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

Laboratorio 2 de Organización de Computadores

Integrantes:	Cristian Espinoza
Curso:	Organización de Computadores
Sección	L1
Profesor:	Daniel Wladdimiro
Ayudante:	Sebastian Pinto-Agüero

22 de Octubre de 2017

Tabla de contenidos

Índice de figuras

1. Introducción

En el presente documento se ha pedido a los alumnos de la asignatura Organización de computadores que presenten el trabajo realizado en el laboratorio numero dos, el cual al realizarlo conlleva el aprendizaje de los contenidos en forma practica y teórica simultáneamente.

Como lo confirma la teoría en cátedra, en la organización de computadores lo esencial es el camino de datos, que representa el camino que recorren las señales eléctricas a través de las distintas unidades fundamentales, con el objetivo de realizar una instrucción en especifica. *Pipeline* permite que múltiples instrucciones estén traspaladas en la ejecución, sin embargo, el uso de *pipeline* puede generar ciertos problemas denominados *hazard de datos*, los cuales se pueden solucionar con el uso de ciertas técnicas.

La solución del presente laboratorio se abarca con la herramienta de división en sub-problemas, con el fin de poder abarcar en su totalidad cada una de sus dificultades. Esta es capaz de leer dos archivos de entrada, el primero de ellos contiene las instrucciones del programa en *MIPS*, el segundo contiene los valores de los registros, al primer archivo leído se le aplica *pipeline* de 5 etapas considerando los *hazard de control, hazard de datos, forwarding, NOPs*, predicción del branch (*Branch no taken*), la solución propuesta termina entregando dos archivos de salida, el primero de ellos entrega la traza del programa ejecutado y el segundo archivo contiene los distintos *Hazard* que fueron encontrados dentro de la ejecución del programa. Destacar que el alcance que se presenta es el cumplimiento del desarrollo completo del laboratorio.

1.1. Motivación

El laboratorio propuesto por los docentes de la asignatura organización de computadores, permitirá desarrollar los conocimientos entregados en cátedra respecto a la aplicación de la técnica de *pipeline* , *hazar de control*, *hazar de datos*, *forwarding*, *NOPs*, para así mejorar el rendimiento de los alumnos dentro de la asignatura.

1.2. Objetivos

- **Objetivo general**: Aprender, aclarar y profundizar el concepto de *pipeline*.
- **Objetivos específico**: Realizar un programa capaz de solucionar el problema expuesto en el enunciado, ocupando las técnicas contempladas en clases, para así lograr el objetivo general.

1.3. Datos

- El archivo uno contiene las instrucciones del programa en *MIPS*.
- El archivo dos contiene los valores de los registros.

1.4. Herramientas

Las herramientas utilizadas para la realización de este laboratorio son:

- *Sublime text 3*. (C)
- Gcc version 6.2.0.
- *Sharlatex*.

1.5. Estructura del informe

Por ultimo, en este informe se presenta un marco teórico necesario para el entendimiento de todo el desarrollo del programa, además se cuenta con un desarrollo que explica cómo se realizo, las herramientas y técnicas utilizadas para el desarrollo de este.

2. Marco teórico

A continuación, se dejarán en claro los conceptos que se deben manejar para una adecuada lectura del informe.

2.1. Pipeline

Técnica de implementación, donde múltiples instrucciones son traslapadas en ejecución. Es un paralelismo invisible al programador. Guzmán et al. (2017c)

2.2. Hazard

Cuando una instrucción no puede ser ejecutada en el siguiente ciclo de reloj, esta situación es llamada *Hazard*. De los cuales tenemos:

- ***Hazard* de datos**: Ocurre cuando un conjunto de instrucciones no pueden ejecutarse en el mismo ciclo de reloj, dado que los datos que son necesarios en alguna de ellas no se encuentran listos para ser utilizados por otras instrucciones posteriores. Guzmán et al. (2017c)
- ***Hazard* de control**: Ocurre debido a que en alguna instrucción se debe tomar una decisión. Guzmán et al. (2017c)

2.3. Forwarding

Es una técnica de hardware utilizada para solucionar *hazards* de datos, en la cual los valores almacenados en los *buffers* intermedios, pueden ser traspasados hacia otras instrucciones. De tal manera, que pueden ser obtenidos antes de que sean escritos en algún registro. Guzmán et al. (2017c)

No siempre la utilización de *forwarding* será suficiente para resolver un *hazard de datos*. En el caso de que se requiera leer un registro que escribe una instrucción *lw*, las condiciones anteriores no funcionarían, por lo cual se agrega un *NOPs*. La instrucción *NOP* es una instrucción que no realiza ninguna operación que cambie algún estado dentro de los elementos del camino de datos. Guzmán et al. (2017c)

3. Desarrollo

Para efectos de facilitar las funcionalidades del programa, se crea un arreglo constante, el cual contiene el nombre de todo los registros que existen, el cual se puede observar en la figura 1.

```
const char *registroMars[32] = {"$zero","$at","$v0","$v1","$a0","$a1","$a2","$a3","$t0","$t1",
"$t2","$t3","$t4","$t5","$t6","$t7","$s0","$s1","$s2","$s3","$s4","$s5","$s6","$s7","$t8","$t9",
"$k0","$k1","$gp","$sp","$fp","$ra"};
```

Figura 1: Registro de *MIPS*.

3.1. Estructuras

- ***Buffer***: Esta estructura se encarga de almacenar toda la información que contienen las distintas etapas del *pipeline*, por lo tanto, esta estructura, contiene la totalidad de los registros que están presenten en el camino de datos, las lineas de control, los valores tomados por los *mux* y 3 variables *int* que representan los distintos estados que puede estar la instrucción (*NOP, Branch no taken*, salto *Jump*). En la figura 2, se aprecia la estructura.

```
typedef struct Buffer
{
    int estado;
    int estado1;
    int estado2;
    Instruccion* instruccion;
    LineaDeControl* lineaDeControl;
    char* muxRegDs;
    char* register1;
    char* register2;
    int readData1Id;
    int readData2Id;
    int signoExtendido;
    int ALU;
    char* address;
    char* writeDataMem;
    int readData1Mem;
    char* writeRegister;
    int writeDataWb;
    int zero;
    int aluResult;
    int addPc;
    char* rs;
    char* rt;
    char* rd;
    int posRegistro;
}Buffer;
```

Figura 2: *Buffer*

- **Instrucción**: Esta estructura se encarga de almacenar toda la información de las instrucciones que se leen desde el archivo de texto, por lo tanto, esta estructura, la cual tiene 3 variables de tipo *char*, las cuales representan a los registros que utilizaría la

instrucción *(rs, rt y rd)*. Además, se tienen dos variables de tipo *int*, la primera corresponde al valor inmediato y la segunda variable corresponde al PC, que indica la posición de la instrucción. Por otro lado, se tiene una variable de tipo *char* contiene el nombre de la instrucción leída. Por ultimo, contiene una variable de tipo *Linea de control* la cual cumple el rol de guardar la linea de control de la instrucción. En la figura 3, se aprecia la estructura.

```
typedef struct Instruccion
{
    char *instruccion;
    char *rt;
    char *rs;
    char *rd;
    int inmediato;
    int PC;
    LineaDeControl* lineaDeControl;

}Instruccion;
```

Figura 3: Instrucciones

- **Información**: Esta estructura es la mas relevante, porque es la que contiene a las anteriores, y además almacena todo que se necesita teóricamente, como también de manera practica. Contiene una variable de tipo *int* que contiene la cantidad de instrucciones que contiene el programa de MIPS y por otro lado, contiene un arreglo de *int* que se encarga de guardar los valores que toma cada uno de los registros en el transcurso del programa, un arreglo de *int* que se encarga de representar la memoria del programa que se utiliza para las funciones *'lw' y 'sw' (load word y store word)*, dos variables de tipo *char* que se encarga de guardar si se presentar algún *hazard* en la ejecución (*hazard de control y/o hazar de datos*), y por ultimo, contiene un puntero a la estructura *Buffer*, la cual se encargará de almacenar los valores ontenido en cada una de las etapas del *pipeline*. En la figura 4, se aprecia la estructura.

```
typedef struct Informacion
{
    Instruccion* instrucciones;
    int registros[32];
    int cantidadDeInstrucciones;
    Label etiqueta[100];
    int memoria[1000];
    Buffer* buffer;
    char* hazarDato;
    char* hazarControl;

}Informacion;
```

Figura 4: Información

3.2. Funciones

Las funciones se separan en tres partes, primero entre las que son necesarias para el desarrollo del programa, las que actúan directamente a las instrucciones del programa *MIPS* y por ultimo las funciones que se encargan de lectura y escritura de los archivos.

3.2.1. Funciones necesarias para el programa

Estas funciones son las que se implementaron antes de abordar el objetivo principal de la solución que es encontrar errores en las lineas del control al realizar la traza del programa.

- *void pipeLine(Informacion *informacion, char nombreSalida1[], char nombreSalida2[])*: Esta función se encarga de simular el recorrido que tendrá cada una de las instrucciones utilizando la técnica de *pipeline*, la cual consiste en desglosar cada una de ellas en 5 etapas. Estas se planifican de la siguiente manera: Se escoge esta planificación para no tener perdida de datos al momento de realizar el traspaso entre los *buffers*.
  - *WB*: Esta etapa se encarga de escribir en la memoria el nuevo valor del registro, que corresponderá al valor obtenido en la variable *writeDataWb*.
  - *MEM*: Esta etapa se encarga de guardar en *writeDataWb*, el valor obtenido desde la componente *ALU*, el cual esta guardado en la variable *aluResult*.
  - *EX*: Esta etapa se encarga de guardar en *aluResult*, la operación relacionada a la función de los registros *readData1Id y readData2Id*.
  - *ID*: Esta etapa es la que da vida al proceso de *pipeline*, ya que se encarga de guardar en el *buffer* la totalidad de los valores que ocupara la instrucción durante las 5 etapas.
  - *IF*: Esta etapa se encarga de reconocer el tipo de instrucción que se va a planificar para el siguiente ciclo.
- *void hazarDatoMEM(Informacion* informacion)* y *void hazarDatoEx(Informacion* informacion)*: Estas funciones son las encargadas de detectar si existe *hazar de control*

en la etapas *EX* y/o *MEM* del *pipeline*, con el objetivo de realizar un *forwarding*, para que el dato tenga el valor actual.

- *int isnop(Informacion* informacion)*: Esta función se encarga de verificar si es necesario agregar un *NOP*, además del *forwarding*, este suceso ocurre cuando tenemos una instrucción *lw* que este utilizando un registros *rt*, el cual será requerido por una instrucción de tipo R, en la siguiente instrucción planificada.

3.2.2. Funciones para cada instrucción de *MIPS*

Estas funciones cumplen con la operación que realiza la instrucción de *MIPS* que se lee desde el archivo uno.

Las instrucciones implementadas son: *add, sub, addi, subi, div, mul, beq, jump, lw, sw*. Todas las funciones mencionas tienen retorno *void*.

3.2.3. Funciones de escritura/lectura de los archivos

Estas funciones están encargadas de leer la información presente en los archivos de entrada, además de escribir la salida del programa.

- *Informacion* leerInstrucciones(char nombre[], int numeroDeLineas)*: Esta función se encarga de leer las instrucciones que contiene el archivo de MIPS, guardando la información de las etiquetas e instrucciones en la estructura *Instrucciones* que se menciono anteriormente, además se encarga de inicializar los registros en '0'. Esta función retorna un puntero a *Información*.

- *void escribirArchivoHazar(Informacion* informacion, int ciclo, char nombreArchivo[])* y *void escribirArchivoTraza(Informacion* informacion, int ciclo,char nombreArchivo[])*: Estas funciones se encargan de generar los archivos *".csv"*, que contienen la traza del programa ejecutado en el primer archivo y el detalle de los *hazard* en el segundo archivo.

4. Experimento

4.1. Resultados obtenidos

El programa consta con una interfaz, en la cual el usuario debe ingresar la opción a realizar para poner en curso el programa, además de interactuar insertando los nombre de los archivos que el programa va solicitando para su desarrollo.

En la figura 5, se muestra la salida que entrega el programa al probar el archivo de prueba numero dos:

Ciclo	IF	ID	EX	MEM	WB
1	addi $t0 $zero 2				
2	addi $t1 $zero 2	addi $t0 $zero 2			
3	beq $t0 $t1 NEXT	addi $t1 $zero 2	addi $t0 $zero 2		
4	addi $t1 $t0 3	beq $t0 $t1 NEXT	addi $t1 $zero 2	addi $t0 $zero 2	
5	subi $t2 $t1 1	addi $t1 $t0 3	beq $t0 $t1 NEXT	addi $t1 $zero 2	addi $t0 $zero 2
6				beq $t0 $t1 NEXT	addi $t1 $zero 2
7	add $t1 $t0 1				beq $t0 $t1 NEXT
8		add $t1 $t0 1			
9			add $t1 $t0 1		
10				add $t1 $t0 1	
11					add $t1 $t0 1

Figura 5: Resultado de ejemplo numero dos

4.2. Análisis de resultados

Se puede visualizar que la cantidad de ciclos en un camino de datos monociclo como se realizo en la implementación del laboratorio numero uno es notablemente mayor en comparación con los ciclos de reloj obtenidos en la implementación numero dos, en la cual se utilizo la técnica de *pipeline*.

En si la técnica de *pipeline* muestra ser una técnica bastante eficiente, ya que puede realizar la ejecución de 5 etapas simultáneamente. Considerando el manejo de los hazard de datos aplicando la técnica de *forwarding* y, *hazard de control* utilizando la predicción del *branch (branch no taken)*, los ciclos de reloj tienden a ser menores respecto a una implementación de *pipeline simple*, ya que este último presenta muchas mas esperas generando ciclos de más en la ejecución del programa.

La solución propuesta es escalable, ya que se puede agregar las instrucciones restantes al código fuente. Además, de mencionar que se implemento un algoritmo de $O(n)$.

5. Conclusiones

En este presente Laboratorio se pudo llevar a practica la teoría que se aprendió en la cátedra, pudiendo tener una visión mas integral del cómo llevar a la práctica cada uno de los elementos vistos y estudiados.

Como se ha mencionado anteriormente, se ha podido lograr la implementación en cada uno de los objetivos tanto como generales como específicos. Además, destacar que la mayor dificultad dentro del desarrollo del programa fue la organización del código para obtener la solución requerida.

Finalmente, se debe destacar que los resultados del programa fueron exitosos en las pruebas realizadas, además se implementó un algoritmo de O(n), sin embargo, se debe dejar abierta la posibilidad de posibles mejoras para dicho algoritmo. Es por esto que se espera una retroalimentación, para evitar errores durante el segundo semestre del 2017.

Bibliografía

Guzmán, A., Wladdimiro, D., Alvarez, M., Pinto-Aguero, S., and Undurraga, A. (2017a). Organizacion de computadores laboratorio 1. [Online] http://www.udesantiagovirtual.cl/moodle2/pluginfile.php?file=%2F111799%2Fmod_resource%2Fcontent%2F8%2FOC%20Lab%201.pdf.

Guzmán, A., Wladdimiro, D., and Rosas, E. (2017b). Organizacion de computadores camino de datos y unidad de control. [Online] http://www.udesantiagovirtual.cl/moodle2/pluginfile.php?file=%2F111752%2Fmod_resource%2Fcontent%2F3%2FOC_Procesador.pdf.

Guzmán, A., Wladdimiro, D., and Rosas, E. (2017c). Organizacion de computadores pipeline, datapath y hazard. [Online] http://www.udesantiagovirtual.cl/moodle2/pluginfile.php?file=%2F216491%2Fmod_resource%2Fcontent%2F1%2FOC_Pipeline.pdf.