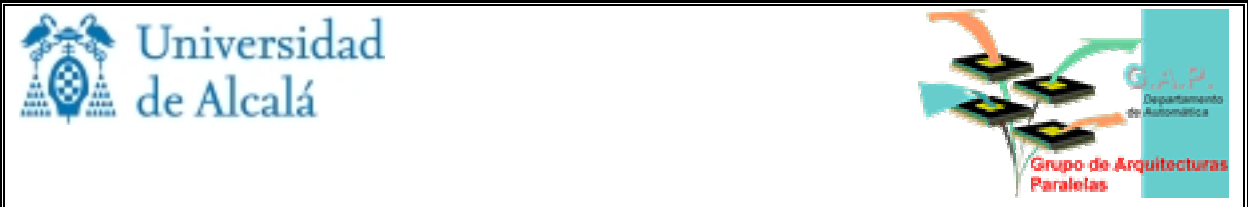


Proposal of test-bench for the x86 instruction set (16 bits subset)

Technical Report TR-UAH-AUT-GAP-2005-21-en

Rafael Rico

Department of Computer Engineering, Universidad de Alcalá, Spain

November 2005

Versión en español:

Propuesta de banco de pruebas para el repertorio de instrucciones x86 (subconjunto de 16 bits)

Informe técnico TR-UAH-AUT-GAP-2005-21-es

Rafael Rico

Departamento de Automática, Universidad de Alcalá, España

Abstract:

With the purpose of evaluating the instruction set architecture impact on the superscalar processing, applying a mathematical method derived from the graph theory, a set of programs is proposed as test-bench.

The application corresponds to the integer processing. The instruction set selected is the x86 one due to its peculiar characteristics with respect to instruction level parallelism.

The methodology of obtaining of execution traces is presented and the work load of each one of the selected programs.

Finally, a characterization of each program is made on the basis of his functionality and to the counts of operations and operands.

Index words: Evaluation of computer architectures, instruction level parallelism, instruction set architecture.

Resumen:

Con el fin de evaluar el impacto de la arquitectura de repertorios de instrucciones sobre el procesamiento superescalar, aplicando un método matemático derivado de la teoría de grafos, se propone un conjunto de programas como banco de pruebas.

El ámbito de aplicación corresponde al procesamiento de números enteros. El repertorio de instrucciones seleccionado es el x86 debido a sus peculiares características respecto al paralelismo a nivel de instrucción.

Se presenta la metodología de obtención de trazas de ejecución y la carga de trabajo de cada uno de los programas seleccionados.

Finalmente, se realiza una caracterización de cada programa en base a su funcionalidad y a los recuentos de operaciones y operandos.

Palabras clave: Evaluación de arquitecturas de computadores, paralelismo a nivel de instrucción, arquitectura del repertorio de instrucciones.

1. Necessity and purpose

The instruction sets architectures can be adapted to a variety of specifications with the purpose of optimizing its performance with respect to some aspect. Throughout the history of computation different criteria have prevailed: to minimize the space of representation with the purpose of generating smaller programs, diminishing the semantic gap between the assembler and the high-level languages to facilitate the work of the compilers, to reduce the compile time, to extend the instruction set life span, to reduce the power consumption, etc.

At the present time, the study of the instruction sets architectures behavior on superscalar processing, universally adopted by the general propose processors, receives great relevance.

On the other hand, the quantitative evaluation is a crucial point in the computer architecture research. The simulation has become the first evaluation tool but the construction of good simulators and the selection of suitable workloads are very delicate tasks [4]. Alternatively, we propose the mathematical analysis of execution traces based on the adaptation of the graph theory to the instruction level parallelism [5, 6].

Therefore, with the purpose of tackling the study of instruction sets architectures impact on the superscalar processing applying our mathematical analysis on execution traces, it is necessary the selection of:

- an instruction set;
- a method of obtaining traces; and
- the definition of a representative test-bench.

2. The x86 instruction set

In superscalar processing the more remarkable losses of performance are due to data dependences between instructions. These data dependences can be inherent from the algorithm that process the information or to other characteristics such as the resources limitation or to the instruction set architecture itself.

Some of those characteristics of the instruction set that cause additional dependences are the dedicated use of registers, the implicit operands (those that depend on the operation and are not specified by the programmer), the registers used for the memory address computation, the condition codes, etc.

The instruction set architecture of the family x86 is an example of all of these characteristics.

With the purpose of preserving the binary compatibility with previous processors, which has yielded undeniable benefits, the x86 instruction set has inherited design characteristics adapted to the past but clearly unfit for superscalar processing requirements. The original instruction set design persecuted two masterful lines: to minimize the space of representation of the instructions and to shorten the gap between the high-level languages and the machine language to ease the compilation process. Nowadays, these requirements are not important and, on the other hand, the limitations imposed by this architecture to the ILP are well known.

The x86 instruction set is the first candidate to attempt this study because to analyze its behaviour in the superscalar setting is extremely interesting.

In view of the results obtained by Huang and Peng, for DOS as well as for Windows95 [3], and considering the extra difficulty of the 32 bits environment due to the great variety of operands, a decision has made to restrict the analysis to 16 bits DOS real mode applications, and to defer the study of 32 bits operands for later work.

3. Execution traces generation

Execution traces generation is based on step-by-step execution mode and the modification of interrupt-service-routine 1 with the purpose of saving the binary format of the instruction in course. For each instruction, the maximum number of bytes that can occupy is saved, concretely 6 bytes for the 16 bits subset of x86 instruction set[1]. In many cases, to save the potential maxima length of instruction format supposes to keep much more information from the necessary one but it simplifies the tracing procedure and it can be used later to analyze other events, like for example, if the jumps are taken or not.

It is not necessary that the programs that are selected to comprise the test-bench have available their source code since we have constructed the tools necessary to be able to work from binary programs. When the source code is not available, the binary image of the program can be “injected” with a virus that causes the step-by-step execution mode before beginning the program itself.

We want to point out that the traces contain the complete sequence of execution for a specific workload, that is, the traced sequences are not partial but correspond to the complete program execution. The objective is not to work with partial mixes of instructions that do not represent the program real behaviour. Nevertheless, the traces do not include the instructions processed in the calls to the system.

4. Test-bench programs and workload

The programs of the test-bench are focused in the integer numbers processing. Real applications that execute the most representative possible variety of functions of the integer numbers processing have been chosen.

Specifically, the test-bench consists in 9 programs. Several operating system utilities from MS-DOS 5.0 (*comp*, *find* and *debug*) as well as

[1] This can cause some sporadic error in the situation of an instruction of 6 bytes with prefix. In that case, 7 bytes would be needed to save the complete instruction.

applications of common use have been selected: a compressing-decompressing utility (*rar* version 1.52) and a C language compiler (*tcc* version 1.0). Also one of the programs of the SPEC95int95 suite has been used, because having the source code in C language, with the intention to be able to make a comparative with binary images compiled under different optimizations.

The workload of these programs has been selected to obtain a reduced instruction count, in order for the trace files to be manageable. In spite of this, traces represent almost 190 million instructions.

Some recent works notice that metric and workload used to evaluate performance can affect the results since they are susceptible to interact [2]. That is, the methodology itself can affect the quantification.

In our case, the merely requirement for the workloads is that they do not generate too long processing. The fact of tracing the complete execution and that the selected programs cover a wide range of fan computational tasks integer numbers assures to dispose a quite real sampling. As far as the metric is concern, we can be sure of its rigor since it consists in applying a mathematical method over the data dependences found in the traces. In no case it interacts with the workloads.

Next, the traced programs and the workloads which they have been put under are described:

i. COMP. This utility of the Operating System MS-DOS (version 5.0) serves to compare the contents of two files searching differences. If the files have different size the comparison aborts since evidently they are different. If they have equal size it compares character to character writing down the differences in *stdout*.

The workload consists of the comparison of two files of 35Kbytes between which 10 differences have been prepared:

```
C:\>COMP file1.txt file2.txt /A
```

ii. DEBUG. It is a debugger of programs that allow assembling and disassembling code, dumping memory content, watching the processor registers, finding strings, executing code in step-by-step mode, etc.

The workload that has been attempted to obtain the trace consists of passing to the application a program and demanding, later in command line, to disassemble 32Kbytes of code:

```
C:\>debug file.exe
-u cs:0 l7fff
0D45:0000 BA9718        MOV     DX, 1897
0D45:0003 8EC2          MOV     ES, DX
0D45:0005 FA            CLI
0D45:0006 8ED2          MOV     SS, DX
0D45:0008 BC109E        MOV     SP, 9E10
0D45:000B FB            STI
0D45:000C B87822        MOV     AX, 2278
...
```

iii. FIND. It is an operating system utility from MS-DOS that scans for a string in a file and displays in *stdout* the lines in which the string has been found. It stands for an integer comparison task.

The trace corresponds to the execution of the utility scanning for a short string (only 4 characters) on a text file of 108,725 bytes:

```
C:\>FIND /N “data” file.txt
```

iv. GO T. It is the program GO, from the SPECint95 suite. It executes the game “go” against itself. The processing has a great part of pattern searching as well as look ahead logic. This type of programs usually consume even a third of its run time in data handling routines, as A. Fernández verifies in his work [7].

Since in this case the source code is available, two compilations with optimizations totally opposed have been made with the purpose of evaluating the possible impact of the compilation process: an optimization in size and another one in speed.

A compiler has been used which is not specifically designed for superscalar code generation: *Borland C++* version 4.0. An image for platform MS-DOS with the opportune optimizations and without debugging information is generated. The floating point operations are emulated, that means, all the execution code involves integer operations.

The present one is an optimization in size (compilation flags - O1 - Os - G).

It has been attained that the trace was not excessively long as a longer trace does not contribute to a different instruction mix but just a greater run time. The long traces cause excessively heavy analysis and the files that contain them difficult to handle. For that reason arguments that generate few move steps have been chosen. Specifically argument “30 4” give rise to 11 steps (moves) before leaving the program.

```
C:\>go 30 4
  1 B B4
  2 W D3
  3 B A2
  4 W C2
  5 B B3
  6 W C1
  7 B D2
  8 W C3
  9 B C4
 10 W pass
 11 B pass
Game over
```

v. GO V. Is the trace corresponding to the same previous source code with the same workload but the compilation has been optimized in speed (compilation flags - O2 - Ot - Ox - G).

vi. RAR C. Is the 1.52 version of August of 1994 of a compressing/decompressing utility that can work in command line or under a complete window *shell* designed in text video mode.

The trace has been generated working as compressing utility.

```
C:\>rar a –m5 –std file.rar @list
```

The files compressed are indicated in the argument file called "list" sending messages to *stdout* (*switch - std*) and with the maximum compression level (*switch - m5*). The list contains 17 files with a total of 543,437 bytes.

vii. RAR D. It corresponds to same previous program but working as decompressing utility.

The trace has been generated from the archive that was compressed previously (*file.rar*) and that occupies 147,489 bytes. The command line is:

```
C:\>rar e –std file.rar
```

viii. SORT. It is an operating system utility from MS-DOS that sorts the input information sending it to *stdout*, to a file or to a device. The input information can be a file or a command output. It represents a comparison task between integers within a sorting algorithm.

In our case, the input is a text file with a relation of names organized in columns separated by tabulators. The output is sent to a text file organized in the same way but in ascending order:

```
C:\>SORT <list.txt >sort.txt
```

ix. TCC. This program is the command line compiler of TURBO C++ 1.0 developing tools, of 1990.

A simple source file has been compiled with default options:

```
#include <stdio.h>

int main(void)
{
   char ch;

   printf("Input a character:");
   ch = getc(stdin);
   printf("The character input was: '%c'\n", ch);
   return 0;
}
```

```
C:\>Tcc example.c
```

5. Test-bench characterization

The test-bench characterization is based on the count of operations, operands, addressing modes, etc. Specifically have been made the following counts:

- distribution of operations;
- distribution of addressing modes;
- registers use;
- memory accesses;
- use of implicit operands;
- procedures calls;
- distribution of jumps; and
- stack traffic.

Based on these counts, the following measures have been evaluated:

- sequential run time;
- sequential CPI;
- so large means of the basic block, and
- average number of instructions by basic block that modify the state register.

The obtained information is very abundant. Table 1 can give us an idea of its volume by the amount traced instructions.

Table 1. Test-bench programs and its traces.

programs	trace file	trace file size	# of processed instructions
COMP	comp.des	4,193,220	689,866
DEBUG	debug.des	53,123,640	8,071,335
FIND	find.des	39,226,350	6,119,641
GO T	got.des	207,013,518	30,636,605
GO V	gov.des	204,972,426	30,290,351
RAR C	rar_c.des	735,383,556	98,244,064
RAR D	rar_d.des	110,220,936	14,782,924
SORT	sort.des	2,568,774	271,989
TCC	tcc.des	6,828,900	1,010,078
TOTAL of processed instructions			**190,116,853**

For the sake of clarity in presentation of our analysis we are going to show first the average results of operations use of the entire test-bench. Next, we are going to describe individually each one of the traced applications based on its operations distributions against the average values. A brief reference of operands use will accomplish in this part.

A small explanation should be done. Since our analysis deals with traces, every conditional branch is already resolved when they are written in the trace file. So the handling of traces assumes a perfect branch prediction allowing code sequences as long as we wish.

a. Average results

With aim to present data in the more organized possible mode, we are going to classify the instructions. The x86 instruction set can be organized in the following categories:

- data movement operations: related to the transference of data
- process operations: arithmetic and logic operations
- control flow operation: instructions that alter the execution sequence
- string operations: related to the management of strings of characters
- control operations: remainder instructions (state and control flags management, halt, etc.)

Next, we displayed the graph that illustrates how the average operation distribution by categories is for the complete test-bench.

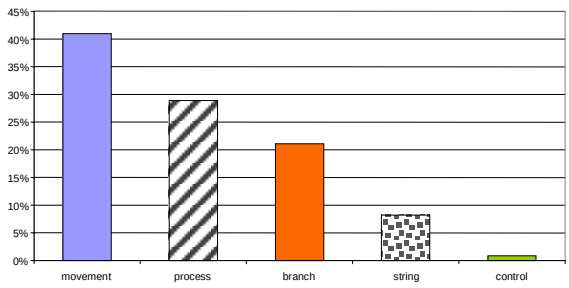


Fig. 1. Average operations distribution by categories.

The distribution is coherent with the results exposed in the work of Adams and Zimmerman about the instruction use in the 8086 processor [1].

The more used category is the data movement operations with something more than 40%. The predominant instruction in this group is MOV with more than 30% of occurrences in average value.

The second group by use amount is the process operations that include arithmetic and logic instructions. CMP instruction highlights (8.31%) followed by ADD (near 5%) and INC (4.21%). These three accumulate more than half of whole occurrences in the group. The execution of logic operations is less frequently than arithmetic ones. Among them SHL/SAL, OR and XOR highlight.

The control flow operations take 20% of every operations being the equality (or zero) or its opposed (inequality or not zero) the main evaluated condition.

Up to here the results are practically equal to the mentioned work. Some differences about string instructions can be appreciated. Its height is greater than the presented in this article due to two aspects. First, we have including in the test-bench a program that makes an intensive use of this type of instructions (SORT) and second, the metric we used with string operations has been different. The string instructions can be modified by a repetition prefix that provokes its iterative execution until the end of the string as it would be executed in a loop. Adams and Zimmerman count just once the string operation while measure the length of the string processed. Whereas, we counts every execution occurrence of these instructions as we would do if they belonged to a loop, without determining the string size.

Finally, the control instructions have a very small weight in the distribution.

For the sake of a deeper knowledge of each one of the programs in the test-bench we are going to present the operation distribution by categories for each one of them.

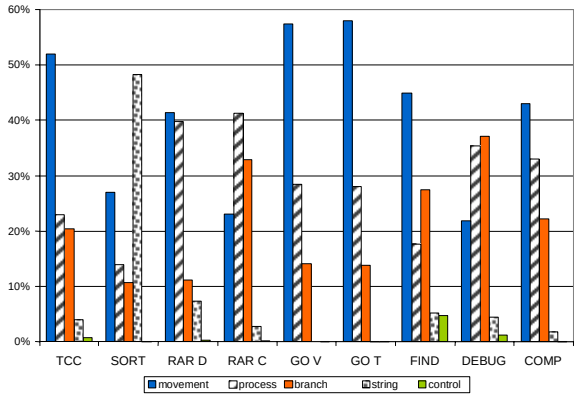


Fig. 2. Operation distribution by categories for each program in the test-bench.

At first sight, it is clear that the programs in the test-bench are very heterogeneous. A common pattern which the distributions adjust to is not found. The following programs are obviously away from the average distribution: SORT, by its excessive use of string operations; RAR C, due to the predominance of the process instructions; FIND, with a significant percentage of branches; and DEBUG with a very small number of data movement operations.

6. Individual characterization

Next, we are going to carry out a characterization of each one of the programs used as test-bench to lighting its future dynamic evaluation.

Specifically, the counts of instructions arranged by categories are presented and the top 25 more used are sorted in descendant order of percentage frequency. The accumulated percentage is indicated and the instructions responsible of the 90% executions occurrences have been shaded. The more used instructions of each category are plotted in a graph.

As a result of the count, the total number of executed instructions, procedures, system calls, and conditional branches are presented and the number of basic blocks and its average size has been calculated.

As far as the operands are concern, certain preliminary information, which will be completed and discussed in later sections, is shown: register and memory accesses as well as its percentage over the total executed instructions. The register accesses are understood as explicit, that is, the accesses associated to the operation code and not indicated in the instruction format are not considered. The registers readings provoke by the effective memory addressing calculation are also not included. In the accounting of memory accesses the addressing modes has not been distinguished.

The simulated trace execution allows obtaining time measurement that are also presented, in this part, merely for the sequential case, that is, the execution time when just a functional unit is available. From this time measurement and the instruction number we find out the sequential CPI.

To carry out the individual characterization the following scheme has been used:

- to describe what the program performs;
- to explain its operations profile related to the task it performs;
- to determine where the data are located (registers, memory);
- to count how many operations are made between registers as normally they are the fastest ones;
- to observe the amount of data movement operations with the stack as it escapes to memory accesses counts[2] and to try to decide if it corresponds to arguments passing or constrains in temporal register allocation;
- to pay attention to procedures calls both subroutines[3] and system calls[4];

[2] PUSH and POP instructions involve memory accesses in the same way that a instruction with data located in memory but they are not counted through the operands.

[3] Count of occurrences of CALL instruction.

[4] Count of occurrences of INT instruction.

- to regard the size of the basic block[5]; and finally,
- to compare its performance *vs.* the sequential CPI.

With respect to the stack traffic it is necessary to say that it has two sources: the argument passing to the procedures and the limited temporary storage (registers).

The C language programming model stores all the processor registers in the stack in each procedure call. It is a way of maintaining the coherence. It is accomplish by means of PUSH instructions. Then it passes the parameters, it performs the procedure, it returns the return value if exits and, finally, it pops the previously saved registers by means of POP instructions.

The RET *n* instruction adjusts the stack pointer to the place that indicates it (*n*) so that it is not used POP instruction to remove arguments parameters or local variables.

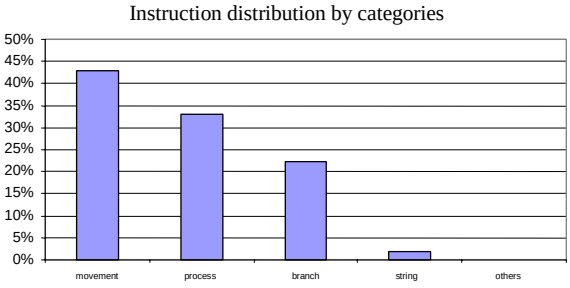
PUSH and POP also has a combined use in the code sequence due to the in the register file of and to the dedicated use of registers.

In summary, if the amount of executed PUSH and POP is similar we can conclude that it must not to be owed to argument passing but to limited temporary storage.

a. COMP program

Dynamic evaluation	
Instructions	689,866
Basic blocks	152,239
Procedures	565
System calls	39
Conditional branches	115,578
Register accesses	385,476
Memory accesses	506,830
Register accesses percentage	55.88%
Memory accesses percentage	73.47%
Instructions by basic block	4.53
Sequential execution time* (seg.)	0.124536660
Sequential CPI	18.05

* 8086 to 100MHz



category	**count**	**%**
data movement	296,796	43.02%
process	227,723	33.01%
branch	153,324	22.23%
		98.26%
string	11,955	1.73%
others	68	0.01%
		1.74%

	operation	%	accumulated
1	MOV	42.26%	42.26%
2	CMP	16.07%	58.33%
3	INC	15.84%	74.17%
4	JNB/JAE	10.24%	84.40%
5	JE/JZ	5.65%	90.05%
6	JMP	5.31%	95.36%
7	SCAS	0.66%	96.03%
8	STOS	0.65%	96.67%
9	JNE/JNZ	0.60%	97.27%
10	OR	0.35%	97.63%
11	PUSH	0.34%	97.97%
12	LODS	0.31%	98.28%
13	SUB	0.25%	98.53%
14	POP	0.21%	98.74%
15	DEC	0.21%	98.94%
16	LES	0.16%	99.11%
17	ADD	0.14%	99.25%
18	MOVS	0.11%	99.36%
19	LOOP	0.09%	99.45%
20	JS	0.09%	99.54%
21	CALL	0.08%	99.62%
22	RET	0.08%	99.69%
23	XCHG	0.04%	99.73%
24	CBW	0.03%	99.76%
25	SHL/SAL	0.02%	99.78%

↑ the instructions responsible of 90% of executions have been coloured

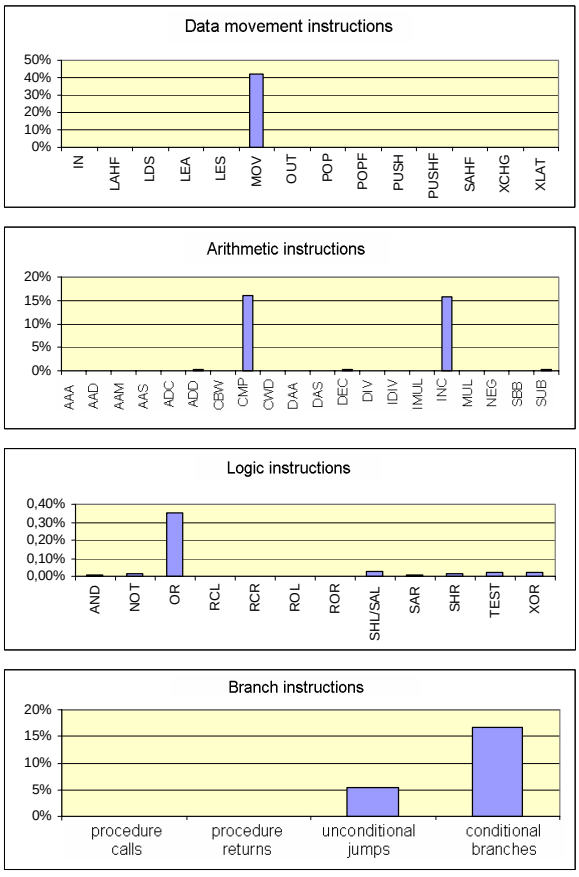


Fig. 3. COMP counts and statistics.

This program performs a character-by-character comparison looking for differences. When it finds differences shows them in screen. The program kernel is compound of comparison and conditional branch.

[5] By definition, a basic block is the instruction set that is executed in sequence so that once executed the first, always all the others are executed [7]. It is not necessary that all the basic blocks have a control flow instruction at the end, but simply that the first instruction is the destination of a branch. In this work, the number of basic blocks has been consider equal to conditional branches (including loops) plus unconditional ones since we do the count on execution traces.

The data movement operations are essential since the x86 instruction set does not allow performing the comparison between two values allocated in memory. The index update is also going to have an important weight in the execution.

The instruction distribution by categories is in the test-bench average. 90% of the executed instructions correspond just to 5 operations: 1 of data movement (MOV with approximately 43%); 2 of process (INC. CMP with something more than 30%); and 2 conditional branches (with almost 16%).

The size of the basic block is quite small. It is logical. In fact, the program only performs comparison and branch in a loop which is repeated as many times as the size of the files to compare. The rest of operations of the application, mainly the reading of the files, are done by means of system calls that are not traced.

The data memory accesses have a very high percentage with respect to both the test-bench average and the total number of data accesses in the program. The explicit register accesses percentage is under the average and the total of operations performed between registers is practically nothing. The stack traffic is also almost zero what gives idea that the program has not appreciated limitations in temporal storage in registers.

The procedure and system calls are extremely few in relation to the number of processed instructions.

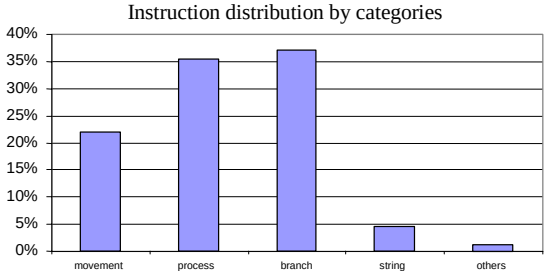
The sequential CPI, that is, the average number of cycles that takes executing an instruction when there is just a functional unit, is the greatest of all the test-bench programs and 6 cycles over the average that is near 12 cycles. This result seems logical considering the important memory traffic in data access. On the other hand, the data life span average is very short, as both items of the comparison change in each step, doing difficult any performance improvement.

b. DEBUG program

Dynamic evaluation	
Instructions	8,071,335
Basic blocks	2,609,294
Procedures	195,159
System calls	13,148
Conditional branches	2,076,253
Register accesses	4,996,936
Memory accesses	1,221,895
Register accesses percentage	61.91%
Memory accesses percentage	15.14%
Instructions by basic block	3.09
Sequential execution time* (seg.)	0.802615270
Sequential CPI	9.94

* 8086 to 100MHz

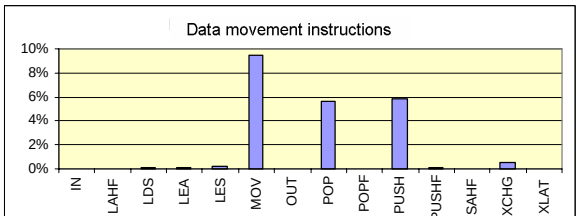
Instruction distribution by categories

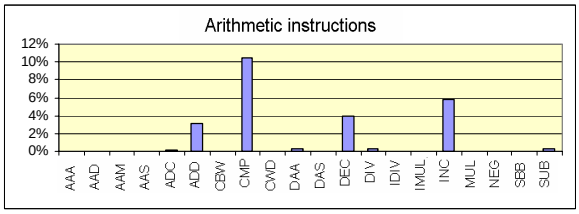


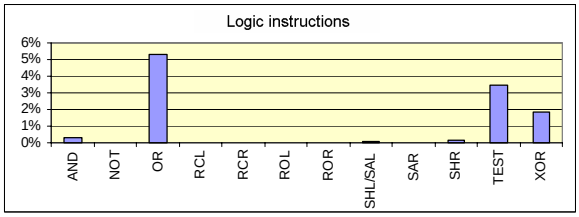
category	**count**	**%**
data movement	1,771,125	21.94%
process	2,852,731	35.34%
branch	2,990,187	37.05%
		94.33%
string	357,946	4.43%
others	99,346	1.23%
		5.67%

	operation	%	accumulated
1	JE/JZ	12.33%	12.33%
2	CMP	10.49%	22.82%
3	JNE/JNZ	10.38%	33.20%
4	MOV	9.45%	42.66%
5	JMP	6.60%	49.26%
6	PUSH	5.80%	55.06%
7	INC	5.72%	60.79%
8	POP	5.62%	66.41%
9	OR	5.28%	71.69%
10	DEC	3.91%	75.60%
11	TEST	3.49%	79.09%
12	ADD	3.06%	82.14%
13	STOS	2.50%	84.64%
14	CALL	2.42%	87.06%
15	RET	2.30%	89.36%
16	XOR	1.84%	91.20%
17	SCAS	1.46%	92.66%
18	JB/JNAE	1.35%	94.01%
19	JNB/JAE	0.92%	94.93%
20	XCHG	0.56%	95.48%
21	CLC	0.53%	96.02%
22	DIV	0.32%	96.34%
23	STC	0.30%	96.64%
24	SUB	0.29%	96.93%
25	JBE/JNA	0.28%	97.21%

† the instructions responsible of 90% of executions have been coloured







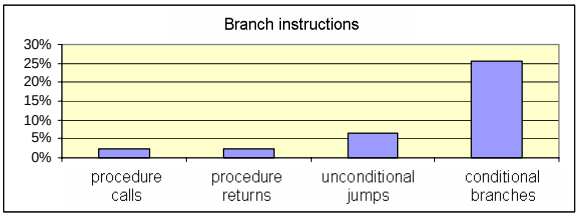


Fig. 4. DEBUG counts and statistics.

The workload has consisted in disassembling a binary program. The process requires the comparison of bit fields with patterns following different itineraries based on the partial results.

The instruction distribution does not fit to the test-bench average. The most used instructions are the control flow ones. We need 16 instructions to reach a 90% of the total processed. The evaluation of zero condition code and the opposite (non-zero) are the main operations within this spreading instruction mix. They sum more than 22%. The weight of the unconditional branch (almost 7%) is important. This corresponds to the switch-case structures used to implement the pattern searching. The control flow instructions are completed with the use of procedure calls and returns. The total number of procedures jumps is quite great in comparison with the rest of programs. This can be justified because field decoding of the instruction format is a modular and repetitive process.

The process instructions are in the second position. With a third of the total, CMP highlights, used to search the decoding patterns. Other process instructions very employ are INC (almost a 6%), OR (slightly more than 5%) and DEC (approximately a 4%).

Finally, the data movement instructions are dominated by MOV (9.45%), PUSH (5.8%) and POP (5.62%). The stack traffic is considerable and gives idea that the register file is not longer enough for the necessities of temporary storage.

The percentage of data that reside in memory is very small (approximately a 15% over the total of instructions). Also, the percentage of data that reside in registers and the operations that are performed between registers are slightly below than average.

This program performs more system calls than the rest in several orders of magnitude. It is necessary to consider that displaying data in screen is a fundamental part of the program which does not happen in the rest of traced applications. Here, for each disassembled instruction, its memory address in *base:displacement* format, the hexadecimal machine code and the assembly language string occupying more than 40 characters are shown in *stdout*.

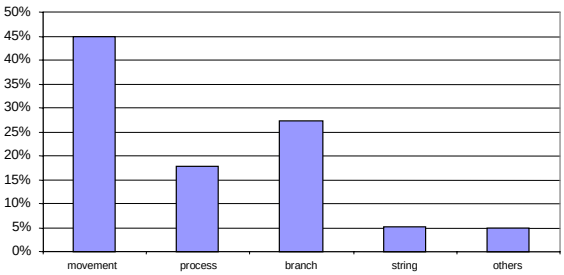
Notice that the basic block size is one of the minors (3 instructions) and that the sequential CPI is one of the best of the whole test-bench (9.94), due to the small percentage of memory accesses.

c. FIND program

Dynamic evaluation	
Instructions	6,119,641
Basic blocks	1,092,370
Procedures	291,984
System calls	151
Conditional branches	797,328
Register accesses	4,132,807
Memory accesses	979,543
Register accesses percentage	67.53%
Memory accesses percentage	16.01%
Instructions by basic block	5.60
Sequential execution time* (seg.)	0.691962840
Sequential CPI	11.31

* 8086 to 100MHz

Instruction distribution by categories



category	**count**	**%**
data movement	2,749,843	44.93%
process	1,083,367	17.70%
branch	1,676,339	27.39%
		90.03%
string	318,271	5.20%
others	291,821	4.77%
		9.97%

	operation	%	accumulated
1	MOV	16.16%	16.16%
2	PUSH	14.39%	30.55%
3	POP	14.39%	44.93%
4	CMP	4.84%	49.77%
5	JMP	4.82%	54.59%
6	CALL	4.77%	59.36%
7	RET	4.77%	64.14%
8	INC	4.76%	68.90%
9	CLC	4.73%	73.63%
10	SCAS	3.42%	77.05%
11	JNE/JNZ	3.33%	80.38%
12	JE/JZ	3.22%	83.60%
13	SUB	3.19%	86.79%
14	JNB/JAE	3.17%	89.95%
15	JCXZ	1.70%	91.66%
16	DEC	1.70%	93.36%
17	TEST	1.62%	94.98%
18	JB/JNAE	1.60%	96.59%
19	LODS	1.60%	98.19%
20	ADD	1.57%	99.76%
21	MOVS	0.10%	99.86%
22	STOS	0.08%	99.94%
23	STC	0.03%	99.98%
24	LOOP	0.01%	99.98%
25	XOR	0.00%	99.99%

† the instructions responsible of 90% of executions have been coloured

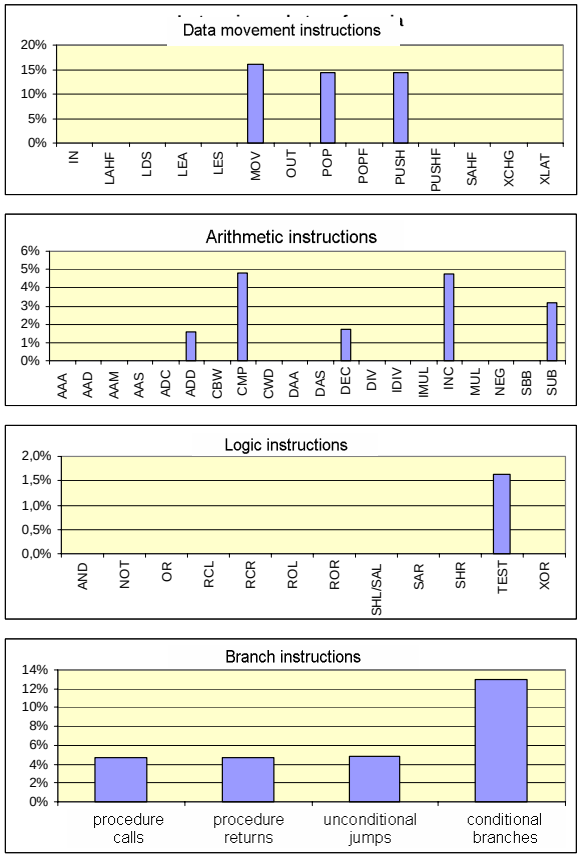


Fig. 5. FIND counts and statistics.

The task traced by FIND utility corresponds to the search of a short string of 4 characters in a text file of 108,725 bytes. In this case, the comparison is made with a pattern string. As soon as a difference takes place, a new data set from the input file is read.

The instruction distribution does not adjust to the test-bench average due to the bifurcation percentage. Given the utility purpose, to find so many branch operations is not strange.

The data movement instructions highlight among top 25 more used instructions. MOV instruction is in the first place with a 16%. However, it is more interesting to see that the use of PUSH and POP follow the MOV one, with an identical percentage in both cases of 14.39%. Without any doubt it must be due to the continuous data in and out to/from the register file forced by limited temporary storage.

The process instructions almost give rise to 18% of the executed operations just by three instructions: CMP, logically due to the application nature, with a 5%, and INC and SUB with 5% and 3% respectively.

The rest of instructions, responsible of the 90% of the work done by the processor, correspond to control flow operations. There are a great number of unconditional branches (almost a 5%) because the comparisons with the pattern string usually are aborted, in most of the cases, before reaching the end. There are a great number of call/returns to/from procedures (near 5%): quite superior to the rest of test-bench programs. Finally, three kinds of conditional branches take 9% of occurrences.

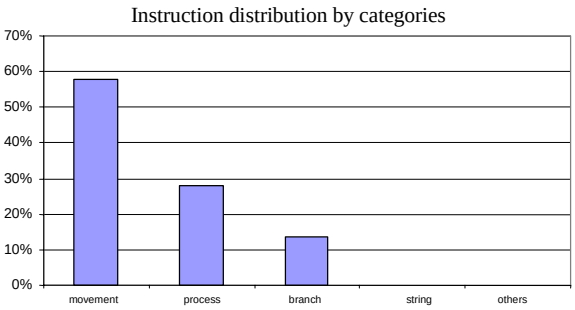
The data reside basically in registers, being those data located in memory the smallest percentage of all test-bench together with SORT. Nevertheless, it is necessary to notice that the traffic with memory through the stack (15%) is even greater than the caused by the explicit data accesses to memory (13%).

The number of system calls is irrelevant. The sequential CPI and the basic block size are in the average of the test-bench.

d. GO program optimized for size

Dynamic evaluation	
Instructions	30.636.605
Basic blocks	3.829.543
Procedures	197.427
System calls	32
Conditional branches	2.972.717
Register accesses	32.214.654
Memory accesses	10.300.772
Register accesses percentage	105.15%
Memory accesses percentage	33.62%
Instructions by basic block	8.00
Sequential execution time* (seg.)	3.782201650
Sequential CPI	12.35

* 8086 to 100MHz



category	**count**	**%**
data movement	17.772.338	58.01%
process	8.610.893	28.11%
branch	4.224.388	13.79%
		99.91%
string	9.117	0.03%
others	19.869	0.06%
		0.09%

	operation	%	accumulated
1	MOV	49.66%	49.66%
2	ADD	12.20%	61.86%
3	CMP	9.34%	71.20%
4	PUSH	4.14%	75.34%
5	JNE/JNZ	3.87%	79.22%
6	JE/JZ	2.88%	82.10%
7	JMP	2.80%	84.89%
8	POP	2.50%	87.39%
9	AND	2.46%	89.86%
10	INC	1.88%	91.74%
11	LES	1.46%	93.20%
12	JNLE/JG	0.91%	94.11%
13	SUB	0.87%	94.98%
14	JL/JNGE	0.86%	95.84%
15	CALL	0.64%	96.48%
16	RETF	0.64%	97.13%
17	JNL/JGE	0.58%	97.71%
18	JLE/JNG	0.56%	98.27%
19	DEC	0.35%	98.61%
20	LEA	0.25%	98.86%

21	SHL/SAL	0.24%	99.09%
22	IMUL	0.22%	99.32%
23	XOR	0.17%	99.49%
24	TEST	0.16%	99.65%
25	OR	0.16%	99.82%

↑ the instructions responsible of 90% of executions have been coloured

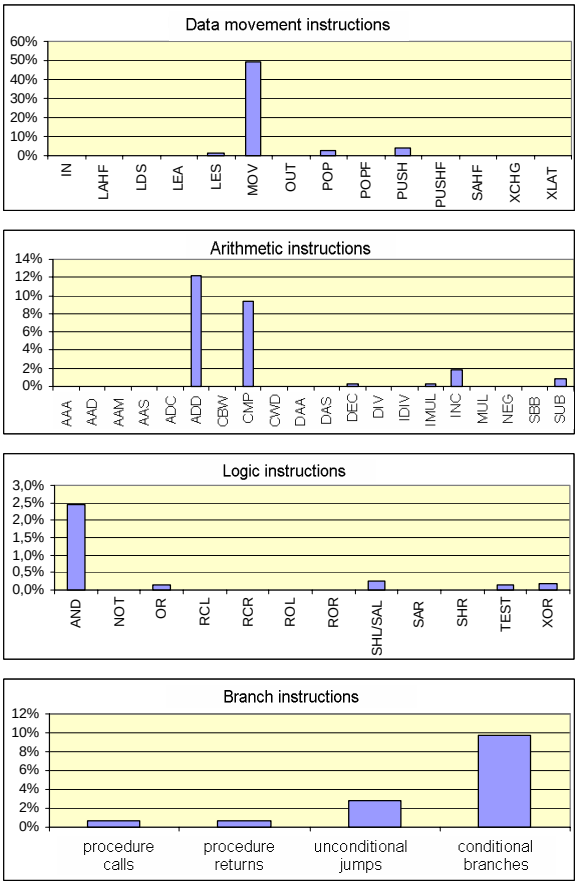


Fig. 6. GO (optimized for size) counts and statistics.

The fourth trace corresponds to program GO from SPECint95 *suite* compiled with the optimized for size option (it refers to the program static image). It is verified that it saves space: 596,602 bytes against 599,018 bytes of the same code compiled with the optimization for speed, although the difference is ridiculous (less than 3,000 bytes).

The frequency of use distribution by categories adjusts to the test-bench average being despicable the amount of string instructions and those of the class 'others'. It is the program that uses more data movement instructions reaching almost 60%. Only 10 instructions take 90% of the executed ones and among them, the use of MOV emphasizes with a 50%. The accesses to the stack, with PUSH and POP, increase a 7% the frequency of the transferences.

The process instructions append slightly more than 28% thanks to the execution of ADD (12.20%), CMP (9.34%), AND (2.46%) and INC (almost 2%).

The branch instructions are represented by the evaluation of zero and non-zero conditional codes and by the unconditional jump.

The accesses to data in memory are in the average although it is necessary to emphasize the stack traffic due to PUSH and POP. The use of operands located in registers is quite over the average being important the amount of instructions that operate between registers. Everything indicates that the architecture with more general propose registers, capable to save temporary data without transferring them to memory, would optimize the program performance.

The number of procedure calls is not great and the system calls are despicable against the executed instructions. The sequential CPI is in the average and the size basic block is over the average.

The obtained results are consistent with those presented in the work of Agustín Fernández [7] in which it makes a review of the SPEC95 programs executed on an Alpha architecture:

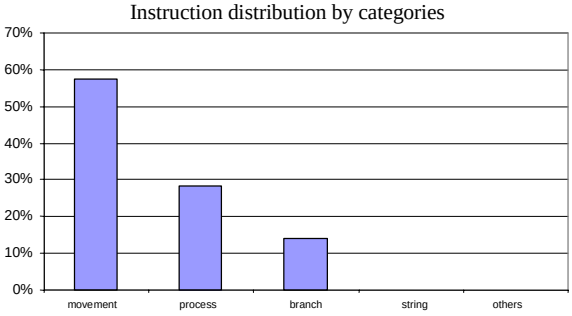
Table 2. Comparative of GO on two architectures.

8086	**concept**	**Alpha** [7]
58.01%	data movement instructions	50.42%
28.11%	process operations	37.58%
13.79%	control flow operations	12.00%
12.0%	basic blocks (over operations)	14.0%
0.64%	procedures (over operations)	0.86%
8.0	operations by basic block	6.9
MOV		LDx/STx
ADD	more used operation codes	ADDx
CMP		CMP
JNE		BNE

e. GO program optimized for speed

Dynamic evaluation	
Instructions	30,290,351
Basic blocks	3,881,186
Procedures	197,032
System calls	21
Conditional branches	2,982,710
Register accesses	33,017,158
Memory accesses	8,995,946
Register accesses percentage	109.00%
Memory accesses percentage	29.70%
Instructions by basic block	7.80
Sequential execution time* (seg.)	3.478782710
Sequential CPI	11.48

* 8086 to 100MHz



category	**count**	**%**
data movement	17,390,698	57.41%
process	8,604,322	28.41%
branch	4,275,243	14.11%
		99.93%
string	280	0.00%
others	19,808	0.07%
		0.07%

	operation	%	accumulated
1	MOV	49.38%	49.38%
2	ADD	12.56%	61.94%
3	CMP	8.80%	70.74%
4	JNE/JNZ	4.77%	75.51%
5	PUSH	3.98%	79.49%
6	JMP	2.97%	82.46%
7	AND	2.49%	84.95%
8	JE/JZ	2.07%	87.01%
9	POP	2.05%	89.06%
10	INC	1.94%	91.00%
11	LES	1.77%	92.78%
12	JNLE/JG	0.90%	93.68%
13	OR	0.85%	94.53%
14	JL/JNGE	0.77%	95.30%
15	JNL/JGE	0.72%	96.02%
16	CALL	0.65%	96.67%
17	RETF	0.65%	97.32%
18	JLE/JNG	0.58%	97.90%
19	SUB	0.54%	98.44%
20	DEC	0.35%	98.79%
21	SHL/SAL	0.24%	99.03%
22	IMUL	0.23%	99.25%
23	LEA	0.22%	99.48%
24	XOR	0.21%	99.68%
25	TEST	0.17%	99.85%

↑ the instructions responsible of 90% of executions have been coloured

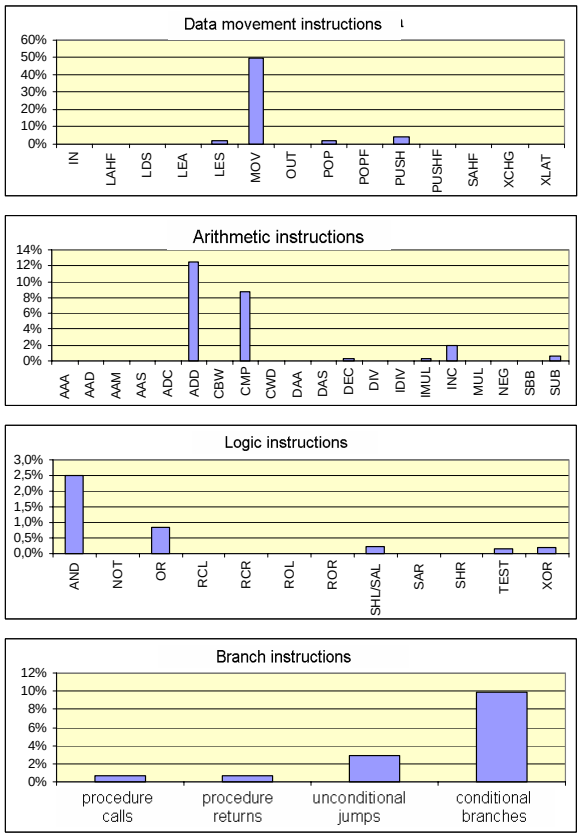


Fig. 7. GO (optimized for speed) counts and statistics.

This trace corresponds to the execution of the same source code that in the previous case but compiled with the optimized for speed option. It is obvious that the optimization has been doubly effective since it executed less instructions (around a 1% less) and the sequential CPI has been reduced in almost a cycle with respect to the previous version (11.48 in place of 12.35). The binary image of the program is something larger than the previous one. This demonstrates that a static saving of memory does not bring any advantage in run time. The optimized for size option is inherited of past requirements, when the memory was a small and expensive recourse, but now it does not imply a performance improvement.

The commentaries that can be done are very similar to those of the previous trace: the frequency of use is practically equal to the previous program, the instructions responsible for the 90% of the executions are the same ones, etc. Perhaps, we can talk of subtle differences.

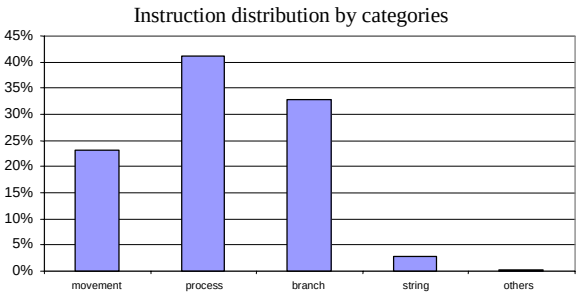
It is observed that the use of CMP instruction has descended half a point in percentage, that is, the program makes less comparison. Also, slightly smaller stack traffic is performed. The use of data in registers and operations between registers grows and diminishes the data memory accesses. Really, the performance has improved minimizing the comparison and the transferences with memory.

The size of the basic block is something smaller than in previous trace as the total of branches is similar but the number of instructions processed is smaller.

f. RAR programa compressing

Dynamic evaluation	
Instructions	98,244,064
Basic blocks	32,234,587
Procedures	18,127
System calls	751
Conditional branches	31,391,554
Register accesses	71,641,112
Memory accesses	30,680,803
Register accesses percentage	72.92%
Memory accesses percentage	31.23%
Instructions by basic block	3.05
Sequential execution time* (seg.)	9.347562283
Sequential CPI	9.51

* 8086 to 100MHz



category	**count**	**%**
data movement	22,649,984	23.05%
process	40,437,561	41.16%
branch	32,270,837	32.85%
		97.06%
string	2,730,412	2.78%
others	155,270	0.16%
		2.94%

	operation	%	accumulated
1	MOV	21.71%	21.71%
2	JE/JZ	19.90%	41.61%
3	SHL/SAL	12.72%	54.33%
4	CMP	10.23%	64.56%

5	JB/JNAE	5.77%	70.33%
6	XOR	3.73%	74.06%
7	DEC	3.49%	77.55%
8	TEST	2.45%	80.01%
9	JNB/JAE	2.33%	82.33%
10	ADD	2.23%	84.57%
11	JNE/JNZ	2.17%	86.74%
12	CMPS	2.00%	88.73%
13	SUB	1.75%	90.48%
14	AND	1.56%	92.04%
15	SHR	1.21%	93.25%
16	JMP	0.86%	94.11%
17	INC	0.80%	94.91%
18	XCHG	0.67%	95.58%
19	LOOP	0.62%	96.19%
20	STOS	0.60%	96.79%
21	JLE/JNG	0.33%	97.12%
22	OR	0.32%	97.44%
23	NOT	0.32%	97.77%
24	LDS	0.32%	98.09%
25	JNBE/JA	0.31%	98.40%

† the instructions responsible of 90% of executions have been coloured

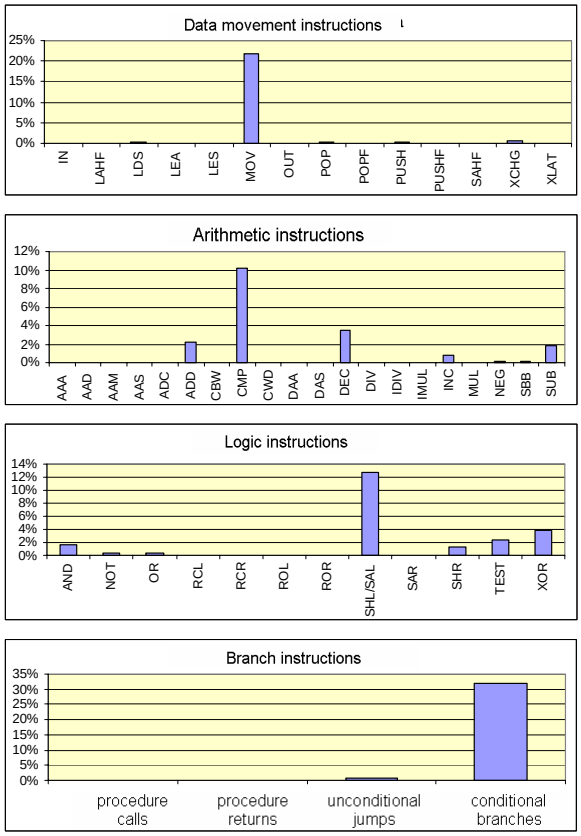


Fig. 8. RAR (compressing) counts and statistics.

This trace corresponds to the execution of RAR as compressing utility.

The instructions distribution does not adjust to the average since the process and control flow ones prevail over the data movement. The 90% of the processed operations is supported by 13 instructions and among them the first is MOV that compiles practically all the transferences (near 22%). The stack is not used as temporary storage since PUSH and POP have a despicable percentage of use.

The process operations are represented mainly by the left logic displacement (SAL) with almost a 13% and the comparisons with more than a 10%. Other arithmetical instructions or logics are: XOR (3.73%), DEC (3.49%), TEST (2.45%), ADD (2.23%) and SUB (1.75%).

The branch instructions are represented by 4 different types of conditional branches, occupying the branch if-zero the second place among the more used instructions with almost a 20%.

The use of operands in memory is in the average whereas the use of operands in registers is below the average. Nevertheless, the percentage of operations between registers is especially elevated.

The system calls and subroutines are practically void.

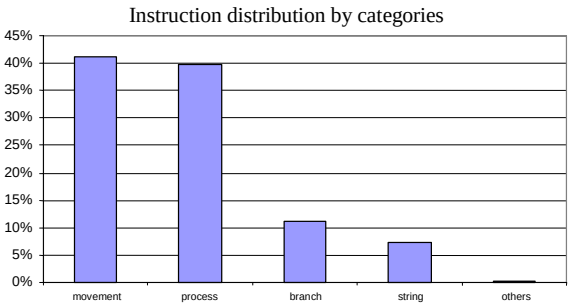
The sequential CPI is the best of the entire test-bench proposed in this research with 9.51. Surely, this is due to the great amount of operations between registers, those that consume less cycles in this architecture, and to the reduced stack traffic.

The basic block size is also the minor among the whole test-bench.

g. RAR program decompressing

Dynamic evaluation	
Instructions	14,782,924
Basic blocks	1,629,633
Procedures	14,735
System calls	254
Conditional branches	1,398,728
Register accesses	15,633,667
Memory accesses	5,006,326
Register accesses percentage	105.75%
Memory accesses percentage	33.87%
Instructions by basic block	9.07
Sequential execution time* (seg.)	1.552791730
Sequential CPI	10.50

* 8086 to 100MHz



category	**count**	**%**
data movement	6,106,813	41.31%
process	5,875,812	39.75%
branch	1,659,101	11.22%
		92.28%
string	1,093,980	7.40%
others	47,218	0.32%
		7.72%

	operation	%	accumulated
1	MOV	39.01%	39.01%
2	XOR	8.95%	47.96%
3	SUB	6.23%	54.19%
4	CMP	4.56%	58.75%
5	ADD	4.47%	63.22%
6	SHR	4.38%	67.60%
7	STOS	3.72%	71.32%

8	SHL/SAL	3.56%	74.88%
9	MOVS	3.43%	78.31%
10	JNB/JAE	1.94%	80.25%
11	AND	1.83%	82.08%
12	INC	1.82%	83.89%
13	DEC	1.59%	85.48%
14	JMP	1.56%	87.04%
15	LOOP	1.50%	88.54%
16	XCHG	1.42%	89.96%
17	JNE/JNZ	1.33%	91.29%
18	JNBE/JA	1.25%	92.54%
19	JS	1.15%	93.69%
20	JE/JZ	1.10%	94.79%
21	SBB	1.04%	95.84%
22	JB/JNAE	0.72%	96.56%
23	ADC	0.60%	97.16%
24	JBE/JNA	0.46%	97.62%
25	PUSH	0.40%	98.02%

↑ the instructions responsible of 90% of executions have been coloured

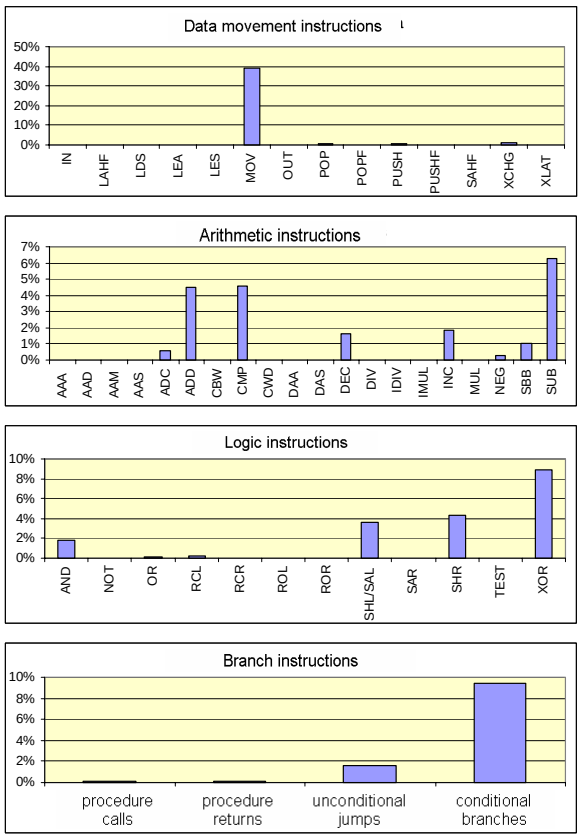


Fig. 9. RAR (decompressing) counts and statistics.

The seventh trace has been composed running RAR as a decompressing utility. The previously compressed file has been taken as input recovering its contained files. The compressing task executes much more instructions than the decompressing one (almost 100 million rather than about 15). The justification seems to be because the decompression has a determinist output whereas the compression is a heuristic task, cradle in test algorithms.

The use profile is totally different from the previous trace. It is almost adjusted to the average with a process operations percentage somewhat above.

Among the 25 top used instructions there are up to 6 different conditional branches but without an appreciable weight. The amount instructions responsible of the 90% of the processing are also considerable. All it gives idea that the instructions distribution is extraordinarily spread.

The first instruction is the MOV with a 39% of the total and it takes the complete percentage due to data movement instructions.

The arithmetic/logic operations include up to 9 instructions with more than 35% of the total percentage.

The bifurcations are represented by several instructions among which to mention LOOP should be done by its infrequent use in the test-bench.

It is necessary to emphasize that a 7% of executed instructions belong to string handling.

This trace is the one that more operations make between registers and one of which uses more data allocated in registers. The use of memory operands is in the average. The stack traffic is very small. We deduce that the number of registers is sufficient for temporary storage.

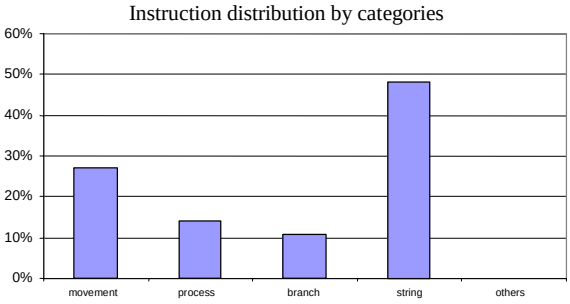
It has a good sequential CPI. The great number of fast operations (those are executed between registers) contributes to it although it is a little waned by a use of memory operands slightly superior to the case of RAR as compressing utility.

The size of the basic block is one of the larger ones of the test-bench, with 9 instructions. Both the system and the procedures calls are despicable.

h. SORT program

Dynamic evaluation	
Instructions	271,989
Basic blocks	29,144
Procedures	9
System calls	15
Conditional branches	28,802
Register accesses	130,653
Memory accesses	32,792
Register accesses percentage	48.04%
Memory accesses percentage	12.06%
Instructions by basic block	9.33
Sequential execution time* (seg.)	0.028361530
Sequential CPI	10.43

* 8086 to 100MHz



category	**count**	**%**
data movement	73,362	26.97%
process	38,008	13.97%
branch	29,162	10.72%
		51.67%
string	131,352	48.29%
others	105	0.04%
		48.33%

	operation	%	accumulated
1	MOVS	44.75%	44.75%
2	MOV	11.76%	56.50%
3	PUSH	6.00%	62.50%
4	POP	6.00%	68.50%
5	ADD	4.50%	73.00%
6	XLAT	3.22%	76.22%
7	CMP	3.15%	79.37%
8	SUB	3.05%	82.42%
9	JNBE/JA	2.93%	85.36%
10	SCAS	1.89%	87.25%
11	INC	1.71%	88.96%
12	LODS	1.61%	90.56%
13	LOOPZ/LOOPE	1.61%	92.17%
14	JNE/JNZ	1.57%	93.74%
15	JE/JZ	1.50%	95.24%
16	OR	1.48%	96.73%
17	JB/JNAE	1.47%	98.20%
18	JNB/JAE	1.47%	99.66%
19	JMP	0.13%	99.79%
20	DEC	0.06%	99.85%
21	STOS	0.05%	99.89%
22	JCXZ	0.04%	99.93%
23	CLD	0.02%	99.95%
24	SHR	0.02%	99.97%
25	STD	0.02%	99.98%

† the instructions responsible of 90% of executions have been coloured

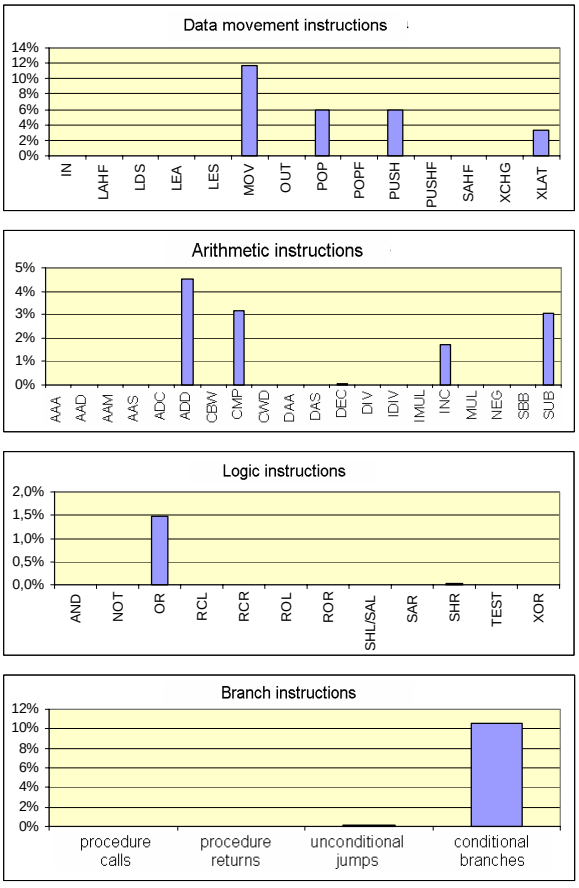


Fig. 10. SORT counts and statistics.

This trace belongs to the execution of SORT utility on a disordered file generating a sorted new one. It implies the application of a sorting algorithm on strings and the copy of the sorted ones.

The instruction distribution by categories leaves the average as far as the string operations are concerned. In fact, if the average profile of the test-bench has a percentage of a little more than 8% for this category is due to this trace. Obviously, if instead of counting each repetition of the string instruction it would be counted just its appearance in the code sequence, the total percentage would adjust better to the rest of the programs.

Twelve are the instructions that monopolize 90% of the executed operations. MOVS is the first with approximately a 45%. Other string instructions are SCAS and LODS with a 1.89% and a 1.61% respectively. Notice that once the strings have been sorted they have to be copied completely in the output. The percentage of MOVS occurrences is due to that.

The 3 consecutive places after MOVS are occupied by data movement instructions. The second position is for MOV with almost a 12% and PUSH and POP follow it with a total of 12% distributed to equal parts

Among the process operations we can emphasize the use of ADD (4.5%), CMP (3%), SUB (3%) and INC. (almost 2%).

The branch instructions take a 10% over the total executed instructions but a unique conditional branch instruction (JNBE/JA with almost a 3%) appears among the operations responsible of 90% of the executed instructions. In fact, the string instructions are repeated since they work in the same way as control flow structures based on loops. For that reason, the number of implicit conditional jumps is larger.

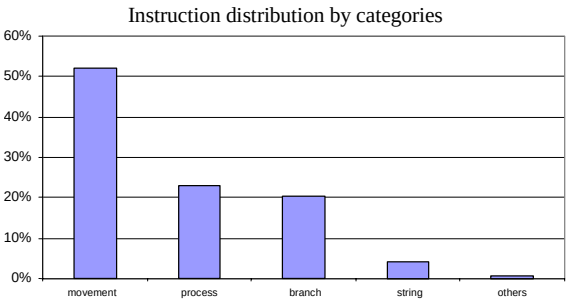
The SORT trace uses few data, as much in registers as in memory, in comparison with the test-bench average values. Nevertheless, they are implicit in the string instructions. Also, considerable memory traffic is observed thanks to PUSH/POP instructions without having a significant number of procedure calls. Then, all the stack traffic is due to limitations in temporary storage. The system calls are inappreciable.

The sequential CPI is good and the size of the basic block quite large although if we took the repetitions due to string instructions as conditional branches, the basic block would be smaller.

i. TCC program

Dynamic evaluation	
Instructions	1,010,078
Basic blocks	159,463
Procedures	23,182
System calls	56
Conditional branches	124,397
Register accesses	770,882
Memory accesses	370,182
Register accesses percentage	76.32%
Memory accesses percentage	36.65%
Instructions by basic block	6.33
Sequential execution time* (seg.)	0,142059500
Sequential CPI	14,06

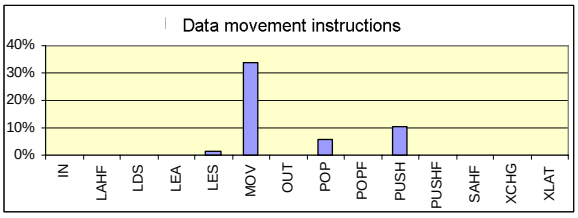
* 8086 to 100MHz

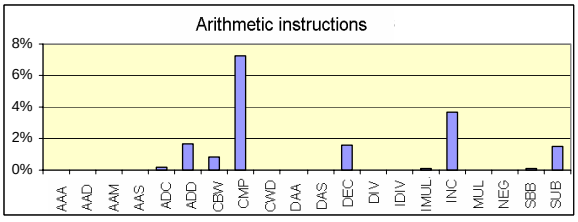


category	count	%
data movement	524,947	51.97%
process	230,835	22.85%
branch	205,812	20.38%
		95.20%
string	40,892	4.05%
others	7,592	0.75%
		4.80%

	operation	%	accumulated
1	MOV	33.87%	33.87%
2	PUSH	10.64%	44.51%
3	CMP	7.29%	51.80%
4	POP	5.78%	57.58%
5	JE/JZ	3.99%	61.58%
6	JNE/JNZ	3.79%	65.37%
7	INC	3.66%	69.02%
8	JMP	3.47%	72.49%
9	CALL	2.30%	74.79%
10	OR	2.10%	76.89%
11	RETF	1.74%	78.64%
12	ADD	1.63%	80.26%
13	DEC	1.59%	81.85%
14	SCAS	1.57%	83.42%
15	SUB	1.53%	84.95%
16	LES	1.41%	86.37%
17	STOS	1.37%	87.74%
18	JB/JNAE	1.31%	89.05%
19	LOOP	1.18%	90.24%
20	SHL/SAL	1.08%	91.32%
21	XOR	0.99%	92.31%
22	TEST	0.90%	93.21%
23	CBW	0.86%	94.07%
24	LODS	0.86%	94.94%
25	RET	0.55%	95.49%

† the instructions responsible of 90% of executions have been coloured





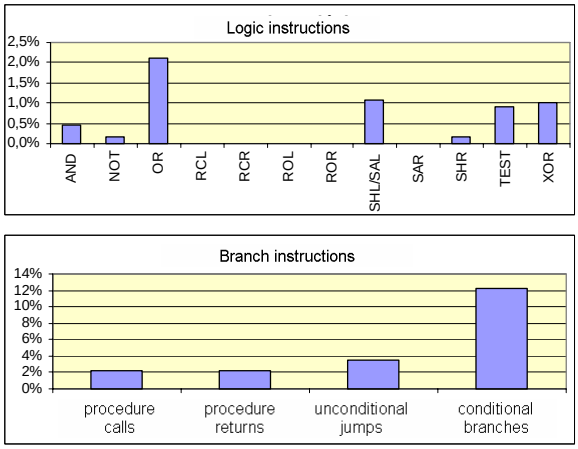


Fig. 11. TCC counts and statistics.

It is the trace generated by the compilation of a simple source program through TCC. The distribution profile is similar to the average with a branch instruction weight something larger than the average as it must be in a compiler since the analysis phase implies patterns searching and batteries of control flow structures of *switch-case* type.

The instruction set responsible of the 90% of the entire execution is large, what has to do with a great dispersion of operations. The first instruction is the MOV, with almost a 34% of use frequency, followed of PUSH with a 10.64%. The POP instruction is in fourth place with near a 6%.

Among the process instructions, CMP highlights in third position with a 7.29% of total frequency. It is logical because this operation is frequently repeated due to the pattern searching. Other operations of this class are INC, OR, ADD, DEC and SUB that accumulate around 10%.

The branch instructions are represented mainly by three conditional ones, the unconditional jump and a high percentage of procedures calls/returns. Notice, that LOOP is used in more of a 1%.

Notice how LES instruction and far return (RETF) are used. Without doubt, it must be due to the program size (it is the larger one after GO) what implies the continuous segment change.

Also some string instructions are used.

The operands both allocated in registers and in memory, adjust to the test-bench average values. The number of operations between registers, nevertheless, is below the average.

The stack traffic is very important.

The system calls are not significant.

The sequential CPI is not very good being 2 cycles over the average. The basic block size is just in the average value.

7. References

[1] T. L. Adams and R. E. Zimmerman, "An analysis of 8086 instruction set usage in MS DOS programs," in *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-III)*, April 1989, pp. 152 - 160.

[2] D. G. Feitelson. "Metric and Workload Effects on Computer Systems Evaluation," *IEEE Computer*, vol. 36, 9, September, 2003.

[3] I. J. Huang and T. C. Peng, "Analysis of x86 Instruction Set Usage for DOS/Windows Applications and Its Implication on Superscalar Design," *IEICE Transactions on Information and Systems*, Vol.E85-D, No. 6, pp. 929-939, June 2002. (SCI).

[4] K. Skadron, M. Martonosi, *D.* I. August, M. *D.* Hill, *D.* J. Hill and V. S. Pai. "Challenges in Computer Architecture Evaluation," *IEEE Computer*, vol. 36, 8, August, 2003.

[5] R. Durán, R. Rico, "On Applying Graph Theory to ILP Analysis," Technical Note TN-UAH-AUT-GAP-2005-01-en. Available at: http://atc2.aut.uah.es/~gap/

[6] R. Durán, R. Rico, "Quantification of ISA Impact on Superscalar Processing," in *Proceedings of Eurocon2005*, November 2005.

[7] A. Fernández., "Un análisis cuantitativo del Spec95," Informe técnico UPC-DAC-1999-12, Universidad Politécnica de Cataluña, Barcelona, 1999.