Lab06 - ARCH - 2019-II

{TA:ariana.villegas, Prof: jgonzalez} @utec.edu.pe

25 September 2019

1 Indicaciones

1. Fecha de **entrega**: `Miércoles 15 de Octubre del 2019 a las 12pm en Canvas`

2 Objetivos

1. Aprender a programar el procesador en *low level*.
2. Implementar un programa que pueda ser ejecutado en un procesador MIPS.
3. Entender los diferentes modos de addressing de un procesador.

3 Usando MARS

MARS (MIPS Assembler and Runtime Simulator) es un IDE para programación en lenguaje Assembly.

1. Ejecutar Mars.jar. El archivo run_mars.bat puede usarse en Windows, contiene el comando:

   `java -jar Mars.jar`.

2. Abrir el archivo lab06.asm (template para escribir nuestros programas).
3. En *Settings* y *Memory Configuration*, seleccionar *Compact, Data at Address 0* y luego click en *Apply and Close*.
4. Ahora podemos escribir y simular nuestro programa. Click en el botón *Assemble the current file* y luego en *Run the current program* (ejecución sin parar)o en *Run one step at the time* (una instrucción por vez, F7

cumple la misma función).

5. El simulador muestra un *Text Segment* (nuestras instrucciones assembled), y un *Data Segment* (usado con operaciones LW y SW).

6. Para mas información y otros códigos de ejemplo revisar:

   `https://courses.missouristate.edu/KenVollmar/mars/tutorial.htm`

4 Ejercicios

Los sistemas de cómputo son usados para ejecutar programas complejos. La mayoría de programas son escritos en un lenguaje high-level, como C++ o Java. Sin embargo, para ejecutarlos estos deben ser traducidos a operaciones básicas que el procesador pueda ejecutar, esto es, lenguaje *assembly*. Escribir código assembly puede ser tedioso, pero nos permite tener un control total de lo que se ejcuta: el procesador ejecuta exactamente aquellas instrucciones especificadas en el código assembly.

En este laboratorio, escribiremos algunos programas usando las operaciones básicas que soporta el ALU diseñado en el Lab04.

4.1 Un programa simple que usa un set de instrucciones limitado

En este ejercicio, escribiremos un programa en assembly-level pensando en un procesador MIPS que use el ALU del Lab04, el cual soporta un conjunto de instrucciones aritméticas muy limitado.

NOTA: revisar el MIPS Cheat Sheet, libros y slides como material de apoyo.

El objetivo del programa es encontrar la suma de enteros positivos consecutivos desde A hacia B:

$S = A + (A + 1) + (A + 2) + ... + (B - 1) + B$

Como sabemos, esta suma puede ser calculada usando la ecuación de Gauss. Sin embargo, el método de Gauss requiere multiplicaciones y divisiones, ninguna de las cuales puede ser implementada directamente con nuestro ALU actual.

Una característica importante de los microprocesadores es que, por mas limitados en las operaciones que puedan hacer, estas las realizan rápido. Mientras que el trabajo de sumar 1000 números debe haber sido un trabajo pesado para el pequeño Gauss cuando estaba en el colegio, nuestro lento procesador con nuestra ALU (operando a 50 MHz) puede sumar hasta un millón de números

mas rápido de lo que pestañeamos. Este es principal beneficio de los sistemas de cómputo: ejecutar muy rápido un gran número de cálculos.

1. Usar el simulador MARS (MIPS simulator) para escribir un programa en assembly que calcule la suma propuesta.
2. Usar solo las instrucciones ADD, SUB, SLT, XOR, AND, OR, NOR, J, ADDI and BEQ. **NOTA:** A pesar que no tenemos implementadas las instrucciones J, ADDI o BEQ en nuestro ALU, úsenlas libremente si lo consideran necesario.
3. Inicializar su data (A y B) usando la instrucción ADDI, y guardar el resultado en el registro $\$t2$. Estudiar la operación de ADDI, notar que solo manipula 16-bit signed numbers, lo que significa que A y B deben ser números de 16-bit.
4. Al escribir su programa, puede usar registros temporales $\$t0$ al $\$t7$ para retener sus variables.

4.2 Un programa más complejo: Suma de Diferencias Absolutas

La Suma de Diferencias Absolutas (en inglés, SAD) es un algoritmo ampliamente usado para medir la similitud entre imágenes. Este calcula la diferencia absoluta entre cada pixel en la imagen original y el pixel correspondiente en la imagen usada para comparación. SAD es usado en múltiples aplicaciones como reconocimiento de objetos, generación de mapas de disparidad para imágenes stereo, y estimación de movimiento para compresión de video.

Las siguientes figuras se ven iguales (ver Figura 1), sin embargo, son imágenes de tomadas en diferentes ángulos de cámara. De esta forma, la suma de diferencias absoulutas provee el siguiente mapa de disparidad (ver Figura 2).

Figure 1: Ejemplo de imágenes para comparar usando la Suma de Diferencias

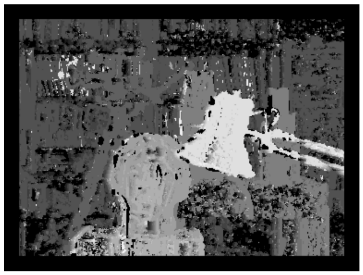

Figure 2: Mapa de disparidad obtenido usando el algoritmo SAD

En este ejercicio implementarán el algoritmo SAD. Pueden usar el MIPS ISA completo para su implementación (MARS implementa todas las instrucciones).

El código en C del algoritmo SAD es el siguiente:

```
// Implementation of the absolute value of differences
int abs_diff(int pixel_left, int pixel_right) {
    int abs_diff = abs(pixel_left - pixel_right);
    return abs_diff;
}

// Recursive sum of a vector
int recursive_sum(int arr[], int size) {
        if(size==0)
                return 0;
        else
                return recursive_sum(arr,size-1)+arr[size-1];
}

// main function
int main() {
        int sad_array[9];
        int image_size = 9; // 3x3 image
        // These vectors must be stored in memory
        int left_image[9] = {5, 16, 7, 1, 1, 13, 2, 8, 10};
        int right_image [9] = {4, 15, 8, 0, 2, 12, 3, 7, 11};

        for (i = 0; i < image_size; i++)
                sad_array[i] = abs_diff(left_image[i], right_image[i]);

        sad_value = recursive_sum(sad_array, image_size);
}
```

left_image[0]	left_image[1]	left_image[2]
left_image[3]	left_image[4]	left_image[5]
left_image[6]	left_image[7]	left_image[8]

Figure 3: Array de pixels

Abrir en MARS el archivo `lab06_sad.asm`. Contiene partes necesarias de código para implementar el algoritmo SAD. Completar el código assembly en las secciones TODO.

4.2.1 TODO1: Inicializando datos en memoria

Las imágenes son almacenadas en la memoria de datos como un array de pixeles. Cada pixel en la imagen es representado por un valor de 0 a 16 equivalente a *grayscale*. Por ejemplo, una imagen de 3×3 pixels es almacenada como un array de 9 elementos (ejemplo, `left_image[]`), y cada elemento del array corresponde al pixel de la imagen como sigue:

En el ejercicio anterior (suma Gauss) no podíamos usar instrucciones LW y SW, por lo que la data era inicializada alamacenándola directamente en los registros. Esto funciona cuando tenemos poca data, pero en general, un programa funciona con grandes cantidades de data y debe ser almacenada en memoria. En este ejercicio almacenaremos los pixels de la imagen en memoria. Asegurarse que el **data segment comienza en el address 0x00000000**. El layout de su memoria debe ser como la Figura 4:

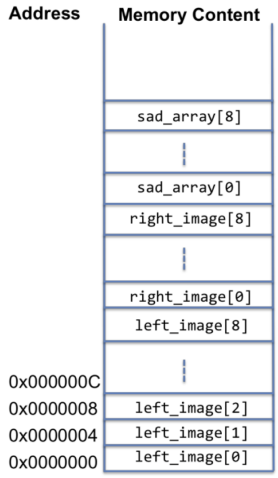

Figure 4: Layout de Memoria

Implementar la sección de inicializando data en memoria usando la instrucción SW y la data inicial especificada en el programa en C.

4.2.2 TODO2 a 5: implementar funciones

1. **TODO2:** Implementar la función `abs_diff()`. Revisar el capítulo 6 del libro Harris.

2. **TODO3:** Implementar la función `recursive_sum(int arr[], int size)`. Esta toma como primer argumento el address del primer elemento del array `sad_array[]`, y el segundo argumento es el número de pixeles en la imagen. **Observación:** es una función recursiva y por lo tanto necesita almacenar los registros correspondientes en el stack antes de llamar a la función recursiva; de otra forma estos serán sobreescritos por la función llamada.

3. **NOTA:** las funciones anteriores están implementadas en los archivos `helper_abs_diff.asm` y `helper_recursive_sum.asm`. Las personas que implementen sus propias funciones recibirán puntos adicionales en la evaluación (indicar en su informe que su función es propia).

4. Completar la función main para hacer el llamado de funciones. Bajo el TODO4, llenar la sección *"loop:"* para hacer un loop sobre los elementos de la imagen. Para obtener los pixels se debe usar la instrucción LW. Luego, llamar a la rutina `abs_diff()`, guardar el resultado en la correspondiente posición de `sad_array[]`. Despues de la ejecución del loop, deben hacer un jump a `end_loop`.

5. Luego de la ejecución de loop, agregar todos los elementos del array `sad_array[]` usando la función implementada `recursive_sum()`, y almacenar el resultado final en $\$t2$. Bajo el TODO5, completar la sección `end_loop:` para preparar los argumentos para llamar la función `recursive_sum()`, y guardar el resultado en $\$t2$.

La Figura 5 muestra los contenidos esperados en memoria al finalizar la ejecución del programa.

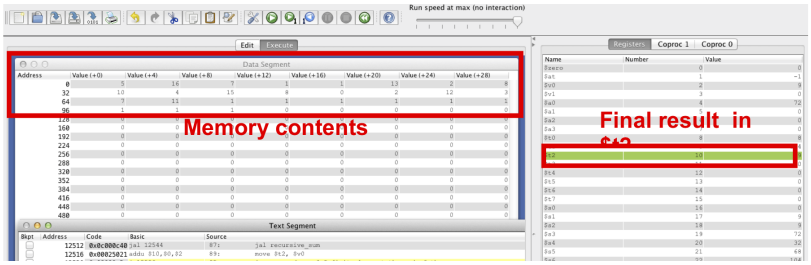

Figure 5: Memoria al final de la ejecución del programa

5 Indicaciones

1. Implementar los archivos indicados en assembly.
2. Escribir un informe explicando. Incluir el código en el informe explicando las funciones (en Latex pueden usar lstlisting).
3. Subir en un .tar **solamente** archivos requeridos y necesarios (.asm, .txt, etc.) e informe final en pdf.