


Dokumentation Assembler (Medianfilter)

Aufgabenname: Projektaufgabe 1.404:

Nachrichtentechnik – Filtern V

Gruppennummer: ss18-g33

- Projektleiter: Ruben Bachmann
(Matrikelnummer: 03693902)
 - Dokumentation: Franziska Steinle
(Matrikelnummer: 03702739)
 - Abschlussvortrag: Roland Würsching
(Matrikelnummer: 03648533)
- 

Inhalt

Allgemeiner Teil.....	2
Lösungsansatz.....	2
Abweichung von der Spezifikation	2
Bewertung des Lösungsansatz	2
Anwenderdokumentation	3
Entwicklerdokumentation	4
Hauptfunktion:	4
Kopierfunktion:.....	4
Sortierfunktion:	4
Swapfunktion:	6
Median und Speicherfunktion:.....	6
Tests:	7
Main-funktion:.....	7
Menü-Funktion:.....	7
Random-Test:	7
Custom-Test:	7
Funktionen-Test:	7
Bekannte Probleme:.....	8
Potentielle Weiterentwicklungen:	8

Allgemeiner Teil

Lösungsansatz

In diesem Projekt wurde ein Medianfilter, der ein verrauschtes Signal in ein gefiltertes Signal umwandelt, in Assembler realisiert. Dazu wurden vier Unterfunktionen geschaffen: Die Kopierfunktion (`copy_array`), die Sortierfunktion (`sort_array`), die Swapfunktion (`swap_int`) und die Speicherfunktion (`find_medians`). In der Hauptfunktion (`filter5`) werden zuerst die Eingaben auf Korrektheit überprüft und bei falschen Eingaben wird eine Error Message auf die Zieladresse gelegt und die Funktion beendet. Daraufhin wird die Kopierfunktion aufgerufen, die das vorliegende Signal an die Zieladresse kopiert, damit das ursprüngliche Signal beim Sortieren nicht überschrieben wird. Danach ruft die Hauptfunktion die Sortierfunktion auf. Diese Funktion teilt das verrauschte Signal in Blöcke auf und sortiert diese nacheinander mithilfe von Selection Sort, dabei wird die Swapfunktion aufgerufen, die zwei Werte im Speicher vertauscht. Als letztes ruft die Hauptfunktion, die Speicherfunktion auf. In dieser wird der Median jedes Blockes gefunden und an die richtige Speicherstelle gelegt. Schlussendlich wird die Funktion beendet.

Abweichung von der Spezifikation

Im folgenden Teil werden die Abweichungen von der Spezifikation behandelt. Wir schauen uns die Funktionen einzeln an. In der Kopierfunktion und Swapfunktion wurde sich genau an der Spezifikation orientiert und nichts verändert. Bei der Hauptfunktion wurden allerdings ein paar Kleinigkeiten geändert, so wurde in der Spezifikation nicht erwähnt, wie die Fehler bei falscher Eingabe behandelt werden sollen. Diese Fehlerbehandlung wurde in der Implementierung hinzugefügt, indem ein bestimmter Fehlercode auf die Zieladresse gelegt wird.

Die Funktionen der Sortierfunktion wurden erweitert, so sortiert sie nicht nur einen Block, sondern zerteilt das verrauschte Signal in die Blocks und sortiert sie dann jeweils. Das Einteilen in Blocks wurde also von der Hauptfunktion in die Sortierfunktion verschoben. Für das Finden des Median und an der richtigen Position abspeichern wurde eine weitere Funktion geschaffen, anstatt wie geplant diese Operationen direkt in der Hauptfunktion durchzuführen. Durch diese Veränderungen gewinnt die Hauptfunktion an Übersichtlichkeit und kümmert sich nur noch um die Fehlererkennung und das Aufrufen von Funktionen.

Zudem sortiert die Sortierfunktion einen Block vollständig und nicht bis zur Hälfte wie geplant. Dadurch gewinnt die Funktion an Übersichtlichkeit, da so ein Register weniger verwendet werden muss, in dem dann die Blocklänge/2 als Vergleich gespeichert werden müsste. Auch ist die Implementierung dadurch vereinfacht worden. Die Laufzeit wurde verschlechtert, allerdings wurde in dieser Aufgabe der Laufzeit keine Beachtung geschenkt.

Bewertung des Lösungsansatz

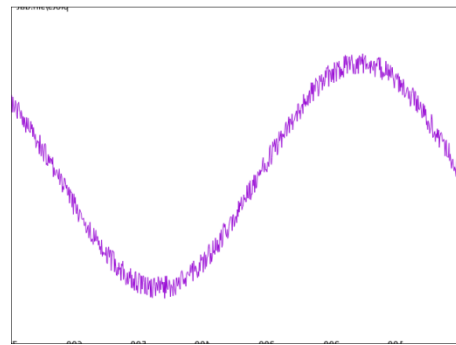
Es wurde entschieden anstatt Bubble Sort Selection Sort zum Sortieren zu verwenden. Die Vorteile, die in der Spezifikation genannt wurde: die Swapfunktion wird weniger aufgerufen als beim Bubble Sort und die einfachere Implementierung. Die Swapfunktion wird wie angenommen in jedem Block nur maximal $N(\text{Blocklänge}) - 1$ mal aufgerufen und die Funktion ist durch zwei Schleifen einfach zu implementieren. Der Nachteil an Selection Sort war, dass mehr Register benötigt werden, als beim Bubble Sort, allerdings war das bei der Implementierung nicht hinderlich, da genug Register frei, also unbenutzt waren.

Anwenderdokumentation

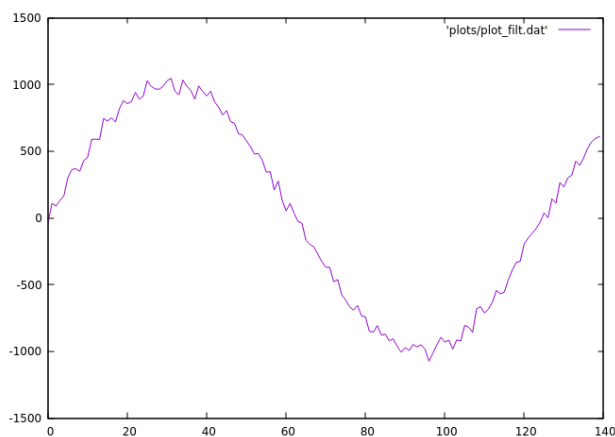
In dem C-Programm wird mithilfe des Assembler Programms ein verrauschtes Signal a seinem unverrauschten Signal b nähergebracht, indem die Werte von a in gleich große Abschnitte mit ungerader Länge eingeteilt werden. Diese Abschnitte werden dann sortiert und der Median, also der Wert, der nach dem Sortieren in der Mitte liegt, wird abgespeichert.

Dadurch wird versucht, das Signal b wiederherzustellen, was natürlich nicht immer 100% gelingt. Um dieses Programm auszuführen, wird als Betriebssystem 64-Bit

Ubuntu 16.04 benötigt. Auch nasm wurde zur Implementierung der Filterfunktion verwendet, wird



Beispiel eines ungefilterten Signals



Beispiel eines gefilterten Signals

also benötigt, um diese zu kompilieren. Ebenso gcc, um das C-Rahmenprogramm und die Tests zu kompilieren. Um das Projekt auszuführen muss am Terminal über den richtigen Dateipfad make aufgerufen werden. Dadurch wird dann das Makefile aufgerufen, dass dann das Main C-Programm aufruft, das dann wiederum die Assembler Datei filter5 mit zufälligen Werten aufruft. Innerhalb von Main wird erst das ungefilterte Signal, dann das gefilterte Signal in den Terminal ausgegeben.

```
user@ERA-VM:~/ERA/ASMS$ ./main
Creating signal ...

NOISY SIGNAL
-----
-22 26 29 12 -5 -48 -20 -37 -25 -17 8 -19 13 40 8 -38 15 10 -28 21 4 27 34 -35 29 -29 -24 16
-28 -11 42 -48 -34 21 14 -37 -24 -4 29 1 29 -13 -15 -8 29 43 -44 -6 -44 -22 17 -38 7 -47 29 -
14 27 -45 5 49 -5 -3 3 -37 21 19 -24 -1 -35 -45 2 -4 44 37 -10 -24 -17 -4 20 41 -24 40 5 34 4
3 -16 22 20 42 29 -29 -13 29 -24 -48 0 45 -22 49 12 36 -47 -42
-----

Applying median filter ...

FILTERED SIGNAL
-----
12 -25 8 10 27 -24 14 -4 -8 -22 7 5 -3 -24 2 -4 34 22 -24 12
-----

Completed. Exiting ...
```

Beispiel einer Ausgabe im Terminal

Entwicklerdokumentation

Hauptfunktion:

Zuerst wird genauer auf die Implementierung des Assemblerteils eingegangen. Dieser beginnt mit der Hauptfunktion filter5. In dieser wird zu Beginn überprüft, ob die Eingabe korrekt ist, also ob die Blocklänge größer 0 und ungerade ist. Ist das nicht der Fall wird eine Fehlermeldung, also eine festgelegte Zahlenfolge, auf die Zieladresse gelegt und danach das Programm beendet. Sind die Eingaben korrekt, wird zunächst copy_array aufgerufen, gefolgt von sort_array und find_median.

```
17 ; Main function: Check parameters, copy array, sort blocks, find and write medians
18 ; edi: int N (block length)
19 ; esi: int *start (start address)
20 ; edx: int length (signal length)
21 ; ecx: int *dest (destination address)
22 filter5:
23     ; check if block length is positive
24     cmp edi, 0
25     jle error_blockLength
26
27     ; check if block length is odd
28     test edi, 1
29     je error_blockLength
30
31     ; check if signal length is non-negative
32     test edx, edx
33     js error_signalLength
34
35     ; Copy the array from start to dest
36     call copy_array
37
38     ; Sort the copied array by blocks
39     call sort_array
40
41     ; Find medians and write them to dest
42     call find_medians
43
44     ; End function
45     ret
```

Kopierfunktion:

In copy_array wird nach der Sicherung der Register eine Schleife begonnen, in der die verbleibende Signallänge als Abbruchbedingung fungiert. In der Schleife werden jeweils 4 Byte aus der Startadresse über einen kurzen Zwischenspeicher auf die Zieladresse gelegt, da eine direkt Verschiebung in Assembler nicht möglich ist. Daraufhin werden beide Adressen um 4 erhöht und die verbleibende Signallänge dekrementiert. Nach dem Abbruch der Schleife werden die Register wieder in den Ursprungszustand versetzt und die Methode wird beendet.

```
47
48 ; Copy the array from start to dest
49 ; esi: start
50 ; edi: dest
51 copy_array:
52     push rax
53     push rsi
54     push rcx
55     push rdx
56
57     ; Loop over the array at start and copy integers to dest
58     copy_loop:
59         cmp edx, 0 ; if no values are left, end loop
60         je end_copy
61
62         mov eax, [esi] ; copy current value (4 bytes)
63         mov [ecx], eax ; post value at destination (4 bytes)
64
65         add esi, 4 ; move to next int (4 bytes)
66         add ecx, 4 ; move to next int (4 bytes)
67         dec edx ; decrement counter
68
69         jmp copy_loop
70
71     end_copy:
72         pop rdx
73         pop rcx
74         pop rsi
75         pop rax
76         ret
```

Sortierfunktion:

Nun werden wir sort_array genauer betrachten. Erst werden natürlich die Register gesichert und dann wird das Offset auf 0 und die Blockstartadresse auf die Zieladresse gesetzt. Dann beginnt wieder eine Schleife, die abbricht, wenn das Offset größer als die Signallänge ist, nachdem das Offset um die Blocklänge erhöht wurde. Der Restblock, der kleiner als die normale Blockgröße ist, wird

ignoriert und nicht sortiert.

In der Schleife wird

sort_block aufgerufen und

dann die Blockstartadresse

um 4 mal die Blocklänge

erhöht, da es sich um 32 Bit

Werte handelt, jede Adresse

zeigt ja auf 8 Bit, deshalb

muss die Blocklänge mal 4

genommen werden. Um

diese Rechnung zu realisieren

wird zuerst die Blocklänge

mal 4 genommen, indem sie

2 mal nach links geshiftet

wird, dann wird addiert und

dann wieder 2 mal nach

rechts geshiftet, um die

Original Blocklänge wiederherzustellen. Dies ist allerdings

nur möglich, da die Blocklänge, einen eher kleinen Wert hat

und so durch das shiften keine Werte verloren gehen. Nach

dem Schleifenabbruch werden die Register

wiederhergestellt und die Funktion beendet.

```
79 ; Divide the array at dest into blocks of length N and sort them in ascending order
80 ; Incomplete block at the end will be ignored
81 ; edi: N
82 ; edx: length
83 ; esi: dest
84 ; eax: offset
85 ; r10: block start address
86 sort_array:
87     push rax
88     push r10
89
90     mov eax, 0
91     mov r10d, ecx
92
93 sort_array_loop:
94     ; offset += N
95     add eax, edi
96     ; if offset > length, end loop
97     cmp eax, edx
98     jg end_sort_array
99
100 ; sort block
101 call sort_block
102
103 ; dest += N*4
104 shl edi, 2
105 add r10d, edi
106 shr edi, 2
107
108 jmp sort_array_loop
109
```

```
109
110 end_sort_array:
111     pop r10
112     pop rax
113     ret
114
```

In der Hilfsfunktion sort_block wird der Selection Sort realisiert, dazu gibt es zwei Schleifen, die

ineinander liegen. In der äußeren Schleife wird der Block einmal durchgegangen, sie wird also sooft

ausgeführt, wie ein Block groß ist, mithilfe eines Zählers, der bei 0 beginnt. Die innere Schleife

beginnt jeweils mit dem Wert des Zählers aus der äußeren Schleife und endet bei der Blocklänge. Die

```
115
116 ; Sort single block using the selection sort algorithm
117 ; edi: N
118 ; r10d: block start address
119 sort_block:
120     push rax
121     push rbx
122     push rsi
123     push r8
124     push r9
125
126     mov eax, 0 ; outer loop counter
127     mov r8d, r10d ; outer loop address
128
129 sort_outer_loop:
130     cmp eax, edi
131     je end_outer_loop
132
133     mov ebx, eax ; inner loop counter
134     mov r9d, r8d ; address of current minimum
135     mov esi, [r8d] ; value of hold current minimum
136
137 sort_inner_loop:
138     inc ebx
139     ; if end is reached, then minimum is found
140     cmp ebx, edi
141     je end_inner_loop
142
143     ; check if minimum needs to be updated
144     cmp esi, [r10d + 4*ebx]
145     jle sort_inner_loop
146
```

```
147 ; set new minimum:
148 mov r9d, r10d
149 shl ebx, 2
150 add r9d, ebx
151 shr ebx, 2
152 mov esi, [r9d]
153
154 jmp sort_inner_loop
155
156 end_inner_loop:
157 call swap_int
158 inc eax
159 add r8d, 4
160
161 jmp sort_outer_loop
162
163 end_outer_loop:
164 pop r9
165 pop r8
166 pop rsi
167 pop rbx
168 pop rax
169 ret
```

äußere Schleife setzt die Adresse des aktuellen Minimums und deren Wert, zur Verwendung in der inneren Schleife. Nach der inneren Schleife wird die Swapfunktion swap_int aufgerufen und der Zähler und die Adresse des aktuellen Minimums erhöht. In der inneren Schleife zuerst der Schleifenzähler erhöht und dann wird das aktuelle Minimum mit dem nächsten Wert verglichen, ist dieser größer kehren wir an den Schleifenanfang zurück. Wenn er kleiner ist, werden die Adresse und der Wert des neuen Minimums gespeichert, dabei wird wieder die Technik mit dem Shiften der Blocklänge verwendet, um die richtige Speicheradresse zu bekommen. Ist die äußere Schleife beendet, wird auch die Hilfsfunktion beendet.

Swapfunktion:

In dieser Funktion wird das aktuelle Minimum, das in der inneren Schleife von sort_block gewonnen wurde, durch Zwischenspeichern der Werte, mit dem aktuellen Element der äußeren Schleife vertauscht.

```

172 ; swap two integers stored at r8d and r9d
173 swap_int:
174     push rax
175     push rbx
176
177     mov eax, [r8d]    ; save first value
178     mov ebx, [r9d]    ; save second value
179     mov [r8d], ebx    ; write second value
180     mov [r9d], eax    ; write first value
181
182     pop rbx
183     pop rax
184     ret
185

```

Median und Speicherfunktion:

```

185
186 ; Find medians and write them to dest
187 ; eax: number of blocks
188 ; ebx: current address
189 ; edx: counter
190 ; esi: temp value for median
191 find_medians:
192     push rax
193     push rbx
194     push rdx
195     push rsi
196
197     ; #blocks = length / N (rounded off)
198     mov eax, edx
199     mov edx, 0
200     div edi
201
202     mov edx, 0
203
204     ; find first median:
205     ; dest + 4 * (N-1)/2 = dest + 2N - 2
206     mov ebx, edi
207     shl ebx, 1
208     sub ebx, 2
209     add ebx, ecx
210

```

verrechnet mit der Zahl des aktuellen Blocks. Dann wird der nächste Median bestimmt, indem einfach die vierfache Blocklänge dazu addiert wird, wie schon erwähnt bezieht sich das Vierfach auf die 32-bit Zahlen, dazu wird wieder das Shiften verwendet. Danach werden die Register wiederhergestellt und das gefilterte Signal liegt jetzt an seiner Zieladresse vor.

In der Funktion find_median werden zunächst die Register gesichert, dann wird die Anzahl der Blocks bestimmt, dazu muss der Divisor erst nach eax geschrieben werden und das Ergebnis findet sich dann auch in eax. Daraufhin wird der erste Median errechnet, mit der Formel: Adresse+ 2*Blocklänge-2, wobei mit der Adresse die Zieladresse gemeint ist, an der sich ja die sortierten Blöcke befinden. Dann beginnt eine Schleife, die die bereits errechnete Anzahl der Blöcke als Abbruch hat, der Zähler beginnt bei 0. Nun wird der Median über seine Adresse bestimmt und dann an seine Zieladresse geschrieben, also an die Zieladresse

```

210
211 median_loop:
212     cmp edx, eax
213     ; all blocks traversed
214     je end_median_loop
215
216
217     ; write medians sequentially, beginning at dest
218     mov esi, [ebx]
219     mov [ecx + 4*edx], esi
220
221     ; move to next median (add 4*N)
222     shl edi, 2
223     add ebx, edi
224     shr edi, 2
225     inc edx
226     jmp median_loop
227
228 end_median_loop:
229     pop rsi
230     pop rdx
231     pop rbx
232     pop rax
233     ret
234

```

Tests:

Main-funktion:

Die Tests werden alle mithilfe des test_filter C-Programms ausgeführt. Hierzu wird erst in der main-Funktion das Menü ausgegeben und dann die Methode menu_loop aufgerufen.

```
user@ERA-VM:~/ERA/ASM/tests$ ./test_filter
Main Menu:
(1) Randomly generated signals
(2) Custom input
(3) Plot functions
(4) Exit
>
```

Menü im Terminal

Menü-Funktion:

In der Funktion menu_loop wird die Eingabe, die die Auswahl des Testes bestimmt, eingelesen und dann in einem switch durch verschiedene cases repräsentiert. Durch eine Ausgabe nach den speziellen weiteren Eingaben gefragt. Daraufhin werden die einzelnen Methoden aufgerufen, mit der jeweiligen Eingabe als Parameter.

Random-Test:

Mit der Eingabe 1 gelangt man zu dem diesem Test, bei dem der Filter durch zufällige Zahlen als Signal getestet wird. Dafür wurden die zusätzlichen Eingaben: Anzahl der Test, die Größenordnung der Werte, die Länge des Signal und die Länge der Blocks angefordert.

```
user@ERA-VM:~/ERA/ASM/tests$ ./test_filter
Main Menu:
(1) Randomly generated signals
(2) Custom input
(3) Plot functions
(4) Exit
> 1
Enter <tests> <range> <signal length> <block length>: 5000 200 100 5
5000 test runs completed.
SUCCESS: 5000 - FAIL: 0.
See debug.txt for detailed output.
```

Beispiel eines Random-Testes

Zuerst werden die Eingaben auf Richtigkeit überprüft und dann werden Signale erzeugt, die auch negative Zahlen beinhalten können (bei range 200, sind es also Zahlen zwischen -100 und 100). Mit jedem Signal wird ein Testrun durchgeführt, der das Ergebnis des Assemblerprogramms mit dem Ergebnis eines Medianfilters in C-Code vergleicht. Bei gleichem Ergebnis wird der Success counter erhöht, sonst der Fail counter. Schlussendlich werden die Zähler ausgegeben und das Programm beendet.

Custom-Test:

Beim Custom-Test wird das Signal aus bereits vorhandenen Eingaben gefiltert. Hierzu wurde die Blocklänge eingegeben. Nachdem die Eingabe und die Datei „custom_signal.dat“ überprüft wurden, werden auf die Signale, die extern gespeichert wurden, wieder Testruns mit Success und Fail countern ausgeführt. Das Ergebnis wird wieder ausgegeben.

```
user@ERA-VM:~/ERA/ASM/tests$ ./test_filter
Main Menu:
(1) Randomly generated signals
(2) Custom input
(3) Plot functions
(4) Exit
> 2
Enter block length: 5
12 test runs completed.
SUCCESS: 12 - FAIL: 0.
See debug.txt for detailed output.
```

Beispiel eines Custom-Tests

Funktionen-Test:

```
user@ERA-VM:~/ERA/ASM/tests$ ./test_filter
Main Menu:
(1) Randomly generated signals
(2) Custom input
(3) Plot functions
(4) Exit
> 3
Select function:
(1) sin(x)
(2) cos(x)
(3) exp(x)
(4) log(x)
(5) sqrt(x)
(6) Custom
> 1
```

Im Funktionen-Test wurde durch den Input die zu behandelnde Funktion gegeben. Hierbei ist die Blocklänge immer 5 und die Länge des Signals immer 100. Nach dem Überprüfen der Eingaben wird die entsprechende Datei aus dem Ordner

←Auswahlmenü der Funktionen

plots ausgewählt und dann ein verrauschtes Signal erzeugt und mit dem Assembler Programm gefiltert.

Bekannte Probleme:

Doch die Anwendung des Filters auf das verrauschte Signal hat auch einige Probleme. So ist die weitere Verwendung des neuen Signales in einem Programm schwierig, da sich die Länge des Signals um einiges verkürzt hat.

Zudem ist die Wahrscheinlichkeit nicht sehr groß, dass man das Originalsignal genau wiederherstellen konnte. Muss man also Informationen aus dem Signal lesen, sind diese immer noch verfälscht.

Potentielle Weiterentwicklungen:

Um das Problem der verkürzten Signallänge zu beheben kann man verschiedene Methoden anwenden. Zum einen könnte man einfach jeden Median reproduzieren, also sooft wie ein Block lang ist. Dadurch erreicht man wieder die Originallänge (bis auf ein paar Werte am Ende des Signals, die beim Erstellen des gefilterten Signals ignoriert wurden).

Eine andere Möglichkeit ist, dass man nicht nur den Median vervielfacht, sondern Werte wählt, die zwischen den zwei Median liegen, falls diese noch nicht benachbarte Zahlen sind. So kann man auch ein durchgängiges Signal erschaffen, das keine Sprünge aufweist.