# Estudi Previ

1. Significat “inlinging” de funció.

El compilador fica el codi de la funció en el codi on es crida a la funció en compte de fer tota la complexitat de saltar a una nova rutina.

1. Quin parametre de gcc permet fer “inlining”. Digues si s’activa al fer “-O2”. Perquè serveix “-finline-limit”?

Per permetre “inlining” podem fer servir “-finline-small-functions” i farà inline d’aquelles funcions que el codi de tota la gestió de la crida + de la funció sigui superior al de la funció. Si, quan activem “O2” s’activa el inlining”.

“-finline-limit”: Limita la mida de les funcions que el compilador ha de fer inline.

<https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html>

1. Forma de saber si ensamblador hi ha la funció “Juanito”. Digues també com saber si la funció és invocada o no.

“gcc -S file.c” i quan tinguem el fitxer en codi “.asm” hem de fer “cat file.S | grep Juanito”.

Si obtenim resultats i hi ha un “call Juanito” podrem dir que, almenys, hi ha una linea que s’encarregaria d’invocar-la (Però hauríem d’analitzar el codi per saber si realment s’invoca).

1. Calcula num inst. Estàtiques i dinàmiques.

Digues MIPS, IPC, CPI i Freq.

# t\_exec = 14ms ; cicles = 16x10^6

```pseudo-asm

%eax = $(sp + 0x5c) // i = \*pos\_mem

% (sp + 0x58) += %eax // res += i

% (sp + 0x5c) += 1 // \*pos\_mem++

if (%(sp + 0x5c) - 0xf423f) no\_jump // Si es 0, s’activa ZF i salta

inici

```

Nº inst. Estàtiques: 5

Nº inst. Dinàmiques: 999.999(5) + 4 = 4.999.999 = N 🡪 N = 5x10^6

MIPS = N/(14ms \* 10^6) = **357 = MIPS**

CPI = 16x10^6 / 5x10^6 = **3.2 = CPI**

IPC = CPI^-1 = **0.3125 = IPC**

t\_exec = 14x10^-3 = 5x10^6 \* 3.2 \* Freq^-1 🡪 **Freq = 1.14 GHz**

1. Ara executem amb l’opció “-O”. Calcula MIPS, CPI, Freq i speed-up.

Digues coses iguals i diferents.

# t\_exec = 7ms ; cicles = 8x10^6

```pseudo-asm

%ebx += %eax

%eax += 1

if (%eax - $0xf4240 == 0) jump

inici

```

Nº inst. dinàmiques = 999.999\*(4) + 3 = 3.999.999 🡪 N = 4x10^6

MIPS = (4x10^6)/(10^6 \* 7x10^-3) = **571 = MIPS**

CPI = (8x10^6)/(4x10^6) = **2 = CPI**

t\_exec = 7x10^-3 = 2 \* 4x10^6 \* Freq^-1 🡪 **Freq = 1.14Ghz**

speed-up = (14x10^-3)/(7x10^-3) = **2 = speed-up**

La freqüència ha de continuar sent la mateixa perquè el HW no l’hem canviat.

Respecte el programa, donat que ara triga menys, la quantitat d’instruccions per segon ha augmentat.

El CPI ha disminuït perquè també ho ha fet la quantitat d’instruccions executades i els cicles necessaris.

1. t\_exec del programa complet 200 ms. Calcula speedup màxim si codi anterior instantani.

Calcula speedup amb només la part mencionada amb l’opcio “-O”

# O0 🡪 No optimitzar; -O1 = -O

t\_exec \_noop = 200ms = t\_exec\_resta + t\_exec\_codi1 = t\_exec\_resta + 14x10^-3 🡪 t\_exec\_resta = 200-14 = 186ms

speed-up\_insta = 200/(186+0) = **1.08 = speed-up\_insta**

speed-up\_op = 200/(186+7) = **1.04 = speed-up\_op**

1. Digues “Que”, “Com”, “Que fer” per mesurar MIPS i CPI.

MIPS = N\_inst / t\_exec\*10^6

CPI = cicles/N\_inst

Per saber N\_inst executem “valgrind --tool=lackey ./prog”.

Per saber cicles hem de fer mod. al codi i ficar “getticks()” i “elapsed(t2,t1)”

Per saber t\_exec hem de fer mod. al codi i ficar “GetTime()” i fer la resta.

1. Calcula mitjana aritmètica i geomètrica. Descarta valors extrems i recalcula.

Mitjana\_aritmetica\_normal = (10+8+13+15+2)/5 = 9.6 ms

Mitjana\_geometrica\_normal = (10\*8\*13\*15\*2)^1/5 = 7.58 ms

Mitjana\_aritmetica\_ mod = (10+8+13)/3 = 10.3 ms

Mitjana\_ geometrica \_mod = (10\*8\*13)^1/3 = 10.13 ms

Obs que ha bastant la mitjana calculada. Això es degut a que un dels valors extrems era 2 el que feia baixar bastant la mitjana en el mode normal.