ldsp ra,88(sp)  90: 6446 c.ldsp s0,80(sp)  92: 64a6 c.ldsp s1,72(sp)  94: 6906 c.ldsp s2,64(sp)  96: 79e2 c.ldsp s3,56(sp)  98: 6125 c.addi16sp sp,96  9a: 8082 c.jr ra |

Мы видим, как происходит выход из подпрограммы *“c.jr ra”,* также мы видим, как сочетание инструкций auipc и jalr заменяют псевдоинструкцию call.

Рассмотрим таблицу символов и таблицу перемещений с помощью команд:

|  |
| --- |
| riscv64-unknown-elf-objdump -t main.o insertion.o  riscv64-unknown-elf-objdump -r main.o insertion.o |

Листинг 2.8. Таблица символов

|  |
| --- |
| main.o: file format elf64-littleriscv  SYMBOL TABLE:  0000000000000000 l df \*ABS\* 0000000000000000 main.c  0000000000000000 l d .text 0000000000000000 .text  0000000000000000 l d .data 0000000000000000 .data  0000000000000000 l d .bss 0000000000000000 .bss  0000000000000000 l d .rodata.str1.8 0000000000000000 .rodata.str1.8  0000000000000000 l d .rodata 0000000000000000 .rodata  0000000000000000 l .rodata 0000000000000000 .LANCHOR0  0000000000000000 l .rodata.str1.8 0000000000000000 .LC1  0000000000000010 l .rodata.str1.8 0000000000000000 .LC2  0000000000000018 l .rodata.str1.8 0000000000000000 .LC3  0000000000000044 l .text 0000000000000000 .L2  0000000000000078 l .text 0000000000000000 .L3  0000000000000000 l d .comment 0000000000000000 .comment  0000000000000000 l d .riscv.attributes 0000000000000000 .riscv.attributes  0000000000000000 g F .text 000000000000009c main  0000000000000000 \*UND\* 0000000000000000 puts  0000000000000000 \*UND\* 0000000000000000 printf  0000000000000000 \*UND\* 0000000000000000 InsertionSort  insertion.o: file format elf64-littleriscv  SYMBOL TABLE:  0000000000000000 l df \*ABS\* 0000000000000000 insertion.c  0000000000000000 l d .text 0000000000000000 .text  0000000000000000 l d .data 0000000000000000 .data  0000000000000000 l d .bss 0000000000000000 .bss  0000000000000042 l .text 0000000000000000 .L1  0000000000000022 l .text 0000000000000000 .L6  0000000000000012 l .text 0000000000000000 .L4  0000000000000030 l .text 0000000000000000 .L3  0000000000000000 l d .comment 0000000000000000 .comment  0000000000000000 l d .riscv.attributes 0000000000000000 .riscv.attributes  0000000000000000 g F .text 0000000000000044 InsertionSort |

В таблице символов main.o имеется запись: символ “puts” типа \*UND\*. Эта запись означает, что символ “puts” использовался в ассемблерном коде, из которого был получен данный объектный файл, но не был определен; ассемблер сделал вывод о том, что символ должен быть определен где-то еще, и отразил это в таблице символов. То же самое относится и к символу “printf” и “InsertionSort”

Листинг 2.9. Таблица перемещений

|  |
| --- |
| main.o: file format elf64-littleriscv  RELOCATION RECORDS FOR [.text]:  OFFSET TYPE VALUE  000000000000000c R\_RISCV\_HI20 .LANCHOR0  000000000000000c R\_RISCV\_RELAX \*ABS\*  0000000000000010 R\_RISCV\_LO12\_I .LANCHOR0  0000000000000010 R\_RISCV\_RELAX \*ABS\*  0000000000000028 R\_RISCV\_HI20 .LC1  0000000000000028 R\_RISCV\_RELAX \*ABS\*  000000000000002c R\_RISCV\_LO12\_I .LC1  000000000000002c R\_RISCV\_RELAX \*ABS\*  0000000000000030 R\_RISCV\_CALL puts  0000000000000030 R\_RISCV\_RELAX \*ABS\*  0000000000000040 R\_RISCV\_HI20 .LC2  0000000000000040 R\_RISCV\_RELAX \*ABS\*  0000000000000046 R\_RISCV\_LO12\_I .LC2  0000000000000046 R\_RISCV\_RELAX \*ABS\*  000000000000004a R\_RISCV\_CALL printf  000000000000004a R\_RISCV\_RELAX \*ABS\*  000000000000005c R\_RISCV\_CALL InsertionSort  000000000000005c R\_RISCV\_RELAX \*ABS\*  0000000000000064 R\_RISCV\_HI20 .LC3  0000000000000064 R\_RISCV\_RELAX \*ABS\*  0000000000000068 R\_RISCV\_LO12\_I .LC3  0000000000000068 R\_RISCV\_RELAX \*ABS\*  000000000000006c R\_RISCV\_CALL puts  000000000000006c R\_RISCV\_RELAX \*ABS\*  0000000000000074 R\_RISCV\_HI20 .LC2  0000000000000074 R\_RISCV\_RELAX \*ABS\*  000000000000007a R\_RISCV\_LO12\_I .LC2  000000000000007a R\_RISCV\_RELAX \*ABS\*  000000000000007e R\_RISCV\_CALL printf  000000000000007e R\_RISCV\_RELAX \*ABS\*  0000000000000054 R\_RISCV\_BRANCH .L2  0000000000000088 R\_RISCV\_BRANCH .L3  insertion.o: file format elf64-littleriscv  RELOCATION RECORDS FOR [.text]:  OFFSET TYPE VALUE  0000000000000002 R\_RISCV\_BRANCH .L1  0000000000000010 R\_RISCV\_RVC\_JUMP .L6  000000000000001e R\_RISCV\_BRANCH .L1  000000000000002c R\_RISCV\_BRANCH .L4  0000000000000032 R\_RISCV\_BRANCH .L4  000000000000003c R\_RISCV\_BRANCH .L3  0000000000000040 R\_RISCV\_RVC\_JUMP .L4 |

Здесь содержится информация обо всех «неоконченных» инструкциях.

Записи типа “R\_RISCV\_RELAX” заносятся в таблицу перемещений в дополнение к записям типа “R\_RISCV\_CALL” и сообщают компоновщику, что пара инструкций, обеспечивающих вызов подпрограммы может быть оптимизирована.

##### *Компиляция*

Выполним компоновку следующей командой:

|  |
| --- |
| riscv64-unknown-elf-gcc.exe -march=rv64iac -mabi=lp64 -v main.o insertion.o |

Листинг 2.10. Фрагмент исполняемого файла

|  |
| --- |
| riscv64-unknown-elf-objdump -j .text -d -M no-aliases a.out >a.ds |
| 0000000000010156 <main>:  10156: 711d c.addi16sp sp,-96  10158: ec86 c.sdsp ra,88(sp)  1015a: e8a2 c.sdsp s0,80(sp)  1015c: e4a6 c.sdsp s1,72(sp)  1015e: e0ca c.sdsp s2,64(sp)  10160: fc4e c.sdsp s3,56(sp)  10162: 67f5 c.lui a5,0x1d  10164: c1878793 addi a5,a5,-1000 # 1cc18 <\_\_clzdi2+0x62>  10168: 638c c.ld a1,0(a5)  1016a: 6790 c.ld a2,8(a5)  1016c: 6b94 c.ld a3,16(a5)  1016e: 6f98 c.ld a4,24(a5)  10170: 739c c.ld a5,32(a5)  10172: e42e c.sdsp a1,8(sp)  10174: e832 c.sdsp a2,16(sp)  10176: ec36 c.sdsp a3,24(sp)  10178: f03a c.sdsp a4,32(sp)  1017a: f43e c.sdsp a5,40(sp)  1017c: 6575 c.lui a0,0x1d  1017e: bf050513 addi a0,a0,-1040 # 1cbf0 <\_\_clzdi2+0x3a>  10182: 296000ef jal ra,10418 <puts>  10186: 0020 c.addi4spn s0,sp,8  10188: 03010913 addi s2,sp,48  1018c: 84a2 c.mv s1,s0  1018e: 69f5 c.lui s3,0x1d  10190: 408c c.lw a1,0(s1)  10192: c0098513 addi a0,s3,-1024 # 1cc00 <\_\_clzdi2+0x4a>  10196: 1d6000ef jal ra,1036c <printf>  1019a: 0491 c.addi s1,4  1019c: ff249ae3 bne s1,s2,10190 <main+0x3a>  101a0: 002c c.addi4spn a1,sp,8  101a2: 4529 c.li a0,10  101a4: 030000ef jal ra,101d4 <InsertionSort>  101a8: 6575 c.lui a0,0x1d  101aa: c0850513 addi a0,a0,-1016 # 1cc08 <\_\_clzdi2+0x52>  101ae: 26a000ef jal ra,10418 <puts>  101b2: 64f5 c.lui s1,0x1d  101b4: 400c c.lw a1,0(s0)  101b6: c0048513 addi a0,s1,-1024 # 1cc00 <\_\_clzdi2+0x4a>  101ba: 1b2000ef jal ra,1036c <printf>  101be: 0411 c.addi s0,4  101c0: ff241ae3 bne s0,s2,101b4 <main+0x5e>  101c4: 4501 c.li a0,0  101c6: 60e6 c.ldsp ra,88(sp)  101c8: 6446 c.ldsp s0,80(sp)  101ca: 64a6 c.ldsp s1,72(sp)  101cc: 6906 c.ldsp s2,64(sp)  101ce: 79e2 c.ldsp s3,56(sp)  101d0: 6125 c.addi16sp sp,96  101d2: 8082 c.jr ra  00000000000101d4 <InsertionSort>:  101d4: 4785 c.li a5,1  101d6: 04a7d063 bge a5,a0,10216 <InsertionSort+0x42>  101da: 88ae c.mv a7,a1  101dc: fff5031b addiw t1,a0,-1  101e0: 4801 c.li a6,0  101e2: 557d c.li a0,-1  101e4: a809 c.j 101f6 <InsertionSort+0x22>  101e6: 0705 c.addi a4,1  101e8: 070a c.slli a4,0x2  101ea: 972e c.add a4,a1  101ec: c310 c.sw a2,0(a4)  101ee: 2805 c.addiw a6,1  101f0: 0891 c.addi a7,4  101f2: 02680263 beq a6,t1,10216 <InsertionSort+0x42>  101f6: 0048a603 lw a2,4(a7)  101fa: 0008071b addiw a4,a6,0  101fe: 87c6 c.mv a5,a7  10200: fe0843e3 blt a6,zero,101e6 <InsertionSort+0x12>  10204: 4394 c.lw a3,0(a5)  10206: fed670e3 bgeu a2,a3,101e6 <InsertionSort+0x12>  1020a: c3d4 c.sw a3,4(a5)  1020c: 377d c.addiw a4,-1  1020e: 17f1 c.addi a5,-4  10210: fea71ae3 bne a4,a0,10204 <InsertionSort+0x30>  10214: bfc9 c.j 101e6 <InsertionSort+0x12>  10216: 8082 c.jr ra |

Мы видим, что адресация для вызовов функций изменилась на абсолютную.

##### **3.Создание статической библиотеки**

Выделим функцию InsertionSort в отдельную статическую библиотеку. Для этого надо получить объектный файл insetion.o и собрать библиотеку.

|  |
| --- |
| riscv64-unknown-elf-gcc -march=rv64iac -mabi=lp64 -O1 -c insertion.c -o insertion.o  riscv64-unknown-elf-ar -rsc lib.a insertion.o |

Рассмотрим список символов библиотеки:

Листинг 3.1 Список символов lib.a

|  |
| --- |
| riscv64-unknown-elf-nm lib.a |
| insertion.o:  0000000000000042 t .L1  0000000000000030 t .L3  0000000000000012 t .L4  0000000000000022 t .L6  0000000000000000 T InsertionSort |

В выводе утилиты “nm” кодом “T” обозначаются символы, определенные в

соответствующем объектном файле.Теперь, имея собранную библиотеку, создадим исполняемый файл тестовой программы *‘main.c”* с помощью следующей команды:

|  |
| --- |
| riscv64-unknown-elf-gcc -march=rv64iac -mabi=lp64 -O1 main.c lib.a -o main.out |

Убедимся, что в состав программы вошло содержание объектного файла insertion.o, при помощи таблицы символов исполняемого файла

Листинг 3.1 Фрагмент списка символов main.out.

|  |
| --- |
| riscv64-unknown-elf-objdump -t main.out >main.ds |
| main.out: file format elf64-littleriscv  SYMBOL TABLE:  ----------------------------  0000000000019c9c g F .text 0000000000000024 \_\_ascii\_wctomb  00000000000101d4 g F .text 0000000000000044 InsertionSort  0000000000019f9a g F .text 0000000000000020 \_fiprintf\_r |

Процесс выполнения команд выше можно заменить make-файлами, которые произведут создание библиотеки и сборку программы.

Листинг 3.2. Makefile для создания статической библиотеки “makeLibrary”

|  |
| --- |
| CC=riscv64-unknown-elf-gcc  AR=riscv64-unknown-elf-ar  CFLAGS=-march=rv64iac -mabi=lp64  all: lib  lib: insertion.o  $(AR) -rsc lib.a insertion.o  del -f \*.o  insertion.o: insertion’s  $(CC) $(CFLAGS) -c insertion.c -o insertion.o |

Листинг 3.3. Makefile для сборки исполняемого файла “makeApp”

|  |
| --- |
| TARGET=main.out  CC=riscv64-unknown-elf-gcc  CFLAGS=-march=rv64iac -mabi=lp64  all:  make -f makeLibrary  $(CC) $(CFLAGS) main.c lib.a -o $(TARGET)  del -f \*.0 \*.a |

Теперь с помощью GNU make выполним сначала makeLibrary, а затем makeApp, для создания библиотеки.

|  |
| --- |
|  |
| Рис. 3.1 Выполнение make файлов. |

Посмотрим таблицу символов полученного c помощью makefile исполняемого файла:

Листинг 3.3 Фрагмент списка символов main.out (makefile).

|  |
| --- |
| riscv64-unknown-elf-objdump -t main.out >main.ds |
| main.out: file format elf64-littleriscv  SYMBOL TABLE:  ----------------------------  0000000000019c9c g F .text 0000000000000024 \_\_ascii\_wctomb  00000000000101d4 g F .text 0000000000000044 InsertionSort  0000000000019f9a g F .text 0000000000000020 \_fiprintf\_r |

Мы видим, что исполняемый файл аналогичен созданному в терминале файлу.

##### **Вывод**

В ходе лабораторной работы изучена пошаговая компиляция программы на языке C. Также была создана статическая библиотека и произведена сборка программы с помощью Makefile.