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

ORGANIZACION DE COMPUTADORES: LABORATORIO 2

RAÚL OLIVARES

Profesores:
Erika Rosas
Nicolás Hidalgo
Felipe Garay

Santiago - Chile
7 de noviembre de 2016

TABLA DE CONTENIDOS

ÍNDICE DE FIGURAS

CAPÍTULO 1. INTRODUCCIÓN

Para mejorar la eficiencia del procesador se creo con el tiempo una nueva micro-arquitectura de procesador llamada superescalar, esta micro-arquitectura es capaz de ir ejecutando más de una instrucción a la vez gracias a una estructura en pipeline con diversas etapas , en cada etapa se va realizando un diferente proceso para la ejecución de una instrucción permitiendo así después de despejar una etapa, esta sea ocupada por una nueva instrucción.

Lamentablemente algunas veces la instrucción siguiente necesita valores que se cambian por la instrucción anterior y que todavía no son guardados en el registro o memoria, o no se sabe si se debe realizar la instrucción o otra ya que el branch no se encuentra listo. Estas dependencias son conocidas como hazards y son importantes para el procesador saber cuando ocurren para así adaptarse y no generar errores o malgastar recursos, haciendo que el paralelismo en las instrucciones sea posible de realizar correctamente.

1.1 OBJETIVO

Para el siguiente laboratorio se busca crear un pipeline virtual con sus 5 etapas conocidas, que puedan ejecutar las instrucciones que se le entrega en formato asambler de MIPS, entendiendo como funciona este sistema en el procesador físico y las problemáticas en realizar las instrucciones que nos son asignadas, para luego crear un detector de hazards que nos indique la linea de código donde se producen y el tipo de hazards con el que nos encontramos.

1.1.1 Objetivos específicos

- Aprender en que consisten las distintas etapas que posee el pipeline físico de un procesador y como interactúan entre si.
- Crear un sistema que simule un pipeline virtual en el computador.
- Lograr una herramienta que detecte los hazards, su tipo y ubicación en el código entregado.
- Proponer mejoras para el código entregado para que este posea menos numero de hazards.

1.2 PROBLEMA

¿Como se detectan los hazards en un pipeline?. Basado en el conocimiento entregado en clases entre cada etapa del pipeline se encuentra un buffer de datos conteniendo la información entregada en la etapa anterior y la información requerida para realizar la operación en la siguiente etapa, a través de estos buffer podremos saber si las variables que se necesitan en una instrucción están siendo utilizadas por otras instrucciones permitiendo identificar hazards de datos previamente a realizar una instrucción.

1.3 HERRAMIENTAS

Todo el laboratorio se realizara en el sistema operativo Ubuntu/Linux, en este se utilizara el lenguaje C para crear una herramienta que simule un pipeline virtual. Además se realizara en un cuaderno los recorridos

del pipeline de las distintas instrucciones para mostrar mejor como funcionan las mejoras implementadas a los distintos programas y así justificarlas.

1.4 ORGANIZACIÓN DEL DOCUMENTO

Este documento se dividirá en una serie de secciones para su mejor entendimiento y comprensión. Empezando por la Introducción donde se plantea la idea de este laboratorio y sus objetivos,el Marco teórico con todos los términos y definiciones que se necesita para desarrollar el laboratorio,el Desarrollo de como se llevo a cabo esta solución, sus justificaciones y su implementación en un programa. Una sección de Análisis donde se revisara los resultados de probar los programas entregados en lenguaje assambler y las mejoras que se realizaron a los distintos programas, seguido por ultimo con las Conclusiones que dejo este laboratorio.

CAPÍTULO 2. MARCO TEÓRICO

2.1 SUPERESCALAR

Superescalar es el término utilizado para designar un tipo de micro-arquitectura de procesador capaz de ejecutar más de una instrucción por ciclo de reloj a diferencia de un procesador escalar capaz de ejecutar solamente una instrucción por ciclo de reloj. La microarquitectura superescalar utiliza el paralelismo de instrucciones además del paralelismo de flujo, este último gracias a la estructura en pipeline. La estructura típica de un procesador superescalar consta de un pipeline con las siguientes etapas:

- Lectura (fetch).
- Decodificación (decode).
- Lanzamiento (dispatch).
- Ejecución (execute).
- Escritura (writeback).
- Finalización (retirement).

En un procesador superescalar, el procesador maneja más de una instrucción en cada etapa. leyendo la instrucción, decodificándola, obteniendo los valores de las variables que necesita la instrucción, llevando a cabo esa instrucción, para luego guardar los nuevos valores de las variables en memoria y finalmente repitiendo el proceso con una nueva instrucción.

2.2 HAZARD

Un procesador superescalar es capaz de ejecutar más de una instrucción simultáneamente únicamente si las instrucciones no presentan algún tipo de dependencia (hazard). Los tipos de hazards entre instrucciones son :

- Hazard estructural, esta ocurre cuando dos instrucciones requieren el mismo tipo unidad funcional y su número no es suficiente.
- Hazard de datos, esta ocurre cuando una instrucción necesita del resultado de otra instrucción para ejecutarse, por ejemplo R1¡=R2+R3 y R4¡=R1+5.
- Hazard de control, esta ocurre cuando una instrucción depende de una estructura de control y no se puede determinar el flujo correcto hasta la evaluación de la estructura de control, por ejemplo, if R1¡R2 then R3¡=R4+R5 else R6¡=R7+5.

2.3 BUFFERS

un búfer (del inglés, buffer) es un espacio de memoria, en el que se almacenan datos de manera temporal, normalmente para un único uso. Como se vio anteriormente el pipeline tiene distintas etapas donde se realiza una determinado operación para resolver una instrucción, entre cada una de estas etapas se encuentra un buffer, él cual almacena los datos del proceso anterior, los guarda y los envía al proceso siguiente para que estos no se pierdan, esperando que el proceso siguiente haya sido completado previamente.

2.4 INSTRUCCIONES MIPS

A continuación se describe la sintaxis y significado de las distintas instrucciones en lenguaje assambler que se de dan como necesarios de ejecutar en este laboratorio.En general el lenguaje assambler posee la siguiente sintaxis: Nombre de la instrucción Operando 1, Operando 2, Operando 3...

- Suma add: Suma las variables b y c y pone el resultado en a.

```
add a, b, c
```

- Operación lw(load worl): Copia datos desde la memoria a un registro, el formato es el nombre de la operacion seguido por el registro donde se cargara el dato, luego una constante y el registro usado para acceder a memoria.

```
lw a, 4(b)
```

- Operación sw(store worl): guarda el valor de a en memoria,accediendo a memoria en la posicion b del arreglo c

```
sw a, b(c)
```

- Multiplicación mull: multiplica b y c,y luego guarda el resultado en a.

```
mul a, b, c
```

- División div: divide b y c,y luego guarda el resultado en a.

```
div a, b, c
```

- Operación NOP: espacio de espera en que no se realiza ninguna operación en el pipeline, busca rellenar la espera de una variable que se necesita y no se encuentra lista todavia.
- Operación j: salta a la instrucción que se encuentra en la dirección que entrega
- Operacion branch: Salta a la instrucción situada en la dirección especificada(label) si los registros(a y b) cumplen determinada condicion.

```
beq a, b, label
```

2.5 REORDENA-MIENTO DE INSTRUCCIONES

Entre los distintos métodos para mejorar el paralelismo de las instrucciones en el pipeline se encuentra el reordena-miento de instrucciones, donde para evitar que se produzcan hazards de datos en el programa se busca reordenar las instrucciones que no interfieran con el resultado del programa, retrasando instrucciones que necesitan variables que no se encuentran listas todavía y adelantando operaciones que no interfieren con el programa para así disminuir los ciclos de reloj necesarios para ejecutar un programa.

```
Antes de reordenar:
lw  t0,  0( t1 )
addi  t0,  t0,  2
lw  t2,  4( t1 )


Después de reordenar:
lw  t0,  0( t1 )
lw  t2,  4( t1 )
addi  t0,  t0,  2
```

CAPÍTULO 3. DESARROLLO

A continuación se describe las decisiones y justificaciones de como se decidió realizar la implementación del pipeline virtual y como se desarrollo la herramienta dentro del programa para detectar los distintos tipos de hazards que se pueden encontrar en un programa hecho en lenguaje assambler.

3.1 REGISTROS

Los valores de los registros son guardados y manejados a través de una estructura de datos llamada registro.[en] el archivo en lenguaje C registros.c, esta estructura contiene todos las variables que ocupara el registro como valores enteros, en formato int, desde el registro t0...t9, a1..a3, v0, v1, lo y hi. Esta estructura sera manejada a través de funciones para obtener y editar los valores del registro, a partir de la función .[o]btenerI"se obtendrá el valor del registro buscado, mientras que en la función .[ed]itarïngresara el valor y el nombre del registro a modificar junto a la estructura registro y devolverá la estructura registro, con sus valores actualizados.

3.2 PIPELINE

Para la realización del pipeline virtual se decidió abstraer las distintas etapas del procesador en funciones, en cada función se realiza una operación al igual que en el pipeline físico del procesador, en la primera etapa, denominada como función ÏF"donde ingresa la instrucción y es desglosada, obteniendo la operación que debe realizar y el nombre de los registros que van a utilizar. A partir de eso los datos pasan a través de una estructura denominada "buffer"por almacenar datos temporales entre etapas igual que en el pipeline normal, desde ese buffer los datos pasan a la siguiente función denominada ÏD"son obtenidas los valores de los registros de la instrucción a partir de los registros.

```
 54  buffer IF(char *instruccion ,int n , buffer b){
302  }
303
304  //asigna valores a los registros
305  buffer ID(buffer b, registro r){
346  }
347
348  buffer EX(buffer b){
378  }
379
380
381  buffer MEM(buffer b,int memoria[]){
412  }
413
414  registro WB(buffer b, registro r){
436  }
```

Figura 3-2: Funciones pipeline

```
 4
 5  typedef struct registro
 6  {
 7      int $t0;
 8      int $t1;
 9      int $t2;
10      int $t3;
11      int $t4;
12      int $t5;
13      int $t6;
14      int $t7;
15      int $t8;
16      int $t9;
17
18      int $a0;
19      int $a1;
20      int $a2;
21      int $a3;
22      int $zero;
23      int $v0;
24      int $v1;
25      int lo;
26      int hi;
27
28  }registro;
29
30  registro editar(registro r, char* nombre , int valor){...
54  }
55
56  int obtenerI(registro r, char nombre[]){...
84  }
85
```

Figura 3-1: Estructura archivo registro

La siguiente etapa, la función .[EX]"se encarga de hacer las operaciones aritméticas de suma, resta, multiplicación o división de los registros que son ingresados, y guardar en un buffer los resultados para la siguiente etapa. En la función "MEM"se encarga de cargar o guardar el valor de un registro en la memoria, esta memoria "virtual.[es]ta representada por un arreglo implementado como ARREGLO[5000]. Por ultimo la función "WBrepresenta su misma etapa del pipeline donde los valores de los registros que han sido alterados serán guardados en los registros.

3.3 DECISIONES

A continuación se describe como se ejecutan distintas operaciones en el simulador de pipeline, como los branch y las operaciones aritméticas mul y div, entre otras operaciones.

Mul y div: Estas dos operaciones con el formato mul t1, t2, t3 son en realidad seudo operaciones , ya que representan 2 operaciones, la primera mul t2 , t3 multiplica los registros y guarda el resultado en el registro "lo", mientras con la operación mflo t1 se obtiene el valor que se encuentra en el registro "lo" se guarda en el registro t1. Como se puede apreciar, existe un hazard de datos entre estas dos operaciones , ya que mflo tiene que esperar que la operación mul cargue el resultado en el registro lo antes de poder obtenerlo. Este hazard siempre ocurrirá igualmente con la operación div, por lo que no tiene un sentido practico para este laboratorio, en que se busca crear una herramienta que detecte hazard, hacer inca-pie en el hazard de datos que producen estas 2 operaciones a lo largo del pipeline, por lo que se decidió implementar las seudo-operaciones mul y div como operaciones enteras, por ejemplo, a partir de una operación con formato mul t1, t2, t3 se multiplicaran las variables t2 y t3 en la etapa EX, y luego el resultado de esta operación pasara inmediatamente a ser guardada en el registro t1 al final de la etapa WB.

Branch: Para las operación beq, blt, bgt, empezaran por descifrarse en la etapa IF, para luego en la etapa ID obtener los valores de los registros con los que se medirá la condición, en la etapa EX se restaran estos dos registros y se obtendrá un resultado, ese resultado decidirá si se produce un salto o no a otra instrucción del código en la etapa MEM, ya que en esta etapa se medirá si la condición se ha cumplido, si esta se cumple el pipeline accionara el método para saltar a la siguiente instrucción. De igual caso la operación j, obtiene en IF la dirección donde saltara y por conveniencia se decidió por que este salto actuara igualmente en la etapa MEM como las otras instrucciones de branch.

3.4 DETECTOR DE HAZARDS

Para almacenar y enviar datos entre etapas se a implementado una estructura llamada buffer, donde se irán guardando los valores y nombres de los distintos registros que ocuparan las instrucciones. Luego se compararan los distintos buffers para ver si se producen hazards de datos, comparando si el registro que se necesita para una operación en un buffer, se va a cambiar en un buffer anterior y este no ha llegado a la etapa final donde se edita el valor del registro.

Para los hazards de control se consideran todos las instrucciones tipo branch, donde la operación siguiente a realizar no se sabe con certeza, por lo que se generara tiempos de espera para poder saber hacia donde saltara. La instrucción j para este pipeline virtual también saltara en una etapa posterior por lo que

aunque sabemos a donde saltara, esta pausa el pipeline, ya que las siguientes instrucciones en orden normal no pueden ingresar y se esperara hasta que este llegue a la etapa MEM para dar el salto, por lo que también se considerara como un hazard de control.

3.5 RESULTADOS

Para el archivo de prueba "factorial.asm.este dejo los siguientes resultados en el archivo STATEEND.txt con los valores finales que toman los registros y el archivo Hazards.txt con todos los hazards encontrados durante la ejecución del programa, con sus tipos y la linea donde se encuentra. Como ejemplo a continuación se probo el programa factorial.asm y estos fueron los resultados.

```
1  $t0: 1
2  $t1: 1
3  $t2: 120
4  $t3: 0
5  $t4: 0
6  $t5: 0
7  $t6: 0
8  $t7: 0
9  $t8: 0
10 $t9: 0
11 $a0: 0
12 $a1: 0
13 $a2: 0
14 $a3: 0
15 $v0: 0
16 $v1: 0
17 $lo: 0
18 $hi: 0
19
```

Figura 3-3: Resultados: valores de los registros pipeline

```
1  tipo: hazard de datos
2  linea: 2
3  tipo: hazard de control
4  linea: 7
5  tipo: hazard de datos
6  linea: 9
7  tipo: hazard de control
8  linea: 10
9  tipo: hazard de control
10 linea: 7
11 tipo: hazard de datos
12 linea: 9
13 tipo: hazard de control
14 linea: 10
15 tipo: hazard de control
16 linea: 7
17 tipo: hazard de datos
18 linea: 9
19 tipo: hazard de control
20 linea: 10
21 tipo: hazard de control
22 linea: 7
23 tipo: hazard de datos
24 linea: 9
25 tipo: hazard de control
26 linea: 10
27 tipo: hazard de control
28 linea: 7
29
```

Figura 3-4: Hazards encontrados

CAPÍTULO 4. ANÁLISIS

4.1 DETECTOR DE HAZARD

si comprobamos los resultados del detector de hazards para el programa anterior factorial.asm con el recorrido del pipeline se podrá notar que no toma en cuenta un hazard de datos de la linea t2.

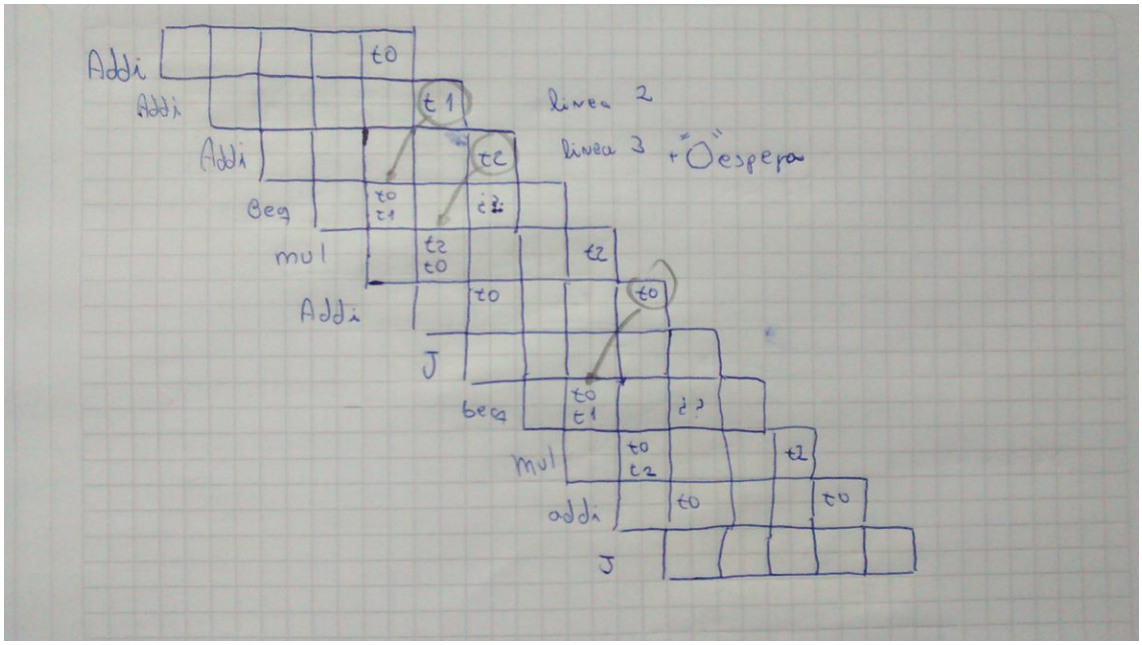

Figura 4-1: Pipeline archivo facorial.asm

Esto se debe a que el detector de hazard funciona en tiempo real, por lo que va variando a medida que crea esperas para reparar los hazards anteriores que son encontrados. En el caso del programa factorial.asm, se detecto un hazard de datos en la linea 2, ya que eran datos que necesitaba la instrucción de entrada: beq t0,t1, fin. para solucionar este hazard se agrego una espera antes de beq y después de la instrucción addi t2, zero, 1 por lo que al correr el pipeline a un lado desapareció el hazard de datos de la linea 3.

4.2 MEJORAS ARCHIVOS

A continuación se presenta la mejora del del programa exp-aprox.asm, la mejora consistió en reordenar el programa para que las instrucciones que modificaban el registro t6 se realizaran antes y así disminuir los hazards de datos y los ciclos que necesitaría el programa para correr.

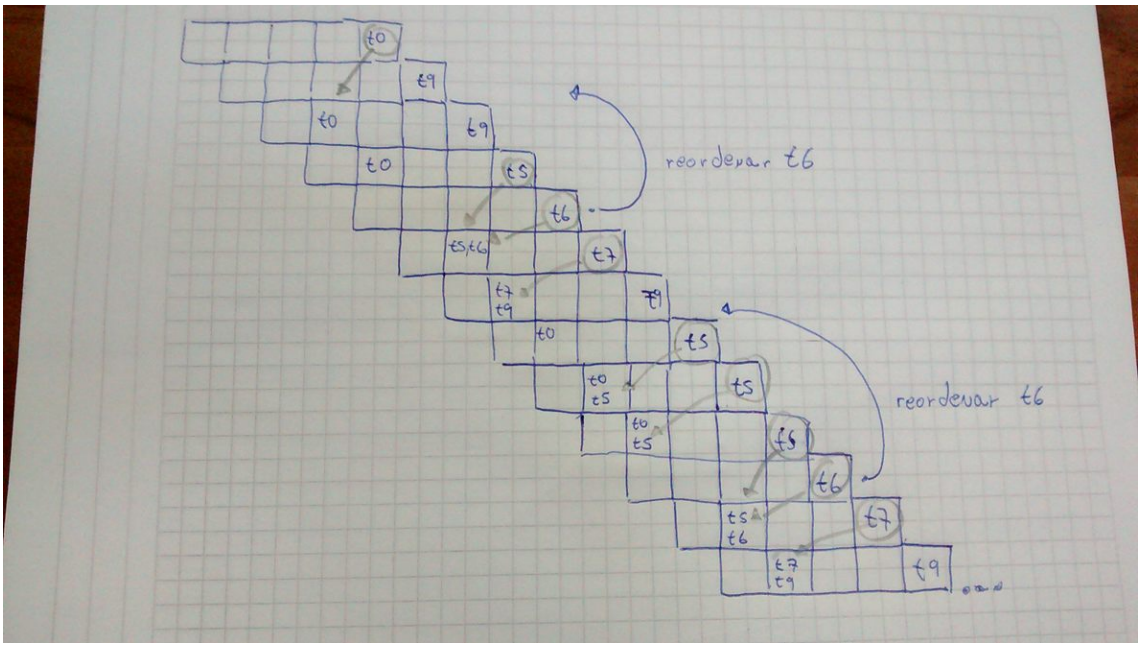

Figura 4-2: Pipeline archivo exp-aprox.asm

CAPÍTULO 5. CONCLUSIÓN

A partir de este laboratorio se logro aplicar los conocimientos aprendidos durante el semestre, sobre el funcionamiento del procesador y como este va realizando distintos procesos para llevar a cabo una instrucción, llegando a implementar un pipeline virtual que pudiera ejecutar distintas instrucciones en lenguaje assambler. Se logro separa correctamente las etapas del pipeline en funciones que realizaran las operaciones correspondientes, además de implementar una herramienta que pudiese detectar los hazards que tuviese un programa, para ello a medida que el pipeline iba ejecutando las instrucciones, el detector de hazards comparaba distintas instrucciones para comprobar que no usaran un registro que todavía no estaba guardado en la memoria de los registros.

Entre los métodos para lograr una mejora en los programas se puede señalar que el reordena-miento de instrucciones es un método practico y conveniente aunque no es fácil de utilizar cuando los programas son pequeños, logra obtener mejoras significativas en el numero de hazards que se presentan en el programa.

CAPÍTULO 6. BIBLIOGRAFÍA

Garay, F. (2016). Laboratorio 2 organización de computadores: pipeline.

Olivos, E. R. (2016). Organización de computadores: capitulo iii: hazard.

Wikipedia. (2016). Mips (procesador) — wikipedia, the free encyclopedia. [Online; accessed 5-noviembre-2016]. Recuperado desde https://es.wikipedia.org/wiki/MIPS_(procesador)