ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

ESTRUCTURA DE COMPUTADORES

Grado en Ingeniería Informática
Grado en Ingeniería de Computadores

Tema 4

Ejercicios propuestos
Programación en ensamblador de MIPS

CURSO 2020-21

Susana Mata Fernández
Luis Rincón Córcoles

Área de Arquitectura y Tecnología de Computadores

Introducción

En esta actividad práctica se realizarán diversos ejercicios de programación en ensamblador de `MIPS`, utilizando para ello la herramienta `MARS`, realizada por Peter Sanderson y Kenneth Vollmar de la Missouri State University. Será preciso descargar la última versión del mismo. Puesto que `MARS` está escrito en Java, para poder ejecutarlo será preciso tener instalada una versión reciente del Java Runtime Environment. Se proporcionarán esqueletos para los programas que deben realizarse.

Trabajo preparatorio

Es imprescindible estudiar al completo los contenidos teóricos del bloque de programación en ensamblador.

Antes de comenzar

El material necesario para realizar los ejercicios en el simulador `MARS` se encuentra en el archivo comprimido `Tema04-ejercicios_propuestos-fuentes.zip`. Este archivo deberá ser copiado en una carpeta de un medio de almacenamiento, junto con el archivo ejecutable de `MARS`, y descomprimido posteriormente. Sólo aparecen archivos fuente en ensamblador.

1. Evaluación de una expresión con enteros (I)

Realizar un programa en ensamblador que evalúe la siguiente expresión con números enteros:

$$w = x - y * (z + 2)$$

A continuación se muestra una posible solución en C:

```
void entrada (void);
void salida (void);

int x, y, z, w;

int main (void) {
   entrada();
   w = x - y * (z+2);
   salida();
   return 0;
}
```

En el fichero `expresion_enteros_01.asm` se proporciona un esqueleto de la solución.

2. Evaluación de una expresión con enteros (II)

Diseñar un programa en ensamblador de MIPS que evalúe la siguiente expresión aritmética, donde todas las variables son de tipo entero:

$$e = a + \frac{b \times c}{4} - d \times (a + 2)$$

Adicionalmente existirá una variable carácter llamada `signo` en la que se escribirá el carácter `'+'` si el contenido final de `e` es positivo o nulo, mientras que si `e` termina siendo negativo, en `signo` se copiará el carácter `'-'`.

A continuación se muestra una posible solución escrita en C:

```
void entrada (void);
void salida (void);

int a,b,c,d,e;
int signo;

int main (void) {
   entrada();
   e = a + (b*c) / 4 - d*(a+2);
   if (e >= 0)
      signo = 0;
   else
      signo = 1;
   salida();
   return 0;
}
```

En el fichero `expresion_enteros_02.asm` se proporciona un esqueleto de la solución.

3. Cálculo del máximo entre dos datos enteros

Realizar un programa en ensamblador que, dados dos números enteros, calcule cuál de ellos contiene el valor máximo.

Se realizarán dos versiones del mismo. La primera se basará en el siguiente código en C:

```
void entrada (void);
void salida (void);

int x,y,maximo;

int main (void) {
   entrada();
   maximo = x;
   if (maximo < y)
      maximo = y;
   salida();
   return 0;
}
```

La segunda versión se basará en el siguiente código en C:

```
void entrada (void);
```

```
void salida (void);

int x, y, maximo;

int main (void) {
   entrada();
   if (x >= y)
      maximo = x;
   else
      maximo = y;
   salida();
   return 0;
}
```

En el fichero `maximo.asm` se proporciona un esqueleto de solución que sirve para ambas versiones del programa. Éstas se codificarán en ficheros separados, uno por versión.

4. Triángulo

Tres segmentos s_1, s_2 y s_3 de longitudes dadas l_1, l_2 y l_3 pueden formar triángulo si y sólo si se cumplen simultáneamente las siguientes desigualdades:

$$l_1 < l_2 + l_3$$
$$l_2 < l_1 + l_3$$
$$l_3 < l_1 + l_2$$

Diseñar un programa ensamblador de MIPS que compruebe si tres segmentos de longitudes conocidas pueden formar triángulo. En caso afirmativo, será preciso poner un 1 en una variable llamada `triang`, y en caso negativo se pondrá en `triang` un 0.

A continuación se muestra una posible solución en C:

```
void entrada (void);
void salida (void);

int lon1, lon2, lon3;
register int tmp1, tmp2, tmp3; // Temporales
int triang;

int main (void) {
   entrada();
   tmp1 = lon2 + lon3;
   tmp2 = lon1 + lon3;
   tmp3 = lon1 + lon2;
   if ( (lon1<tmp1) && (lon2<tmp2) && (lon3<tmp3) )
      triang = 1;
   else
       triang = 0;
   salida();
   return 0;
}
```

En el fichero `triangulo.asm` se proporciona un esqueleto de la solución.

5. Serie de Fibonacci

Realizar un programa en ensamblador que calcule el término enésimo de la serie de Fibonacci. Esta serie se define del siguiente modo:

$$f_n = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ f_n = f_{n-1} + f_{n-2} & n > 1 \end{cases}$$

A continuación se muestra una posible solución en C:

```
void entrada (void);
void salida (void);

int n, f;
register int f1, f2;
register int i;

int main (void) {
   entrada();
   f2 = 0;
   f1 = 1;
   for (i = 2; i <= n; i++) {
      f = f1 + f2;
      f2 = f1;
      f1 = f;
   }
   salida();
   return 0;
}
```

En el fichero `fibonacci.asm` se proporciona un esqueleto de la solución.

6. Máximo común divisor y mínimo común múltiplo

Realizar un programa en ensamblador que calcule el máximo común divisor y el mínimo común múltiplo de dos números a y b. Para calcular el máximo común divisor se utilizará el algoritmo de Euclides. El mínimo común múltiplo se calculará multiplicando los dos datos de entrada y dividiendo el resultado por el máximo común divisor recién calculado.

A continuación se muestra una posible solución en C:

```
void entrada (void);
void salida (void);

int x, y, mcd, mcm;
register int tmp, resto;


int main (void) {
   entrada();
   if (x >= y) {
```

```
      tmp = x;
      mcd = y;
   }
   else {
      tmp = y;
      mcd = x;
   }
   do {
      resto = tmp % mcd;
      if (resto != 0) {
         tmp = mcd;
         mcd = resto;
      }
   } while (resto != 0);
   mcm = x * y / mcd;
   salida();
   return 0;
}
```

En el fichero `mcd_mcm.asm` se proporciona un esqueleto de la solución.

7. Término enésimo de una serie

Realizar un programa en ensamblador que calcule el término T(n) de la sucesión definida por:

$$T(i) = \begin{cases} 1 & i = 0 \\ 2 \times T(i-1) & i\ impar \\ 2 \times T(i-1) - 1 & i\ par \end{cases}$$

A continuación se muestra una posible solución en C:

```
void entrada (void);
void salida (void);

int n, tn;
register int i;

int main (void) {
   entrada();
   tn = 1;
   for (i = 1; i <= n; i++) {
      tn = 2 * tn;
      if (i % 2 == 0)
         tn = tn - 1;
   }
   salida();
   return 0;
}
```

En el fichero `termino_serie.asm` se proporciona un esqueleto de la solución.

8. Números perfectos

Un número natural es perfecto si la suma de sus divisores propios (distintos de él mismo) es igual a sí mismo. Por ejemplo, el número 6 es perfecto, ya que sus divisores propios son 1, 2 y 3, que sumados dan 1+2+3=6.

Realizar un programa en ensamblador que, dado un dato de tipo entero positivo (32 bits), indique si se trata de un número perfecto. Se puede partir de la siguiente solución en C:

```
void entrada (void);
void salida (void);

int n, perfecto;
register int i, suma;

int main (void) {
   entrada();
   suma = 1;
   i = 2;
   while (i < n) {
      if (n % i == 0)
         suma = suma + i;
      i++;
   }
   perfecto = (n == suma);
   salida();
   return 0;
}
```

En el fichero `perfecto.asm` se proporciona un esqueleto de la solución.

9. Cálculo de la raíz cuadrada de un número mediante el método de Newton-Raphson

El método de Newton-Raphson sirve para calcular las raíces de una ecuación $f(x) = 0$ de forma iterativa. La fórmula de recurrencia se obtiene a partir de la definición de derivada de una función:

$$f'(x_i) = \frac{f(x) - f(x_i)}{x - x_i}$$

De aquí se obtiene la ecuación de la recta tangente a $f(x)$ en $x = x_i$:

$$t(x) = f'(x_i) \cdot (x - x_i) + f(x_i)$$

Tal como puede verse en la figura 1, dicha recta tangente corta el eje de abscisas en el punto $x = x_{i+1}$ donde:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \tag{1}$$

La ecuación 1 es la base del método. Para calcular la solución de una función $f(x)$, se tomará inicialmente un valor semilla $x = x_0$ suficientemente próximo a la raíz buscada, y que constituirá

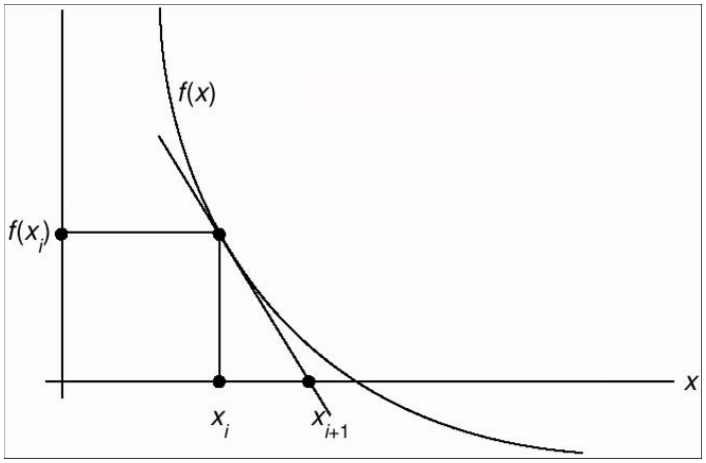

Figura 1: Función $f(x)$ y su recta tangente.

la primera aproximación a la solución. A continuación, se aplicará la ecuación 1 para obtener una nueva aproximación $x = x_1$, repitiéndose el proceso sucesivamente hasta que el método converja a una solución $x = x_{i+1}$, cosa que detectaremos cuando sea cierta la expresión $|x_{i+1} - x_i| < \varepsilon$, donde ε es un valor positivo y muy próximo a 0, establecido a priori en función de la precisión buscada.

Se desea diseñar en ensamblador de MIPS un programa que calcule la raíz cuadrada de un número Z mediante el método de Newton-Raphson. Se partirá de la función

$$f(x) = x^2 - Z \tag{2}$$

cuya raíz es

$$x = \sqrt{Z}$$

La derivada de la función es

$$f'(x) = 2x \tag{3}$$

con lo que, sustituyendo las ecuaciones 2 y 3 en la ecuación 1 se obtiene la siguiente ecuación de recurrencia:

$$x_{i+1} = 0,5 \cdot \left(x_i + \frac{Z}{x_i} \right) \tag{4}$$

A continuación se presenta una posible solución en C al problema pedido:

```
void entrada (void);
void salida (void);

float z, raiz_z, epsilon;
register float xi, xi_1;

int main (void) {
   entrada();
   xi_1 = 1; /* Primera aproximacion a la solucion */
   do {
```

```
      xi = xi_1;
      xi_1 = 0.5*(xi-z/xi);
   while (fabs(xi_1-xi) >= epsilon);
   raiz_z = xi_1;
   salida();
   return 0;
}
```

El fichero `raiz_cuadrada.asm` contiene un esqueleto de la solución.

10. Copia de vectores de enteros

Diseñar un programa en ensamblador de `MIPS` que copie un vector de 16 números de 32 bits en complemento a 2 en otro vector de igual tamaño.

Se realizarán varias versiones del programa. La primera versión copiará los elementos en orden creciente de índice. Para ello se puede partir del siguiente código en C:

```
void salida (void);

#define N 16

int x[N] = {100,200,300,400,500,600,700,800,900,1000,1100,1200,1300,1400,1500,1600};
int y[N];
register int i;

int main (void) {
   for (i = 0; i < N; i++)
      y[i] = x[i];
   salida();
   return 0;
}
```

La segunda versión es similar a la primera, pero ahora el recorrido por el vector se realizará mediante punteros, y no mediante índices. Se puede partir del siguiente código en C:

```
void salida (void);

#define N 16

int x[N] = {100,200,300,400,500,600,700,800,900,1000,1100,1200,1300,1400,1500,1600};
int y[N];
register int *px, *px_fin, *py;
register int i;

int main (void) {
   px = x;
   px_fin = x + N;
   py = y;
   while (px != px_fin) {
      *py = *px;
      px++;
```

```
      py++;
   }
   salida();
   return 0;
}
```

La tercera versión copiará los elementos en orden decreciente de índice. Para ello se puede partir del siguiente código en C:

```
void salida (void);

#define N 16

int x[N] = {100,200,300,400,500,600,700,800,900,1000,1100,1200,1300,1400,1500,1600};
int y[N];
register int i;

int main (void) {
   for (i = N-1; i >= 0; i--)
      y[i] = x[i];
   salida();
   return 0;
}
```

La cuarta y última versión es similar a la tercera, pero ahora el recorrido por el vector se realizará mediante punteros, y no mediante índices. Se puede partir del siguiente código en C:

```
void salida (void);

#define N 16

int x[N] = {100,200,300,400,500,600,700,800,900,1000,1100,1200,1300,1400,1500,1600};
int y[N];
register int *px, *px_ini, *py;
register int i;

int main (void) {
   px_ini = x;
   px = x + N;
   py = y + N;
   while (px != px_ini) {
      px--;
      py--;
      *py = *px;
   }
   salida();
   return 0;
}
```

En el fichero `vectores_int_copia.asm` se proporciona un esqueleto de programa que sirve para cada una de las soluciones. Éstas se codificarán en ficheros separados, uno por versión.

11. Suma de vectores de enteros

Realizar un programa en ensamblador de `MIPS` que, dados dos vectores que contienen 16 números de 32 bits expresados en complemento a 2, sume sus componentes y genere como resultado un nuevo vector de igual tamaño que los anteriores. Se puede partir del siguiente código en C:

```
void salida (void);

#define N 16

int x[N] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16};
int y[N] = {100,200,300,400,500,600,700,800,900,1000,1100,1200,1300,1400,1500,1600};
int z[N];
register int i;

int main (void) {
   for (i = 0; i < N; i++)
      z[i] = x[i] + y[i];
   salida();
   return 0;
}
```

En el fichero `vectores_int_suma.asm` se proporciona un esqueleto de la solución.

12. Valores máximo y mínimo de un vector de enteros

Diseñar un programa en ensamblador de `MIPS` que calcule los valores máximo y mínimo de un vector de enteros. Se puede partir del siguiente código en C:

```
void salida (void);

#define N 16

int vector[N] = {1,-3,55,8905,-111,90,543,-4321,0,43,99,-34,-55,4231,45,-4532543}
int maximo, minimo;
register int i, aux;

int main (void) {
   maximo = vector[0];
   minimo = maximo;
   for (i = 1; i < N; i++) {
      aux = vector[i];
      if (aux < minimo)
         minimo = aux;
      else if (aux > maximo)
         maximo = aux;
   }
   salida();
   return 0;
}
```

En el fichero `vector_int_minmax.asm` se proporciona un esqueleto de la solución.

13. Número de unos

Realizar un programa en ensamblador que, dado un dato de tamaño palabra (32 bits), inspeccione los bits que lo componen y contabilice cuántos de entre ellos se encuentran a 1.

A continuación se muestra una posible solución en C:

```
void entrada (void);
void salida (void);

int dato, unos;
register int tmp, mascara;

int main (void) {
   entrada();
   unos = 0;
   tmp = dato;
   mascara = 0x80000000;
   while (tmp != 0) {
      if (tmp & mascara != 0)
         unos = unos + 1;
      tmp = tmp << 1;
   }
   salida();
   return 0;
}
```

En el fichero `unos.asm` se proporciona un esqueleto de la solución.

14. Convertir las letras minúsculas de un *string* a mayúsculas

Diseñar un programa en ensamblador de `MIPS` que, dada una cadena de caracteres, convierta las letras minúsculas que contenga a mayúsculas.

Se puede partir del siguiente código en C:

```
void entrada (char *str);
void salida (void);

register int i;
char string[30];
register char caracter;

int main (void) {
   entrada(string);
   i = 0;
   while ( (caracter = string[i]) != '\0' ) {
      if ( (caracter >= 'a') && (caracter <= 'z') )
         string[i] = caracter -'a' + 'A';
      i++;
   }
   salida();
   return 0;
```

```
}
```

En el fichero `string_to_upper.asm` se proporciona un esqueleto de la solución.

15. Contar cifras en un *string*

Diseñar un programa en ensamblador de `MIPS` que, dada una cadena de caracteres, cuente cuántos de ellos corresponden con dígitos del '0' al '9'.

Se puede partir del siguiente código en C:

```
void entrada (char *str);
void salida (void);

int digitos;
register int i;
char string[30];
register char caracter;

int main (void) {
   entrada(string);
   digitos = 0;
   i = 0;
   while ( (caracter = string[i]) != '\0' ) {
      if ( (caracter >= '0') && (caracter <= '9') )
         digitos++;
      i++;
   }
   salida();
   return 0;
}
```

En el fichero `string_contar_cifras.asm` se proporciona un esqueleto de la solución.

16. Palíndromos

Diseñar un programa en ensamblador de `MIPS` que determine si una cadena de caracteres `ASCII` es o no palíndroma. Una cadena es palíndroma si se lee igual de derecha a izquierda que de izquierda a derecha.

Se puede partir del siguiente código en C:

```
void entrada (void);
void salida (void);

register char *ini, *fin;
char string[30];
char palin;

int main (void) {
   entrada();
```

```
   // Buscar el final de la tira
   ini = fin = string;
   while (*fin != '\0')
      fin++;
   fin--;
   // Comparar los caracteres
   palin = 'S';
   while (ini < fin) {
      if (*ini == *fin) {
         ini++;
         fin--;
      }
      else {
         palin = 'N';
         break;
      }
   }
   salida();
   return 0;
}
```

En el fichero `string_palindromo.asm` se proporciona un esqueleto de la solución.

17. Subrutina de ordenación de tres valores enteros

Realizar un programa en ensamblador de `MIPS` que ordene tres números enteros de mayor a menor. Para ello se diseñará una subrutina auxiliar que realice la tarea antedicha. La subrutina se denominará `ordenar3`. Los tres parámetros de la subrutina se pasarán por referencia a través de los registros `$a0`, `$a1` y `$a2`. La subrutina pondrá el número más grande en la variable pasada por referencia en primer lugar (apuntada por `$a0`), el número intermedio en la variable pasada por referencia en segundo lugar (apuntada por `$a1`), y el número menor en la variable pasada por referencia en tercer lugar (apuntada por `$a2`). Se podrá partir de la siguiente versión del programa en C:

```
void entrada (void);
void salida (void);
void ordenar3 (int *x, int *y, int *z);

int x,y,z;

int main (void) {
   entrada();
   ordenar3(&x,&y,&z);
   salida();
   return 0;
}

void ordenar3 (int *x, int *y, int *z) {
   int tmp;

   if (*x < *y) {
      tmp = *x;
```

```
      *x = *y;
      *y = tmp;
   }
   if (*x < *z) {
      tmp = *x;
      *x = *z;
      *z = tmp;
   }
   if (*y < *z) {
      tmp = *y;
      *y = *z;
      *z = tmp;
   }
   return;
}
```

En el fichero `main_ordenar.asm` se proporciona la reserva de espacio para variables, así como el programa principal completo. El alumno realizará la subrutina en el fichero `ordenar3.asm`.

18. Media aritmética de un vector de datos en coma flotante

Diseñar un programa en ensamblador de `MIPS` que calcule la media aritmética de un vector de números en coma flotante de precisión simple. Se utilizará una subrutina llamada `promedio`, que recibirá como parámetros un vector de datos de tipo `float` y un entero con el número de elementos del mismo.

Se podrá partir de la siguiente versión del programa en C:

```
void entrada (void);
void salida (void);
float promedio (float *poly, int n);

float vector[10] = {1.5, 2.5, 3.5, 4.5, 5.5, 6.5, 7.5, 8.5, 9.5, 10.5};
int n = 10;
register float suma;
float media;

int main (void) {
   suma = promedio(vector,n);
   salida();
   return 0;
}

float promedio (float *vec, int n) {
   register int i;
   register float v;

   i = 0;
   v = 0;
   do {
      v = v + vec[i];
      i++;
```

```
   } while (i < n);
   return (v/n);
}
```

El fichero `main_promedio.asm` contiene la reserva de espacio para variables y el programa principal completo. El alumno realizará la subrutina en el fichero `vector_float_promedio.asm`.

19. Polinomios

Se pretende diseñar un programa en ensamblador de `MIPS` que calcule el valor de un polinomio $P(x)$ para un valor concreto de la variable x. Para ello se utilizará una subrutina llamada `horner`, que recibirá como parámetros un vector con los coeficientes del polinomio, un entero que indica su grado y otro entero para la variable independiente, y realizará la evaluación del polinomio mediante el algoritmo de Horner.

Se podrá partir de la siguiente versión del programa en C:

```
void salida (void);
float horner (float *poly, int grado, float x);

float polinomio[10] = {11.0,5.0,3.5,4.0,0.0,0.0,0.0,0.0,0.0,0.0};
int grado = 3;
float x = 2.5;
float valor;

int main (void) {
   valor = horner(polinomio,grado,x);
   salida();
   return 0;
}

float horner (float *poly, int grado, float x) {
   register int i;
   register float v;

   i = grado;
   v = 0;
   do {
      v = v * x + poly[i];
      i--;
   } while (i >= 0);
}
```

En el fichero `main_polinomio.asm` se proporciona la reserva de espacio para variables, así como el programa principal completo. El alumno realizará la subrutina en el fichero `horner.asm`.

20. Números combinatorios

Se pretende diseñar un programa en ensamblador de MIPS que calcule el valor de un número combinatorio. Recordar que un número combinatorio se expresa como:

$$\binom{n}{k}$$

y que su valor se calcula como:

$$\binom{n}{k} = \frac{n!}{k! \cdot (n-k)!}$$

Para realizar el cálculo se desarrollará una subrutina auxiliar de tipo función llamada `factorial` que calcula el factorial de un número, y que presenta las siguientes características:

- Recibe su único parámetro de entrada en el registro de argumento $a0.
- Devuelve el resultado en el registro $v0.
- Es una subrutina "hoja".

Se podrá partir de la siguiente versión del programa en C:

```
int n, k, r;

int main (void) {
   entrada();
   r = factorial(n) / (factorial(k)*factorial(n-k));
   salida();
   return 0;
}

int factorial (int n) {
   register int i, f;

   f = 1;
   i = 1;
   while (i < n) {
         i = i + 1;
         f = f * i;
      }
   return f;
}
```

En el fichero `main_combinatorio.asm` se proporciona la reserva de espacio para variables, así como el programa principal completo. El alumno realizará la subrutina en el fichero `factorial.asm`.