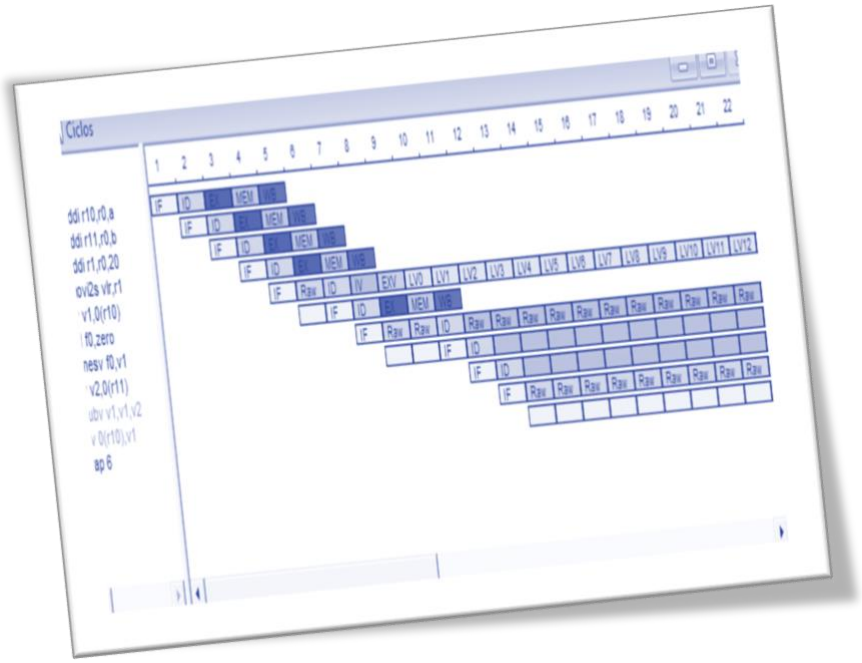
UNIVERSITAT POLITÈCNICA DE VALÈNCIA
CAMPUS D'ALCOI



UD4: EVALUACIÓN DE PRESTACIONES PROCESADOR VECTORIAL

ARQUITECTURA E INGENIERÍA DE COMPUTADORES:

Pau Micó

Marcos del Amo Fernández
Pablo Gimenez Villa
Grado en Ingeniería Informática
Curso 2023-2024

1 INTRODUCCIÓN

Una vez entendida la arquitectura básica y los principios de funcionamiento del simulador DLXV (ver documento de Introducción al DLXV) se te proponen a continuación una serie de ejercicios a ejecutar en este mismo simulador.

Para la SESIÓN 1 de trabajo deberás leer detalladamente el manual de usuario del DLXV. Además, también tendrás que realizar una comparativa de las prestaciones obtenidas del procesamiento de un conocido bucle matemático entre un procesador escalar segmentado (DLX) y un procesador vectorial (DLXV).

En la SESIÓN 2 de trabajo se plantea la resolución de una resta vectorial condicionada mediante la máscara VMR para, a posteriori, evaluar las prestaciones de la ejecución del código en el DLXV.

1.1 OBJETIVOS

- aprender a utilizar el simulador de un procesador vectorial llamado DLXV
- evaluar las prestaciones y mejoras obtenidas en la ejecución de cierto tipo de programas sobre un microprocesador vectorial (DLXV), respecto de las prestaciones ofrecidas por un microprocesador segmentado (DLX)
- aprender a aprovechar el paralelismo de datos con un microprocesador vectorial

1.2 METODOLOGÍA

La duración de la práctica es de 240 minutos, repartidos en dos sesiones de 120 minutos que se utilizarán para la resolución de las cuestiones planteadas en el boletín de la práctica. Al final de la misma (SESIÓN 2) el alumno deberá presentar (en formato `pdf`) una memoria justificativa del trabajo realizado estructurada en los siguientes apartados:

- portada identificativa (nombre alumno, curso, título práctica, bloque temático)
- detalle de los contenidos (índice)
- introducción y objetivos
- desarrollo
- conclusiones personales
- bibliografía

La nota obtenida en este acto de evaluación formará parte de la nota del bloque temático dedicado a los Procesadores Vectoriales.

2 SIMULADOR DLXV

DLXV es un simulador de microprocesadores vectoriales basado en un estándar de arquitectura similar al del del procesador segmentado DLX y que utiliza una adaptación del set de intrucciones del procesador segmentado MIPS, que ya deberías conocer. En esta práctica vas a utilizar ensamblador como lenguage de programación, por lo que sería muy interesante que repasaras lo aprendido en la asignatura de *Estructura de Computadores*. Antes de empezar con las simulaciones se te propone una lectura detallada del manual de usuario del DLXV. Para ello deberás instalar el código ejecutable para Windows del simulador, que se puede descargar desde aquí. El manual del usuario del DLXV (en formato pdf) se encuentra accesible en el mismo directorio de instalación del simulador.

3 BUCLE DAXPY

Un problema vectorial típico es el del cálculo de la operación $\boldsymbol{Z = a \cdot X + Y}$ (donde $\boldsymbol{X}$ e $\boldsymbol{Y}$ son dos vectores, que inicialmente residen en memoria y $\boldsymbol{a}$ es un escalar). Este problema se resuelve con la implementación del conocido bucle SAXPY o DAXPY (según si es simple o doble precisión). Precisamente este cálculo iterativo es el que constituye uno de los bucles internos del conocido benchmark Linpack (colección de rutinas de álgebra lineal) utilizado para la caracterización de los sistemas supercomputadores que aparecen en top500.

3.1 CODIFICACIÓN ESCALAR DEL BUCLE

El propio simulador facilita el código escalar del bucle en el programa EDAXPY.S. Selecciónalo y cárgalo en el simulador. Como comprobarás en este caso, al ejecutar el código en el procesador DLXV sólo se utilizarán las unidades escalares del procesador, desaprovechando tanto el paralelismo de datos que presentan los vectores como la unidad vectorial del procesador (que no se usa). Para una codificación en modo escalar el código se llama EDAXPY. El vector X es de 64 componentes tipo double (8 bytes) y aparece cargado en memoria de datos a partir de la dirección 1000h (***Rx***). El vector Y es del mismo tipo que X y aparece cargado en memoria de datos a partir de la dirección 1200h (***Ry***). El vector resultante Z se almacenará a partir de la dirección 1400h (***Rz***). El valor del escalar a aparece en la dirección de memoria 1600h (***a***):

```
; Código DAXPY para procesador ESCALAR


; En este punto se definirían los parámetros de entrada


      LD F0, a            ; carga escalar de a
      ADDI R4, Rx, #512   ; última dirección de memoria con el dato
                          ; a cargar (512 = 64 datos x 8 bytes)
LOOP:
      LD F2, 0(Rx)        ; Carga dato 'i' del vector X → X(i)
      MULTD F2, F0, F2    ; a * X(i)
      LD F4, 0(Ry)        ; Carga dato 'i' del vector Y → Y(i)
      ADDD F4, F2, F4     ; a * X(i) + Y(i)
      SD 0(Rz), F4        ; almacena en resultado en Y(i)
      ADDI Rx, Rx, #8     ; Incrementa el índice de memoria del vector X
      ADDI Ry, Ry, #8     ; Incrementa el índice de memoria del vector Y
      ADDI Rz, Rz, #8     ; Incrementa el índice de memoria del vector Z
      SUB R20, R4, Rx     ; Calcula el trozo de vector que nos queda por procesar
      BNZ R20, LOOP       ; Comprueba si se ha terminado de procesar el vector
```

```
; En este punto se seguiría con la implementación o se termina con un TRAP #0

; El TRAP #0 devuelve el control del micro al SO
```

3.2 CODIFICACIÓN VECTORIAL DEL BUCLE

Si queremos optimizar el tiempo de CPU aprovechando el paralelismo de los datos que aparecen organizados en vectores, podemos optar por codificar el bucle DAXPY para ejecutarlo en un procesador vectorial. Carga en el simulador el programa VDAXPY.S. Para la codificación de este ejemplo concreto del DAXPY vamos a suponer que el número de elementos o longitud de un registro vectorial (Maximum Vector Length Register, MVLR = 64) coincide con la longitud de la operación vectorial (es decir, ***X, Y*** y ***Z*** son vectores de 64 componentes) y que las direcciones de comienzo de ***X, Y, Z*** se identifican con las etiquetas ***Rx, Ry*** y ***Rz*** cargados en el segmento de datos de la memoria (ver punto anterior). Además, en este caso el programa para la versión vectorial del bucle se llama VDAXPY y resulta:

```
; Código DAXPY para procesador VECTORIAL


; En este punto se definirían los parámetros de entrada


      LD F0, a            ; Carga escalar de a

      LV V1, Rx           ; Carga vector X

      MULTSV V2, F0, V1   ; a * X (multiplicación escalar por vector)

      LV V3, Ry           ; Carga vector Y

      ADDV V4, V2, V3     ; a * X(i) + Y(i) (suma vectorial)

      SV Rz, V4           ; almacena el resultado


; En este punto se seguiría con la implementación o se termina con un TRAP #6
```

```
; El TRAP #6 permite devolver el control al SO pero terminando

; las operaciones vectoriales que todavía quedan pendientes
```

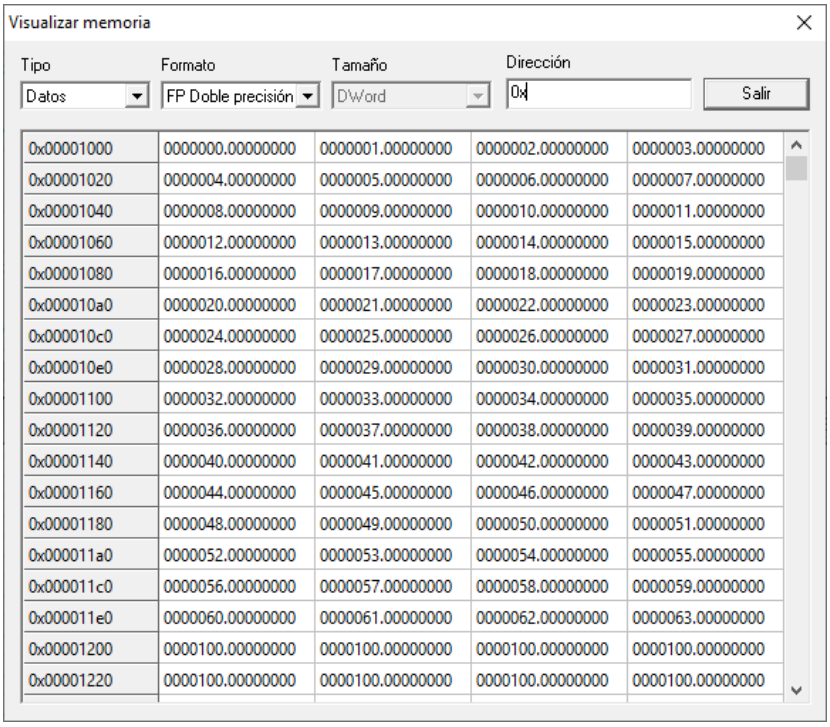
4 SESIÓN 1: CODIFICACIÓN ESCALAR VS. VECTORIAL

En esta SESIÓN 1 vas a comparar las prestaciones de las dos versiones del bucle DAXPY (la escalar y la vectorial) obtenidas al ejecutarlas sobre el simulador de un procesador vectorial DLXV.

4.1 EJECUCIÓN EN MODO ESCALAR

Para la ejecución de la versión escalar del DAXPY sobre el DLXV vamos a analizar primero la ruta de datos escalar del DLXV. Sobre la imagen presentada por el simulador, observa cómo en la etapa de ejecución del cauce escalar se incluyen varias unidades funcionales en paralelo (multiplicación FP, suma/resta FP, división FP y el resto de operaciones de enteros). Este hecho ya supone una mejora importante respecto del procesador segmentado sencillo DLX, que sólo presentaba una ruta de datos única para su etapa de ejecución.

Para la simulación del bucle DAXPY en modo escalar, carga el programa **`edaxpy.s`** del repositorio de programas de ejemplo que ya vienen instalados en el DLXV. Accede a la ventana de memoria de datos (mira en la configuración a partir de qué dirección de memoria se cargan por defecto los datos, etiquetas Rx y Ry) y comprueba el valor numérico de los componentes de los vectores. Se recomienda en este caso que visualices la memoria de datos con formato FP de doble precisión (tamaño DWord). Comprueba los valores del vector X (a partir de la dirección de memoria 1000h), Y (a partir de la 1200h), vector Z (a partir de la 1400h).



0x00001000	0000000.00000000	0000001.00000000	0000002.00000000	0000003.00000000
0x00001020	0000004.00000000	0000005.00000000	0000006.00000000	0000007.00000000
0x00001040	0000008.00000000	0000009.00000000	0000010.00000000	0000011.00000000
0x00001060	0000012.00000000	0000013.00000000	0000014.00000000	0000015.00000000
0x00001080	0000016.00000000	0000017.00000000	0000018.00000000	0000019.00000000
0x000010a0	0000020.00000000	0000021.00000000	0000022.00000000	0000023.00000000
0x000010c0	0000024.00000000	0000025.00000000	0000026.00000000	0000027.00000000
0x000010e0	0000028.00000000	0000029.00000000	0000030.00000000	0000031.00000000
0x00001100	0000032.00000000	0000033.00000000	0000034.00000000	0000035.00000000
0x00001120	0000036.00000000	0000037.00000000	0000038.00000000	0000039.00000000
0x00001140	0000040.00000000	0000041.00000000	0000042.00000000	0000043.00000000
0x00001160	0000044.00000000	0000045.00000000	0000046.00000000	0000047.00000000
0x00001180	0000048.00000000	0000049.00000000	0000050.00000000	0000051.00000000
0x000011a0	0000052.00000000	0000053.00000000	0000054.00000000	0000055.00000000
0x000011c0	0000056.00000000	0000057.00000000	0000058.00000000	0000059.00000000
0x000011e0	0000060.00000000	0000061.00000000	0000062.00000000	0000063.00000000
0x00001200	0000100.00000000	0000100.00000000	0000100.00000000	0000100.00000000
0x00001220	0000100.00000000	0000100.00000000	0000100.00000000	0000100.00000000

Vector X de 64 componentes con valores que van de 0 a 63

Para la configuración por defecto del simulador, ejecuta el programa paso a paso y contesta a las siguientes cuestiones:

- Configuración del sistema:
    - ¿cómo resuelve el simulador los riesgos por dependencia de datos?

      En la ejecución escalar, el procesador desaprovecha tanto el paralelismo de datos que ofrecen los vectores como las unidades vectoriales del procesador. Por lo que, en este caso, los riesgos por dependencia de datos se resuelven mediante la inserción de ciclos de parada. Dicha configuración tiene "adelantamiento de resultados" con lo que las instrucciones con dependencia de datos pueden realizar la etapa EX en el mismo ciclo en el que la instrucción con la que tienen dependencia se encuentra realizando la etapa MEM (sin tener que pasar por WB).

- ¿cómo resuelve el simulador los riesgos de control?

En el código tras la instrucción de salto hay una instrucción nop, y en esta configuración no hay salto retardado, sino que se predice no tomar. Por lo que el simulador resuelve los riesgos de control mediante la detención de la segmentación por un ciclo.

- ¿cómo resuelve el simulador los riesgos estructurales?

Mediante unidades funcionales en paralela.

- Riesgos:
  - Enumera los distintos tipos de riesgos que aparecen durante la ejecución del código. En este caso, ¿pueden aparecer riesgos por dependencia de datos de tipo RAW, WAR i WAW?

Durante la ejecución del código, aparecen los siguientes riesgos por dependencia de datos:

1. Tipo RAW:
   a. Aparece un riesgo entre la instrucción ld f2, 0(r1) y multd f2, f0, f2, con lo que se inserta 1 ciclo de parada.
   b. Aparece un riesgo entre la instrucción multd f2, f0, f2 y addd f4, f2, f4, con lo que se insertan 6 ciclos de parada.
   c. Aparece un riesgo entre la instrucción sd 0(r3), f4 y addd f4, f3, f4, con lo que se insertan 3 ciclos de parada.
   d. Aparece un riesgo entre la instrucción sub r5, r4, r1 y bnez r5, loop, con lo que se inserta 1 ciclo de parada.
2. Tipo WAR, no aparece ningún riesgo de este tipo.
3. Tipo WAW, no aparece ningún riesgo de este tipo.

Esas dependencias se repiten en cada iteración, suman un total de 11(1+6+3+1) ciclos insertados de parada por iteración. Por 64 iteraciones tenemos un total de 704 ciclos de parada debido a dependencias de datos RAW.

En la traza del simulador cuenta 6 parones por dependencia RAW en la instrucción sd 0(r3), f4. Por lo que cuenta 17 parones por iteración, un total de 1088 parones por RAW.

- Para cada tipo de riesgo, indica exactamente el número de veces que provocan la detención de la segmentación (cuántos ciclos de parada se insertan debido a cada riesgo en concreto)

Indicado en el apartado anterior.

- Número de instrucciones (todas escalares)
    - Número de instrucciones de carga ejecutadas (Nc) = 129
    - Número de instrucciones de almacenamiento ejecutadas (Nl) = 64
    - Número de instrucciones escalares en coma flotante ejecutadas (Nfp) = 64 multiplicaciones + 64 sumas/restas = 128
    - Número de instrucciones vectoriales ejecutadas (Nv) = 0
    - Número total de instrucciones ejecutadas NT =647
    - Comprueba si se cumple que `NT = Nc + Nl + Nfp + Nv` y justifica en cualquier caso tu respuesta

`NT = Nc + Nl + Nfp + Nv = 129 + 64 + 128 + 0 = 321`

`No se cumple, porque no estamos contando las instrucciones de suma/resta con enteros, las instrucciones de salto, ni la instrucción nop final que sí que tiene en cuenta el simulador.`

***Nº operaciones suma/resta enteros** = 5 (fuera del bucle) + 4 (dentro del bucle) x 64 (nº iteraciones) = 261*

***Nº nop** = 1 * 64 (una por cada iteración) = 64*

Trap = 1

Con lo que no hemos tenido en cuenta estas 3 anteriores, que juntas suman = 261 + 64 + 1 = 326

Ahora si se cumple => 321 + 326 = NT = 647

- Estimación de las prestaciones del sistema

- Valor del CPI al final de la ejecución

Ciclos = 1354 ciclos

N. º Instrucciones = 647 instrucciones

CPI = 1354 / 647 = 2,093 ciclos /instrucción

- Número de ciclos de espera totales 1151

Los ciclos de espera debido a las dependencias RAW (comentados anteriormente) = 704 + 63 = 767

O

Tal como aparece en el simulador = ciclos de espera RAW + ciclos espera control = 1088 + 63 = 1151

1. El número de ciclos de espera total en este caso viene dado por la expresión: $ciclos\ Espera\ RAW + ciclos\ Espera\ Control = 1088 + 63 = 1151$

- Número de veces que se ejecuta el bucle

El bucle consta de 64 iteraciones (tamaño del vector).

- Número de ciclos de espera introducidos por cada pasada del bucle

Comentado anteriormente

18, última 17

- ¿A qué se deben las detenciones que se producen en el anterior código?

Se deben a que la configuración actual del simulador predice que no se toma el salto.

- ¿Se puede configurar el simulador de forma que se minimicen el número de detenciones? ¿cómo?

Si, utilizando el salto retardado, de forma que ahora si se ejecute la instrucción siguiente al bucle, si reestructuráramos el código y pusiéramos una instrucción (en vez de usar la instrucción nop) ahorraríamos ciclos.

- Aceleración respecto de la ejecución del código en un procesador segmentado, sin múltiples unidades funcionales (tipo DLX) (el ejecutable del simulador DLX lo puedes descargar desde

aquí). En este caso, seguramente deberás adaptar el código en ensamblador **`edaxpy.s`** a la versión original del DLX

```
.data
x: .word 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
    20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39,
    40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59,
    60, 61, 62, 63
y: .word 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100,
    100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100,
    100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100,
    100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100
z: .space 256
a: .word 1

.text
main: addi r1, r0, x   ; r1 = dirección de x
    addi r2, r0, y   ; r2 = dirección de y
    addi r3, r0, z   ; r3 = dirección de z
    lw r4, 0(a)     ; r4 = escalar a
    addi r5, r1, 256 ; r5 = dirección final de x
loop: lw r6, 0(r1)     ; r6 = elemento de x
    mul r7, r4, r6   ; r7 = a * x[i]
    lw r8, 0(r2)     ; r8 = elemento de y
```

```
add r9, r7, r8   ; r9 = a * x[i] + y[i]
sw 0(r3), r9     ; z[i] = a * x[i] + y[i]
addi r1, r1, 4   ; Siguiente elemento de x
addi r2, r2, 4   ; Siguiente elemento de y
addi r3, r3, 4   ; Siguiente elemento de z
sub r10, r5, r1
bnez r10, loop
nop
trap 0
```

Se podría adaptarlo y la mejora vendría obtenida por la siguiente expresión:

Speed up = T(sin mejorar) / T(mejorado) = x/1354

- ¿Cuál sería la equivalencia entre el ciclo de reloj del procesador segmentado DLX con el del procesador vectorial DLXV? (fíjate en los ciclos de latencia para unidad funcional más lenta en el DLXV)

Sabiendo que el Segmentado DLX se ajusta el tiempo de las etapas al tiempo de etapa del peor caso (siendo en este caso el peor caso 24 ciclos (DIV FP)), calculamos que 1 ciclo del Segmentado DLX equivale a 24 ciclos del DLXV.

Ahora intenta optimizar la ejecución del código mediante la modificación de la configuración del simulador (SIN modificar las latencias de la unidades funcionales). Se te propone una configuración con salto retardado y reordenación del código. Una vez optimizado vuelve a calcular el valor del CPI.

- ¿Cuál es la aceleración que obtenemos en este caso?

Esta es la reordenación de código utilizada:

```
; Ejemplo escalar-vectorial
;
; Bucle DAXPY con instrucciones escalares:
```

```
; Realiza la operaciÛn Z = aX + Y donde X e Y son vectores
; que residen en memoria, y a es un escalar.
;
;
        .data
x:      .double 0, 1, 2, 3, 4, 5, 6, 7, 8, 9                    ;Vector X
        .double 10, 11, 12, 13, 14, 15, 16, 17, 18, 19
        .double 20, 21, 22, 23, 24, 25, 26, 27, 28, 29
        .double 30, 31, 32, 33, 34, 35, 36, 37, 38, 39
        .double 40, 41, 42, 43, 44, 45, 46, 47, 48, 49
        .double 50, 51, 52, 53, 54, 55, 56, 57, 58, 59
        .double 60, 61, 62, 63

y:      .double 100, 100, 100, 100, 100, 100, 100, 100, 100, 100 ;Vector Y
        .double 100, 100, 100, 100, 100, 100, 100, 100, 100, 100
        .double 100, 100, 100, 100, 100, 100, 100, 100, 100, 100
        .double 100, 100, 100, 100, 100, 100, 100, 100, 100, 100
        .double 100, 100, 100, 100, 100, 100, 100, 100, 100, 100
        .double 100, 100, 100, 100, 100, 100, 100, 100, 100, 100
        .double 100, 100, 100, 100

z:      .space 512   ;Vector Z
```

```
a:      .double 1    ;Escalar a


        .text
main:   addi r1,r0,x    ;r1 contiene la direcciÛn de x
        addi r2,r0,y    ;r2 contiene la direcciÛn
        addi r3,r0,z    ;r3 contiene la direcciÛn de z
        ld f0,a(r0)     ;f0 escalar
        addi r4,r1,512  ;64 elementos son 512 bytes
loop:   ld f2,0(r1)
        multd f2,f0,f2
        ld f4,0(r2)
        addd f4,f2,f4
        sd 0(r3),f4
        addi r1,r1,8
        addi r2,r2,8
        sub r5,r4,r1
        bnez r5,loop
        addi r3,r3,8; Hueco de retardo
        trap 6          ; Fin del programa
```

Obtenemos las siguientes estadísticas:

1290 ciclos (nos ahorramos 1 ciclo en cada iteración, ahorramos 64 ciclos)

646 instrucciones

1034 CICLOS

583 INSTRUCCIONES

CPI = 1,997 ciclos / instrucción

Vemos que conseguimos un mejor CPI con esta nueva configuración, por lo que mejoramos las prestaciones. Además, ahora tenemos menos ciclos, por lo que también hemos reducido el tiempo de ejecución. Ahora vemos cuánto hemos mejorado:

En este caso obtenemos la siguiente aceleración:

Speed up = T (sin mejorar, con predicción de no saltar y nop) / T(mejorado, reordenación y salto retardado) = 1354/1290 = 1,0496

4.2 EJECUCIÓN EN MODO VECTORIAL

En este caso vamos a aprovechar el paralelismo presentado por los datos del programa, lo que nos permite codificar el bucle DAXPY en una versión para ser ejecutada en el procesador vectorial DLXV. Para ello carga el programa **`vdaxpy.s`** del repositorio de programas de ejemplo ya instalados en el DLXV. Para la configuración por defecto del simulador, ejecuta el programa paso a paso y contesta a las siguientes cuestiones:

- repite el cálculo de los parámetros (número de instrucciones y evaluación de las prestaciones del sistema) obtenidos en el apartado anterior, pero, en este caso, para la versión vectorial del bucle

Repetimos nuevamente el apartado de "Número de instrucciones" y "Estimación de las prestaciones del sistema"

Número de instrucciones

- Número de instrucciones de carga ejecutadas (Nc)

3 instrucciones de carga => 1 de carga escalar y 2 de carga vectorial

La de carga escalar => ld f0, a(r0)

Las de carga vectorial => lv v0, 0(r1) y lv v2, 0(r2)

Nc = 3

- Número de instrucciones de almacenamiento ejecutadas (Nl)

1 instrucción de almacenamiento vectorial => sv 0 (r3), v3

Nl = 1

- Número de instrucciones escalares en coma flotante ejecutadas (Nfp)

0 instrucciones escalares en coma flotante ejecutadas

Nfp = 0

- Número de instrucciones vectoriales ejecutadas (Nv)

2 instrucciones vectoriales

1-multsv v1, f0, v0

2-addv v3, v1, v2

Nv = 2

- Número total de instrucciones ejecutadas NT =12
- Comprueba si se cumple que `NT = Nc + Nl + Nfp + Nv` y justifica en cualquier caso tu respuesta

NT = Nc + Nl + Nfp + Nv = 3 + 1 + 0 + 2 = 6

No se cumple, porque no estamos contando las instrucciones de suma/resta con enteros, la instrucción *"movi2s vlr, r4"*, ni la instrucción trap final que sí que tiene en cuenta el simulador.

Nº operaciones suma/resta enteros = 4

Instrucción movi2s vlr, r4 = 1

Trap = 1

Con lo que no hemos tenido en cuenta estas 3 anteriores, que juntas suman 6.

Ahora si se cumple => 6 + 6 = NT = 12

Estimación de las prestaciones del sistema

- Valor del CPI al final de la ejecución

Ciclos = 315 ciclos

Instrucciones = 12 instrucciones

CPI = 315 / 12 = 26,250 Ciclos de reloj / instrucción.

- Número de ciclos de espera totales

Los ciclos de espera debido a las dependencias RAW son = 227

- Número de veces que se ejecuta el bucle

En esta configuración de ejecución en modo vectorial no existe bucle como tal, pues el bucle está codificado para ejecutarlo en un procesador vectorial, donde se supone que en este ejemplo del DAXPY el número de elementos o longitud de un registro vectorial coincide con la longitud de la operación vectorial (MVLR = Nº componentes del vector = 64)

- Número de ciclos de espera introducidos por cada pasada del bucle

No hay bucle

- Número de ciclos de espera introducidos antes de entrar en el bucle

No hay bucle

- ¿A qué se deben las detenciones que se producen en el anterior código?

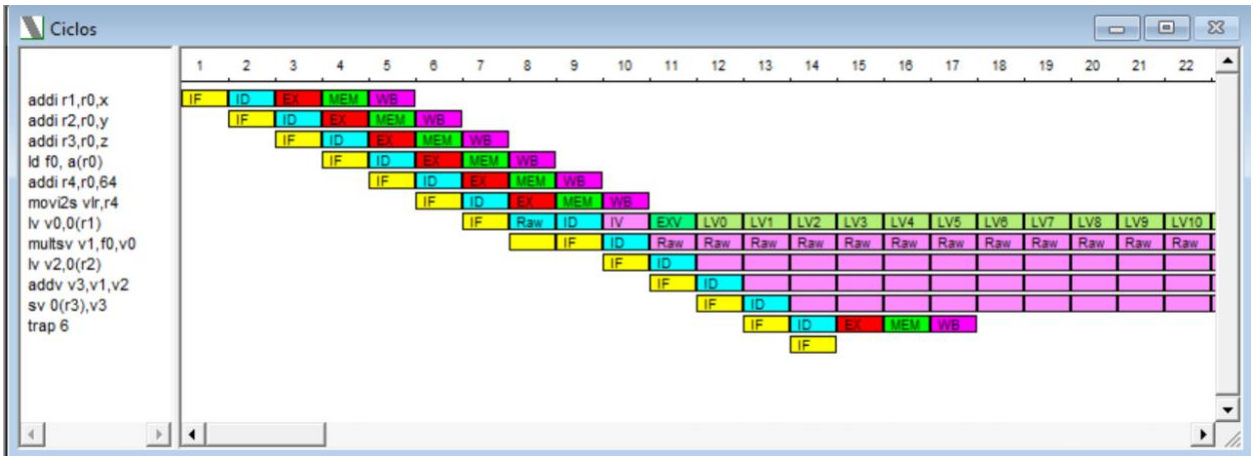
Las detenciones producidas se deben a riesgos de dependencia de datos RAW.

- ¿Se puede configurar el simulador de forma que se minimicen el número de detenciones? ¿cómo?

Si, podríamos utilizar la configuración de encadenamiento vectorial. Esta configuración permite que un cauce pueda empezar a procesar los resultados que produce otro cauce a medida que éste va terminando de procesar componentes (el segundo cauce no tiene que esperar a que termine el procesamiento de todo el operando vectorial en el primer cauce).

Ahora analiza qué sucede realmente durante la ejecución de las instrucciones vectoriales, en su correspondiente cauce vectorial. Para ello deberás contestar razonadamente a las siguientes preguntas (incluye, si quieres, capturas de pantalla):

- ¿qué tipo de riesgo detiene la segmentación en el ciclo 8 y por qué?

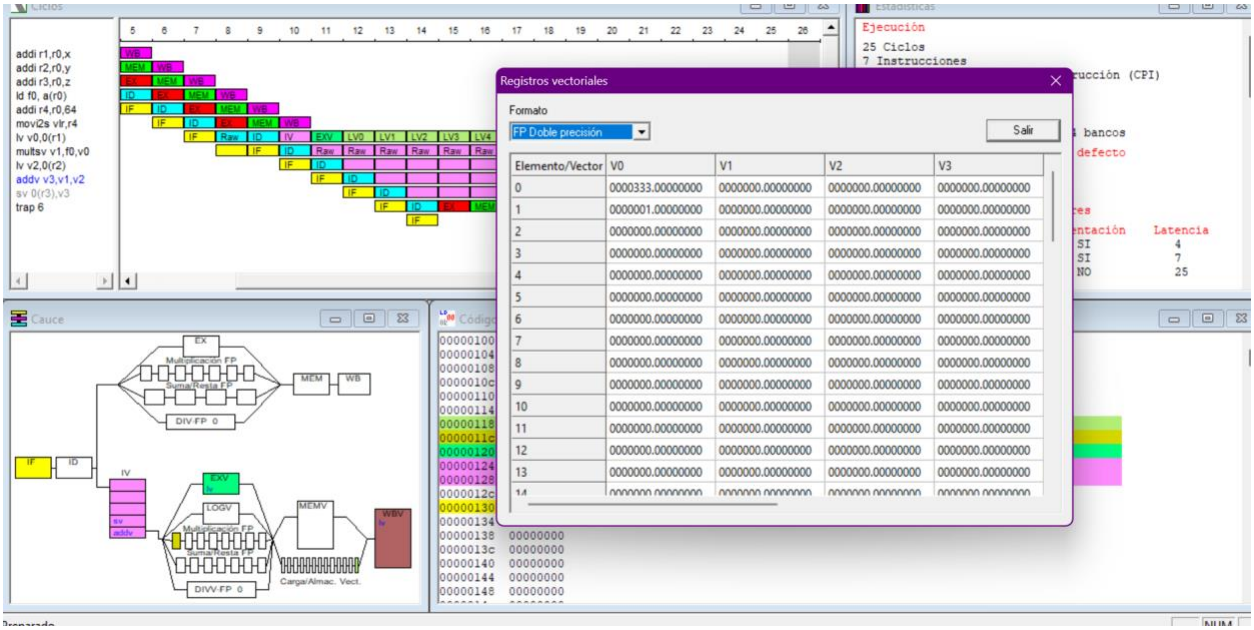


En el ciclo 8, el riesgo que detiene la segmentación es un riesgo de datos RAW (Read After Write), también conocido como riesgo de dependencia de datos. Este riesgo ocurre porque la instrucción movi2s vlr, r4, que se encuentra en la etapa MEM (acceso a memoria), debe completar su escritura en el registro especial vlr antes de que la instrucción lv v0, 0(r1) pueda usar el valor actualizado de vlr para realizar su operación de carga. Hasta que la instrucción movi2s no complete su acceso a memoria y la escritura en el registro vlr, la instrucción lv debe esperar, lo que causa la detención en la segmentación.

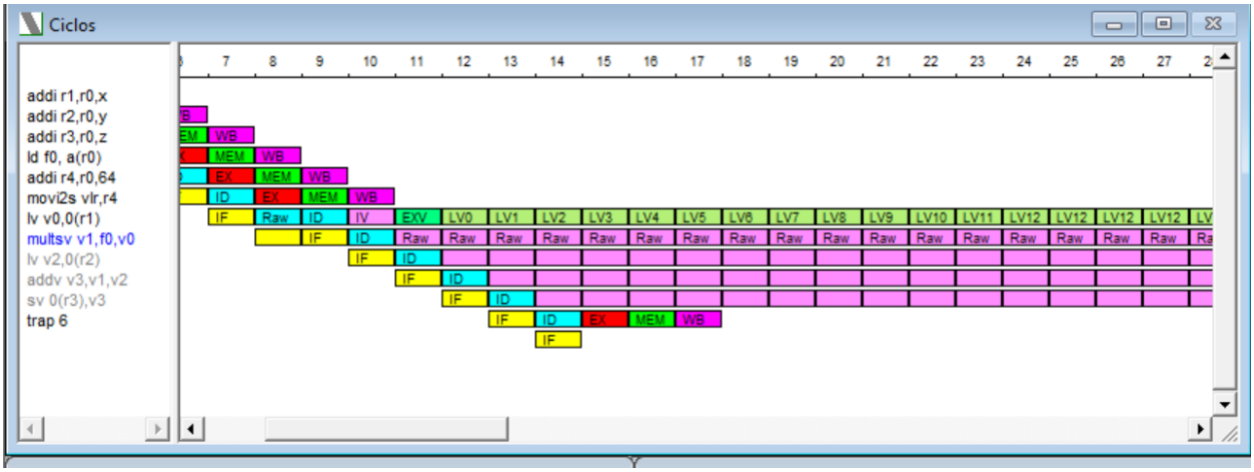
- ¿qué sucede a partir del ciclo 11 entre las instrucciones de carga vectorial y multiplicación?

A partir del ciclo 11, sucede que comienza la ejecución de la instrucción de carga vectorial 'lv v0, 0(r1)', y la multiplicación 'multsv v1, f0, v0' no puede empezar a ejecutarse hasta que la de carga vectorial termine, por riesgos de dependencia RAW. Por lo que a partir del ciclo 11, la instrucción de multiplicación queda a la espera de la de carga vectorial, insertando ciclos de espera.

- ¿en qué ciclo empiezan a cargarse efectivamente las componentes del vector `X` en `V0`? ¿por qué? (fíjate en la ventana de acceso al banco de registros vectoriales)



En esta captura de los registros vectoriales se ve como se empiezan a cargarse las componentes a partir del ciclo 25 (cambié el primer valor a 333 única y exclusivamente para que se pueda visualizar bien en la captura cuando se carga el primer valor, antes era un 0 y no se diferenciaba bien).



La razón por la que empiezan a cargarse las componentes a partir del ciclo 25 reside en la latencia de 13 ciclos de la UF de carga/almacenamiento vectorial. Tras la decodificación en la instrucción 'lv' en el ciclo 9.

- ¿cuándo termina la carga del vector `V0`? ¿Por qué?

La carga del vector V0 termina en el ciclo 88, 63 ciclos después del primer elemento, porque terminan de cargarse todas las componentes.

- ¿se detiene la segmentación en algún punto durante la carga de `V0`? ¿Por qué?

Se detiene la segmentación durante toda la carga de V0, ya que la siguiente instrucción de multiplicación vectorial presenta riesgos de dependencia de datos RAW y debe esperar.

- ¿qué sucede durante la carga del registro vectorial `V2`

Durante la carga del registro vectorial V2, está habiendo un riesgo por dependencia de datos RAW, que inserta ciclos de parada para la operación de suma vectorial. Además, destacamos que esta operación de carga se realiza paralelamente (ya que no existe ningún tipo de riesgo que lo impida) junto con la operación de multiplicación anterior, pero que está terminará antes, 5 ciclos antes, ya que la latencia de la UF de multiplicación es menor en que la de carga/almacenamiento en esa cantidad.

- ¿cuántos ciclos se consumen durante la operación de suma vectorial? ¿Por qué?

La instrucción de suma vectorial se capta en el ciclo 11, pero no es hasta el ciclo 89 que no comienza su ejecución, por razones de riesgos de dependencia de datos comentadas anteriormente.

Comienza la carga en V3 a partir del ciclo 174, que es cuando acaba la latencia de la UF de suma/resta vectorial FP (, y termina nuevamente 63 ciclos después (en el ciclo 237), cargando las componentes restantes de V3.

Por lo tanto, la suma vectorial es una instrucción que dura desde el ciclo 11 hasta el 238 (237 + 1 de la etapa WBV), en total pasan 227 ciclos desde que se capta.

Para terminar este apartado:

- repite el cálculo de los parámetros (número de instrucciones y evaluación de las prestaciones del sistema) obtenidos en el apartado anterior pero, esta vez, configurando el simulador con la opción de `'Encadenamiento vectorial'`

Número de instrucciones

  - Número de instrucciones de carga ejecutadas (Nc)

3 instrucciones de carga => 1 de carga escalar y 2 de carga vectorial

La de carga escalar => ld f0, a(r0)

Las de carga vectorial => lv v0, 0(r1) y lv v2, 0(r2)

Nc = 3

- Número de instrucciones de almacenamiento ejecutadas (Nl)

1 instrucción de almacenamiento vectorial => sv 0 (r3), v3

Nl = 1

- Número de instrucciones escalares en coma flotante ejecutadas (Nfp)

0 instrucciones escalares en coma flotante ejecutadas

Nfp = 0

- Número de instrucciones vectoriales ejecutadas (Nv)

2 instrucciones vectoriales

1-multsv v1, f0, v0

2-addv v3, v1, v2

Nv = 2

- Número total de instrucciones ejecutadas NT =12
- Comprueba si se cumple que `NT = Nc + Nl + Nfp + Nv` y justifica en cualquier caso tu respuesta

`NT = Nc + Nl + Nfp + Nv = 3 + 1 + 0 + 2 = 6`

`No se cumple, porque no estamos contando las instrucciones de suma/resta con enteros, la instrucción "movi2s vlr, r4", ni la instrucción trap final que sí que tiene en cuenta el simulador.`

`Nº operaciones suma/resta enteros = 4`

`Instrucción movi2s vlr, r4 = 1`

Trap = 1

Con lo que no hemos tenido en cuenta estas 3 anteriores, que juntas suman 6.

Ahora si se cumple => 6 + 6 = NT = 12

Apartado idéntico al de la configuración anterior, ya que el código sigue siendo el mismo, lo que cambiará son las prestaciones:

Estimación de las prestaciones del sistema

- Valor del CPI al final de la ejecución

Ciclos = 216 ciclos

Instrucciones = 12 instrucciones

CPI = 216 / 12 = 18,000 Ciclos de reloj / instrucción.

- Número de ciclos de espera totales

Los ciclos de espera debido a las dependencias RAW son = 87

Los ciclos de espera debido a riesgos estructurales (entre sv 0(r3), v3 y lv v2, 0(r2), pero no sé porque sv no hace LV11, desde el ciclo 110 hasta el 150) = 41 ciclos

Ciclos de espera totales = 87 + 41 = 128 ciclos

- Número de veces que se ejecuta el bucle

En esta configuración de ejecución en modo vectorial no existe bucle como tal, pues el bucle está codificado para ejecutarlo en un procesador vectorial, donde se supone que en este ejemplo del DAXPY el número de elementos o longitud de un registro vectorial coincide con la longitud de la operación vectorial (MVLR = Nº componentes del vector = 64)

- Número de ciclos de espera introducidos por cada pasada del bucle

No hay bucle

- Número de ciclos de espera introducidos antes de entrar en el bucle

No hay bucle

- ¿A qué se deben las detenciones que se producen en el anterior código?

Como he comentado en el apartado anterior, las detenciones producidas se deben a riesgos de dependencia de datos RAW y a riesgos estructurales (por acceso a UF de carga/almacenamiento vectorial)

- ¿Se puede configurar el simulador de forma que se minimicen el número de detenciones? ¿cómo?

El simulador está configurado con encadenamiento vectorial, adelantamiento de resultados, además en la operación DAXPY el número de elementos o longitud de un registro vectorial coincide con la longitud de la operación vectorial (MVLR = N.º componentes del vector = 64). No existe ninguna configuración del DLXV que mejore las prestaciones (más allá de disminuir las latencias, tamaño del cauce etc.)

4.3 ANÁLISIS COMPARATIVO

Finalmente, analiza la mejora de las prestaciones obtenida al comparar entre la ejecución en el DLXV de las dos versiones del código (la escalar y la vectorial) para una configuración optimizada del simulador (la que mejores resultados haya proporcionado, en cada caso). Contesta justificadamente a las siguientes cuestiones:

- a la vista del CPI obtenido en cada caso ¿qué configuración del DLXV y sobre qué código se obtienen las mejores prestaciones?

4.3.1. VERSIÓN ESCALAR *OPTIMIZADA (con reordenación de código y salto retardado)*

Ciclos = 1290 ciclos

N. º Instrucciones = 646 instrucciones

CPI = 1290 / 646 = 1,997 ciclos / instrucción.

4.3.2. VERSIÓN VECTORIAL *OPTIMIZADA (con encadenamiento vectorial)*

Ciclos = 216 ciclos

Instrucciones = 12 instrucciones

CPI = 216 / 12 = 18 Ciclos de reloj / instrucción.

La versión escalar optimizada del código en el procesador DLXV tiene un CPI de aproximadamente 2 ciclos por instrucción, lo que indica una eficiencia razonablemente alta para una ejecución escalar. Por otro lado, la versión vectorial optimizada muestra un CPI mucho más alto de 18 ciclos por instrucción. Sin embargo, el número total de ciclos para la ejecución vectorial es significativamente menor, lo que sugiere una ejecución mucho más rápida a pesar del CPI elevado.

Esto se debe a la drástica reducción en el número de instrucciones que se necesitan para realizar la misma tarea, la propia motivación del procesamiento vectorial sugiere esto, aprovechar el paralelismo de datos y aumentar significativamente la velocidad de procesamiento, de modo que una sola instrucción vectorial es capaz de codificar un bucle entero.

Por lo tanto, la configuración del DLXV que ofrece las mejores prestaciones es la versión vectorial optimizada, ya que completa la tarea en muchos menos ciclos en total.

- si queremos evaluar las prestaciones del procesador vectorial mediante la ejecución del benchmark de Linpack, ¿qué configuración de DLXV recomendarías para obtener la velocidad de pico?

Para evaluar las prestaciones del procesador vectorial DLXV mediante el benchmark de Linpack, que intensivamente realiza cálculos sobre vectores y matrices, recomendaría una configuración que maximice el uso del paralelismo de datos. Esto incluiría:

1. Un MVL (Maximum Vector Length) grande para manejar grandes vectores y minimizar la fragmentación de operaciones.
2. Unidades de carga y almacenamiento vectorial completamente segmentadas para una transferencia eficiente de datos.
3. Uso de encadenamiento vectorial para reducir la latencia entre operaciones consecutivas.
4. Optimización del uso de memoria entrelazada para acelerar el acceso a datos vectoriales.

Estas características permitirían a DLXV alcanzar su velocidad de pico en un benchmark diseñado para aprovechar el procesamiento vectorial.

6 CONCLUSIÓN

En conclusión, este estudio demuestra cómo la implementación vectorial en un simulador de procesador DLXV supera significativamente a la versión escalar en términos de rendimiento. Aunque el CPI de la versión vectorial es más alto, la reducción drástica en el número total de instrucciones necesarias resulta en una ejecución más rápida y eficiente. Esto subraya la importancia del paralelismo de datos en el procesamiento vectorial, resaltando la eficacia de las instrucciones vectoriales en comparación con las escalares para ciertas tareas computacionales.

7 BIBLIOGRAFÍA

- J. Ortega, M. Anguita, A. Prieto, *Arquitectura de Computadores,* Ed.Thomson
- P. de Miguel, *Fundamentos de Computadores*, Ed. Paraninfo
- C. Hamacher, Z. Vranesic, S.G. Zaky, *Computer Organization*
- *Pau Micó Tormos, Apuntes disponibles en Poliformat, UPV, 2023. Última consulta 29 de Diciembre de 2023*