Manual de Laboratorio para Arquitectura y Organización de Computadoras Usando XSPIM

Este manual esta basado en las notas de Laboratorio del curso de Arquitectura de Computadoras que se imparte en la Facultad de Ciencias de la Universidad Nacional Autónoma de México, que desarrolló Julián Estévez para el curso 1999-I. Y esas notas, a su vez, están basadas en el curso de SPIM, un simulador del procesador MIPS R2000/R3000 desarrollado por James R. Laurus en la Universidad de Wisconsin-Madison.

XSPIM se encuentra disponible a través de Internet en http://www.cs.wisc.edu/~larus/spim.html en sus versiones para sistemas UNIX/X11 y MS-Windows. Contando con una versión no gráfica (SPIM) para MS-DOS. También se localiza en este sitio la documentación del simulador.

Varias preguntas y ejercicios fueron tomadas de la primera edición del libro "*Computer organization and design: the hardware/spftware interface"* de David A. Patterson y John L. Hennesy. Algunas sesiones son basadas en las sesiones del curso Computer Organization II impartido en 1998 durante el periodo de primavera en la Universidad de Texas en San Antonio.

La primer versión de este manual fue utilizado por primera ocasión durante el curso de Arquitectura de Computadoras correspondiente al semestre 99-I impartido por la profesora Ana Luisa Solís González-Cosió en la Facultad de Ciencias de la UNAM. Siendo utilizado posteriormente por el profesor José De Jesús Galavíz Casas. Con algunas modificaciones fue utilizado también por el profesor Jorge L. Ortega Arjona, teniendo que actualizarse para esta nueva versión para el curso 2003-I.

Hugo González Medina

Noviembre 2002

Cuando en una sesión de laboratorio se requiera realizar un programa y/o reporte, éste deberá entregarse por e-mail. El mensaje electrónico deberá contener un archivo .tar.z o .tgz(o en su defecto .zip) con los programas incluidos y/o el reporte de la sesión (si es necesario). Al expandir el archivo tgz, deberá contener un directorio llamado como el login de tu cuenta, luego un subdirectorio llamado como la sesión (por ejemplo sesion1, o sesion2), finalmente, los programas y/o reportes a entregar. Los reportes deberán ser enviados en formato postscript (ps) o en archivo PDF, cualquier otro formato no será recibido. Y algo muy importante, los programas y/o reportes deberán de entregarse en la fecha indicada, no se recibirán después.

Sesión 1

Usando XSPIM, un simulador del MIPS R2000/R3000

1.1 Objetivo General

Presentar las características principales de XSPIM, el simulador del procesador MIPS R2000/R3000 que usaremos a lo largo de las sesiones del laboratorio.

1.2 Objetivos específicos

Al terminar esta sesión de laboratorio:

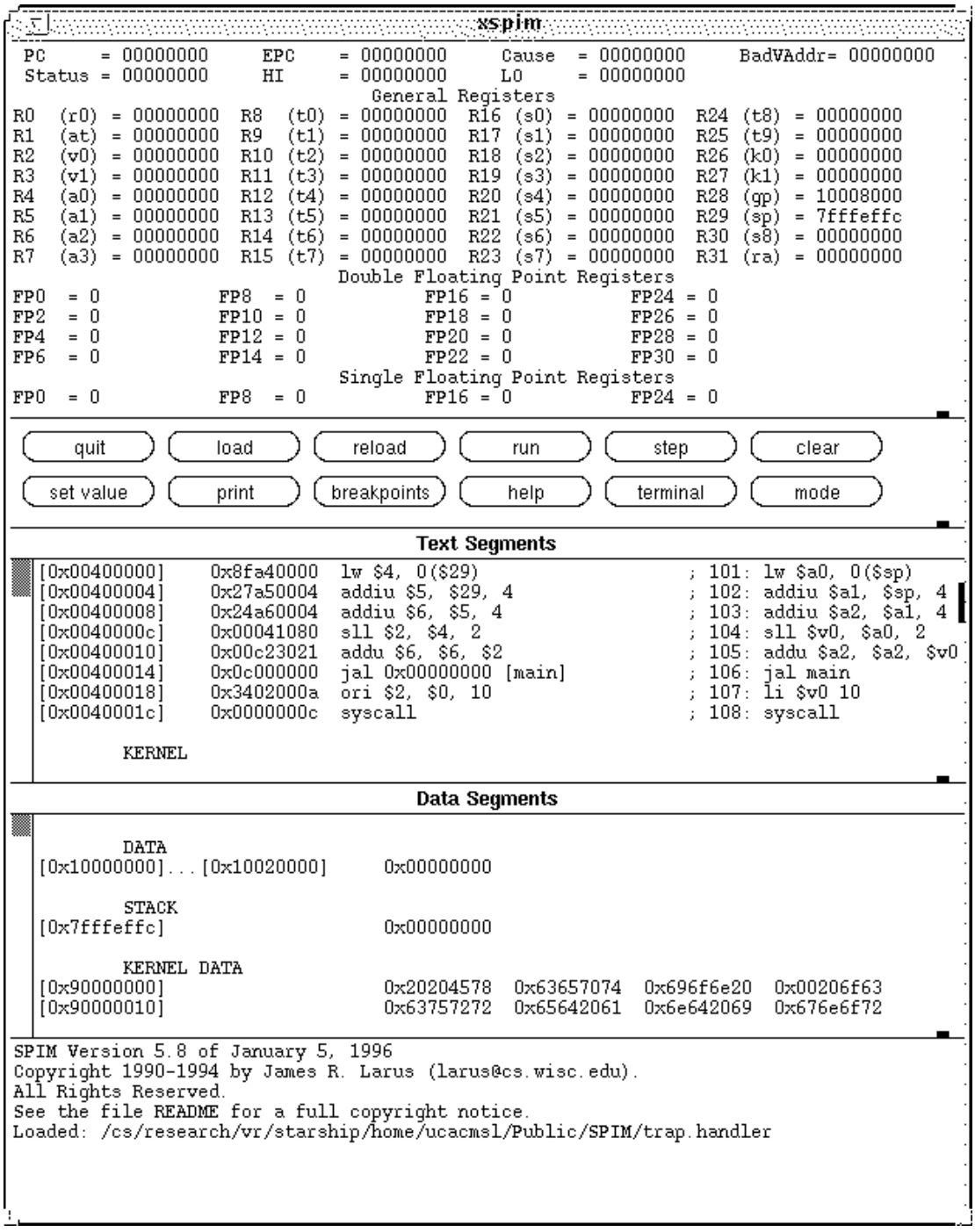
- Serás capaz de cargar y ejecutar programas de MIPS en XSPIM.
- Examinar/modificar el contenido de las localidades de memoria
- Examinar/modificar el contenido de los registros
- Tendrás las nociones básicas para el funcionamiento de XSPIM

1.3 Introducción

XSPIM es un simulador que ejecuta programas para las computadoras RISC basadas en los procesadores MIPS R2000/R3000 y ha sido desarrollado por James R. Laurus en la Universidad de Wisconsin-Madison. XSPIM puede leer archivos que contienen programas en lenguaje ensamblador y permite su ejecución tanto paso a paso como sin interrupciones. XSPIM provee al usuario, además, una interface con algunos servicios del sistema operativo.

Evidentemente, la ejecución de código MIPS en una máquina con este procesador resultaría mucho más rápida que el empleo de un simulador, sin embargo, en cursos como este se usan simuladores para tal propósito porque, sin tomar en cuenta que máquinas con tales procesadores no se encuentran disponibles en cualquier lugar, los simuladores brindan mejores ambientes de desarrollo para programadores de bajo nivel que las máquinas reales ya que, en general, pueden detectar un mayor número de errores.

Aquí tienes una imagen de la interfaz del xspim como apoyo.



1.4 Cargando y ejecutando un programa en ensamblador

1. Teclea el siguiente archivo, simple.s.
2. Comienza la ejecución de xspim.
3. Carga el archivo simple.s usando el botón load.
4. Ejecuta el botón run. Como consecuencia de la ejecución debe aparecer una ventana que hace las veces de consola (en ella despliega la salida del programa y se toman las entradas del teclado) con el mensaje =). Debes tener cuidado al manejar la consola, pues bajo algunos manejadores de ventanas cerrar esta ventana con los controles en su borde implica terminar la ejecución de la aplicación; si deseas cerrar la ventana, mejor utiliza el botón terminal del panel de control.
5. Vuelve a cargar el archivo simple.s y ejecútala paso a paso ayudándote de la caja de diálogo que aparece al presionar el botón step del panel de control. Trata de entender el efecto que tienen las instrucciones ejecutadas sobre los registros.

1.4.1 simple.s

El código es dado en el laboratorio

1.5 Examinando y modificando el contenido de la memoria

Para examinar el contenido de la memoria basta con ver el panel de los segmentos de datos y buscar la localidad de memoria deseada. En el panel de segmentos se observa primero un número hexadecimal entre corchetes cuadrados y luego un grupo de cuatro hexadecimales de 32bits cada uno en el mismo renglón. El primer hexadecimal indica la localidad de memoria en la que está almacenado el primer byte del primer grupo de cuatro y los siguientes bytes del grupo se encuentran en los quince bytes siguientes de la memoria.

1. Carga (o vuelve a cargar) el archivo simple.s.
2. Usando el botón set value del panel de control, fija el valor de la localidad de memoria 0x100010000 al valor 0x2a2a2a2a.
3. Ejecuta el programa y verás que ahora el mensaje que se imprime ahora es ****.

1.6 Examinando y modificando el contenido de los registros

En el primer panel de la ventana de xspim se muestran los registros del MIPS y su contenido. Como podrás observar, se tienen 32 registros de propósito general de 32 bits cada uno. Nota que cada uno tiene asociado un identificador de dos letras que se muestran entre paréntesis. Posteriormente aprenderemos el propósito de cada uno de estos registros.

Para modificar el contenido de un registro se sigue un procedimiento análogo al visto en la sección 1.5.

1. Carga (o vuelve a cargar) el archivo simple.s
2. Ejecuta paso a paso el programa hasta que el registro llamado PC (Program Counter) contenga el valor 0x00400028.
3. Usando el botón set value del panel de control fija el valor del registro $9 en 5.
4. Continua la ejecución del programa y verás que ahora el mensaje que se imprime ahora es =(.

1.7 Preguntas de revisión

1. Describe cada uno de los paneles de la ventana de xspim.
2. Describe cada uno de los botones del panel de control de xspim.
3. Según la documentación de SPIM. ¿Cuáles son las características sorprendentes del simulador respecto a la máquina real?
4. Según la documentación de SPIM. ¿por qué no es posible modificar el contenido del registro 0 (R0) del MIPS?
5. Revisa las instrucciones del ensamblador del MIPS descritas en la documentación de SPIM y explica cuál fue la causa del cambio en el mensaje impreso por simple.s en el punto 1.4.

Sesión 2

Usando las directivas del ensamblador del MIPS y las llamadas al sistema en XSPIM

2.1 Objetivo General

Estudiar y entender el uso tanto de las directivas del ensamblador del MIPS como de las llamadas al sistema provistas por XSPIM.

2.2 Objetivos específicos

Al terminar esta sesión de laboratorio:

- Podrás distinguir y usar apropiadamente las directivas del ensamblador del MIPS que pueden simularse en XSPIM.
- Sabrás cómo utilizar las llamadas al sistema provistas por XSPIM en los programas en lenguaje ensamblador del MIPS que realices durante las siguientes sesiones del laboratorio, en particular, sabrás cómo
    - Imprimir y leer cadenas
    - Imprimir y leer enteros
    - Imprimir y leer flotantes
    - Imprimir y leer dobles

2.3 Directivas del ensamblador

Seguramente notaste en el programa simple.s de la sesión anterior, algunas palabras como .data, .globl o .asciiz. Las palabras, como estas, que comienzan en punto (".") se llaman directivas del ensamblador (o seudo códigos de operación) y sirven para decirle al ensamblador cómo traducir un programa, pero no producen instrucciones de máquina.

De la documentación de SPIM, encuentra la parte donde se refiere a las directivas del ensamblador del MIPS que XSPIM entiende. Existen de hecho muchas más directivas, pero no las consideramos a lo largo de estas sesiones de laboratorio.

2.4 Llamadas al sistema en XSPIM

XSPIM provee un pequeño número de servicios parecidos a los del sistema operativo a través de la llamada del sistema syscall. Para solicitar un servicio del sistema en un programa, se carga el código del servicio en el registro $v0. Los registros donde se colocan los argumentos de cada servicio, al igual que aquellos donde se reciben sus valores de retorno, dependen de cada servicio, como se muestra en la tabla 1.

Servicio	Código	Argumentos	Resultados
print_int	1	$a0 = integer	
print_float	2	$f12 = float	
print_double	3	$f12 = double	
print_string	4	$a0 = string	
read_int	5		integer (in $v0)
read_float	6		float (in $f0)
read_double	7		double (in $f0)
read_string	8	$a0 = buffer, $a1 = length	
sbrk	9	$a0 = amount	address (in $v0)
exit	10		

Tabla 1

Ejecuta el programa syscall.s (el código se te proporcionará en el laboratorio) en XSPIM. Observa el uso de las directivas del ensamblador y el procedimiento para hacer uso de los servicios del sistema.

2.5 Preguntas de revisión

1. Todo programa en ensamblador del XSPIM (o por lo menos todo programa utilizado en XSPIM) debe tener definido el símbolo .globl main a partir de cuya dirección comenzará a ejecutarse. ¿Qué directiva del ensamblador debe utilizarse siempre que se declara dicho símbolo?
2. ¿Qué directivas deben anteceder siempre a un grupo de instrucciones en un programa en ensamblador del MIPS?
3. En la arquitectura MIPS, cada palabra (word) de memoria se forma con 32 bits. Si en cierta porción de un programa se encuentra el código

```
.data
.byte       3 #byte b
.word       5 #palabra w
```

¿En qué byte de la memoria, respecto a la localidad donde se almacena el byte b, se comienza a almacenar la palabra w al ejecutarse el programa?

4. En términos de la pregunta anterior, ¿Cómo debe modificarse el segmento de código para que la palabra w se comience a almacenar en el byte "siguiente" de la memoria? El significado de "siguiente" depende de si el sistema es big- o little-endian.
5. Tomando como modelo el programa syscalls.s, escribe la versión en ensamblador del MIPS del ya tradicional programa que imprime la frase *Hola mundo*. Guarda el programa en el archivo hola.s.

Sesión 3

Programando en lenguaje ensamblador del MIPS R2000

3.1 Objetivo General

Introducir los conceptos fundamentales de la programación en lenguaje ensamblador del procesador MIPS R2000.

3.2 Objetivos específicos

Al terminar esta sesión de laboratorio:

- Tendrás claras las nociones básicas de la arquitectura basada en el conjunto de instrucciones del procesador MIPS R2000.
- Sabrás usar las instrucciones para realizar las operaciones aritméticas de suma y resta.
- Sabrás usar las instrucciones de carga y almacenamiento de datos
- Sabrás usar la seudo instrucción move
- Conocerás los distintos modos de direccionamiento en la arquitectura MIPS

3.3 Lenguaje ensamblador

Los programas en lenguaje ensamblador se caracterizan por ser "Orientados a líneas", esto es, el ensamblador los traduce una línea a la vez. El ensamblador reconoce cuatro tipos de líneas:

- Líneas vacías, que constan sólo de espacios en blanco y que son ignorados por el ensamblador.
- Líneas de etiqueta, que consisten de un identificador seguido de dos puntos (":"). Los identificadores deben comenzar con una letra (o un guión bajo) que puede estar seguida de cualquier número de letras, números o guiones bajos.
- Líneas de directivas, que forman con una etiqueta opcional y una directiva del ensamblador seguida posiblemente de argumentos.
- Líneas de instrucciones, en las que una etiqueta opcional es seguida por el nombre de una instrucción del ensamblador y sus operandos.

Como habrás notado en los programas de las sesiones anteriores, cada línea puede concluir con un comentario. Los comentarios comienzan con el carácter "#", a partir del cual el ensamblador ignora los restantes caracteres en la línea.

3.3.1 Etiquetas

En un programa en lenguaje ensamblador, una etiqueta es simplemente un nombre para una localidad o dirección de memoria. En el MIPS cada dirección es un valor de 32bits, de manera que las etiquetas no son más que valores de 32bits cuando se las usa en programas de ensamblador.

3.4 La CPU y los coprocesadores del MIPS

La unidad entera del MIPS (la CPU) cuenta con 32 registros de propósito general que almacenan valores enteros de 32bits. Los nombres de los registros van desde $0 hasta $31, y tienen nombres alternativos según se muestra en la tabla siguiente.

Número	Simbólico	Función
$0	$zero	siempre contiene cero
$1	$at	utilizado por assembler, no lo uses
$2-$3	$v0, $v1	valores de retorno de subrutina
$4-$7	$a0-$a3	argumentos de subrutina
$8-$15	t0-$t7	registros temporarios
$16-$23	$s0-$s7	registros subrutina de subrutina, tienen que ser guardados por una función llamada antes de que puedan ser usados
$24,$25	$t8, $t9	registros temporarios, pueden ser sobrescritos por registros reservados del manejador interrupt/trap de subrutina, no usar puntero global, usado para acceder a variables estáticas y externas
$26,$27, $28	$k0, $k1, $gp	
$29	$sp	stack pointer
$30	$s8/$fp	registro de subrutina, utilizado comúnmente como frame pointer
$31	$ra	dirección de retorno

Además de por la CPU, los procesadores MIPS están formados por una colección de coprocesadores que realizan tareas auxiliares u operan sobre tipos de datos, como los números de punto flotante. En XSPIM se simulan dos coprocesadores. El coprocesador 0 maneja las excepciones, las interrupciones y el sistema de memoria virtual, aunque los detalles sobre este último se omiten. El coprocesador 1 conforma la unidad de procesamiento de números de punto flotante.

3.5 Instrucciones para suma y resta

El número natural de operandos para una operación como la suma es tres: los dos números que se suman y el lugar donde se recibe la suma. Con el propósito de lograr que el hardware se mantenga simple, en la arquitectura MIPS la mayoría de las instrucciones aritméticas y lógicas tienen exactamente tres operandos.

A diferencia de los programas en lenguaje de más alto nivel, los operandos de las instrucciones aritméticas en ensamblador del MIPS no pueden ser variables, sino solamente registros. Es tarea de los compiladores el asociar eficientemente las (posiblemente numerosas) variables de los programas con los (relativamente pocos) registros con que cuanta la arquitectura MIPS.

En la siguiente tabla se muestran las instrucciones del ensamblador del MIPS que sirven para sumar y restar.

Instrucción	Ejemplo	Significado	Comentarios
add	add $1, $2, $3	$1=$2+$3	3 operandos; datos en registros
subtract	sub $1, $2, $3	$1=$2-$3	3 operandos; datos en registros

3.6 Instrucciones de transferencia de datos

Como en MIPS las operaciones aritméticas sólo ocurren entre registros, el lenguaje ensamblador debe incluir instrucciones que transfieren datos entre la memoria y los registros. Para referirse a una palabra en memoria, una instrucción debe proveer su dirección. La memoria puede considerarse como un enorme arreglo unidimensional y las direcciones (empezando en cero) son simplemente los índices a este arreglo. Como no sólo las palabras, sino también los bytes son útiles en muchos programas, la arquitectura MIPS direcciona a los bytes individuales; cuando se dice "Memory[i]", se esta haciendo referencia al i-ésimo byte de la memoria, y no a la i-ésima palabra de la memoria.

La instrucción de transferencia que mueve datos de la memoria a los registros se llama load (carga) y su contraparte, que transfiere datos de los registros a la memoria, se llama store (almacena). El formato y significado de ambas instrucciones se presenta en la siguiente tabla.

Instrucción	Ejemplo	Significado	Comentarios
load word	lw $1, 100($2)	$1= Memory[$2 +100]	Datos de la memoria a los registros
store word	sw $1, 100($2)	Memory[$2 + 100]=$1	Datos de los registros a la memoria

Como puede verse, el formato de de las instrucciones load y store es idéntico: el nombre de la instrucción seguido del registro a ser cargado (almacenado), la dirección inicial de memoria y, finalmente, un registro que contiene el desplazamiento, a partir de la dirección inicial, del elemento que será almacenado (cargado).

Ejemplo 3.1
Supongamos que A es un arreglo de palabras (números de 32bits) y que el compilador ha asociado las variables g y h con los registros $17 y $18, respectivamente. Supongamos también que en el registro $19 se encuentra almacenado el valor 4 x i y que A comienza a almacenarse en la dirección Astart. La expresión en java g= h + A[i] se traduce a ensamblador del MIPS como

```
lw $8, Astart($19) #el registro $8 recibe temporalmente a A[i]
add $17, $18, $8 #g=h + A[i]
```

3.7 La pseudo instrucción move

La pseudo instrucción move es usada simplemente para copiar el valor almacenado en un registro a otro registro, como se explica en la siguiente tabla.

Instrucción	Ejemplo	Significado	Comentarios
move	move $1, $2	$1=$2	Copia de datos entre registros

3.8 Modos de direccionamiento

La MIPS es una arquitectura de carga/almacenamiento, lo que significa que sólo las instrucciones load y store tienen acceso a la memoria. Las instrucciones que realizan cálculos operan sólo sobre valores en registros. La máquina MIPS llana (bare) sólo provee el tipo de direccionamiento de memoria c(rx) que vimos en las instrucciones lw y sw (ver la tabla anterior de load y store) y que usa la suma del valor inmediato (i.e. constante) c y el almacenado en el registro xx como la dirección. La máquina virtual MIPS, que es la usamos al programar en ensamblador, nos provee los modos de direccionamiento para instrucciones de carga y almacenamiento que se presenta en la siguiente tabla.

Formato	Cálculo de la dirección
(registro)	contenido del registro
inm	inmendiato
inm(registro)	inmediato + contenido del registro
símbolo	dirección del símbolo (definido por una etiqueta)
símbolo ± inm	dirección del símbolo ± inmendiato
símbolo ± inm(registro)	dirección del símbolo ± (inmendiato + contenido del registro)

Ejemplo 3.2
Supongamos que en el registro $t1 se encuentra el valor 0x10000000 y que el símbolo S se encuentra almacenado en la memoria a partir de la dirección 0x10000000. Las siguientes instrucciones son equivalentes:

```
lw $t0, ($t1)
lw $t0, 0x10000000
lw $t0, 0($t1)
lw $t0, S
lw $t0, S + 0
lw $t0, S - 0($0)
```

3.9 Preguntas y ejercicios de revisión

1. Supongamos que A es un arreglo de palabras y que el compilador ha asociado a la variable h con el registro $17. Supongamos también que en el registro $19 se encuentra almacenado el valor 4 x i y que A comienza a almacenarse en la dirección Astart. ¿Cuál es el código en ensamblador del MIPS que corresponde a la siguiente expresión en java? A[i] = h - A[i];

2. ¿Por qué en el ejercicio anterior y en el ejemplo 3.1 se considera el valor 4 x i y no simplemente el valor i?

3. Haz un programa en ensamblador (pqrs.s) que imprima como salida la línea (p + q) - (r + s) = 18
Usa las líneas de código

```
        .data
p:      .word 13 #p=13
q:      .word 20 #q=20
r :     .word 6 #r=6
s :     .word 9 #s=9
```

para comenzarlo.

4. Consulta la documentación de XSPIM y describe brevemente todas las variables de las instrucciones de suma, resta, carga, almacenamiento y movimiento de datos de tipo entero.

5. Supongamos que el ensamblador del MIPS no tuviera dada explícitamente una manera de copiar datos entre registros. Explica cómo podrías emular (con las otras instrucciones que conoces) esta operación.

Sesión 4

Implementando estructuras de control en MIPS

4.1 Objetivo General

Cubrir la implementación de estructuras de control usando el conjunto de instrucciones del MIPS

4.2 Objetivos específicos

Al terminar esta sesión de laboratorio:

- Habrás estudiado los distintos tipos de bifurcaciones de MIPS
- Habrás aprendido a implementar los enunciados if en MIPS
- Habrás aprendido a implementar ciclos en MIPS
- Habrás aprendido a implementar enunciados switch en MIPS

2.3 Estructuras de control

Lo que distingue a las computadoras de las calculadoras simples es la habilidad de tomar decisiones. Basándose en los datos de entrada y en los valores que van calculando, las computadoras pueden ejecutar distintos grupos de instrucciones.

En los lenguajes de programación de más alto nivel la toma de decisiones se representa comúnmente a través de los enunciados (statements) if combinados, a veces, con enunciados goto.

2.4 El enunciado if en MIPS

MIPS incluye instrucciones para la toma de decisiones similares a un anunciado if con un goto: la instrucción

beq $8, $9, L1

significa "ir a(seguir ejecutando desde) la dirección etiquetada L1 si el valor en el registro $8 es igual al valor en el registro $9". beq significa branch equal (bifurca si es igual).

Ejemplo 4.1

En el siguiente segmento de código en C, f, g, h, i y j son variables:

```
if( i == j) goto L1;
f = g + h;
L1:      f = f - i;
```

Asumiendo que las cinco variables corresponden a los registros del $16 al $20, el código MIPS compilado es:

```
beq $19, $20, L1
add $16, $17, $18
L1:   sub $16, $16, $19
```

Las instrucciones pueden etiquetarse porque, dado que los porgramas también se almacenan en memoria, tienen asociada una dirección.

Ejemplo 4.2

Los compiladores frecuentemente crean bifurcaciones y etiquetas aún cuando no aparezcan en el código original. Usando las mismas variables y registros del ejemplo anterior, el código en C:

```
if( i == j)
   f = g + h;
else
   f = f - i;
```

se traduce a MIPS como:

```
bne $19, $20, Else
add $16, $17, $18
j Exit
Else: sub $16, $16, $19
Exit:
```

En el ejemplo anterior, la segunda instrucción MIPS realiza la parte "then" del if y la cuarta corresponde al "else". Para evitar la cuarta instrucción cuando i == j debemos "saltar" hasta la etiqueta Exit, lo cual nos lleva a una clase de bifurcación no condicional que instruye a la máquina para que siempre bifurque hacia otra dirección. Para distinguir entre bifurcaciones condicionales y no condicionales, en MIPS esta última clase se denomina jump (salto), y se abrevia como j.

En la siguiente tabla se resumen las instrucciones de bifurcación y salto.

Instrucción	Ejemplo	Significado	Comentarios
branch on equal	beq $1, $2, L	if($1 == $2)goto L	Prueba de igualdad y bifurcación
branch on not equal	bne $1, $2, L	if($1 != $2) goto L	Prueba de desigualdad y bifurcación
jump	j 1000	goto 1000	Salto a la dirección destino
jump register	j $31	goto $31	Para enunciados switch

4.5 Ciclos en MIPS

La toma de decisiones es importante tanto para escoger entre dos alternativas (como en los enunciados if) como para iterar un cálculo (como en los ciclos). De hecho, las mismas instrucciones de ensamblador se usan como piezas fundamentales en ambos casos.

Ejemplo 4.3

Este es un ciclo en C:

```
Loop: g = g + A[i];
i = i + j;
if (i != h) goto Loop;
```

Supongamos que A es un arreglo de 100 palabras y que el compilador asocia las variables g, h, i y j con los registros del $17 al $20, respectivamente. Supongamos que el arreglo comienza en Astart y que en el registro $10 se encuentra almacenado un 4.

El código MIPS correspondiente a este segmento en C es:

```
Loop: mult $9, $9, $10
      lw   $8, Astart($9)
      add $17, $17, $8
      add $19, $19, $20
      bne $19, $18, Loop
```

Como en el cuerpo de ciclos se modifica el valor de i, debemos multiplicar este valor por 4 en cada iteración. En una sesión posterior de laboratorio revisaremos la instrucción mult.

Ejemplo 4.4

Por supuesto, los programadores acostumbran no escribir ciclos con gotos, de manera que es tarea de los compiladores el traducir ciclos tradicionales a la versión apropiada del lenguaje ensamblador. El segmento de código en C

```
while ( A[i] == k)
        i = i + j;
```

se traduce a MIPS como

```
Loop: mult $9, $9, $10
        lw $8, $Astart($9)
        bne $8, $21, Exit
        add $19, $19, $20
        j Loop
Exit:
```

Bajo el supuesto de que i, j y k corresponden a los registros, $19, $20 y $21, de que el arreglo A empieza en la dirección Astart, y de que el registro $10 contiene un 4.

4.6 El enunciado switch en MIPS

La mayoría de los lenguajes de programación tienen un enunciado case o switch que permite seleccionar una de muchas alternativas dependiendo de un sólo valor. Una manera de implementar un switch es como una secuencia de if-then-else. Sin embargo, en algunas ocasiones las alternativas pueden ser codificadas eficiententemente como una tabla de alternativas de secuencias de instrucciones, de manera que el programador sólo necesite indexar la tabla de direcciones de salto (jump address table) y saltar luego a la secuencia apropiada. Para tal efecto, en MIPS se incluye la instrucción jr (jump register), que representa un salto incondicional a la dirección contenida en el registro que dicha instrucción tiene por argumento.

Ejemplo 4.5

El siguiente bloque de código en Java escoge entre cuatro alternativas dependiendo de si el valor de k es 0, 1, 2 ó 3.

```
switch(k){
        case 0: f = i + j; break;
        case 1: f = g+ h; break;
        case 2: f = g - h; break;
        case 3: f = i -j; break;
}
```

Suponiendo que las seis variables f a k corresponde a los seis registros $16 a $21, que en el registro $10 contiene un 4, y que las cuatro palabras en memoria comenzando en la dirección JumpTable tienen las direcciones correspondientes a las etiquetas L0, L1, L2 y L3 respectivamente. El código MIPS correspondiente es

```
Loop: mult $9, $19, $21
        lw    $8, JumpTable($9)
        jr    $8
```

```
L0:    add  $16, $19, $20
       j    Exit
L1:    add  $16, $17, $18
       j    Exit
L2:    sub  $16, $17, $18
       j    Exit
L3:    sub  $16, $19, $20
Exit:
```

4.5 Preguntas y ejercicios de revisión

1. Existen seis condiciones relativas entre los valores de dos registros. Suponiendo que la variable i corresponde al registro $19 y que la variable j corresponde al $20, ¿Cómo se traducen a ensamblador del MIPS los siguientes enunciados en C?
   (a) if (i == j) goto L1;
   (b) if (i != j) goto L1;
   (c) if (i <j) goto L1;
   (d) if (i <= j) goto L1;
   (e) if (i > j) goto L1;
   (f) if (i >= j) goto L1;
2. Las instrucciones slt, beq y bne se usan en MIPS, junto con el valor constante del registro 0, para crear todas las condiciones relativas de la pregunta anterior. Averigua el significado de la instrucción slt y explica cómo implementarías la pseudo instrucción blt (branch on less than).
3. La instrucción beq $2, $3, L1 compara el contenido de los registros $2 y $3 y bifurca hacia L1 si on iguales. Desafortunadamente no existe ninguna instrucción para comparar $2 con un valor inmediato como 14. Escribe una secuencia de instrucciones de MIPS que bifurquen hacia L1 si $2 es igual a 14. Nota: las respuestas con más de dos instrucciones se considerarán erróneas.
4. La traducción del segmento de C del ejemplo 4.4 utiliza tanto una bifurcación condicional como una no condicional durante cada iteración. Sólo compiladores muy pobres producen código con este gasto en saltos. Rescribe el código en ensamblador de manera que se use sólo un tipo de bifurcación durante cada iteración. Si el número de iteraciones sobre el ciclo es 10, ¿cuál es el número de instrucciones ejecutadas antes y después de la optimización?
5. El siguiente programa trata de copiar palabras desde la dirección en el registro $4 a la dirección en el registro $5, contando el número de palabras copiadas en el registro $2. El programa termina de copiar cuando encuentra una palabra igual a 0. No es necesario preservar el contenido de los registros $3, $4 y $5. La palabra nula (igual a 0) debe copiarse pero no contarse.

   ```
   Loop: lw   $3, 0($4) # lee la siguiente palabra de la fuente
         addi $2, $2, 1 #incrementa el número de palabras contadas
         sw   $3, 0($5) #escribe en el destino
         addi $4, $4, 1 #avanza el apuntador a la siguiente fuente
         addi $5, $5, 1 #avanza el apuntador al siguiente destino
         bne  $3, $0, Loop #cicla si la palabra copiada != 0
   ```

   Hay varios bugs en este programa MIPS. Corrígelos y escribe la versión libre de errores del programa.

Sesión 5

Representando, ensamblando y desensamblando instrucciones en MIPS

5.1 Objetivo General

Entender la manera en que se representan las instrucciones que son ejecutadas por los procesadores MIPS.

5.2 Objetivos específicos

Al terminar esta sesión de laboratorio:

- Sabrás cómo se representan los distintos tipos de instrucciones en máquina con arquitectura MIPS
- Habrás aprendido el significado de cada uno de los campos de las instrucciones en MIPS
- Habrás construido parte de un desensamblador de MIPS.

5.3 Introducción

Los humanos estamos acostumbrados a utilizar números en base 10 la mayor parte del tiempo (¿Qué otras bases usamos?) pero en computación, dado que los números se mantienen en los circuitos de las computadoras como grupos de señales altas o bajas, usamos números binarios para representarlos.

5.4 Representando instrucciones en MIPS

Las instrucciones también se forman con series de señales electrónicas alto/bajo y por ello pueden ser representadas con números base dos. De hecho, cada parte, o campo, de una instrucción puede considerarse como número individual.

Al fijar el tamaño de todas las instrucciones en 32 bits, los diseñadores de la arquitectura MIPS tuvieron que recurrir a distintos formatos de instrucción para las distintas clases de instrucciones. Estos formatos se muestran en la siguiente tabla.

Formato	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	Comentarios
R	Op	rs	rt	rd	shamt	funct	Para instrucciones aritméticas
I	Op	rs	rt	dirección/inmediato			Para transferencias, bifurcaciones y operaciones con inmediatos
J	Op	dirección destino					Para saltos

Los campos de una instrucción MIPS en formato R son:

- op: operación de la instrucción
- rs: primer registro operando
- rt: segundo registro operando
- rd: registro destino; recibe el resultado de la operación
- shamt: magnitud de corrimiento (shift amount)
- funct: función; selecciona la variante de la operación en el campo op

5.5 Código de máquina MIPS

El valor de los campos (en decimal) para las instrucciones que hemos revisado durante las sesiones anteriores de laboratorio, esto es, sus códigos máquina, se muestran en la tabla siguiente.

Instrucción	Formato	Ejemplo						Comentarios
add	R	0	2	3	1	0	32	add $1, $2, $3
addi	I	8	2	1	100			addi $1, $2, 100
sub	R	0	2	3	1	0	34	sub $1, $2, $3
lw	I	35	2	1	100			lw $1, 100($2)
sw	I	43	2	1	100			sw $1, 100($2)
beq	I	4	1	2	100			beq $1, $2, 100
bne	I	5	1	2	100			bne $1, $2, 100
slt	R	0	2	3	1	0	42	slt $1, $2, $3
j	J	2	10000					j 10000
jr	R	0	31	0	0	0	8	jr $31
jal	J	3	10000					jal 10000

5.6 Preguntas y ejercicios de revisión

1. Implementa en java (o en C) el método (o función)

public parse_format(int instruction){}

y los métodos ahí pedidos, de la clase Instruction_Parse_type.

```
public class Instruction_Parse_type{

        private char formatType;
        private byte op, rs, rt, rd, shamt;
        private short inmmediate_or_address;
        private long target;
        private byte funct;

   public Instruction_Parse_type(){
        //Inicializo mis variables
```

```
    }

    public void Print(){//o privado si quieren
        //Imprime los valores de los miembros privados y decodificados
        // de la instrucción dada
    }

    public parse_format(int instruction){
        //prototipo de función, recibo la instrucción en formato hexadecimal
        //de 8 dígitos la instrucción en formato [0x]XXXXXXXX
    }

    //algunas funciones o variables auxiliares
    //extras deberán ser privadas
        public void Realiza_todo(){

        }

    public static void main(String argv[]){
                Instruction_Parse_type Instr = new Instruction_Parse_type();
                Instr.Realiza_todo();
        }
}
```

Donde la clase contiene:

```
private char formatType;
private byte op, rs, rt, rd, shamt;
private short inmmediate_or_address;
private long target;
private byte funct;
```

Dicho método analiza gramaticalmente (parses) y separa a una instrucción MIPS en todos sus posibles campos. La función NO determina en qué formato se encuentra una instrucción.

Por ejemplo, para la instrucción 0xa3b4902f la función fija los valores: op, 0x28; rs, 0x1d; rt, 0x14; rd, 0x12; shamt, 0x00; funct, 0x2f; inmediate_or_addres, 0x902f; y target, 0x3b4902f. Nota que algunos de los campos se traslapan.

Escribe un método principal que lea de la entrada estándar números hexadecimales que representan instrucciones(usando el formato 0x seguido de 8 caracteres hexadecimales), los pase como argumento a parse_format y despliegue, a través del método Print, los valores de los campos, tanto en base 10 como en base 16.

La clase Instruction_Parse_type es un elemento básico en la construcción de un desensamblador de MIPS.

2. Considera el siguiente programa MIPS:

```
.data
A: .word 1, 3,5,7,9,11,13,15
.text
.globl main
main:
addi $t1, $0, 0 #$t1=0 (1)
addi $t2, $0, 11 #$t2=11 (2)
loop:
lw $t3, A($t1) #$t3= A[i] (3)
beq $t3, $t2, done #if($t2 == $t2) goto done (4)
sw $0, A($t1) #A[i] = 0 (5)
addi $t1, $t1, 4 #i++ (6)
j loop #goto loop (7)
done:
```

Suponiendo que A se refiere a la dirección 1000 (base 10), que loop etiqueta a la dirección 40000(base 10) y que done corresponde a la dirección 40020 (base 10), escribe (con números hexadecimales) el código ensamblado para las 7 instrucciones del programa.

Sesión 6

Manipulando bits

6.1 Objetivo General

Presentar las operaciones para manipulación de bits provistas en MIPS

6.2 Objetivos específicos

Al terminar esta sesión de laboratorio, serás capaz de escribir programas en MIPS que usen:

- las operaciones lógicas con bits
- las operaciones de corrimiento

6.3 Introducción

Los patrones de bits no tienen ningún significado inherente; pueden representar lo mismo enteros con signo, enteros sin signo, número de punto flotante o instrucciones. Hablando en términos de lenguaje ensamblador, lo que un patrón de bits representa depende solamente de lo que una instrucción realice con sus bits.

6.4 Operaciones Lógicas con bits

La tabla de abajo resume las operaciones lógicas en MIPS. Como las demás operaciones de manipulación de datos, existen dos formatos de instrucción para la mayoría de las operaciones lógicas: en el primero los dos operandos fuente son registros(instrucción tipo R) y en el segundo un registro y el otro es un valor inmediato (instrucción tipo I).

Al fijar el tamaño de todas las instrucciones en 32 bits, los diseñadores de la arquitectura MIPS tuvieron que recurrir a distintos formatos de instrucción para las distintas clases de instrucciones. Estos formatos se muestran en la siguiente tabla.

Instrucción	Ejemplo	Significado	Comentario
and	and $1, $2, $3	$1 = $2 & $3	3 registros, Y lógico
and inmediate	andi $1, $2, 100	$1 = $2 & 100	Y lógico registro/constante
or	or $1, $2, $3	$1 = $2 | $3	3 registros, O lógico
or inmediate	ori $1, $2, 100	$1 = $2 | 100	O lógico registro/constante
not or	nor $1, $2, $3	$1 = ~($2 |$3)	No O lógico
not	not $1, $2	$1 = ~ $2	No lógico

exclusive or	xor $1, $2, $3	$1 = $2 ^ $3	3 registros, O exclusivo lógico
exclusive or inmediate	xori $1, $2, 100	$1 = $2 ^ 100	O exclusivo registro/constante

6.5 Operaciones de corrimiento

Aunque las primeras computadoras se enfocaban en las operaciones con palabras completas, pronto quedó claro que resulta útil operar sobre campos de bits dentro de una palabra o, inclusive, con bits individuales. Se tenían que considerar instrucciones que simplificaran, entre otras cosas, el empaquetamiento y desempaquetamiento de bits en palabras.

Una clase de tales operaciones se denomina corrimientos (shifts). Mueven todos los bits en una palabra a la izquierda o a la derecha, llenando los bits vacíos con ceros. Por ejemplo, si el registro $16 contuviera:

00000000000000000000000000001101

después de un corrimiento a la izquierda de 8 bits, contendría

00000000000000000000110100000000

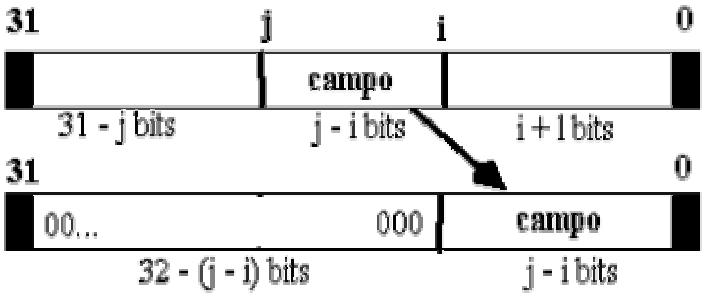
Las variantes de las operaciones (lógicas) de corrimiento en MIPS, shift left logical y shift right logical, se describen en la tabla siguiente.

Instrucción	Ejemplo	Significado	Comentarios
shift left logical	sll $1, $2, 10	$1 = $2 << 10	SHL registro/constante
shift left logical variable	sllv $1, $2, $3	$1 = $2 << $3	SHL, 3 registros operandos, se consideran sólo los 5 bits menos significativos de $3
shift right logical	srl $1, $2, 10	$1 = $2 >>10	SHR registro/constante
shift rightlogical variable	srlv $1, $2, $3	$1 = $2 >> $3	SHR, 3 registros operandos, se consideran sólo los 5 bits menos significativos de $3

6.6 Preguntas y ejercicios de revisión

1.En MIPS la operación lógica NOT se realiza a través de la pseudo instrucción not. Existen por lo menos dos formas distintas de implantar esta operación como secuencias de instrucciones: una usando la instrucción xor y otra usando la instrucción nor. Explícalas.

2. Algunas arquitecturas tienen instrucciones específicas para extraer un campo arbitrario de un registro de 32 bits y colocarlo como los bits menos significativos de otro registro (con todos los demás bits puestos a cero), como se muestra en la siguiente figura:



Escribe una secuencia de instrucciones MIPS para extraer un campo del registro $16 (según la figura) y colocarlo en el registro $17 usando los valores constantes i=7 y j=19.

4. Escribe un programa en MIPS (count.s) que cuente e imprima el número de bits encendidos (o sea, unos) en una localidad de memoria n (una variable) establecida en el área de datos. Inicializa el valor de n a 0xabc970fe.

Sesión 7

Haciendo aritmética en MIPS

7.1 Objetivo General

Cubrir los conceptos relativos a las operaciones aritméticas en MIPS

7.2 Objetivos específicos

Al terminar esta sesión de laboratorio,

- Serás capaz de escribir programas en MIPS que usen las operaciones aritméticas de suma y resta en MIPS
- Entenderás cómo son representados los números negativos en las computadoras
- Entenderás cómo son representados los números de punto flotante en las computadoras

7.3 Números negativos

Sabemos que las palabras en MIPS miden 32 bits, por lo que podemos, de manera natural, representar los números enteros positivos desde 0 (32 ceros) hasta 4,294,967,295 (32 unos). Sin embargo, al necesitarlos en un gran número de programas, debemos también considerar a los números negativos encontrando un método que nos permita representar a ambas clases de números de manera eficiente.

La convención más ampliamente adoptada para representar números enteros con signo es la representación complemento a dos, en la que

la mitad positiva de los números, desde 0 hasta 2^31 - 1, usan su representación binaria normal, mientras que los siguientes patrones de bits, desde 100...000 hasta 111...111, representan a los números negativos desde -2^31 hasta -1.

Esta representación asegura que x + (-x) = 0, salvo para x = -2^31(=100...000), que no tiene un número positivo correspondiente. Además, la notación complemento a dos tiene la ventaja de que los números negativos se distinguen de los positivos ene l bit más significativo, el "bit de signo".

Derivadas de la construcción de la notación complemento a dos, surgen las siguientes observaciones prácticas:

- para encontrar el negativo del número simplemente se cambia cada 0 en 1, y cada 1 en 0, y finalmente, se suma 1 al resultado.

- para convertir un número complemento a dos representado en n bits a una representación con más de n bits simplemente se toma el bit de signo y se replica hacia la izquierda llenando los n bits que sean necesarios. Este proceso se conoce como extensión de signo.

7.4 Suma y resta

La suma es justamente lo que se espera que sea: los dígitos binarios se suman bit a bit de derecha a izquierda, como se hace a mano. La resta usa la suma: el sustraendo simplemente se niega antes de ser sumado al minuendo.

La complejidad que representa la suma en las computadoras es la posibilidad de que la suma sea demasiado grande como para poder ser representada apropiadamente. No importa qué clase de notación se use, siempre es posible que la suma de dos números de 32 bits sea demasiado grande como para ser representado en 32 bits. A este evento se le conoce como sobre flujo (overflow).

A pesar de que al sumar dos enteros sin signo puede ocurrir un sobre flujo, usualmente éste se ignora debido a que tales números son generalmente usados para representar direcciones de memoria que son, a diferencia de los números naturales, finitas.

En MIPS se tienen dos clases de instrucciones aritméticas: por un lado están add, addi y sub, que operan con números complemento a dos y que causan excepciones al haber sobre flujo; y por otro lado están addu (add unsigned), addiu (add inmediate unsigned) y subu (substract unsigned), que nos generan excepciones con los sobre flujos.

7.5 Preguntas y ejercicios de revisión

1.Convierte los siguientes números base 10 a sus representaciones binarias complemento a dos de 32 bits.

512, -1023, -4000000

2. Convierte los siguientes números binarios en notación complemento a dos de 32 bits a sus representaciones en base 10.

11111111111111111111111000001100
11111100111111100010000010000000000
01111111111110001010000000000001111

3. Supongamos que tenemos dos palabras de 32 bits almacenadas en sendos registros. Esboza un criterio para decidir cuál de los números representados por las palabras es mayor si los números se representan en notación complemento a dos. ¿Funcionaría el mismo criterio si las palabras representaran enteros sin signo? (NOTA: no deberá incluirse código MIPS en la respuesta)

4. MIPS ofrece dos versiones de la comparación set-on-less-than para manejar enteros con y sin signo: set on lesss than (stl) y set on less than inmediate (stli) trabajan con enteros con signo, mientras que set on less than unsigned (stlu) y set on less than inmediate unisgned (stliu) operan con enteros sin signo. Supongamos que en el registro $16 se encuentra el número binario 11111111111111111111111111111111 y que el registro $17 contiene al número binario 00000000000000000000000000000001. ¿Cuál es el contenido del registro $8 y $9 después de las siguientes instrucciones?

slt $8, $16, $17
sltu $9, $16, $17

5. Sean A y B dos números binarios de 32 bits en notación complemento a dos. Determina las condiciones bajo las cuales A+B y A-B producirán un sobre flujo.

Sesión 8

Implementando un simulador del MIPS (1)

8.1 Objetivo General

Construir un simulador de un sólo ciclo del MIPS.

8.2 Objetivos específicos

Al finalizar esta sesión de laboratorio habrás escrito en JAVA, el código necesario para implementar archivo de registros (register file) y la ALU mostrados en la Figura 5.7 del libro *Computer Organization & Design: The hardware/software interface* de Patterson y Hennessy.

8.3 Ejercicios de revisión

En el directorio correspondiente a esta sesión de laboratorio coloca los archivos ALU.java y Registers.java.

1. Implementa todos los métodos públicos declaradas en la clase ALU. No debes agregar ninguna otra función o variable pública, pero puedes agregar funciones o variables que sean privados. Implementa el método main, un programa para probar la ALU. El programa deberá pedir al usuario dos valores de entrada y una operación, llamar a las funciones adecuadas en ALU, realizar la operación y reportar el resultado. Usa como guía los comentarios dados en los archivos.

2. Implementa todos los métodos públicos declarados en la clase Registers.java. No debes agregar ninguna otra función o variable pública, pero puedes agregar funciones o variables que sean privados. Implementa el método main, un programa para probar Registers. El programa debe permitir al usuario seleccionar el método de Registers que desea llamar preguntándole, además, los parámetros con que el método debe ser llamado. Usa como guía los comentarios dados en los archivos.

ALU.java

```
//Agregar bibliotecas de otras clases
//import java.*

public class ALU{
        private long ALURightInput;
        private long ALULeftInput;
        private long ALUResult;
        private long ALUZero;
        private long ALUOperation;
```

```
public ALU(){
        ALURightInput=0;
        ALULeftInput=0;
        ALUResult=0;
        ALUZero=0;
        ALUOperation=0;
}

/**
  Regresa el valor actual de ALUResult
*/
public long getALUResult(){

        return 1;
}

/**
  Regresa el valor actual de ALUZero
*/
public long get ALUZero(){

        return 1;
}

/**
  Fija la operacion de la ALU. Como efecto lateral evalua
  la operacion con las entradas actuales de la ALU
  y fija a ALUResult y ALUZero.

  Los valores de las entradas de control de la ALU son:
    control      funcion
    000          AND
    001          OR
    010          add
    110          substract (left - right)
    111          set on less than
  Nota: Los valores debe fijarse en valores entre 0 y 7.
  Si el código de control no es valido, no debe cambiarse ALUOperation
  o reevaluar las salidas de la ALU.
*/
public void setALUOperation(long val){

}

/**
  Fija el valor de la entrada izquierda de la ALU en 'val'.
  Como efecto lateral, reevalúa las salidas de la ALU como en
  SetALUOperation.
 */
public void setALULeftInput(long val){
```

```
        }

        /**
       Fija el valor de la entrada derecha de la ALU en 'val'.
       Como efecto lateral, reevalúa las salidas de la ALU como en
       SetALUOperation.
        */
      public void setALURightInput(long val){

      }

        /**
          Programa para probar la ALU.
         */
        public static void main(String args[]){

        }
}
```

Registers.java

```
import java.io.*;

public class Registers{

        private long reg[];
        private long writeData;
        private long readData1;
        private long readData2;
        private long readRegister1;
        private long readRegister2;
        private long writeRegister;

        public Registers(){
                reg=new long[32];
                writeData=0;
                readData1=0;
                readData2=0;
                readRegister1=0;
                readRegister2=0;
                writeRegister=0;
        }

        /**
          Provoca que el valor de 'writeData' sea almacenado en 'reg[writeRegister]'.
         */
        public void assertRegWrite(){

        }
```

```
    /**
       Exporta los valores de los 32 registros (conun formato agradable)
       a un archivo.
    */
    public void displayRegisters(String archivo){

    }

    /**
       Devuelve el valor del registro indicado por el valor en 'readRegister1'
    */
    public long getReadData1Value(){

    }

    /**
   Devuelve el valor del registro indicado por el valor en 'readRegister2'
    */
public long getReadData2Value(){

}

    /**
       Fija 'readReadRegister1' en 'val'. Como efecto lateral,
       fija 'readData1' al valor del registro 'readRegister1'.
     */
    public void setReadRegister1(long val){

    }

    /**
   Fija 'readReadRegister2' en 'val'. Como efecto lateral,
   fija 'readData2' al valor del registro 'readRegister2'.
    */
public void setReadRegister2(long val){

}

    /**
       Fija 'writeData' en 'val'
     */
    public void setWriteData(long val){

    }

    /**
       Fija 'writeRegister' en 'val'
    */
    public void setWriteRegister(long val){

    }
```

```
	/**
	  Aqui va el programa principal
	 */
	public void execute(){

	}

	/**
	  Metodo Main, no tocar
	 */
	public static void main(String argv[]){
		Registers registros=new Registers();
		registros.execute();

	}
}
```

Sesión 9

Implementando un simulador del MIPS (2)

9.1 Objetivo General

Construir un simulador de un sólo ciclo del MIPS.

9.2 Objetivos específicos

Al finalizar esta sesión de laboratorio habrás escrito en JAVA, el código necesario para implementar las estructuras de control y memoria necesarios para construir el simulador del MIPS.

9.3 Ejercicios de revisión

En el directorio correspondiente a esta sesión de laboratorio coloca los archivos Control.java, DataMemory.java e InstructionMemory.java.

1. Implementa todos los métodos públicos declaradas en la clase Control.java. No debes agregar ninguna otra función o variable pública, pero puedes agregar funciones o variables que sean privados. Completa el método main, un programa basado en un menú para probar los distintos métodos relacionados con la estructura de control. (¡Esta sesión requiere haber concluido satisfactoriamente la sesión 5 del laboratorio!).

2. Implementa todos los métodos públicos declarados en la clase DataMemory.java. No debes agregar ninguna otra función o variable pública, pero puedes agregar funciones o variables que sean privados. Completa el método main, un programa basado en un menú para probar los distintos métodos relacionados con la memoria de datos.

3. Implementa todos los métodos públicos declarados en la clase InstructionMemory.java. No debes agregar ninguna otra función o variable pública, pero puedes agregar funciones o variables que sean privados. Completa el método main, un programa basado en un menú para probar los distintos métodos relacionados con la memoria de instrucciones.

Instruction_Parse_type.java (Sesión 5)

Control.java

```
import java.io.*;

public class Control{

        private int ALUOp;
        private int ALUSrc;
```

```
private int Branch;
private int MemRead;
private int MemWrite;
private int MemtoReg;
private int Jump;
private int RegDst;
private int RegWrite;
private int currentInstruction;

private Instruction_Parse_type decodedInstruction;
//Para decodificar la instruccion leida
//Ver la sesion 5

/**
  En el constructor inicializo mis miembros
  a cero y crear el objeto Instruction_Parse_type
*/
public Control(){

}

/**
  Llama a Parse_format de la clase Instruction_Parse_format con
  'instruction' y fija a 'decodedInstruction' con el valor
  de la instruccion decodificada.
*/
public void DecodeInstruction(int instruccion){

}

/**
  Saca los valores de las líneas de control a un archivo
*/
public void dumpControlLines(String the_file){

}

/**
  Regresa la operacion para la ALU de acuerdo con la tabla 5.15
  del libro de Patterson y Hennessy. Los bits del campo de función
  se obtienen del campo 'funct' del objeto 'decodedInstruction'
  y la ALUOp es el valor actual de la línea de control 'ALUOp'
*/
public void getALUControl(){

}

/**
  Fija las líneas de control segun el campo 'op' de 'decodedInstruction'
  como se indica en la figura 5.27 del libro de Patterson y Hennessy.
 */
```

```
        public void setControl(){

        }

        /**
          Regresa el valor dado por los 16 bits bajos de 'p' al ser extendido
          en signo (sign-extend) a 32 bits.
        */
        public int signExtend(int p){

        }
        /**
          programa para probar las líneas de Control
        */
        public void make(){

        }

        public static void main(String argv[]){
                Control control=new Control();
                control.make();
                //El main es intocable
        }
```

DataMemory.java

```
import java.io.*;

public class DataMemory{
        const DATAMEMORYSIZE= 100;

        private long dataMem[];
        private long Address;
        private long WriteData;
        private long ReadData;

        /**
          Inicio mis variables a cero
        */
        public DataMemory(){

        }

        /**
          Escribe el valor de 'writeData' a 'dataMem'['Address'/4].
          Nota: la memoria es direccionable por palabras, pero la dirección es para
          una memoria direccionable por bytes.
         */
        public void assertMemWrite(){
```

```
	}

	/**
	  Regresa el valor de 'ReadData'
	 */
	public long getReadData(){

	}

	/**
	  Fija 'Address' a 'val' y 'ReadData' a 'dataMem['Address'/4].
	*/
	public void setAddress(long val){

	}

	/**
	  fija 'WriteData' a 'val'
	*/
	public void setWriteData(long val){

	}

	/**
	  Lee 'numWords' palabras de 'the_file' y las pone en las
	  primeras 'numWords' entradas de 'dataMem'.
	  Las palabras en el archivo se representan en hexadecimales

	  Devuelve true si tuvo existo, false en caso contrario.
	*/
	public boolean initializeDataMemory(String the_file, int numWords){

	}

	/**Saca el contenido de 'dataMem' desde 'startAddress' hasta 'endAddress'
	  al archivo 'the_file' en un formato legible.
	*/
	public void dumpDataMemory(String the_file, int startAddress, int endAddress){

	}

	public void make(){
		//realizo todo aqui
	}

	public static void main(String argv){
		DataMemory dataMemory=new DataMemory();
		dataMemory.make();
		//El main es intocable
	}
}
```

InstructionMemory.java

```
import java.io.*;

public class InstructionMemory{

	const INSTRUCTIONMEMORYSIZE 100;

	private int instructionMem[];//de tamanio INSTRUCTIONMEMORYSIZE
	private int readAddress;
	private int Instruction;

	/**
	  Inicio mis variables en cero y creo el arreglo de tamanio INSTRUCTIONMEMORYSIZE
	*/
	public InstructionMemory(){

	}

	/**
	  Saca los valores desde 'instructionMem'['startAddress'] hasta
	  'instructionMem'['endAddress'] al archivo 'the_file'
	*/
	public void dumpInstructionMemory(String the_file,int startAddress,int endAddress){

	}

	/**
	  Regresa el valor actual de 'Instruction'
	*/
	public int getInstruction(){

	}

	/**
	  Lee 'numInst' MIPS de 'the_file' y las pone en las primeras 'numInst' entradas de
	  'instructionMem'. Las instrucciones se representan en hexadecimales.
	  Regresa true si tuvo exito, false en caso contario.
	*/
	public boolean initializeInstructionMemory(String the_file, int numInst){

	}

	/**
	  Fija el valor de 'instructionReadAddress' a 'val' y fija
	  'Instruction' a 'instructionMem'['val'/4].
	  Nota: Este es un modulo de memoria direccionable por palabras,
	  pero 'val' es para una memoria direccionable por bytes.
	*/
	public void setAddress(int val){
```

```
	}

	public void make(){
		//Aqui se hace todo
	}

	public static void main(String argv[]){
		InstructionMemory InMe=new InstructionMemory();
		InMe.make();
		//El main es intocable
	}
}
```

Sesión 10

Implementando un simulador del MIPS (3)

10.1 Objetivo General

Construir un simulador de un sólo ciclo del MIPS.

10.2 Objetivos específicos

Al finalizar esta sesión de laboratorio habrás escrito en JAVA, el código necesario para implementar un simulador de un solo ciclo del MIPS.

10.3 Ejercicios de revisión

Copia al directorio correspondiente a esta sesión los archivos que utilizaste en las sesiones 8 y 9. Coloca allí también los archivos Execute.java, MIPSsim.java, instructions.dat y dataMemory.dat.

Execute.java

```
import java.awt.*;
import java.awt.event.*;

public class Execute extends Frame implements ActionListener{

        private ALU alu;
        private Control control;
        private DataMemory dataMem;
        private InstructionMemory InstMem;
        private Registers registers;

        private long PC;

        /**
           Inicio todos mis objetos y PC a cero
        */
        public Execute(){

        }

        /**
           Ejecuta la instrucción cuya dirección esta en PC
        */
        public void executeInstruction(){

        }

        /**
```

```
            Regresa el valor actual del PC
        */
        public long getPC(){

        }

        /**
            Fija el PC a val
        */
        public void setPC(long val){

        }
}
```

MIPSsim.java

```
import java.awt.*;
import java.awt.event.*;

public class MIPSsim extends Frame implements ActionListener{



        public MIPSsim(){

        }

        public static void main(String argv[]){

        }

}
```

Incorpora Execute.java a Control.java

1. Implementa todos los métodos públicos nuevos en Control.java. El método más importante, executeInstruction, ejecuta la instrucción cuya dirección esté en el PC. Debes acceder memoria de instrucciones, a la de datos, a los registros, a la ALU y las estructuras de control sólo a través de los método públicos escritos en las sesiones 8 y 9. Si no lo hiciste, fija setControl, la línea de control ALUOp, de acuerdo con el opcode de las instrucciones en la Figura 5.15 del libro de Patterson y Hennessy. Revisa la Sesión 5.

2. Completa en MIPSsim.java un programa que tome cuatro argumentos de la línea de comando:

(a) El nombre del archivo para iniciar la memoria de instrucciones.
(b) El número de instrucciones a leer
(c) El nombre del archivo para inicializar la memoria de datos.
(d) El número de datos a leer.

El programa debe:

(a) Iniciar la memoria de instrucciones usando el archivo pasado en la línea de comando.
(b) Iniciar la memoria de datos usando el archivo pasado en la línea de comando.

(c) En una ventana, desplegar un menú con las siguientes operaciones:

i. Ejecutar la siguiente instrucción (llamar a executeInstruction).
ii. Desplegar la memoria de datos en algún lugar de la ventana (llamar a dumpDataMemory).
iii. Desplegar los registros en algún lado de la ventana (llamar a displayRegisters).

Para probar el simulador se pueden emplear los archivos instructions.dat y datamemory.dat.