



Laboratorio N° 2

Integrantes: Gabriel Gaete L.

Curso: Organización de Computadores

Sección 0-L-2

Profesor: Nestor González & Leonel Medina

Ayudantes: Ricardo Álvarez & Matías Fuentes

5 de Junio de 2018

Tabla de contenidos

1. Introducción	1
2. Marco teórico	2
2.1. Conocimientos generales previos	2
2.2. Camino de datos o <i>datapath</i>	2
2.3. Pipeline	3
2.4. Multiple-issue	3
3. Desarrollo	4
3.1. Almacenamiento y estructuras para el manejo de datos	4
3.2. Identificación de hazards	5
3.3. Procesamiento del Pipeline	6
3.4. Procesamiento del Multiple-Issue	7
4. Experimentos a realizar	8
4.1. Pruebas realizadas	8
4.1.1. Prueba 1	8
4.2. Resultados obtenidos	9
4.2.1. Resultados prueba 1	9
4.3. Análisis de resultados	10
4.3.1. Prueba 1	10
5. Conclusiones	11

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 programación 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 humano siempre busca que las cosas tengan un mejor rendimiento, es que en los datapath existen técnicas para lograr esto, tales como *pipeline*, *multiple-issue*, entre otras. Ante esto surge la interrogante ¿cómo se ejecutan las instrucciones con estas tecnologías?

La principal motivación para llevar a cabo este laboratorio es comprender la forma en que se ejecutan las instrucciones utilizando las técnicas de optimización de rendimiento anteriormente nombradas.

El objetivo principal de este informe es explicar como fue abordado el problema, para posteriormente llegar a una solución viable. Por otro lado, el objetivo principal del laboratorio es comprender e indentificar la forma en que las instrucciones son procesadas al implementar estrategias de optimización.

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

2. Marco teórico

2.1. Conocimientos generales previos

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*, ya sean 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 en un procesador *multiple-issue*.

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 las instrucciones MIPS, fueron almacenados dentro de listas enlazadas, en la que cada nodo de estas listas, corresponde a una línea de cada 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

Posterior a esto, cada línea de instrucción es llevada a otra lista enlazada, en la que se separa cada parte de la instrucción en si, 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), 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 si la soluciona).


```

void writePipelinedFile(InstructionLinkedList* instructions, char* filename){
    FILE* f;
    f = fopen(filename, "w");

    if (! f) exit(1);

    InstructionLinkedList* stack;
    stack = createInstructionLinkedList();

    fprintf(f, "CICLOS;IF;ID;EX;MEM;WB\n");

    InstructionNode* node;
    node = instructions->first;

    int cycle;
    cycle = 1;
    int lw;

    while (node){
        lw = 0;

        pushInstruction(stack, node);
        printInstructions(stack, f, cycle);

        if (getID(stack) && getIF(stack) && determineLwHazard(getID(stack), getIF(stack))){
            lw = 1;
            popInstruction(stack);
            pushInstruction(stack, createStall());
        }

        cycle++;
        if (lw == 0) node = node->next;
    }

    finishPipeline(stack, f, cycle);

    fclose(f);
}

```

Figura 6: Algoritmo que permite escribir en un archivo la ejecución del pipeline.

Cabe destacar que el algoritmo presentado en la Figura 6, no funciona correctamente con códigos MIPS que presenten saltos, es decir, instrucciones *beq* y/o *jump*.

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 7: Archivo de entrada 1, contiene las instrucciones MIPS.

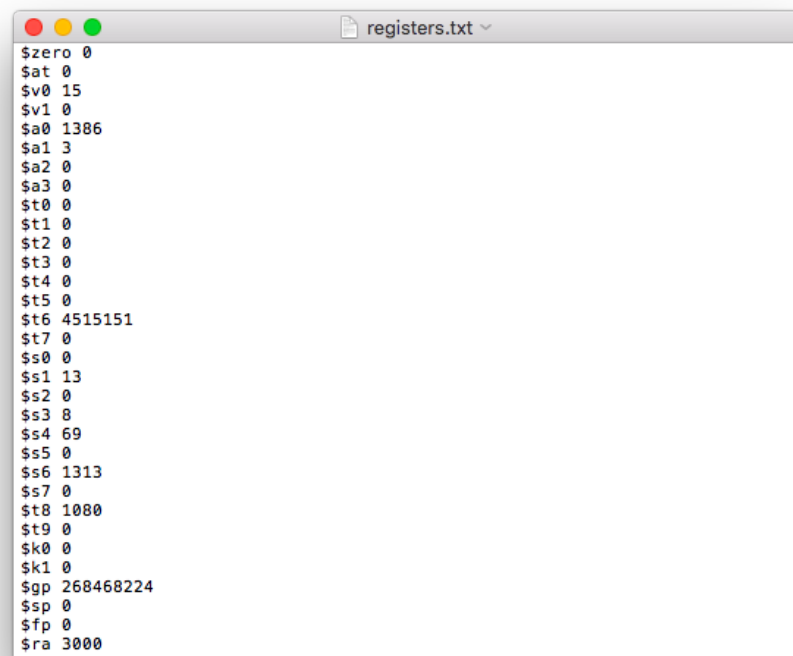
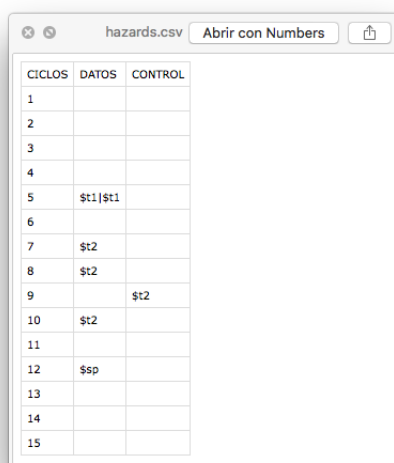


Figura 8: Archivo de entrada 2, contiene el valor de los registros.

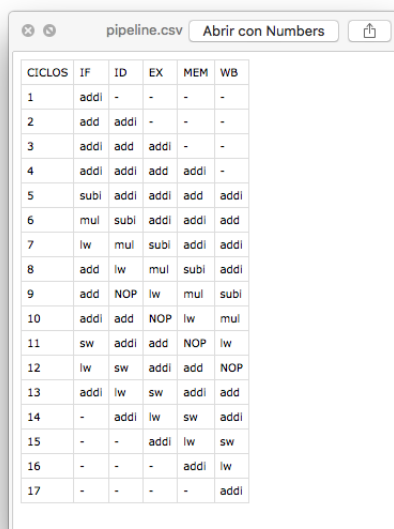
4.2. Resultados obtenidos

4.2.1. Resultados prueba 1



CICLOS	DATOS	CONTROL
1		
2		
3		
4		
5	\$t1 \$t1	
6		
7	\$t2	
8	\$t2	
9		\$t2
10	\$t2	
11		
12	\$sp	
13		
14		
15		

Figura 9: Archivo de salida1, contiene la detección de hazards.



CICLOS	IF	ID	EX	MEM	WB
1	addi	-	-	-	-
2	add	addi	-	-	-
3	addi	add	addi	-	-
4	addi	addi	add	addi	-
5	subi	addi	addi	add	addi
6	mul	subi	addi	addi	add
7	lw	mul	subi	addi	addi
8	add	lw	mul	subi	addi
9	add	NOP	lw	mul	subi
10	addi	add	NOP	lw	mul
11	sw	addi	add	NOP	lw
12	lw	sw	addi	add	NOP
13	addi	lw	sw	addi	add
14	-	addi	lw	sw	addi
15	-	-	addi	lw	sw
16	-	-	-	addi	lw
17	-	-	-	-	addi

Figura 10: Archivo de salida2, contiene la traza del pipeline.

4.3. Análisis de resultados

4.3.1. Prueba 1

Al analizar los resultados entregados por la prueba 1, se puede determinar que los hazards encontrados se ajustan a los esperados, puesto que realiza correctamente la traza del programa, e identifica de buena forma los hazards. A la hora de analizar el pipeline, se puede ver que los NOP fueron insertados en el lugar adecuado, dado que se produce un error a la hora de realizar el *load word*. Por otra parte, no fue generado el archivo de salida con las instrucciones en un procesador *multiple-issue*.

5. Conclusiones

Concluyendo, en esta experiencia de laboratorio, se permite poner en práctica el pipeline, y entender tanto los beneficios que este puede traer, como también los posibles problemas (hazards de datos y de control) y sus soluciones (forwarding, esperas, etc).

Finalmente, puesto que se ha desarrollado la fase de pipeline como se esperaba, es que este objetivo ha sido logrado, sin embargo, el objetivo de realizar un ejemplo de procesador multiple-issue no resultó como se esperaba.