UNIVERSIDAD DE SANTIAGO DE CHILE
FACULTAD DE INGENIERÍA
DEPARTAMENTO DE INGENIERÍA INFORMÁTICA

Laboratorio Nº 2

Integrantes:	Gabriel Gaete L.
Curso:	Organización de Computadores
Sección	0-L-2
Profesor:	Nestor González & Leonel Medina
Ayudantes:	Ricardo Álvarez & Matías Fuentes

30 de Julio de 2018

Tabla de contenidos

Índice de figuras

1. Introducción

El presente informe pretende explicar cómo fue solucionado un problema que involucra el concepto de camino de datos, pipeline y paralelismo a nivel de instrucción, utilizando el lenguaje de programación estructurado C bajo el estándar de ANSI-C.

En la asignatura de "Organización de computadores", se puede ver que una de las partes esenciales de los computadores corresponde al camino de datos, o "datapath", el cual consiste en el camino que recorren las diferentes señales eléctricas a lo largo de un conjunto de unidades funcionales que permiten llevar a cabo la ejecución de las distintas instrucciones entregadas a un computador. Dado que el ser humano siempre busca que las cosas tengan un mejor rendimiento, es que en los datapath existen técnicas para lograrlo, tales como *pipeline*, *multiple-issue*, entre otras. Ante esto surge la interrogante ¿cómo se ejecutan las instrucciones con estas tecnologías?

1.1. Enunciado del problema

El problema a resolver es el siguiente. A partir de dos archivos de entrada, uno con instrucciones en MIPS, el otro con los valores iniciales de cada registro, determinar en tres archivos de salida, los posibles *hazards* (pérdidas de control o de datos), junto con su respectiva representación del *pipeline*. Por otro lado, un tercer archivo de salida debe indicar cómo sería el procesamiento de las instrucciones en un procesador *multiple-issue*.

1.2. Motivación

La principal motivación para llevar a cabo este laboratorio es comprender la forma en que se ejecutan las instrucciones utilizando *pipeline* y un procesador *multiple-issue*.

1.3. Objetivos

1.3.1. Objetivos generales

Desarrollar un programa capaz de simular un código en MIPS ejecutado tanto en *pipeline* como en *multiple-issue*.

1.3.2. Objetivos específicos

Entender la importancia de un procesador que permite la ejecución de instrucciones tanto en un *pipeline* como *multiple-issue*, y además entender el cómo se puede disminuir la cantidad de ciclos dentro de un procesador al utilizar estas herramientas.

1.4. Herramientas

Las herramientas utilizadas en la implementación de la solución a este laboratorio son las siguientes:

1. Lenguaje de programación C.
2. Uso de estructuras.
3. Uso de memoria dinámica.

1.5. Estructura del informe

El presente escrito detalla a continuación conceptos que fueron la base para llevar a cabo la solución de este problema, tales como *pipeline*, *datapath*, etc, con el fin de dar a conocer los conceptos investigados, para posteriormente llegar a una descripción de la solución.

2. Marco teórico

2.1. Conocimientos generales previos

A modo de manual, cabe mencionar que para una mayor facilidad a la hora de compilar y ejecutar los códigos fuentes del programa, se ha facilitado un *makefile*. Para su utilización, se tienen los siguientes comandos:

- *make all* permite la compilación el código fuente.
- *make run* permite ejecutar el programa.
- *make clean* permite eliminar el archivo ejecutable, luego de la utilización del programa.

Por último, cabe considerar que los archivos de entrada DEBEN estar en la misma carpeta que el programa ejecutable. Además, se recomienda que estos archivos sean de extensión .txt

2.2. Camino de datos o *datapath*

El camino de datos corresponde a la ruta que sigue un dato de entrada dentro de un procesador, hasta que se obtiene un output. Está compuesto de unidades funcionales que pueden manejar información de forma conjunta. Comúnmente, dentro de un microprocesador, los caminos de datos están compuestos principalmente por *registros de instrucción*, una *unidad aritmético lógica*, *multiplexores*, una *unidad de control* y una *memoria*, tanto para instrucciones como para el almacenamiento de información.

2.3. Pipeline

El pipeline corresponde a una técnica que permite implementar paralelismo a nivel de instrucciones dentro de un solo procesador. Ésta técnica consiste principalmente en mantener ocupada a cada parte del procesador, a través de una división de las instrucciones entrantes en una serie de pasos secuenciales, que se realizan por diferentes unidades del procesador que trabajan de manera simultánea. Ésta técnica aumenta el rendimiento del CPU a una velocidad de reloj determinada, pero en algunos casos puede aumentar la latencia debido a la sobrecarga adicional del proceso del pipeline en sí.

2.4. Multiple-issue

Ésta técnica permite ir más allá del pipeline, permitiendo ejecutar más de una instrucción por cada ciclo de reloj. Esto se logra replicando componentes del procesador con el fin de tener múltiples instrucciones en cada etapa del pipeline. Un procesador multiple-issue debe lidiar con los posibles riesgos de control y de datos. La siguiente imagen muestra un ejemplo de una ejecución de instrucciones en un procesador multiple-issue.

Instruction type	Pipe stages							
ALU or branch instruction	IF	ID	EX	MEM	WB			
Load or store instruction	IF	ID	EX	MEM	WB			
ALU or branch instruction		IF	ID	EX	MEM	WB		
Load or store instruction		IF	ID	EX	MEM	WB		
ALU or branch instruction			IF	ID	EX	MEM	WB	
Load or store instruction			IF	ID	EX	MEM	WB	
ALU or branch instruction				IF	ID	EX	MEM	WB
Load or store instruction				IF	ID	EX	MEM	WB

Figura 1: Ejemplo de Multiple-Issue

3. Desarrollo

En primer lugar, el problema presentado en el laboratorio fue dividido en varios sub-problemas, con el fin de ser resuelto por partes. Estos sub-problemas, fueron clasificados de la siguiente forma:

1. Almacenamiento y estructuras para el manejo de datos.
2. Identificación de hazards.
3. Procesamiento del pipeline.
4. Procesamiento del multiple-issue.

3.1. Almacenamiento y estructuras para el manejo de datos

El primero de los sub-problemas a resolver, se trató de encontrar una forma de almacenar los datos leídos de los archivos, puesto que el lenguaje de programación C no es un buen lenguaje para procesar texto. Los datos leídos, a partir del archivo en que se indican los valores de cada registro, fueron almacenados dentro de listas enlazadas, en la que cada nodo de estas listas, corresponde a una linea de este archivo.

```
typedef struct linesNode{
        char* line;
        struct linesNode* next;
}LinesNode;

typedef struct listOfLines{
        LinesNode* first;
        LinesNode* last;
        int length;
}ListOfLines;
```

Figura 2: Estructura que almacena las líneas de los archivos

Por otra parte, cada línea de instrucción MIPS es llevada a otra lista enlazada, en la que se separa cada parte de la instrucción en sí, tales como el nombre, los registros que utiliza, el tipo de instrucción, etc.

```
typedef struct instructionNode{
        char* instruction;
        char* firstOperand;
        char* secondOperand;
        char* thirdOperand;
        char* label;
        int offset;
        int type;

        struct instructionNode* next;
}InstructionNode;

typedef struct instructionLinkedList{
        InstructionNode* first;
        InstructionNode* last;
        int length;
}InstructionLinkedList;
```

Figura 3: Estructura que almacena las instrucciones del archivo.

3.2. Identificación de hazards

Una de las posibles estrategias que se puede utilizar a la hora de identificar los hazards dentro de un código MIPS, incluye la implementación de registros 'especiales', denominados *buffers*, los cuales permiten almacenar los valores de un ciclo. Dentro de estos buffers, se comparan ciertos elementos, tales como el registro de destino de una instrucción, con los registros de "lectura"de la instrucción siguiente. La detección de hazard de datos se guían por estos casos:

```
if (EX/MEM.RegWrite=1)
   and (EX/MEM.Mux_RegDst != 0)
   and (EX/MEM.Mux_RegDst = ID/EX.Rs)

if (EX/MEM.RegWrite=1)
   and (EX/MEM.Mux_RegDst != 0)
   and (EX/MEM.Mux_RegDst = ID/EX.Rt)

if (MEM/WB.RegWrite=1)
   and (MEM/WB.Mux_RegDst != 0)
   and (MEM/WB.Mux_RegDst = ID/EX.Rs)

if (MEM/WB.RegWrite=1)
   and (MEM/WB.Mux_RegDst != 0)
   and (MEM/WB.Mux_RegDst = ID/EX.Rt)
```

Figura 4: Condiciones que deben cumplirse para que exista un hazards de datos.

Por otro lado, existe un hazard especial, el cual produce una espera (stall). Este hazard puede verse en la siguiente figura:

```
if (ID/EX.MemRead=1)
   and ((ID/EX.Rt=IF/ID.Rs)
   or (ID/EX.Rt=IF/ID.Rt))
      stall the pipeline
```

Figura 5: Condiciones que deben cumplirse para que exista un hazards de datos.

Siguiendo estas condiciones, el algoritmo implementado en este laboratorio se basa en una comparación directa entre instrucciones, haciendo una "traza"del programa, es decir, realizando las distintas instrucciones y verificando si se cumplen o no las condiciones anteriormente dadas.

3.3. Procesamiento del Pipeline

Siguiendo la misma estrategia que para la identificación de hazards, el procesamiento del pipeline se hace mediante una estructura de datos un tanto especial. Primero, recordar que las instrucciones se procesan dentro de una lista enlazada. Para llevar a cabo el pipeline, estas instrucciones se van ingresando a una cola, sin embargo, si la instrucción ingresada produce un hazard en que se haga necesario ingresar una espera (cabe decir, un hazard que no pueda solucionarse directamente mediante la implementación del forwarding, o de un adelantamiento de datos), se realiza un *pop*, (simulando que la cola se convierte en una pila), se inserta una instrucción especial, la cual simula una espera, para posteriormente volver a ingresar la instrucción que produjo esta espera (asumiendo que el forwarding esta vez sí la soluciona).

3.4. Procesamiento del Multiple-Issue

Por último, el último sub-problema que se debió solucionar, fue el procesamiento del multiple-issue, sin embargo, por diferentes motivos, no logró ser implementado en este laboratorio.

4. Experimentos a realizar

Para probar el programa realizado, se tienen los siguientes archivos de prueba:

4.1. Pruebas realizadas

4.1.1. Prueba 1

```
addi $t0, $zero, 2
add $t7, $t2, $t2
addi $t1, $zero, 2
addi $t1, $t0, 3
subi $t2, $t1, 1
mul $t2, $t2, $t2
lw $t2, 8($s0)
add $t7, $t2, $t2
addi $sp, $sp, -8
sw $t7, 0($sp)
lw $t6, 0($sp)
addi $t1, $t0, 1
```

Figura 6: Archivo de entrada 1, contiene las instrucciones MIPS.

```
$zero 0
$at 0
$v0 0
$v1 0
$a0 0
$a1 0
$a2 0
$a3 0
$t0 0
$t1 1
$t2 0
$t3 1
$t4 0
$t5 0
$t6 0
$t7 0
$s0 0
$s1 0
$s2 0
$s3 0
$s4 0
$s5 0
$s6 0
$s7 0
$t8 1
$t9 0
$k0 0
$k1 0
$gp 0
$sp 0
$fp 0
$ra 0
```

Figura 7: Archivo de entrada 2, contiene el valor de los registros.

4.2. Resultados obtenidos

4.2.1. Resultados prueba 1

CICLOS	DATOS	CONTROL
1		
2		
3		
4		X
5		X
6		X
7		
8	$t6	
9		
10		
11		

Figura 8: Archivo de salida1, contiene la detección de hazards.

CICLOS	IF	ID	EX	MEM	WB
1	addi	-	-	-	-
2	beq	addi	-	-	-
3	mul	beq	addi	-	-
4	div	NOP	beq	addi	-
5	sub	div	NOP	beq	addi
7	mul	NOP	NOP	NOP	beq
8	addi	mul	NOP	NOP	NOP
9	div	addi	mul	NOP	NOP
10	sub	div	addi	mul	NOP
11	-	sub	div	addi	mul
12	-	-	sub	div	addi
13	-	-	-	sub	div
14	-	-	-	-	sub

Figura 9: Archivo de salida2, contiene la traza del pipeline.

4.3. Análisis de resultados

Se ha logrado que el programa sea capaz de identificar los distintos hazards presentes dentro de un código MIPS, tanto hazards de control como de datos, para luego simular su ejecución en un procesador que permite *pipeline*, el cual, debido a la correcta identificación de hazards, es capaz de insertar esperas en los lugares adecuados. Por otra parte, no se ha logrado procesar el código MIPS dentro de un procesador *multiple-issue*.

5. Conclusiones

Los objetivos indicados al inicio de este informe se han cumplido de manera parcial, puesto que se ha logrado solucionar una parte del laboratorio, correspondiente a la generación del archivo con el código MIPS en un procesador pipelined, bajo el estándar de programación ANSI C. Sin embargo, no se ha logrado llevar a cabo la implementación de un procesador multiple-issue.

A pesar de lo dicho anteriormente, se está conforme con los resultados obtenidos, y de los conocimientos que éste laboratorio ha entregado.

Para futuras experiencias, se espera mejorar la investigación previa a la implementación de una solución, además de una perfeccionamiento en los algoritmos, los cuales se espera que sean más compactos e intuitivos.

A pesar de no haber logrado una parte del laboratorio, el realizar la parte de pipeline ha sido fructífero, sobretodo desde el punto de vista de los conocimientos asimilados. Tener la idea de cómo se optimiza un conjunto de instrucciones dependiendo de cómo funciona un procesador, puede ser una herramienta muy eficaz a la hora de comparar diferentes arquitecturas.