Introducción a la arquitectura MIPS64

Área de Arquitectura y Tecnología de Computadores
Versión 2.0,
28/09/2017

Índice

Objetivos de la sesión

En esta sesión se introduce el simulador WinMIPS64, que se utilizará para analizar la microarquitectura segmentada en 5 etapas de los procesadores MIPS64. Los principales objetivos de esta sesión son:

- Utilizar el simulador WinMIPS64.

- Introducir la programación en ensamblador para MIPS64.

Conocimientos y materiales necesarios

Para poder realizar esta sesión, el alumno debe:

- Revisar la teoría correspondiente al tema dedicado a la CPU. Es importante tener claras las ideas teóricas sobre la arquitectura del juego de instrucciones MIPS64.

- Esta sesión no viene acompañada de cuestionario.

Para llevar a cabo la práctica es necesario disponer del simulador WinMIPS64, que corre en Windows, por lo que necesitamos una máquina con dicho sistema operativo.

1. Introducción al simulador WinMIPS64

- Arranca Linux.
- Cambia al directorio de tu repositorio y sincronízalo con el repositorio de Bitbucket para actualizar los cambios que se hubiesen realizado con anterioridad.

```
$> git pull
```

- Descarga los ficheros necesarios para la práctica y descomprímelos con estas órdenes:

```
$> wget http://rigel.atc.uniovi.es/grado/2ac/files/sesion2-1.tar.gz
```

- `$> tar xvfz sesion2-1.tar.gz`

- Añade los ficheros al índice git, confirma los cambios y súbelos al servidor con estas órdenes:

```
$> git add sesion2-1
```

- `$> git commit`
- `$> git push`

- Cámbiate al directorio `sesion2-1`.
- Copia los ficheros a Windows.

Como has visto en las clases teóricas, MIPS64 es una arquitectura RISC de 64 bits con un juego de instrucciones muy simple. En las prácticas de la asignatura utilizaremos el simulador WinMIPS64 para estudiar el comportamiento del *pipeline*

de estos procesadores en detalle.

El programa WinMIPS64 permite simular la ejecución de programas escritos en ensamblador para MIPS64 y comprobar el funcionamiento del *pipeline*.

- El simulador se ha descargado junto con el resto de ficheros de la práctica.

- Ejecuta el simulador. Te aparecerá una ventana parecida a la de la figura 1.

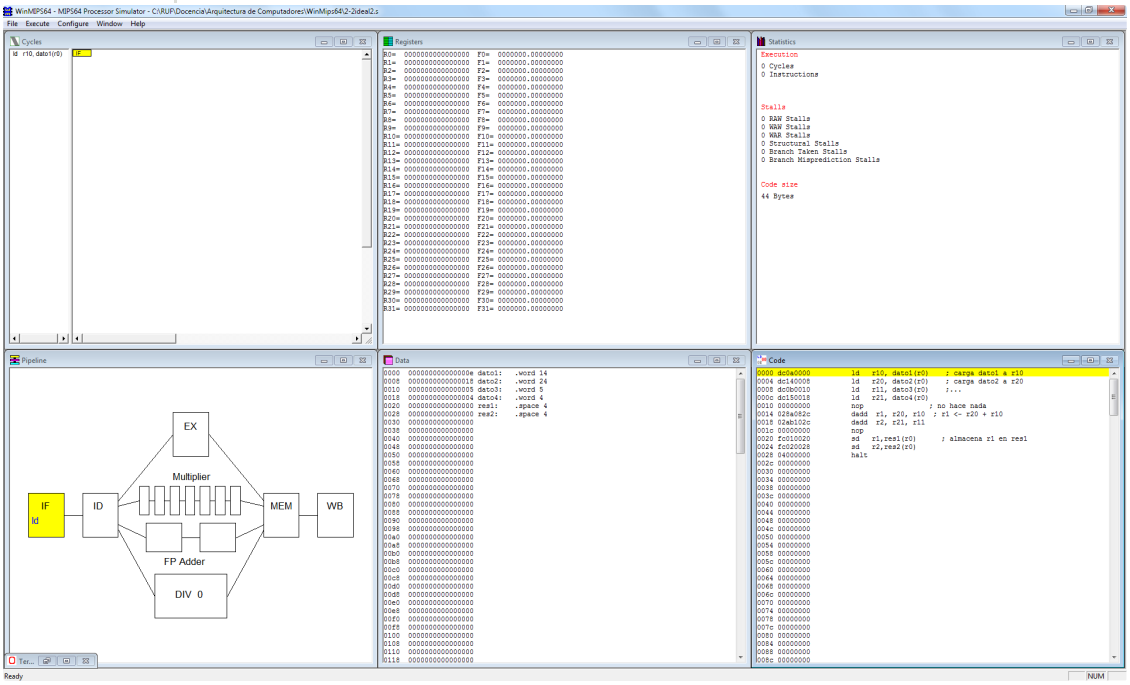

Figura 1. Ventana principal del simulador WinMIPS64

Como ves, aparecen seis ventanas que muestran diferentes vistas del simulador. A continuación se describe brevemente el contenido de cada una:

- **Code** (figura 2): muestra el código del programa cargado. Cada línea muestra la información relativa a una instrucción. La primera columna es el desplazamiento sobre la dirección base donde se carga el código del programa, la segunda es el código de la instrucción en hexadecimal, mientras

que la última columna es la propia instrucción desensamblada.

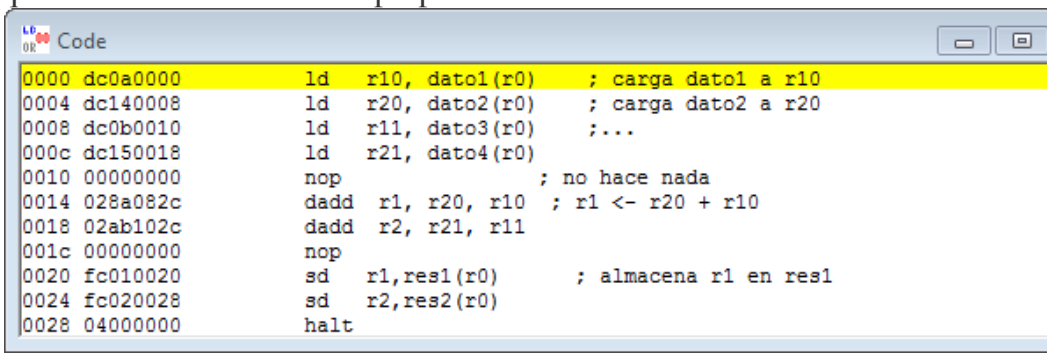

Figura 2. Ventana de código del simulador WinMIPS64

- **Register**: muestra el contenido de los registros del procesador, aunque omite los registros de control. Los registros de propósito general se denominan de `R0` a `R31`; los de punto flotante de precisión doble de `F0` a `F31`.

- **Clock Cycle Diagram** (figura 3): muestra la evolución del *pipeline* de forma esquemática según avanzan los ciclos de reloj. Se utiliza una representación parecida a la vista en las clases de teoría. Las instrucciones a ejecutar se muestran por filas y los ciclos transcurridos por columnas. Cada combinación de fila y columna se etiqueta con la etapa del cauce correspondiente.

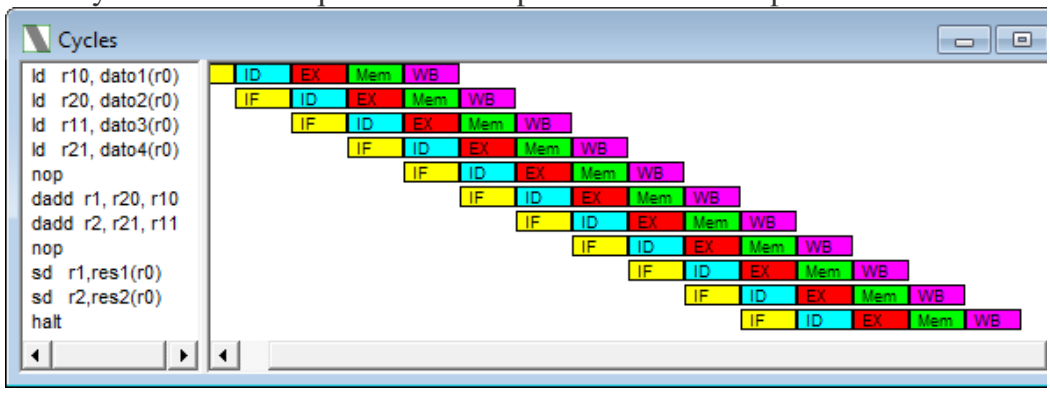

Figura 3. Ventana de ciclos de reloj del simulador WinMIPS64

- **Pipeline** (figura 4): muestra gráficamente la evolución del pipeline durante la

ejecución. En esta vista aparecen representadas las diferentes unidades funcionales del camino de datos.

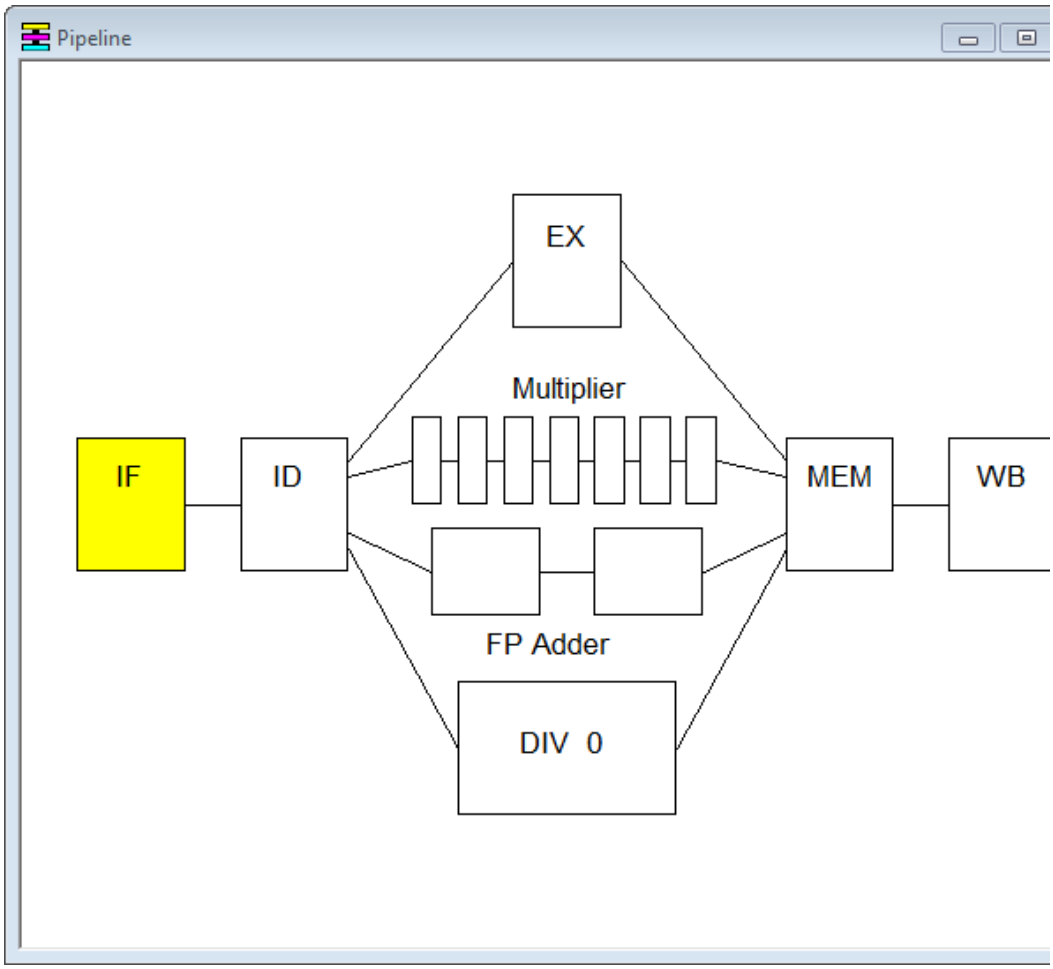

Figura 4. Ventana del pipeline del simulador WinMIPS64

- **Statistics**: muestra estadísticas sobre la ejecución actual.

Toda simulación comienza con la carga en el simulador de un fichero fuente de ensamblador MIPS64. Para ello, se utiliza la opción de menú File ▸ Open. A partir de aquí, el simulador permite la ejecución completa del programa cargado, ciclo a ciclo o con múltiples ciclos. Todas estas opciones se encuentran en el menú menu:Execute.

Existe una séptima ventana, que inicialmente aparece oculta, denominada **Terminal**

sobre la que pueden realizarse operaciones de entrada y salida. Es posible imprimir en la terminal tanto texto como gráficos muy simples y también se puede realizar entrada por el teclado. Para mostrar esta ventana debe seleccionarse la opción de menú Window ▸ Terminal.

1.1. Ensamblador de WinMIPS64

Un programa escrito en el ensamblador del simulador WinMIPS64 tiene dos secciones: una primera sección de datos opcional y una sección de código.

```
    .data
    ; Variables definition

    .code
    ; Program code

    halt ; always at the end of all programs to stop simulation
```

En la sección de datos, que comienza con la directiva `.data`, se declaran las variables que se utilizan en el programa. La declaración de una variable implica la reserva de un espacio en memoria que depende del tamaño del tipo de la variable. Recuerda que la arquitectura MIPS64 puede trabajar con diferentes tipos de datos (byte, media palabra, palabra, doble palabra), si bien el ensamblador añade un tipo adicional: cadena de caracteres. A esta variable se le asigna un nombre simbólico de forma que puede ser utilizada en la sección de código. Como sabes, este nombre simbólico en última instancia se traduce a un desplazamiento en memoria. A continuación se muestran varias definiciones de variables a modo de ejemplo.

```
    .data
    ; Definitions of variables
string:      .ascii  "This is a string" ; 1 byte per character (16
bytes)
anot_string: .asciiz "This is a string" ; 1 byte per char + end of
stream (17 bytes)
double_word: .word 0x1278     ; 8 bytes
word:        .word32 192      ; 4 bytes
half_word:   .word16 90,12    ; 2 bytes + 2 bytes (2 half words)
octet:       .byte 45         ; 1 byte
space:       .space 24        ; allocates 24 bytes in memory
```

La sección de código comienza con la directiva `.code` o `.text` y contiene el código del programa. Se recomienda identar el código de tal forma que se liberen las primeras columnas para definir etiquetas como destinos de las instrucciones de salto. El programa empezará a ejecutarse por la primera instrucción de la sección de código. La última instrucción que se ejecuta será siempre `halt`, que no es una instrucción de MIPS64, sino que es una instrucción específica del simulador y se utiliza para detenerlo.

Como acabas de ver, además de declaraciones de variables, instrucciones y etiquetas, un programa escrito en ensamblador WinMIPS64 contiene directivas. Las directivas no son instrucciones del programa, sino que afectan al comportamiento del simulador. Todas las directivas que soporta el simulador aparecen listadas en la tabla 1.

Tabla 1. Listado de directivas soportadas por WinMIPS64

Directiva	Descripción
`.data`	Indica el comienzo de la sección de datos
`.text / .code`	Indica el comienzo de la sección de código
`.space N`	Reserva N bytes en memoria
`.byte B1,B2,...`	Inicializa una zona de memoria con octectos
`.word16 W1,W2,...`	Inicializa una zona de memoria con medias palabras
`.word32 W1,W2,...`	Inicializa una zona de memoria con palabras
`.word W1,W2,...`	Inicializa una zona de memoria con dobles palabras

`.ascii CADENA`	Inicializa una zona de memoria con una cadena ASCII
`.asciiz CADENA`	Inicializa una zona de memoria con una cadena ASCII y el terminador 0
`.double D1,D2,...`	Inicializa una zona de memoria con reales de precisión doble
`.org DIR`	Indica la dirección de ensamblado de la instrucción o variable que sigue a continuación

Por supuesto, a las regiones de memoria reservadas por las directivas se les pueden indicar etiquetas a utilizar como nombres simbólicos para referirse a ellas, tal como se mostró anteriormente.

Como sabes de las clases teóricas, existen 32 registros enteros de nombre `r0` a `r31` y otros tantos de punto flotante de precisión doble de nombre `f0` a `f31`. Para el caso de los registros enteros, algunos de ellos tienen un cometido especial. Para no complicar en exceso la práctica, consideraremos únicamente el significado específico de los registros `r0` (que vale siempre 0) y `r31` que se utiliza para guardar la dirección de retorno cuando se realizan llamadas a procedimientos.

A continuación se muestra, a modo de ejemplo, un programa que calcula el máximo de una lista de números naturales.

```
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
```

```
;
; Search for the maximum of an array of positive integers
;
    .data

array:  .word 6,9,12,92,100,2,3,1 ; array of numbers
count:  .word 8                   ; item count
max:    .space 8                  ; maximum

    .code

main:
    xor    r8, r8, r8     ; temporary maximum (0 is the minimum)
    xor    r9, r9, r9     ; memory index of the current item
    ld     r10, count(r0) ; remaining items

loop:
    ld     r11, list(r9)   ; current item
    daddi  r10, r10, -1    ; count--
    daddui r9, r9, 8       ; move index to next item (64 bits)
    sltu   r2, r8, r11     ; unsigned comparison
    movn   r8, r11, r2     ; update temporary maximum if greater
    beqz   r10, end        ; end of the array
    j      loop

end:
    sd     r8, max(r0)     ; maximum
    halt
```

- Carga el programa `2-1max.s` en el simulador utilizando la opción de menú File ▸ Open.
- Ejecuta el programa hasta el final pulsando la tecla `F4` o eligiendo la opción de menú Execute ▸ Run to.

Tras la ejecución del programa verás que la ventana **Cycles** ha cambiado para reflejar cómo se ejecutan las instrucciones en el *pipeline* de la CPU. Además, en la ventana **Registers** se muestran los valores finales de los registros de la CPU, mientras que en la ventana **Data** se muestra el contenido de la memoria de datos.

- Fíjate en el valor final de la posición de memoria apuntada por la etiqueta `max`. Comprueba que si se interpreta como un entero de 64 bits este valor coincide con el máximo de la lista.

Otra forma de ejecutar los programas es ciclo a ciclo, lo que requiere conocer el funcionamiento interno de *pipeline* para comprender cómo avanza la ejecución.

La microarquitectura que incorpora el simulador WinMIPS64 define un cauce de ejecución segmentado en 5 etapas:

1. **IF**: búsqueda del código de instrucción e incremento del contador de programa.
2. **ID**: decodificación de la instrucción y lectura de registros.
3. **EX**: ejecución y cálculo de direcciones efectivas.
4. **MEM**: acceso a memoria.
5. **WB**: escritura diferida.

El esquema de ejecución de instrucciones sería el siguiente. Cuando el cauce está lleno se ejecutan 5 instrucciones en paralelo, tantas como etapas:

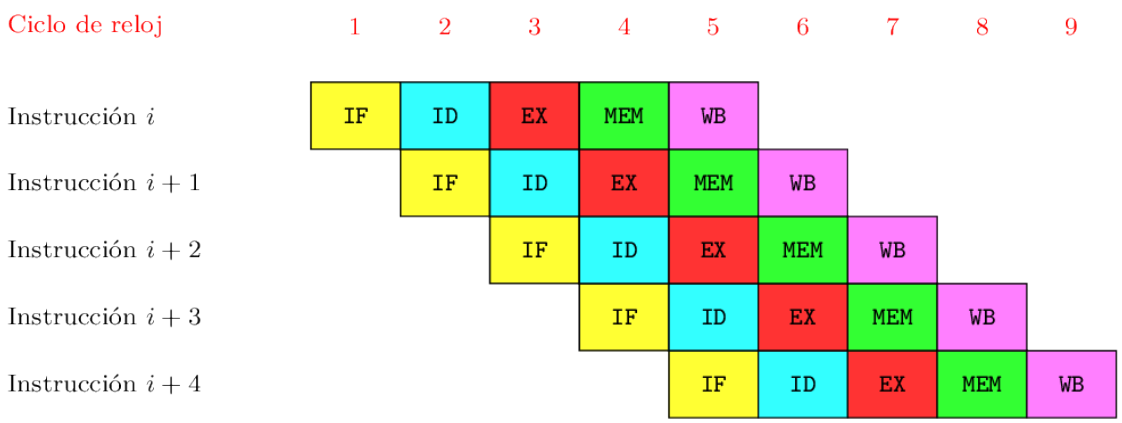

Figura 5. Ejecución segmentada de instrucciones en WinMIPS64

En la siguiente sesión práctica se abordará con más detalle la ejecución de las instrucciones sobre el *pipeline*.

- Reinicia el simulador para volver a ejecutar el programa eligiendo la opción de menú File ▸ Reset MIPS64.
- Ejecuta el programa ciclo a ciclo pulsando la tecla `F7` o eligiendo la opción de menú Execute ▸ Single Cycle. También puedes ejecutar varios ciclos mediante la tecla `F5` o la opción de menú Execute ▸ Multi Cycle. En este último caso puedes configurar el número de ciclos a ejecutar en la opción Configure ▸ Multi-Step.
- Comprueba cómo avanza la ejecución ejecutando ciclo a ciclo.

1.2. Llamadas a procedimientos

Dentro de un programa escrito para WinMIPS64 es posible definir procedimientos para reutilizar código. Para ello, se utiliza el registro `r31` para guardar la dirección de retorno. A continuación se muestra un programa que guarda en la variable `result` el número de dígitos impares que hay en la cadena `string`.

```
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
```

```
;
; Gets the count of odd numbers in a string
;
    .data
string:  .asciiz "12345678"
result:  .word 0

    .text
main:
    xor    r16, r16, r16   ; odd numbers count
    xor    r17, r17, r17   ; index to current item
loop:
    lbu    r4, string(r17) ; load first digit (ascii code)
    beqz   r4, end         ; end of string?
    jal    procedure       ; call procedure
    dadd   r16, r16, r2    ; update odd numbers count
    daddui r17, r17, 1     ; move to next index
    j      loop

end:
    sd     r16, result(r0) ; save result
    halt

procedure:
    ; Receives an ascii code in r4. Returns 1 in r2 if odd, 0 in r2
otherwise
    andi   r2, r4, 1       ; check the least significant bit (1 if odd)
    jr     r31             ; return
```

Fíjate cómo se utiliza la instrucción `jal procedure` para saltar al procedimiento que comienza por la etiqueta `procedure`. Esta instrucción guarda en el registro `r31` la dirección de retorno, que es la dirección de la instrucción siguiente. La instrucción `jr r31` situada al final del procedimiento permite retornar la ejecución a la instrucción siguiente a la de salto.

Es importante tener en cuenta que bajo estas condiciones no se podrían encadenar llamadas a procedimientos, ya que una segunda llamada a procedimiento escribiría de nuevo sobre el registro `r31`, perdiéndose la dirección de retorno del primer procedimiento.

La solución implicaría el manejo de un marco de pila para cada procedimiento donde guardar el valor del registro `r31` al principio del procedimiento para así

poder restaurarlo justo antes de retornar. No obstante, para no complicar en exceso esta práctica, no se abordará la gestión del marco de pila.

Esto implica que los programas a desarrollar **no pueden anidar llamadas a procedimientos**.

1.3. Operaciones de entrada/salida

WinMIPS64 también permite realizar entrada/salida por la consola. Además de para mostrar y leer texto, también son posibles ciertas operaciones gráficas muy simples. Para ello, la consola se divide en dos partes: la consola de texto y la consola gráfica. Esta última se representa como una matriz de píxeles.

La entrada/salida tal como se explica en este apartado es específica del simulador WinMIPS64.

Para realizar operaciones de entrada/salida sobre la consola es necesario utilizar dos registros de la interfaz de la consola mapeados en direcciones de memoria. Los registros a utilizar son los siguientes:

- `CONTROL`. Es un registro de 32 bits mapeado a partir de la dirección `0x10000`. Sirve para señalizar la operación que se realizará al leer/escribir sobre el registro de datos.

- `DATA`. Es un registro de 32 bits mapeado a partir de la dirección `0x10008`. A través de este registro se leen y envían datos de la consola, y se realizan operaciones gráficas simples.

La tabla 2 muestra el significado de asignar diferentes valores al registro de control.

Tabla 2. Operaciones posibles sobre la consola

Operación	**Valor de `CONTROL`**	**Valor de `DATA`**
Imprimir entero sin signo en la consola de texto	1	El entero sin signo a imprimir
Imprimir entero con signo en la consola de texto	2	El entero con signo a imprimir
Imprimir número real en la consola de texto	3	El número real a imprimir
Imprimir cadena de caracteres en la consola de texto	4	La dirección de la cadena a imprimir
Dibujar un píxel en la consola gráfica	5	En `DATA` se escribe el color RGB, en `DATA+5` la coordenada *x* y en `DATA+4` la coordenada *y*.
Borrar la consola de texto	6	—
Borrar la consola gráfica	7	—
Leer entero o número real de la consola de texto	8	El valor leído
Leer un byte de la consola de texto sin eco	9	El valor leído

A continuación se muestra un programa que realiza algunas operaciones de entrada y salida a modo de ejemplo.

```
1
2
```

```
;
; Input/output examples
;
        .data
int:    .word 0xF9876543987625AA    ; a 64-bit integer
```

```
message:  .asciiz "Hello, World!\n"     ; the message
prompt:   .asciiz "Number of pixels [1,49]: "
key:      .asciiz "Press any key to exit\n"

double:   .double 32.786        ; a double
x:        .byte 0               ; coordinates of a point
y:        .byte 0
color:    .byte 255,0,255,0     ; the colour magenta

CONTROL: .word32 0x10000
DATA:    .word32 0x10008

    .text
main:
    ; Print an unsigned int
    ld     r4, int(r0)    ; Integer to print
    jal    printUInt

    ; Print a double
    l.d    f0, double(r0) ; Double to print
    jal    printDouble

    ; Print a string
    daddi  r4, r0, message
    jal    printString

    ; Read an integer
    daddi  r4, r0, prompt
    jal    printString
    jal    readInt

    ; Draw a diagonal on screen (r2 --> number of pixels)
    dadd   r16, r2, r0
    lwu    r6, color(r0)  ; the same color for all pixels
again:
    dadd   r4, r16, r0
    dadd   r5, r16, r0
    jal    drawPixel
    daddi  r16, r16, -1
    bnez   r16, again

    ; Clear the terminal screen
    jal    clearTerminalScreen

    ; Finish
    daddi  r4, r0, key
    jal    printString
    jal    waitForKey
```

```
    ; Clear the graphical screen
    jal     clearTerminalScreen
    jal     clearGraphicalScreen

    halt


printUInt:
    ; Print r4 as integer
    lwu     r24, DATA(r0)     ; r24 = address of DATA register
    lwu     r25, CONTROL(r0) ; r25 = address of CONTROL register

    daddi   r8, r0, 1         ; set 1 for unsigned integer output
    sd      r4, 0(r24)        ; write integer to DATA register
    sd      r8, 0(r25)        ; write to CONTROL register and make it
happen
    jr      r31               ; return

printDouble:
    ; Print f0 as double
    lwu     r24, DATA(r0)     ; r24 = address of DATA register
    lwu     r25, CONTROL(r0) ; r25 = address of CONTROL register

    daddi   r8, r0, 3         ; set 3 for double output
    s.d     f0, 0(r24)        ; write double to DATA register
    sd      r8, 0(r25)        ; write to CONTROL register and make it
happen
    jr      r31               ; return

printString:
    ; Print the string pointed by r4
    lwu     r24, DATA(r0)     ; r24 = address of DATA register
    lwu     r25, CONTROL(r0) ; r25 = address of CONTROL register

    daddi   r8, r0, 4         ; set 4 for string output
    sd      r4, 0(r24)        ; write address of message to DATA register
    sd      r8, 0(r25)        ; write to CONTROL register and make it
happen
    jr      r31               ; return

drawPixel:
    ; Draw pixel (r4,r5) with color r6
    lwu     r24, DATA(r0)     ; r24 = address of DATA register
    lwu     r25, CONTROL(r0) ; r25 = address of CONTROL register

    daddi   r8, r0, 5         ; set 4 for drawing a pixel
```

```
    sb      r4, 5(r24)        ; store x in DATA + 5
    sb      r5, 4(r24)        ; store y in DATA + 4
    sw      r6, 0(r24)        ; store colour in DATA
    sd      r8, 0(r25)        ; write to CONTROL register and make it
happen
    jr      r31               ; return

clearTerminalScreen:
    lwu     r25, CONTROL(r0) ; r25 = address of CONTROL register

    daddi   r8, r0, 6         ; set 6 for clearing the terminal
    sd      r8, 0(r25)        ; write to CONTROL register and make it
happen
    jr      r31

clearGraphicalScreen:
    lwu     r25, CONTROL(r0) ; r25 = address of CONTROL register

    daddi   r8, r0, 7         ; set 7 for clearing the graphical terminal
    sd      r8, 0(r25)        ; write to CONTROL register and make it
happen
    jr      r31

readInt:
    ; Read an integer an return it in r2
    lwu     r24, DATA(r0)     ; r24 = address of DATA register
    lwu     r25, CONTROL(r0) ; r25 = address of CONTROL register

    daddi   r8, r0, 8         ; set 8 for number input
    sd      r8, 0(r25)        ; write to CONTROL register and make it
happen
    ld      r2, 0(r24)        ; read the integer
    jr      r31

waitForKey:
    ; Read a byte without echo an return the ascii code in r2
    lwu     r24, DATA(r0)     ; r24 = address of DATA register
    lwu     r25, CONTROL(r0) ; r25 = address of CONTROL register

    daddi   r8, r0, 9         ; set 9 for byte input with no echo
    sd      r8, 0(r25)        ; write to CONTROL register and make it
happen
    lb      r2, 0(r24)        ; read the integer
    jr      r31
```

2. Programación para el simulador

WinMIPS64

Como se ha visto, las posibilidades que ofrece WinMIPS64 para simular la ejecución de programas es limitada. Es un simulador más orientado a analizar el comportamiento del camino de datos en la ejecución de pequeños trozos de código y no tanto en el desarrollo de grandes programas. No obstante, con el objetivo de familiarizarnos con el lenguaje ensamblador del simulador se planteará el desarrollo de pequeños programas.

- Escribe un programa de nombre `2-1strlength.s` que calcule la longitud de una cadena de texto. Tanto la cadena como el resultado se almacenan en memoria.

- Convierte el código para leer obtener la longitud de la cadena de caracteres en un procedimiento que reciba como parámetro la dirección donde se encuentra la cadena (puntero a la cadena). El procedimiento debe retornar el número de caracteres de la cadena pasada por parámetro.

- Escribe un programa de nombre `2-1compare.s` que lea dos cadenas de texto por la consola y compare si la longitud de ambas cadenas es la misma. A continuación se muestra el esqueleto del programa:

```
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
```

```
    .data
string1:        .space 50    ; first string
string2:        .space 50    ; second string
max_length:     .word  49    ; 49 chars + end of
stream

CONTROL:        .word32 0x10000
DATA:           .word32 0x10008

    .code
    ; Read first string
    daddi   r4, r0, string1    ; the address to
```

```
store the first string
    ld      r5, max_length(r0) ; the maximum number
of chars
    jal     readString

    ; TODO: Read second string

    ; TODO: Get the length of first string

    ; TODO: Get the length of second string

    ; TODO: Print on screen a message if equals/not
equals

    halt


readString:
    ; Arguments:
    ;    r4: the address to store the string
    ;    r5: the maximum number of chars to read
    lwu     r24, DATA(r0)      ; r24 = address of
DATA register
    lwu     r25, CONTROL(r0)   ; r25 = address of
CONTROL register

    ; TODO: loop reading byte without echo until
ascii=13 or maximum number of chars

    ; TODO: Add terminator char (0)

    j       r31
```

Archivos de la práctica

En tu repositorio debes tener los archivos `2-1max.s`, `2-1procedure.s`, `2-1testio.s`, `2-1strlength.s` y `2-1compare.s`. Recuerda sincronizar tu repositorio local con el repositorio en Bitbucket.