Assembler Assignment

ES153 Engineering Software

MR TJ KENNAUGH
0013679

School of Engineering, University of Warwick

01/03/01

Introduction

The aim of this assignment is to optimise a bubble sort algorithm using assembly code for use on an 8051 microprocessor. Wedit32 is used to simulate the program, allowing easy development of a solution. The aim is to reduce the program size and execution time by optimising the code. It is possible to optimise the code to approximately half its original size and a third of its execution time.

Bubble Sort Algorithm

The bubble sort algorithm is used to arrange an array of bytes into ascending order the algorithm in C is given below:

```
#define N 5
void main(void) {
idata unsigned char a[N];
data unsigned char i,j,k;
for (i=0;i<N-1;i++) {
  for(j=0;j<N-1;j++)
    if(a[j]>a[j+1]) {
      k=a[j]; a[j]=a[j+1]; a[j+1]=k;
      }
  }
}
```

The algorithm sorts a 5-byte array of data into ascending order. E.g. given:

```
05 04 03 02 01
```

The algorithm would sort this to:

```
01 02 03 04 05
```

It accomplishes this by comparing each byte with the one next to it and swapping them around if the right byte is smaller than the left. It performs this 4 times to ensure that the whole array is sorted. The method for the above data is shown below:

```
1.  05 04 03 02 01
2.  04 05 03 02 01
3.  04 03 05 02 01
4.  04 03 02 05 01
5.  04 03 02 01 05
6.  03 04 02 01 05
7.  03 02 04 01 05
8.  03 02 01 04 05
9.  03 02 01 04 05
10. 02 03 01 04 05
11. 02 01 03 04 05
12. 01 02 03 04 05
```

Note: duplicate results have been removed.

The program will perform 4 passes 4 times i.e. 16 potential swap operations. This is inefficient since during the first pass the largest byte will always be moved to the end, removing the need to check the last byte. A cumulative effect is produced meaning that on the last run only the 1st and 2nd byte need to be checked.

Implementation in Assembler

Wedit32 is used throughout this assignment. This is a powerful simulator for the 8051 microprocessor, which is the processor the code is to be written for.

The compiler will convert the C program to assembler as shown on the assignment sheet (Appendix B). A modified version with registers mapped is shown below:

```
;bubble.a51
$include (reg51.INC)
; R2 = i R3 = j R4 = k
CSEG at 0100h

main:
  MOV R2,#00
main1:
  MOV R3,#00
main2:
  MOV A,R3
  ADD A,#65
  MOV R0,A
  MOV A,R3
  ADD A,#64
  MOV R1,A
  MOV B,@R0
  MOV A,@R1
  CJNE A,B,main3
  SETB C
main3:
  JC main4
  MOV A,@R1
  MOV R4,A
  MOV A,R3
  ADD A,#64
  MOV A,@R0
  MOV @R1,A
  MOV A,R4
  MOV @R0,A
main4:
  INC R3
  MOV A,R3
  CJNE A,#04,main2
  INC R2
  MOV A,R2
  CJNE A,#04,main1
  JMP main
  END
```

The listing from the assembler can be used to find the number of bytes used in this program. For the program above this is found to be 42 bytes.
The time taken for this program to sort 5 descending bytes (worst case) inputted into data memory locations 40_H to 44_H can be calculated by running the program with a break point at the JMP main instruction. The time is reset using Ctrl + T before the simulation is started. Running the program with initial input (`05 04 03 02 01`) the time to sort is measured as 373μs.
The translation for the C code works but is inefficient. The aim of the assignment is to optimise this code for size and speed.
Printouts produced during the development can be found in appendix A.

Initially it may be seen that the use of the ADD command is inefficient and instead of adding 64 (corresponding to the data location 40_H) to the register controlling the pointers each time the register may be initially set to 64. Also the last CJNE command is inefficient, the register can be made to count down and the DJNZ instruction can be used. See Appendix A. Page II

The section of code can after main2: can be further improved by removing the need to decrement and increment again. See Appendix A. Page III

As a more efficient method of implementing the CJNE A,B,main3 SETB C main3 JC main4 the SUBB function may be used as this will set the carry bit if A<B the same as the CJNE and the SETB C and pointer relative jump are not required.

Hence:

```
CJNE A,B,main3
SETB C
main3:
JC main4
```

Can be replaced with:

```
SUBB A,B
JC main4
```

Reducing the execution time to 333μs. See Appendix A. Page IV.

It may be seen that there is an inefficiency in the original algorithm. At the first run of the inner for loop the largest byte will always be at the end (right or the array (i.e. in the correct position) thus requiring the program to check if this byte (5) is smaller than byte 6 is unnecessary. The limit of this CJNE function may be decrement after each run thus run 2 will only check byte to byte 4 run 2 to byte 3 etc. R5 is set and used to decrement this limit, reducing the execution time to 268μs. See Appendix A. Page V.

It may be seen that the 8051's XCH command could be used to reduce the number of commands required in the swapping of the memory addresses. Therefore also noting that a register is being set to the value it already contains the 8 lines of code between JC and main4: can be reduced to the 3 lines:

```
MOV A,@R1
XCH A,@R0
MOV @R1,A
```

Reducing the execution time further to 208μs. See Appendix A. Page VI

It is unnecessary to move @R0 to B before using the SUBB command and MOV A,R3 can be moved to before main2: (since it is set when the program jumps here). The final execution time is 182μs and the final program size is 29 bytes. See Appendix A. Page VI.

The program has therefore been reduced from 42 to 29 bytes and the execution time has been more than halved from 373μs to 182μs. An reduction in the time of more than half is often extremely significant when dealing with embedded systems.

The final program is shown below (Also in Appendix A. Page VII)

```
;bubble.a51
$include (reg51.INC)
; R2 = i R3 = j R4 = k
CSEG at 0100h

main:
  MOV R5,#68
  MOV R2,#04
main1:
  MOV R3,#64
  MOV A,R3
main2:
  MOV R1,A
  MOV R0,A
  INC R0
```

```
  MOV A,@R1
  SUBB A,@R0
  JC main4
  MOV A,@R1
  XCH A,@R0
  MOV @R1,A
main4:
  INC R3
  MOV A,R3
  MOV B,R5
  CJNE A,B,main2
  DEC R5
  DJNZ R2,main1
  JMP main
  END
```

Appendices

Appendix A: Development Stages

I. Original Code
II. Optimisation of pointer removing ADD function and changing CJNE to DJNZ
III. Optimisation removing INC/DEC (349µs execution time)
IV. Optimisation of CJNE for greater than, replacing with SUBB (333µs execution time)
V. Reduction of program execution time by removing need to calculate to end each time (268µs execution time)
VI. Optimisation of memory swapping using XCH (208µs execution time 32bytes)
VII. Final optimisation removing unnecessary instructions (182µs execution time 29bytes) (Final Program)

Appendix B Assignment Sheet