

WinDLXV

Manual de usuario

Septiembre, 2005

Índice general

Introducción	11
Usuarios de este manual	11
Requisitos mínimos	12
Instalación	12
Archivos del programa	13
1. Conocer el entorno del simulador	15
1.1. Barra de título	15
1.2. Barra de menús	16
1.3. Barra de iconos	17
1.4. Barra de estado	17
1.5. Área de trabajo	17
2. Ventanas	19
2.1. Código	19
2.2. Datos	20
2.3. Registros	21
2.4. Cauce	23
2.5. Ciclos	26
2.6. Estadísticas	27
2.7. Ventana Entrada/Salida	28
3. Menús	31
3.1. Menú Ficheros	31
3.1.1. Cargar un programa en el simulador	32
3.1.2. Recargar un programa en el simulador	34
3.1.3. Reinicializar el procesador DLXV	34
3.1.4. Reinicializar el procesador DLXV sin cargar ningún programa	35
3.1.5. Salir de WinDLXV	35
3.2. Menú Ejecutar	35

Índice general

3.2.1.	Ejecutar un único ciclo de reloj	36
3.2.2.	Ejecutar varios ciclos de reloj	36
3.2.3.	Ejecutar sin pausas	36
3.2.4.	Parar la simulación	36
3.2.5.	Visualizar la ventana de Entrada/Salida	37
3.3.	Menú Configuración	37
3.3.1.	Modificar la arquitectura del procesador	38
3.3.2.	Definir número de ciclos a ejecutar en la simulación de múltiples ciclos	39
3.3.3.	Definir comportamiento ante un salto	40
3.3.4.	Activar adelanto de resultados	40
3.3.5.	Activar encadenamiento vectorial	40
3.3.6.	Guardar configuración	40
3.3.7.	Recuperar configuración	41
3.4.	Menú Memoria	41
3.4.1.	Visualizar memoria	42
3.4.2.	Modificar memoria	43
3.4.3.	Visualizar símbolos	45
3.4.4.	Ensamblar	45
3.5.	Menú Registros	46
3.5.1.	Registros escalares de propósito general	47
3.5.2.	Registros escalares de punto flotante	48
3.5.3.	Registros vectoriales	49
3.5.4.	Registros especiales	50
3.6.	Menú Herramientas	51
3.6.1.	Carpeta de datos	51
3.6.2.	Editor	52
3.7.	Menú Ventanas	54
3.8.	Menú ?	55
3.8.1.	Ayuda de WinDLXV	55
3.9.	Acceso rápido a las funciones de menú	56
4.	Escritura de un programa	59
5.	Simulación paso a paso	63
5.1.	Escritura de un programa	63
5.2.	Carga de un programa	64
5.3.	Definiendo la arquitectura del procesador	66
5.4.	Ejecución ciclo a ciclo	66
5.5.	Definiendo puntos de ruptura	68
5.6.	Modificando componentes durante la simulación	70

6. Ensamblador DLXV	71
6.1. Directivas del ensamblador	71
6.2. Sintaxis del lenguaje ensamblador	76
6.3. Instrucciones	76
6.3.1. Instrucciones de transferencias de datos	76
6.3.2. Instrucciones aritméticas/lógicas	77
6.3.3. Instrucciones de control de flujo	77
6.3.4. Instrucciones de punto flotante	78
6.3.5. Instrucciones vectoriales	78
6.4. Formato de las instrucciones	79
7. Juego de instrucciones del DLXV	81
7.1. Instrucciones de trasferencias de datos	81
7.2. Instrucciones aritméticas/lógicas	89
7.3. Instrucciones de control de flujo	99
7.4. Instrucciones de punto flotante	102
7.5. Instrucciones vectoriales	109
8. Traps	121
8.1. Fin de la ejecución del programa	121
8.2. Apertura de un fichero	122
8.3. Cierre de un fichero	123
8.4. Lectura de un fichero o de la entrada estándar	124
8.5. Escritura de un fichero	125
8.6. Escritura formateada por la salida estándar	126

Índice de figuras

1.1. Ventana principal WinDLXV.	15
1.2. Barra de iconos.	17
2.1. Ventana <i>Código</i>	19
2.2. Ventana <i>Datos</i>	20
2.3. Modificar memoria desde la Ventana <i>Datos</i>	21
2.4. Ventana <i>Registros</i>	22
2.5. Modificar un registro desde la Ventana <i>Registros</i>	23
2.6. Ventana <i>Cauce</i>	24
2.7. Ventana <i>Ciclos</i>	26
2.8. Ventana <i>Estadísticas</i>	27
2.9. Ventana Entrada/Salida.	28
3.1. Menú <i>Ficheros</i>	31
3.2. Caja de diálogo para cargar un programa en el simulador.	32
3.3. Carga de un programa en WinDLXV.	34
3.4. Menú <i>Ejecutar</i>	36
3.5. Menú <i>Configuración</i>	37
3.6. Modificar la arquitectura del procesador simulado.	38
3.7. Caja de diálogo para guardar configuración del procesador DLXV simulado.	41
3.8. Menú <i>Memoria</i>	42
3.9. Visualizar la memoria.	43
3.10. Modificar una posición de memoria.	44
3.11. Visualizar símbolos.	45
3.12. Ensamblar.	46
3.13. Menú <i>Registros</i>	46
3.14. Registros de propósito general.	47
3.15. Registros de punto flotante.	48
3.16. Registros vectoriales.	49
3.17. Registros especiales.	50

Índice de figuras

3.18. Menú <i>Herramientas</i>	51
3.19. Caja de diálogo para modificar la carpeta de datos.	52
3.20. Editor.	53
3.21. Menú <i>Ventanas</i>	55
3.22. Menú ?.	55
5.1. Simulando paso a paso: edición.	64
5.2. Simulando paso a paso: carga de un programa en WinDLXV.	65
5.3. Simulando paso a paso: ejecución de un programa ciclo a ciclo.	68
5.4. Simulando paso a paso: añadiendo un punto de ruptura.	69
5.5. Simulando paso a paso: parada en un punto de ruptura.	70
6.1. Formato de las instrucciones para DLXV.	79

Índice de tablas

3.2. Acceso rápido a las funciones más frecuentes.	57
--	----

Introducción

WinDLXV (*Windows DeLuXe Vectorial*) es una aplicación bajo entorno Windows que simula el cauce del procesador DLXV. Este procesador descrito en [Hen96] es una máquina vectorial que tiene como parte escalar la propia de DLX, procesador también descrito en [Hen96], y como parte vectorial la extensión lógica de DLX.

Esta aplicación de simulación permite la creación, carga y posterior ejecución de programas escritos en ensamblador DLXV, pudiéndose observar la evolución de la simulación y la interacción entre los diferentes componentes de la arquitectura DLXV (cauce, registros, memoria, dispositivo entrada/salida, etc.).

La aplicación permite además, la modificación de la arquitectura DLXV simulada: si las unidades funcionales son segmentadas o no, tiempos de latencia de las unidades funcionales, adelanto de resultados en el cauce, comportamiento del procesador ante una instrucción de salto, estructura de la memoria, etc. Así, como el contenido de otros de sus componentes (memoria, valor de los registros) mientras se desarrolla la ejecución del programa.

El usuario tiene la posibilidad de hacer su simulación de tres formas diferentes, ciclo a ciclo de reloj, por varios ciclos de reloj, o ejecución sin pausas (hasta el final del programa o hasta un punto de ruptura previamente definido por él). Después de cada paso se actualizará los contenidos de los componentes que fueron alterados durante el ciclo o ciclos respectivos, con lo cual se logra un perfecto entendimiento de todos los pasos que conllevan la ejecución de una instrucción en un procesador vectorial segmentado.

Usuarios de este manual

Este manual está destinado a los usuarios del simulador, que normalmente serán estudiantes que desean hacer prácticas para profundizar en

los conceptos de segmentación de cauce (*pipelining*) y procesadores vectoriales. Por ello, es imprescindible el estudio previo de estos conceptos en las clases teóricas.

El manual está disponible desde el simulador, a través del comando de ayuda. Así, el estudiante podrá obtener en cualquier momento la información deseada para el manejo del simulador.

Requisitos mínimos

WinDLXV debe instalarse en un ordenador con la siguiente configuración mínima:

- Ordenador personal Pentium o superior.
- 64 Mb de memoria RAM.
- Unidad de CD.
- En el ordenador debe estar instalado el sistema operativo Microsoft Windows, en sus versiones 98SE, Milenium, 2000, NT 4.0, XP o 2003 Server.
- Tarjeta gráfica SVGA color (800 x 600) o superior, compatible con MS-Windows.
- Ratón compatible con MS-Windows.

Para la utilización de WinDLXV, se requiere su instalación previa en el disco duro del ordenador.

Instalación

Para instalar el programa WinDLXV, se siguen los siguientes pasos:

- Se introduce el CD-ROM en la unidad.
- Se ejecuta D:setup.exe (siendo D: la unidad en la que se encuentra el CD-ROM).

El resto de pasos a seguir se van describiendo en pantalla. Una vez seguido este proceso, el programa se encuentra ya instalado en el disco duro del ordenador.

Si la instalación del programa se ha realizado correctamente, WinDLXV aparecerá en el menú de inicio de Windows, desde donde se puede ejecutar (Inicio▷ Programas▷ WinDLXV). De igual forma, se habrá creado un acceso directo desde el escritorio de Windows.

Archivos del programa

Los archivos necesarios para el funcionamiento de WinDLXV, y que se instalan en el disco duro en el momento de la instalación o se crean posteriormente, son:

- Archivo del programa:

WinDLXV.exe

- Archivos de configuración del programa:

windlxv.ini almacena la configuración de la arquitectura simulada.

windlxv.las almacena el último programa almacenado correctamente en el simulador.

windlxv.pth almacena la carpeta de datos.

- Archivos de ejemplo

En la carpeta programas se incluye una serie de ejemplos de programas escritos en ensamblador para cargar en el simulador WinDLXV.

- Archivos de ayuda

WinDLXV.pdf manual de usuario.

Capítulo 1

Conocer el entorno del simulador

La ventana del programa WinDLXV como muestra la Figura 1.1 consta de las siguientes partes:

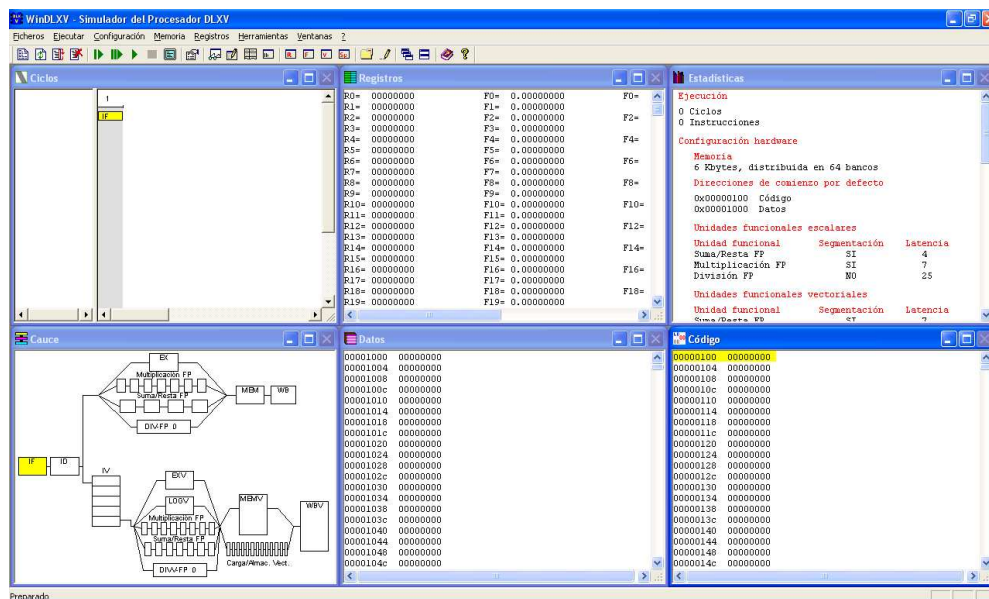


Figura 1.1: Ventana principal WinDLXV.

1.1. Barra de título

Situada en la parte superior de la ventana, tiene como título el nombre del programa "WinDLXV - Simulador del Procesador DLXV" y el del pro-

grama seleccionado para simular. A la derecha de esta barra aparecen los iconos comunes a todos los programas desarrollados para Windows:



Minimizar la aplicación, para retirarla momentáneamente de la pantalla.



Restaurar el tamaño de la ventana de aplicación.



Maximizar el tamaño de la ventana de aplicación.



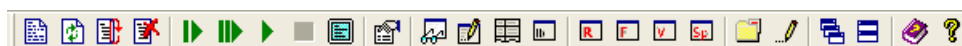
Cerrar la aplicación (salir del programa).

1.2. Barra de menús

Bajo la barra de título, se encuentra la barra de menús desplegables. Los menús disponibles son los siguientes:

Ficheros	En este menú se encuentran todas las funciones necesarias para gestionar los programas a simular (cargar, recargar), para reinicializar el procesador DLXV y simulador, y para salir de la aplicación.
Ejecutar	En este menú se encuentran las funciones que permiten ejecutar el código ensamblado en la memoria y visualizar la Ventana E/S.
Configuración	En este menú se encuentran las funciones para configurar la arquitectura DLXV a simular, y gestionar los ficheros de configuración de la arquitectura simulada.
Memoria	En este menú se encuentran las funciones que permiten visualizar y modificar la memoria de código y datos, así como visualizar los símbolos definidos en el programa ensamblado en la memoria.
Registros	En este menú se encuentran las funciones que permiten visualizar y modificar los registros del procesador: registros escalares de propósito general, registros escalares de punto flotante, registros vectoriales y registros especiales.

REFERENCES



0

Table 1

© 2006 The Authors
Journal compilation © 2006 Blackwell Publishing Ltd

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 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 207 208 209 210 211 212 213 214 215 216 217 218 219 220 221 222 223 224 225 226 227 228 229 230 231 232 233 234 235 236 237 238 239 240 241 242 243 244 245 246 247 248 249 250 251 252 253 254 255 256 257 258 259 260 261 262 263 264 265 266 267 268 269 270 271 272 273 274 275 276 277 278 279 280 281 282 283 284 285 286 287 288 289 290 291 292 293 294 295 296 297 298 299 300 301 302 303 304 305 306 307 308 309 310 311 312 313 314 315 316 317 318 319 320 321 322 323 324 325 326 327 328 329 330 331 332 333 334 335 336 337 338 339 340 341 342 343 344 345 346 347 348 349 350 351 352 353 354 355 356 357 358 359 360 361 362 363 364 365 366 367 368 369 370 371 372 373 374 375 376 377 378 379 380 381 382 383 384 385 386 387 388 389 390 391 392 393 394 395 396 397 398 399 400 401 402 403 404 405 406 407 408 409 410 411 412 413 414 415 416 417 418 419 420 421 422 423 424 425 426 427 428 429 430 431 432 433 434 435 436 437 438 439 440 441 442 443 444 445 446 447 448 449 450 451 452 453 454 455 456 457 458 459 460 461 462 463 464 465 466 467 468 469 470 471 472 473 474 475 476 477 478 479 480 481 482 483 484 485 486 487 488 489 490 491 492 493 494 495 496 497 498 499 500 501 502 503 504 505 506 507 508 509 510 511 512 513 514 515 516 517 518 519 520 521 522 523 524 525 526 527 528 529 530 531 532 533 534 535 536 537 538 539 540 541 542 543 544 545 546 547 548 549 550 551 552 553 554 555 556 557 558 559 560 561 562 563 564 565 566 567 568 569 570 571 572 573 574 575 576 577 578 579 580 581 582 583 584 585 586 587 588 589 590 591 592 593 594 595 596 597 598 599 600 601 602 603 604 605 606 607 608 609 610 611 612 613 614 615 616 617 618 619 620 621 622 623 624 625 626 627 628 629 630 631 632 633 634 635 636 637 638 639 640 641 642 643 644 645 646 647 648 649 650 651 652 653 654 655 656 657 658 659 660 661 662 663 664 665 666 667 668 669 670 671 672 673 674 675 676 677 678 679 680 681 682 683 684 685 686 687 688 689 690 691 692 693 694 695 696 697 698 699 700 701 702 703 704 705 706 707 708 709 710 711 712 713 714 715 716 717 718 719 720 721 722 723 724 725 726 727 728 729 730 731 732 733 734 735 736 737 738 739 740 741 742 743 744 745 746 747 748 749 750 751 752 753 754 755 756 757 758 759 760 761 762 763 764 765 766 767 768 769 770 771 772 773 774 775 776 777 778 779 780 781 782 783 784 785 786 787 788 789 790 791 792 793 794 795 796 797 798 799 800 801 802 803 804 805 806 807 808 809 810 811 812 813 814 815 816 817 818 819 820 821 822 823 824 825 826 827 828 829 830 831 832 833 834 835 836 837 838 839 840 841 842 843 844 845 846 847 848 849 850 851 852 853 854 855 856 857 858 859 860 861 862 863 864 865 866 867 868 869 870 871 872 873 874 875 876 877 878 879 880 881 882 883 884 885 886 887 888 889 890 891 892 893 894 895 896 897 898 899 900 901 902 903 904 905 906 907 908 909 910 911 912 913 914 915 916 917 918 919 920 921 922 923 924 925 926 927 928 929 930 931 932 933 934 935 936 937 938 939 940 941 942 943 944 945 946 947 948 949 950 951 952 953 954 955 956 957 958 959 960 961 962 963 964 965 966 967 968 969 970 971 972 973 974 975 976 977 978 979 980 981 982 983 984 985 986 987 988 989 990 991 992 993 994 995 996 997 998 999 1000 1001 1002 1003 1004 1005 1006 1007 1008 1009 1010 1011 1012 1013 1014 1015 1016 1017 1018 1019 1020 1021 1022 1023 1024 1025 1026 1027 1028 1029 1030 1031 1032 1033 1034 1035 1036 1037 1038 1039 1040 1

- Memoria de código.
- Memoria de datos.
- Registros del procesador.
- Cauce.
- Diagrama de ciclos de reloj.
- Estadísticas.

Todas estas ventanas se pueden minimizar o maximizar al estilo de las ventana principal del programa (excepto cerrar). Asimismo, en aquellas que sea aplicable, es posible desplazarse a través de la ventana usando la barra de desplazamiento (*scroll*), o las teclas de posición (\uparrow, \downarrow).

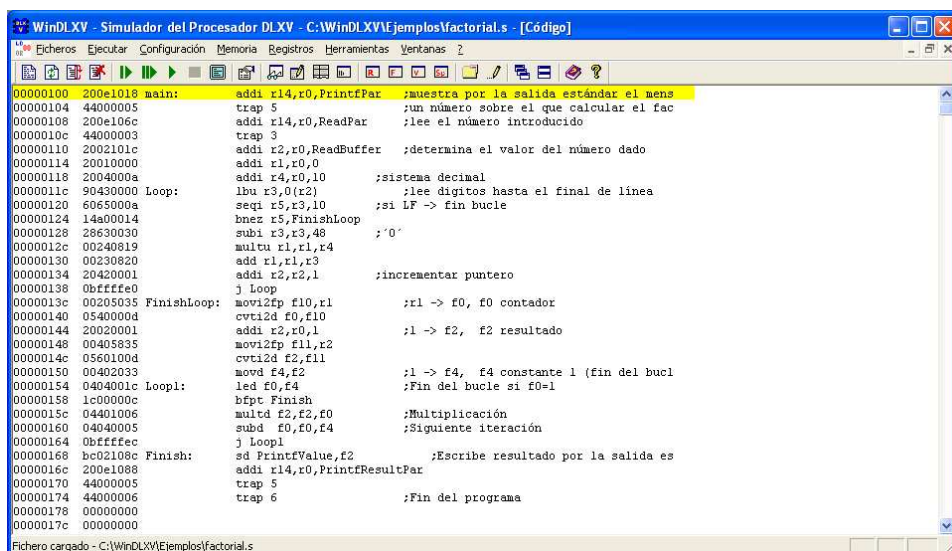
Capítulo 2

Ventanas

2.1. Código

En la ventana *Código* son visualizadas las instrucciones DLX/DLXV (sección .text) que hay almacenadas en memoria junto con sus direcciones. Todas las instrucciones son de 32 bits y deben estar alineadas.

Las instrucciones son mostradas en dos formatos: ensambladas, en hexadecimal, y desensambladas, con los comentarios, etiquetas, etc., que se añadieron en el fichero fuente del programa cargado. La Figura 2.1 muestra el contenido de esta ventana después que se ha cargado un programa.



```
00000100 200e1018 main:      addi r14,r0,PrintfPar ;muestra por la salida estándar el mens
00000104 44000005            trap 5 ;un número sobre el que calcular el fac
00000108 200e106c            addi r14,r0,ReadPar ;lee el número introducido
0000010c 44000003            trap 3
00000110 2002101c            addi r2,r0,ReadBuffer ;determina el valor del número dado
00000114 20010000            addi r1,r0,0
00000118 2004000a            addi r4,r0,10 ;sistema decimal
0000011c 90430000 Loop:      lbu r3,0(r2) ;lee dígitos hasta el final de línea
00000120 6065000a            seqi r5,r3,10 ;si LF -> fin bucle
00000124 14a00014            bnez r5,FinishLoop
00000128 28630030            subi r3,r3,48 ;'0'
0000012c 00240819            multu r1,r1,r4
00000130 00230820            add r1,r1,r3
00000134 20420001            addi r2,r2,1 ;incrementar puntero
00000138 0bfffffe0          j Loop
0000013c 00205035 FinishLoop: movi2fp f10,r1 ;r1 -> f0, f0 contador
00000140 05400004            cvti2d f0,f10
00000144 20020001            addi r2,r0,1 ;l -> f2, f2 resultado
00000148 00405835            movi2fp f11,r2
0000014c 0560100d            cvti2d f2,f11
00000150 00402033            movd f4,f2 ;l -> f4, f4 constante 1 (fin del bucl
00000154 0404001c Loop1:    led f0,f4 ;Fin del bucle si f0=1
00000158 1c00000c            bfpt Finish
0000015c 04401006            multd f2,f2,f0 ;Multiplicación
00000160 04040005            subd f0,f0,f4 ;Siguiente iteración
00000164 0bfffffec          j Loop1
00000168 bc02108c Finish:   sd PrintfValue,f2 ;Escribe resultado por la salida es
0000016c 200e1088            addi r14,r0,PrintfResultPar
00000170 44000005            trap 5
00000174 44000006            trap 6 ;Fin del programa
00000178 00000000
0000017c 00000000
```

Figura 2.1: Ventana *Código*.

Desde esta ventana se puede añadir o eliminar un punto de ruptura (*breakpoint*) en una instrucción de código. Para ello se hará doble clic sobre la instrucción sobre la que se desea añadir el punto de ruptura, y de nuevo doble clic para eliminarlo. La instrucción con un punto de ruptura es mostrada en color *azul*. En la sección 5.5 veremos con más detalle los puntos de ruptura.

Cuando una instrucción está ejecutándose en una etapa determinada del cauce, el color característico de esa etapa es utilizado como color de fondo para esa instrucción.

Durante la ejecución de un programa es posible modificar alguna de las instrucciones a ejecutar. Para ello seleccione la función de menú *Memoria* ▷ *Ensamblar*, para ensamblar una instrucción en una dirección específica, o *Memoria* ▷ *Modificar Memoria* para modificar directamente la instrucción ya ensamblada.

2.2. Datos

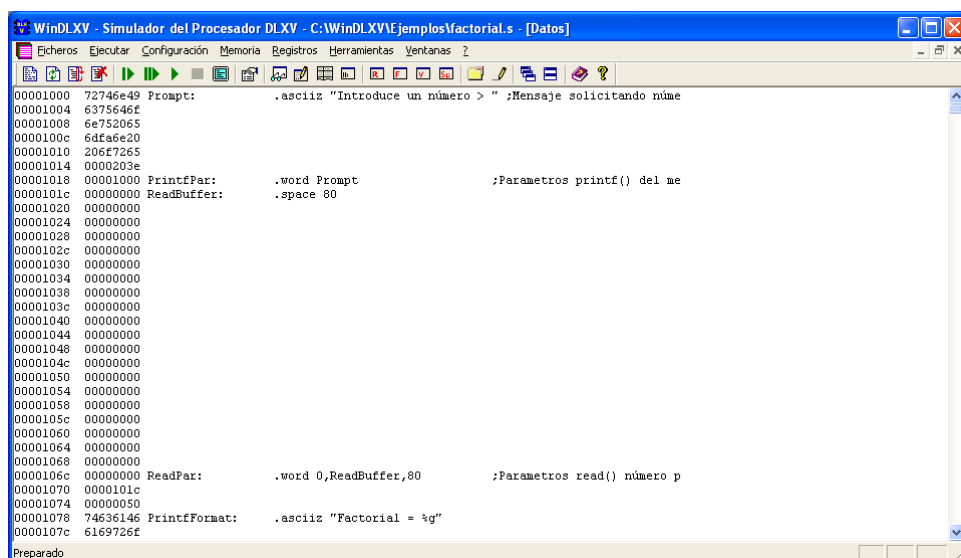


Figura 2.2: Ventana *Datos*.

En la ventada *Datos* son visualizados los datos relativos al programa almacenado en memoria (sección `.data`) junto con sus direcciones. Los datos son mostrados en formato hexadecimal. Adicionalmente se muestra

el contenido del fichero fuente cargado. Veamos en la Figura 2.2 un ejemplo del contenido de esta ventana después de cargado un programa.

Si se desea ver el contenido de la memoria en otro formato, elija la función de menú *Memoria* ▸ *Visualizar Memoria*, donde podrá seleccionar, además del formato hexadecimal, los formatos decimal, ASCII y punto flotante (simple o doble precisión).

Durante la ejecución de un programa se permite modificar el contenido de una posición de memoria, para ello hágase doble clic sobre la posición a modificar, aparecerá una caja de diálogo como la mostrada en la Figura 2.3, desde la que podrá modificar el contenido de un word (4 bytes) en formato hexadecimal. Si se desea en otro formato u otro tamaño, selecciónese la función de menú *Memoria* ▸ *Modificar Memoria*.

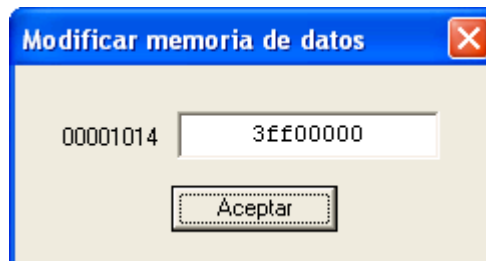
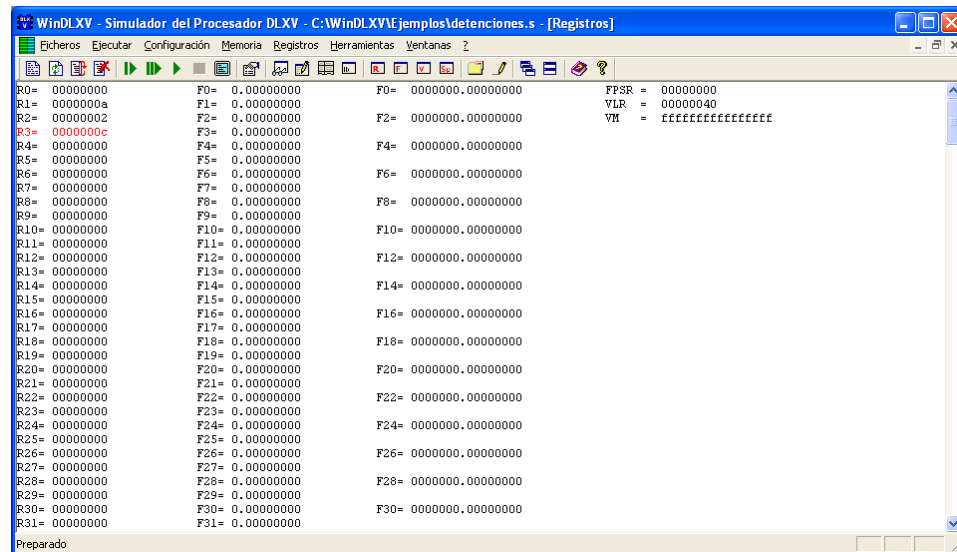


Figura 2.3: Modificar memoria desde la Ventana *Datos*.

2.3. Registros

Desde la ventana *Registros* se puede visualizar el contenido de todos los registros del procesador DLXV como muestra la Figura 2.4. Los registros disponibles son:

- **Registros escalares de propósito general:** son 32 registros (R0, R1, ..., R31) de 32 bits. Se visualizan en formato hexadecimal.
- **Registros escalares de punto flotante:** son 32 registros (F0, F1, ..., F31) de 32 bits si se consideran de simple precisión o 16 registros (F0, F2, ..., F30) si se consideran de doble precisión (parejas par-impar). Se visualizan en los dos formatos, punto flotante en simple y doble precisión.

Figura 2.4: Ventana *Registros*.

- **Registros vectoriales:** son 8 registros (V0, V1, ..., V7), donde cada registro contiene 64 doubles palabras. Se visualizan en formato punto flotante doble precisión.
- **Registros especiales:** son
 1. FPSR (*Floating-Point Status Register*): registro de estado de 1 bit de longitud utilizado para comparaciones y excepciones de punto flotante.
 2. VLR (*Vector-Length Register*): registro de longitud vectorial utilizado para controlar la longitud de cualquier operación vectorial.
 3. VM (*Vector-Mask Register*): registro de máscara que tiene 64 bits, es decir, un bit por cada uno de los elementos de los registros vectoriales, utilizado para enmascarar en las operaciones vectoriales todos aquellos elementos cuyo bit asociado en este registro sea igual a cero.

Si el registro es visualizado en color *gris*, indica que su valor será actualizado por una instrucción que está ejecutándose. Y si está seleccionado adelante de resultados, el registro se muestra en el color característico de la etapa en que su valor está ya disponible.

Desde esta ventana se permite modificar interactivamente el contenido de un registro durante la ejecución de un programa: para ello, hágase doble clic sobre el registro a modificar, aparecerá una caja de diálogo como la mostrada en la Figura 2.5, desde la que se podrá modificar el contenido de ese registro en formato hexadecimal (si se trata de un registro escalar de propósito general), o en punto flotante (si se trata de un registro escalar de punto flotante o vectorial). Si se desea hacer en otro formato, selecciónese la función de menú *Registros* ▸ *función* (según el tipo de registro).

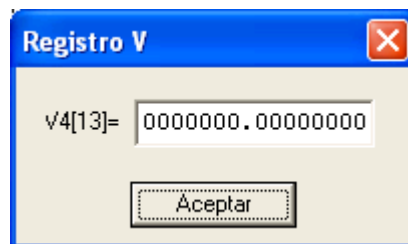


Figura 2.5: Modificar un registro desde la Ventana *Registros*.

Nota: No se podrá modificar el registro de propósito general R0 y los registros especiales (FPSR, VLR y VM).

2.4. Cauce

La ventana *Cauce* muestra el diagrama de las etapas del cauce del procesador DLXV como se muestra en la Figura 2.6. Estas etapas son:

- IF. Búsqueda de instrucción.
- ID. Decodificación de la instrucción.

En esta etapa se comprueba si las instrucciones son escalares o vectoriales. Si son escalares pasan a la etapa de ejecución (EX) del cauce escalar, mientras que si son vectoriales, pasan a la etapa de identificación de instrucciones vectoriales (IV).

Para que una instrucción pueda pasar de la etapa ID a la IV deben comprobarse las dependencias con los datos escalares o registros en coma flotante. Una vez libres de dependencias estos datos son leídos y permanecen asociados a la instrucción vectorial, la cual pasa a la etapa IV.

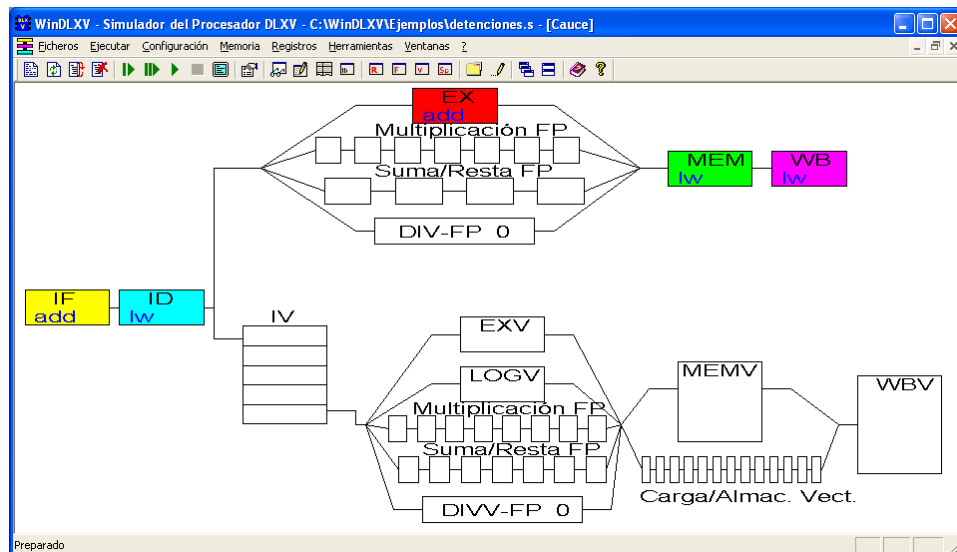


Figura 2.6: Ventana *Cauce*.

Con objeto de reducir los riesgos de control, en esta etapa también es calculada la dirección destino de los saltos condicionales.

- Cauce escalar. Es el propio del procesador DLX.
 - EX. Ejecución. Una vez comprobado que no hay dependencias de datos en la etapa ID, la instrucción escalar pasa a la etapa de ejecución. Esta se compone de las siguientes unidades funcionales:
 1. Unidad entera.
 2. Multiplicador FP.
 3. Sumador FP.
 4. Divisor FP.
 - MEM. Acceso a memoria.
 - WB. Escritura de resultado.
- Cauce vectorial.
 - IV. Identificación de instrucciones vectoriales.

En la etapa IV pueden estar un máximo de N instrucciones vectoriales encoladas, si llega una instrucción N+1, entonces se deberá esperar en ID, deteniendo por tanto todo el cauce.

Las instrucciones vectoriales que se encuentran en IV son emitidas por orden, es decir, la etapa IV se comporta como una cola FIFO, de forma que una instrucción que puede emitirse potencialmente es aquella que llegó antes a la etapa IV. Para que sea posible emitir una instrucción vectorial debe comprobarse:

1. Dependencias estructurales: con la unidad de operación vectorial necesaria y con los puertos de lectura/escritura.
2. Dependencias de datos.

Si se emite con éxito una operación se intentará emitir la siguiente de la etapa IV, y así sucesivamente hasta que no quede ninguna instrucción en IV, o bien, se produzca alguna dependencia y no pueda emitirse la correspondiente instrucción vectorial. Una vez emitida una instrucción vectorial, ésta pasa a la etapa de ejecución.

- EXV. Ejecución vectorial. En cada ciclo de reloj, por cada una de las instrucciones que hay en la etapa de ejecución, se lanza una operación de un elemento individual del vector.

Se compone de las siguientes unidades funcionales:

1. Unidad entera.
 2. Unidad lógica.
 3. Multiplicador FP.
 4. Sumador FP.
 5. Divisor FP.
- MEMV. Acceso a memoria. El procesador DLXV dispone de una unidad de carga/almacenamiento vectorial.
 - WBV. Escritura de resultado.

En la etapa de escritura hay que tener en cuenta si el elemento que se va a escribir resulta enmascarado por el registro de máscara (VM) asociado a la instrucción, y en tal caso inhibir dicha escritura.

Cuando se alcanza el último elemento del vector, definido por el valor del registro de longitud (VLR) asociado a la instrucción vectorial, se da por terminada la ejecución de dicha instrucción.

Durante la ejecución de un programa se muestra también qué instrucción está en cada una de las etapas.

2.5. Ciclos

La ventana *Ciclos* visualiza las operaciones que se realizan en cada ciclo de reloj y en cada etapa. Como puede apreciarse en la Figura 2.7, cada columna representa el estado del cauce en un ciclo de reloj, y la mostrada en color *gris* indica el siguiente ciclo de reloj a ejecutar.

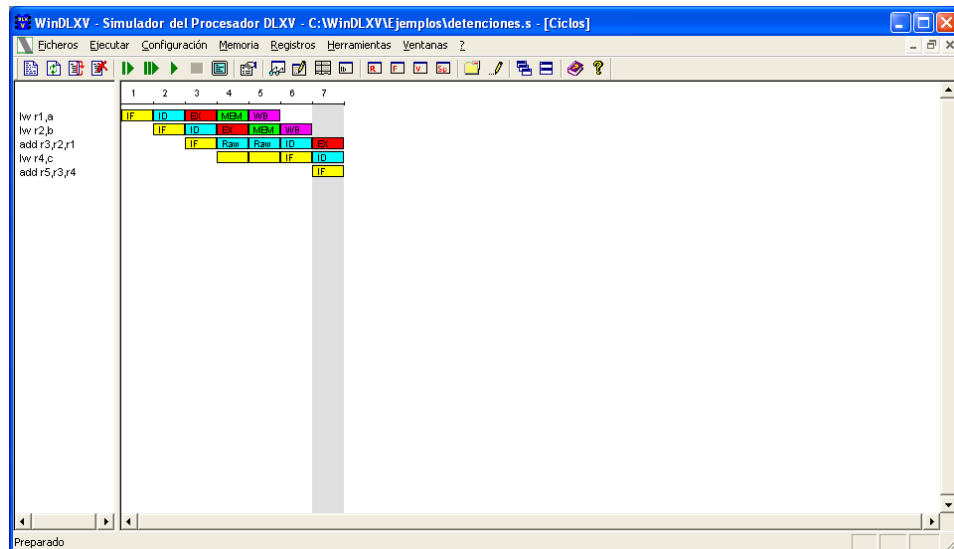


Figura 2.7: Ventana *Ciclos*.

Las detenciones son representadas en cajas coloreadas del color asociado a la etapa detenida. La etiqueta que aparece en el interior de las cajas proporciona más información sobre el tipo de detención:

- Raw: parón por dependencia de datos RAW (lectura después de escritura).
- Waw: parón por dependencia de datos WAW (escritura después de escritura).
- War: parón por dependencia de datos WAR (escritura después de lectura).
- Str: parón estructural (no existen suficientes recursos hardware para ejecutar la instrucción).

La instrucción que causa un parón es mostrada en color *azul*, y las instrucciones detenidas por ella son mostradas en *gris*.

2.6. Estadísticas

La ventana *Estadísticas* visualiza estadísticas sobre la simulación que está siendo realizada. En la Figura 2.8 se muestra como los datos son organizados en los siguientes grupos:

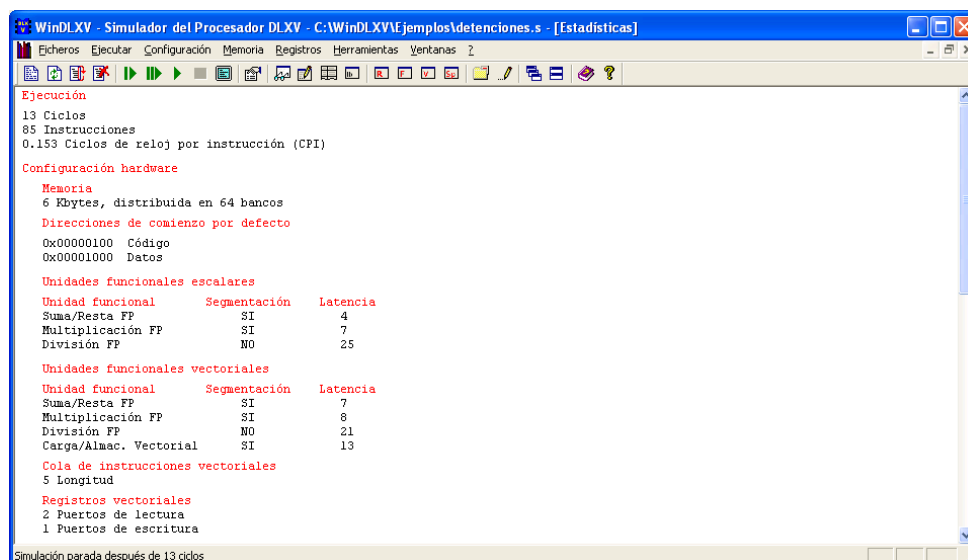


Figura 2.8: Ventana *Estadísticas*.

1. Ejecución: proporciona el número de ciclos consumidos, el número de instrucciones ejecutadas y el número de ciclos por instrucción (CPI).
2. Configuración hardware: contiene la configuración de la arquitectura DLXV con la cual se ha realizado la simulación, memoria, características de las unidades funcionales escalares y vectoriales, si el adelanto de resultados está activado, si existe encadenamiento vectorial, tratamiento de los saltos, etc.
3. Saltos: detalla el número de saltos efectivos y no efectivos realizados.
4. Instrucciones de carga/almacenamiento: detalla el número de cargas y almacenamientos escalares y vectoriales realizados.
5. Instrucciones de punto flotante: proporciona el número total de sumas, multiplicaciones y divisiones de punto flotante.

6. Instrucciones vectoriales: proporciona el número total de sumas, multiplicaciones y divisiones de vectores.
7. Traps: muestra el total de traps realizados.
8. Parones: proporciona el número de parones RAW (lectura después de escritura), WAW (escritura después de escritura), WAR (escritura después de lectura) y estructurales.
9. Estado de la memoria: detalla el estado actual de la memoria (tamaño del programa cargado en bytes y direcciones de comienzo del código y los datos).

2.7. Ventana Entrada/Salida

La ventana de entrada/salida representa el dispositivo estándar de entrada/salida. Se muestra:

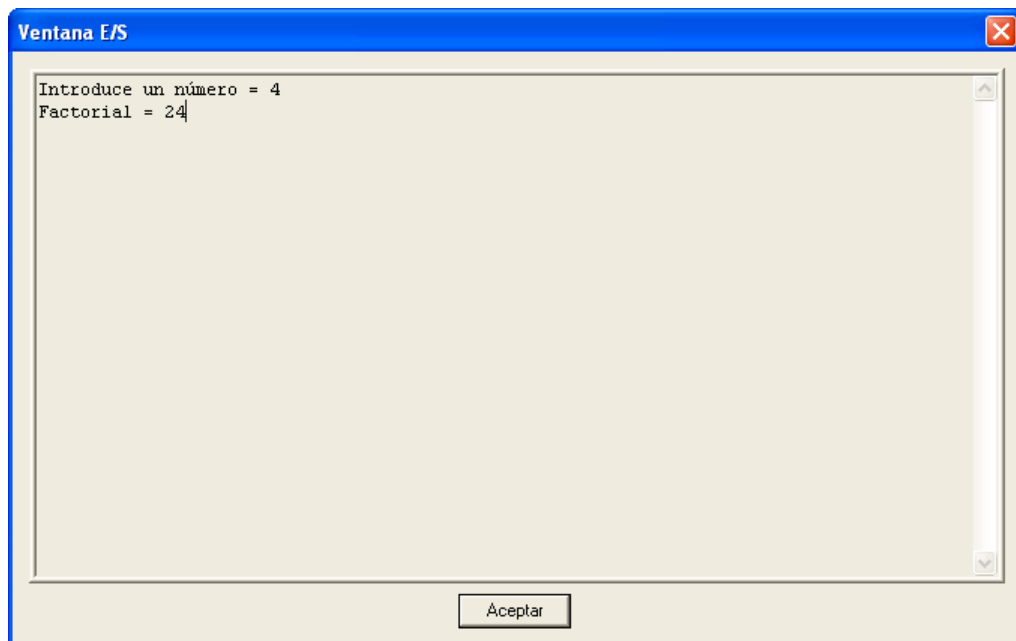


Figura 2.9: Ventana Entrada/Salida.

- cuando se ejecuta un trap en el que se envía datos formateados al dispositivo estándar de salida.

- cuando se ejecuta un trap en el se necesitan leer datos desde la entrada estándar.
- en cualquier momento a petición del usuario, cuando se ejecuta la función de menú *Ejecutar* ▷ *Ventana E/S*.

La ventana dispone de un botón de *Aceptar* que se debe pulsar para cerrar la ventana o validar los datos introducidos en ella durante el trap de lectura. Esta validación de datos también es posible pulsando la tecla ENTER.

La Figura 2.9 muestra el contenido de la ventana E/S después de la ejecución de un programa en el que se han enviado datos formateados, `printf()`, a la salida estándar, y se ha leído datos desde la entrada estándar (en concreto, un número sobre el que calcular su factorial).

Capítulo 3

Menús

3.1. Menú Ficheros

El menú *Ficheros* contiene todas las funciones necesarias para los programas a simular (cargar, recargar), para reinicializar el procesador DLXV y simulador, y para salir de la aplicación:

Cargar	Permite cargar un programa en el simulador.
Recargar	Permite recargar un programa en el simulador.
Reset DLXV	Permite reinicializar el procesador DLXV.
Reset Completo	Permite reinicializar el procesador DLXV sin cargar ningún programa en el simulador.
Salir	Permite salir de la aplicación.

La Figura 3.1 muestra las funciones disponibles en el menú *Ficheros*.



Figura 3.1: Menú *Ficheros*.

3.1.1. Cargar un programa en el simulador

La función *Ficheros*▷*Cargar* permite cargar y ensamblar un programa en el simulador.

Al ejecutarse la función, se muestra una caja de diálogo como la de la Figura 3.2 en la que se puede seleccionar el programa a cargar en el simulador. Este programa es un fichero con extensión *.s*, escrito en ensamblador DLX/DLXV. Esta caja de diálogo cuenta con los siguientes elementos:

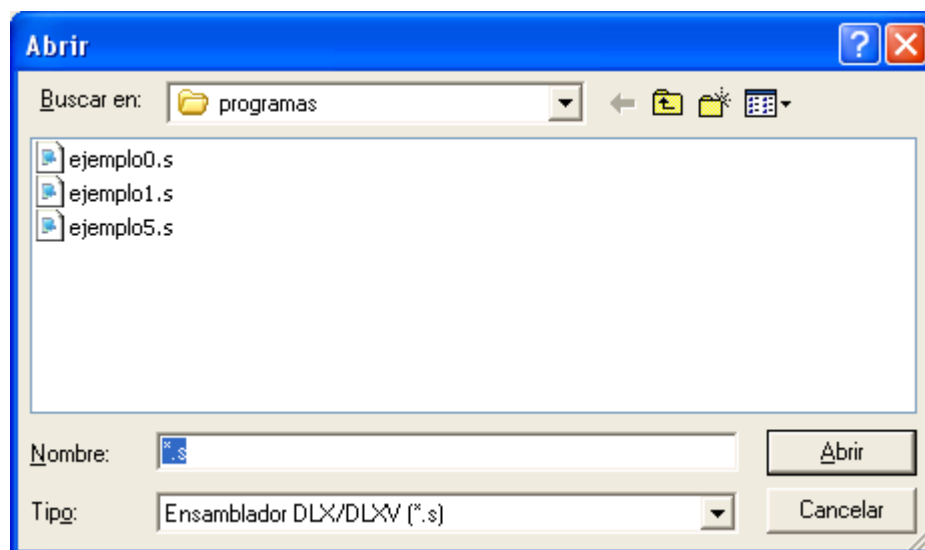


Figura 3.2: Caja de diálogo para cargar un programa en el simulador.

- La lista desplegable *Buscar en:* permite seleccionar la unidad de disco y la carpeta desde la que se desee cargar el archivo del programa a simular. Se encuentran disponibles tanto las unidades locales, como todas las accesibles a través de una red a que se tenga acceso. La carpeta por defecto donde comenzar la búsqueda de los ficheros con extensión *.s* será la carpeta de instalación de WinDLXV, a no ser que se haya seleccionado otra carpeta de trabajo (ver función de menú *Herramientas*▷*Carpeta de datos*).

- Iconos: el significado de cada uno de los iconos es el siguiente:



Ir a la última carpeta visitada.



Subir un nivel. Este icono permite seleccionar la carpeta de un nivel superior a la que se encuentra seleccionada.



Crear nueva carpeta. Permite crear una carpeta nueva dentro de la seleccionada. Una vez creada, es posible asignarle un nombre nuevo.



Menú Ver. Permite mostrar la lista de archivos en modo iconos grandes, iconos pequeños, lista, detalles o vistas en miniatura.

- La lista de archivos y carpetas: muestra todos los archivos de ensamblador DLX/DLXV y las carpetas existentes dentro de la carpeta seleccionada. En esta lista es posible realizar las operaciones de archivo habituales que permite el explorador de Windows (copiarlos, eliminarlos, cambiar sus nombres, etc.). Para ello, puede pulsarse sobre un archivo con el botón derecho del ratón, con lo que se muestra un menú contextual.
- La casilla Nombre: permite introducir el nombre del archivo a cargar. Su extensión es siempre .s.
- La lista desplegable Tipo: muestra el formato de archivo que puede cargarse en WinDLXV (*Ensamblador DLX/DLXV*).

Una vez introducido el nombre del archivo, se pulsa el botón *Abrir* y se procede a la carga y ensamblado del programa. Si se producen errores de ensamblado, estos se visualizan en una caja de diálogo alternativa y todos los datos escritos en memoria serán considerados inválidos.

Si por el contrario, la carga del programa se completó correctamente, veremos que las ventanas del simulador se actualizan como aparece en la Figura 3.3:

1. En la barra de título de la ventana principal podremos ver el nombre del programa cargado.
2. La ventana Código y Datos contienen el programa cargado en la memoria del procesador.
3. En la ventana Código, está seleccionada la primera instrucción a ejecutarse. Será de color amarillo, color correspondiente a la etapa IF (primera etapa a ejecutarse en el cauce del procesador DLXV).
4. En la ventana Cauce, aparece en la etapa IF, la primera instrucción del programa a ejecutar.

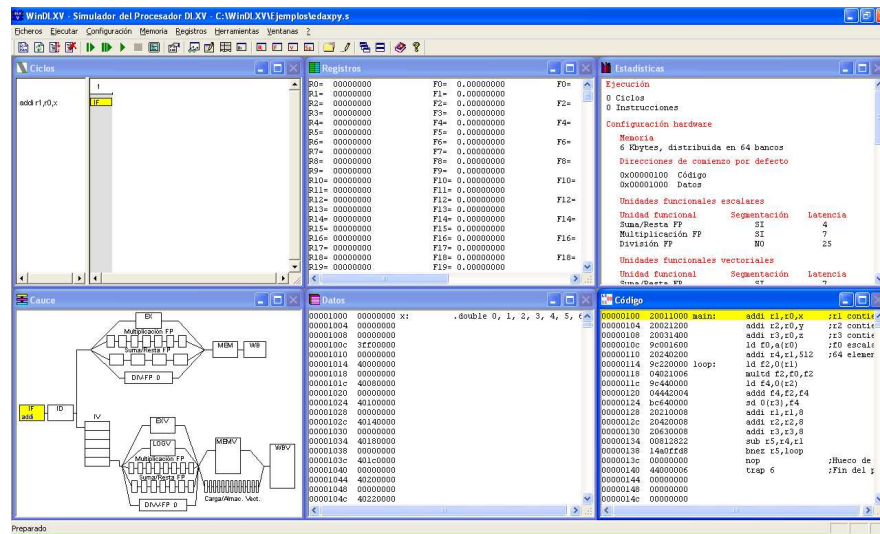


Figura 3.3: Carga de un programa en WinDLXV.

5. Igualmente, en la ventana Ciclos, aparece como el primer ciclo a ejecutar la etapa IF de la primera instrucción del programa cargado.
6. Como la simulación no ha comenzado, en la ventana Estadísticas sólo se habrá actualizado el grupo relativo al Estado de la memoria, con el tamaño y direcciones de comienzo del programa cargado.

La próxima vez que se arranque la aplicación WinDLXV, se cargará automáticamente este programa. La aplicación WinDLXV guarda cuál ha sido el último fichero cargado correctamente.

3.1.2. Recargar un programa en el simulador

Una vez cargado un fichero correctamente con la opción *Cargar*, puede ser recargado automáticamente con la función *Ficheros* ▸ *Recargar*. Esto permitirá estar recargando el fichero cómodamente en el simulador mientras se actualiza el programa con un editor externo.

3.1.3. Reinicializar el procesador DLXV

La función *Ficheros* ▸ *Reset DLXV* permite reinicializar el procesador y simulación:

- Se inicializa el cauce: las ventanas Ciclos, Cauce y Código muestran como siguiente instrucción a ejecutar la primera instrucción del programa cargado.
- Se inicializan los registros.
- Se inicializan las estadísticas.
- Se borra el contenido de la Ventana E/S.

3.1.4. Reinicializar el procesador DLXV sin cargar ningún programa

La función *Ficheros* ▸ *Reset completo* permite, además de reinicializar el procesador y simulación como la función *Reset DLXV*, eliminar de la memoria el programa cargado.

3.1.5. Salir de WinDLXV

La Función *Ficheros* ▸ *Salir* permite salir de la aplicación WinDLXV. Antes de abandonar la aplicación se guardará la configuración actual de la arquitectura del procesador. Esta configuración será usada como configuración por defecto la próxima vez que se arranque la aplicación WinDLXV.

3.2. Menú Ejecutar

El menú *Ejecutar* contiene todas las funciones que permiten ejecutar el código ensamblado en la memoria y visualizar la Ventana E/S:

Un ciclo	Ejecuta un único ciclo de reloj del procesador DLXV.
Múltiples ciclos	Ejecuta varios ciclos de reloj.
Sin pausas hasta	Ejecuta hasta un punto de ruptura definido previamente o hasta el final del programa.
Parar	Para la simulación.
Ventana E/S	Visualiza la ventana de Entrada/Salida.

La Figura 3.4 muestra las funciones disponibles en el menú *Ejecutar*.



Figura 3.4: Menú *Ejecutar*.

3.2.1. Ejecutar un único ciclo de reloj

La función *Ejecutar* ▸ *Un ciclo* permite ejecutar un ciclo de reloj del procesador. Después de ejecutar ese ciclo de reloj, se actualizarán los componentes alterados durante ese ciclo.

La simulación ciclo a ciclo permite analizar y comprender todos los pasos que conllevan la ejecución de una instrucción en el cauce del procesador DLXV.

3.2.2. Ejecutar varios ciclos de reloj

La función *Ejecutar* ▸ *Varios ciclos* permite ejecutar varios ciclos de reloj. El número de ciclos de reloj a ejecutar con esta función puede ser previamente fijado con la función de menú *Configuración* ▸ *Multi-Ciclos*.

3.2.3. Ejecutar sin pausas

El usuario puede definir un punto de ruptura en el programa (ver sección 5.5 de este manual) y luego seleccionar la función *Ejecutar* ▸ *Sin pausas hasta*. El programa se ejecutará sin pausas hasta este punto, salvo que haya un problema previo que ocasione la parada de la simulación. Si no se define ningún punto de ruptura, entonces se ejecutará sin pausas hasta el final del programa.

En la barra de estado, se podrá comprobar el número de ciclos de reloj ejecutados.

3.2.4. Parar la simulación

La función *Ejecutar* ▸ *Parar* permite parar la simulación.

3.2.5. Visualizar la ventana de Entrada/Salida

La función *Ejecutar* ▸ *Ventana E/S* permite visualizar la ventana de Entrada/Salida aunque la simulación no esté funcionando.

3.3. Menú Configuración

El menú *Configuración* contiene todas las funciones que permiten configurar la arquitectura DLXV a simular y gestionar los ficheros de configuración de la arquitectura simulada:

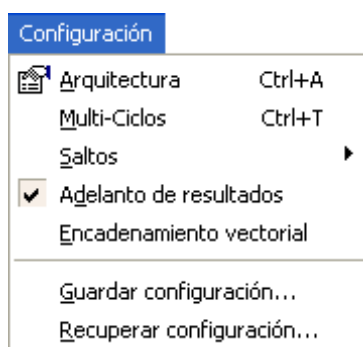


Figura 3.5: Menú *Configuración*.

Arquitectura	Permite modificar la arquitectura del procesador.
Multi-Ciclos	Permite definir el número de ciclos a ejecutar en la simulación de múltiples ciclos.
Saltos	Permite definir el comportamiento del procesador ante una instrucción de salto.
Adelanto de resultados	Permite activar el adelanto de resultados en el cauce.
Encadenamiento vectorial	Permite activar el encadenamiento vectorial.
Guardar configuración	Permite guardar la configuración actual de la arquitectura del procesador DLXV.

Recuperar configuración Permite recuperar una configuración de la arquitectura del procesador DLXV.

La Figura 3.5 muestra las funciones disponibles en el menú Configuración.

3.3.1. Modificar la arquitectura del procesador

La función *Configuración* ▸ *Arquitectura* permite modificar la arquitectura del procesador. Como se muestra en la Figura 3.6 los siguientes aspectos pueden ser configurados:

Modificar Arquitectura

Memoria

Tamaño de memoria (Kbytes)

Bancos de memoria

Direcciones de comienzo por defecto

Código Datos

Aceptar

Cancelar

Unidades funcionales escalares

	Segmentación	Latencia
Suma/Resta FP	<input checked="" type="checkbox"/>	<input type="text" value="3"/>
Multiplicación FP	<input checked="" type="checkbox"/>	<input type="text" value="6"/>
División FP	<input type="checkbox"/>	<input type="text" value="24"/>

Unidades funcionales vectoriales

	Segmentación	Latencia
Suma/Resta FP	<input checked="" type="checkbox"/>	<input type="text" value="6"/>
Multiplicación FP	<input checked="" type="checkbox"/>	<input type="text" value="7"/>
División FP	<input type="checkbox"/>	<input type="text" value="20"/>
Carga/Almac. Vectorial	<input checked="" type="checkbox"/>	<input type="text" value="12"/>

Cola de instrucciones vectoriales

Longitud

Registros vectoriales

	Lectura	Escritura
Número de puertos	<input type="text" value="2"/>	<input type="text" value="1"/>

¡Atención: Esta operación causará un reset!

Figura 3.6: Modificar la arquitectura del procesador simulado.

1. Memoria:

- Tamaño de la memoria (en Kbytes).
- Número de bancos de memoria. Su valor deberá ser potencia de 2.

- Direcciones de comienzo del código y los datos del programa ensamblado si no se especifica ninguna dirección en las directivas `.text` y `.data`, respectivamente.

2. Unidades funcionales escalares:

- Si las unidades de suma y multiplicación de punto flotante se encuentran segmentadas o no.
- El valor de la latencia de las unidades funcionales de suma, multiplicación y división de punto flotante.

3. Unidades funcionales vectoriales:

- Si las unidades de suma y multiplicación de punto flotante, y la unidad de carga/almacenamiento vectorial se encuentran segmentadas o no.
- El valor de la latencia de las unidades funcionales de suma, multiplicación y división de punto flotante, y de la unidad de carga/almacenamiento vectorial.

4. Cola de instrucciones vectoriales:

- Máximo número de instrucciones vectoriales encoladas en la etapa de identificación de instrucciones vectoriales IV.

5. Registros vectoriales:

- Número de puertos de lectura de un registro vectorial.
- Número de puertos de escritura de un registro vectorial.

La modificación de uno cualquiera de estos aspectos de la arquitectura provocará un reset del procesador.

3.3.2. Definir número de ciclos a ejecutar en la simulación de múltiples ciclos

La función *Configuración* ▸ *Multi-ciclos* permite definir el número de ciclos a ejecutar en la simulación de múltiples ciclos (ver función de menú *Ejecutar* ▸ *Múltiples Ciclos*).

3.3.3. Definir comportamiento ante un salto

La función *Configuración* ▸ *Salto* permite definir el comportamiento del procesador ante una instrucción de salto. Existen dos opciones:

1. Predicción de no tomar: supone que el salto no se va a tomar. Si el salto es efectivo, se detendrá el cauce y recomenzará la búsqueda de la nueva instrucción a ejecutar.
2. Salto retardado: ejecuta siempre la instrucción siguiente al salto, sea el salto efectivo o no efectivo. El número de huecos a ocupar es solamente uno, ya que la arquitectura simulada es capaz de determinar que la instrucción es un salto y la dirección del salto, en la etapa ID del cauce.

3.3.4. Activar adelanto de resultados

La función *Configuración* ▸ *Adelanto de resultados* permite activar o desactivar el adelanto de resultados (*forwarding*) en el cauce.

Si se activa el adelanto de resultados, en la ventana *Registros* podremos ver en qué etapa está disponible el contenido del registro, ya que éste se muestra en el color característico de esa etapa.

3.3.5. Activar encadenamiento vectorial

La función *Configuración* ▸ *Encadenamiento vectorial* permite activar o desactivar el encadenamiento (*chaining*) vectorial.

3.3.6. Guardar configuración

La función *Configuración* ▸ *Guardar configuración* permite guardar la configuración actual de la arquitectura del procesador DLXV simulado.

Al ejecutarse la función, se muestra una caja de diálogo como la de la Figura 3.7.

Esta caja de diálogo es análoga a la ya explicada para la función de menú *Ficheros* ▸ *Cargar* (ver sección 3.1.1). En este caso, el archivo a guardar será del tipo *Arquitectura DLXV*, con extensión *.cfg*. También en este caso, la carpeta por defecto para guardar el fichero de configuración será la carpeta de instalación de WinDLXV, a no ser que se haya seleccionado otra carpeta de trabajo.

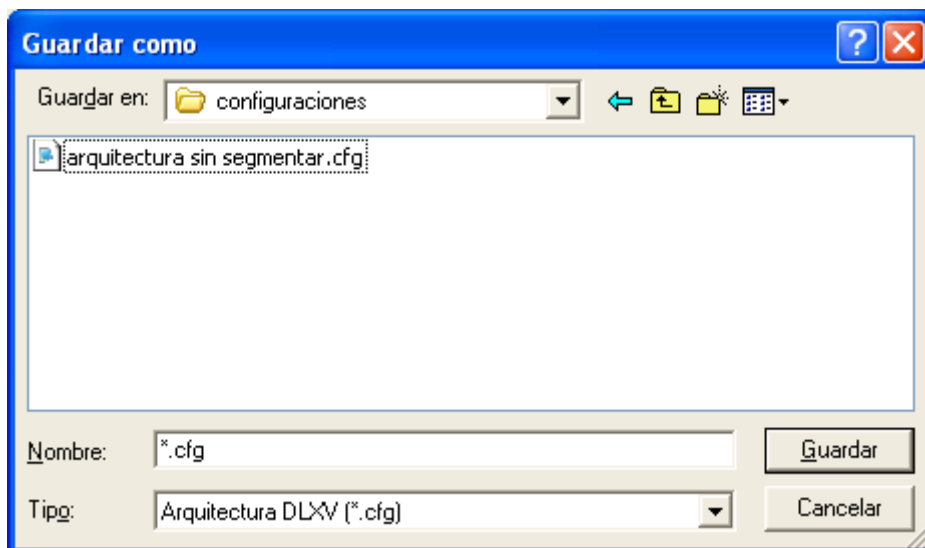


Figura 3.7: Caja de diálogo para guardar configuración del procesador DLXV simulado.

La configuración que se guarda corresponde a los valores seleccionados en las otras funciones del menú *Configuración: Arquitectura, Multi-Ciclos, Saltos, Adelanto de resultados, Encadenamiento vectorial*.

3.3.7. Recuperar configuración

La función *Configuración ▸ Recuperar configuración* permite recuperar una configuración de la arquitectura del procesador DLXV guardada anteriormente con la función anterior (*Guardar Configuración*).

Al ejecutarse esta función aparece una caja de diálogo donde seleccionar el fichero con extensión `.cfg` con la configuración a cargar. Al igual que el comando *Guardar Configuración*, la carpeta por defecto desde la que recuperar el fichero será la carpeta de instalación de WinDLXV, a no ser que se haya seleccionado otra carpeta de trabajo.

La carga de una nueva configuración supondrá un reset del procesador.

3.4. Menú Memoria

El menú *Memoria* contiene las funciones que permiten visualizar y modificar la memoria de código y datos, así como visualizar los símbolos definidos en el programa ensamblado en la memoria:

Visualizar memoria	Permite visualizar (y modificar) toda la memoria de datos o código.
Modificar memoria	Permite modificar una determinada dirección de memoria.
Símbolos	Permite visualizar los símbolos definidos en la memoria de datos y código.
Ensamblar	Permite ensamblar una instrucción.

La Figura 3.8 muestra las funciones disponibles en el menú Registros.

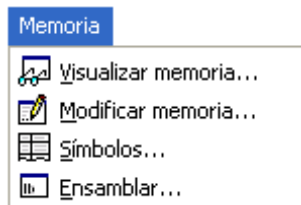


Figura 3.8: Menú *Memoria*.

3.4.1. Visualizar memoria

La función *Memoria* ▸ *Visualizar memoria* permite visualizar toda la memoria de datos o código en un formato dado. Con esta función también es posible modificar el contenido de una dirección en el formato que se está visualizando.

Al ejecutarse la función, se muestra una caja de diálogo como la de la Figura 3.9.

En esta caja de diálogo se visualizan el contenido de toda la memoria de datos o código. Existe unos menús desplegables en que se puede seleccionar el tipo de memoria y el formato:

- Tipo. El tipo de memoria a visualizar, datos o código.
- Formato. Si se ha seleccionado la memoria de datos, se puede seleccionar el formato en el que se desea visualizar la memoria: hexadecimal, decimal, ASCII, en punto flotante (simple y doble precisión). Si por el contrario, se ha seleccionado la memoria de código, este menú desplegable permanece desactivo (las instrucciones siempre se visualizan en formato hexadecimal).

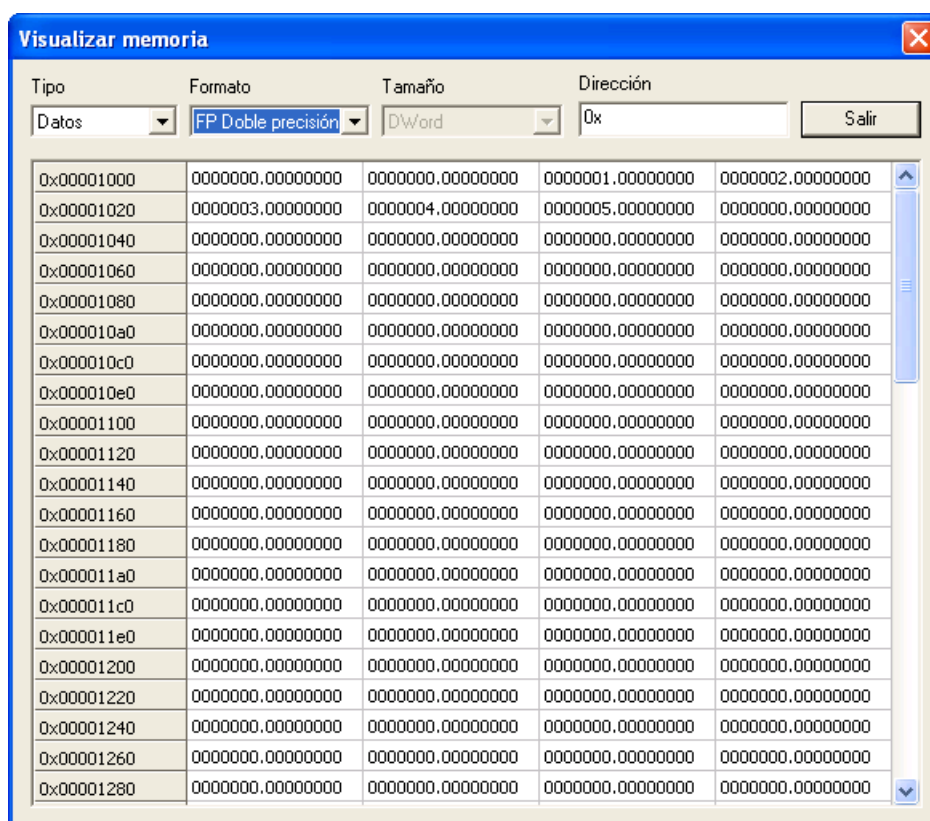


Figura 3.9: Visualizar la memoria.

- **Tamaño.** En el caso de que se seleccione el formato hexadecimal o decimal en el menú anterior, este menú desplegable se activará, y se podrá seleccionar el tamaño: Byte (8bits), HalfWord (16bits), Word (32 bits) o DWord (64 bits).

Con esta función de menú pueden visualizarse todas las posiciones de memoria: la caja de diálogo dispone de un *scroll* para ello, y además de un buscador de una determinada posición de memoria (campo *Dirección*).

Para modificar el contenido de una posición de memoria basta modificar la casilla correspondiente a su valor y luego pulsar la tecla ENTER.

3.4.2. Modificar memoria

La función *Memoria* ▸ *Modificar memoria* permite modificar una posición de memoria en un formato dado.

Al ejecutarse la función, se muestra una caja de diálogo como la de la Figura 3.10.

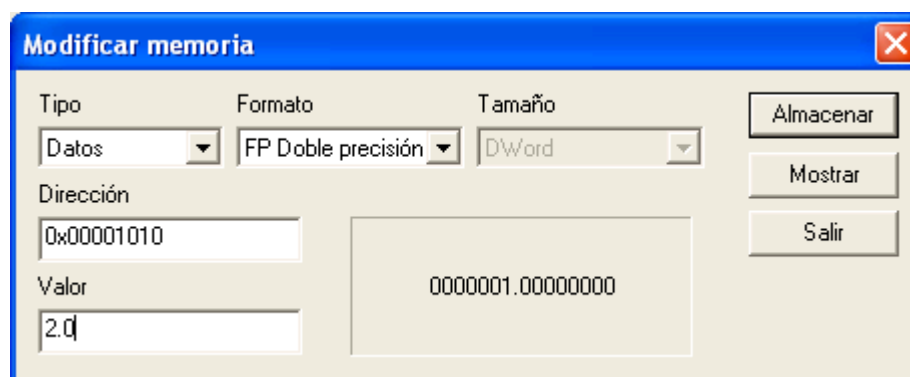


Figura 3.10: Modificar una posición de memoria.

En esta caja de diálogo existen unos menús desplegables en que se puede seleccionar el tipo de memoria (datos o código), el formato y tamaño:

- **Tipo.** El tipo de memoria a visualizar, datos o código.
- **Formato.** Si se ha seleccionado la memoria de datos, se puede seleccionar el formato en el que se desea visualizar la memoria: hexadecimal, decimal, ASCII, en punto flotante (simple y doble precisión). Si por el contrario, se ha seleccionado la memoria de código, este menú desplegable permanece desactivado (las instrucciones siempre se visualizan en formato hexadecimal).
- **Tamaño.** En el caso de que se seleccione el formato hexadecimal o decimal en el menú anterior, este menú desplegable se activará, y se podrá seleccionar el tamaño: Byte (8bits), HalfWord (16bits), Word (32 bits) o DWord (64 bits).

Una vez seleccionado el tipo de memoria, formato y tamaño de la memoria a modificar, se introduce la dirección de la posición de la memoria que va a modificarse (campo *Dirección*). Puede chequearse el valor actual de esa posición de memoria mediante el botón *Mostrar*. A continuación, se introduce el nuevo valor de esa posición de memoria (campo *Valor*), en el formato seleccionado, y se confirma su cambio con el botón *Almacenar*.

3.4.3. Visualizar símbolos

La función *Memoria* ▸ *Visualizar símbolos* permite visualizar los símbolos (etiquetas) definidos en la memoria de datos y código.

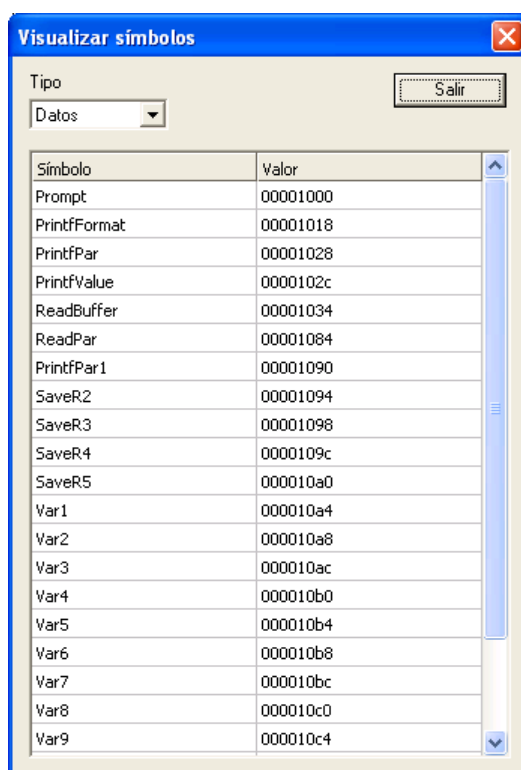


Figura 3.11: Visualizar símbolos.

Al ejecutarse la función, se muestra una caja de diálogo como la de la Figura 3.11.

En esta caja de diálogo existen un menú desplegable en que se puede seleccionar el tipo de símbolos a visualizar, de datos o código.

3.4.4. Ensamblar

La función *Memoria* ▸ *Ensamblar* permite ensamblar una instrucción.

Al ejecutarse la función, se muestra una caja de diálogo como la de la Figura 3.12.

Con esta función de menú se permite ensamblar una instrucción y actualizar una dirección de código con este nuevo valor. Si no se especifica

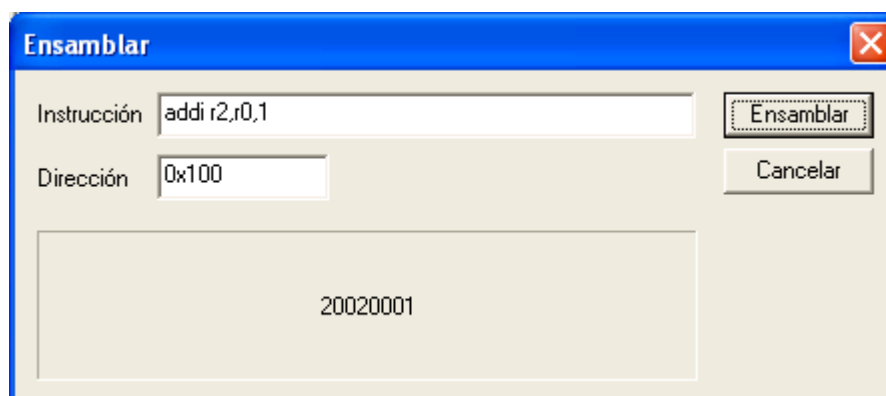


Figura 3.12: Ensamblar.

ninguna dirección, la instrucción sólo es ensamblada, y mostrada en esta caja de diálogo.

3.5. Menú Registros

El menú *Registros* contiene las funciones que permiten visualizar y modificar los registros del procesador:

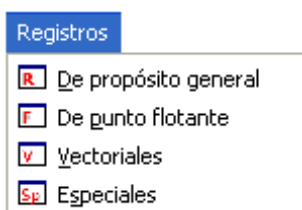


Figura 3.13: Menú *Registros*.

De propósito general	Permite visualizar y modificar los registros escalares de propósito general GPR.
De punto flotante	Permite visualizar y modificar los registros escalares de punto flotante FPR.
Vectoriales	Permite visualizar y modificar los registros vectoriales.

Especiales Permite visualizar los registros especiales.

La Figura 3.13 muestra las funciones disponibles en el menú Registros.

3.5.1. Registros escalares de propósito general

La función *Registros* ▸ *De propósito general* permite visualizar y modificar los registros escalares de propósito general.

El procesador DLXV dispone de 32 registros de propósito general GPR (R0, R1, ..., R31) de 32 bits. El valor de R0 es siempre 0, y no es posible modificar su valor con esta función.

Al ejecutarse la función, se muestra una caja de diálogo como la de la Figura 3.14.

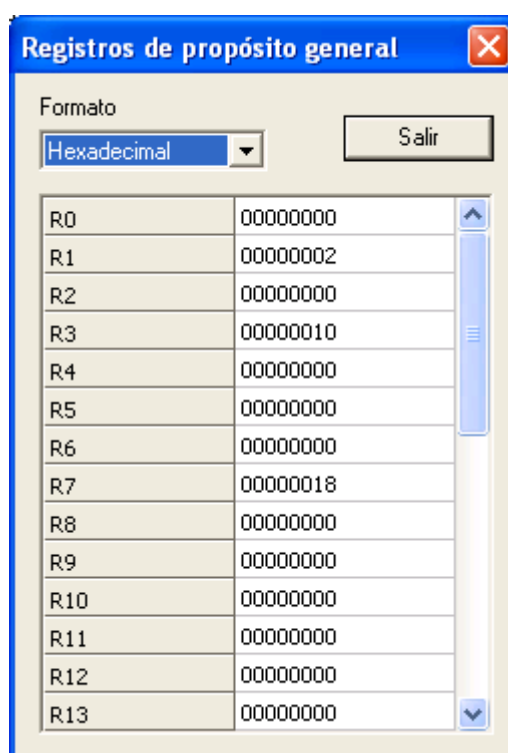


Figura 3.14: Registros de propósito general.

En esta caja de diálogo se visualizan todos los registros de propósito general. Existe un menú desplegable en que se puede seleccionar el formato en el que se desea visualizarlos: hexadecimal o decimal.

Para modificar el contenido de un registro basta modificar la casilla correspondiente a su valor y luego pulsar la tecla ENTER.

3.5.2. Registros escalares de punto flotante

La función *Registros* ▸ *De punto flotante* permite visualizar y modificar los registros escalares de punto flotante.

El procesador DLXV dispone de registros de punto flotante (FPR), que se pueden utilizar como 32 registros (F0, F1, ..., F31) de simple precisión (32 bits), o como pareja par-impar que contienen valores de doble precisión (F0, F2, ..., F30).

Al ejecutarse la función, se muestra una caja de diálogo como la de la Figura 3.15.

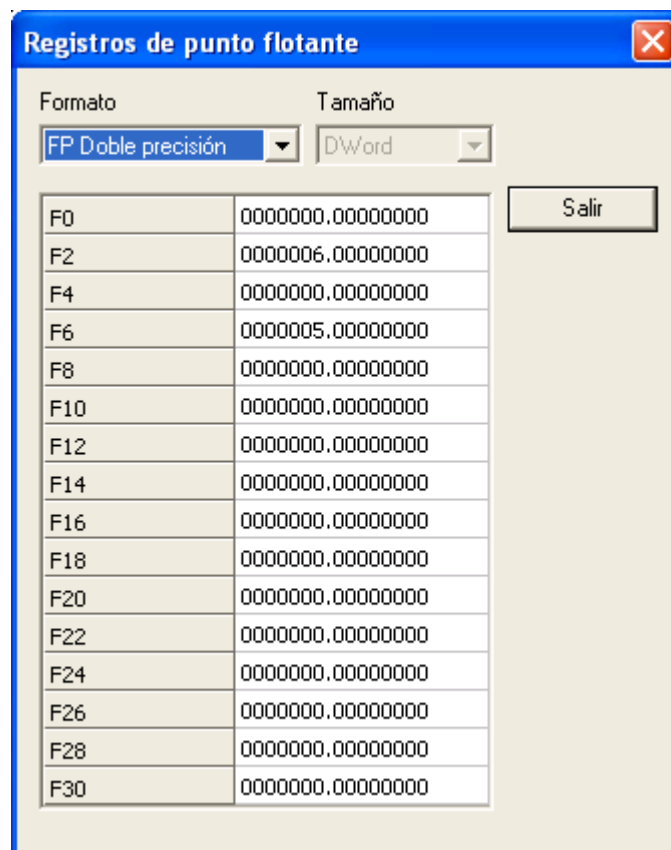


Figura 3.15: Registros de punto flotante.

La caja de diálogo dispone de dos menús despegables:

- **Formato.** Los registros de punto flotante se pueden visualizar en tres formatos: punto flotante simple precisión, punto flotante doble precisión y hexadecimal.
- **Tamaño.** En el caso de que se seleccione el formato hexadecimal en el menú anterior, este menú desplegable se activará, y se podrá seleccionar 2 tamaños: Word (32 bits) o DWord (64 bits).

Una vez elegido el formato y tamaño (si es aplicable) se visualizarán todos los registros de punto flotante en el formato seleccionado. Para modificar el contenido de un registro basta modificar la casilla correspondiente a su valor y luego pulsar ENTER.

3.5.3. Registros vectoriales

La función *Registros* ▸ *Vectoriales* permite visualizar y modificar los registros vectoriales.

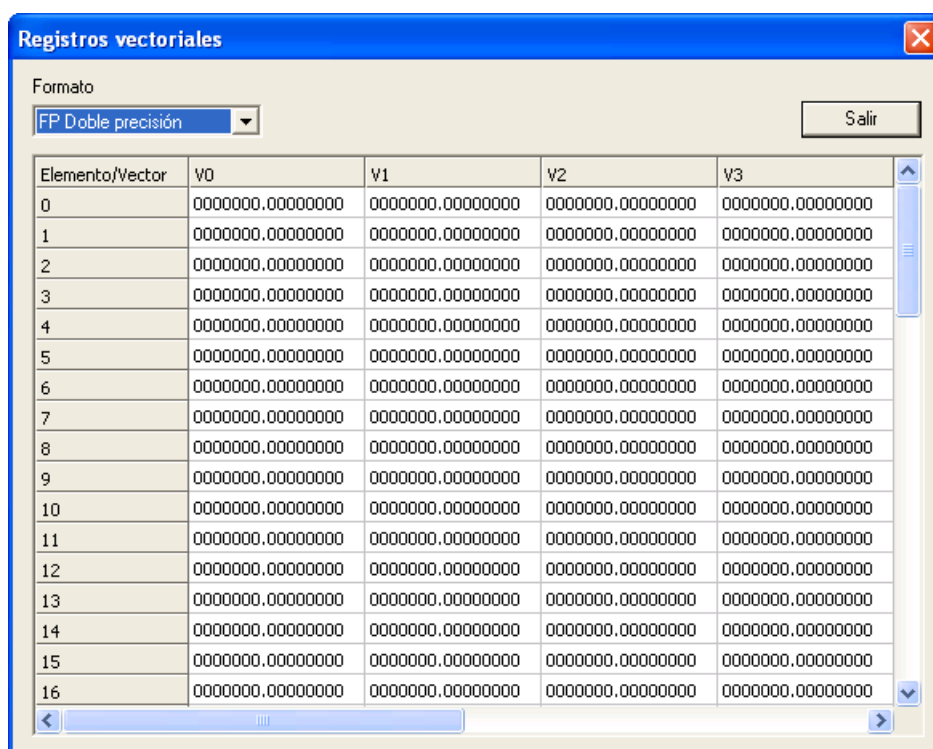


Figura 3.16: Registros vectoriales.

El procesador DLXV dispone de 8 registros vectoriales (V0, V1, ..., V7), donde cada registro contiene 64 dobles palabras (elementos).

Al ejecutarse la función, se muestra una caja de diálogo como la de la Figura 3.16.

En esta caja de diálogo se visualizan todos los registros vectoriales. Existe un menú desplegable en que se puede seleccionar el formato en el que se desea visualizarlos: punto flotante doble precisión o hexadecimal.

Para modificar el contenido de un elemento de un registro basta modificar la casilla correspondiente a su valor y luego pulsar ENTER.

3.5.4. Registros especiales

La función *Registros* ▸ *Especiales* permite visualizar los registros especiales. Estos registros no son modificables a través de esta función de menú.

El procesador DLXV dispone de los siguientes registros especiales:

1. FPSR (*Floating-Point Status Register*): registro de estado de 1 bit de longitud utilizado para comparaciones y excepciones de punto flotante.
2. VLR (*Vector-Length Register*): registro de longitud vectorial utilizado para controlar la longitud de cualquier operación vectorial.
3. VM (*Vector-Mask Register*): registro de máscara que tiene 64 bits, es decir, un bit por cada uno de los elementos de los registros vectoriales, utilizado para enmascarar en las operaciones vectoriales todos aquellos elementos cuyo bit asociado en este registro sean igual a cero.

Al ejecutarse la función, se muestra una caja de diálogo como la de la Figura 3.17.

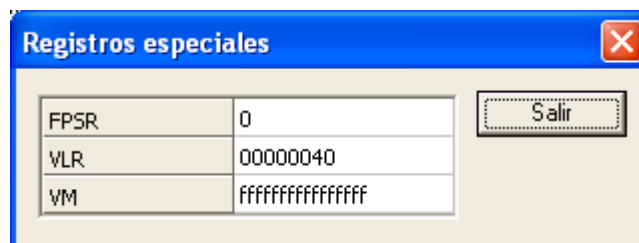


Figura 3.17: Registros especiales.

Todos los registros son visualizados en formato hexadecimal.

3.6. Menú Herramientas

El menú *Herramientas* contiene las funciones que permiten definir la carpeta de trabajo, acceder a un editor desde el que se pueda crear o modificar el programa a simular, y automáticamente cargarlo en el simulador, o activar u ocultar la barra de iconos y de estado:

Barra de iconos Muestra u oculta la barra de iconos.

Barra de estado Muestra u oculta la barra de estado.

Carpeta de datos Permite definir la carpeta de datos.

Editar Permite acceder a un editor desde el que se pueda crear o modificar el programa a simular.

La Figura 3.18 muestra las funciones disponibles en el menú *Herramientas*.

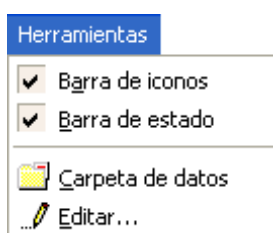


Figura 3.18: Menú *Herramientas*.

3.6.1. Carpeta de datos

La función *Herramientas* > *Carpeta de Datos* permite definir la carpeta de datos. Esta carpeta de datos será:

- La carpeta donde la aplicación WinDLXV recupere/guarde las características de la arquitectura por defecto (al arrancar la aplicación).
- La carpeta donde la aplicación WinDLXV recupere/guarde el programa a ensamblar y cargar en memoria por defecto (al arrancar la aplicación).
- La carpeta por defecto para buscar el fichero a cargar cuando se seleccione la función de menú *Ficheros* > *Cargar*.

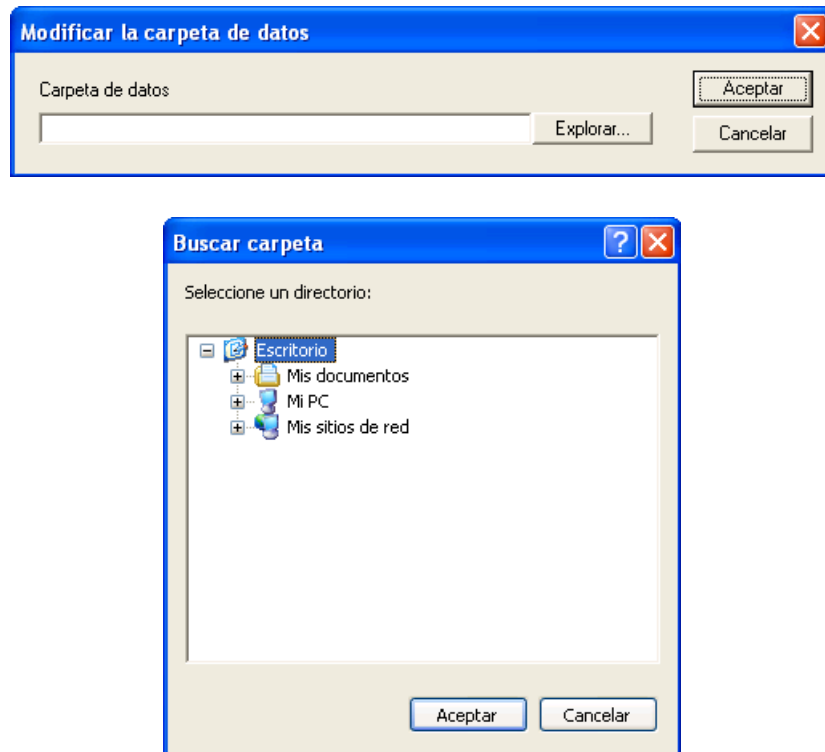


Figura 3.19: Caja de diálogo para modificar la carpeta de datos.

- La carpeta por defecto para guardar la configuración actual de la arquitectura en un fichero (función de menú *Configuración* ▸ *Guardar configuración*).
- La carpeta por defecto desde la que recuperar una configuración de arquitectura (función de menú *Configuración* ▸ *Recuperar configuración*).

Al ejecutarse la función, se muestra una caja de diálogo como la de la Figura 3.19 en la que al pulsar el botón *Explorar* se abre otra caja de diálogo en la que se puede seleccionar una carpeta. Esta será a partir de ahora la carpeta de trabajo.

3.6.2. Editor

La función *Ficheros* ▸ *Editar* permite acceder a un editor desde el que se pueda crear o modificar el programa a simular.

Al ejecutarse la función, se muestra una caja de diálogo como la de la Figura 3.20. Esta caja de diálogo cuenta con los siguientes elementos:

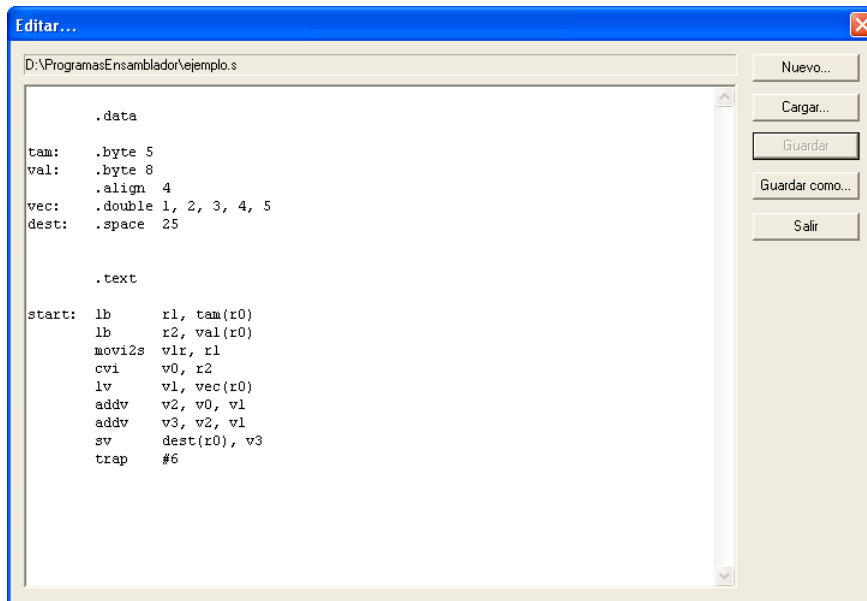


Figura 3.20: Editor.

- Nombre del fichero abierto: en la parte superior de la caja de diálogo.
- Botones: en la parte derecha de la caja de diálogo aparecen los botones característicos de Windows:
 1. Nuevo: permite crear un nuevo fichero. Al pulsar este botón se limpia el área de edición.
 2. Cargar: permite cargar un fichero de tipo ensamblador DLX / DLXV ya existente.
 3. Guardar: permite guardar el archivo que se encuentre seleccionado. El archivo permanece abierto, lo que permite continuar trabajando en él.
 4. Guardar como: permite guardar el archivo que se encuentre seleccionado con otro nombre. Automáticamente, se cierra el archivo que se encontraba seleccionado, se crea una copia de él con el nuevo nombre y se abre dicha copia.

5. Salir: permite salir del editor y volver al simulador. Si los cambios no han sido guardados, aparece una caja de diálogo en la que se pregunta si desea guardar los cambios. Si los guarda, o ya fueron guardados anteriormente, automáticamente se ensambla el nuevo fichero y se carga en la memoria del simulador.

- Área de edición: el resto de la caja de diálogo.

3.7. Menú Ventanas

El menú *Ventanas* contiene todas las funciones habituales de Windows que permiten organizar las distintas ventanas de trabajo existentes en la aplicación WinDLXV:

Cascada	Permite organizar todas las ventanas de trabajo de modo que se coloquen unas detrás de otras, pero visualizándose todas sus barras de título.
Mosaico	Permite organizar todas las ventanas de trabajo de modo que se visualicen completamente todas al máximo tamaño posible. El mosaico es de tipo horizontal.
Organizar iconos	Permite alinear en la parte inferior izquierda de la pantalla las ventanas iconificadas.
Código	Permite activar la ventana <i>Código</i> .
Estadísticas	Permite activar la ventana <i>Estadísticas</i> .
Datos	Permite activar la ventana <i>Datos</i> .
Registros	Permite activar la ventana <i>Registros</i> .
Cauce	Permite activar la ventana <i>Cauce</i> .
Ciclos	Permite activar la ventana <i>Ciclos</i> .

La Figura 3.21 muestra las funciones disponibles en el menú *Ventanas*.

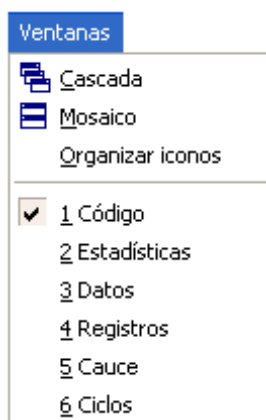


Figura 3.21: Menú *Ventanas*.

3.8. Menú ?

El menú *Ayuda* contiene una completa ayuda del simulador WinDLXV, así como el mensaje de *copyright* del programa:

Ayuda de WinDLXV Muestra este manual de usuario.

Acerca de WinDLXV Muestra información del programa: versión y propietario.

La Figura 3.22 muestra las funciones disponibles en el menú *Ayuda*.

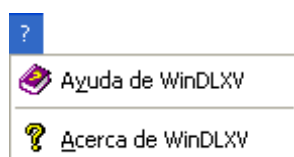























Figura 3.22: Menú ?.

3.8.1. Ayuda de WinDLXV

Esta función muestra el manual on-line del programa WinDLXV. Este manual se encuentra en formato PDF, por lo que se necesita tener instalado la aplicación Adobe Acrobat Reader. Esta aplicación se encuentra en la carpeta Adobe Acrobat del CD de instalación del programa WinDLXV.

3.9. Acceso rápido a las funciones de menú

Las funciones de uso más frecuente disponen de iconos en la barra de iconos y/o tiene definidas teclas aceleradoras. La Tabla 3.2 muestra los iconos y teclas aceleradoras definidas:

Función	Icono	Tecla aceleradora
Ficheros▷ Cargar		Ctrl+O
Ficheros▷ Recarga		F10
Ficheros▷ Reset DLXV		Ctrl+R
Ficheros▷ Reset completo		Ctrl+F
Ficheros▷ Salir	no tiene	no tiene
Ejecutar▷ Un ciclo		F7
Ejecutar▷ Múltiples ciclos		F8
Ejecutar▷ Sin pausas hasta		F4
Ejecutar▷ Parar		F5
Ejecutar▷ Ventana E/S		no tiene
Configuración▷ Arquitectura		Ctrl+A
Configuración▷ Multi-Ciclos	no tiene	Ctrl+T
Configuración▷ Saltos	no tiene	no tiene
Configuración▷ Adelanto de resultados	no tiene	no tiene
Configuración▷ Encadenamiento vectorial	no tiene	no tiene
Configuración▷ Guardar configuración	no tiene	no tiene
Configuración▷ Recuperar configuración	no tiene	no tiene
Memoria▷ Visualizar memoria		no tiene
Memoria▷ Modificar memoria		no tiene
Memoria▷ Símbolos		no tiene
Memoria▷ Ensamblar		no tiene
Registros▷ De propósito general		no tiene
Registros▷ De punto flotante		no tiene
Registros▷ Vectoriales		no tiene
Registros▷ Especiales		no tiene
Herramientas▷ Barra de iconos	no tiene	no tiene
Herramientas▷ Barra de estado	no tiene	no tiene
Herramientas▷ Carpeta de datos		no tiene
Herramientas▷ Editar		no tiene
Ventanas▷ Cascada		no tiene




Función	Icono	Tecla aceleradora
Ventanas▷ Mosaico		no tiene
Ventanas▷ Organizar iconos	no tiene	no tiene
?▷ Ayuda de WinDLXV		no tiene
?▷ Acerca de WinDLXV		no tiene

Tabla 3.2: Acceso rápido a las funciones más frecuentes.

Capítulo 4

Escritura de un programa

Los programas pueden escribirse directamente con el editor incluido en el simulador, función de menú *Herramientas* ▷ *Editar*, o bien utilizar cualquier otro editor de texto plano, y luego cargarlo al simulador con la función de menú *Ficheros* ▷ *Cargar*.

En ambos casos los programas se guardarán en archivos con extensión `.s`, y serán códigos en lenguaje ensamblador DLX/DLXV.

La sintaxis básica utilizada por WinDLXV tiene las siguientes características:

- Los comentarios empiezan por el símbolo “`;`”, todo lo que aparezca en la misma línea a continuación de este símbolo será ignorado.
- Los programas se dividen en dos partes:
 1. `.text`: sección obligatoria en todos los programas, contiene el conjunto de las instrucciones del programa.
 2. `.data`: sección opcional, aunque normalmente necesaria. Es la sección de declaración de las variables del programa.

Características de `.text`

La sección `.text` es obligatoria en todos los programas, contiene el conjunto de instrucciones del programa. Los elementos siguientes se guardan en el segmento de texto.

Tiene las siguientes características:

- Si no se proporciona dirección con la directiva `.text`, esta sección será cargada en la dirección por defecto definida en WinDLXV, la cual podrá ser modificada por el usuario a través de la función de menú *Configuración* ▷ *Arquitectura*.

- La primera instrucción de la sección `.text` será considerada por el simulador como la primera instrucción del programa a ejecutar.
- Las etiquetas van seguidas por dos puntos (:). Una etiqueta válida es una secuencia de caracteres alfanuméricos.
- En la línea de etiqueta puede haber una instrucción o no. Pero si no hay una instrucción en la ventana *Código* no aparecerá el nombre de la etiqueta al lado de la instrucción, y sólo se podrá ver con la función de menú *Memoria* > *Símbolos*.
- Las instrucciones válidas implementadas en este simulador se muestran en el capítulo 7.
- Todas las instrucciones son de 32 bits y deberán estar alineadas a este valor.
- Por defecto los números se representan en base 10.
- Se pueden incluir llamadas al sistema operativo (*traps*), para solicitar algún servicio, como abrir un fichero, cerrarlo, imprimir por pantalla, etc. Ver capítulo 8.

Características de `.data`

La sección `.data` contiene la declaración de las variables del programa. Los elementos siguientes se guardan en el segmento de datos. Esta sección es opcional, aunque normalmente necesaria.

Si no se proporciona dirección con la directiva `.data`, esta sección será cargada en la dirección por defecto definida en WinDLXV, la cual podrá ser modificada por el usuario a través de la función de menú *Configuración* > *Arquitectura*.

La declaración de variables del programa se ajusta a las siguientes reglas:

- La declaración de una variable sigue este formato:
 1. En primer lugar debe ir un identificador válido (etiqueta). Se considerará válida cualquier secuencia de caracteres alfanuméricos.
 2. A continuación se indica el tipo de variable. En la sección 6.1 se muestra la relación de tipos implementados en WinDLXV, sus características y la estructura de la definición.

3. Finalmente se inicializa la variable.

- Las cadenas de caracteres se encierran entre comillas dobles.
- Los números se consideran en base 10 por defecto. Si van precedidos del prefijo 0x se interpretan en hexadecimal.
- Todos los accesos a memoria deberán estar alineados.

Capítulo 5

Simulación paso a paso

Una vez conocido el entorno de simulación de WinDLXV, sus distintas ventanas y las funciones de menú disponibles, veamos a continuación paso a paso una sesión de simulación ilustrándolo con un ejemplo.

5.1. Escritura de un programa

Para escribir el programa tenemos dos opciones:

1. Utilizar el editor integrado en el simulador WinDLXV.
2. Utilizar un editor ASCII externo, y luego cargar el programa en el simulador.

Usaremos la primera opción, el editor propio de WinDLXV, aunque la otra opción será muy similar, bastará crear un fichero con extensión `.s` con el código ensamblador.

Para utilizar el editor de WinDLXV, el primer paso será abrir la aplicación WinDLXV (Inicio▷ Programas▷ WinDLXV). Una vez estamos dentro del simulador WinDLXV, podemos ya escribir nuestro programa. Para ello abriremos el editor, función de menú Ficheros▷ Editar.

Supongamos que se desea escribir un programa en lenguaje ensamblador DLXV, que sume dos números A y B guardados en memoria, y que el resultado de dicha suma C se guarde también en memoria.

```
;*****  
;*** Suma de dos números C=A+B ***
```

```
; *****  
      .data  
A:      .word 2  
B:      .word 8  
C:      .word 0  
  
      .text  
main:   lw r1,A  
        lw r2,B  
        add r3,r2,r1  
        sw C,r3  
        trap 6
```

Escribimos este código en el área de edición como muestra la Figura 5.1.

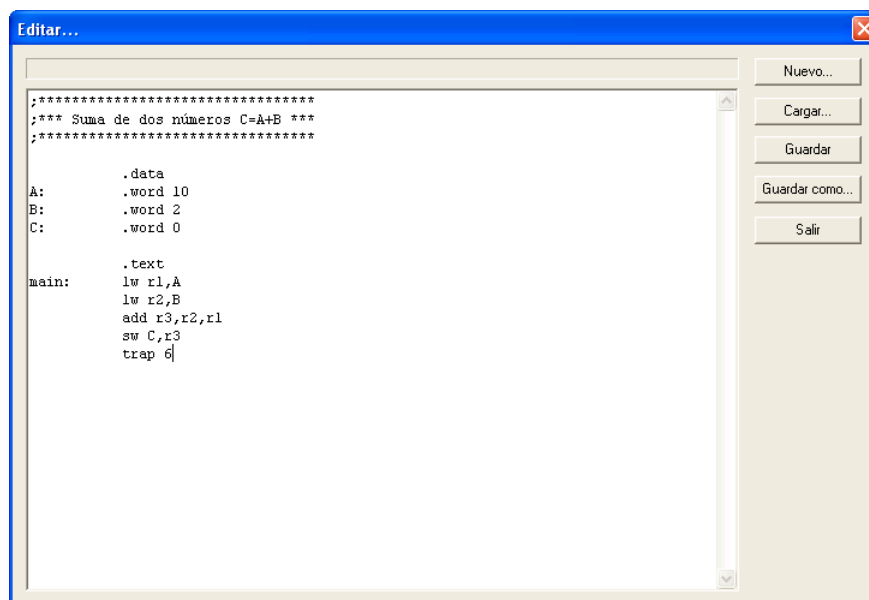


Figura 5.1: Simulando paso a paso: edición.

A continuación lo salvaremos el fichero como `suma.s`, para ello pulsamos el botón *Guardar como*.

5.2. Carga de un programa

Una vez editado el programa a simular, lo ensamblaremos y cargaremos en la memoria del simulador. Para ello, pulsamos el botón *Salir*, y

automáticamente se cargará en la memoria como muestra la Figura 5.2. Si se usó el editor externo para la creación del fichero, entonces, el código se cargará en la memoria del simulador con la función de menú *Ficheros* ▶ *Cargar*.

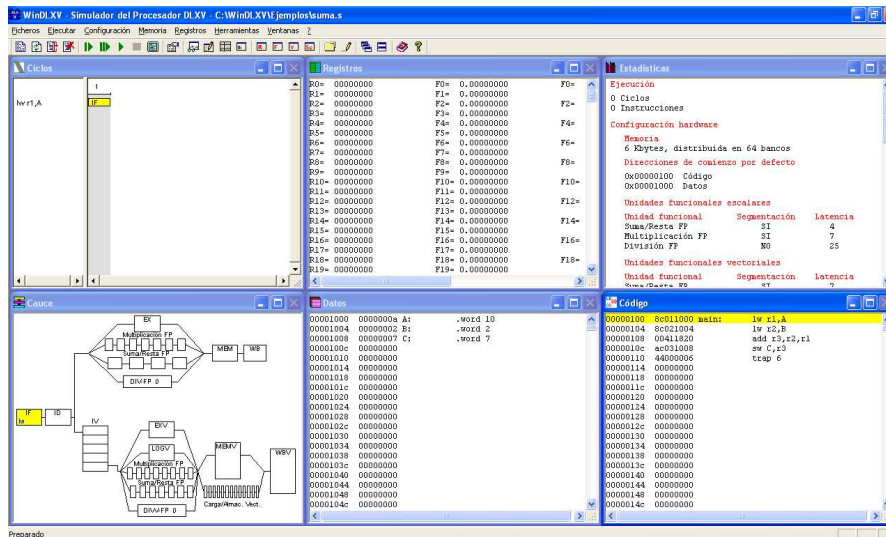


Figura 5.2: Simulando paso a paso: carga de un programa en WinDLXV.

En ambos casos podemos observar:

1. En la barra de título de la ventana principal podremos ver el nombre del programa cargado `suma.s`.
2. La ventana Código y Datos contienen el programa cargado en la memoria del procesador.
3. En la ventana Código, está seleccionada la primera instrucción a ejecutarse (`lw r1, A`). Será de color amarillo, color correspondiente a la etapa IF.
4. En la ventana Cauce, aparece en la etapa IF, esta primera instrucción (`lw`).
5. En la ventana Ciclos, aparece como el primer ciclo a ejecutar la etapa IF de la instrucción `lw r1, A`.
6. En la ventana Estadísticas podemos ver que el programa cargado tiene un tamaño de 20 bytes de código y 12 bytes de datos.

5.3. Definiendo la arquitectura del procesador

En este punto, definiríamos la arquitectura del procesador simulado con la función de menú *Configuración* ▷ *Arquitectura*, u otra característica del procesador como, por ejemplo, si hay adelanto de resultados en el cauce (*Configuración* ▷ *Adelanto de resultados*), encadenamiento vectorial (*Configuración* ▷ *Encadenamiento vectorial*), o el comportamiento ante un salto (*Configuración* ▷ *Saltos*).

En nuestro ejemplo sólo activaremos el adelanto de resultados en el cauce. Lo haremos a través de la función de menú *Configuración* ▷ *Adelanto de resultados*.

5.4. Ejecución ciclo a ciclo

El mandato de ejecución ciclo a ciclo (función de menú *Ejecutar* ▷ *Un ciclo*) permite ejecutar un solo ciclo de reloj de tal forma que se pueden apreciar los datos sobre los que se opera antes y después de su ejecución.

Ciclo 1 Presionando *Ejecutar* ▷ *Un ciclo* (o simplemente **F7**) avanza la simulación un ciclo.

En la ventana Código, el color de la primera instrucción ha cambiado a azul y la segunda instrucción se muestra en amarillo. Estos colores indican la siguiente etapa a ejecutarse. Azul para ID y amarillo para IF.

En la ventana Cauce, aparece en la etapa IF la segunda instrucción `lw r2, B`, mientras que la primera instrucción `lw r1, A` ha avanzado a la etapa ID.

En la ventana Ciclos podemos ver que se ha ejecutado la etapa IF de la primera instrucción, y las siguientes etapas son las etapas ID para la primera instrucción y la etapa IF para la segunda instrucción.

Ciclo 2 Presionando **F7** de nuevo, se vuelven a actualizar los colores en la ventana de código, introduciendo rojo para la etapa EX. La instrucción `add r3, r2, r1` entra en el cauce, en el próximo ciclo será tratada en la etapa IF.

Ciclo 3 Presionando **F7** de nuevo, se vuelven a actualizar los colores en la ventana de código, introduciendo verde para la etapa MEM. La instrucción `sw C, r3` entra en el cauce, en el próximo ciclo será tratada en la etapa

IF. En la ventana Registros aparece R1 en color gris indicando que su valor será actualizado por una instrucción que está ejecutándose (`lw r1, A`).

Ciclo 4 Presionando F7 de nuevo, cada etapa del cauce escalar está ocupada con una instrucción. Como está activado el adelanto de resultados, el valor de R1 está ya disponible desde la etapa MEM, se muestra en la ventana Registros con el color característico de esta etapa.

Ciclo 5 Presionando F7 de nuevo, el valor de R2 está disponible desde la etapa MEM al estar activado el adelanto de resultados, pero no está disponible a tiempo para la instrucción `add r3, r2, r1`, y se produce un parón del tipo RAW (lectura después de escritura). En la barra de estado se indica que se ha producido este parón, y en la ventana Ciclos se muestra en color azul la instrucción `add r3, r2, r1` por ser la que ha provocado el parón, y en color gris las instrucciones `sw C, r3` y `trap 6` al estar detenidas por la primera.

Ciclo 6 Presionando F7 de nuevo, la instrucción `add r3, r2, r1` pasa la etapa de ejecución EX, y su resultado, guardado en R3 está disponible desde la etapa EX por estar activado el adelanto de resultados. En la ventana Registros se muestra en color rojo, color característico de la etapa EX.

Ciclo 7 Presionando F7 de nuevo, el valor de R3 en la ventana registros cambia al color característico de la etapa MEM.

Ciclo 8 Presionando F7 de nuevo, el valor de R3 es guardado en la posición de memoria C.

Ciclo 9 Presionando F7 de nuevo, en este ciclo ya sólo se encuentra activa la instrucción `trap`.

Ciclo 10 Presionando F7 de nuevo, la simulación ha finalizado.

La Figura 5.3 muestra la ventana Ciclos después de haber ejecutado este programa ciclo a ciclo.

Fijémonos ahora en la ventana Estadísticas, veríamos que se han contabilizado 2 cargas y 1 almacenamiento escalar, que se ha producido 1 parón

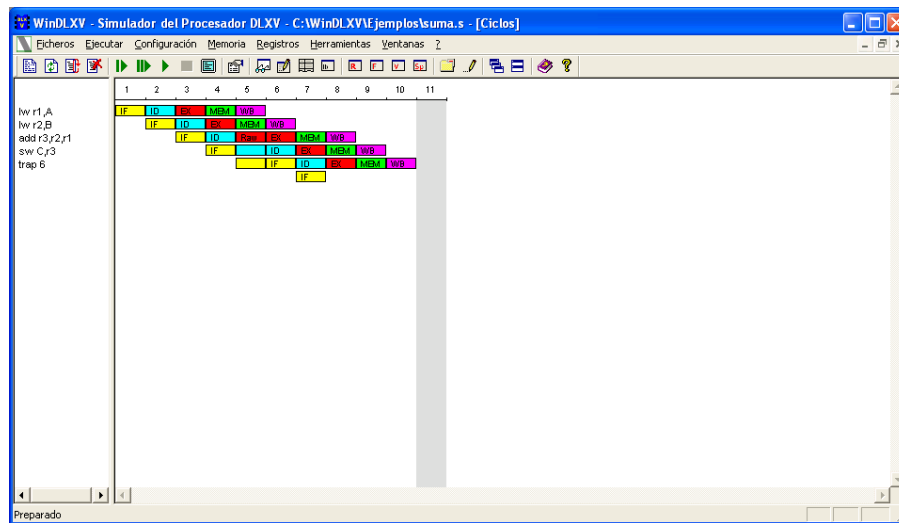


Figura 5.3: Simulando paso a paso: ejecución de un programa ciclo a ciclo.

de tipo RAW en el cauce. También veríamos que se han necesitado 10 ciclos de reloj para ejecutar 5 instrucciones, resultando 2 ciclos de reloj por instrucción (CPI = 2).

La ventana Estadísticas es útil para comparar el efecto que produce los cambios en la configuración de la arquitectura. En este ejemplo, ¿qué pasaría si desactivásemos el adelanto de los resultados en el cauce?, ¿cuáles serían los resultados?

5.5. Definiendo puntos de ruptura

La simulación ciclo a ciclo permite analizar exhaustivamente qué ocurre en cada ciclo de reloj, pero el principal problema es que la ejecución de un programa puede ser muy larga y esta forma de “ejecución controlada” puede llegar a ser muy lenta. El caso más habitual es que se quiera analizar una zona de código de un programa. En esta situación es conveniente definir un punto de ruptura al inicio del código que se va a analizar, y a continuación ejecutar el código sin pausas hasta ese punto de ruptura, y luego continuar la ejecución ciclo a ciclo.

En el ejemplo que hemos estado analizado ciclo a ciclo supongamos que ponemos un punto de ruptura en la instrucción `add r3, r2, r1`, y a continuación seguimos la ejecución ciclo a ciclo.

Primero de todo reinicializaremos el procesador (si previamente he-

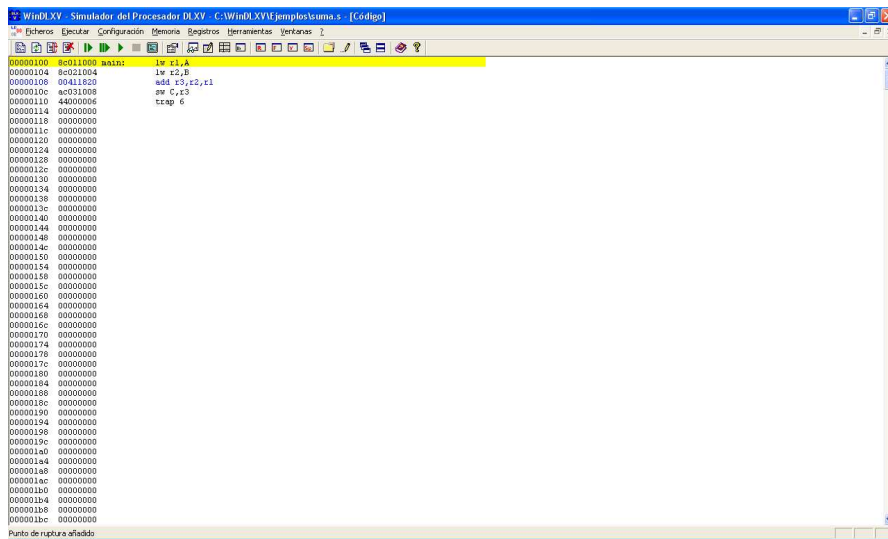


Figura 5.4: Simulando paso a paso: añadiendo un punto de ruptura.

mos ejecutado ciclo a ciclo) a través de la función de menú *Ficheros* ▸ *Reset DLXV*.

A continuación añadiremos el punto de ruptura: para ello en la ventana Código haremos doble clic sobre la instrucción en la que vamos a añadir el punto de ruptura, `add r3, r2, r1` en nuestro caso. Como muestra la Figura 5.4 la instrucción `add` cambia a color azul, indicando que tiene un punto de ruptura y en la barra de estado se indica que se ha añadido un punto de ruptura.

Una vez definido el punto de ruptura, podemos realizar la ejecución del programa hasta ese punto de ruptura (*Ejecutar* ▸ *Sin pausas hasta*). La ejecución se detendrá cuando la instrucción `add` entra en la etapa IF. En la Figura 5.5 vemos que se han ejecutado 2 ciclos de reloj hasta que se alcanzado el punto de ruptura.

Ahora, ya podemos continuar la ejecución ciclo a ciclo (Ciclo 3 de la ejecución ciclo a ciclo explicada anteriormente).

Si se desea eliminar un punto de ruptura añadido previamente, basta hacer doble clic de nuevo sobre la instrucción. Cuando se reinicializa el procesador se eliminan todos los puntos de ruptura añadidos.

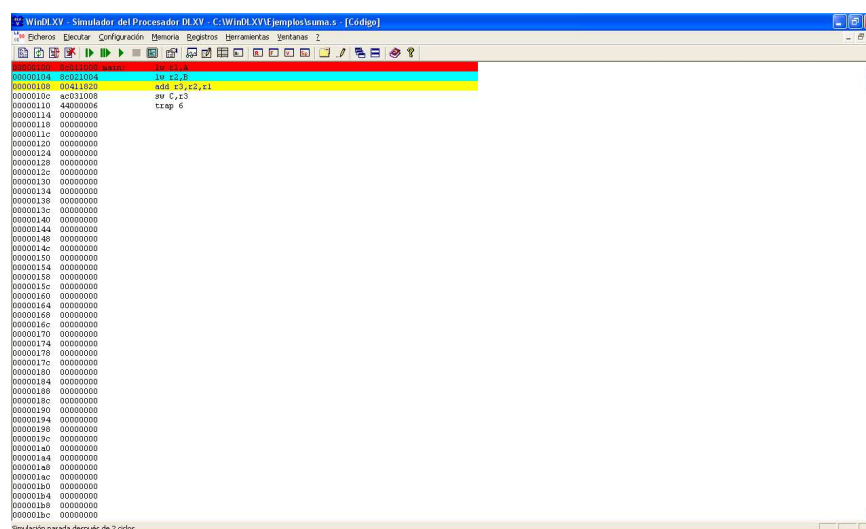


Figura 5.5: Simulando paso a paso: parada en un punto de ruptura.

5.6. Modificando componentes durante la simulación

WinDLXV permite alterar el contenido de la memoria y registros del procesador en cualquier momento de la simulación.

Su alteración puede realizarse de forma rápida haciendo doble clic sobre el registro (ventana Registros) o posición de memoria (ventana Datos) que se desea modificar. Aparecerá una caja de diálogo desde la que modificar su contenido. Ello debe realizarse en el formato definido por defecto para cada uno de los componentes: hexadecimal para los registros de propósito general GPR, punto flotante en simple o doble precisión para los registros de punto flotante FPR, punto flotante en doble precisión para los registros vectoriales y hexadecimal para el contenido de una posición de memoria.

Pero WinDLXV dispone de funciones de menú para modificar el contenido de registros y memoria en cualquier formato, e incluso ensamblar una nueva instrucción en una posición de memoria (ver funciones de menú *Registros* y *Memoria* en la barra de menús).

Capítulo 6

Ensamblador DLXV

El programa WinDLXV permite traducir instrucciones de ensamblador a instrucciones de máquina del procesador DLXV. Por tanto, a partir de un fichero fuente escrito en el ensamblador de dicho procesador, se generará y cargará en memoria un ejecutable por el simulador. Las instrucciones que admite el ensamblador desarrollado para este simulador son:

- **Pseudo-instrucciones o directivas de ensamblador:** Son instrucciones del ensamblador que no se traducen en el código binario. Son órdenes que indican cómo se debe generar el código binario.
- **Instrucciones:** Son instrucciones del ensamblador que se traducen a una instrucción máquina del DLXV.

6.1. Directivas del ensamblador

Las directivas del ensamblador es un conjunto de pseudo-instrucciones que sirven para definir datos, rutinas y todo tipo de información para que el programa ejecutable sea creado de determinada forma y en determinado lugar. Las siguientes directivas son aceptadas por el simulador WinDLXV:

.align n

Alinea el siguiente dato sobre un límite de 2^n byte.

A continuación se muestra un ejemplo del uso de esta directiva:

```
.byte 3  
.align 2  
.word 10
```

Como resultado el siguiente dato de la directiva align, la palabra cuyo valor es 10 se alinearía a 4 bytes.

.ascii “cadena de caracteres”

Almacena la cadena presente en la línea de la directiva como una lista de caracteres, pero no la termina con el carácter NUL. Para incluir caracteres especiales es necesario que vayan precedidos por el carácter “\” que indica que éste y el siguiente especifican un carácter. Se han definido los siguientes caracteres especiales:

- \": indica que es un carácter comilla doble (carácter con código ASCII decimal 34).
- \n: indica que es un salto de línea (carácter con código ASCII decimal 10).
- \r: indica que es un retorno de carro (carácter con código ASCII decimal 13).
- \0: indica que es el carácter NUL (carácter con código ASCII decimal 0).
- \t: indica que es el tabulador (carácter con código ASCII decimal 9).
- \\: indica que es el propio carácter \ (carácter con código ASCII decimal 92).

A continuación se muestra un ejemplo del uso de esta directiva:

```
.org 100
.ascii "1234"
```

Como resultado se cargarán los caracteres con códigos hexadecimales 0x31, 0x32, 0x33 y 0x34 en las posiciones de memoria 100, 101, 102 y 103 respectivamente.

.asciiz “cadena de caracteres”

Almacena la cadena presente en la línea de la directiva como una lista de caracteres, y la termina con el carácter NUL.

Si consideramos el mismo ejemplo mostrado en la directiva .ascii, el resultado sería que se cargarán los caracteres con códigos hexadecimales 0x31, 0x32, 0x33 y 0x34 en las posiciones de memoria 100, 101, 102 y 103, y el carácter 0x00 (NUL) en la posición 104.

.byte b1, b2, ..., bn

Almacena los n bytes (8 bits) en posiciones consecutivas de memoria. Los n bytes pueden especificarse en los siguientes formatos:

- 0xn: es un valor expresado en hexadecimal.
- n: es un valor expresado en decimal.

A continuación se muestra un ejemplo del uso de esta directiva:

```
.org 100
.byte 20, 0x20
```

Como resultado se cargarán los bytes con valores hexadecimales 0x14 (20 en decimal) y 0x20 en las posiciones de memoria 100 y 101, respectivamente.

.data [dirección]

Los elementos siguientes son almacenados en el segmento de datos. Si se proporciona una dirección, los elementos serán cargados comenzando en esa dirección, en otro caso lo hace a partir de la dirección de defecto configurada en WinDLXV (ver función de menú *Configuración* ▸ *Arquitectura*).

A continuación se muestra un ejemplo del uso de esta directiva:

```
.data 0x100
.byte 20, 0x20
```

Como resultado los elementos señalados con la directiva byte se cargarán en el segmento de datos que comienza en la dirección 0x100.

.double d1, d2, ..., dn

Almacena los n números de punto flotante de doble precisión en posiciones consecutivas de memoria.

A continuación se muestra un ejemplo del uso de esta directiva:

```
.org 100
.double 1.0, 3.0
```

Como resultado se cargarán los números de punto flotante 1.0 y 3.0 en doble precisión (8 bytes) a partir de las posiciones 100 y 108, respectivamente.

.float f1, f2, ..., fn

Almacena los n números de punto flotante de simple precisión en posiciones consecutivas de memoria.

A continuación se muestra un ejemplo del uso de esta directiva:

```
.org 100
.float 1.0, 3.0
```

Como resultado se cargarán los números de punto flotante 1.0 y 3.0 en simple precisión (4 bytes) a partir de las posiciones 100 y 104, respectivamente.

.org dirección

Especifica en que posiciones de memoria se ubicarán las variables o el código que aparece a continuación.

A continuación se muestra un ejemplo del uso de esta instrucción:

```
.org 140
.byte 230, 10, 7
```

Como resultado se reservaría 3 bytes a partir de la dirección 140, y se inicializarían con los valores 230, 10 y 7.

.space n

Reserva n bytes de espacio en el segmento actual (debe ser el segmento de datos en WinDLXV). Las posición de memoria reservadas se cargan con un valor indefinido.

A continuación se muestra un ejemplo del uso de esta directiva:

```
.org 140
.space 12
```

Como resultado se reservarían 3 palabras de memoria (12 bytes) a partir de la posición 140.

.text [dirección]

Los elementos siguientes son almacenados en el segmento de código. Si se proporciona una dirección, los elementos serán cargados comenzando en esa dirección, en otro caso lo hace a partir de la dirección de defecto configurada en WinDLXV (ver función de menú *Configuración* ▸ *Arquitectura*).

A continuación se muestra un ejemplo del uso de esta directiva:

```
.text 0x800
addv v1,v2,v7
```

Como resultado la instrucción `addv` se colocaría en el segmento de código que comienza en la dirección 0x800.

.word w1, w2, ..., wn

Almacena los *n* words (32 bits) en posiciones consecutivas de memoria. Los *n* bytes pueden especificarse en los siguientes formatos:

- 0xn: es un valor expresado en hexadecimal.
- n: es un número positivo expresado en decimal.
- -n: es un número negativo expresado en decimal.
- etiqueta: es un puntero a una dirección de memoria.

A continuación se muestra un ejemplo del uso de esta directiva:

```
path:      .org 0
           .ascii "C:\Ejemplos\ejemplo1.s"
           .org 100
           .word 230, 0x1040, path
```

Como resultado se cargará los valores hexadecimales 0x00E6 (230 en decimal), 0x1040 , 0x0000 (dirección de path) a partir de las posiciones de memoria 100, 104 y 108, respectivamente.

6.2. Sintaxis del lenguaje ensamblador

El programa escrito en lenguaje ensamblador se compone de un conjunto de líneas ensamblador con la siguiente sintaxis:

Etiqueta : Mnemónico Operandos ; Comentarios

Cada uno de estos campos debe ir separado por uno o varios blancos o tabuladores. A continuación se describe cada uno de ellos:

- **Etiqueta:** Es una cadena de caracteres alfanuméricos. Este campo es sensible a las mayúsculas. Esto significa que las etiquetas LISTA, Lista, LisStA y lista son todas distintas. Esta cadena de caracteres deberá ir seguida del carácter `:`.
- **Mnemónico:** Es una cadena de caracteres que representa la operación que realiza la instrucción. En el capítulo 7 se hace una descripción exhaustiva de todas las instrucciones consideradas en el ensamblador del DLXV.
- **Operandos:** Es la lista de operandos sobre los que trabaja la instrucción. Estos operandos van separados por comas. Dependiendo de la instrucción se admitirán unos operandos u otros. En el caso de que un operando involucre una etiqueta, se deberá tener en cuenta que el nombrado de las mismas es sensible a las mayúsculas.
- **Comentarios:** Es una cadena de cualquier tipo de caracteres precedida por el carácter `;`. Cuando el programa ensamblador encuentra el carácter `;` ignora el resto de caracteres hasta el final de la línea. Estos comentarios, ignorados por el ensamblador, se utilizan para hacer el programa más fácilmente mantenible en el futuro.

6.3. Instrucciones

Clasificaremos las instrucciones en grupos de acuerdo a su finalidad. Estas instrucciones se describirán de forma exhaustiva en el capítulo 7.

6.3.1. Instrucciones de transferencias de datos

Transfieren datos entre registros y memoria, o entre registros enteros y de punto flotante o registros especiales.

Cualquier de los registros de propósito general, de punto flotante o vectorial se pueden cargar o almacenar, excepto cargar R0 que no tiene efecto. Hay un modo único de direccionamiento, registro base + desplazamiento de 16 bits con signo. Las cargas de media palabra y de byte ubican el objeto cargado en la parte inferior del registro. La parte superior del registro se rellena, bien con la extensión del signo de los valores cargados o con ceros, dependiendo del código de operación. Los números en punto flotante y simple precisión ocupan un registro de punto flotante, mientras que los valores en doble precisión ocupan un par. La conversión entre simple y doble precisión deben realizarse explícitamente.

6.3.2. Instrucciones aritméticas/lógicas

Las operaciones incluyen operaciones aritméticas sencillas y operaciones lógicas: suma, resta, AND, OR, XOR, y desplazamientos (*shifts*). También se proporcionan las formas inmediatas de todas estas instrucciones, con un inmediato con signo-extendido a 16 bits. La operación LHI (carga superior inmediato) carga la mitad superior de un registro, mientras pone a 0 la mitad inferior. Esto permite que, con dos instrucciones, se construya una constante completa de 32 bits.

También hay instrucciones de comparación, que comparan dos registros ($=$, \neq , $<$, $>$, \leq , \geq). Si la condición es cierta, estas instrucciones colocan un 1 en el registro destino (para representar verdadero); en otro caso colocan el valor 0. También hay formas inmediatas para estas comparaciones.

6.3.3. Instrucciones de control de flujo

El control se realiza mediante un conjunto de bifurcaciones y saltos. Las tres instrucciones de bifurcación están diferenciadas por las dos formas de especificar la dirección destino y por si existe o no enlace. Dos bifurcaciones (J, JR) utilizan un desplazamiento con signo de 26 bits sumado al contador del programa (de la instrucción que sigue secuencialmente la bifurcación) para determinar la dirección destino; las otras dos instrucciones de bifurcación (JAL, JALR) especifican un registro que contiene la dirección destino. Hay dos tipos de bifurcación: bifurcación simple, y bifurcación y enlace (utilizada para llamadas a procedimientos). La última coloca la dirección de vuelta en R31.

Todos los saltos son condicionales. La condición de salto se especifica en la instrucción, que puede examinar el registro fuente para compararlo cero o no cero; éste puede ser el valor de un dato o el resultado de una

comparación. La dirección destino del salto se especifica con un desplazamiento con signo de 16 bits que se suma al contador del programa.

6.3.4. Instrucciones de punto flotante

Las instrucciones de punto flotante manipulan los registros de punto flotante e indican si la operación a realizar es en simple o doble precisión. Las operaciones en simple precisión se pueden aplicar a cualquiera de los registros, mientras que las operaciones en doble precisión se aplican sólo a una pareja par-impar (por ejemplo, F4, F5), que se designa por el número de registro par. Las instrucciones de carga y almacenamiento, para los registros de punto flotante, transfieren datos entre los registros de punto flotante y memoria en simple y doble precisión. Las operaciones MOVF y MOVD copian, en punto flotante y simple precisión (MVF) o doble precisión (MOVD), un registro en otro registro del mismo tipo. Las operaciones MOVFP2I y MOVI2FP transfieren datos entre un registro en punto flotante y un registro entero; transferir un valor en doble precisión a dos registros enteros requiere dos instrucciones. También existen multiplicaciones y divisiones enteras que funcionan en registros de puntos flotante de 32 bits, así como conversiones de entero a punto flotante y viceversa.

Las operaciones de punto flotante son suma, resta, multiplicación y división; se utiliza el sufijo D para doble precisión y el F para simple precisión (por ejemplo, ADDD, ADDF, SUBD, SUBF, MULTD, MULTF, DIVD, DIVF). Las comparaciones en punto flotante inicializan un bit del registro de estado especial de punto flotante (FPSR) que puede ser examinado por un tipo de saltos: BFPT y BFPF, salto en punto flotante cierto y salto en punto flotante falso.

6.3.5. Instrucciones vectoriales

Las instrucciones vectoriales tienen el mismo nombre que las instrucciones escalares añadiéndole la letra "V". Son operaciones de punto flotante y doble precisión. Por tanto, ADDV es una suma de dos vectores en doble precisión. Las operaciones vectoriales toman como entrada o un par de registros vectoriales (ADDV) o un registro vectorial y un registro escalar, lo que se designa añadiendo "SV" (ADDSV) o "VS" (SUBVS). En estos dos últimos casos, el valor del registro escalar se utiliza como entrada para todas las operaciones (por ejemplo, la operación ADDSV sumará el contenido de un registro escalar a cada elemento de un registro vectorial). Las operaciones vectoriales siempre tienen como destino un registro vectorial.

Los nombres LV y SV denotan cargar vector y almacenar vector, un operando es el registro vectorial que se va a cargar o almacenar, y el otro operando, que es un registro de propósito general, tiene la dirección de comienzo del vector en memoria. Existen otras instrucciones de carga y almacenamiento no secuencial que son LVWS/SVWS y LVI/SVI donde el acceso a memoria es con una determinada separación (*stride*) o con vector de índice creado con la instrucción CVI.

Además existen instrucciones que operan sobre los dos registros especiales usados en las operaciones vectoriales: registro de longitud del vector y registro de máscara de vector.

6.4. Formato de las instrucciones

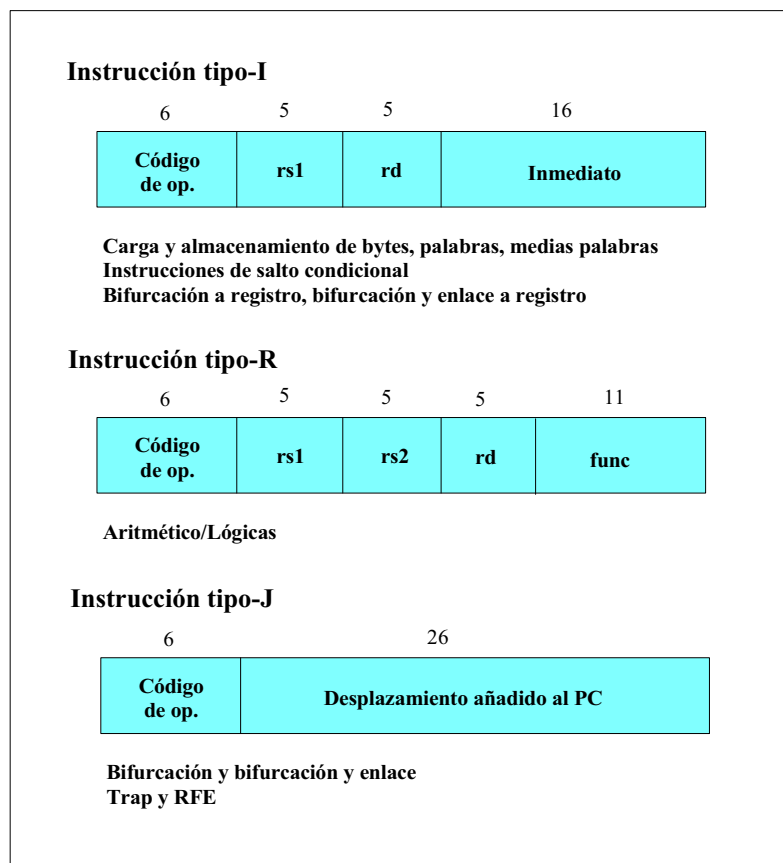


Figura 6.1: Formato de las instrucciones para DLXV.

Todas las instrucciones son de 32 bits con un código de operación principal de 6 bits. La Figura 6.1 muestra la organización de las instrucciones.

Capítulo 7

Juego de instrucciones del DLXV

A continuación se describe el juego de instrucciones del procesador DLXV que se han incluido en simulador WinDLXV.

7.1. Instrucciones de trasferencias de datos

Instrucción LB

Operación:

$$\begin{aligned}addr &\leftarrow ((desp_{16})^{16} \text{ ## } desp_{16...31}) + GPR[rs1] \\ mem &\leftarrow M[addr] \\ GPR[rd] &\leftarrow (mem_0)^{24} \text{ ## } mem_{24...31}\end{aligned}$$

Sintaxis:

```
lb rd, desp(rs1)
lb rd, desp      ; rs1 = r0
```

Descripción:

La instrucción LB lee 1 byte del sistema de memoria y lo carga en el registro destino rd. La dirección de memoria se obtiene sumando a la dirección base del registro rs1 el valor inmediato de 16 bits extendido con su signo. El byte se carga en la parte inferior del registro rd, rellenando la parte superior con la extensión del signo del valor cargado.

Instrucción LBU

Operación:

$$\begin{aligned}addr &\leftarrow ((desp_{16})^{16} \text{##} desp_{16...31}) + GPR[rs1] \\ mem &\leftarrow M[addr] \\ GPR[rd] &\leftarrow (0)^{24} \text{##} mem_{24...31}\end{aligned}$$

Sintaxis:

```
lbu rd, desp(rs1)
lbu rd, desp      ; rs1 = r0
```

Descripción:

La instrucción **LBU** lee 1 byte del sistema de memoria y lo carga en el registro destino rd. La dirección de memoria se obtiene sumando a la dirección base del registro rs1 el valor inmediato de 16 bits extendido con su signo. El byte se carga en la parte inferior del registro rd, rellenando la parte superior con ceros.

Instrucción SB

Operación:

$$\begin{aligned}addr &\leftarrow ((desp_{16})^{16} \text{##} desp_{16...31}) + GPR[rs1] \\ M[addr] &\leftarrow GPR[rd]_{24...31}\end{aligned}$$

Sintaxis:

```
sb desp(rs1), rd
sb desp, rd      ; rs1 = r0
```

Descripción:

La instrucción **SB** escribe el byte (8 bits) menos significativo del registro especificado en rd en el sistema de memoria. La dirección de memoria se obtiene sumando a la dirección base del registro rs1 el valor inmediato de 16 bits extendido con su signo.

Instrucción LH

Operación:

$$\begin{aligned}addr &\leftarrow ((desp_{16})^{16} \text{##} desp_{16...31}) + GPR[rs1] \\ mem &\leftarrow M[addr] \\ GPR[rd] &\leftarrow (mem_0)^{16} \text{##} mem_{16...31}\end{aligned}$$

Sintaxis:

```
lh rd, desp(rs1)
lh rd, desp          ; rs1 = r0
```

Descripción:

La instrucción LH lee 2 bytes del sistema de memoria y los carga en el registro destino rd. La dirección de memoria se obtiene sumando a la dirección base del registro rs1 el valor inmediato de 16 bits extendido con su signo. Los 2 bytes se cargan en la parte inferior del registro rd, rellenando la parte superior con la extensión del signo del valor cargado.

Instrucción LHU

Operación:

$$\begin{aligned} addr &\leftarrow ((desp_{16})^{16} \text{ ## } desp_{16...31}) + GPR[rs1] \\ mem &\leftarrow M[addr] \\ GPR[rd] &\leftarrow (0)^{16} \text{ ## } mem_{16...31} \end{aligned}$$

Sintaxis:

```
lhu rd, desp(rs1)
lhu rd, desp          ; rs1 = r0
```

Descripción:

La instrucción LHU lee 2 bytes del sistema de memoria y los carga en el registro destino rd. La dirección de memoria se obtiene sumando a la dirección base del registro rs1 el valor inmediato de 16 bits extendido con su signo. Los 2 bytes se cargan en la parte inferior del registro rd, rellenando la parte superior con ceros.

Instrucción SH

Operación:

$$\begin{aligned} addr &\leftarrow ((desp_{16})^{16} \text{ ## } desp_{16...31}) + GPR[rs1] \\ M[addr] &\leftarrow GPR[rd]_{16...31} \end{aligned}$$

Sintaxis:

```
sh desp(rs1), rd
sh desp, rd      ; rs1 = r0
```

Descripción:

La instrucción **SH** escribe los 2 bytes menos significativos del registro especificado en *rd* en el sistema de memoria. La dirección de memoria se obtiene sumando a la dirección base del registro *rs1* el valor inmediato de 16 bits extendido con su signo.

Instrucción LW

Operación:

$$\begin{aligned} addr &\leftarrow ((desp_{16})^{16} \# \# desp_{16...31}) + GPR[rs1] \\ mem &\leftarrow M[addr] \\ GPR[rd] &\leftarrow mem_{0...31} \end{aligned}$$

Sintaxis:

```
lw rd, desp(rs1)
lw rd, desp      ; rs1 = r0
```

Descripción:

La instrucción **LW** lee 1 palabra (4 bytes) del sistema de memoria y la carga en el registro destino *rd*. La dirección de memoria se obtiene sumando a la dirección base del registro *rs1* el valor inmediato de 16 bits extendido con su signo.

Instrucción SW

Operación:

$$\begin{aligned} addr &\leftarrow ((desp_{16})^{16} \# \# desp_{16...31}) + GPR[rs1] \\ M[addr] &\leftarrow GPR[rd]_{0...31} \end{aligned}$$

Sintaxis:

```
sw desp(rs1), rd
sw desp, rd      ; rs1 = r0
```

Descripción:

La instrucción **SW** escribe el contenido del registro especificado en *rd* en el sistema de memoria. La dirección de memoria se obtiene sumando a la dirección base del registro *rs1* el valor inmediato de 16 bits extendido con su signo.

Instrucción LF

Operación:

$$\begin{aligned} addr &\leftarrow ((desp_{16})^{16} \text{ ## } desp_{16...31}) + GPR[rs1] \\ mem &\leftarrow M[addr] \\ FPR[fd] &\leftarrow mem_{0...31} \end{aligned}$$

Sintaxis:

```
lf fd, desp(rs1)
lf fd, desp      ; rs1 = r0
```

Descripción:

La instrucción LF carga un punto flotante en simple precisión desde el sistema de memoria al registro de punto flotante destino fd. La dirección de memoria se obtiene sumando a la dirección base del registro rs1 el valor inmediato de 16 bits extendido con su signo.

Instrucción LD

Operación:

$$\begin{aligned} addr &\leftarrow ((desp_{16})^{16} \text{ ## } desp_{16...31}) + GPR[rs1] \\ mem &\leftarrow M[addr] \\ FPR[fd + 1] \text{ ## } FPR[fd] &\leftarrow mem_{0...63} \end{aligned}$$

Sintaxis:

```
ld fd, desp(rs1)
ld fd, desp      ; rs1 = r0
```

Descripción:

La instrucción LD carga un punto flotante en doble precisión desde el sistema de memoria a los registros de punto flotante destino fd y fd+1. La dirección de memoria se obtiene sumando a la dirección base del registro rs1 el valor inmediato de 16 bits extendido con su signo.

Instrucción SF

Operación:

$$\begin{aligned} addr &\leftarrow ((desp_{16})^{16} \text{##} desp_{16...31}) + GPR[rs1] \\ M[addr] &\leftarrow FPR[fd]_{0...31} \end{aligned}$$

Sintaxis:

```
sf desp(rs1), fd
sf desp, fd      ; rs1 = r0
```

Descripción:

La instrucción **SF** escribe el punto flotante en simple precisión desde el registro de punto flotante *fd* al sistema de memoria. La dirección de memoria se obtiene sumando a la dirección base del registro *rs1* el valor inmediato de 16 bits extendido con su signo.

Instrucción SD

Operación:

$$\begin{aligned} addr &\leftarrow ((desp_{16})^{16} \text{##} desp_{16...31}) + GPR[rs1] \\ M[addr] &\leftarrow FPR[fd + 1]_{0...31} \text{##} FPR[fd]_{0...31} \end{aligned}$$

Sintaxis:

```
sd desp(rs1), fd
sd desp, fd      ; rs1 = r0
```

Descripción:

La instrucción **SD** escribe el punto flotante en doble precisión desde los registros de punto flotante *fd* y *fd+1* al sistema de memoria. La dirección de memoria se obtiene sumando a la dirección base del registro *rs1* el valor inmediato de 16 bits extendido con su signo.

Instrucción MOVI2S

Operación:

$$vlr \leftarrow GPR[rs1]$$

Sintaxis:

```
movi2s vlr, rs1
```

Descripción:

La instrucción **MOVI2S** transfiere el contenido del registro rs1 al registro de longitud vectorial (VLR).

Instrucción MOVS2I

Operación:

$$GPR[rs1] \leftarrow vlr$$

Sintaxis:

`movs2i rd, vlr`

Descripción:

La instrucción **MOVS2I** transfiere el contenido del registro de longitud vectorial (VLR) al registro rd.

Instrucción MOVF

Operación:

$$FPR[fd] \leftarrow FPR[fs1]$$

Sintaxis:

`movf fd, fs1`

Descripción:

La instrucción **MOVF** copia el contenido del registro en punto flotante fs1 en el registro de punto flotante fd.

Instrucción MOVD

Operación:

$$FPR[fd + 1] \text{ ## } FPR[fd] \leftarrow FPR[fs1 + 1] \text{ ## } FPR[fs1]$$

Sintaxis:

`movd fd, fs1`

Descripción:

La instrucción **MOVD** copia el contenido del par de registros de punto flotante fs1 y fs1+1 en el par de registros de punto flotante fd y fd+1.

Instrucción MOVFP2I

Operación:

$$GPR[rd] \leftarrow FPR[fs1]$$

Sintaxis:

`movfp2i rd, fs1`

Descripción:

La instrucción **MOVFP2I** transfiere 32 bits del registro en punto flotante fs1 en el registro GPR rd.

Instrucción MOVI2FP

Operación:

$$FPR[fd] \leftarrow GPR[rs1]$$

Sintaxis:

`movi2fp fd, rs1`

Descripción:

La instrucción **MOVI2FP** transfiere 32 bits del registro GPR rs1 en el registro de punto flotante fd.

Instrucción MOVF2S

Operación:

$$vm \leftarrow FPR[fs1 + 1] \text{ \#\# } FPR[fs1]$$

Sintaxis:

`movf2s vm, fs1`

Descripción:

La instrucción **MOVF2S** transfiere el contenido de los registros de punto flotante fs1 y fs1+1 al registro de máscara vectorial.

Instrucción MOVS2F

Operación:

$$FPR[fd + 1] \text{ \#\# } FPR[fd] \longleftarrow vm$$

Sintaxis:

`movs2f fd,vm`

Descripción:

La instrucción **MOVS2F** transfiere el contenido del registro de máscara vectorial a los registros de punto flotante `fd` y `fd+1`.

7.2. Instrucciones aritméticas/lógicas

Instrucción ADD

Operación:

$$GPR[rd] \longleftarrow GPR[rs1] + GPR[rs2]$$

Sintaxis:

`add rd,rs1,rs2`

Descripción:

La instrucción **ADD** suma el contenido del registro `rs1` con el contenido del registro `rs2` y almacena el resultado en `rd`. Es siempre una suma con signo.

Instrucción ADDI

Operación:

$$GPR[rd] \longleftarrow GPR[rs1] + ((imm_{16})^{16} \text{ \#\# } imm_{16...31})$$

Sintaxis:

`addi rd,rs1,imm`

Descripción:

La instrucción **ADDI** suma el contenido del registro `rs1` con el valor inmediato con signo `imm` y almacena el resultado en `rd`. Es siempre una suma con signo.

Instrucción ADDU

Operación:

$$GPR[rd] \leftarrow GPR[rs1] + GPR[rs2]$$

Sintaxis:

```
addu rd, rs1, rs2
```

Descripción:

La instrucción **ADDU** suma el contenido del registro rs1 con el contenido del registro rs2 y almacena el resultado en rd. Es siempre una suma sin signo.

Instrucción ADDUI

Operación:

$$GPR[rd] \leftarrow GPR[rs1] + (0^{16} \text{ ## } imm_{16...31})$$

Sintaxis:

```
addui rd, rs1, imm
```

Descripción:

La instrucción **ADDUI** suma el contenido del registro rs1 con el valor inmediato sin signo imm y almacena el resultado en rd. Es siempre una suma sin signo.

Instrucción SUB

Operación:

$$GPR[rd] \leftarrow GPR[rs1] - GPR[rs2]$$

Sintaxis:

```
sub rd, rs1, rs2
```

Descripción:

La instrucción **SUB** resta el contenido del registro rs1 con el contenido del registro rs2 y almacena el resultado en rd. Es siempre una resta con signo.

Instrucción SUBI

Operación:

$$GPR[rd] \leftarrow GPR[rs1] - ((imm_{16})^{16} \text{ ## } imm_{16...31})$$

Sintaxis:

```
subi rd,rs1,imm
```

Descripción:

La instrucción **SUBI** resta el contenido del registro rs1 con el valor inmediato con signo imm y almacena el resultado en rd. Es siempre una resta con signo.

Instrucción SUBU

Operación:

$$GPR[rd] \leftarrow GPR[rs1] - GPR[rs2]$$

Sintaxis:

```
subu rd,rs1,rs2
```

Descripción:

La instrucción **SUBU** resta el contenido del registro rs1 con el contenido del registro rs2 y almacena el resultado en rd. Es siempre una resta sin signo.

Instrucción SUBUI

Operación:

$$GPR[rd] \leftarrow GPR[rs1] - (0^{16} \text{ ## } imm_{16...31})$$

Sintaxis:

```
subui rd,rs1,imm
```

Descripción:

La instrucción **SUBUI** resta el contenido del registro rs1 con el valor inmediato sin signo imm y almacena el resultado en rd. Es siempre una resta sin signo.

Instrucción MULT

Operación:

$$GPR[rd] \leftarrow GPR[rs1] \cdot GPR[rs2]$$

Sintaxis:

```
mult rd,rs1,rs2
```

Descripción:

La instrucción **MULT** multiplica el contenido del registro rs1 con el contenido del registro rs2 y almacena el resultado en rd. Es siempre una multiplicación con signo. Si el resultado de la multiplicación no puede representarse como un entero de 32 bits se produce una excepción de *overflow*.

Instrucción MULTU

Operación:

$$GPR[rd] \leftarrow GPR[rs1] \cdot GPR[rs2]$$

Sintaxis:

```
multu rd,rs1,rs2
```

Descripción:

La instrucción **MULTU** multiplica el contenido del registro rs1 con el contenido del registro rs2 y almacena el resultado en rd. Es siempre una multiplicación sin signo. Si el resultado de la multiplicación no puede representarse como un entero de 32 bits se produce una excepción de *overflow*.

Instrucción DIV

Operación:

$$GPR[rd] \leftarrow GPR[rs1] \div GPR[rs2]$$

Sintaxis:

```
div rd,rs1,rs2
```

Descripción:

La instrucción **DIV** divide el contenido del registro rs1 entre el contenido del registro rs2 y almacena el resultado en rd. Es siempre una división con signo. Si el contenido de rs2 es cero provoca una excepción de división entera por cero.

Instrucción DIVU

Operación:

$$GPR[rd] \leftarrow GPR[rs1] \div GPR[rs2]$$

Sintaxis:

```
divu rd, rs1, rs2
```

Descripción:

La instrucción **DIVU** divide el contenido del registro rs1 entre el contenido del registro rs2 y almacena el resultado en rd. Es siempre una división sin signo. Si el contenido de rs2 es cero provoca una excepción de división entera por cero.

Instrucción AND

Operación:

$$GPR[rd] \leftarrow GPR[rs1] \text{ and } GPR[rs2]$$

Sintaxis:

```
and rd, rs1, rs2
```

Descripción:

La instrucción **AND** hace la operación lógica and del registro rs1 con el registro rs2 y el resultado lo almacena en el registro rd.

Instrucción ANDI

Operación:

$$GPR[rd] \leftarrow (0^{16} \text{ ## } imm_{16...31}) \text{ and } GPR[rs1]$$

Sintaxis:

```
andi rd, rs1, imm
```

Descripción:

La instrucción **ANDI** extiende el valor inmediato de 16 bits hasta 32 con ceros y realiza la operación lógica and con el registro rs1. El resultado es almacenado en el registro rd.

Instrucción OR

Operación:

$$GPR[rd] \leftarrow GPR[rs1] \text{ or } GPR[rs2]$$

Sintaxis:

`or rd, rs1, rs2`

Descripción:

La instrucción **OR** hace la operación lógica or del registro rs1 con el registro rs2 y el resultado lo almacena en el registro rd.

Instrucción ORI

Operación:

$$GPR[rd] \leftarrow (0^{16} \text{ ## } imm_{16...31}) \text{ or } GPR[rs1]$$

Sintaxis:

`ori rd, rs1, imm`

Descripción:

La instrucción **ORI** extiende el valor inmediato de 16 bits hasta 32 con ceros y realiza la operación lógica or con el registro rs1. El resultado es almacenado en el registro rd.

Instrucción XOR

Operación:

$$GPR[rd] \leftarrow GPR[rs1] \text{ xor } GPR[rs2]$$

Sintaxis:

`xor rd, rs1, rs2`

Descripción:

La instrucción **XOR** hace la operación lógica xor del registro rs1 con el registro rs2 y el resultado lo almacena en el registro rd.

Instrucción XORI

Operación:

$$GPR[rd] \leftarrow GPR[rs1] \text{ xor } (0^{16} \text{ ## } imm_{16..31})$$

Sintaxis:

```
xori rd,rs1,imm
```

Descripción:

La instrucción **XORI** extiende el valor inmediato de 16 bits hasta 32 con ceros y realiza la operación lógica xor con el registro rs1. El resultado es almacenado en el registro rd.

Instrucción LHI

Operación:

$$GPR[rd] \leftarrow imm_{16..31} \text{ ## } 0^{16}$$

Sintaxis:

```
lhi rd,imm
```

Descripción:

La instrucción **LHI** carga el valor inmediato de 16 bits en la mitad superior del registro rd, mientras pone a cero la mitad inferior de rd.

Instrucción SLL

Operación:

$$\begin{aligned} desp &\leftarrow GPR[rs2]_{27..31} \\ GPR[rd] &\leftarrow GPR[rs1]_{desp..31} \text{ ## } 0^{desp} \end{aligned}$$

Sintaxis:

```
sll rd,rs1,rs2
```

Descripción:

La instrucción **SLL** realiza el desplazamiento lógico hacia la izquierda del registro rs1 el número de bits indicado en los 5 bits menos significativos del registro rs2, insertando ceros en los bits vaciados.

Instrucción SRL

Operación:

$$\begin{aligned} desp &\leftarrow GPR[rs2]_{27...31} \\ GPR[rd] &\leftarrow 0^{desp} \## GPR[rs1]_{0...(31-desp)} \end{aligned}$$

Sintaxis:

`srl rd, rs1, rs2`

Descripción:

La instrucción **SRL** realiza el desplazamiento lógico hacia la derecha del registro rs1 el número de bits indicado en los 5 bits menos significativos del registro rs2, insertando ceros en los bits vaciados.

Instrucción SRA

Operación:

$$\begin{aligned} desp &\leftarrow GPR[rs2]_{27...31} \\ GPR[rd] &\leftarrow (GPR[rs1]_0)^{desp} \## GPR[rs1]_{0...(31-desp)} \end{aligned}$$

Sintaxis:

`sra rd, rs1, rs2`

Descripción:

La instrucción **SRA** realiza el desplazamiento aritmético hacia la derecha del registro rs1 el número de bits indicado en los 5 bits menos significativos del registro rs2, duplicando el bit de signo en los bits vaciados.

Instrucción SLLI

Operación:

$$\begin{aligned} desp &\leftarrow imm_{27...31} \\ GPR[rd] &\leftarrow GPR[rs1]_{desp...31} \## 0^{desp} \end{aligned}$$

Sintaxis:

`slli rd, rs1, imm`

Descripción:

La instrucción **SLLI** realiza el desplazamiento lógico hacia la izquierda del registro rs1 el número de bits indicado en los 5 bits menos significativos del valor inmediato imm, insertando ceros en los bits vaciados.

Instrucción SRLI

Operación:

$$\begin{aligned} desp &\leftarrow imm_{27...31} \\ GPR[rd] &\leftarrow 0^{desp} \## GPR[rs1]_{0...(31-desp)} \end{aligned}$$

Sintaxis:

`srli rd, rs1, rs2`

Descripción:

La instrucción **SRLI** realiza el desplazamiento lógico hacia la derecha del registro `rs1` el número de bits indicado en los 5 bits menos significativos del valor inmediato `imm`, insertando ceros en los bits vaciados.

Instrucción SRAI

Operación:

$$\begin{aligned} desp &\leftarrow imm_{27...31} \\ GPR[rd] &\leftarrow (GPR[rs1]_0)^{desp} \## GPR[rs1]_{0...(31-desp)} \end{aligned}$$

Sintaxis:

`srai rd, rs1, rs2`

Descripción:

La instrucción **SRAI** realiza el desplazamiento aritmético hacia la derecha del registro `rs1` el número de bits indicado en los 5 bits menos significativos del valor inmediato `imm`, duplicando el bit de signo en los bits vaciados.

Instrucciones S__ (LT, GT, LE, GE, EQ, NE)

Operación:

$$\begin{aligned} \text{si } (GPR[rs1] \text{ cond } GPR[rs2]) \\ GPR[rd] &\leftarrow 0^{31} \## 1 \\ \text{si no} \\ GPR[rd] &\leftarrow 0^{32} \end{aligned}$$

Sintaxis:

`s__ rd, rs1, rs2`

Descripción:

La instrucción `s__` compara los registros `rs1` y `rs2` según la condición LT, GT, LE, GE, EQ, NE. Si la condición es cierta coloca un 1 en el registro `rd`, en otro caso coloca el valor 0. Las siguientes condiciones son posibles:

- LT: menor ($rs1 < rs2$)
- GT: mayor ($rs1 > rs2$)
- LE: menor o igual ($rs1 \leq rs2$)
- GE: mayor o igual ($rs1 \geq rs2$)
- EQ: igual ($rs1 = rs2$)
- NE: no igual ($rs1 \neq rs2$)

Instrucción `S__I` (LT, GT, LE, GE, EQ, NE)

Operación:

$$\begin{aligned} \text{si } (GPR[rs1] \text{ cond } ((imm_{16})^{16} \text{ ## } imm_{16...31})) \\ GPR[rd] &\leftarrow 0^{31} \text{ ## } 1 \\ \text{sino} \\ GPR[rd] &\leftarrow 0^{32} \end{aligned}$$

Sintaxis:

`s__i rd, rs1, imm`

Descripción:

La instrucción `s__I` compara el registro `rs1` y el valor inmediato de 16 bits con signo `imm` según la condición LT, GT, LE, GE, EQ, NE. Si la condición es cierta coloca un 1 en el registro `rd`, en otro caso coloca el valor 0. Las siguientes condiciones son posibles:

- LT: menor ($rs1 < imm$)
- GT: mayor ($rs1 > imm$)
- LE: menor o igual ($rs1 \leq imm$)
- GE: mayor o igual ($rs1 \geq imm$)
- EQ: igual ($rs1 = imm$)
- NE: no igual ($rs1 \neq imm$)

7.3. Instrucciones de control de flujo

Instrucción BEQZ

Operación:

$$\begin{aligned} & \text{si } (GPR[rs1] = 0) \\ & \quad PC \leftarrow (PC + 4) + ((dest_{16})^{16} \text{ ## } dest_{16...31}) \end{aligned}$$

Sintaxis:

`beqz rs1, dest`

Descripción:

La instrucción **BEQZ** examina el registro `rs1`. Si es igual a cero efectúa el salto. Para calcular la dirección de destino el desplazamiento inmediato de 16 bits indicado en `dest` es extendido con signo y sumado a la dirección de la instrucción siguiente.

Instrucción BNEZ

Operación:

$$\begin{aligned} & \text{si } (GPR[rs1] \neq 0) \\ & \quad PC \leftarrow (PC + 4) + ((dest_{16})^{16} \text{ ## } dest_{16...31}) \end{aligned}$$

Sintaxis:

`bnez rs1, dest`

Descripción:

La instrucción **BNEZ** examina el registro `rs1`. Si es distinto de cero efectúa el salto. Para calcular la dirección de destino el desplazamiento inmediato de 16 bits indicado en `dest` es extendido con signo y sumado a la dirección de la instrucción siguiente.

Instrucción BFPT

Operación:

$$\begin{aligned} & \text{si } (FPSR = true) \\ & \quad PC \leftarrow (PC + 4) + ((dest_{16})^{16} \text{ ## } dest_{16...31}) \end{aligned}$$

Sintaxis:

bfpt dest

Descripción:

La instrucción **BFPT** examina el registro de bit de estado de punto flotante. Si es cierto (*true*) efectúa el salto. Para calcular la dirección de destino el desplazamiento inmediato de 16 bits indicado en dest es extendido con signo y sumado a la dirección de la instrucción siguiente.

Instrucción BFPP

Operación:

$$\begin{aligned} & \text{si } (FPSR = false) \\ & \quad PC \leftarrow (PC + 4) + ((dest_{16})^{16} \text{ ## } dest_{16...31}) \end{aligned}$$

Sintaxis:

bfpf rs1, dest

Descripción:

La instrucción **BFPF** examina el registro de bit de estado de punto flotante. Si no es cierto (*false*) efectúa el salto. Para calcular la dirección de destino el desplazamiento inmediato de 16 bits indicado en dest es extendido con signo y sumado a la dirección de la instrucción siguiente.

Instrucción J

Operación:

$$PC \leftarrow (PC + 4) + ((dest_0)^5 \text{ ## } dest_{5...31})$$

Sintaxis:

j dest

Descripción:

La instrucción **J** provoca un salto incondicional. Para calcular la dirección de destino el desplazamiento inmediato de 26 bits indicado en dest es extendido con signo y sumado a la dirección de la instrucción siguiente.

Instrucción JR

Operación:

$$PC \leftarrow GPR[rs1]$$

Sintaxis:

`jr rs1`

Descripción:

La instrucción JR provoca un salto incondicional a la dirección contenida en el registro rs1.

Instrucción JAL

Operación:

$$PC \leftarrow (PC + 4) + ((dest_0)^5 \text{ ## } dest_{5...31})$$
$$R31 \leftarrow PC + 4$$

Sintaxis:

`jal dest`

Descripción:

La instrucción JAL provoca un salto incondicional a subrutina (bifurca y enlaza). Para calcular la dirección de destino el desplazamiento inmediato de 26 bits indicado en dest es extendido con signo y sumado a la dirección de la instrucción siguiente. La dirección de retorno se almacena en R31.

Instrucción JALR

Operación:

$$PC \leftarrow GPR[rs1]$$
$$R31 \leftarrow PC + 4$$

Sintaxis:

`jalr rs1`

Descripción:

La instrucción JALR provoca un salto incondicional a una subrutina (bifurca y enlaza). La dirección destino se encuentra en el registro rs1. La dirección de retorno se almacena en R31.

Instrucción TRAP

Operación:

$$PC \leftarrow ((dest_0)^5 \text{ ## } dest_{5...31})$$
$$IAR \leftarrow PC + 4$$

Sintaxis:

trap dest

Descripción:

La instrucción **TRAP** transfiere el control a una rutina del sistema operativo. Para calcular la dirección de destino el desplazamiento inmediato de 26 bits indicado en dest es extendido con signo y sumado a la dirección de la instrucción siguiente. Ver sección 8.

Instrucción RFE

Operación:

$$PC \leftarrow IAR$$

Sintaxis:

rfe dest

Descripción:

La instrucción **RFE** vuelve al código del usuario desde una excepción. Instrucción no implementada en WinDLXV.

7.4. Instrucciones de punto flotante

Instrucción ADDD

Operación:

$$FPR[fd + 1] \text{ ## } FPR[fd] \leftarrow$$
$$\leftarrow (FPR[fs1 + 1] \text{ ## } FPR[fs1]) +$$
$$+ (FPR[fs2 + 1] \text{ ## } FPR[fs2])$$

Sintaxis:

addd fd, fs1, fs2

Descripción:

La instrucción **ADDD** suma los números flotantes en doble precisión contenidos en la pareja de registros de punto flotante $fs1/fs1+1$ y la pareja $fs2/fs2+1$. La suma la almacena en la pareja de registros de punto flotante $fd/fd+1$.

Instrucción ADDF

Operación:

$$FPR[fd] \leftarrow FPR[fs1] + FPR[fs2]$$

Sintaxis:

`addf fd, fs1, fs2`

Descripción:

La instrucción **ADDF** suma los números flotantes en simple precisión contenidos en los registros de punto flotante $fs1$ y $fs2$. La suma la almacena en el registro de punto flotante fd .

Instrucción SUBD

Operación:

$$\begin{aligned} FPR[fd+1] \&\& FPR[fd] \leftarrow \\ \leftarrow & (FPR[fs1+1] \&\& FPR[fs1]) - \\ - & (FPR[fs2+1] \&\& FPR[fs2]) \end{aligned}$$

Sintaxis:

`subd fd, fs1, fs2`

Descripción:

La instrucción **SUBD** resta al número flotante en doble precisión de la pareja de registros $fs1/fs1+1$, el número flotante de doble precisión de la pareja $fs2/fs2+1$. La resta la almacena en la pareja de registros $rd/rd+1$.

Instrucción SUBF

Operación:

$$FPR[fd] \leftarrow FPR[fs1] - FPR[fs2]$$

Sintaxis:

```
subf fd, fs1, fs2
```

Descripción:

La instrucción **SUBF** resta al número flotante en simple precisión del registro fs1, el número flotante del registro fs2. La resta la almacena en el registro de punto flotante fd.

Instrucción MULTD

Operación:

$$\begin{aligned} FPR[fd + 1] \text{ ## } FPR[fd] &\leftarrow \\ &\leftarrow (FPR[fs1 + 1] \text{ ## } FPR[fs1]) \cdot \\ &\cdot (FPR[fs2 + 1] \text{ ## } FPR[fs2]) \end{aligned}$$

Sintaxis:

```
multd fd, fs1, fs2
```

Descripción:

La instrucción **MULTD** multiplica los números flotantes en doble precisión de la pareja de registros fs1/fs1+1 y la pareja fs2/fs2+1. El resultado de la multiplicación lo almacena en la pareja de registros de punto flotante fd/fd+1. Si el resultado de la multiplicación no puede representarse como un punto flotante de 64 bits se produce una excepción de *overflow*.

Instrucción MULTF

Operación:

$$FPR[fd] \leftarrow FPR[fs1] \cdot FPR[fs2]$$

Sintaxis:

```
multf fd, fs1, fs2
```


Descripción:

La instrucción **MULTF** multiplica los números flotantes en simple precisión contenidos en los registros de punto flotante $fs1$ y $fs2$. El resultado de la multiplicación lo almacena en el registro de punto flotante fd . Si el resultado de la multiplicación no puede representarse como un punto flotante de 32 bits se produce una excepción de *overflow*.

Instrucción DIVD

Operación:

$$\begin{aligned} FPR[fd + 1] \text{ \#\# } FPR[fd] &\leftarrow \\ &\leftarrow (FPR[fs1 + 1] \text{ \#\# } FPR[fs1]) \div \\ &\div (FPR[fs2 + 1] \text{ \#\# } FPR[fs2]) \end{aligned}$$

Sintaxis:

`divd fd, fs1, fs2`

Descripción:

La instrucción **DIVD** divide el número flotante en doble precisión de la pareja de registros $fs1/fs1+1$ entre el número flotante en doble precisión de la pareja $fs2/fs2+1$. El resultado de la división lo almacena en la pareja de registro de punto flotante $fd/fd+1$. Si el contenido de $fs2/fs2+1$ es 0.0 provoca una excepción de división por cero.

Instrucción DIVF

Operación:

$$FPR[fd] \leftarrow FPR[fs1] \div FPR[fs2]$$

Sintaxis:

`divf fd, fs1, fs2`

Descripción:

La instrucción **DIVF** divide el número flotante en simple precisión del registro $fs1$ entre el número flotante del registro $fs2$. El resultado de la división lo almacena en el registro de punto flotante fd . Si el contenido de $fs2$ es 0.0 provoca una excepción de división por cero.

Instrucción CVTF2D

Operación:

$$FPR[fd + 1] \#\# FPR[fd] \longleftarrow FPR[fs1]$$

Sintaxis:

`cvtf2d fd, fs1`

Descripción:

La instrucción **CVTF2D** convierte el número flotante en simple precisión contenido en el registro `fs1` a número flotante en doble precisión, y almacena el resultado en la pareja de registros `fd` y `fd1`.

Instrucción CVTF2I

Operación:

$$FPR[fd] \longleftarrow FPR[fs1]$$

Sintaxis:

`cvtf2i fd, fs1`

Descripción:

La instrucción **CVTF2I** convierte el número flotante en simple precisión contenido en el registro `fs1` a número entero, y almacena el resultado en el registro de punto flotante `fd`.

Instrucción CVTD2F

Operación:

$$FPR[fd] \longleftarrow FPR[fs1 + 1] \#\# FPR[fs1]$$

Sintaxis:

`cvtd2f fd, fs1`

Descripción:

La instrucción **CVTD2F** convierte el número flotante en doble precisión contenido en la pareja de registros `fs1` y `fs1+1` a número flotante en simple precisión, y almacena el resultado en el registro `fd`.

Instrucción CVTD2I

Operación:

$$FPR[fd] \leftarrow FPR[fs1 + 1] \text{ ## } FPR[fs1]$$

Sintaxis:

`cvtd2i fd, fs1`

Descripción:

La instrucción **CVTD2I** convierte el número flotante en doble precisión contenido en la pareja de registros `fs1` y `fs1+1` a número entero, y almacena el resultado en el registro `fd`.

Instrucción CVTI2F

Operación:

$$FPR[fd] \leftarrow FPR[fs1]$$

Sintaxis:

`cvti2f fd, fs1`

Descripción:

La instrucción **CVTI2F** convierte el número entero contenido en el registro de punto flotante `fs1` a número flotante en simple precisión, y almacena el resultado en el registro `fd`.

Instrucción CVTI2D

Operación:

$$FPR[fd + 1] \text{ ## } FPR[fd] \leftarrow FPR[fs1]$$

Sintaxis:

`cvti2d fd, fs1`

Descripción:

La instrucción **CVTI2D** convierte el número entero contenido en el registro `fs1` a número flotante en doble precisión, y almacena el resultado en la pareja de registros `fd` y `fd+1`.

Instrucciones __D (LT, GT, LE, GE, EQ, NE)

Operación:

$$\begin{aligned} & \text{si } ((FPR[fs1+1] \text{ ## } FPR[fs1]) \text{ cond } (FPR[fs2+1] \text{ ## } FPR[fs2])) \\ & \quad FPSR \leftarrow 1 \\ & \text{si no} \\ & \quad FPSR \leftarrow 0 \end{aligned}$$

Sintaxis:

$$\text{__D } fs1, fs2$$

Descripción:

La instrucción __D compara los puntos flotantes en doble precisión de las parejas de registros de punto flotante fs1/fs1+1 y fs2/fs2+1 según la condición LT, GT, LE, GE, EQ, NE. Si la condición es cierta coloca un 1 en el registro de estado especial de punto flotante (FPSR), en otro caso coloca el valor 0. Las siguientes condiciones son posibles:

- LT: menor ($fs1+1 \text{ ## } fs1 < fs2+1 \text{ ## } fs2$)
- GT: mayor ($fs1+1 \text{ ## } fs1 > fs2+1 \text{ ## } fs2$)
- LE: menor o igual ($fs1+1 \text{ ## } fs1 \leq fs2+1 \text{ ## } fs2$)
- GE: mayor o igual ($fs1+1 \text{ ## } fs1 \geq fs2+1 \text{ ## } fs2$)
- EQ: igual ($fs1+1 \text{ ## } fs1 = fs2+1 \text{ ## } fs2$)
- NE: no igual ($fs1+1 \text{ ## } fs1 \neq fs2+1 \text{ ## } fs2$)

Instrucciones __F (LT, GT, LE, GE, EQ, NE)

Operación:

$$\begin{aligned} & \text{si } (FPR[fs1] \text{ cond } FPR[fs2]) \\ & \quad FPSR \leftarrow 1 \\ & \text{si no} \\ & \quad FPSR \leftarrow 0 \end{aligned}$$

Sintaxis:

$$\text{__F } fs1, fs2$$

Descripción:

La instrucción `__F` compara los puntos flotantes en simple precisión de los registros `fs1` y `fs2` según la condición LT, GT, LE, GE, EQ, NE. Si la condición es cierta coloca un 1 en el registro de estado especial de punto flotante (FPSR), en otro caso coloca el valor 0. Las siguientes condiciones son posibles:

- LT: menor ($fs1 < fs2$)
- GT: mayor ($fs1 > fs2$)
- LE: menor o igual ($fs1 \leq fs2$)
- GE: mayor o igual ($fs1 \geq fs2$)
- EQ: igual ($fs1 = fs2$)
- NE: no igual ($fs1 \neq fs2$)

7.5. Instrucciones vectoriales

Instrucción ADDV

Operación:

$$\begin{aligned} V[vd(0)] &\leftarrow V[vs1(0)] + V[vs2(0)] \\ V[vd(1)] &\leftarrow V[vs1(1)] + V[vs2(1)] \\ &\dots \\ V[vd(63)] &\leftarrow V[vs1(63)] + V[vs2(63)] \end{aligned}$$

Sintaxis:

`addv vd, vs1, vs2`

Descripción:

La instrucción `ADDV` suma los componentes de los registros vectoriales `vs1` y `vs2` para obtener los componentes correspondientes del registro vectorial `vd`. El número de componentes sobre el que se hará la suma será el especificado en el registro de longitud vectorial (VLR). La suma no se realizará sobre aquellos componentes que no tengan su bit correspondiente a 1 en el vector de máscara (VM).

Instrucción ADDSV

Operación:

$$\begin{aligned} V[vd(0)] &\leftarrow (FPR[fs1 + 1] \text{ ## } FPR[fs1]) + V[vs2(0)] \\ V[vd(1)] &\leftarrow (FPR[fs1 + 1] \text{ ## } FPR[fs1]) + V[vs2(1)] \\ &\dots \\ V[vd(63)] &\leftarrow (FPR[fs1 + 1] \text{ ## } FPR[fs1]) + V[vs2(63)] \end{aligned}$$

Sintaxis:

`addsv vd, fs1, vs2`

Descripción:

La instrucción **ADDSV** suma el punto flotante en doble precisión contenido en la pareja de registros de punto flotante `fs1` y `fs1+1` con cada uno de los componentes del registro vectorial `vs2` para obtener los componentes correspondientes del registro vectorial `vd`. El número de componentes sobre el que se hará la suma será el especificado en el registro de longitud vectorial (VLR). La suma no se realizará sobre aquellos componentes que no tengan su bit correspondiente a 1 en el vector de máscara (VM).

Instrucción SUBV

Operación:

$$\begin{aligned} V[vd(0)] &\leftarrow V[vs1(0)] - V[vs2(0)] \\ V[vd(1)] &\leftarrow V[vs1(1)] - V[vs2(1)] \\ &\dots \\ V[vd(63)] &\leftarrow V[vs1(63)] - V[vs2(63)] \end{aligned}$$

Sintaxis:

`subv vd, vs1, vs2`

Descripción:

La instrucción **SUBV** resta los componentes del registro vectorial `vs2` a los de `vs1` para obtener los componentes correspondientes del registro vectorial `vd`. El número de componentes sobre el que se hará la resta será el especificado en el registro de longitud vectorial (VLR). La resta no se realizará sobre aquellos componentes que no tengan su bit correspondiente a 1 en el vector de máscara (VM).

Instrucción SUBVS

Operación:

$$\begin{aligned} V[vd(0)] &\leftarrow V[vs1(0)] - (FPR[fs2 + 1] \text{ ## } FPR[fs2]) \\ V[vd(1)] &\leftarrow V[vs1(1)] - (FPR[fs2 + 1] \text{ ## } FPR[fs2]) \\ &\dots \\ V[vd(63)] &\leftarrow V[vs1(63)] - (FPR[fs2 + 1] \text{ ## } FPR[fs2]) \end{aligned}$$

Sintaxis:

`subvs vd, vs1, fs2`

Descripción:

La instrucción **SUBVS** resta el punto flotante en doble precisión contenido en la pareja de registros de punto flotante fs2 y fs2+1 a cada uno de los componentes del registro vectorial vs1 al para obtener los componentes correspondientes del registro vectorial vd. El número de componentes sobre el que se hará la resta será el especificado en el registro de longitud vectorial (VLR). La resta no se realizará sobre aquellos componentes que no tengan su bit correspondiente a 1 en el vector de máscara (VM).

Instrucción SUBSV

Operación:

$$\begin{aligned} V[vd(0)] &\leftarrow (FPR[fs1 + 1] \text{ ## } FPR[fs1]) - V[vs2(0)] \\ V[vd(1)] &\leftarrow (FPR[fs1 + 1] \text{ ## } FPR[fs1]) - V[vs2(1)] \\ &\dots \\ V[vd(63)] &\leftarrow (FPR[fs1 + 1] \text{ ## } FPR[fs1]) - V[vs2(63)] \end{aligned}$$

Sintaxis:

`subsv vd, fs1, vs2`

Descripción:

La instrucción **SUBSV** resta cada uno de los componentes del registro vectorial vs2 al punto flotante en doble precisión contenido en la pareja de registros de punto flotante fs1 y fs1+1 para obtener los componentes correspondientes del registro vectorial vd. El número de componentes sobre el que se hará la resta será el especificado en el registro de longitud vectorial (VLR). La resta no se realizará sobre aquellos componentes que no tengan su bit correspondiente a 1 en el vector de máscara (VM).

Instrucción MULTV

Operación:

$$\begin{aligned} V[vd(0)] &\leftarrow V[vs1(0)] \cdot V[vs2(0)] \\ V[vd(1)] &\leftarrow V[vs1(1)] \cdot V[vs2(1)] \\ &\dots \\ V[vd(63)] &\leftarrow V[vs1(63)] \cdot V[vs2(63)] \end{aligned}$$

Sintaxis:

```
multv vd, vs1, vs2
```

Descripción:

La instrucción **MULTV** multiplica los componentes de los registros vectoriales *vs1* y *vs2* para obtener los componentes correspondientes del registro vectorial *vd*. El número de componentes sobre el que se hará la multiplicación será el especificado en el registro de longitud vectorial (VLR). La multiplicación no se realizará sobre aquellos componentes que no tengan su bit correspondiente a 1 en el vector de máscara (VM). Si el resultado de la multiplicación de uno de los componentes no puede representarse como un punto flotante de 64 bits se produce una excepción de *overflow*.

Instrucción MULTSV

Operación:

$$\begin{aligned} V[vd(0)] &\leftarrow (FPR[fs1 + 1] \text{ ## } FPR[fs1]) \cdot V[vs2(0)] \\ V[vd(1)] &\leftarrow (FPR[fs1 + 1] \text{ ## } FPR[fs1]) \cdot V[vs2(1)] \\ &\dots \\ V[vd(63)] &\leftarrow (FPR[fs1 + 1] \text{ ## } FPR[fs1]) \cdot V[vs2(63)] \end{aligned}$$

Sintaxis:

```
multsv vd, fs1, vs2
```

Descripción:

La instrucción **MULTSV** multiplica el punto flotante en doble precisión contenido en la pareja de registros de punto flotante *fs1* y *fs1+1* con cada uno de los componentes del registro vectorial *vs2* para obtener los componentes correspondientes del registro vectorial *vd*. El número de componentes sobre el que se hará la multiplicación será el especificado en el

registro de longitud vectorial (VLR). La multiplicación no se realizará sobre aquellos componentes que no tengan su bit correspondiente a 1 en el vector de máscara (VM). Si el resultado de la multiplicación de uno de los componentes no puede representarse como un punto flotante de 64 bits se produce una excepción de *overflow*.

Instrucción DIVV

Operación:

$$\begin{aligned} V[vd(0)] &\leftarrow V[vs1(0)] \div V[vs2(0)] \\ V[vd(1)] &\leftarrow V[vs1(1)] \div V[vs2(1)] \\ &\dots \\ V[vd(63)] &\leftarrow V[vs1(63)] \div V[vs2(63)] \end{aligned}$$

Sintaxis:

`divv vd, vs1, vs2`

Descripción:

La instrucción **DIVV** divide los componentes del registro vectorial `vs2` entre los componentes de `vs1` para obtener los componentes correspondientes del registro vectorial `vd`. El número de componentes sobre el que se hará la división será el especificado en el registro de longitud vectorial (VLR). La división no se realizará sobre aquellos componentes que no tengan su bit correspondiente a 1 en el vector de máscara (VM). Si el contenido de uno de los componentes de `vs2` es 0.0 provoca una excepción de división por cero.

Instrucción DIVVS

Operación:

$$\begin{aligned} V[vd(0)] &\leftarrow V[vs1(0)] \div (FPR[fs2 + 1] \#\# FPR[fs2]) \\ V[vd(1)] &\leftarrow V[vs1(1)] \div (FPR[fs2 + 1] \#\# FPR[fs2]) \\ &\dots \\ V[vd(63)] &\leftarrow V[vs1(63)] \div (FPR[fs2 + 1] \#\# FPR[fs2]) \end{aligned}$$

Sintaxis:

`divvs vd, vs1, fs2`

Descripción:

La instrucción **DIVVS** divide los componentes del registro vectorial *vs1* entre el punto flotante en doble precisión contenido en la pareja de registros de punto flotante *fs2* y *fs2+1* para obtener los componentes correspondientes del registro vectorial *vd*. El número de componentes sobre el que se hará la división será el especificado en el registro de longitud vectorial (VLR). La división no se realizará sobre aquellos componentes que no tengan su bit correspondiente a 1 en el vector de máscara (VM). Si el contenido de la pareja de registros *fs2* y *fs2+1* es 0.0 provoca una excepción de división por cero.

Instrucción DIVSV

Operación:

$$\begin{aligned} V[vd(0)] &\leftarrow (FPR[fs1 + 1] \text{ ## } FPR[fs1]) \div V[vs2(0)] \\ V[vd(1)] &\leftarrow (FPR[fs1 + 1] \text{ ## } FPR[fs1]) \div V[vs2(1)] \\ &\dots \\ V[vd(63)] &\leftarrow (FPR[fs1 + 1] \text{ ## } FPR[fs1]) \div V[vs2(63)] \end{aligned}$$

Sintaxis:

`divsv vd, fs1, vs2`

Descripción:

La instrucción **DIVSV** divide el punto flotante en doble precisión contenido en la pareja de registros de punto flotante *fs1* y *fs1+1* entre cada uno de los componentes del registro vectorial *vs2* para obtener los componentes correspondientes del registro vectorial *vd*. El número de componentes sobre el que se hará la división será el especificado en el registro de longitud vectorial (VLR). La división no se realizará sobre aquellos componentes que no tengan su bit correspondiente a 1 en el vector de máscara (VM). Si el contenido de uno de los componentes de *vs2* es 0.0 provoca una excepción de división por cero.

Instrucción LV

Operación:

$$\begin{aligned} V[vd(0)] &\leftarrow M[GPR[rs1]] \\ V[vd(1)] &\leftarrow M[GPR[rs1] + 8] \end{aligned}$$

...

$$V[vd(63)] \leftarrow M[GPR[rs1] + 8 * 63]$$

Sintaxis:

lv vd, rs1

Descripción:

La instrucción LV carga el registro vectorial vd desde memoria. Los componentes están en posiciones consecutivas a partir de la dirección contenida en rs1. El número de componentes a cargar será el especificado en el registro de longitud vectorial (VLR). La carga no se realizará sobre aquellos componentes que no tengan su bit correspondiente a 1 en el vector de máscara (VM).

Instrucción SV

Operación:

$$M[GPR[rs1]] \leftarrow V[vd(0)]$$
$$M[GPR[rs1] + 8] \leftarrow V[vd(1)]$$

...

$$M[GPR[rs1] + 8 * 63] \leftarrow V[vd(63)]$$

Sintaxis:

sv rs1, vd

Descripción:

La instrucción SV almacena los componentes del registro vectorial vd en posiciones de memoria consecutivas a partir de la dirección contenida en rs1. El número de componentes que se almacenen será el especificado en el registro de longitud vectorial (VLR). El almacenamiento no se realizará sobre aquellos componentes que no tengan su bit correspondiente a 1 en el vector de máscara (VM).

Instrucción LVWS

Operación:

$$V[vd(0)] \leftarrow M[GPR[rs1]]$$
$$V[vd(1)] \leftarrow M[GPR[rs1] + GPR[rs2]]$$

...

$$V[vd(63)] \leftarrow M[GPR[rs1] + 63 * GPR[rs2]]$$

Sintaxis:

`lvws vd, rs1, rs2`

Descripción:

La instrucción **LVWS** carga el registro vectorial `vd` a partir de la dirección contenida en `rs1` con el desplazamiento contenido en `rs2`. El número de componentes a cargar será el especificado en el registro de longitud vectorial (VLR). La carga no se realizará sobre aquellos componentes que no tengan su bit correspondiente a 1 en el vector de máscara (VM).

Instrucción SVWS

Operación:

$$M[GPR[rs1]] \leftarrow V[vd(0)]$$
$$M[GPR[rs1] + GPR[rs2]] \leftarrow V[vd(1)]$$

...

$$M[GPR[rs1] + 63 * GPR[rs2]] \leftarrow V[vd(63)]$$

Sintaxis:

`svws rs1, rs2, vd`

Descripción:

La instrucción **SVWS** almacena los componentes del registro vectorial `vd` a partir de la dirección contenida en `rs1` con el desplazamiento contenido en `rs2`. El número de componentes que se almacenen será el especificado en el registro de longitud vectorial (VLR). El almacenamiento no se realizará sobre aquellos componentes que no tengan su bit correspondiente a 1 en el vector de máscara (VM).

Instrucción LVI

Operación:

$$V[vd(0)] \leftarrow M[GPR[rs1] + V[vs2(0)]]$$
$$V[vd(1)] \leftarrow M[GPR[rs1] + V[vs2(1)]]$$

...

$$V[vd(63)] \leftarrow M[GPR[rs1] + V[vs2(63)]]$$

Sintaxis:

`lvi vd, rs1, vs2`

Descripción:

La instrucción **LVI** carga el registro vectorial `vd` con componentes en las direcciones `rs1 + componente de vs2`, es decir, `vd[i]` se lee de `rs1+vs2[i]`. El número de componentes a cargar será el especificado en el registro de longitud vectorial (VLR). La carga no se realizará sobre aquellos componentes que no tengan su bit correspondiente a 1 en el vector de máscara (VM).

Instrucción SVI

Operación:

$$\begin{aligned} M[GPR[rs1] + V[vs2(0)]] &\leftarrow V[vd(0)] \\ M[GPR[rs1] + V[vs2(1)]] &\leftarrow V[vd(1)] \\ &\dots \\ M[GPR[rs1] + V[vs2(63)]] &\leftarrow V[vd(63)] \end{aligned}$$

Sintaxis:

`svi rs1, vs2, vd`

Descripción:

La instrucción **SVI** almacena los componentes del registro vectorial `vd` en las direcciones `rs1 + componente de vs2`, es decir, `vd[i]` se escribe en la dirección `rs1+vs2[i]`. El número de componentes que se almacenen será el especificado en el registro de longitud vectorial (VLR). El almacenamiento no se realizará sobre aquellos componentes que no tengan su bit correspondiente a 1 en el vector de máscara (VM).

Instrucción CVI

Operación:

$$\begin{aligned} V[vd(0)] &\leftarrow 0 \\ V[vd(1)] &\leftarrow GPR[rs1] \\ &\dots \\ V[vd(63)] &\leftarrow 63 * GPR[rs1] \end{aligned}$$

Sintaxis:

`cvi vd, rs1`

Descripción:

La instrucción CVI crea un registro vectorial de índices almacenando en cada componente de vd los valores 0, rs1, 2*rs1, 3*rs1, ..., 63*rs1.

Instrucciones S__V (EQ, NE, GT, LT, GE, LE)

Operación:

$$\begin{aligned} \text{si } (V[vs1(i)] \text{ cond } V[vs2(i)]) & \quad 0 \leq i \leq 63 \\ VM(i) & \leftarrow 1 \\ \text{si no} & \\ VM(i) & \leftarrow 0 \end{aligned}$$

Sintaxis:

`S__V vs1, vs2`

Descripción:

La instrucción S__V compara los componentes de vs1 y vs2 según la condición EQ, NE GT, GE, LT o LE y actualiza el bit correspondiente del registro de máscara VM con un 1 si la condición se cumple y con un 0 si la condición no se cumple. Las siguientes condiciones son posibles:

- LT: menor ($vs1 < vs2$)
- GT: mayor ($vs1 > vs2$)
- LE: menor o igual ($vs1 \leq vs2$)
- GE: mayor o igual ($vs1 \geq vs2$)
- EQ: igual ($vs1 = vs2$)
- NE: no igual ($vs1 \neq vs2$)

Instrucciones S__SV (EQ, NE, GT, LT, GE, LE)

Operación:

$$\begin{aligned} & \text{si } ((FSR[fs1 + 1] \text{ ## } FSR[fs1]) \text{ cond } V[vs2(i)]) \\ & \quad VM(i) \leftarrow 1 \\ & \text{si no} \qquad \qquad \qquad 0 \leq i \leq 63 \\ & \quad VM(i) \leftarrow 0 \end{aligned}$$

Sintaxis:

S__SV fs1, vs2

Descripción:

La instrucción S__SV compara el contenido de la pareja de registros de punto flotante fs1 y fs1+1 con cada una de las componentes de vs2 según la condición EQ, NE GT, GE, LT o LE y actualiza el bit correspondiente del registro de máscara VM con un 1 si la condición se cumple y con un 0 si la condición no se cumple. Las siguientes condiciones son posibles:

- LT: menor (fs1+1 ## fs1 < vs2)
- GT: mayor (fs1+1 ## fs1 > vs2)
- LE: menor o igual (fs1+1 ## fs1 ≤ vs2)
- GE: mayor o igual (fs1+1 ## fs1 ≥ vs2)
- EQ: igual (fs1+1 ## fs1 = vs2)
- NE: no igual (fs1+1 ## fs1 ≠ vs2)

Instrucción POP

Operación:

$$GPR[rd] \leftarrow \text{número de unos}[vm]$$

Sintaxis:

pop rd, vm

Descripción:

La instrucción POP cuenta el número de unos del vector máscara VM y pone el resultado en el registro rd.

Instrucción CVM

Operación:

$$vm \leftarrow (1)^{64}$$

Sintaxis:

cvm

Descripción:

La instrucción **CVM** pone a uno todos los componentes del vector máscara VM.

Capítulo 8

Traps

Los traps simulan las llamadas al sistema. La llamada que se realiza depende del número asignado a la misma:

- Trap #0 Termina la ejecución del programa, pero sin terminar las operaciones pendientes.
- Trap #1 Abre un fichero.
- Trap #2 Cierra un fichero.
- Trap #3 Lee de un fichero o de la entrada estándar.
- Trap #4 Escribe en un fichero.
- Trap #5 Envía la salida formateada al dispositivo de salida estándar.
- Trap #6 Termina la ejecución del programa, terminando también las operaciones que queden pendientes en las unidades de proceso.

8.1. Fin de la ejecución del programa

Existen 2 traps que permiten la finalización de la ejecución del programa:

- Trap #0 Termina la ejecución del programa, pero sin terminar las operaciones pendientes.
- Trap #6 Termina la ejecución del programa, terminando también las operaciones que queden pendientes en las unidades de proceso.

8.2. Apertura de un fichero

El Trap #1 permite abrir un fichero para lectura o escritura. Es equivalente a la llamada al sistema `open()` en UNIX/DOS.

Parámetros de entrada

1. Nombre del fichero: es la dirección de la cadena conteniendo el nombre del fichero que se quiere abrir. La cadena debe acabar con el carácter NUL.
2. Modo: establece la forma en que se va a trabajar con el fichero. Algunas constantes que definen los modos básicos son:

0x0000	O_RDONLY	abre en modo lectura.
0x0001	O_WRONLY	abre en modo escritura.
0x0002	O_RDWR	abre en modo lectura-escritura.
0x0008	O_APPEND	abre en modo apéndice (escritura desde el final).
0x0100	O_CREAT	crea el fichero y lo abre (si existía lo machaca).
0x0200	O_TRUNC	abre el fichero y trunca su longitud a cero.
0x0400	O_EXCL	usado con O_CREAT. Si el fichero existe, se retorna un error.
0x4000	O_TEXT	abre el fichero en modo texto.
0x8000	O_BINARY	abre el fichero en modo binario.

Los modos pueden combinarse, simplemente sumando las constantes, o haciendo un “or” lógico, como en este ejemplo:

$$0x0102 : \quad O_CREAT \parallel O_RDWR$$

3. Acceso: sólo se ha de emplear cuando se incluya la opción O_CREAT, y es un entero que define los permisos de acceso al fichero creado.

0x0400	S_IREAD	acceso sólo lectura.
0x0200	S_IWRITE	acceso sólo escritura.

La dirección del primer parámetro debe almacenarse en el registro **R14**. Los siguientes parámetros se situarán en la dirección R14+4 y R14+8.

Parámetros de salida

El trap #1 retorna en el registro **R1** un descriptor válido si el fichero se ha podido abrir, y el valor -1 en caso de error; mostrándose en este caso en la barra de estado del simulador la causa del error.

Ejemplo

```
.data
Filename: .asciiz "C:\ejemplos\data.dat"
          .align 2
Par:      .word Filename
          .word 0x0102 ;modo de acceso
          .word 0x0400 ;permisos de acceso
FileDescr: .space 4

          .text
addi r14,r0,Par ;r14: dirección
                    ;primer parametro
trap 1           ;open()
sw FileDescr(r0),r1 ;r1: descriptor
                    ;del fichero
;...
```

8.3. Cierre de un fichero

El Trap #2 permite cerrar un fichero abierto previamente con el Trap #1. Es equivalente a la llamada al sistema `close()` en UNIX/DOS.

Parámetros de entrada

1. Descriptor: es el descriptor del archivo a cerrar (obtenido previamente al abrir el fichero con el Trap #1).

La dirección del parámetro de entrada debe almacenarse en el registro **R14**.

Parámetros de salida

En caso de que la llamada se realice correctamente, en **R1** se almacena el valor 0, en caso contrario, se almacena -1 indicando que ha habido un error y se mostrará en la barra de estado del simulador la causa del error.

Ejemplo

```
.data
FileDescr: .space 4 ;Descriptor del fichero
              ;a cerrar
              ; (ver ejemplo Trap 1)

.text
addi r14,r0,FileDescr;r14:dirección
                                ;parametro
trap 2                          ;close()
;...                            ;r1=0 o error
```

8.4. Lectura de un fichero o de la entrada estándar

El Trap #3 permite leer información de un fichero o de la entrada estándar. Es equivalente a la llamada al sistema `read()` en UNIX/DOS.

Previamente a leer información de un fichero, se ha de abrir con el Trap #1.

Parámetros de entrada

1. Descriptor: es el descriptor sobre el que se pretende actuar. Será el descriptor obtenido previamente al abrir el fichero con el Trap #1.

Como descriptor se puede utilizar el 0 para leer de la entrada estándar (ventana E/S del simulador WinDLXV).

2. Destino: es un puntero al área de memoria donde se va a efectuar la transferencia.
3. Número: número de bytes que se van a transferir.

La dirección del primer parámetro de entrada debe almacenarse en el registro **R14**. Los siguientes parámetros se situarán en la dirección **R14+4** y **R14+8**.

Parámetros de salida

En caso de que la llamada se realice correctamente, en **R1** se almacena el número de bytes que realmente se han transferido. En caso de error,

retorna -1 y en la barra de estado del simulador se mostrará la causa del error.

Ejemplo

```
ReadBuf:    .data
            .space 80 ;Memoria reservada para
                ;80 bytes
ReadPar:    .word 0      ;Descriptor
            .word ReadBuf ;Dirrección destino
            .word 80      ;Número de bytes

            .text
            addi r14,r0,ReadPar;r14:dirección
                ;primer parámetro
            trap 3        ;read()
            ;...          ;r1: bytes
                ;leídos
```

8.5. Escritura de un fichero

El Trap #4 permite escribir información en un fichero. Es equivalente a la llamada al sistema `write()` en UNIX/DOS.

Previamente a escribir información en un fichero, se ha de abrir con el Trap #1.

Parámetros de entrada

1. Descriptor: es el descriptor sobre el que se pretende actuar. Será el descriptor obtenido previamente al abrir el fichero con el Trap #1.
2. Origen: es un puntero al área de memoria desde la que se va a efectuar la transferencia.
3. Número: número de bytes que se van a transferir.

La dirección del primer parámetro de entrada debe almacenarse en el registro **R14**. Los siguientes parámetros se situarán en la dirección $R14+4$ y $R14+8$.

Parámetros de salida

En caso de que la llamada se realice correctamente, en **R1** se almacena el número de bytes realmente escritos, en caso contrario, se almacena -1 indicando que ha habido un error y se mostrará en la barra de estado del simulador la causa del error.

Ejemplo

```
.data
Buffer:  .space 80 ;Memoria a cargar
          ; (previamente almacenada)
WritePar: .space 4      ;Descriptor open()
          .word Buffer   ;Dirrección buffer
          ;a copiar
          .word 80       ;Número de bytes

.text
addi r14,r0,WritePar;r14: dirección
          ;primer parám.
trap 4    ;write()
;...      ;r1: bytes
          ;escritos
```

8.6. Escritura formateada por la salida estándar

El Trap #5 permite escribir información formateada por la salida estándar. Es equivalente a la función `printf()` en C.

Parámetros de entrada

1. Formato de la cadena: es la dirección de la cadena formateada equivalente a la usada en la función `printf()` de C. Por ejemplo, algunos de los indicadores de tipos admitidos son:
 - `%c`: el argumento es tratado como un entero, y presentado como el carácter con ese valor ASCII.
 - `%d`: el argumento es tratado como un entero, y presentado como un número decimal con signo.
 - `%u`: el argumento es tratado como un entero, y presentado como un número decimal sin signo.

- %f: el argumento es tratado como un flotante, y presentado como un número de punto flotante.
- %g: el argumento es tratado como un flotante en doble precisión, y presentado como un número de punto flotante en doble precisión.
- %s: el argumento es tratado y presentado como una cadena.
- %x: el argumento es tratado como un entero y presentado como un número hexadecimal.

2. Argumentos de acuerdo con el formato definido en 1.

La dirección de la cadena debe almacenarse en el registro **R14**. Los argumentos se situarán en la dirección R14+4, R14+8, ...

Parámetros de salida

En caso de que la llamada se realice correctamente, en **R1** se almacena el número de bytes que realmente se han transferido. En caso de error, retorna -1 y en la barra de estado del simulador se mostrará la causa del error.

Ejemplo

```
.data
FormatStr: .asciiz "Valor = %d\n"
          .align 2

PrintfPar: .word FormatStr ;Dirección cadena
          .word 24        ;Argumentos

.text
addi r14,r0,PrintfPar;r14:dirección
                                ;primer parám.
trap 5                          ;printf()
;...                            ;r1: bytes
                                ;transferidos
```


Bibliografía

- [Hen96] Hennessy, J. L., Patterson, D. A.: *Computer Architecture: A Quantitative Approach*, 2nd edition, , Morgan Kauffman, 1996.
- [Hen02] Hennessy, J. L., Patterson, D. A.: *Computer Architecture: A Quantitative Approach*, 3rd edition, Morgan Kauffman, 2002.
- [Ort05] Ortega, J., Anguita, M., Prieto, A.: *Arquitectura de Computadores*, Wiley-Interscience, 2005.
- [ElR05] El-Rewini, H., Abd-El-Barr, M.: *Advanced computed architecture and parallel processing*, Paraninfo, S.A, 2005.
- [Par02] Parhami, B.: *Introduction to Parallel Processing. Algorithms and Architectures*, Kluwer Academic Publishers, 2002.
- [Kan88] Kane, G.: *MIPS RISC Architecture*, Prentice-Hall, 1988.
- [Bri03] Britton, R: *MIPS Assembly Language Programming*, Prentice-Hall, 2003.