TEC | Tecnológico de Costa Rica

Computer School

Prof: Dr. Francisco J. Torres Rojas

Project 0

Old-fashioned multiplication

Technical Report

José Ceciliano Granados 2016087245
Luis Alejandro Garita 2016094679

Version 1.0
February 21, 2018

Index

1 Introduction

The search of this report is to make an analysis on the data gathered in tests of how long does it take for a computer to do different multiplication versions by a certain number of times, always with the same random numbers. There will be three different multiplications versions used for this report:

1. Empty version, the function only return zero.
2. Standard version, the function uses the language usual multiplication method (i.e. A * B).
3. Ancient version, this algorithm uses an old algorithm explained in the project specification.

To make a more interesting analysis, the three methods showed before will be made in two different languages. First in a high level language as C, and then in a low level language as assembler, in this case NASM.

2 Program Structure

2.1 Basics

As explained before, the idea is to repeat each method of multiplication a number N times and get the time it took for each method to finish the process. Before starting is necessary to know and understand the global variables that will be used in the program. Now the first step is to get the random generated numbers that will be used over and over for every multiplication method, and save them in a variable. The second step is to print for the user the tables that will be used for the program first in the form A*B and later in the form B*A. Next step is do each multiplication step N times and show the results and times in screen for A*B and vice versa. And last but not least wait for the user to press a key to see the next method. For an example run of the program, please check the annexes 1 through 7

2.2 Variables

The program uses the following global variables:

- clock_t tInicio: Used to take the time when the function starts.
- clock_t tFin: Used to take the time when the function finishes.
- clock_t tDecorrido: Will have the time between tInicio and tFin.
- int num_aleatorio[7]: Is a list with the 8 random generated numbers.
- int mat[4][4]: Is a matrix of 4 by 4 that saves the result of the operation.
- int n=10000000: Is the n with the amount of times the user wants to repeat each operation.

```
///Variables
clock_t tInicio, tFin, tDecorrido;
int num_aleatorio[7], mat[4][4], n=10000000;
```

2.3 Multiplication Functions

The functions for multiplication made in C where the following in the order empty, standard and ancient:

```c
///Funciones para multiplicar
int f_vacia(int A, int B){
    return 0;
}

int f_estandar(int A,int B){
    return A*B;
}

int f_antigua(int A,int B){
    int P = 0;
    while(B != 0){
        if((B%2) != 0){
            P += A;
        }
        A += A;
        B = B >> 1;
    }
    return P;
}

extern int f_vaciaE(int, int);
extern int f_estandarE(int, int);
extern int f_antiguaE(int, int);
```

Notice that the last three are just defining that those functions will exist, the extern implies those three are actually programmed in assembler as follows:

```
global f_vaciaE
global f_estandarE
global f_antiguaE

section .text

f_vaciaE:
    xor eax,eax     ; deja en cero eax
    ret

f_estandarE:
    mov eax, edi    ; edi = A, esi = B
    mul esi         ; multiplica A por B
    ret

f_antiguaE:
    mov edx, edi    ; edx = A
    mov ebx, esi    ; ebx = B
    xor eax, eax    ; eax = P

    BNoEsCero:      ; mientras B no sea cero

    cmp ebx, 0
    je  BEsCero     ; si B es 0 sale del while

        test ebx, 1 ; Test hace un and con edx con 1
        jnz esImpar ; Si es impar va a quedar prendido el bit 1
        jmp esPar

        esImpar:
            add eax, edx    ; P = P + A
        esPar:
            add edx, edx    ; A = A + A
            shr ebx, 1      ; B = B/2

        jmp BNoEsCero

    BEsCero:
    ret
```

2.4 Writing Functions

To make sure all the times and random generated numbers were properly saved in a text file, there are a few functions dedicated only to the matter. This helped a lot in the process of analysing the data. They were the following:

```
///Funciones para guardar datos en un archivo
void guardarTiempo(int ns){
    FILE *fp;
    fp = fopen("/home/alejandro/Documentos/2018-ISemestre/Analisis_de_Algoritmos/T1AA/Datos/Tiempos.txt", "a");
    fprintf(fp, "%d\t", ns);
    fclose(fp);
}

void nuevaLinea(){
    FILE *fp;
    fp = fopen("/home/alejandro/Documentos/2018-ISemestre/Analisis_de_Algoritmos/T1AA/Datos/Tiempos.txt", "a");
    fprintf(fp, "\n");
    fclose(fp);
}

void guardarAleatorios(){
    FILE *fp;
    fp = fopen("/home/alejandro/Documentos/2018-ISemestre/Analisis_de_Algoritmos/T1AA/Datos/Aleatorios.txt", "a");
    for( int i=0; i<8; i++){
        fprintf(fp, "%d\t", num_aleatorio[i]);
    }
    fprintf(fp, "\n");
    fclose(fp);
}
```

2.5 Interface functions

One of the most important parts of the program is to show the data obtained to the user. So that's why there are a lot function which purpose is to show the interface in a nice and clear way. The most important interface functions will be explained.

- escribir_base: It writes the two matrices used, not with the results but with the operands used, in both A*B and B*A.

```
void escribir_base(void){
    printf("\t PRESENTACION DE LA TABLA BASE:\n\n");
    for (int i = 0; i < 4; i++) {
        for (int j = 0; j < 4; j++)
            printf("%4d X %-4d\t", num_aleatorio[i],num_aleatorio[j+4]);
            putchar('\n');
    }
    printf("\n-------------------------------------------------------------\n");
    printf("\t PRESENTACION DE LA TABLA ALTERNA:\n\n");
    for (int i = 0; i < 4; i++) {
        for (int j = 0; j < 4; j++)
            printf("%4d X %-4d\t", num_aleatorio[i+4],num_aleatorio[j]);
            putchar('\n');
    }
    printf("\n-------------------------------------------------------------\n");
}
```

- escribir: In this case it presents both matrices results, plus the time it took for it to finish.

```c
void escribir(long unsigned int tiempo, int tipo){
    if(tipo == 1){
        for (int i = 0; i < 4; i++) {
            for (int j = 0; j < 4; j++)
                printf("%8d\t", mat[i][j]);
                putchar('\n');
        }
        printf("\n\nTiempo de ejecución para la tabla base: %lu ms \n\n ", tiempo);
        printf("-------------------------------------------------------------\n");
        printf("\t TABLA ALTERNA\n\n");
    }
    else{
        for (int i = 0; i < 4; i++) {
            for (int j = 0; j < 4; j++)
                printf("%8d\t", mat[j][i]);
                putchar('\n');
        }
        printf("\n\nTiempo de ejecución para la tabla alterna: %lu ms \n \n ",tiempo);
        printf("-------------------------------------------------------------\n");
    }
}
```

- escribir_num_aleatorio: This one just prints the 8 random numbers used.

```c
void escribir_num_aleatorio(void){
    printf("\n\t NUMEROS ALEATORIOS USADOS:\n\n");
    for (int i = 0; i < 8; i++) {
        printf("\t |%d -> %4d|\n", i,num_aleatorio[i]);
    }
    printf("-------------------------------------------------------------\n");
}
```

- siguiente: This one is really important, it waits for the user to press enter so the program can continue.

```c
void siguiente(char* version){
    printf("Precione enter para %s\n", version);
    getchar();
    system("clear");
}
```

- versionVaciaC: For this functions where NNNN is the versions name and L the language. This are the most important because they are ones that actually call the multiplication functions and store the result in the matrix. Only one will be shown because all of them are very similar.

```c
void versionVaciaC(){
    printf("\n\n  \t\t VERSIÓN VACIA\n");
    printf("-------------------------------------------------------------\n");
    printf(" \n \t\t PROGRAMADO EN C\n\n");
    printf("-------------------------------------------------------------\n");
    printf("\t TABLA BASE \n\n");

    /************* TABLA BASE ****************/
    tInicio = clock();
    for (int r = 0; r < n; r++) {
        for (int i = 0; i < 4; i++) {
            for (int j = 0; j < 4; j++){
                mat[i][j] = f_vacia(num_aleatorio[i],num_aleatorio[j+4]);
            }
        }
    }
    tFin = clock();
    terminarBase(tInicio, tFin, tDecorrido);
    /************* FORMA B X A ****************/
    tInicio = clock();
    for (int r = 0; r < n; r++) {
        for (int i = 0; i < 4; i++) {
            for (int j = 0; j < 4; j++){
                mat[i][j] = f_vacia(num_aleatorio[j+4],num_aleatorio[i]);
            }
        }
    }
    tFin = clock();
    terminarAlterna(tInicio, tFin, tDecorrido);
}
```

2.6 Others

There were other important functions used.

- terminar: this functions finished the run of one of the methods and is where the time is calculated.

```c
void terminarBase(){
    tDecorrido = ((tFin - tInicio) / (CLOCKS_PER_SEC / 1000));
    guardarTiempo(tDecorrido);
    escribir(tDecorrido, 1);
}

void terminarAlterna(){
    tDecorrido = ((tFin - tInicio) / (CLOCKS_PER_SEC / 1000));
    guardarTiempo(tDecorrido);
    escribir(tDecorrido, 2);
    escribir_num_aleatorio();
}
```

- main: This is the principal function, that starts the program and call the rest of the functions

```c
int main(void){
    //for(int repet=0; repet<75; repet++){
    srand(time(NULL));
    for( int i= 0; i<8; i++){
        num_aleatorio[i] = rand() % 9999;
    }

    /*printf("Digíte el numero N: ");
    scanf("%d",&n);
    getchar();
    system("clear");*/

    system("clear");
    escribir_base();
    siguiente("ver la solución vacía");

    escribir_base();
    versionVaciaC();
    siguiente("ver solucion en ensamblador");
    escribir_base();
    versionVaciaASM();
    siguiente("continuar con la solucion en la versión estándar");

    escribir_base();
    versionEstandarC();
    siguiente("ver solucion en ensamblador");
    escribir_base();
    versionEstandarASM();
    siguiente("continuar con la solucion en la versión antigua");

    escribir_base();
    versionAntiguaC();
    siguiente("ver solucion en ensamblador");
    escribir_base();
    versionAntiguaASM();
    siguiente("finalizar");

    nuevaLinea();
    guardarAleatorios();
    //}
    return 0;
}
```

3 Data Analysis

Now is presented three table with interesting data obtained for the program after the use of a N number of one million and 100 tests. There's data such as the highest time, the lowest time, the average time, the difference between the highest and lowest, the most repeated time and how many times it repeated. Also there will be a deeper analysis from the data obtained for each version of the program. For the full table, check in annexes for table 8.1, 8.2 and 8.3.

3.1 Considerations

For this test, the specifications and operating system of the computer in which the tests were carried out are taken into account, which are:

- OS: Ubuntu 16.04.03
- RAM Memory: 12056MB
- Processor: Intel Core i7-6500U CPU @ 2.50GHz

3.1.1 The N number choice

Before starting all the tests, there was the idea of trying the program with 500 thousand, 1 million or 10 million times to see which one could give significant data without during an eternity. The discarded ones were 500 thousand because the information it gave wasn't significant enough and 10 million because it was too slow, and for the amount of test that were planned, it wasn't good enough. So at the end the test with a N of 1 million, were really fast and give some interesting information that could be work on.

3.1.2 The amount of tests

To have a significant amount of data there was needed to run the program a lot of times, and after a little deliberation between the members of the project it was defined that 100 tests would give enough information for this analysis.

3.2 Empty Version

Table 1: Empty Version Data

	Empty			
	C		ASM	
	AXB	BXA	AXB	BXA
Higher	98	67	82	52
Average	83.50	65.37	68.54	49.11
Lower	64	62	53	48
Lower-Higher Difference	34	5	29	4
Most repeated	89	66	77	49
Times repeated	16	40	17	67

It might be expected that this version will be basically immediate, however, it averages a time of 83.5ms for A*B and 65.3ms for the operations written in Language C, this operation is the result of what the program takes to communicate with the operating system and create a space in memory that returns a 0, what we can observe is the response time of the memory and as though it does not perform any significant function it shows at the time that we are limited by access to memory. At the same time we have the comparison with the assembly language which averages 68.5ms for A*B and 49.1 for B*A, which shows the limitation of the time of access to memory, but still shows a better time in a low level language in comparison to the high level counterpart.

3.2.1 What is the empty version of the algorithms for?

It doesn't really serves a purpose, because as it has been shown it always return 0 so it never does anything. But as the testing has shown it does takes time to do, and that's why the empty function is good it show how long does it takes for the computer to access memory and return a value, in this case 0.

3.3 Standard Version

Table 2: Standard Version Data

	Standard			
	C		ASM	
	AXB	BXA	AXB	BXA
Higher	107	75	85	53
Average	90.16	71.86	73.26	52.45
Lower	74	71	56	52
Lower-Higher Difference	33	4	29	1
Most repeated	78	72	82	52
Times repeated	15	73	15	55

We observe an average a little bit higher to the one in the empty function (between 5 and 10 ms slower). For the standard multiplication operation in C language, we have an average of 90.1ms in A*B and 71.8ms for B*A which shows the access time to memory plus the calculation of Multiplication processing, using the algorithm imposed by the compiler. Now if we focus on the assembly counterpart we observe a really good result which shows an average of 73.2ms for A*B and 52.45 for B*A, this can be deduced to the memory access time and that working in the registers themselves is a lot faster in assembler which reduces use other system resources to perform the calculation, making it faster.

3.4 Ancient Version

Table 3: Ancient Version Data

	Ancient			
	C		ASM	
	AXB	BXA	AXB	BXA
Higher	691	665	478	482
Average	639.27	608.59	422.88	408.68
Lower	549	528	366	340
Lower-Higher Difference	142	137	112	142
Most repeated	622	623	416	428
Times repeated	3	8	5	5

This function shown by the J.Nyberg paper shows that the average C language time is 639.2ms for A*B and 608.5ms for B*A, far exceeding the result of the multiplication algorithm of the system, this because these instructions require the interpretation and storage of more variables in memory at the same time that requires the use of processing and access to memory, which undoubtedly makes it an inefficient algorithm for multiplication.
For assembly language we have an average of 422.88ms for A*B and 408.6 in B*A which leads us to think that said language, again by being low level and working directly in the registers, requires a greater amount of resources than the standard multiplication method, but makes it faster than the on programmed in C.

3.4.1 Is the ancient algorithm good? How much?

After all the tests done, it's safe to say that the ancient algorithm is not good for computers. Actually it's way worse than the standard version, in the case of C is close to 10 times worse. This doesn't mean that this version is completely bad, is just that in the specific case for computers is not good enough, for example, in some cases it might be faster than the standard version for some humans without a calculator.

3.5 Other Information

3.5.1 Is there gain in assembler versions?

Yeah, there is and is actually quite significant This is a list with how much percent faster is the assembler version against the C version:

- Empty Version (A*B): 17.91% faster
- Empty Version (B*A): 24.87% faster
- Standard Version (A*B): 18.74% faster
- Standard Version (B*A): 27.01% faster
- Ancient Version (A*B): 33.85% faster
- Ancient Version (B*A): 32.85% faster

This shows perfectly that the assembler version is way faster than his counterpart versions in C

3.5.2 Does the order of the arguments affect? There are some rule?

Unexpectedly enough B*A is a lot faster than A*B in all six versions, either NASM or C.

- Empty Version (C): 21.71% faster
- Empty Version (NASM): 28.35% faster
- Standard Version (C): 20.30% faster
- Standard Version (NASM): 28.41% faster
- Ancient Version (C): 4.80% faster
- Ancient Version (NASM): 3.36% faster

Even in the ancient version is still faster, not by a great margin though. But in the cases of the empty and the standard version, the difference is significant. It's even more in the NASM versions of the previously said ones.

3.5.3 Interesting Data

From all the data there are some interesting occurrences, that will be shown now:

- The Standard, NASM, in both versions is so fast that, is even faster than Empty, C, in both versions.

- The most consistent version: This achievement goes to the Standard, C, B*A version. In this version the most repeated time is 72ms and is repeated a whopping 73 times in 100 tests.
- The least variable version: The Standard, NASM, B*A version. In this version, in all the 100 tests, there were only two different times 52ms and 53ms, no more nor less.
- The most variable version: The Ancient, C, A*B version, which is also the slowest, is the one with most variables. It's also the least consistent version, the most repeated time was 622ms with only 3 times from the 100 tests.

4 Annexes

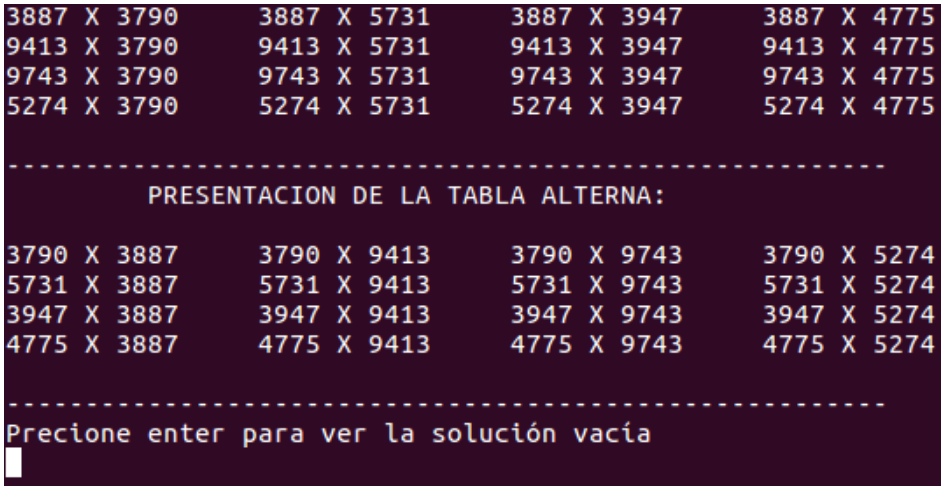

Figure 1: Presentation of the numbers generated that will be multiplied

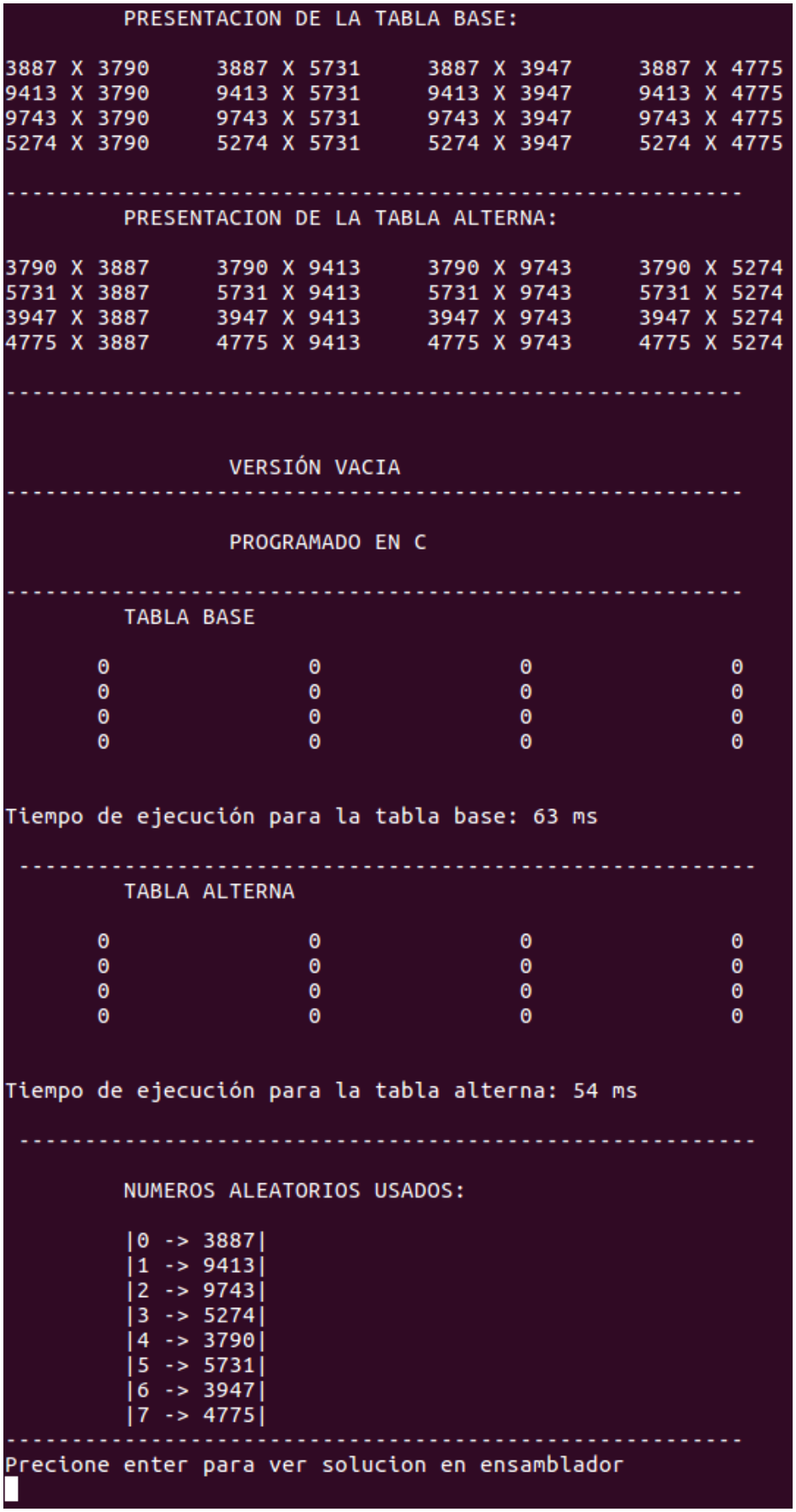

Figure 2: Empty function programmed in C language

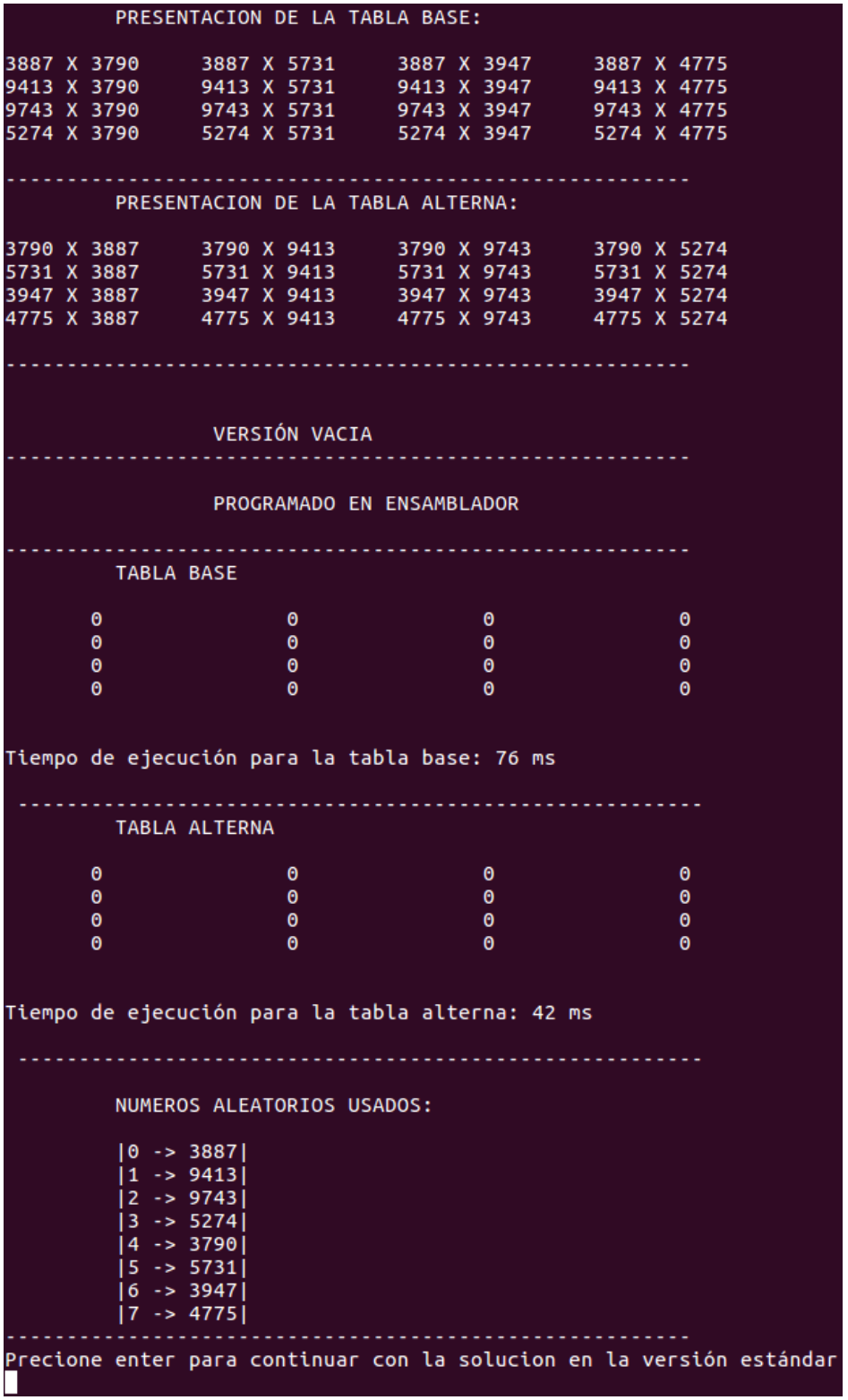

Figure 3: Empty function programmed in Assembler Language

Figure 4: Standard function programmed in Assembler Language

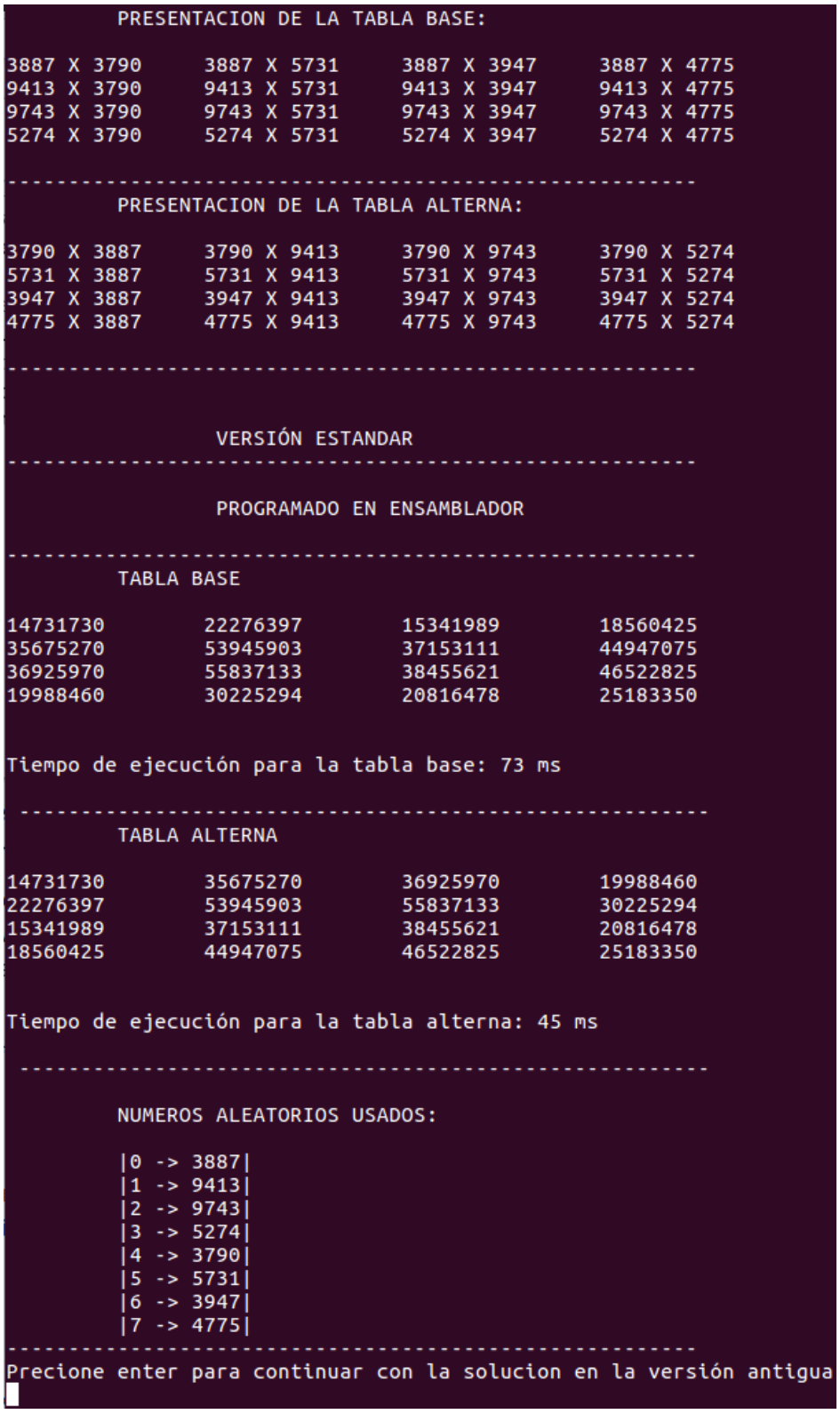

Figure 5: Standard function programmed in C language

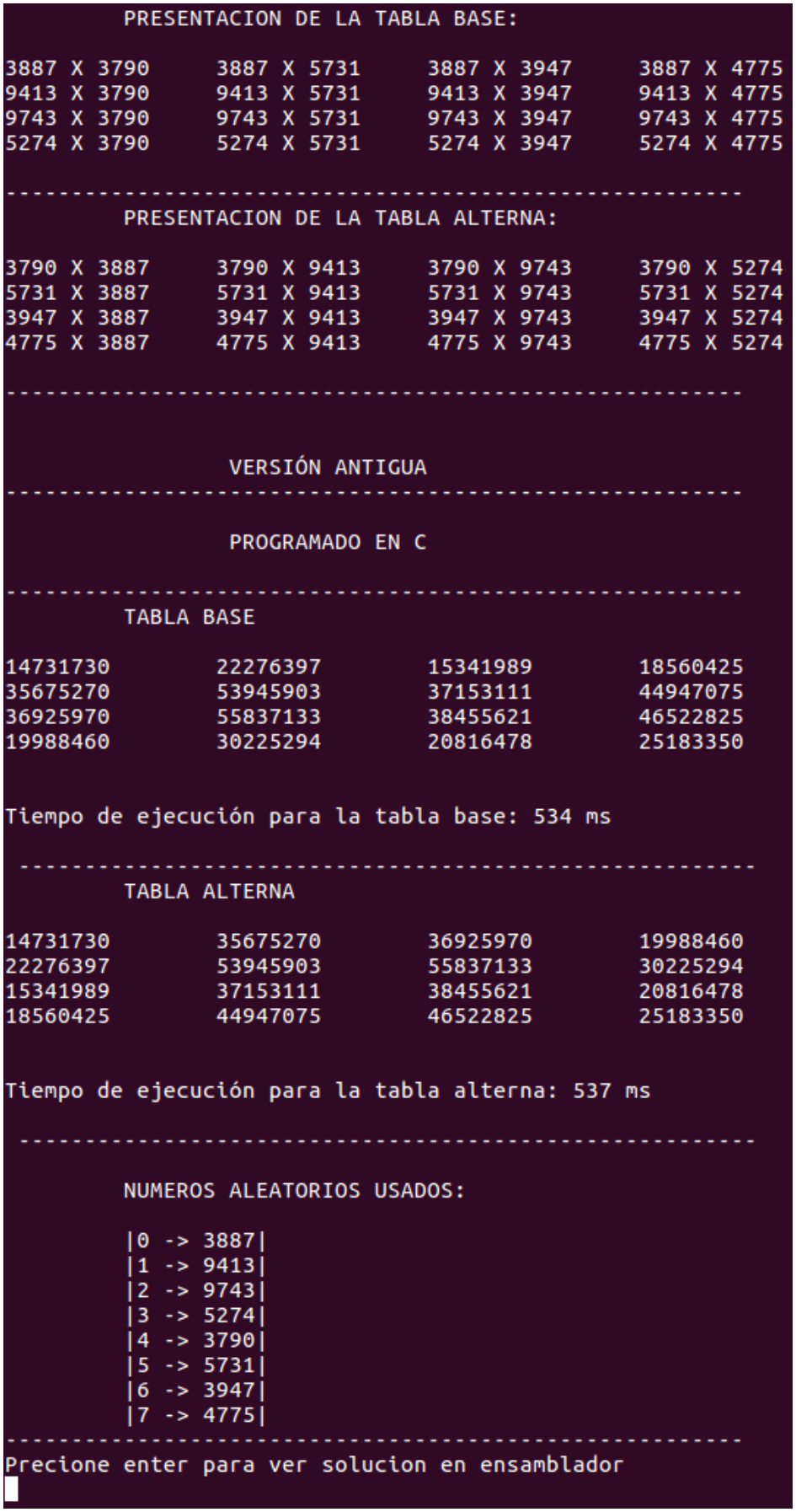

Figure 6: Ancient function programmed in C Language

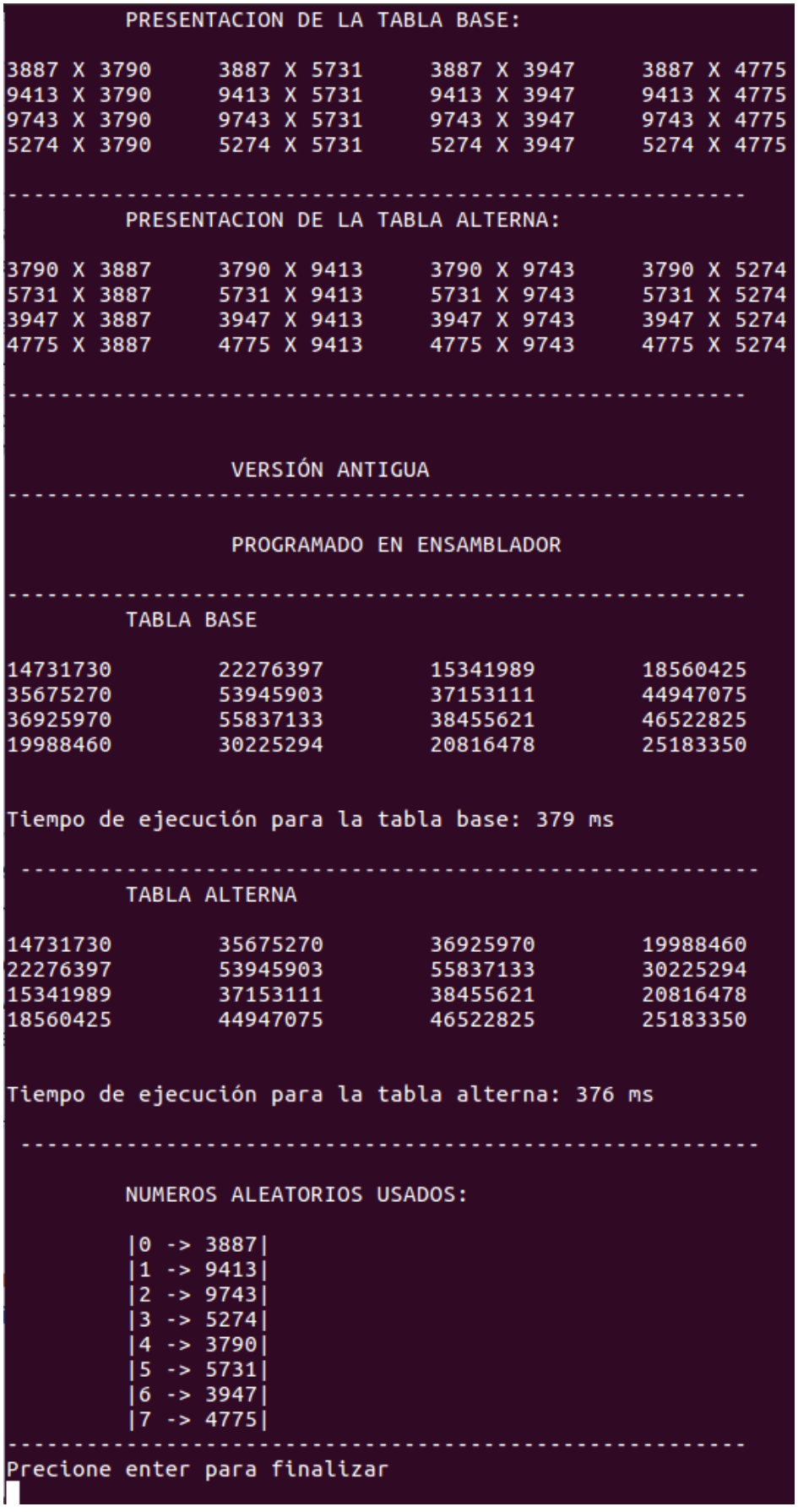

Figure 7: Ancient function programmed in Assembler Language

Data obtained 8.1

#	Time per version											
	Empty				Standard				Ancient			
	C		ASM		C		ASM		C		ASM	
	AXB	BXA	AXB	BXA	AXB	BXA	AXB	BXA	AXB	BXA	AXB	BXA
1	93	63	63	50	101	72	81	52	666	592	421	421
2	70	62	76	50	81	74	83	52	627	624	436	418
3	87	63	60	49	99	72	81	53	622	567	402	365
4	87	63	80	49	97	72	84	53	663	623	422	428
5	90	63	76	49	103	73	82	52	659	621	402	424
6	86	64	74	49	77	72	79	53	610	598	413	388
7	91	63	77	49	98	72	77	52	678	528	431	341
8	89	64	73	50	98	72	81	53	620	582	409	398
9	92	63	77	49	102	72	83	52	593	561	416	370
10	89	65	76	49	98	71	79	52	668	594	419	393
11	88	63	78	49	102	72	81	53	658	628	433	406
12	91	64	78	49	99	72	83	53	689	628	446	441
13	69	65	80	49	80	73	82	53	650	632	425	442
14	67	64	77	49	103	72	82	53	639	620	445	399
15	85	62	75	50	97	72	79	53	633	579	406	393
16	89	65	77	49	101	72	80	53	649	582	409	358
17	87	67	57	49	77	72	80	52	667	620	444	396
18	86	66	73	49	74	73	79	53	585	636	406	456
19	86	66	78	50	100	72	82	52	667	615	434	415
20	89	65	76	49	101	72	63	53	614	614	422	401
21	86	65	78	50	80	72	81	53	645	575	435	377
22	93	66	74	49	101	72	78	53	582	616	409	433
23	92	66	78	50	97	72	82	52	672	595	435	369
24	65	66	53	49	99	72	80	53	645	637	448	468
25	70	66	79	50	99	72	82	52	669	623	470	395
26	90	66	78	49	102	72	79	53	683	591	420	396
27	86	65	59	50	98	72	58	52	581	601	416	423
28	64	65	77	52	80	71	59	53	633	625	414	419
29	90	67	57	49	99	72	82	53	589	582	377	373
30	88	65	57	49	82	72	60	52	615	591	392	405
31	71	65	60	49	96	71	65	52	669	623	423	417
32	89	66	55	48	99	72	57	53	663	640	425	382
33	86	65	74	49	77	72	60	53	617	619	433	395
34	65	67	77	49	102	71	56	53	663	600	420	398
35	69	66	64	49	100	71	61	52	600	603	381	404
36	86	65	77	50	98	71	79	53	640	575	413	363
37	91	65	77	49	78	71	61	53	601	641	404	461
38	69	66	75	49	98	72	75	53	636	636	433	403
39	92	65	77	49	100	72	80	53	628	623	416	409

Data obtained 8.2

#	Time per version											
	Empty				Standard				Ancient			
	C		ASM		C		ASM		C		ASM	
	AXB	BXA	AXB	BXA	AXB	BXA	AXB	BXA	AXB	BXA	AXB	BXA
40	87	67	78	50	78	72	58	52	627	634	407	454
41	89	65	73	48	100	72	58	53	588	625	396	415
42	91	66	78	48	102	72	76	52	680	609	448	410
43	89	67	54	48	78	72	82	52	654	592	445	373
44	87	65	57	49	100	72	81	52	676	612	428	406
45	87	64	75	49	75	71	85	53	642	600	425	422
46	87	66	64	49	96	72	82	53	631	636	441	452
47	90	66	57	49	103	72	60	52	622	616	420	423
48	69	66	54	49	99	72	84	52	643	608	403	392
49	68	66	73	49	97	72	65	53	654	631	430	405
50	71	65	55	49	98	71	58	52	661	601	431	403
51	90	65	76	49	95	72	75	53	616	630	450	440
52	84	65	77	49	83	71	78	52	629	641	421	431
53	80	66	61	48	99	72	63	52	613	531	415	340
54	87	67	74	49	78	72	58	52	615	604	411	411
55	89	67	60	48	91	75	81	52	618	577	376	400
56	87	66	56	49	82	72	82	52	679	597	476	374
57	89	65	55	49	82	72	82	52	672	626	422	405
58	90	67	71	49	77	71	80	52	623	546	416	355
59	87	65	79	49	98	72	58	52	680	599	429	364
60	89	67	58	49	78	72	82	52	652	612	409	426
61	88	64	55	49	78	72	63	52	691	560	424	357
62	86	66	55	49	79	72	78	53	620	623	431	422
63	87	66	77	49	97	71	58	53	677	623	421	405
64	89	66	56	49	80	71	80	52	661	631	433	428
65	92	66	77	50	85	72	80	53	653	566	430	375
66	89	66	62	49	81	71	77	52	661	622	455	437
67	88	66	73	50	101	71	83	53	688	651	459	461
68	70	66	77	50	98	72	84	53	656	565	407	401
69	88	65	61	49	102	72	60	52	595	578	399	392
70	92	65	59	49	101	71	60	52	637	559	427	366
71	88	65	76	49	96	72	82	52	639	618	417	428
72	89	66	76	48	80	72	60	53	637	596	386	411
73	89	66	56	49	79	72	84	52	608	576	427	361
74	98	65	78	49	102	72	78	52	657	588	452	377
75	68	66	77	50	82	72	63	52	604	646	398	452
76	90	65	77	48	78	71	62	52	622	620	416	419
77	72	64	76	49	97	72	84	53	683	600	478	392
78	90	65	58	49	80	72	59	52	628	608	428	401
79	88	64	75	49	80	72	79	52	602	623	413	444

Data obtained 8.3

#	Time per version											
	Empty				Standard				Ancient			
	C		ASM		C		ASM		C		ASM	
	AXB	BXA	AXB	BXA	AXB	BXA	AXB	BXA	AXB	BXA	AXB	BXA
80	68	65	82	48	83	71	61	52	648	613	420	419
81	89	66	55	49	77	72	78	52	595	619	366	414
82	67	66	56	48	92	72	61	52	649	634	419	401
83	73	66	76	49	101	72	61	52	658	665	448	482
84	88	66	63	50	85	72	79	52	657	608	415	417
85	68	66	56	49	78	71	81	52	631	660	413	386
86	70	65	76	49	85	72	60	53	657	627	460	406
87	76	67	57	49	82	72	63	52	665	553	430	362
88	87	66	57	50	78	72	63	53	662	617	447	441
89	71	67	57	50	76	71	82	53	643	636	425	412
90	87	66	79	49	78	72	79	52	593	614	404	428
91	69	66	55	49	78	72	59	52	676	609	414	392
92	89	66	62	49	99	72	65	52	627	623	421	426
93	88	66	56	49	85	72	63	53	604	641	421	462
94	68	64	77	50	82	71	80	53	549	603	376	443
95	91	65	56	49	107	72	83	52	635	621	423	414
96	88	66	57	49	78	72	84	53	599	631	422	428
97	87	67	55	49	77	71	61	52	612	607	408	431
98	72	66	77	48	78	72	81	53	664	653	463	479
99	88	66	79	50	101	72	82	52	639	613	435	422
100	85	67	73	48	78	72	62	53	682	592	453	430