Instruction Scheduling Program

Braian Tenorio

Facultad de Ingeniería, Universidad Nacional de la Patagonia San Juan Bosco

Arquitectura de computadoras

Lic. Cristian Pacheco

25/04/2024

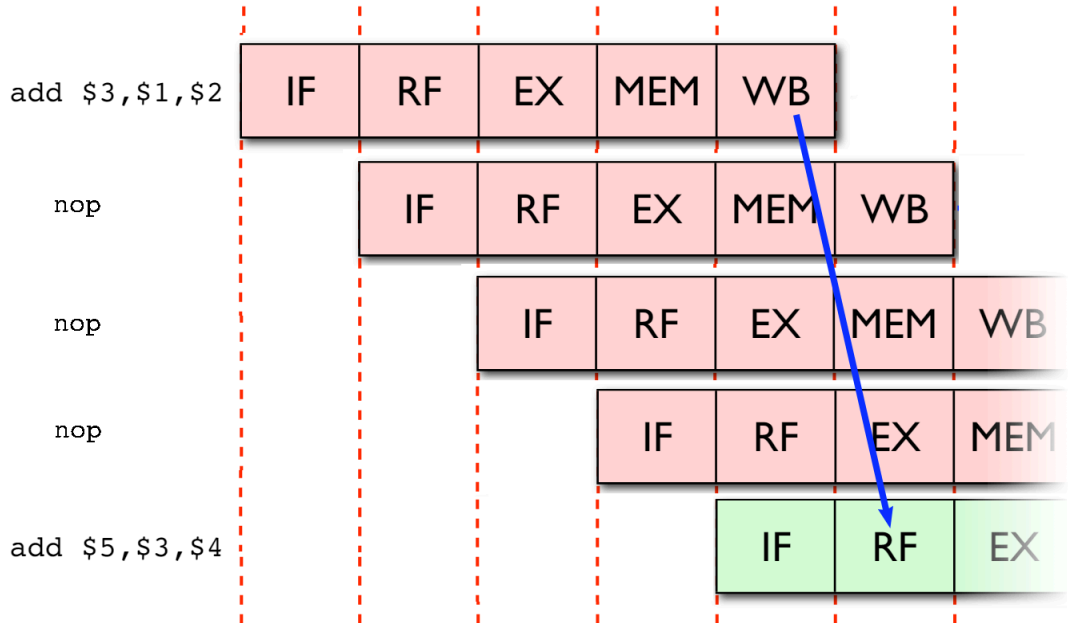
Indice

Introducción

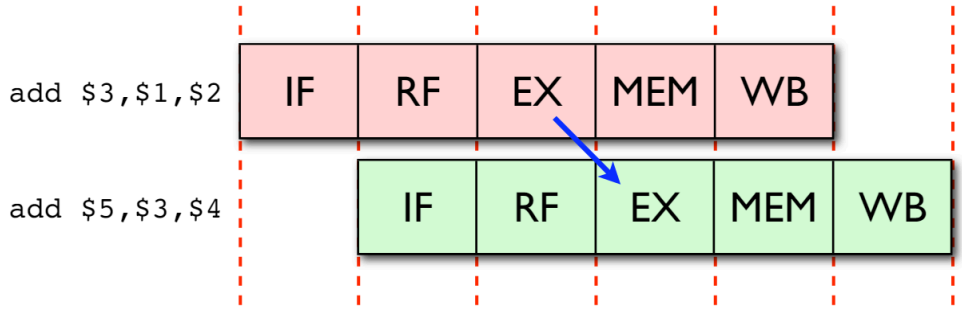
El diseño eficiente de programas en lenguaje ensamblador MIPS es esencial para optimizar el rendimiento de los sistemas informáticos. Aunque el diseño multiciclo permite, en caso ideal, ejecutar una instrucción por ciclo (una vez que se llena el pipeline), con ciclos mucho más cortos que en el diseño uniciclo (una unidad funcional = un ciclo), existen distintos tipos de problemas. En este caso abordaremos los *data hazards.* Estos ocurren cuando una instrucción necesita el resultado de la anterior y no está disponible cuando empieza su ejecución.

```
add $3,$1,$2
add $5,$3,$4
```

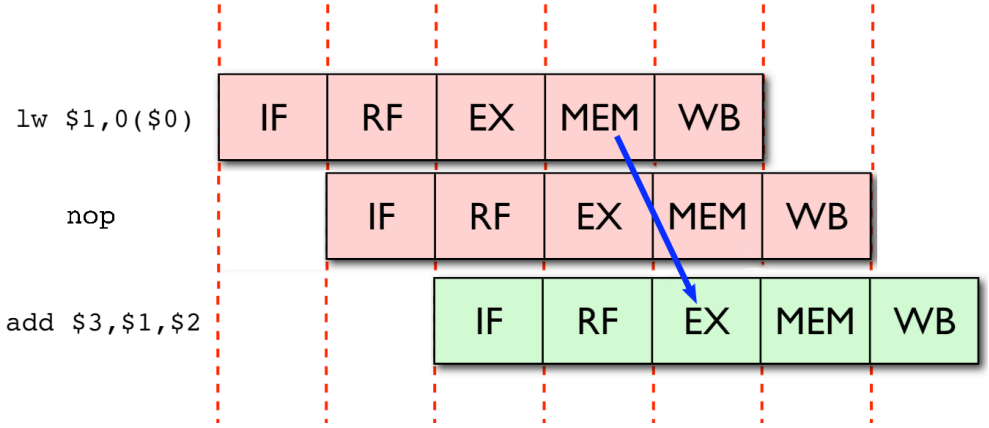
Cuando esto ocurre, debemos demorar el pipeline introduciendo nop's (instrucción que no hace nada para esperar a que escriba el resultado en los registros y la siguiente instrucción pueda ser ejecutada normalmente).



Este problema se reduce drásticamente cuando se usan cortocircuitos (o feed-forwarding). Esto es que se agregan circuitos entre las unidades funcionales para entregar el resultado lo antes posible a la siguiente instrucción y evitar conflictos.



Pero incluso con cortocircuitos hay algunas combinaciones de instrucciones que generan conflictos y se necesita demorar el pipeline usando nop's nuevamente. Como es el caso de una instrucción Load seguida de una instrucción que usa la alu (el dato está después de la etapa de acceso a memoria).



A través de este programa se intentará reducir los conflictos haciendo *planificación estática*, es decir reordenando el código sin afectar su funcionamiento.

Solución teorica

Lo primero que cabe aclarar es que este programa hace planificación *dentro de los bloques básicos* (un bloque básico es un conjunto de instrucciones donde las únicas instrucciones desde y hasta las cuales se pueden saltar son la última o la primera). El fin/comienzo de un bloque básico está marcado por instrucciones de salto, labels y SYSCALL.
Para que el ordenamiento no afecte el normal funcionamiento del programa tenemos que identificar dependencias entre instrucciones y respetar su orden. La dependencias entre instrucciones son:

- Read after write: Ocurre cuando una instrucción intenta leer un valor de un registro o ubicación de memoria antes de que otra instrucción haya terminado de escribir en ese mismo registro o ubicación de memoria.
- Write after read: Se da cuando una instrucción intenta escribir en un registro o ubicación de memoria antes de que otra instrucción haya terminado de leer ese mismo registro o ubicación de memoria.
- Write after write: Ocurre cuando dos instrucciones en paralelo intentan escribir en la misma ubicación de memoria sin que la primera haya completado su escritura.

Para lograr esto, se usará un DAG (directed acyclic graph) donde cada instrucción es un vértice y por cada dependencia entre 2 instrucciones se genera su correspondiente arco en el grafo. Cualquier ordenamiento topológico del grafo mantendrá las dependencias y por ende el funcionamiento del programa no cambiará.

Ahora se definirán algunas reglas para que el ordenamiento sea más efectivo.
Preferimos una instrucción si:

- No genera conflicto con la instrucción anterior.
- Tiene más sucesores en el grafo (por que nos dará más flexibilidad después).
- Está lo más alejado posible de una instrucción que puede ser válidamente planificada como última (esto evita tener que poner todas las instrucciones de un camino crítico juntas).

Se comprobarán las reglas en orden para elegir el mejor candidato.
Entonces el algoritmo seria algo asi:

```
Por cada bloque básico
    Armar DAG
    Mientras el DAG no este vacio
        Seleccionar candidatos (aquellos con inDegree==0)
        Seleccionar instrucción s (usando las reglas en orden)
        Quitar s del DAG
        Vaciar lista de candidatos
```

Herramientas usadas

Para llevar a cabo planificación estática usamos distintas herramientas que nos permitieron analizar el código en distintos niveles:

- **Java**: Se usó java para el programa main, el que reúne a las demás herramientas y dónde está la mayor parte de la lógica del programa. Además se definieron distintas clases y métodos.

- **Jflex**: Es un generador de analizadores léxicos para Java a través de reglas. Su función principal es leer el programa y reconocer secuencias que forman tokens. Este programa me ayudó a identificar registros, labels e instrucciones, que luego servirán para tareas como la formación de bloques básicos o el análisis de dependencia de datos.
- **Cup (Construction of Useful Parsers)**: Es un generador de analizadores sintácticos para java. Su función principal es tomar los tokens del analizador léxico y formar líneas (mi clase abstracta Line) para posteriormente formar bloques básicos.

Tanto el analizador léxico como el sintáctico se generan independientemente del programa main. También se usó las clases de JGraphT para el DAG y algún algoritmo de paths de grafos.

Implementación

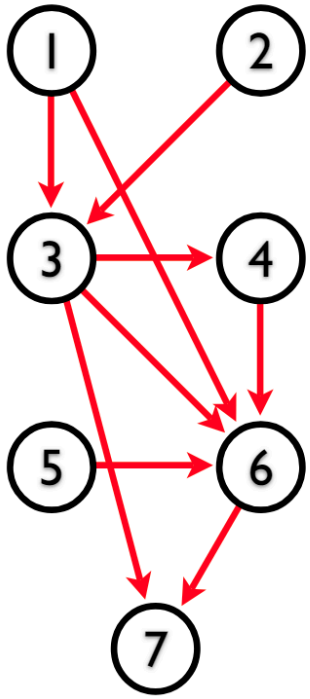
En mi programa, un bloque básico es un conjunto de líneas (clase Line, que a su vez tiene como clases hijas a Instruction, JumpInstruction y LabelDef). La particular construcción de estos objetos también se hace en el analizador sintáctico. Una parte importante de este proceso es guardar los registros que se leen y en los cuales se escribe en cada línea para luego identificar dependencias de datos.
No se le dio un trato particular a la sección de declaración de variables y los comentarios se ignoraron.
Una vez tenemos las líneas definidas, pasamos a formar bloques básicos en el programa main tomando como puntos de inicio y fin las Line de tipo JumpInstruction y LabelDef, es decir, los labels, instrucciones de salto y SYSCALL (porque no se analizará el contenido del registro $v0).
Luego para ordenar cada bloque básico, forme los DAGs de cada uno recorriendo todas las combinaciones de instrucciones creando arcos cuando encuentre alguna dependencia (RAW, WAR o WAW).

Para seleccionar candidatos, agregamos a una lista todos los vértices que tengan inDegree==0. Después aplicamos las 3 reglas en orden a la lista de candidatos.
Para la *primera regla* solo vemos la última instrucción planificada y vemos qué candidato no genera un conflicto si se planifica a continuación. Para la *segunda regla*, calculamos el outDegree de cada candidato y elegimos el mayor de todos. Y para la *tercera regla*, tenemos que ver todos los caminos simples entre cada vértice candidato y el/los posibles últimos, y elegir el que tenga el camino más largo. Si después de todas las reglas no hay una mejor elección se elegirá el primer candidato. Si hay un solo candidato, se elige sin pasar por las reglas. Por último, se escribe el nuevo programa usando las distintas implementaciones de toString de cada clase en un nuevo archivo.

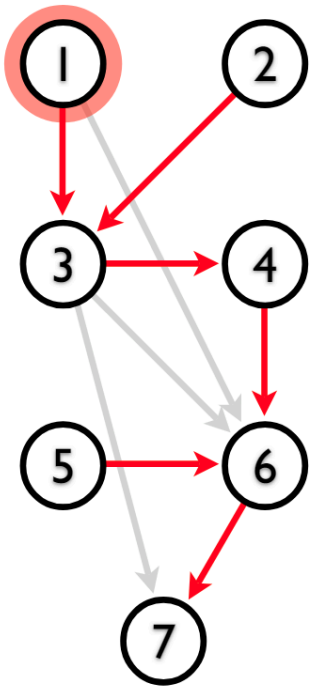
Ejemplo

La siguiente imagen muestra el código a planificar y su grafo de dependencias

```
1 lw $1,0($8)
2 lw $2,4($8)
3 add $3,$1,$2
4 sw $3,12($8)
5 lw $4,8($8)
6 add $3,$1,$4
7 sw $3,16($8)
```



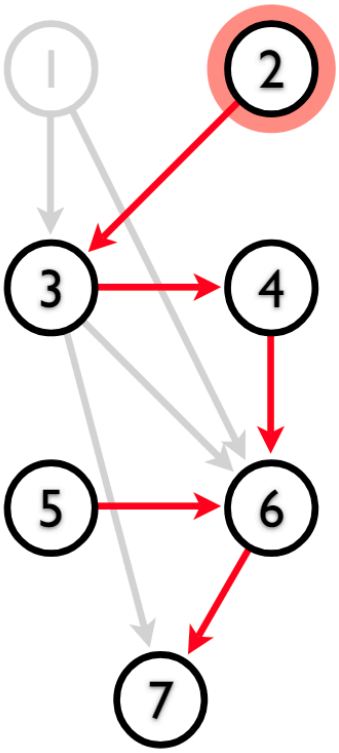
Como las primeras 2 instrucciones candidatas cumplen con las 3 reglas, elegimos la primera de ellas.



Candidatos:
{ 1, 2, 5 }

```
1 lw $1,0($8)
```

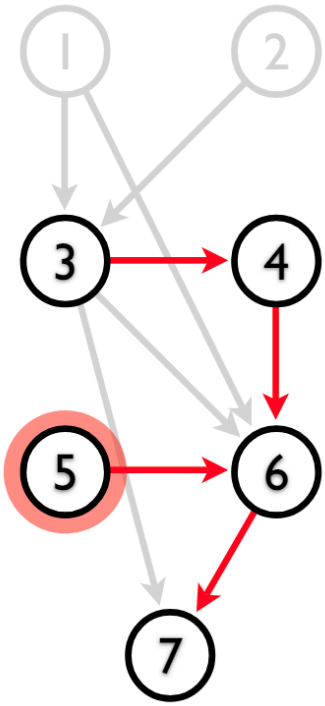
Elegimos la instrucción 2 por la regla 3.



Candidatos:
{ 2, 5 }

```
1 lw $1,0($8)
2 lw $2,4($8)
```

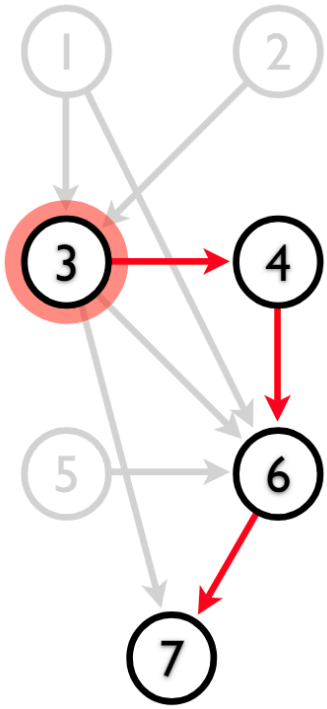
Se suma la instrucción 3 a la lista de candidatos y elegimos la instrucción 5 por la regla 1 (no entra en conflicto con la anterior)



Candidatos:
{ 3, 5 }

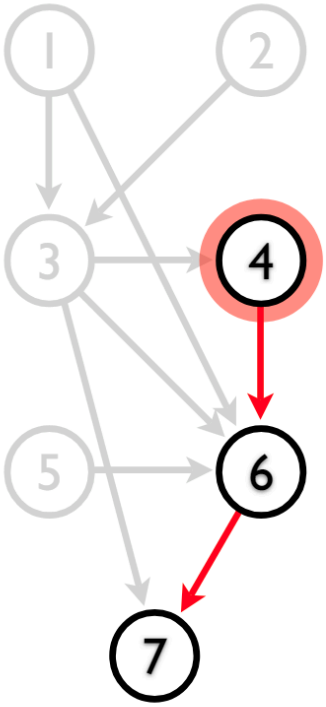
```
1 lw $1,0($8)
2 lw $2,4($8)
5 lw $4,8($8)
```

De aquí en más la lista de candidatos solo va a tener una instrucción por lo que se elige esa única.



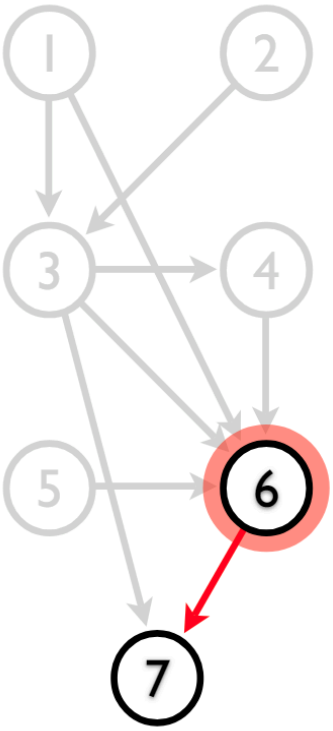
Candidatos:
{ 3 }

```
1 lw $1,0($8)
2 lw $2,4($8)
5 lw $4,8($8)
3 add $3,$1,$2
```



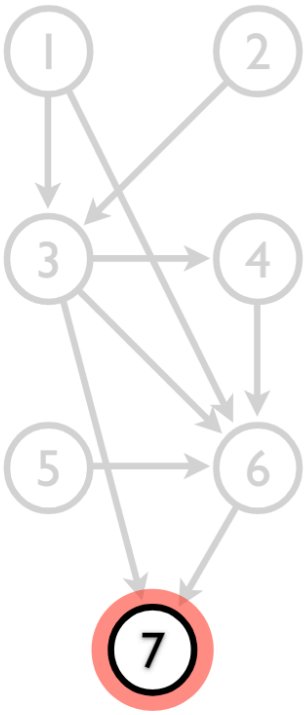
Candidatos:
{ 4 }

```
1 lw $1,0($8)
2 lw $2,4($8)
5 lw $4,8($8)
3 add $3,$1,$2
4 sw $3,12($8)
```



Candidatos:
{ 6 }

```
1 lw $1,0($8)
2 lw $2,4($8)
5 lw $4,8($8)
3 add $3,$1,$2
4 sw $3,12($8)
6 add $3,$1,$4
```



Candidatos:
{ 7 }

```
1 lw $1,0($8)
2 lw $2,4($8)
5 lw $4,8($8)
3 add $3,$1,$2
4 sw $3,12($8)
6 add $3,$1,$4
7 sw $3,16($8)
```

Comparación de resultado final

Codigo original:

```
1 lw $1,0($8)
2 lw $2,4($8)
3 add $3,$1,$2
4 sw $3,12($8)
5 lw $4,8($8)
6 add $3,$1,$4
7 sw $3,16($8)
```

2 conflictos

13 ciclos

Codigo nuevo:

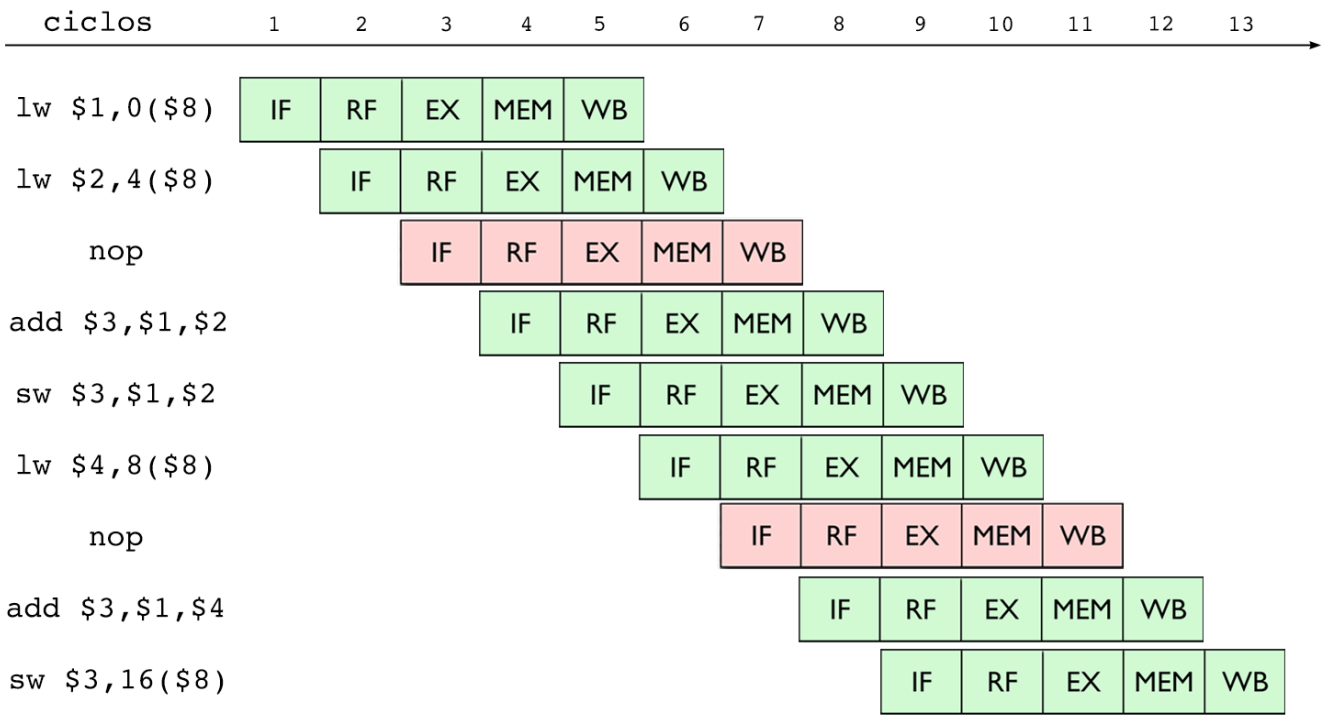
```
1 lw $1,0($8)
2 lw $2,4($8)
5 lw $4,8($8)
3 add $3,$1,$2
4 sw $3,12($8)
6 add $3,$1,$4
7 sw $3,16($8)
```

ningun conflicto

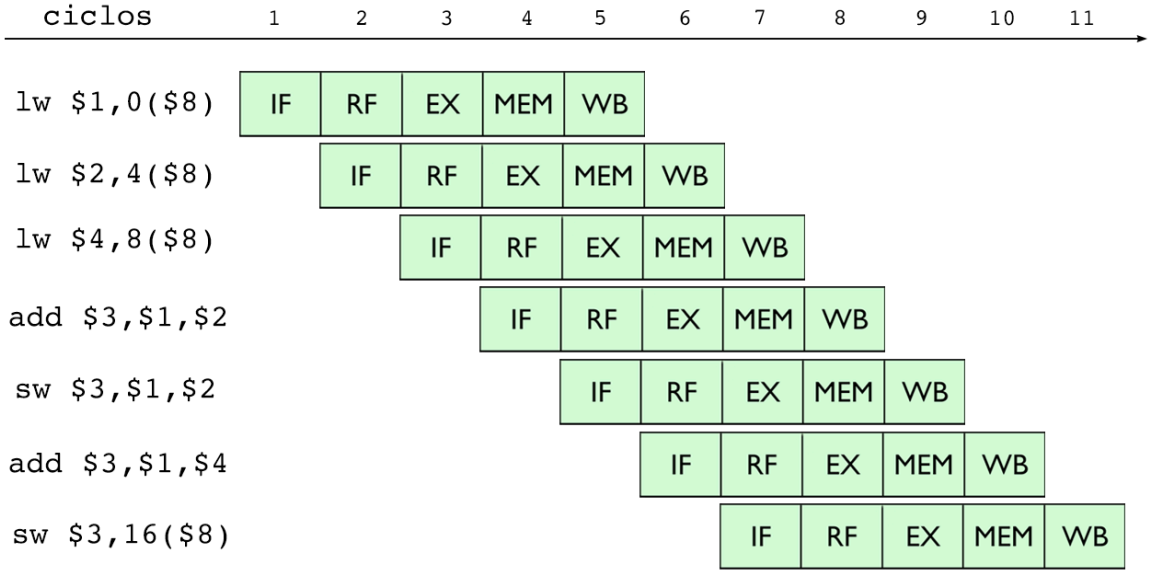
11 ciclos

Diagrama de etapas

Programa original



Programa final



Repositorio de GitHub

Repositorio del proyecto con el código e instrucciones para la instalación.
https://github.com/braiantenorio/Instruction-Scheduling-MIPS

Bibliografía

Diseño del procesador multiciclo - Jorge Dignani
Dependencias - Jorge Dignani
Pipelining – MIPS Implementation – Computer Architecture
Pipeline Hazards – Computer Architecture
Handling Data Hazards – Computer Architecture
Instruction Scheduling
Efficient DAG construction and heuristic calculation for instruction scheduling
An Experimental Evaluation of List Scheduling. Keith D. Cooper, Philip J. Schielke, and Devika Subramanian