

INTEGRANTES DEL GRUPO 1:

- Oscar Casado Lorenzo
- Wenceslao Diego Pacheco Guevara
- Joaquín Amat Pérez
- Josué Perea Martínez
- Jorge Vázquez López
- Jesús Plaza Ortiz
- Joaquín José Cerdán López



CRYENGINE® 3

ÍNDICE

Capítulo	Contenido	Página
1	Introducción	3
2	Tipos de Benchmark.....	3
2.1	Definición de Programas Reales	3
2.2	Definición de Núcleos(kernels).....	3
2.3	Definición de Benchmarks reducidos (toys).....	4
2.4	Definición de Benchmarks sintéticos.....	5
3	Benchmarks reducidos	7
4	La criba de Eratóstenes en C++.....	7
4.1	Comprobación del funcionamiento de la Criba de Eratóstenes	9
5	La recta de regresión.....	10
5.1	La recta de regresión en C++	11
5.2	La recta de regresión en ensamblador	13
5.3	la recta de regresión en ensamblador con Instrucciones SSE	19
5.4	Comprobación del funcionamiento de la recta de regresión	22
6	SPEC	25
7	Ordenadores del grupo.....	27
8	Resultados y Evaluación	30
8.1	Resultados Benchmark reducidos.....	30
8.2	Resultados SPEC	34
9	Conclusiones.....	35
10	Grafo del desarrollo del proyecto	36
11	Referencias	39

1 - Introducción

En esta fase de prácticas se planea desarrollar una serie de benchmarks reducidos que evalúen el rendimiento de los computadores de los diferentes miembros del grupo. Se crearán 3 benchmarks implementadas de manera diferente (C++, ensamblador y ensamblador con instrucciones SSE) sobre el mismo algoritmo y el objetivo es comparar el tiempo de ejecución que ocupa cada programa. Además, se podrá observar de forma gráfica los resultados.

Por otra parte, se usará un benchmark ya existente llamado SPEC CPU2000, fue producido por SPEC (*Standard Performance Evaluation Corporation*). Contiene dos benchmark suites: CINT2000 para medir y comparar el rendimiento de computación intensiva de enteros, y CFP2000 para medir y comparar el rendimiento de computación intensiva en flotantes.

2 - Tipos de Benchmark

Para poder entender con mayor detalle el proyecto, vamos a definir los distintos tipos de benchmark que existen:

2.1 – Definición de Programas Reales

Lo conforman los compiladores de C, software de tratamiento de texto como TeX y herramientas CAD como Spice. Este tipo de benchmark es el más nuevo respecto a los demás.

2.2 – Definición de Núcleos (Kernels)

Realmente son programas reales pero son pequeños fragmentos clave de programas reales que parametrizan bastante la carga del trabajo a la que somete ese programa real a la CPU. Unos ejemplos de benchmark de este tipo son Livermore Loops o Linpack.

Benchmark Linpack

fue desarrollado en el Argonne National Laboratory por **Jack Dongarra** en 1976, y es uno de los más usados en sistemas científicos y de ingeniería.

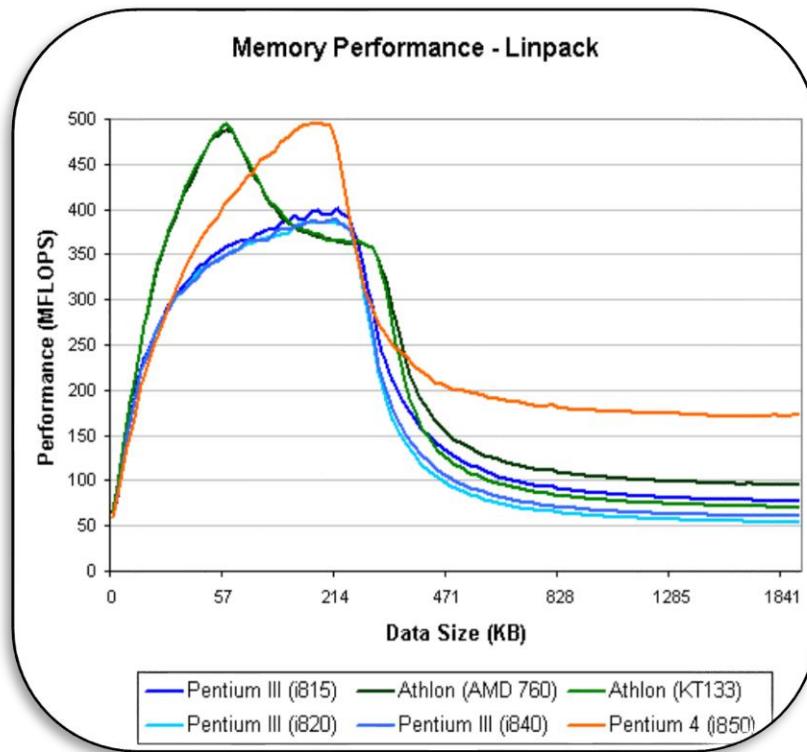
La característica principal de **Linpack** es que hace un uso muy intensivo de las operaciones de coma flotante, por lo que sus resultados son muy dependientes de la capacidad de la **FPU** (*floating point unit*) que tenga el sistema. Además, pasan la mayor parte del tiempo ejecutando unas rutinas llamadas **BLAS** (*Basic Linear Algebra Subroutines* o Subrutinas de Álgebra Lineal Básica).

Linpack **está compuesto por tres benchmarks** de orden 100, orden 1.000, y un tercero de Computación altamente paralela (un algoritmo especialmente diseñado para buscar los beneficios de los multiprocesadores).

El reporte del benchmark describe el rendimiento para **resolver un problema de matrices generales densas** $Ax = b$, a tres niveles de tamaño y oportunidad de optimización: problemas de 100 x 100 (optimización del bucle interno), problemas de 1.000 por 1.000 (optimización de tres bucles - el programa entero) y un problema paralelo escalable.

Los resultados se expresan en **MFlops** (millones de operaciones de coma flotante por segundo), siempre referidas a operaciones de suma y multiplicación de doble precisión (64 bits).

En la siguiente imagen podemos contemplar como es un análisis gráfico de distintos microprocesadores con este benchmark.



2.3 – Definición de Benchmarks reducidos (toys)

Son aquellos que tienen pocas líneas de código (entre 10 y 100 líneas) pero que usan grandes recursos del computador, estos usan algoritmos que calculan la Criba de Eratóstenes, un puzzle, ordenamiento rápido (Quicksort) o la recta de regresión.

A lo largo del proyecto se explicarán 3 ejemplos de benchmark reducidos, todos ellos orientados al mismo problema, la recta de regresión.

2.4 – Definición de Benchmarks Sintéticos

Se crean con el propósito de simular la frecuencia media de operaciones y operados de un gran conjunto de programas. Estos son más específicos, están especialmente diseñados para medir el rendimiento de un componente individual de un ordenador, normalmente llevando el mismo escogido a su máxima capacidad. Unos ejemplos son Dhrystone y Whetstone

Dhrystone

Fue desarrollada por **Reinkol P. Weicker** de **Siemens-Nixdorf Information System** en 1984. Originalmente fue publicado en ADA, aunque hoy en día la mayoría utiliza la versión en C distribuida por **Rick Richardson**.

Este benchmark contiene muchas instrucciones simples, llamadas a procedimientos, condicionales, pocas operaciones de coma flotante y bucles. No realiza llamadas al sistema. Usa pocas variables globales y ejecuta operaciones con punteros. Está compuesto por un **53% de instrucciones de asignación, 32% de instrucciones de control y un 15% de llamadas a procedimiento.**

El Dhrystone compara el rendimiento del procesador usando una máquina de referencia, **la VAX 11/780**, que es la máquina que corre a 1 **MIPS** (*Millones de instrucciones por segundo*). Al ser tan pequeño, el Dhrystone entra completamente en la caché interna, de esta manera no mide el resto del sistema pero presenta la ventaja de que **mide solamente la capacidad del procesador para trabajar con enteros**.

En realidad lo que refleja es más bien el rendimiento del compilador y sus bibliotecas que el procesador en sí mismo. Particularmente Dhrystone contiene dos instrucciones en las cuales el lenguaje de programación y su traducción se llevan el mayor tiempo de ejecución: una asignación de cadena de caracteres y una comparación de cadena de caracteres.

Los resultados de Dhrystone se expresan como el número de finalizaciones de iteración del programa por segundo. Otra representación común del punto de referencia de Dhrystone es el **DMIPS** (Dhrystone MIPS) obtenida cuando el resultado Dhrystone se divide por 1757.

3 – Benchmarks reducidos

Para el desarrollo de los benchmarks reducidos del proyecto, el algoritmo que se ha escogido realizar es la recta de regresión. Pero para facilitar la compresión de un benchmark reducido exemplificaremos su funcionamiento con el algoritmo basado en la criba de Eratóstenes (uno de los más sencillos) para mas tarde entrar en profundidad con la recta de regresión en la cual se basa el proyecto.

4 – La criba de Eratóstenes en C++

La criba de Eratóstenes consiste en hallar todos los números primos menores a un número natural n . Para ello se crea una tabla con todos los números menores de n empezando por el 2.

A continuación, se eliminan todos los números de la tabla que sean múltiplos y por cada iteración si su cuadrado es mayor que n , entonces finaliza el programa, sino prueba con el siguiente número de la tabla.

Este algoritmo nos permite hallar todos los números primos desde 2 hasta N . Para determinar dichos números, se realizan 4 pasos:

- **Paso 1:** Listamos todos los números entre 2 hasta N .
- **Paso 2:** El primer número de la lista se marca como primo.
- **Paso 3:** Todos los múltiplos de este número marcado como primo, se tachan de la lista.
- **Paso 4:** Si el cuadrado del primer número no tachado ni marcado como primo es menor que N , volvemos al paso 2; en caso contrario, el algoritmo termina y la lista que quede es la lista de todos los números primos entre 2 y N .



```
1 #include <iostream>
2 #include <vector>
3 #include <math.h>
4 #include <ctime>
5
6 using namespace std;
7
8 const int n = 100000;
9
10
11 int main() {
12     vector<int> primos;
13
14     int primo;
15     int cont = 0;
16
17     for (int i = 2; i < n; i++) {
18         primos.push_back(i);
19     }
20
21     primo = primos[cont];
22
23     double ini = clock();
24
25     do {
26         for (int i = 0; i < primos.size(); i++) {
27             if (primos[i] % primo == 0 && primos[i] != primo) {
28                 primos.erase(primos.begin() + i);
29             }
30         }
31         cont++;
32         primo = primos[cont];
33     } while (pow(primo, 2) < n);
34     double fin = clock();
35     double time = (double(fin - ini) / ((clock_t) 1000));
36
37     for (int i = 0; i < primos.size(); i++) {
38         cout << primos[i] << " ";
39     }
40
41     cout << "Tiempo empleado: " << time << endl;
42
43     return 0;
44 }
```

```
vector<int> primos;
int primo;
int cont = 0;

for (int i = 2; i < n; i++) {
    primos.push_back(i);
}

primo = primos[cont];
ultimo = primos[cont];
```

El código se basa en un **do_while** que comprende los pasos 2, 3 y 4 del algoritmo. El **primer for** añade todos los números de 2 hasta N-1 como primos y a partir de este vector, se borrarán los que no lo sean. También consideramos el primer número como primo. Usamos un contador llamado **cont** para ir cambiando de número primo seleccionado.

Inicializamos el reloj para calcular el tiempo que tarda (variable **ini**), y ya empieza el **do_while**, en el que siempre se ejecutará el for al menos una vez.

Este for borra los múltiplos del número primo actual (nunca se borrará el número en sí). Incrementamos

el contador para coger el siguiente número (obviamente, no tachado), y repetimos el proceso si el cuadrado de este es menor que N.

Una vez termine el bucle, calculamos el tiempo y mostramos por pantalla la lista de números y el tiempo empleado.

4.1 – Comprobación del funcionamiento de la Criba de Eratóstenes.

Para comprobar el correcto funcionamiento del algoritmo, probaremos a ejecutar el programa, pero hasta 50.

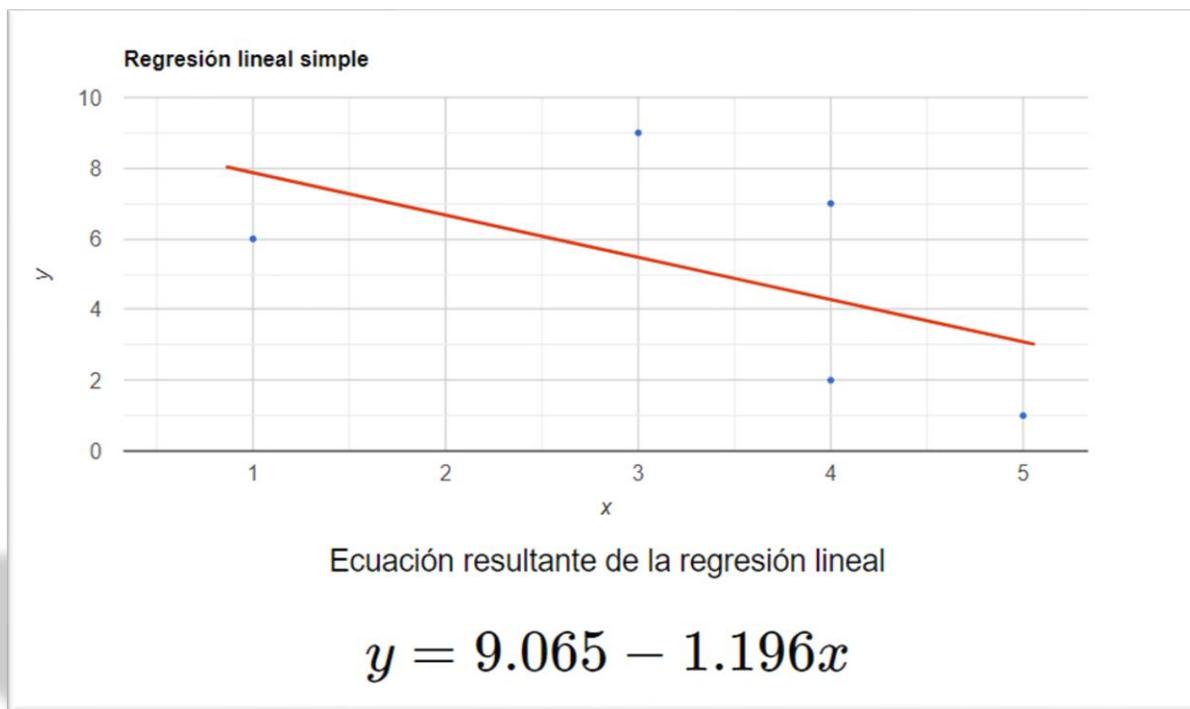
En la siguiente imagen se puede apreciar el resultado de la ejecución y se puede afirmar no solo que funciona correctamente sino que es casi inmediato para un tamaño pequeño.

```
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 Tiempo empleado: 0
```

5 – La recta de regresión

La **regresión lineal simple** es un método estadístico con el que podemos analizar y entender las relaciones existentes entre dos variables cuantitativas. A una de estas variables se le denomina como **variable independiente**, y a la otra se le denomina **variable dependiente**.

En la regresión lineal simple, se pretende predecir los valores de la variable dependiente a partir de los valores de la variable independiente. Para ello, a partir de un **diagrama de dispersión** de los datos se identifica la línea recta que mejor se ajusta a la nube de puntos. A dicha línea se le conoce como **recta de regresión**. A continuación, podemos contemplar una recta de regresión de Y sobre X para los puntos (4,2), (1,6), (3,9), (5,1), (4,7)



Utiliza el procedimiento de **mínimos cuadrados**. Cada punto x_i de la primera variable tendrá, por una parte, el valor correspondiente a la segunda variable y_i , y por otra, su imagen por la recta de regresión $y=mx_i+n$. Entre estos dos valores existirá una diferencia $d_i=mx_i+n-y_i$.

La recta se calcula con la condición de que la suma de los cuadrados de todas estas diferencias $\sum(mx_i+n-y_i)^2$ sea mínima.

El procedimiento que utilizaremos en las 3 implementaciones viene dada por el cálculo de los elementos de la fórmula de la recta de regresión:

$$y - \bar{y} = \frac{\sigma_{xy}}{\sigma_x^2} (x - \bar{x})$$

\bar{y} = Media aritmética de las ordenadas

\bar{x} = Media aritmética de las abscisas

σ_x^2 = Varianza de X

σ_{xy} = Covarianza de X e Y

5.1 – La recta de regresión en C++

Primero definimos en número de puntos, en nuestro caso hemos definido 10000 elementos.

```
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <ctime>
using namespace std;

const int n = 10000;

double rectaC(){
    :
}
```



A continuación, procedemos a **rellenar los puntos con valores aleatorios**, cada valor de x o de y solamente puede tomar valores comprendidos entre 0 y 9. Es muy importante esto ya que, si metiésemos números aleatorios muy grandes, su sumatorio podría desbordar la capacidad del tipo de dato **INT**

Además, declaramos los parámetros que consideramos necesarios para el cálculo de la función (*mediaX*, *mediaY*, Covarianza, Varianza) y una variable **inicio** que indica el tiempo de ejecución.

Calcula los sumatorios de todas las variables, para más tarde emplearlos en la resolución final de las mismas.

```
//Cálculo de medias
for (int i = 0; i < n; i++) {
    //Sumatorio necesario para la mediaX
    mediaX += x[i];
    //Sumatorio necesario para la mediaY
    mediaY += y[i];
    //Sumatorio necesario para la varianzax
    varianzaX += x[i] * x[i];
    //Sumatorio necesario para la covarianzaXY
    covarianzaXY += x[i] * y[i];
}
//Reajustamos las medias
mediaX /= n;
mediaY /= n;

//Cálculo de varianzaX = (Sumatorio(x[i]^2) / n) - mediaX^2
varianzaX = (varianzaX / n) - (mediaX * mediaX);
//Cálculo de la covarianzaXY = (Sumatorio(x[i]*y[i]) / n) - (mediaX * mediaY)
covarianzaXY = (covarianzaXY / n) - (mediaX * mediaY);
```

```
//Creación y relleno de los arrays
int x[n];
int y[n];

srand(time(NULL));
for (int i = 0; i < n; i++) {
    x[i] = rand() % 10;
    y[i] = rand() % 10;
}

//Declaracion de variables
float mediaY, mediaX, covarianzaXY, varianzaX;
//Inicialización
mediaX = mediaY = covarianzaXY = varianzaX = 0.0;

int inicio = clock();
```

Por último, tenemos que calcular la pendiente (**m**) y la ordenada en el origen (**b**) para que la función tenga la forma de $y = mx+n$. Con los datos ya calculados anteriormente simplemente tendríamos que despejar de la fórmula la y , para hallar dichos valores.

```
//FORMULA DE LA RECTA DE REGRESION: y - mediaY = (covarianzaXY / varianzaX)*(x - mediaX)
//                                              y = mx + n
float m = covarianzaXY/varianzaX;
float b = ((covarianzaXY / varianzaX) * (-mediaX)) + mediaY;

int fin = clock();
return (double(fin - inicio) / ((clock_t)1000));
```

5.2 – La recta de regresión en ensamblador

Al igual que la implementación en C++, usaremos la variable global **n** para definir el tamaño de los puntos (*coordX* y *coordY*). El procedimiento inicial es el mismo que el explicado anteriormente.

```
double rectaEnsamblador() {

    //Creación y relleno de los arrays
    int coordX[n];
    int coordY[n];

    srand(time(NULL));
    for (int i = 0; i < n; i++) {
        coordX[i] = rand() % 10;
        coordY[i] = rand() % 10;
    }

    int sumX, sumY, sumXCuadrado, sumXY;
    sumX = sumY = sumXCuadrado = sumXY = 0;

    float auxiliar, mediaX, mediaY, varianzaX, covarianzaXY, m, b;
    auxiliar, mediaX, mediaY, varianzaX, covarianzaXY, m, b = 0.0;

    clock_t inicio = clock();
```

Como hemos visto en la implementación de C++, se requiere de un bucle que acumule los sumatorios (variables de tipo **int**). Y por otro lado reajustar finalmente las mismas (variables de tipo **float**), al presentar formulaciones con denominadores se presenta la problemática de la coma flotante.

Por todo ello decidimos que la solución más sencilla era desglosar dichos cálculos en enteros y decimales como veremos a continuación.

```
_asm {
    inicializar:
        //Arrays de coordenadas
        mov esi, 0;
```

En el registro **esi** guardaremos un contador que se incrementará en el siguiente bucle. El objetivo del registro es utilizarlo como si fuese un índice de un bucle normal por lo que lo inicializamos a 0 con la instrucción **mov**.

Bucle, Suma de Abscisas y ordenadas

```
bucle:
    //contador >= n -> terminar
    cmp esi, 10000;
    jae terminar;

    //mediaX += x[i];
    mov edx, [coordX + 4*esi];
    mov eax, sumX;
    add eax, edx;
    mov sumX, eax;

    //mediaY += y[i];
    mov ebx, [coordY + 4 * esi];
    mov eax, sumY;
    add eax, ebx;
    mov sumY, eax;
```

Para obtener los elementos de la recta de regresión debemos recorrer los vectores definidos anteriormente, y para conseguir obtener un bucle en ensamblador x8086 debemos indicar una etiqueta que denota una dirección de memoria, de manera que si deseamos iterar las siguientes instrucciones tendríamos que usar alguna instrucción de salto e indicar la etiqueta correspondiente.

La etiqueta que se puede apreciar en la imagen de la izquierda es denominada

bucle, además en imágenes posteriores esta definida otra etiqueta llamada **terminar**, el propósito de esta es utilizar un salto condicional dentro del bucle de forma que podamos hacer un **bucle while**.

Es decir, con la etiqueta **cmp** comparamos el registro **esi** (contador) con el valor de **n** (que hemos decidido que es 10000) y al ejecutarse actualiza **EFLAGS**, después con la instrucción de salto condicional **jae** salta a la etiqueta **terminar** si en la última instrucción **cmp** el operando de la izquierda era menor que el de la derecha de esta forma conseguimos **while(esi < 10000)**.



Por otro lado, en la imagen anterior podemos ver como calcula la suma de las abscisas. En la instrucción **mov edx, [coordX + 4*esi]** estamos usando el **modo de direccionamiento en memoria** para recorrer el vector coordX y obtener en cada iteración en el registro edx la componente correspondiente de dicho vector. Accedemos a la posición de memoria que se encuentra nuestro contador, es necesario tener en cuenta para ello que nuestro vector esta compuesto por int, estos ocupan un total de 32 bits o lo que es lo mismo, 4 bytes. Esto produce que los accesos a memoria sean así.

Luego, usamos el registro eax como pivote para obtener la suma, es decir con **mov eax, sumX** obtengo en eax la suma actual, después con **add eax, edx** sumo ambos registros y lo guardo en eax y por último con **mov SumX, eax** actualizo la suma ya que metemos en SumX el valor que tiene el registro eax. Se utiliza la misma estrategia para calcular la suma de las ordenadas, pero usando esta vez el registro **ebx** para guardar los elementos del vector.

Sumatorio de X*X y X*Y

En cuanto a la varianza de X necesitamos calcular el sumatorio de las abscisas al cuadrado, para ello aplicaremos las instrucciones **mul**, **add** y **mov**.

Primero tenemos que guardar en el registro **eax** el valor del vector en cada iteración

```
//varianzaX += x[i] * x[i]
mov eax, edx;
mul edx; //Despues de multiplicar edx se resetea a 0.
mov ecx, sumXCuadrado;
add eax, ecx;
mov sumXCuadrado, eax;

//covarianzaXY += x[i] * y[i]
mov edx, [coordX + 4 * esi];
mov eax, edx;
mul ebx;
mov ecx, sumXY;
add eax, ecx;
mov sumXY, eax;

//Incrementamos el contador
inc esi;

//Siguiente iteracion, salto incondicional
jmp bucle;

terminar:
```

(ya vimos anteriormente que dicho valor queda registrado en edx) así que usamos **mov eax, edx**, después usamos la instrucción **mul edx**, esta instrucción lo que hace realmente es introducir en el registro eax el resultado de multiplicar edx con eax (es decir $\text{coordX[esi]} * \text{coordX[esi]}$).

Cabe destacar que cuando usamos la instrucción **mul**, el registro operando se pone a 0. El resto de instrucciones sirve para controlar el sumatorio, y ya fue explicado anteriormente como utilizar el pivote eax para controlar los sumatorios.

Para el cálculo de la covarianza necesitaremos el sumatorio de multiplicar cada abscisa con su ordenada correspondiente (es decir $\text{coordX[esi]} * \text{coordY[esi]}$).

Para desarrollar este cálculo tenemos que introducir otra vez en el registro edx el valor de coordX[esi] , por eso utilizamos **mov edx, [coordX + 4*esi]**. A continuación, guardamos en eax el valor que contiene el registro edx con la instrucción **mov eax, edx**, luego se utiliza la instrucción **mul ebx** (en ebx queda registrado el valor de coordY[esi]) para obtener en eax la multiplicación deseada. Y al igual que antes el sumatorio ya fue explicado así que se omite la reiteración.

Por último, es posible observar como el registro esi se autoincrementa con la instrucción **inc**, y además hay un salto incondicional **jmp** que retorna la ejecución del programa al principio del bucle.

Media, Varianza, Covarianza, m, b

Una vez obtenidos todos los sumatorios debemos calcular las variables finales que nos llevarán a la resolución final de la nube de puntos. Las operaciones necesarias para esto producirán decimales, que de omitirlos influirían gravemente en el resultado de nuestra implementación.

Por todo ello ahora entra en juego la **FPU** junto a todo su set de instrucciones característico. Como se muestra en la captura el procedimiento que seguiremos para llevar a cabo cálculos de esta tipología es necesario comprender el comportamiento del '**heap**'. Las instrucciones que usaremos en esta práctica las podríamos clasificar informalmente como:

- **Carga:** **fild**, carga una variable entera de memoria en el tope de la pila, también llamado **STO** tras realizar una conversión a **float**. Existe una variable muy similar (**fld**) basada en cargar en STO un número que sea float directamente.
- **Modificación:** Aplican la operación indicada (división '**fdiv**', multiplicacion '**fmul**', suma '**fadd**') al tope de la pila junto al argumento de la misma, que suele ser una variable en memoria. El resultado sobrescribe al STO.
- **Descarga:** **fstp**, Guardan en una variable el contenido de **STO**.

Medias Aritméticas

```
//Coloca un numero entero (que pasa a float) en el tope de la pila
fild sumX;
//Divide STO (tope pila) pasando a float entre 'n'
fidiv n;
//Mueve el contenido de STO a una posicion de memoria
fstp mediaX;

fild sumY;
fidiv n;
fstp mediaY;
```

Varianza X

```
//VARIANZA = (sumXCuadrado / n) - (mediaX*mediaX)
fld sumXCuadrado;
fidiv n;
fstp varianzaX;

//auxiliar = (mediaX*mediaX)
fld mediaX;
fmul mediaX;
fstp auxiliar;

fld varianzaX;
fsub auxiliar;
fstp varianzaX;
```

¿Qué es la varianza?
¿Qué es la covarianza?

Covarianza XY

```
//COVARIANZA = (sumXY / n) - (mediaX*mediaY)
fld sumXY;
fidiv n;
fstp covarianzaXY;

//auxiliar = (mediaX*mediaY)
fld mediaX;
fmul mediaY;
fstp auxiliar;

fld covarianzaXY;
fsub auxiliar;
fstp covarianzaXY;
```

Pendiente (m) y constante (b)

```
//m = covarianzaXY/varianzaX
fld covarianzaXY;
fdiv varianzaX;
fstp m;

//b = ((covarianzaXY / varianzaX)* (-mediaX)) + mediaY
fld b;
fsub mediaX;
fmul covarianzaXY;
fdiv varianzaX;
fadd mediaY;
fstp b;
}

clock_t fin = clock();

return (double(fin - inicio) / ((clock_t)1000));
}
```

5.3 – La recta de regresión en ensamblador con instrucciones SSE

Esta implementación en ensamblador difiere en la anterior debido a que haremos uso de instrucciones SSE, estas instrucciones están orientadas al paradigma **SIMD** (*Simple Instruction Multiple Date*), es decir a ejecutar varias instrucciones en un bloque.

El código inicial es prácticamente igual al anterior, solamente que ahora usaremos vectores dinámicos (*int) para almacenar los puntos y calcularemos el número de iteraciones a realizar dividiendo el número de puntos entre 4.



```
double rectasse() {
    int *coordX = new int[n];
    int *coordY = new int[n];

    srand(time(NULL));
    for (int i = 0; i < n; i++) {
        coordX[i] = rand() % 10;
        coordY[i] = rand() % 10;
    }

    int sumX, sumY, sumXCuadrado, sumXY;
    sumX = sumY = sumXCuadrado = sumXY = 0;

    float auxiliar, mediaX, mediaY, varianzaX, covarianzaXY, m, b;
    auxiliar = mediaX = mediaY = varianzaX = covarianzaXY = m = b = 0.0;

    int ultimaIter = n / 4;

    clock_t inicio = clock();
```

```
_asm {
    inicializar:
        mov ebx, 0;
        mov edx, 0;
        mov ecx, 0;
        mov eax, 0;
        mov esi, coordX;
        mov edi, coordY;
```

a)

```
bucle:
    cmp ecx, ultimaIter;
    jae terminar;

    //Registro sumX
    movdqu xmm0, [esi];
    paddd xmm2, xmm0;

    //Registro sumY
    movdqu xmm1, [edi];
    paddd xmm3, xmm1;
```

```
//Registro sumXCuadrado
pmulld xmm0, xmm0;
paddl xmm4, xmm0;

movdqu xmm0, [esi]; //REINICIALIZAR
//Registro sumXY
pmulld xmm1, xmm0;
paddl xmm5, xmm1;

add esi, 16;
add edi, 16;

inc ecx;
jmp bucle;
```



```

terminar:
    //sumatorioX
    phaddd xmm2, xmm2;
    phaddd xmm2, xmm2;
    movdqu sumX, xmm2;
    mov ebx, sumX;

    //sumatorioY
    phaddd xmm3, xmm3;
    phaddd xmm3, xmm3;
    movdqu sumY, xmm3;
    mov edx, sumY;

```

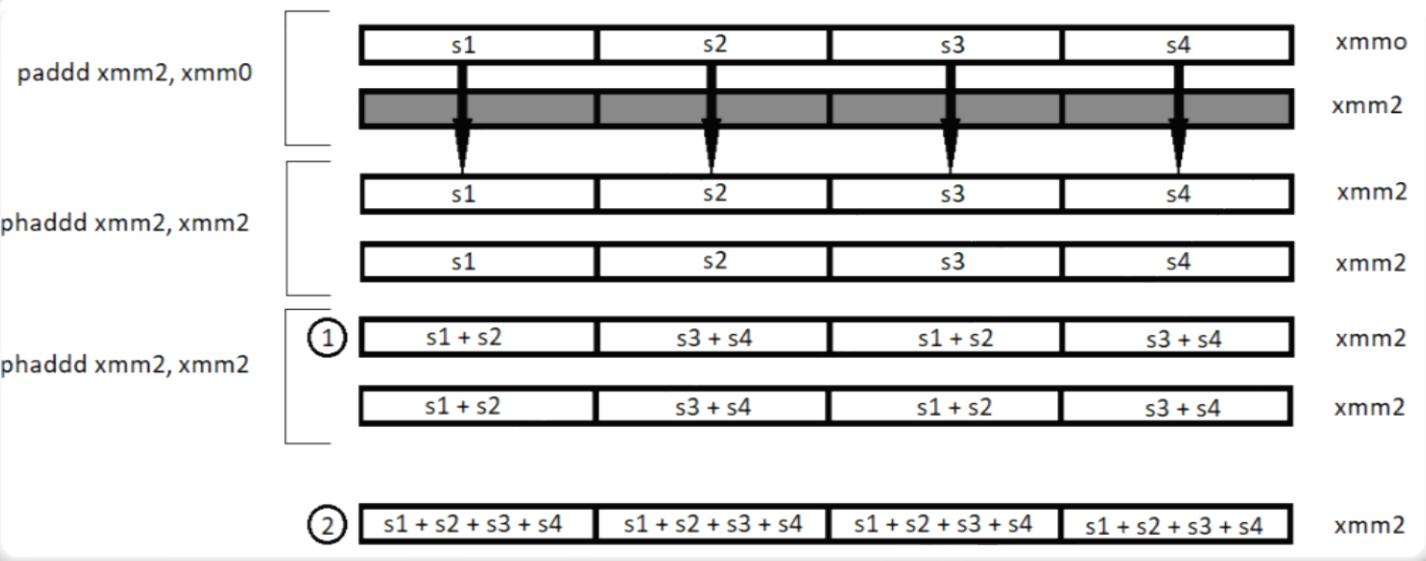
```

//SumatorioCuadrado
phaddd xmm4, xmm4;
phaddd xmm4, xmm4;
movdqu sumXCuadrado, xmm4;
mov eax, sumXCuadrado;

//SumatorioXY
phaddd xmm5, xmm5;
phaddd xmm5, xmm5;
movdqu sumXY, xmm5;
mov ecx, sumXY;

mov sumX, ebx;
mov sumY, edx;
mov sumXCuadrado, eax;
mov sumXY, ecx;

```



Este procedimiento se aplica en el cálculo del sumatorio de X, de Y, de X*Y y de X*X. Primero, en la etiqueta "**bucle**", sumamos los registros **xmm0** y **xmm2** mediante la instrucción **padd**, al no inicializar el registro xmm2 se queda a 0 y se realiza una copia de xmm0.

Tras haber hecho la copia, en la etiqueta "**terminar**", realizamos los sumatorios mediante la repetición de la instrucción **phadd**. Al pasarle como parámetros el registro xmm2, se suman sus elementos y como podemos ver en el **índice 1**, tenemos el mismo valor en los elementos 1,3 y 2,4.



Volvemos realizar la instrucción **phaddd** con xmm2 como parámetros y todos los elementos del registro contienen el mismo valor, el sumatorio, como se aprecia en el **índice 2**.

En cuanto al resto del código, utilizamos el mismo método que en el apartado anterior para terminar de calcular los valores (refiriéndome al uso del **FPU**)

5.4 – Comprobación del funcionamiento de la recta de regresión

Para poder comprobar que los algoritmos funcionan correctamente he modificado partes del código como el número de puntos a 8 y el tipo de dato que devuelve cada función para visualizar el resultado.

Cambios en el código

```
const int n = 8;  
  
void rectaC(){  
}
```

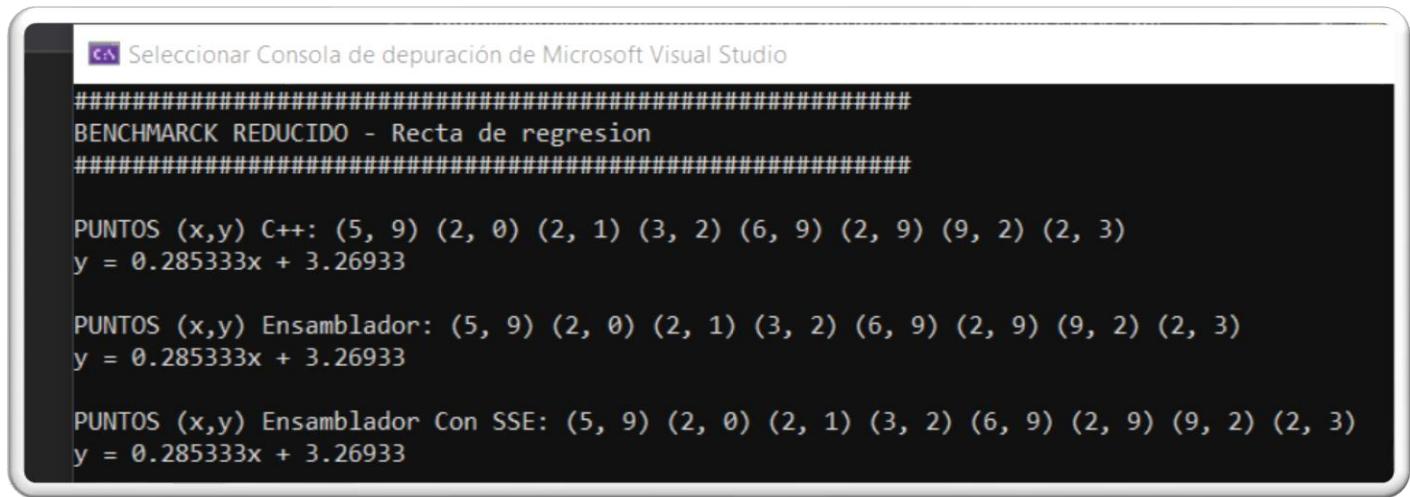
```
//double rectaEnsamblador() {  
void rectaEnsamblador() {  
}
```

```
//double rectaSSE() {  
void rectaSSE() {  
}
```

El siguiente código se repite en las 3 implementaciones

```
cout << endl << "PUNTOS (x,y) C++: ";
for (int i = 0; i < n; i++) {
    cout << "(" << x[i] << ", " << y[i] << ")";
}
cout << endl;
cout << "y = " << m << "x + " << b << endl;
//return (double(fin - inicio) / ((clock_t)1000));
```

Si ejecutamos el programa nos da el siguiente resultado:



```
Seleccionar Consola de depuración de Microsoft Visual Studio
#####
BENCHMARCK REDUCIDO - Recta de regresion
#####

PUNTOS (x,y) C++: (5, 9) (2, 0) (2, 1) (3, 2) (6, 9) (2, 9) (9, 2) (2, 3)
y = 0.285333x + 3.26933

PUNTOS (x,y) Ensamblador: (5, 9) (2, 0) (2, 1) (3, 2) (6, 9) (2, 9) (9, 2) (2, 3)
y = 0.285333x + 3.26933

PUNTOS (x,y) Ensamblador Con SSE: (5, 9) (2, 0) (2, 1) (3, 2) (6, 9) (2, 9) (9, 2) (2, 3)
y = 0.285333x + 3.26933
```

Podría existir la posibilidad de una pérdida de datos debido a que en SSE agrupamos los puntos en **paquetes de 4**. Sin embargo, al haber escogido como muestra **n = 10000** que es múltiplo de 4 no se producen errores.

De todas formas, en el peor de los casos estaríamos perdiendo un total de 3 puntos que comparados con el total de muestra, y sumado a que los valores no difieren en gran medida unos de otros. Al llevar a cabo los cálculos finales no afectaría en gran medida al resultado.



	X	Y	var
1	5	9	
2	2	0	
3	2	1	
4	3	2	
5	6	9	
6	2	9	
7	9	2	
8	2	3	
9			

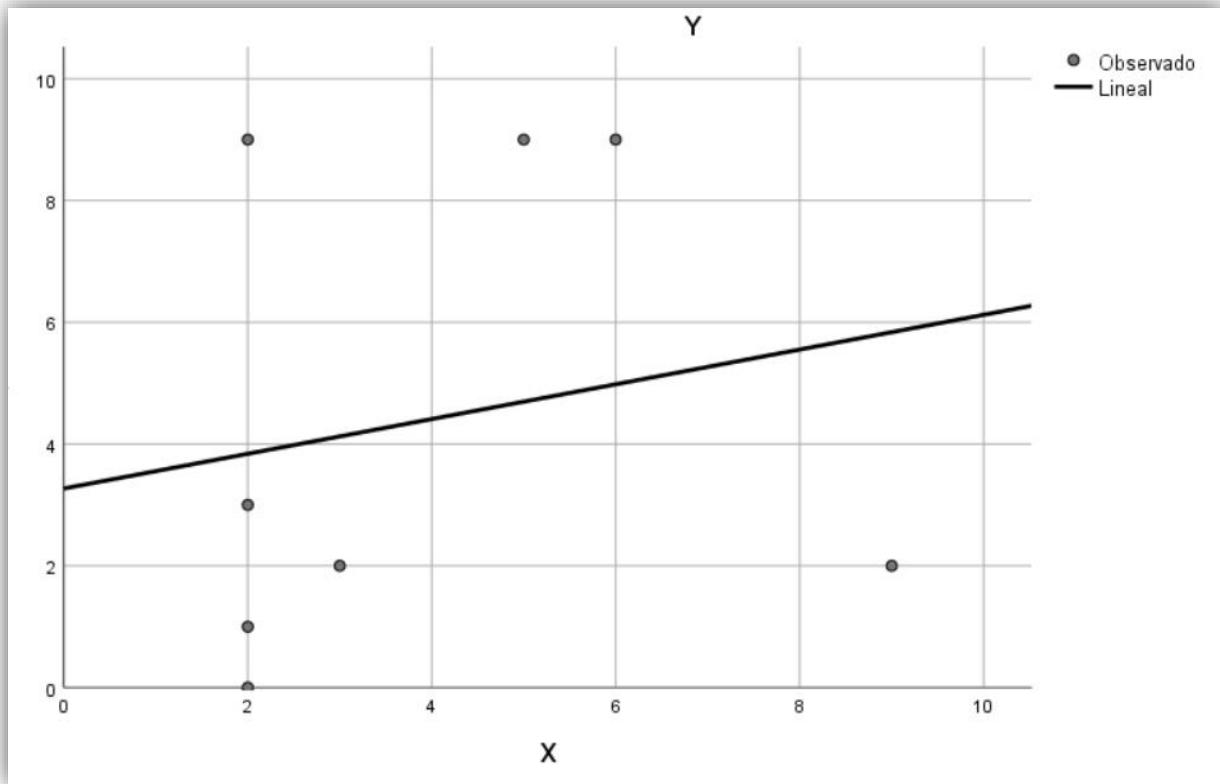
Insertando los puntos en un programa estadístico llamado SPSS (*Statistical Package for the Social Sciences*) puedo obtener no solo la función que representa la recta de regresión, sino que también la gráfica.

Resumen de modelo y estimaciones de parámetro						
Variable dependiente: Y						
Ecuación	R cuadrado	Resumen del modelo				Estimaciones de parámetro
		F	gl1	gl2	Sig.	
Lineal	,035	,220	1	6	,656	Constante b1
La variable independiente es X.						

Observando el resumen que nos proporciona **SPSS** nos fijamos en las dos últimas columnas, donde la **constante** hace referencia a la letra "**b**" y **b1** hace referencia a "**m**", por lo que la recta sería $y = 0,285x + 3,269$.

Con dichos resultados ya podemos confirmar que los algoritmos tienen un correcto funcionamiento.

La gráfica de dicha recta de regresión es la siguiente:



7 – SPEC

En el apartado de **Introducción** ya se definió SPEC, pero en este apartado entraremos en detalle y señalaremos los resultados que se han generado al ejecutar el benchmark en los computadores de todo el equipo.

La "C" en **CINT2000** y **CFP2000** denotan que son benchmarks a nivel de componentes, en oposición a benchmarks de todo el sistema.

El **objetivo** de los benchmarks de CPU de SPEC es proveer una medida de rendimiento para comparar sistemas sobre la base de una carga de trabajo intensiva en cómputo bien conocida, con énfasis en la capacidad del procesador del sistema, la jerarquía de memoria y el compilador.

CINT2000 contiene 11 aplicaciones escritas en C y una en C++ (252.eon) que son usadas como benchmarks.



- **164.gzip** Utilidad de compresión de datos.
- **175.vpr** Direcciónamiento y ubicación de circuitos FPGA (matriz de puertas lógicas programable en campo).
- **176.gcc** Compilador C.
- **181.mcf** Resolutor de costo mínimo de flujo de red.
- **186.crafty** Programa de ajedrez.
- **197.parser** Procesamiento de lenguaje natural.
- **252.eon** Efectos producidos por distintas fuentes de luz.
- **253.perlbench** Perl (es un lenguaje de programación, está basado en un estilo de bloques como los del C o AWK)
- **254.gap** Teoría de grupo computacional.
- **255.vortex** Base de datos orientada a objetos.
- **256.bzip2** Utilidad de compresión de datos.
- **300.twolf** Simulador de ubicación y ruteo.

La siguiente tabla representa la recopilación de los resultados obtenidos al ejecutar el benchmark.

	Ximo	Joaquín	Josué	Jorge	Oscar	Diego	Jesús
164.gzip	80,6	70,5	77	98,2	67,1	135	56,4
175.vpr	87,5	47,5	52,9	68,1	44,1	80	40,1
176.gcc	39,4	23,7	27,6	19,4	21,8	34,4	21,6
181.mcf	50,4	30,6	32,9	22	26,4	77,8	18,4
186.crafty	46	27,2	30,8	20,8	25,8	38,4	23,5
197.parser	128	66,8	75,9	52	63,8	105	53,1
252.eon	39	28,7	29,8	20,3	25	34,7	22,2
254.gap	46,9	33,6	28,1	20,4	25,2	40,2	23,7
255.vortex	64,9	45,7	36,5	27,6	35,6	67,7	33,1
256.bzip2	96,4	55	55	39,9	48,9	83,3	42,7
300.twolf	153	98,3	83,7	66,7	79,2	162	62,4

En el apartado de **Resultados y Evaluación** se profundiza el análisis de los resultados.

9 – Ordenadores del grupo

En este apartado indicaremos los ordenadores personales de cada miembro, ya que en el siguiente apartado hablaremos de los resultados y es importante saber los aspectos importantes de las máquinas con los que se ejecutó los distintos benchmark.

Joaquín José Cerdán López

Precio: 900 euros

Año de compra: 2019

Tipología: Portátil

Componentes:

- Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz 2.00 GHz
- 8GB de memoria RAM
- 8MB de memoria caché
- Intel (R) UHD Graphics 620
- SSD 512 GB



Jorge Vázquez López



Precio: 1000 euros

Año de compra: 2019

Tipología: Portátil

Componentes:

- Intel(R) i7-8750H CPU @ 2.20GHz 4.10GHz
- 16GB de memoria RAM
- 9MB de memoria caché
- NVIDIA GeForce GTX 1060 GDDR5 @ 6.0 GB
- SSD M.2 128 GB
- Disco Duro 1TB HDD

a

Josué Perea Martínez



Precio: 929€ euros

Año de compra: 2016

Tipología: Portátil

Componentes:

- Intel(R) Core(TM) i7-6500U CPU @ 2.50GHz 2.60GHz
- 16GB de memoria RAM
- 4MB de memoria caché
- NVIDIA GeForce GTX 950M @ 4.0GB
- SSD 128 GB
- Disco Duro 1TB

Oscar Casado Lorenzo

Precio: 1000 euros

Año de compra: 2018

Tipología: Portátil

Componentes:

- Intel(R) i7-7700HQ CPU @ 2.80GHz 3.81GHz
- 16GB de memoria RAM
- 6MB de memoria caché
- NVIDIA GeForce GTX 1050 GDDRS @ 4.0 GB
- SSD 128 GB
- Disco Duro 1TB





Joaquín Amat Pérez



Precio: 700 euros

Año de compra: 2018

Tipología: Portátil

Componentes:

- Intel(R) Core(TM) i7-7500U CPU @ 2.70GHz 2.90 GHz
- 8GB de memoria RAM
- 4MB de memoria caché
- Intel (R) HD Graphics 620
- Disco Duro 1TB

Wenceslao Diego Pacheco Guevara



Precio: 1050 euros

Año de compra: 2018

Tipología: Portátil

Componentes:

- Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz 2.81 GHz
- 8GB de memoria RAM
- 6MB de memoria caché
- NVIDIA GeForce GTX 1050 GDDRS @ 4.0 GB
- Disco Duro 1TB

Jesús Plaza Ortiz



Precio: 1150

Año de compra: 2018

Tipología: ordenador de sobremesa

Componentes:

- AMD Ryzen 5 2600 Six-Core Processor @ 3.40 GHz
- 16GB de memoria RAM
- 16MB de memoria caché
- AMD RX 580 GDDR5 @ 8.0 GB
- Disco duro 1TB
- SSD 256GB

8 – Resultados y Evaluación

En este apartado indicaremos y analizaremos los resultados obtenidos a lo largo del proyecto. Por otra parte, mostraremos gráficamente la comparación de dichos resultados.

8.1 – Resultados Benchmark reducidos

Nuestro equipo ha desarrollado un único fichero denominado regresión.cpp en el que se muestra el tiempo de ejecución de los 3 benchmark, el número de puntos utilizado en todos ellos es de 10000.

El programa en realidad realiza dos operaciones, la primera es ejecutar los 3 benchmark (recta de regresión en c++, ensamblador y ensamblador con SSE) **1000 veces** e ir sumando cada vez los tiempos de ejecución y la segunda operación es lo mismo, pero ejecutándolo **3000 veces**.

Si ponemos en marcha el programa obtenemos esto:

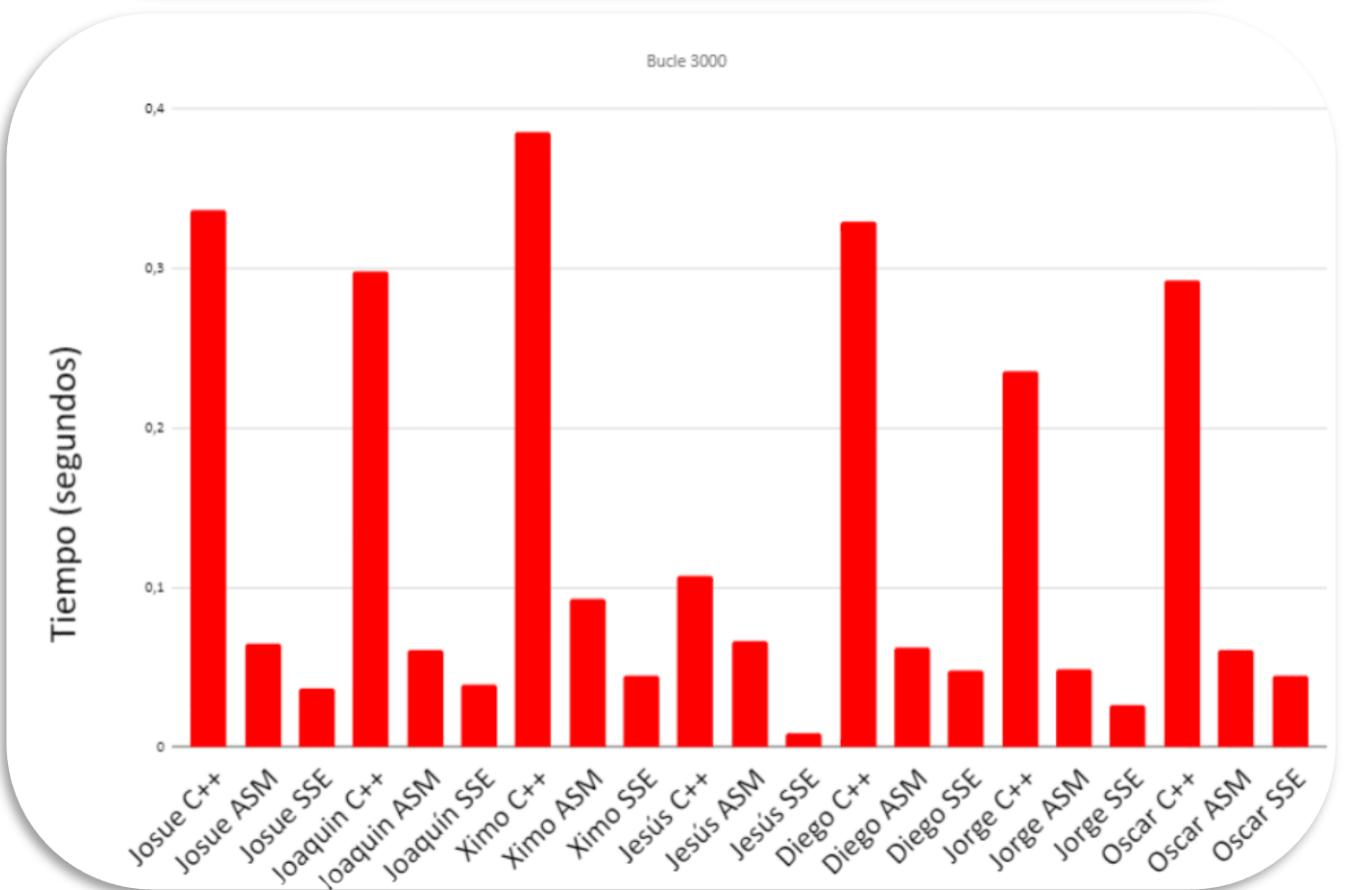
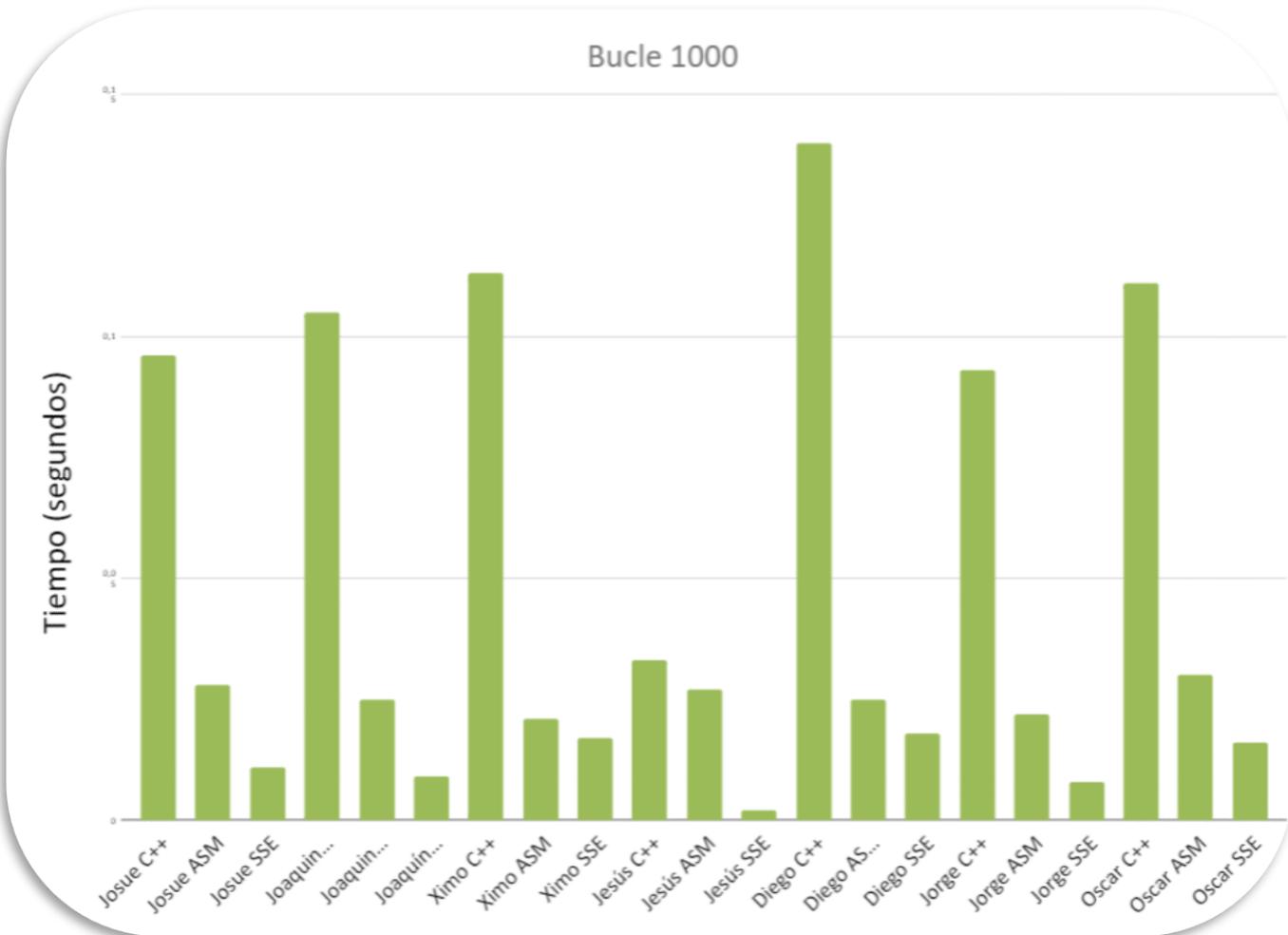
```
Consola de depuración de Microsoft Visual Studio
#####
BENCHMARCK REDUCIDO - Recta de regresion
#####
1a PRUEBA Nube aleatoria de puntos (1000 iteraciones)
EJECUCION: [=====]

-----
2a PRUEBA Nube aleatoria de puntos (3000 iteraciones)
EJECUCION: [=====]
#####
> TABLA DE TIEMPOS OBTENIDA
    | Tiempo C++ | Tiempo Ensamblador | Tiempo SSE
---|---|---|---
1000 it.| 0.106 | 0.023 | 0.01
-----|-----|-----|
3000 it.| 0.307 | 0.055 | 0.023
#####
CryEngine UA ®:
    Joaquin Jose Cerdan Lopez (DIRECTOR)
    Josue Perea Martinez
    Oscar Casado Lorenzo
    Jorge Vazquez Lopez
    Diego Wenceslao Pacheco Guevara
    Joaquin Amat Perez
    Jesus Plaza Ortiz
```

A continuación, se puede observar la recopilación de tiempos.

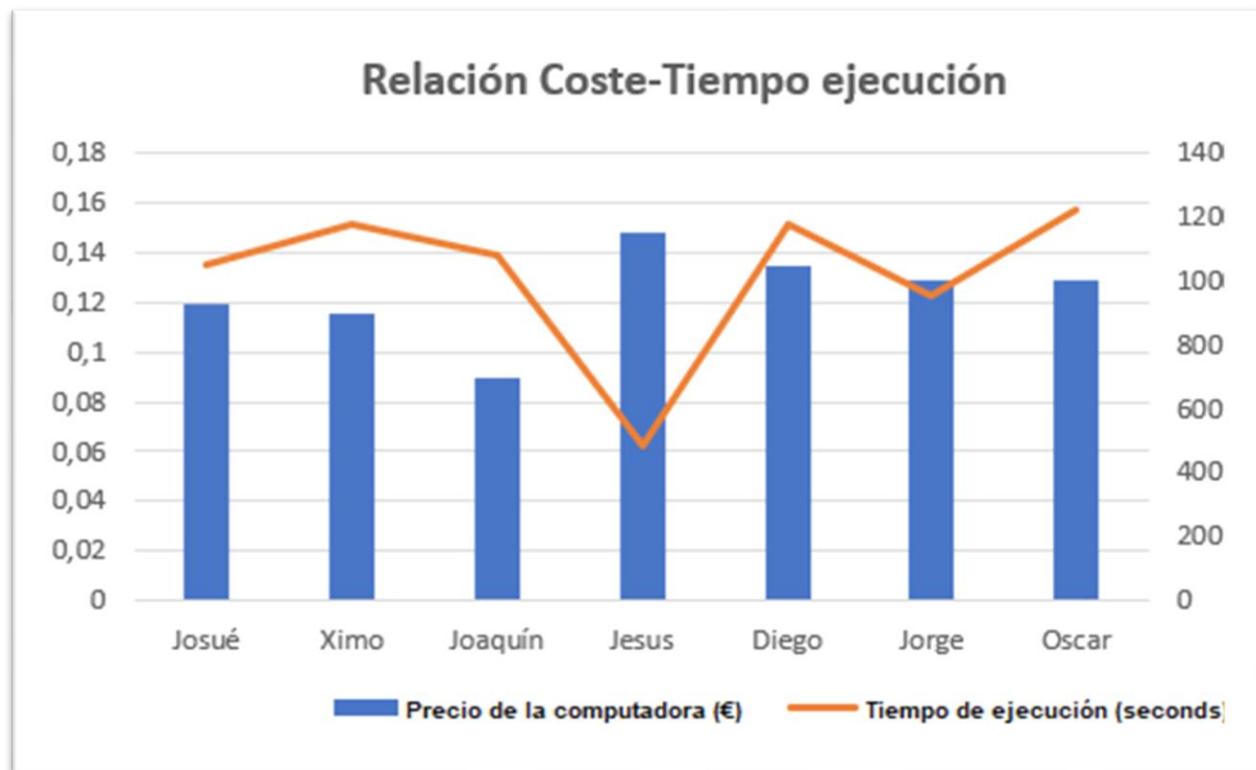
Ordenador	Máquina	Tejec (bucle = 1000) n = 10000	Tejec (bucle = 3000) n = 10000
Josué	Josue C++	0,096	0,336
	Josue ASM	0,028	0,065
	Josue SSE	0,011	0,037
Joaquín	Joaquin C++	0,105	0,298
	Joaquin ASM	0,025	0,061
	Joaquín SSE	0,009	0,039
Ximo	Ximo C++	0,113	0,385
	Ximo ASM	0,021	0,093
	Ximo SSE	0,017	0,045
Jesús	Jesús C++	0,033	0,107
	Jesús ASM	0,027	0,066
	Jesús SSE	0,002	0,009
Diego	Diego C++	0,14	0,329
	Diego ASM	0,025	0,062
	Diego SSE	0,018	0,048
Jorge	Jorge C++	0,093	0,235
	Jorge ASM	0,022	0,049
	Jorge SSE	0,008	0,026
Oscar	Oscar C++	0,111	0,292
	Oscar ASM	0,03	0,061
	Oscar SSE	0,016	0,045

aa



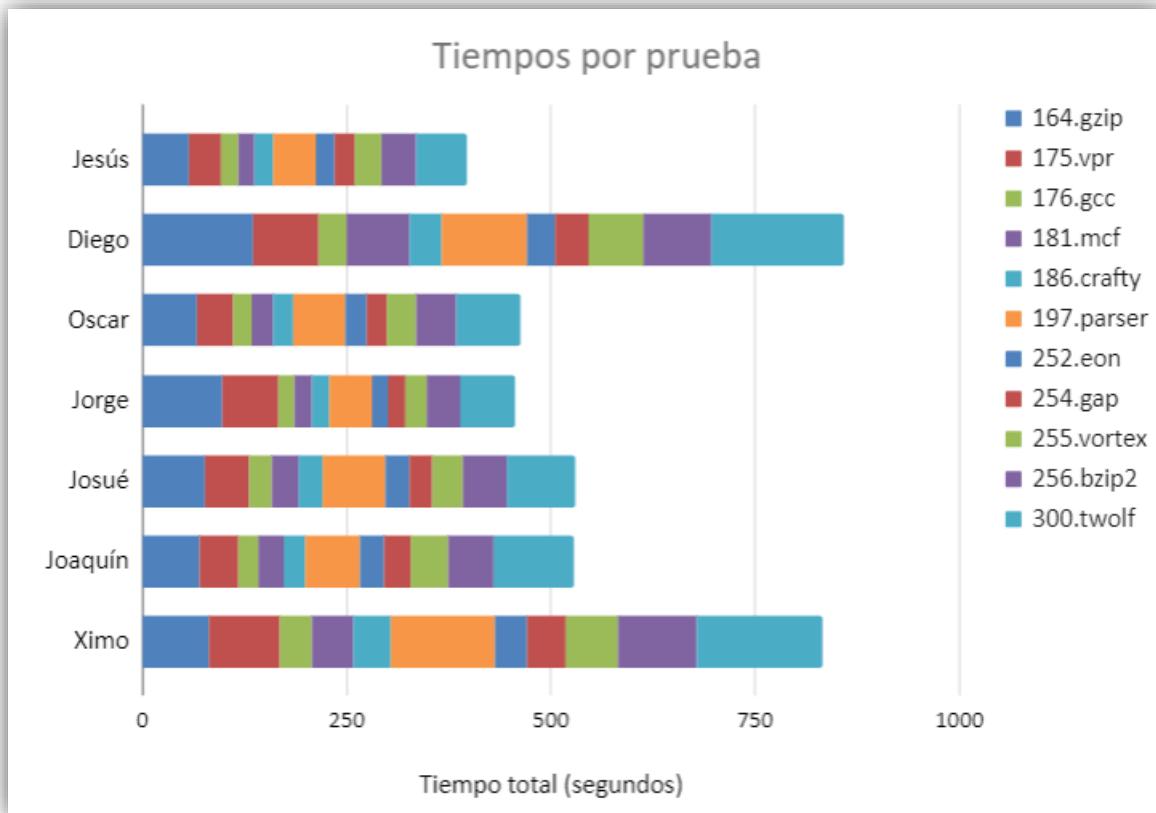
En las dos gráficas anteriores se puede apreciar como la implementación en diferentes lenguajes (C++ y Ensamblador) es increíblemente diferente en cuanto a tiempo de ejecución se refiere. Además, aplicando las instrucciones SSE en ensamblador se puede observar como el tiempo de ejecución es más rápido debido a que estamos ejecutando un conjunto de instrucciones como si fuesen 1 sola (**Cabe destacar que Ximo es Joaquín José Cerdán López**).

En la siguiente imagen podemos observar una relación entre el precio del computador y el tiempo medio de ejecución de los benchmark. En el lado izquierdo tenemos el tiempo y en el derecho el precio expresado en euros (falta un 0 al final de cada fila, en la columna de precio).

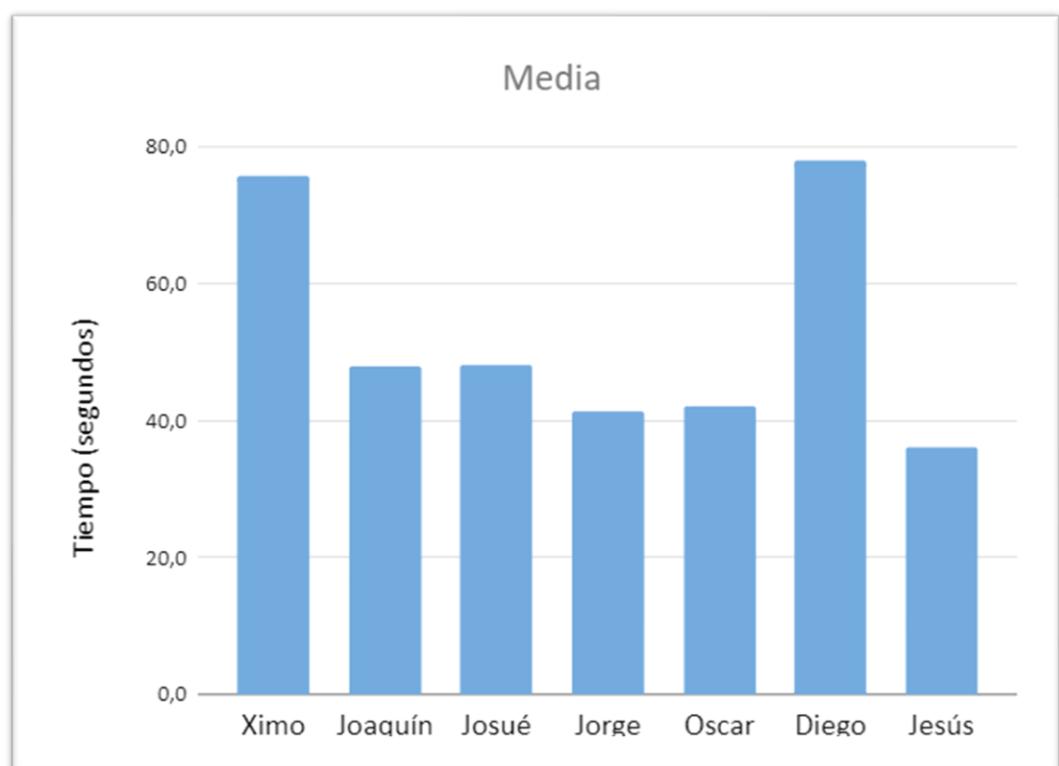


Realmente no podemos sacar una conclusión absoluta a partir de esta gráfica ya que el precio de los ordenadores tienen muy poco margen entre uno y otro, pero es bastante interesante ver los resultados de esta manera. Cabe destacar que pueden influir aspectos externos, como por ejemplo a Jorge que dice tener el ventilador defectuoso.

8.2 – Resultados SPEC



La gráfica anterior representa los datos recopilados en el apartado de **SPEC**, puede ser un poco confuso ver los resultados en esta forma, así que hemos decidido representar una media de todas las pruebas de la siguiente manera:



En la gráfica obtenemos al igual que en los benchmark reducidos que Jesús tiene el computador más rápido. Es normal obtener resultados diferentes de rendimiento ya que es otro tipo de benchmark (es **sintético**).

9 – Conclusiones

- **Equipos para las pruebas:** En nuestro caso, todos los integrantes del grupo tenemos computadores bastante semejantes. Esto puede ser un problema a la hora de evaluar los resultados de las pruebas ya que no veremos una gran diferencia entre los resultados. Debido a esto evaluaremos el rendimiento de forma más estricta y precisa.
- **Erastótenes vs recta de regresión:** A la hora de traducir el código de C++ a ensamblador o SSE, podemos observar que la recta de regresión es más rápida y eficiente. Es por eso que hemos decidido seguir el benchmark con dicho algoritmo.
- **C++ vs ASM:** Viendo las gráficas de las pruebas realizadas para C++ y ASM, podemos afirmar que ASM emplea menos tiempo en sacar los resultados, aproximadamente una cuarta parte de lo que tarda C++ (C++ puede tardar más porque es un lenguaje de alto nivel).
- **ASM vs SSE:** igualmente que para C++, al pasar las pruebas y analizar los resultados obtenidos, podemos afirmar que SSE mejora incluso el tiempo obtenido para ASM. Esto surge debido a que estamos usando el paradigma **SIMD**.

RESULTADOS DE NUESTROS PC (RENDIMIENTO/PRECIO):

MEJOR PC INTEL:

- Intel(R) i7-8750H CPU @ 2.20GHz 4.10GHz
- 16GB de memoria RAM
- 9MB de memoria caché
- NVIDIA GeForce GTX 1060 GDDR5 @ 6.0 GB
- SSD M.2 128 GB
- Disco Duro 1TB HDD

Si comparamos este computador con el resto, este es más potente, a pesar de que todos tienen prestaciones muy semejantes.

Como velocidad más de procesamiento de datos alcanza hasta 4.10 GHz y el precio total del equipo es de 1000€. Este i7-8750H es de octava generación, al igual que el i7-8550U (otro pc que hemos empleado para las pruebas), sin embargo, este es H el cual prioriza en potencia.

Por otro lado, el i7-8550U prioriza en duración y economía del tiempo activo del microprocesador. Su velocidad alcanza los 2.00 GHz y el precio total del equipo es de 900€. Es razonable pensar que obtenemos mejores resultados en un procesador tipo H (0.093s) que en un tipo U (0.113s).

AMD:

- AMD Ryzen 5 2600 Six-Core Processor @ 3.40 GHz

Este es el procesador que mejores resultados ha mostrado. Es un procesador de computador de mesa, todos los demás equipos son portátiles. Los tiempos que ha presentado este procesador son realmente bajos respecto a los demás, mejorando en c++ hasta los 0.033s. El precio total del equipo es de 1150€.

10 – Grafo del desarrollo del proyecto

El grafo que se puede apreciar más abajo es un grafo **PERT** (*Project Evaluation Research Task*), el propósito de este grafo es elaborar una guía del proyecto. Es decir, sirve para planificar el transcurso del proyecto de manera que se pueda ver en cada arco (flecha) el tiempo requerido y en cada nodo una actividad a realizar.

Que dos nodos estén unidos por un arco denota que si uno de los nodos esta en la parte en la que el arco es saliente significa que "precede" al nodo adyacente a este.

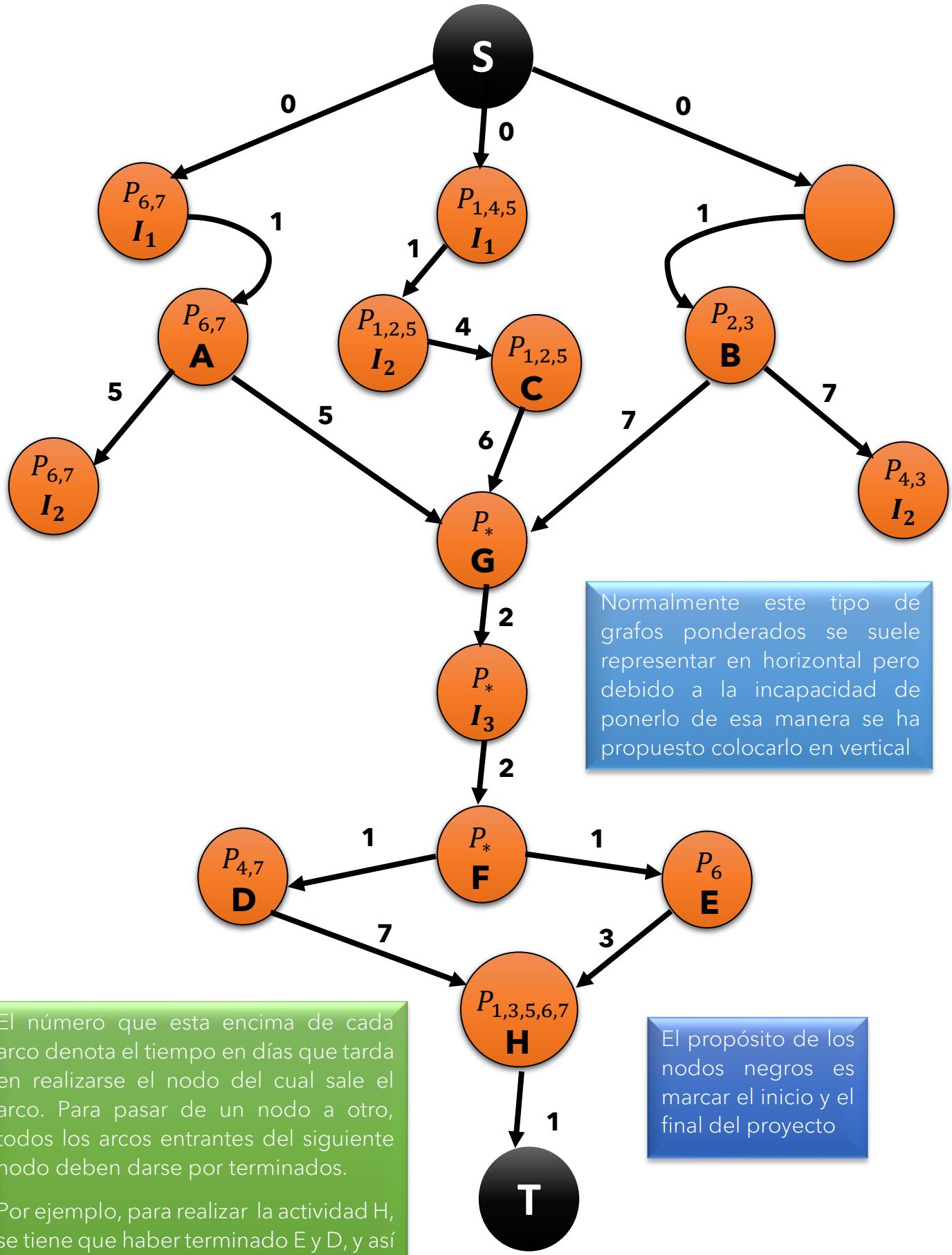
Para poder comprender el grafo debemos declarar una serie de parámetros.

Actividades

- **I₁** = Parte Individual 1.
- **I₂** = Parte Individual 2.
- **I₃** = Parte Individual 3.
- **A** = Implementación del Benchmark reducido en C++.
- **B** = Implementación del Benchmark reducido en Ensamblador x8086
- **C** = Implementación de Benchmark reducido en Ensamblador x8086 con instrucciones SSE.
- **D** = Redacción del trabajo en grupo (memoria).
- **E** = Elaboración de una presentación en powerpoint.
- **F** = Ejecución de SPEC.
- **G** = Recolección de Información de los distintos benchmark elaborados (carpeta drive).
- **H** = Presentación en clase de prácticas del proyecto.

Miembros del grupo

- **P₁** = Joaquín José Cerdán López
- **P₂** = Oscar Casado Lorenzo
- **P₃** = Jorge Vázquez López
- **P₄** = Wenceslao Diego Pacheco Guevara
- **P₅** = Jesús Plaza Ortiz
- **P₆** = Joaquín Amat Pérez
- **P₇** = Josué Perea Martínez
- **P_{*}** = Todos los miembros del grupo



11 – Referencias

Dpto. Tecnología Informática y Computación (2020). Guía Ensamblador x86 , p. 2-3, p. 18. UACloud.

Universidad Tecnológica de la Mixteca, Oaxaca. Capítulo 6. Punto flotante, p. 10-13.
<https://www.utm.mx/~jjf/le/TEMA6.pdf>

Anónimo (2020). Dhystone. Wikipedia.
<https://es.wikipedia.org/wiki/Dhystone>

Anónimo (2020). Standard Performance Evaluation Corporation. Wikipedia.
https://es.wikipedia.org/wiki/Standard_Performance_Evaluation_Corporation

Intel Corporation. 64 and IA-32 Architectures Software Developer's Manual Combined Volumes. Developer zone
<https://software.intel.com/en-us/download/intel-64-and-ia-32-architectures-sdm-combined-volumes-1-2a-2b-2c-2d-3a-3b-3c-3d-and-4>

Departamento de Ciencia de la Computación e Inteligencia Artificial (2019). Gráficos PERT, pp. 23-25. UACloud.

Anónimo (2020). Calculadora de Regresión lineal simple - Ecuación de regresión y gráfica. calculadorasonline.com
<https://calculadorasonline.com/calculadora-de-regresion-lineal-simple/>