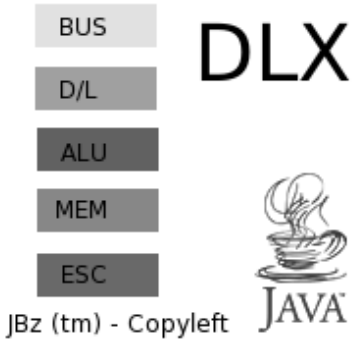
# UNIVERSIDAD NACIONAL DE INGENIERIA
# FACULTAD DE ELECTROTECNIA Y COMPUTACION
# INGENIERIA EN COMPUTACION
# ARQUITECTURA DE MAQUINAS III

## SIMULADOR DLX (jbz)




Integrantes:

Denis José Torres Guadamuz 2001-10500
Marconi Alexander Poveda Chacón 2001-10932
Hamerlin Ramón Silva Ruiz 2001-10552

Sección: 5T3-CO

Prof. Guillermo Loaisiga

19/07/05

## *Introducción*

DLX (jbz) es un simulador del procesador DLX segmentado. Es un programa hecho en JAVA lo cual permite su portabilidad a múltiples arquitecturas de sistemas de cómputo (ver Requerimientos del Sistema).

El simulador DLX (jbz) después de leer un archivo conteniendo código DLX, muestra información importante sobre el CPU (pipeline, registros, memoria, ...) mientras se ejecuta el código paso a paso o continuamente. DLX (jbz) ofrece estadísticas sobre el comportamiento del cause a lo largo del tiempo.

La segmentación encausada es una técnica de implementación que consiste en solapar la ejecución de múltiples instrucciones en el tiempo, en los procesadores que implementan esta técnica la ejecución de una instrucción se divide en diferentes fases y el procesador se organiza en unidades o secciones de ejecución relativamente independientes llamadas segmentos o etapas. El objetivo de la segmentación es conseguir una instrucción por ciclo, aunque cada instrucción por separado ocupe varios ciclos (CPI>1).

# *Diseño del Simulador*

## *Estructura Lógica del Simulador DLX (jbz):*

El simulador DLX (jbz) cuenta con 32 registros enteros de propósito general, nombrados de R0 a R31. Está segmentado en 5 etapas, las cuales son: BUS, D/L, ALU, MEM, y ESC.

| *Etapa* | *Descripción* |
|---|---|
| BUS | Etapa de búsqueda o fetch de la instrucción e incremento del PC |
| D/L | Decodificación/Carga de operandos (registros) en ALU |
| ALU | Ejecución de operaciones y cálculo de direcciones efectivas de datos |
| MEM | Acceso a memoria para lectura (M->R) o para escritura (R->M) |
| ESC | Escritura de resultados en los registros |

Cuenta con una memoria de 100 casillas, numeradas de 000 a 099. En la memoria se almacena el código fuente que se ejecuta en el simulador, así como valores enteros resultado de operaciones de escritura en memoria.

## *Formato de instrucciones del DLX (jbz):*

El formato de instrucción del DLX (jbz) es el siguiente:

| | | |
|---|---|---|
| Cálculo | OP RD,RF1,RF2 | RD = RF1 OP RF2 |
| Acceso a Memoria | LW RD,DB(RF) | RD = M[RF + DB] |
| Escritura en Memoria | SW DB(RF1),RF2 | M[RF1 + DB] = RF2 |

Significado de simbología:

| **Símbolo** | **Descripción** |
|---|---|
| OP | Indica un tipo de operación |
| RD | Registro destino |
| RF1 | Registro fuente primero |
| RF2 | Registro fuente segundo |
| DB | Dirección de memoria base |
| RF | Registro Fuente |

### *Juego de instrucciones del DLX (jbz):*

Cada instrucción cargada en el simulador DLX (jbz) ocupa una casilla de memoria. En el código fuente del programa las MAYUSCULAS y minúsculas se tratan por igual.

| *Instrucción* | *Ejemplo* | *Descripción* |
|---|---|---|
| ADDI | `ADDI R1,R0,7` | Suma el Registro Fuente con una Constante y el resultado se almacena en el Registro Destino. |
| SUBI | `SUBI R1,R0,7` | Resta el Registro Fuente de una Constante y el resultado se almacena en el Registro Destino. |
| ORI | `ORI R1,R0,7` | Aplica la operación OR a nivel de bits, entre el Registro Fuente con una Constante y el resultado se almacena en el Registro Destino. |
| ANDI | `ANDI R1,R0,7` | Aplica la operación AND a nivel de bits, entre el Registro Fuente con una Constante y el resultado se almacena en el Registro Destino. |
| XORI | `XORI R1,R0,7` | Aplica la operación XOR a nivel de bits, entre el Registro Fuente con una Constante y el resultado se almacena en el Registro Destino. |
| ADD | `ADD R3,R1,R2` | Suma el Registro Fuente 1 con el Registro Fuente 2 y el resultado se almacena en el Registro Destino. |
| SUB | `SUB R3,R1,R2` | Resta el Registro Fuente 1 del Registro Fuente 2 y el resultado se almacena en el Registro Destino. |
| MULT | `MULT R3,R1,R2` | Multiplica el Registro Fuente 1 con el Registro Fuente 2 y el resultado se almacena en el Registro Destino. |
| DIV | `DIV R3,R1,R2` | Divide el Registro Fuente 1 entre el Registro Fuente 2 y el resultado se almacena en el Registro Destino. |
| OR | `OR R3,R1,R2` | Aplica la operación OR a nivel de bits, entre el Registro Fuente 1 con el Registro Fuente 2 y el resultado se almacena en el Registro Destino. |
| AND | `AND R3,R1,R2` | Aplica la operación AND a nivel de bits, entre el Registro Fuente 1 con el Registro Fuente 2 y el resultado se almacena en el Registro Destino. |
| XOR | `XOR R3,R1,R2` | Aplica la operación XOR a nivel de bits, entre el Registro Fuente 1 con el Registro Fuente 2 y el resultado se almacena en el Registro Destino. |
| LW | `LW R3,0(R1)` | Carga en el Registro Destino el valor ocupado en la celda de memoria cuya dirección es: la suma de la Dirección Base más el valor del Registro Fuente. |

| *Instrucción* | *Ejemplo* | *Descripción* |
|---|---|---|
| SW | `SW 0(R1),R2` | Guarda el valor del Registro Fuente 2 en la celda de memoria cuya dirección es: la suma de la Dirección Base más el valor del Registro Fuente 1. |
| NOP | `NOP` | Operación de relleno. No realiza ninguna operación. |
| TRAP | `TRAP` | Indica al simulador que el programa ha terminado |
| ; | `;comentario` | Todo texto a la derecha del símbolo punto y coma será omitido por el simulador |

### *Estructura del Código del Simulador DLX (jbz):*

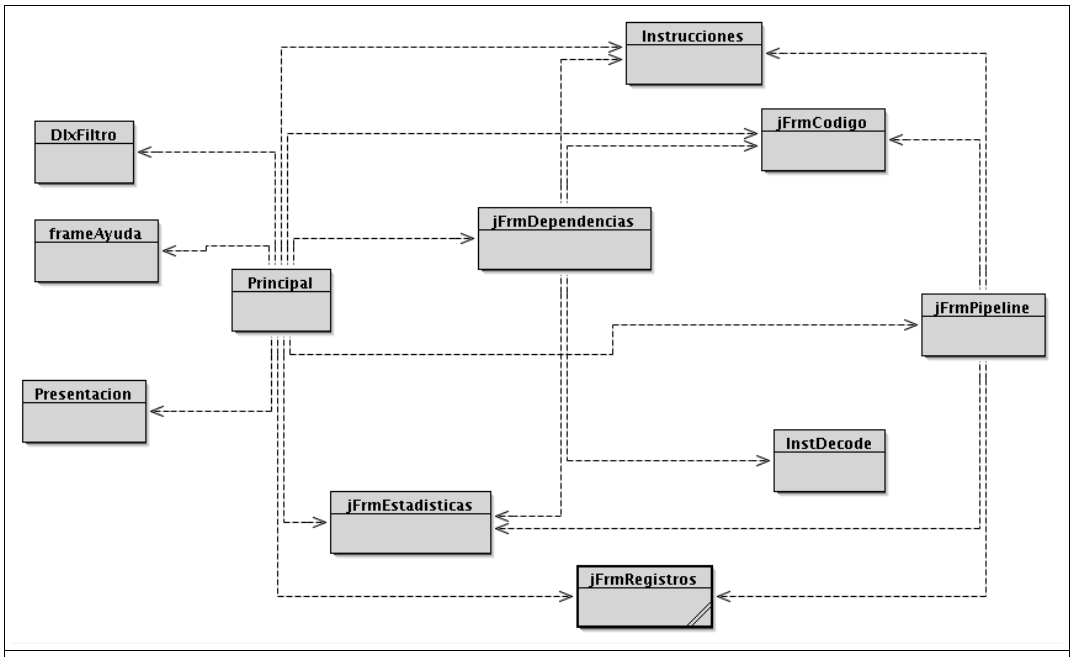



Diagrama de clases del simulador DLX (jbz)

| ***Clase*** | ***Descripción*** |
|---|---|
| Principal.java | Contiene el método Main y es el dlx en si. |
| Presentacion.java | Clase JFrame usada al inicio para mostrar pantalla de bienvenida. |
| Instrucciones.java | Contiene métodos y constantes para manipular, leer y procesar las instrucciones válidas para el simulador. |
| InstDecode.java | Clase para contener la estructura decodificada de cada instrucción, usada para buscar dependencias. |
| jFrmDependencias.java | Clase tipo Ventana encargada de encontrar y mostrar mostrar las dependencias en un código. |
| DlxFiltro.java | Clase usada para filtrar los archivos con extensión .dlx cuando se muestra la ventana Abrir. |
| frameAyuda.java | Clase tipo Ventana para mostrar la ayuda. |
| jFrmCodigo.java | Clase tipo Ventana para mostrar el código cargado en memoria. |
| jFrmEstadisticas.java | Clase tipo Ventana para mostrar las estadísticas |
| iFrmPipeline.java | Clase tipo Ventana para mostrar el pipeline |
| jFrmRegistros.java | Clase tipo Ventana para mostrar los registros |

# *Uso del simulador – Como Correrlo*

## *Requerimientos del simulador DLX (jbz):*

Software:

Es necesario para la ejecución del simulador DLX (jbz) la previa instalación de la plataforma Java 2 Standard Edition.

Sistema operativo: El sistema operativo puede ser cualquiera de los mencionados en la siguiente lista,

1. Plataformas Win32 (Windows 95, Windows 98, Windows NT, o superiores).
2. Plataformas basadas en UNIX (incluye Linux y Solaris).
3. Plataforma MAC OS (Macintosh).

Espacio de disco duro: Menos de 1 MB.

Memoria RAM: Mínimo de 32 MB.

Procesador: Mínimo de 133 MHZ.

## *Instalación:*

Copie la carpeta “dlxjbz” a su disco duro.

## *Ejecución del simulador:*

En una consola de comandos (la ventana de MS-DOS en Windows) ubicarse en el directorio “dlxjbz/bin” y escribir:

java -jar dlxjbz.jar

Nota para usuarios de Windows: en el directorio “dlxjbz/bin” se incluye un archivo por lotes con el nombre de: “run.bat” el cual puede ejecutar la aplicación con sólo ser picado dos veces.

### *Referencia rápida a los menús:*

| Menú | Submenú | Acceso Rápido | Descripción |
|---|---|---|---|
| **Archivo** | | | |
| | Reset | Ctrl + R | Reinicia el Simulador |
| | Abrir | Ctrl + A | Abrir un archivo dlx para ejecutar |
| | Salir | Ctrl + S | Salir del simulador |
| **Ejecutar** | | | |
| | Un Ciclo | F5 | Ejecuta un ciclo (un paso) |
| | Múltiples Ciclos | F6 | Ejecuta una cantidad de ciclos según se especifique |
| | Correr | F7 | Corre por completo todo el programa |
| **Configurar** | | | |
| | Instrucciones en el Pipeline | Ctrl + I | Establece la cantidad de instrucciones que pueden estar en el pipeline |
| **Ventana** | | | |
| | Registros | Ctrl + 1 | Muestra la ventana Registros |
| | Código | Ctrl + 2 | Muestra la ventana Código |
| | Pipeline | Ctrl + 3 | Muestra la ventana Pipeline |
| | Estadísticas | Ctrl + 4 | Muestra la ventana Estadísticas |
| | Dependencias | Ctrl + 5 | Muestra la ventana Dependencias |
| **Ayuda** | | | |
| | Contenido | F1 | Muestra la ayuda |
| | Acerca de | Ctrl + F1 | Muestra información sobre los autores del simulador |

Sientase libre de crear su propios códigos DLX y experimentar con diferentes configuraciones.

## *Plan de Pruebas y Resultados*

### *Prueba 1:*

```
;Ejemplo de código para DLX (jbz)
;Programa que suma 8+9 y guarda el resultado en R3

;Asegurar que el valor de R0 sea 0
XOR R0,R0,R0

;Cargar 8 en R1
ADDI R1,R0,8

;Cargar 9 en R2
ADDI R2,R0,9

;Sumar R1+R2 y guardar en R3
ADD R3,R1,R2

;Final del programa
TRAP
```

El objetivo de la prueba es comprobar el correcto funcionamiento del simulador DLX (jbz) con un código sencillo.

Resultados obtenidos mediante el simulador:

Valores finales de los Registros
R0 = 0
R1 = 8
R2 = 9
R3 = 17

Dependencias
(0) XOR R0,R0,R0
(1) ADDI R1,R0,8 RAW0
(2) ADDI R2,R0,9 RAW0
(3) ADD R3,R1,R2 RAW1 RAW2

Estadísticas

| ***Variable*** | ***Valor*** |
|---|---|
| Numero de Instrucciones Total | 5 |
| Ciclos Ejecutados | 9 |
| Instrucciones Ejecutadas | 4 |
| CPI | 2.25 |
| ILP | 0.44 |
| Instrucciones en el Pipeline | 1 |
| Dependencias RAW | 4 |
| Dependencias WAR | 0 |
| Dependencias WAW | 0 |
| Instrucciones Simultaneas en el Pipe Line | 5 |

Resultado de la prueba: OK

***Prueba 2:***

```
; Ejemplo DLX (jbz)
; Carga y lectura de valores en memoria

XOR R0,R0,R0          ; Asegurar 0 en R0
ADDI R1,R0,15         ; Cargar 15 en R1
ADDI R2,R0,4          ; Cargar 4 en R2

; Escritura en memoria
SW 0(R1),R2           ; M[15]=4    en la casilla 15 escribir 4
SW 1(R1),R2           ; M[15+1]=4  en la casilla 15+1 escribir 4

; Lectura de memoria
LW R3,0(R1)           ; r3 = M[15]
LW R4,1(R1)           ; r4 = M[15+1]

ADD R4,R2,R3          ; r4 = r4+r3

;FIN
TRAP
```

El objetivo de la prueba es comprobar el correcto funcionamiento del simulador DLX (jbz) con instrucciones de acceso a memoria.

Resultados obtenidos mediante el simulador:

Valores finales de los Registros
R0 = 0
R1 = 15
R2 = 4
R3 = 4
R4 = 8

Valores finales de la Memoria
M[015] = 4
M[016] = 4

Dependencias
(0) XOR R0,R0,R0
(1) ADDI R1,R0,15  RAW0
(2) ADDI R2,R0,4  RAW0
(3) SW 0(R1),R2  RAW1 RAW2
(4) SW 1(R1),R2  RAW1 RAW2
(5) LW R3,0(R1)  RAW1
(6) LW R4,1(R1)  RAW1
(7) ADD R4,R2,R3  RAW2 RAW5 WAW6

Estadísticas

| ***Variable*** | ***Valor*** |
|---|---|
| Numero de Instrucciones Total | 9 |
| Ciclos Ejecutados | 13 |
| Instrucciones Ejecutadas | 8 |
| CPI | 1.62 |
| ILP | 0.62 |
| Instrucciones en el Pipeline | 1 |
| Dependencias RAW | 10 |
| Dependencias WAR | 0 |
| Dependencias WAW | 1 |
| Instrucciones Simultaneas en el Pipe Line | 5 |

Resultado de la prueba: OK

***Prueba 3:***

```
; Ejemplo para mostrar que el simulador DLX (jbz)
; reconoce la división entre cero como un error

;I0
XOR R0,R0,R0    ; poner cero en r0

;I1
ADDI R1,R0,8    ; cargar 8 en R1

;I2
DIV R1,R1,R0    ; dividir 8/0 ERROR división por cero

;I3
TRAP  ; Finaliza el programa
```

El objetivo de la prueba es comprobar el correcto funcionamiento del simulador DLX (jbz) cuando un programa intenta dividir ente cero.

Resultado de la prueba: OK, el simulador avisó sobre el intento de división entre cero.

***Prueba 4:***

```
; Ejemplo DLX (jbz)
; Demostración de manejo de acceso no valido a memoria

XOR R0,R0,R0    ; poner cero en r0
ADDI R1,R0,8    ; cargar 8 en r1
ADDI R2,R0,9    ; cargar 9 en r2
ADD R3,R1,R2    ; r3 = r1 + r2

; Intenta acceder a la dirección 120+9
; la cual no existe y provoca un acceso no valido a memoria
SW 120(R2),R3   ; M[120+9] = r3

; Finaliza el programa
TRAP
```

El objetivo de la prueba es comprobar el correcto funcionamiento del simulador DLX (jbz) cuando un programa intenta acceder a una dirección de memoria no existente.

Resultado de la prueba: OK, el simulador avisó sobre el intento de acceso a una dirección de memoria no existente.

Estas pruebas y otras se incluyen en el directorio “ejemplos_dlx” dentro del directorio del simulador.

## *Estudio del Simulador – Variar Parámetros*

Repitiendo la prueba 1, pero variando la cantidad de instrucciones que pueden estar en el Pipeline. Antes de ejecutar el mismo código que el ejemplo de la prueba 1 se ha configurado al simulador DLX (jbz) para que solo permita 3 instrucciones en el pipeline.

Resultados:

Todos los resultados son idénticos a los mostrados en la Prueba 1, no así las estadísticas, las cuales se muestran a continuación.

Estadísticas

| ***Variable*** | ***Valor*** |
|---|---|
| Numero de Instrucciones Total | 5 |
| Ciclos Ejecutados | 11 |
| Instrucciones Ejecutadas | 4 |
| CPI | 2.75 |
| ILP | 0.36 |
| Instrucciones en el Pipeline | 1 |
| Dependencias RAW | 4 |
| Dependencias WAR | 0 |
| Dependencias WAW | 0 |
| Instrucciones Simultaneas en el Pipe Line | 5 |

Observación: Los resultados importantes que destacar son: el CPI resultó ser 2.75 contra 2.25 obtenido antes lo cual significa que al haber menos instrucciones en el pipeline la cantidad de ciclos promedios que hay que ejecutar por cada instrucción aumenta, en cambio el ILP disminuye. Además de que la cantidad total de ciclos necesarios para ejecutar el programa incrementó.

## *Conclusiones*

Desde un punto de vista general: el uso de un simulador del procesador DLX, con la capacidad de poder realizar distintas pruebas, con distintos códigos, y ejecuciones paso a paso, ayuda grandemente a comprender los conceptos de segmentación encausada o pipeline.

En nuestra experiencia: de los datos obtenidos en las distintas pruebas que realizamos con nuestro simulador, encontramos que: mientras más instrucciones a la vez hay en el pipeline la cantidad de ciclos necesarios para ejecutar todo el programa disminuye, lo cual también significa que el CPI disminuye y el ILP aumenta, lo cual se traduce en un aumento del rendimiento del procesador.

## *Bibliografía*


- Segmentación Encausada
  Prof. José L. Díaz Chow
  Depto. de Arquitectura y Sistemas

- WinDLX DLX-Pipeline Simulator
  TU Vienna, Inst. fr Technische Informatik