Working with Assembly loops and Analyzing performance benefits of loop unrolling.

Terrence Viloria

Step1:

1. //macro
2. .section .bss
4. .macro write str, strlen
5. movl $4, %eax
6. movl $1, %ebx
7. movl \str, %ecx
8. movl \strlen, %edx
9. int $0x80
10. .endm
12. .section .text
13. .globl \_start
15. \_start:
16. write $msg, $len1
17. write $msg2, $len2
18. write $msg3, $len3
20. \_exit:
21. movl $1, %eax
22. movl $0, %ebx
23. int $0x80
24. .section .data
25. msg:
26. .ascii "Hello, programmers!\n"
27. len1 = . - msg
29. msg2:
30. .ascii "Welcome to the world of,\n"
31. len2 = . - msg2
33. msg3:
34. .ascii "Linux assembly programming!\n"
35. len3 = . - msg3

Step2:

1. #include <stdio.h>
3. // a
4. int aFor(int first, int last) {
5. int sum = 0;
6. int i;
7. for(i = first; i <= last; i++) {
8. sum += i;
9. }
10. return sum;
12. }
14. // b
15. int bWhile(int first, int last) {
16. int sum = 0;
17. int i = first;
18. while(i <= last){
19. sum+=i;
20. i++;
21. }
22. return sum;
23. }
25. //c
26. int cDoWhile(int first, int last){
27. int sum = 0;
28. int i = first;
29. do{
30. sum+=i;
31. i++;
32. }while (i<=last);
34. return sum;
36. }
38. //d
39. int dGoTo(int first, int last){
40. int sum = 0;
41. int i;
42. i = first;
44. jump:
45. sum+=i;
46. i++;
48. if(i <= last){
49. goto jump;
50. }
51. return sum;
53. }


57. int main(){
59. int first, last;
60. printf("Enter first number: \n");
61. scanf("%d", &first);
63. printf("Enter last number: \n");
64. scanf("%d", &last);
66. int sumFor = aFor(first, last);
67. printf("For loop: %d \n", sumFor);
69. int sumWhile = bWhile(first, last);
70. printf("While loop: %d \n", sumWhile);
72. int sumDoWhile = cDoWhile(first, last);
73. printf("do while loop: %d \n", sumDoWhile);
75. int sumGoTo = dGoTo(first, last);
76. printf("GoTo loop: %d", sumGoTo);
78. return 0;
79. }

Step2 Assembly Code:

1. .file "step2.c"
2. .text
3. .globl \_aFor
4. .def \_aFor; .scl 2; .type 32; .endef
5. \_aFor:
6. LFB13:
7. .cfi\_startproc
8. movl 4(%esp), %edx
9. movl 8(%esp), %ecx
10. cmpl %ecx, %edx
11. jg L4
12. addl $1, %ecx
13. movl $0, %eax
14. L3:
15. addl %edx, %eax
16. addl $1, %edx
17. cmpl %ecx, %edx
18. jne L3
19. ret
20. L4:
21. movl $0, %eax
22. ret
23. .cfi\_endproc
24. LFE13:
25. .globl \_bWhile
26. .def \_bWhile; .scl 2; .type 32; .endef
27. \_bWhile:
28. LFB14:
29. .cfi\_startproc
30. movl 4(%esp), %edx
31. movl 8(%esp), %ecx
32. cmpl %ecx, %edx
33. jg L9
34. addl $1, %ecx
35. movl $0, %eax
36. L8:
37. addl %edx, %eax
38. addl $1, %edx
39. cmpl %ecx, %edx
40. jne L8
41. ret
42. L9:
43. movl $0, %eax
44. ret
45. .cfi\_endproc
46. LFE14:
47. .globl \_cDoWhile
48. .def \_cDoWhile; .scl 2; .type 32; .endef
49. \_cDoWhile:
50. LFB15:
51. .cfi\_startproc
52. movl 4(%esp), %edx
53. movl 8(%esp), %ecx
54. movl $0, %eax
55. L12:
56. addl %edx, %eax
57. addl $1, %edx
58. cmpl %ecx, %edx
59. jle L12
60. ret
61. .cfi\_endproc
62. LFE15:
63. .globl \_dGoTo
64. .def \_dGoTo; .scl 2; .type 32; .endef
65. \_dGoTo:
66. LFB16:
67. .cfi\_startproc
68. movl 4(%esp), %edx
69. movl 8(%esp), %ecx
70. movl $0, %eax
71. L15:
72. addl %edx, %eax
73. addl $1, %edx
74. cmpl %ecx, %edx
75. jle L15
76. ret
77. .cfi\_endproc
78. LFE16:
79. .def \_\_\_main; .scl 2; .type 32; .endef
80. .section .rdata,"dr"
81. LC0:
82. .ascii "Enter first number: \0"
83. LC1:
84. .ascii "%d\0"
85. LC2:
86. .ascii "Enter last number: \0"
87. LC3:
88. .ascii "For loop: %d \12\0"
89. LC4:
90. .ascii "While loop: %d \12\0"
91. LC5:
92. .ascii "do while loop: %d \12\0"
93. LC6:
94. .ascii "GoTo loop: %d\0"
95. .text
96. .globl \_main
97. .def \_main; .scl 2; .type 32; .endef
98. \_main:
99. LFB17:
100. .cfi\_startproc
101. pushl %ebp
102. .cfi\_def\_cfa\_offset 8
103. .cfi\_offset 5, -8
104. movl %esp, %ebp
105. .cfi\_def\_cfa\_register 5
106. andl $-16, %esp
107. subl $32, %esp
108. call \_\_\_main
109. movl $LC0, (%esp)
110. call \_puts
111. leal 28(%esp), %eax
112. movl %eax, 4(%esp)
113. movl $LC1, (%esp)
114. call \_scanf
115. movl $LC2, (%esp)
116. call \_puts
117. leal 24(%esp), %eax
118. movl %eax, 4(%esp)
119. movl $LC1, (%esp)
120. call \_scanf
121. movl 24(%esp), %edx
122. movl 28(%esp), %eax
123. cmpl %eax, %edx
124. jl L24
125. addl $1, %edx
126. movl $0, %ecx
127. L19:
128. addl %eax, %ecx
129. addl $1, %eax
130. cmpl %edx, %eax
131. jne L19
132. L18:
133. movl %ecx, 4(%esp)
134. movl $LC3, (%esp)
135. call \_printf
136. movl 24(%esp), %edx
137. movl 28(%esp), %eax
138. cmpl %eax, %edx
139. jl L25
140. addl $1, %edx
141. movl $0, %ecx
142. L21:
143. addl %eax, %ecx
144. addl $1, %eax
145. cmpl %eax, %edx
146. jne L21
147. L20:
148. movl %ecx, 4(%esp)
149. movl $LC4, (%esp)
150. call \_printf
151. movl 24(%esp), %ecx
152. movl 28(%esp), %eax
153. movl $0, %edx
154. L22:
155. addl %eax, %edx
156. addl $1, %eax
157. cmpl %eax, %ecx
158. jge L22
159. movl %edx, 4(%esp)
160. movl $LC5, (%esp)
161. call \_printf
162. movl 24(%esp), %ecx
163. movl 28(%esp), %eax
164. movl $0, %edx
165. L23:
166. addl %eax, %edx
167. addl $1, %eax
168. cmpl %eax, %ecx
169. jge L23
170. movl %edx, 4(%esp)
171. movl $LC6, (%esp)
172. call \_printf
173. movl $0, %eax
174. leave
175. .cfi\_remember\_state
176. .cfi\_restore 5
177. .cfi\_def\_cfa 4, 4
178. ret
179. L24:
180. .cfi\_restore\_state
181. movl $0, %ecx
182. jmp L18
183. L25:
184. movl $0, %ecx
185. jmp L20
186. .cfi\_endproc
187. LFE17:
188. .ident "GCC: (i686-posix-dwarf-rev0, Built by MinGW-W64 project) 8.1.0"
189. .def \_puts; .scl 2; .type 32; .endef
190. .def \_scanf; .scl 2; .type 32; .endef
191. .def \_printf; .scl 2; .type 32; .endef

 Analyzing the assembly code generated by the compiler for each loop functions. There were not much of a difference in terms of the amount of iterations, number of operations and amount of registers used. This is primarily due to the same logic used.

In the assembly code generated, the for and while loop share the same similarities; on the other hand, the “do while” and “goto” loop functions share their own similarities. The difference between the two groups is for and while loop check the conditions twice. Once before the first iteration and another check in between each successive iteration. While the other two checks the conditions last hence they are “exit-condition” type of loops.

Step3:

Loop Rolled C code

1. #include <stdio.h>

4. // Loop rolled
6. int sum(int arr[], int n) {
7. int sum = 0;
9. for (int i = 0; i < n; i++)
10. {
11. sum+=arr[i];
12. }
13. return sum;
15. }
17. int main(){
18. int elements[50] = {
19. 1,2,3,4,5,6,7,8,9,10,
20. 11,12,13,14,15,16,17,18,19,20,
21. 21,22,23,24,25,26,27,28,29,30,
22. 31,32,33,34,35,36,37,38,39,40,
23. 41,42,43,44,45,46,47,48,49,50
24. };
26. int n = sizeof(elements) / sizeof(elements[0]);
27. int rolled = sum(elements,n);
28. printf("Sum of elements: %d \n", rolled);
30. return 0;
31. }

Loop Rolled Assembly Code

1. .file "step3.c"
2. .text
3. .globl \_sum
4. .def \_sum; .scl 2; .type 32; .endef
5. \_sum:
6. LFB13:
7. .cfi\_startproc
8. movl 4(%esp), %ecx
9. movl 8(%esp), %eax
10. testl %eax, %eax
11. jle L4
12. movl %ecx, %edx
13. leal (%ecx,%eax,4), %ecx
14. movl $0, %eax
15. L3:
16. addl (%edx), %eax
17. addl $4, %edx
18. cmpl %ecx, %edx
19. jne L3
20. ret
21. L4:
22. movl $0, %eax
23. ret
24. .cfi\_endproc
25. LFE13:
26. .def \_\_\_main; .scl 2; .type 32; .endef
27. .section .rdata,"dr"
28. LC1:
29. .ascii "Sum of elements: %d \12\0"
30. .data
31. .align 32
32. LC0:
33. .long 1
34. .long 2
35. .long 3
36. .long 4
37. .long 5
38. .long 6
39. .long 7
40. .long 8
41. .long 9
42. .long 10
43. .long 11
44. .long 12
45. .long 13
46. .long 14
47. .long 15
48. .long 16
49. .long 17
50. .long 18
51. .long 19
52. .long 20
53. .long 21
54. .long 22
55. .long 23
56. .long 24
57. .long 25
58. .long 26
59. .long 27
60. .long 28
61. .long 29
62. .long 30
63. .long 31
64. .long 32
65. .long 33
66. .long 34
67. .long 35
68. .long 36
69. .long 37
70. .long 38
71. .long 39
72. .long 40
73. .long 41
74. .long 42
75. .long 43
76. .long 44
77. .long 45
78. .long 46
79. .long 47
80. .long 48
81. .long 49
82. .long 50
83. .text
84. .globl \_main
85. .def \_main; .scl 2; .type 32; .endef
86. \_main:
87. LFB14:
88. .cfi\_startproc
89. pushl %ebp
90. .cfi\_def\_cfa\_offset 8
91. .cfi\_offset 5, -8
92. movl %esp, %ebp
93. .cfi\_def\_cfa\_register 5
94. pushl %edi
95. pushl %esi
96. andl $-16, %esp
97. subl $224, %esp
98. .cfi\_offset 7, -12
99. .cfi\_offset 6, -16
100. call \_\_\_main
101. leal 24(%esp), %edi
102. movl $LC0, %esi
103. movl $50, %ecx
104. rep movsl
105. movl $50, 4(%esp)
106. leal 24(%esp), %eax
107. movl %eax, (%esp)
108. call \_sum
109. movl %eax, 4(%esp)
110. movl $LC1, (%esp)
111. call \_printf
112. movl $0, %eax
113. leal -8(%ebp), %esp
114. popl %esi
115. .cfi\_restore 6
116. popl %edi
117. .cfi\_restore 7
118. popl %ebp
119. .cfi\_restore 5
120. .cfi\_def\_cfa 4, 4
121. ret
122. .cfi\_endproc
123. LFE14:
124. .ident "GCC: (i686-posix-dwarf-rev0, Built by MinGW-W64 project) 8.1.0"
125. .def \_printf; .scl 2; .type 32; .endef

Loop Unrolled

1. #include <stdio.h>

4. // Loop unrolling
6. int sum\_unrolled(int arr[], int n) {
7. int sum = 0;
9. for (int i = 0; i < n; i+=2)
10. {
11. sum+=arr[i];
13. sum+=arr[i+1];
15. }
16. return sum;
18. }

21. int main(){
22. int elements[50] = {
23. 1,2,3,4,5,6,7,8,9,10,
24. 11,12,13,14,15,16,17,18,19,20,
25. 21,22,23,24,25,26,27,28,29,30,
26. 31,32,33,34,35,36,37,38,39,40,
27. 41,42,43,44,45,46,47,48,49,50
28. };

31. int n = sizeof(elements) / sizeof(elements[0]);
32. int num = sum\_unrolled(elements,n);
33. printf("sum is %d", num);

36. return 0;
37. }

Loop Unrolled Assembly Code

1. .file "step3\_unrolled.c"
2. .text
3. .globl \_sum\_unrolled
4. .def \_sum\_unrolled; .scl 2; .type 32; .endef
5. \_sum\_unrolled:
6. LFB13:
7. .cfi\_startproc
8. movl 4(%esp), %ecx
9. movl 8(%esp), %eax
10. testl %eax, %eax
11. jle L4
12. movl %ecx, %edx
13. leal -4(,%eax,4), %eax
14. andl $-8, %eax
15. leal 8(%ecx,%eax), %ecx
16. movl $0, %eax
17. L3:
18. addl (%edx), %eax
19. addl 4(%edx), %eax
20. addl $8, %edx
21. cmpl %ecx, %edx
22. jne L3
23. ret
24. L4:
25. movl $0, %eax
26. ret
27. .cfi\_endproc
28. LFE13:
29. .def \_\_\_main; .scl 2; .type 32; .endef
30. .section .rdata,"dr"
31. LC1:
32. .ascii "sum is %d\0"
33. .data
34. .align 32
35. LC0:
36. .long 1
37. .long 2
38. .long 3
39. .long 4
40. .long 5
41. .long 6
42. .long 7
43. .long 8
44. .long 9
45. .long 10
46. .long 11
47. .long 12
48. .long 13
49. .long 14
50. .long 15
51. .long 16
52. .long 17
53. .long 18
54. .long 19
55. .long 20
56. .long 21
57. .long 22
58. .long 23
59. .long 24
60. .long 25
61. .long 26
62. .long 27
63. .long 28
64. .long 29
65. .long 30
66. .long 31
67. .long 32
68. .long 33
69. .long 34
70. .long 35
71. .long 36
72. .long 37
73. .long 38
74. .long 39
75. .long 40
76. .long 41
77. .long 42
78. .long 43
79. .long 44
80. .long 45
81. .long 46
82. .long 47
83. .long 48
84. .long 49
85. .long 50
86. .text
87. .globl \_main
88. .def \_main; .scl 2; .type 32; .endef
89. \_main:
90. LFB14:
91. .cfi\_startproc
92. pushl %ebp
93. .cfi\_def\_cfa\_offset 8
94. .cfi\_offset 5, -8
95. movl %esp, %ebp
96. .cfi\_def\_cfa\_register 5
97. pushl %edi
98. pushl %esi
99. andl $-16, %esp
100. subl $224, %esp
101. .cfi\_offset 7, -12
102. .cfi\_offset 6, -16
103. call \_\_\_main
104. leal 24(%esp), %edi
105. movl $LC0, %esi
106. movl $50, %ecx
107. rep movsl
108. movl $50, 4(%esp)
109. leal 24(%esp), %eax
110. movl %eax, (%esp)
111. call \_sum\_unrolled
112. movl %eax, 4(%esp)
113. movl $LC1, (%esp)
114. call \_printf
115. movl $0, %eax
116. leal -8(%ebp), %esp
117. popl %esi
118. .cfi\_restore 6
119. popl %edi
120. .cfi\_restore 7
121. popl %ebp
122. .cfi\_restore 5
123. .cfi\_def\_cfa 4, 4
124. ret
125. .cfi\_endproc
126. LFE14:
127. .ident "GCC: (i686-posix-dwarf-rev0, Built by MinGW-W64 project) 8.1.0"
128. .def \_printf; .scl 2; .type 32; .endef

Analysis of the performance of Loop Unrolling.

I compiled each program 10 times as well as executed both programs 10 times to get an average time. In terms of performance there was not a significant increase as this may be due to only computing 50 elements. This optimization may see a better increase in performance for larger programs. Though in the graph above it is evident that it decreases the time to compile and execute as loop unrolling increases the number of elements to be computed on each iteration step.