Vircon32: Programar en ensamblador

Documento con fecha 2025.04.02 Escrito por Carra

¿Qué es esto?

Este documento es una guía rápida para programar Vircon32 en lenguaje ensamblador. El propósito de esta guía no es enseñar programación general o en ensamblador, sino describir cómo funciona el ensamblador de Vircon32 y servir como una referencia básica de ensamblador para la consola.

¿Quién debería leer esta guía?

Si sólo quieres hacer juegos en Vircon32 no necesitas esta guía: puedes hacer juegos en lenguaje C con menos esfuerzo. Este documento es una guía para programar a bajo nivel. Ya sea para quien quiera más control haciendo juegos directamente en ensamblador o para quien programe en C y necesite incluir secciones ASM para mejorar el rendimiento.

Índice

Este documento se organiza en secciones. Cada una cubre diferentes aspectos de la CPU de Vircon32, la sintaxis de su lenguaje ensamblador y el uso del programa ensamblador.

Introducción

Usar lenguaje ensamblador significa que, en vez de apoyarte en un lenguaje de programación que implemente conceptos abstractos (como funciones o bucles), escribirás tus programas dando instrucciones directas a la propia CPU de Vircon32.

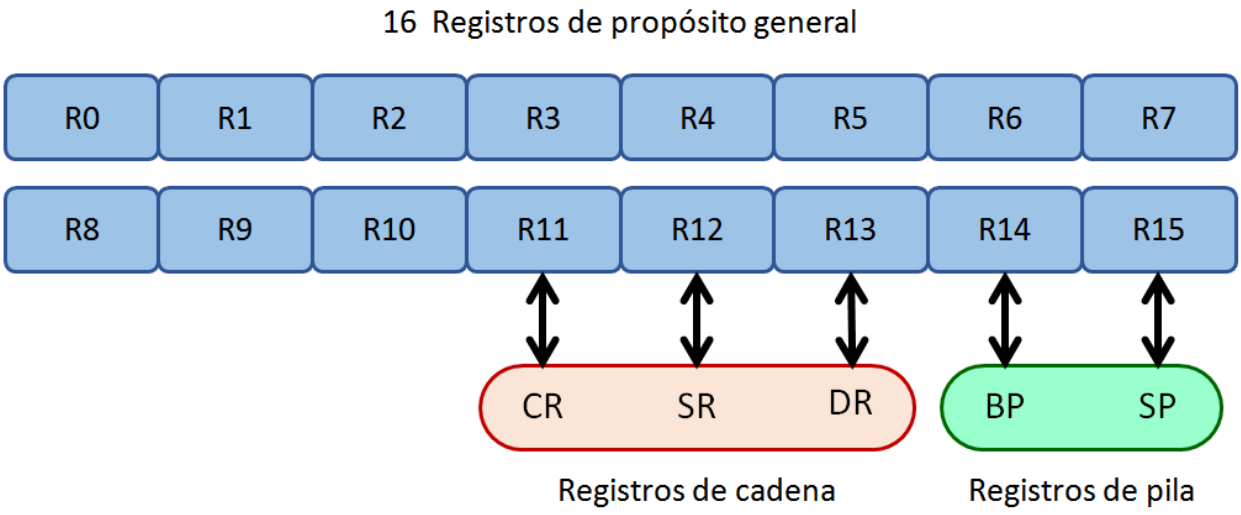
Para ello no basta con conocer el lenguaje ensamblador: necesitarás ciertos conocimientos sobre la CPU y la arquitectura de la consola. Esta sección dará detalles básicos sobre ellos para que puedas empezar, pero si quieres un conocimiento más profundo puedes leer los documentos de especificación parte 2 (arquitectura) y parte 3 (CPU).

La CPU de Vircon32

Vircon32 tiene una CPU de 32 bits simplificada. Su diseño está basado en la familia de procesadores x86, pero tiene sólo 64 instrucciones y con pocas variantes. Y como cada instrucción tarda un sólo ciclo, el rendimiento puede medirse contando instrucciones.

Toda la consola es una máquina de 32 bits pura. Los registros no se pueden usar "parcialmente" para manejar datos de 8 o 16 bits. Ocurre lo mismo con la memoria: cada dirección es una palabra de 32 bits. Todos los tamaños se cuentan en palabras, ya que no se puede acceder a bytes individuales.

La CPU de Vircon32 tiene estos registros accesibles para el programador:



Hay 16 registros de propósito general, de R0 a R15. Cada uno es una única palabra de 32 bits que puede interpretarse con diferentes formatos dependiendo de la instrucción que se ejecute en cada momento. Los últimos 5 de estos registros pueden tener funciones especiales en algunas operaciones de la CPU, y cada uno de ellos recibe un "alias" o nombre alternativo que hace referencia a su papel en esas operaciones.

Aun así, no necesitarás saber esto en detalle hasta que aprendas todas las instrucciones.

Operaciones de pila

La CPU implementa una pila hardware, que se usa para guardar y extraer valores. Su política es sacar primero el último valor guardado. Su función principal es ser usada por las instrucciones de llamada y retorno de subrutinas. Usa los siguientes registros:

R14 = BP (Base Pointer: Puntero base)
R15 = SP (Stack Pointer: Puntero a pila)

El puntero base apunta a la dirección de memoria más alta de la pila (llamada base de la pila), mientras que el puntero a pila registra su posición más baja (lo más alto de la pila).

Operaciones de cadena

Algunas instrucciones de la CPU están diseñadas para operar sobre una serie de direcciones de memoria consecutivas. A veces se llama a estos datos "cadenas", aún si no representan texto. Las operaciones de cadena usan los siguientes registros:

R11 = CR (Count Register: Registro contador)
R12 = SR (Source Register: Registro fuente)
R13 = DR (Destination Register: Registro destino)

Las instrucciones de cadena no se ejecutan sólo una vez: el número de repeticiones lo dicta CR. Y su flujo de datos usará SR y DR como direcciones iniciales.

Sobre el ensamblador de Vircon32

Como la CPU de Vircon32 se basa en procesadores x86, su lenguaje ensamblador también se eligió para ser similar. Instrucciones iguales o similares de Vircon32 tienen el mismo nombre que en x86, como MOV (mover datos) o RET (retorno de subrutina). La sintaxis del ensamblador también es muy similar a la de Intel para x86 (ver sección siguiente).

El lenguaje ensamblador de Vircon32 no distingue mayúsculas y minúsculas. Escribir cualquiera de estas líneas es equivalente: `MOV R0, SP`, `MOV r0, sp` o `mov r0, sp`. Sin embargo el ensamblador de Vircon32 no soporta Unicode. Se recomiendan los archivos de código fuente en ASCII estándar o ASCII extendido con página Latin-1/ISO 8859-1.

Programa de ejemplo

Antes de explicar las características del lenguaje ensamblador de Vircon32, veamos primero un breve ejemplo. Es un programa simple que muestra en pantalla una cuenta atrás: empieza en 5 y cuenta cada segundo. Cuando llega a 0 el programa termina.

Eso es todo: no hay interacción ni sonido y sólo usa la fuente de texto de la BIOS. Aún así debería ser un buen primer ejemplo: usa la mayoría de características del lenguaje, divide su lógica en partes pequeñas, y todos los pasos y decisiones se explican con comentarios.

```
; la ejecución empieza aquí; como nuestro programa principal está más
; más abajo, saltamos a él. Si no, la CPU ejecutará código que no queremos
JMP _program

; ------------ DECLARACIONES ---------------
; definir constantes
%define StartSeconds 5              ; la cuenta atrás empieza en 5
%define FramesPerSecond 60          ; esta console funciona a 60 fps constantes
%define Seconds R0                  ; El registro R0 guardará los segundos actuales

; declarar datos
_digit_characters:
  integer '0','1','2','3','4','5'    ; secuencia de caracteres para la cuenta atrás

; ------------ SUBRUTINAS ---------------
_wait_1_second:
  MOV R1, FramesPerSecond     ; R1 = frames restantes (no usar R0!)
_start_wait_loop:
  WAIT                        ; esperar a que termine el frame
  ISUB R1, 1                  ; ahora queda un frame menos que esperar
  MOV R2, R1                  ; comparar sobreescribe, asi que hacemos una copia
  IGT R2, 0                   ; ¿los frames que quedan (o sea R2) > 0 ? ...
  JT R2, _start_wait_loop     ; ... entonces seguimos en el bucle
  RET                         ; y si no hemos terminado, volver

_print_seconds:
  OUT GPU_Command, GPUCommand_ClearScreen   ; borrar pantalla (por defecto a negro)
  MOV R1, _digit_characters                 ; R1 apunta a '0' (no usar R0!)
  IADD R1, Seconds                          ; R1 apunta al caracter para Seconds
  MOV R1, [R1]                              ; cargar en R1 el valor apuntado
  OUT GPU_SelectedRegion, R1                ; elegir el caracter en textura de BIOS
  OUT GPU_Command, GPUCommand_DrawRegion    ; dibujarlo en el punto de dibj. actual
  RET                                       ; hemos terminado, volver

; ------------ PROGRAMA PRINCIPAL ---------------
_program:
  MOV Seconds, StartSeconds    ; initializar nuestra cuenta de segundos
  OUT GPU_DrawingPointX, 320   ; poner la X de pantalla donde dibujar los números
  OUT GPU_DrawingPointY, 180   ; poner la Y de pantalla donde dibujar los números
_start_main_loop:
  CALL _print_seconds          ; llamar a subrutina para dibujar los segundos
  CALL _wait_1_second          ; llamar a subrutina para esperar 1 segundo
  ISUB Seconds, 1              ; ahora queda 1 segundo menos que esperar
  MOV R1, Seconds              ; comparar sobreescribe, asi que hacemos una copia
  IGE R1, 0                    ; ¿los segundos que quedan (o sea R1) >= 0? ...
  JT R1, _start_main_loop      ; ... entonces seguimos en el bucle
  HLT                          ; y si no fin del programa, la cuenta atrás terminó
```

Podemos distinguir diferentes tipos de elementos en este programa según cómo los interpreta el ensamblador. Aquí estas categorías las hemos resaltado con estos colores:

Comentarios: GRIS	Directivas: MORADO
Etiquetas: AZUL	Definiciones: NEGRO
Declaraciones de datos: **ROSA**	Valores literales: ROJO
Instrucciones: MARRÓN	Registros: **NEGRITA**
Nombres puertos E/S: VERDE	Valores puertos E/S: NARANJA

Comentarios

Los comentarios son una forma de incluir en el programa nuestras propias explicaciones sin que el ensamblador trate de interpretarlas (lo que provocaría errores). Es decir: los comentarios no forman parte del programa en sí.

Los comentarios empiezan con `;` y terminan al final de la línea.

```
; comentario en su propia línea
MOV R2, R1   ; comentario tras una línea de programa
```

Directivas

Las directivas son palabras predefinidas precedidas por `%`. Son comandos especiales para el preprocesador, como definir nombres e incluir archivos externos, que afectan a todo el programa desde ese punto.

```
%ifdef DEBUG
  %include "Aux/DebugFunctions.asm"   ; archivo incluido sólo si activamos DEBUG
%endif
```

El conjunto de directivas soportadas se describe más adelante en su propia sección.

Etiquetas

Las etiquetas son identificadores que empiezan con guión bajo `_`. Al definir etiquetas marcamos puntos concretos del programa, y luego puede usarse la etiqueta como una dirección de memoria en las instrucciones. Lo más común es usarlas en saltos/llamadas a subrutinas y para cargar datos que hayamos declarado.

```
; esto sería un bucle infinito que no hace nada
_start_loop:       ; definir una etiqueta
  WAIT
  JMP _start_loop  ; usar la etiqueta
```

Declaraciones de datos

Estas declaraciones permiten definir distintos tipos de valores literales para que el programa los use. Por ejemplo, puedes querer definir un valor numérico para la gravedad o para la altura de salto de tu personaje. También podemos definir una lista de valores adyacentes del mismo tipo y acceder a ellos después como un array. En cualquier caso todos estos valores serán constantes, ya que se almacenan en la ROM.

```
_bonus_multipliers:  ; usamos una etiqueta para localizar los valores más tarde
  float 1.0, 1.5, 2.0, 3.0, 5.0
```

Las declaraciones de datos soportadas se describen más adelante en su propia sección.

Instrucciones

Una instrucción es una acción mínima que realiza la CPU. Cada una de las acciones de la CPU se identifica con una palabra corta, como `MOV` para mover datos. Después del identificador algunas instrucciones pueden usar hasta 2 operandos separados por comas.

```
MOV R2, 18    ; la instrucción MOV usa 2 operandos
HLT           ; la instrucción HLT no usa operandos
```

El juego de instrucciones de la CPU de Vircon32 se describirá en su propia sección.

Registros

Cada registro de la CPU almacena 1 palabra que la CPU puede usar para operar con su valor. Su contenido puede ser interpretado con diferentes formatos dependiendo del contexto (entero, float, booleano...). Se detallarán más adelante.

El conjunto de registros disponibles ya se definió en la sección de introducción.

```
IADD CR, R0   ; se suma el registro 0 al registro contador (o sea, registro 11)
```

Nombres de puertos E/S

La CPU se comunica con otros chips a través de un conjunto de puertos de entrada/salida que expone cada uno de ellos. Usando las instrucciones IN y OUT la CPU puede leer y escribir valores en un puerto para conseguir diferentes efectos. Los programas en ensamblador se refieren a estos puertos por sus nombres.

```
OUT GPU_DrawingScaleX, 2.0   ; el puerto GPU para escalado en X se pone a zoom 2x
```

La lista de puertos E/S para cada chip de la consola se describirá en su propia sección.

Valores de puertos E/S

La mayoría de puertos de E/S almacenan datos en formatos numéricos, pero algunos sólo pueden procesar un pequeño conjunto de valores específicos. Por ejemplo, el puerto GPU_Command sólo admite valores que la GPU pueda identificar como peticiones de comando:

```
OUT GPU_Command, GPUCommand_DrawRegion  ; la GPU dibujará la región seleccionada
```

La lista de valores para cada puerto E/S concreto se describirá en la misma sección que los nombres de los puertos.

Estructura de un programa

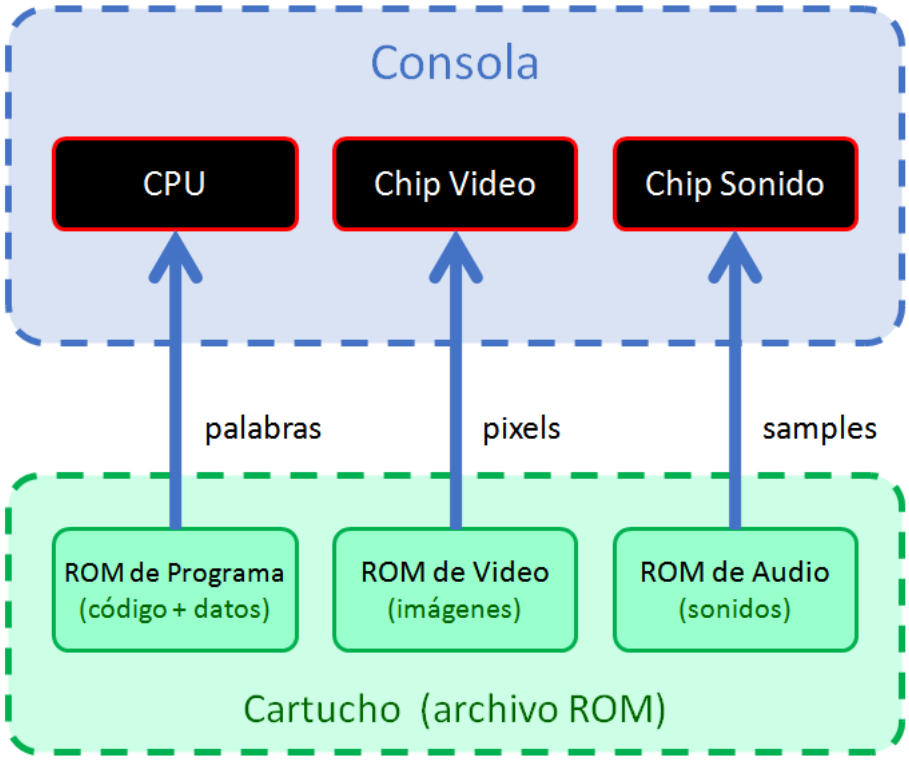
Como habrás visto en el ejemplo, un programa en ensamblador de Vircon32 contiene tanto instrucciones del programa como otros datos no ejecutables que el programa puede necesitar en algún momento. Los datos pueden incluir valores numéricos, cadenas o incluso archivos incrustados en el binario final (ver declaraciones datafile más adelante).

Toda esa información se unirá en una secuencia de palabras. Como este ensamblador (a diferencia de otros) no usa "segmentos" para separar programa y datos, depende del programador asegurarse de que ningún dato se ejecute incorrectamente como instrucción.

Esto se ve en el programa de ejemplo: el programa empieza a ejecutarse en la primera palabra, por lo que el programa debe contener una instrucción antes de declarar datos.

Estructura de una ROM de Vircon32

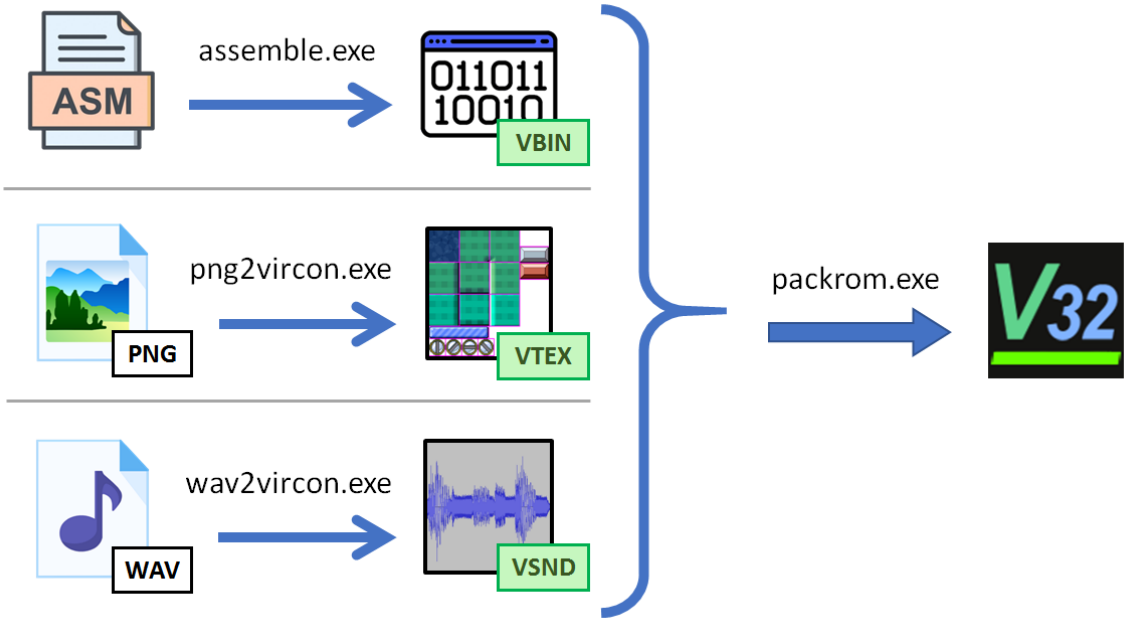
Cuando "ensamblamos" nuestro programa convertimos nuestro código fuente en lenguaje ensamblador en una secuencia de palabras o datos binarios para la CPU. Pero esto aún no es un programa completo de Vircon32. Una ROM de Vircon32 se compone de 3 partes:



Normalmente un programa incluye un conjunto de imágenes y sonidos en el cartucho además del binario del programa. El programa de ejemplo no necesita incluir nada, porque sólo usa la fuente de texto de la BIOS. Pero aún así hay que "empaquetar" el programa en un formato ROM e indicar que no tiene ROM de audio ni ROM de vídeo.

El proceso de creación

El proceso completo para pasar de un código fuente en ensamblador a una ROM ejecutable de Vircon32 tendrá los pasos que se muestran en este diagrama:



Como esperaríamos, para nuestro programa llamaremos al ensamblador que convertirá el código fuente en una ROM de programa. Del mismo modo, si queremos usar imágenes y sonidos en el programa, las herramientas de desarrollo de Vircon32 también incluyen programas para importar imágenes PNG y sonidos WAV en nuestros proyectos. Todos estos programas son herramientas de línea de comandos, por lo que podemos automatizar fácilmente el proceso escribiendo un pequeño script.

Por último, otra herramienta empaquetará esos 3 componentes (o menos: las ROMs de vídeo y audio son opcionales) en la ROM ejecutable final. Para ello tendremos que decirle al empaquetador la ruta de todos los activos que debe incluir. Eso se hace con un XML de definición de ROM. Este es un ejemplo simple de un programa con 2 texturas y 1 sonido:

```
<rom-definition version="1.0">
    <rom type="cartridge" title="Game Test" version="1.0" />
    <binary path="converted/TestProgram.vbin" />
    <textures>
        <texture path="converted/TextureBackground.vtex" />
        <texture path="converted/TextureCharacter.vtex" />
    </textures>
    <sounds>
        <sound path="converted/SoundJump.vsnd" />
    </sound>
</rom-definition>
```

Ten en cuenta que debes indicar las rutas de los activos ya convertidos. Si tu programa incluye una imagen PNG, el XML no debe contener la ruta del propio PNG, sino la ruta del archivo VTEX creado por el conversor PNG.

Uso del ensamblador

El ensamblador de Vircon32 es un programa de línea de comandos, por lo que se usa en una terminal. Seguramente querrás añadir tu carpeta de herramientas de desarrollo a tus variables de entorno para que el comando `assemble` esté disponible en cualquier ruta.

Estas son las opciones que acepta el ensamblador, mostradas usando `assemble --help`:

```
USAGE: assemble [options] file
Options:
  --help       Displays this information
  --version    Displays program version
  -o <file>    Output file, default name is the same as input
  -b           Assembles the code as a BIOS
  -v           Displays additional information (verbose)
Also, the following options are accepted for compatibility
but have no effect: -s
```

La forma más básica de usarlo es: `assemble program.asm`. Si no indicamos archivo de salida con la opción `-o` usará el mismo nombre y (si tiene éxito) creará "program.vbin".

Tipos de datos en la consola

Como dijimos antes, aunque todos los datos de la consola son siempre palabras de 32 bits, la consola interpreta cada palabra con diferentes tipos de datos. Esta sección describe los posibles formatos e interpretaciones de una palabra:

Entero

La palabra se interpreta como un entero con signo de 32 bits. Esto equivale al tipo de datos "int32_t" de C. Ten en cuenta que éste es el único tipo de datos entero que hay: no existen enteros sin signo ni variantes para distintos tamaños.

Booleano

La palabra también se toma como un número entero, pero ese número se interpreta como un valor booleano: Si el valor es 0 la palabra se toma como falso, y cualquier otro valor se interpretará como verdadero. Este tratamiento es equivalente al tipo de datos de C "bool".

Cuando genera un booleano, Vircon32 siempre usa los valores 1 para “verdadero” y 0 para “falso”. Esos son los valores que tienen `true` y `false` en lenguaje ensamblador.

Float

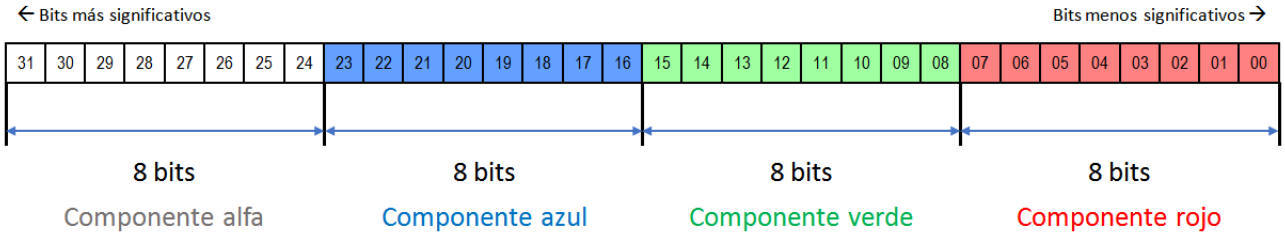
La palabra se interpreta como número de coma flotante de 32 bits. Esto equivale al tipo de datos “float” de C. Pero Vircon32 no soporta valores especiales de float, como NaN (not a number) o infinitos. Usarlos en un programa producirá comportamiento indeterminado.

Binario

Se considera que la palabra es sólo una secuencia de 32 bits, independientes entre sí y sin significado como grupo. Esta interpretación sólo suele usarse en operaciones binarias como AND/OR. Esto equivale a la forma en que C trata los enteros en tales operaciones.

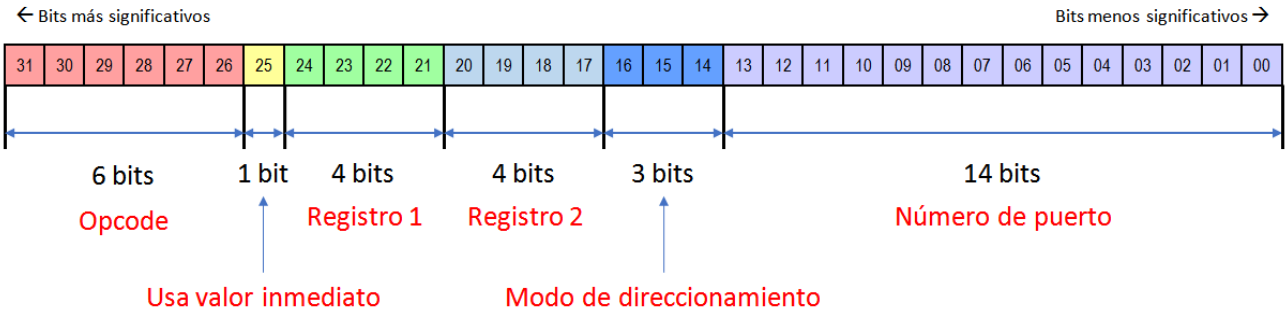
Color de GPU

Cuando la GPU use una palabra como color, la interpretará como RGBA con 4 campos de 8 bits en ese orden. Esto equivale al formato RGBA8 usado en muchas tarjetas gráficas.



Instrucción de CPU

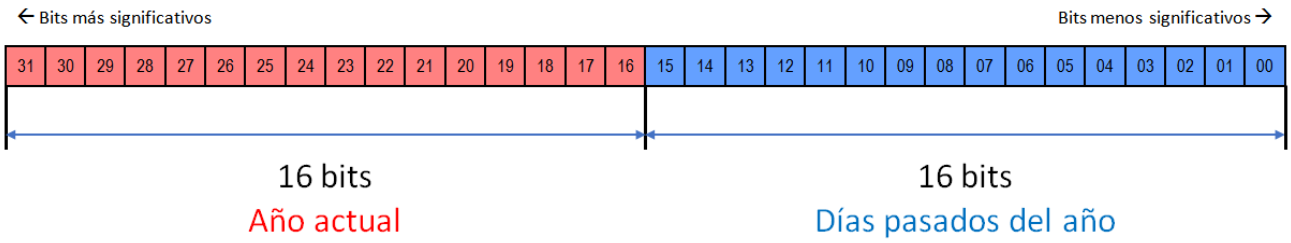
Cuando la CPU ejecuta una palabra como una instrucción la interpreta como un código de instrucción con una serie de campos complementarios, como se ve aquí:



Para programar en ensamblador no necesitas conocer este formato. Basta con saber que la propia instrucción ya incluye los registros usados y los números de puertos de E/S.

Fecha

Es el formato de fecha interno que usa el temporizador de la consola. Agrupa 2 campos, cada uno codificado como un entero sin signo de 16 bits, de esta forma:



Los 16 bits superiores son simplemente el número del año en curso (por ejemplo: 2022). Los 16 bits inferiores indican el número de días ya pasados de ese año. Por ejemplo: si los bits inferiores contienen 3, ahora será 4 de enero (ya pasaron 3 días de este año).

Declarando datos

El componente principal de un programa son las instrucciones, pero éstas a menudo necesitan trabajar con datos. Puedes declarar valores de datos en diferentes formatos con las palabras clave que se presentan aquí. En la práctica todos esos valores literales serán constantes, ya que se guardarán en un cartucho con memoria de sólo lectura.

Declarar floats

Usa la palabra clave `float` para declarar uno o más valores en coma flotante. Ten en cuenta que el ensamblador no soporta la notación exponencial de float, como 0.123E-4. Si declaras floats puedes usar valores enteros y se convertirán a float. Pero ten cuidado: esto no siempre es así en otras partes de los programas.

```
_pi_multiples:  ; usa etiquetas para acceder luego a tus datos
  float 3.1416, 6.2832, 9.4248

_scale_factor:
  float 2       ; se convertirá correctamente al float 2.0
```

Declarar enteros

Con la palabra clave `integer` puedes declarar uno o más valores enteros. Como en Vircon32 sólo existe un tipo entero, también se usan enteros para representar valores no numéricos, como booleanos (true = 1, false = 0) y caracteres (por ejemplo, la letra 'A'). Por lo tanto son válidas todas las siguientes declaraciones de enteros:

```
integer 3, -17          ; valores enteros normales
integer 0xFFA02         ; formato hexadecimal
integer true, false     ; los valores booleanos son enteros
integer 'H', '$'        ; los caracteres también son enteros
```

Los caracteres pueden escaparse cuando sea necesario, con la sintaxis del lenguaje C. Pero observa que los números de caracteres sólo admiten el rango ASCII (de 00h a FFh):

```
integer '\n', '\\', '\'' ; nueva línea, barra inv. y comilla simple
integer '\x4B'           ; carácter número 0x4B = 'K'
```

Declarar cadenas

Con la palabra clave `string` puedes declarar una única cadena de texto. No se admiten varias en una misma declaración. Los caracteres de de una cadena se pueden escapar de la misma forma que al declarar caracteres y, de nuevo, igual que en el lenguaje C.

```
string "Hello \"cruel\" world"  ; hay que escapar comillas dobles
string "Greetings"              ; la segunda cadena necesita otra línea
```

Declarar direcciones de etiquetas

Supongamos que quieres declarar una dirección de memoria como valor para usarla más tarde. Si sabes de antemano la dirección concreta (por ejemplo, el inicio de la tarjeta de memoria), la puedes declarar como un entero normal. Pero el ensamblador también te permite usar etiquetas para declarar direcciones que se resolverán al ensamblar. Esto se hace con la palabra clave `pointer`, ya que es el uso más común para estos valores.

```
; con esto podemos tener punteros a los valores
; float que declaramos en la página anterior
pointer _pi_multiples, _scale_factor
```

Incrustar archivos como datos

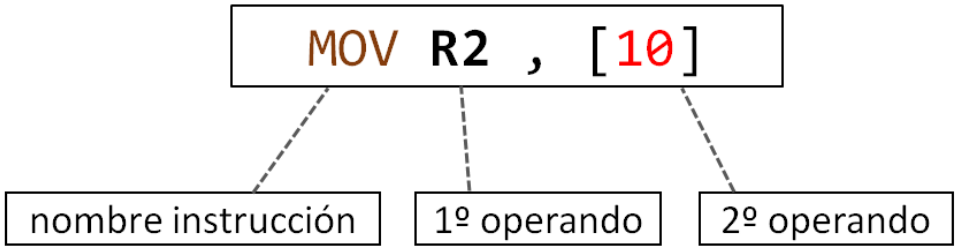
Si nuestro programa necesita datos de gran tamaño, como un mapa de tiles, es mejor guardarlos en un archivo externo y dejar que el ensamblador los incruste directamente en el binario del programa. El ensamblador soporta esto con la palabra clave `datafile`:

```
; el ensamblador accede a este archivo e incrusta su contenido
datafile "Data/Level1.map"
```

La ruta al archivo se da como cadena, que puede tener caracteres escapados igual que las que declaramos. Ten en cuenta que, para que pueda incrustarse, el tamaño del archivo en bytes deberá ser múltiplo de 4. Si no, no cabrá correctamente en palabras de 32 bits.

Instrucciones de la CPU

Los programas en ensamblador se componen de secuencias de instrucciones. Cada instrucción es una acción básica que puede realizar la CPU. Algunas instrucciones son autocontenidas (como HLT para detener la CPU), mientras que otras requieren hasta 2 operandos para definir la acción. Un ejemplo completo podría ser el siguiente:



Esta instrucción, por ejemplo, tomará el contenido de la dirección de memoria 10 (los corchetes indican una posición de memoria) y lo copiará en el registro R2. Cada tipo de instrucción usa siempre el mismo número de operandos, aunque pueden tener variantes que usen operandos de distinto tipo.

En los programas en ensamblador cada instrucción debe estar en una línea separada. Se pueden dejar líneas vacías entre ellas pero no es posible concatenar varias instrucciones en una misma línea de forma parecida a como lenguajes tipo C permiten concatenar diferentes sentencias.

Esta sección dará una breve descripción de todas las instrucciones de la CPU de Vircon32, agrupadas en secciones, junto con ejemplos de todas las variantes que permite cada instrucción.

Control de la CPU	
Instrucción	Descripción breve
HLT Halt	Detiene la CPU, parando el programa. Esto no borra la pantalla ni detiene los sonidos en reproducción. `HLT  ; sin operandos`
WAIT	La CPU espera hasta el siguiente frame. `WAIT  ; sin operandos`

Instrucciones de salto

Instrucción	Descripción breve

JMP
Jump
	La ejecución salta a la dirección indicada.
	`JMP 0x20000015  ; salta a esta dirección de memoria`
	`JMP _mylabel  ; salta a esta etiqueta`
	`JMP R1  ; salta a dirección = valor de R1`

CALL	Llama a la subrutina en la dirección indicada. Al volver de la subrutina la ejecución continuará tras la instrucción call.
	`CALL 0x20000015  ; llama a subr. en esta dir. de memoria`
	`CALL _mysub  ; llama a subrutina en esta etiqueta`
	`CALL R1  ; llama a subr. en direc. = valor de R1`

RET Return	Retorna de la subrutina actual. La ejecución continúa tras la instrucción que la llamó. `RET  ; sin operandos`

JT
Jump if True
	Si su primer operando es verdadero (no nulo) saltará a la dirección de memoria que indica el segundo operando.
	`JT R1, 0x20000015  ; si R1 != 0 salta a esta dirección`
	`JT R1, _mylabel  ; if R1 != 0 salta a esta etiqueta`
	`JT R1, R2  ; if R1 != 0 salta al valor de R2`

JF
Jump if False
	Si su primer operando es falso (nulo) saltará a la dirección de memoria que indica el segundo operando.
	`JF R1, 0x20000015  ; si R1 == 0 salta a esta dirección`
	`JF R1, _mylabel  ; if R1 == 0 salta a esta etiqueta`
	`JF R1, R2  ; if R1 == 0 salta al valor de R2`

Comparación de enteros

Instrucción	Descripción breve

IEQ
Integer Equal
	Comprueba si sus 2 operandos, interpretados como enteros, son iguales. Guarda el resultado booleano en el primer operando.
	`IEQ R0, -19  ; R0 = (R0 == -19)?`
	`IEQ R0, R1  ; R0 = (R0 == R1)?`

INE
Integer
Not Equal
	Comprueba si sus 2 operandos, interpretados como enteros, son distintos. Guarda el resultado booleano en el primer operando.
	`INE R0, -19  ; R0 = (R0 != -19)?`
	`INE R0, R1  ; R0 = (R0 != R1)?`

IGT
Integer
Greater Than
	Comprueba si su primer operando es mayor que el segundo, tomados como enteros. Guarda el resultado booleano en el primer operando.
	`IGT R0, -19 ; R0 = (R0 > -19)?`
	`IGT R0, R1 ; R0 = (R0 > R1)?`

IGE
Integer
Greater or Equal
	Comprueba si su primer operando es mayor o igual que el segundo, tomados como enteros. Guarda el resultado booleano en el primero.
	`IGE R0, -19 ; R0 = (R0 >= -19)?`
	`IGE R0, R1 ; R0 = (R0 >= R1)?`

ILT
Integer
Less Than
	Comprueba si su primer operando es menor que el segundo, tomados como enteros. Guarda el resultado booleano en el primer operando.
	`ILT R0, -19 ; R0 = (R0 < -19)?`
	`ILT R0, R1 ; R0 = (R0 < R1)?`

ILE
Integer
Less or Equal
	Comprueba si su primer operando es menor o igual que el segundo, tomados como enteros. Guarda el resultado booleano en el primero.
	`ILE R0, -19 ; R0 = (R0 <= -19)?`
	`ILE R0, R1 ; R0 = (R0 <= R1)?`

Comparación de floats	
Instrucción	Descripción breve

FEQ
Float Equal
	Comprueba si sus 2 operandos, interpretados como floats, son iguales. Guarda el resultado booleano en el primer operando.
	`FEQ R0, 19.0 ; R0 = (R0 == 19.0)?`
	`FEQ R0, R1 ; R0 = (R0 == R1)?`

FNE
Float
Not Equal
	Comprueba si sus 2 operandos, interpretados como floats, son distintos. Guarda el resultado booleano en el primer operando.
	`FNE R0, 19.0 ; R0 = (R0 != 19.0)?`
	`FNE R0, R1 ; R0 = (R0 != R1)?`

FGT
Float
Greater Than
	Comprueba si su primer operando es mayor que el segundo, tomados como floats. Guarda el resultado booleano en el primer operando.
	`FGT R0, 19.0 ; R0 = (R0 > 19.0)?`
	`FGT R0, R1 ; R0 = (R0 > R1)?`

FGE
Float
Greater or Equal
	Comprueba si su primer operando es mayor o igual que el segundo, tomados como floats. Guarda el resultado booleano en el primero.
	`FGE R0, 19.0 ; R0 = (R0 >= 19.0)?`
	`FGE R0, R1 ; R0 = (R0 >= R1)?`

FLT
Float
Less Than
	Comprueba si su primer operando es menor que el segundo, tomados como floats. Guarda el resultado booleano en el primer operando.
	`FLT R0, 19.0 ; R0 = (R0 < 19.0)?`
	`FLT R0, R1 ; R0 = (R0 < R1)?`

FLE
Float
Less or Equal
	Comprueba si su primer operando es menor o igual que el segundo, tomados como floats. Guarda el resultado booleano en el primero.
	`FLE R0, 19.0 ; R0 = (R0 <= 19.0)?`
	`FLE R0, R1 ; R0 = (R0 <= R1)?`

Movimiento de datos	
Instrucción	Descripción breve

MOV
Move
	Mueve (copia) el valor de su segundo operando al primero.
	`MOV R0, 75 ; R0 = 75`
	`MOV R0, _label ; R0 = dirección de etiqueta`
	`MOV R0, [75] ; R0 = Memoria[ 75 ]`
	`MOV R0, [_label] ; R0 = Memoria[ direc. de etiqueta ]`
	`MOV R0, R1 ; R0 = R1`
	`MOV R0, [R1] ; R0 = Memoria[ R1 ]`
	`MOV R0, [R1+14] ; R0 = Memoria[ R1 + 14 ]`
	`MOV [32], R1 ; Memoria[ 32 ] = R1`
	`MOV [R0], R1 ; Memoria[ R0 ] = R1`
	`MOV [R0+14], R1 ; Memoria[ R0 + 14 ] = R1`

LEA
Load Effective Address
	Recibe como segundo operando una dirección de memoria, y copia esa posición (¡no su contenido!) en el primer operando.
	`LEA R0, [R1] ; R0 = R1`
	`LEA R0, [R1-19] ; R0 = R1 - 19`

PUSH	Guarda el contenido de un registro en lo más alto de la pila de la CPU. `PUSH R0 ; Pila.Guardar( -19 )`
POP	Toma el valor en lo alto de la pila CPU y lo guarda en un registro. `POP R0 ; R0 = Pila.Sacar()`
IN	Lee el valor de un puerto de control E/S y lo guarda en un registro. `IN R1, INP_GamepadLeft ; R1 = Estado izquierda cruceta`
OUT	Escribe el valor de un registro en un puerto de control E/S. `OUT GPU_SelectedRegion, R0 ; Región seleccionada = R0`

Operaciones con cadenas

Instrucción	Descripción breve
MOVS Move String	Implementa un bucle, similar a una versión hardware de memcpy(), para seguir copiando valores entre posiciones de mem. consecutivas: • Memoria[DR] = Memoria[SR] • Se incrementan SR (registro fuente) and DR (registro destino) • Se decrementa CR (registro contador) • Esta instrucción seguirá repetiéndose mientras que CR > 0 `MOVS  ; sin operandos`
SETS Set String	Implementa un bucle, similar a una versión hardware de memset(), para seguir escribiendo un mismo valor en posiciones de mem. consecutivas: • Memoria[DR] = SR • Se incrementa DR (registro destino) • Se decrementa CR (registro contador) • Esta instrucción seguirá repetiéndose mientras que CR > 0 `SETS  ; sin operandos`
CMPS Compare String	Implementa un bucle, similar a una versión hardware de memcmp(), para seguir comparando valores entre posiciones de mem. consecutivas. Guarda el resultado de la comparación en el registro operando: • Registro operando = Memoria[DR] – Memoria[SR] • Si registro operando != 0 se termina el bucle • Se incrementan SR (registro fuente) and DR (registro destino) • Se decrementa CR (registro contador) • Esta instrucción seguirá repetiéndose mientras que CR > 0 `CMPS R1 ; R1 = result. de comparar último par de palabras`

Conversión de datos

Instrucción	Descripción breve
CIF Convert Integer to Float	Realiza una conversión del valor de un registro: lee su contenido como entero y convierte ese valor a un formato float. `CIF R0    ; R0 = (float) R0`
CFI Convert Float to Integer	Realiza una conversión del valor de un registro: lee su contenido como float y convierte ese valor a un formato entero. `CFI R0    ; R0 = (int) R0`

CIB Convert Integer to Boolean	Realiza una conversión del valor de un registro: lee su contenido como entero y convierte ese valor a un formato booleano (o bien 0 o 1). `CIB R0  ; si R0 != 0 entonces R0 = 1`
CFB Convert Float to Boolean	Realiza una conversión del valor de un registro: lee su contenido como float y convierte ese valor a un formato booleano (o bien 0 o 1). `CFB R0  ; si R0 != 0.0 entonces R0 = 1 y si no R0 = 0`

Operaciones binarias	
Instrucción	Descripción breve
NOT	Realiza un NOT binario en un registro invirtiendo todos sus bits. `NOT R0    ; R0 = NOT R0`

AND	Realiza un AND binario (bit por bit) entre sus operandos. Guarda el resultado en su primer operando.
	`AND R0, 35   ; R0 = R0 AND 35`
	`AND R0, R1   ; R0 = R0 AND R1`

OR	Realiza un OR binario (bit por bit) entre sus operandos. Guarda el resultado en su primer operando.
	`OR R0, 35    ; R0 = R0 OR 35`
	`OR R0, R1    ; R0 = R0 OR R1`

XOR
Exclusive OR
	Realiza un OR exclusivo binario (bit por bit) entre sus operandos. Guarda el resultado en su primer operando.
	`XOR R0, 35   ; R0 = R0 XOR 35`
	`XOR R0, R1   ; R0 = R0 XOR R1`

BNOT Boolean NOT	Realiza un NOT booleano en un registro, interpretándolo como un booleano y negándolo. `BNOT R0     ; si R0 == 0 entonces R0 = 1 y si no R0 = 0`

SHL
Bit Shift Left
	Desplaza a la izquierda los bits de su 1º operando. El 2º operando es el número de posiciones a desplazar (si es negativo desplaza a la derecha).
	`SHL R0, 3     ; R0 = R0 << 3`
	`SHL R0, R1    ; R0 = R0 << R1`

Aritmética con enteros

Instrucción	Descripción breve

IADD
Integer Add
	Interpreta ambos operandos como enteros y los suma. Guarda el resultado en su primer operando.
	`IADD R0, 27  ; R0 = R0 + 27`
	`IADD R0, R1  ; R0 = R0 + R1`

ISUB
Integer Subtract
	Interpreta ambos operandos como enteros y resta el segundo al primero. Guarda el resultado en su primer operando.
	`ISUB R0, 27  ; R0 = R0 - 27`
	`ISUB R0, R1  ; R0 = R0 - R1`

IMUL
Integer Multiply
	Interpreta ambos operandos como enteros y los multiplica. Guarda el resultado en su primer operando.
	`IMUL R0, 27  ; R0 = R0 * 27`
	`IMUL R0, R1  ; R0 = R0 * R1`

IDIV
Integer Divide
	Interpreta ambos operandos como enteros y divide el primero por el segundo. Guarda la parte entera del resultado en su primer operando.
	`IDIV R0, 27  ; R0 = R0 / 27`
	`IDIV R0, R1  ; R0 = R0 / R1`

IMOD
Integer Modulus
	Interpreta ambos operandos como enteros y divide el primero por el segundo. Guarda el resto en su primer operando.
	`IMOD R0, 27  ; R0 = Resto( R0 / 27 )`
	`IMOD R0, R1  ; R0 = Resto( R0 / R1 )`

ISGN Integer Sign Change	Interpreta su operando como entero y cambia su signo. `ISGN R0  ; R0 = –R0`

IMIN
Integer Minimum
	Interpreta ambos operandos como enteros y guarda el mínimo de los 2 en su primer operando.
	`IMIN R0, 27  ; R0 = Mínimo( R0, 27 )`
	`IMIN R0, R1  ; R0 = Mínimo( R0, R1 )`

IMAX
Integer Maximum
	Interpreta ambos operandos como enteros y guarda el máximo de los 2 en su primer operando.
	`IMAX R0, 27  ; R0 = Máximo( R0, 27 )`
	`IMAX R0, R1  ; R0 = Máximo( R0, R1 )`

IABS Integer Absolute Value	Interpreta un registro como entero y obtiene su valor absoluto. `IABS R0  ; R0 = ABS( R0 )`

Aritmética con floats

Instrucción	Descripción breve

FADD
Float Add
	Interpreta ambos operandos como floats y los suma. Guarda el resultado en su primer operando.
	`FADD R0, 13.2  ; R0 = R0 + 13.2`
	`FADD R0, R1  ; R0 = R0 + R1`

FSUB
Float Subtract
	Interpreta ambos operandos como floats y resta el segundo al primero. Guarda el resultado en su primer operando.
	`FSUB R0, 13.2  ; R0 = R0 – 13.2`
	`FSUB R0, R1  ; R0 = R0 - R1`

FMUL
Float Multiply
	Interpreta ambos operandos como floats y los multiplica. Guarda el resultado en su primer operando.
	`FMUL R0, 13.2  ; R0 = R0 * 13.2`
	`FMUL R0, R1  ; R0 = R0 * R1`

FDIV
Float Divide
	Interpreta ambos operandos como floats y divide el primero por el segundo. Guarda el resultado en su primer operando.
	`FDIV R0, 13.2  ; R0 = R0 / 13.2`
	`FDIV R0, R1  ; R0 = R0 / R1`

FMOD
Float Modulus
	Interpreta ambos operandos como floats y calcula el “resto en coma flotante” de dividir el primero por el segundo. Guarda el resultado en su primer operando.
	`FMOD R0, 13.2  ; R0 = fmod( R0, 13.2 )`
	`FMOD R0, R1  ; R0 = fmod( R0, R1 )`

FSGN Float Sign Change	Interpreta su operando como float y cambia su signo. `FSGN R0  ; R0 = –R0`

FMIN
Float Minimum
	Interpreta ambos operandos como floats y guarda el mínimo de los 2 en su primer operando.
	`FMIN R0, 13.2  ; R0 = Mínimo( R0, 13.2 )`
	`FMIN R0, R1  ; R0 = Mínimo( R0, R1 )`

FMAX
Float Maximum
	Interpreta ambos operandos como floats y guarda el máximo de los 2 en su primer operando.
	`FMAX R0, 13.2  ; R0 = Máximo( R0, 13.2 )`
	`FMAX R0, R1  ; R0 = Máximo( R0, R1 )`

FABS Float Absolute Value	Interpreta un registro como float y obtiene su valor absoluto. `FABS R0  ; R0 = ABS( R0 )`

Operaciones extendidas con floats	
Instrucción	Descripción breve
FLR Floor	Interpreta un registro como float y lo redondea hacia abajo a un valor entero. El valor aún es un float, no se convierte en entero. `FLR R0 ; R0 = floor( R0 )`
CEIL Ceiling	Interpreta un registro como float y lo redondea hacia arriba a un valor entero. El valor aún es un float, no se convierte en entero. `CEIL R0 ; R0 = ceil( R0 )`
ROUND	Interpreta un registro como float y lo redondea al valor entero más próximo. El valor aún es un float, no se convierte en entero. `ROUND R0 ; R0 = round( R0 )`
SIN Sine	Interpreta un registro como float y obtiene su seno. `SIN R0 ; R0 = sin( R0 )`
ACOS Arc Cosine	Interpreta un registro como float y obtiene su arco coseno. `ACOS R0 ; R0 = acos( R0 )`
ATAN2 2-Argument Arc Tangent	Interpreta 2 registros como floats y obtiene el arco tangente del ángulo dado por el vector: { DeltaX = 2º operando, DeltaY = 1º operando }. `ATAN2 R0, R1 ; R0 = atan2( R0, R1 )`
LOG Logarithm	Interpreta un registro como float y obtiene su logaritmo natural. `LOG R0 ; R0 = log( R0 )`
POW Power	Interpreta ambos registros como floats y obtiene el resultado de elevar el primero a la potencia del segundo. `POW R0, R1 ; R0 = Pow( R0, R1 )`

Directivas del ensamblador

Al igual que en lenguaje C, el ensamblador de Vircon32 incluye un preprocesador. Éste puede realizar una serie de operaciones de texto para alterar nuestro código fuente. Estas operaciones se invocan en el código con directivas, que son un conjunto de identificadores predefinidos que comienzan por `%`.

Esta sección listará todas las directivas soportadas y explicará su uso. Ten en cuenta que, como los demás elementos del lenguaje ensamblador, cada directiva debe escribirse en su propia línea individual.

Definiciones

Con la directiva `%define` podemos declarar identificadores y asignarles un valor de texto. Luego, cuando escribamos el identificador, será reemplazado por ese texto. Por ejemplo:

```
%define Pi  3.1416
%define ClearScreen  OUT GPU_Command, GPUCommand_ClearScreen

; usamos nuestras definiciones
_test:
  MOV R0, Pi
  ClearScreen   ; usar esto ejecutará la instrucción
```

Los identificadores también se pueden definir sin valor. Esto puede ser útil para usar flags en directivas condicionales (ver más abajo).

```
%define DEBUG    ; sin valor: se usará solo como flag para los condicionales
```

También podemos usar una definición sólo en ciertas partes de nuestro código, y eliminarla más tarde. Con la directiva `%undef` podemos eliminar una definición previa.

```
%define PositionX [10]   ; la posición X se guarda en la posición de memoria 10

; la definición sólo se puede usar en esta subrutina
_increment_position_x:
  MOV R0, PositionX
  IADD R0, 1
  MOV PositionX, R0
  RET

%undef PositionX
```

Condicionales

Podemos usar directivas para elegir si ciertas partes del código fuente se incluyen o no en nuestro programa. Aquí las únicas condiciones que podemos usar son `%ifdef` (comprueba si un identificador ha sido definido) y `%ifndef` (comprueba si no ha sido definido). Después de una de estas directivas debemos terminar el bloque con una directiva `%endif`. Esto formará un bloque cerrado, y las líneas contenidas entre ambas directivas se incluirán en nuestro programa sólo si se cumple la condición:

```
%ifdef DEBUG
  CALL _draw_hitboxes   ; en la versión final no se deberían ver las hitboxes
%endif
```

También podemos crear un segundo bloque usando `%else`, para incluir código distinto cuando no se cumpla la condición. ¡Y se pueden anidar diferentes bloques condicionales!

```
%ifdef DEBUG
  %include "DebugRoutines.asm"    ; las rutinas debug hacen más cosas para testear
%else
  %include "ReleaseRoutines.asm"  ; las rutinas release son más rápidas
%endif
```

Mensajes

Podemos usar directivas para mostrar mensajes en el ensamblaje de nuestro programa. La directiva `%warning` mostrará el mensaje como un aviso, que no detendrá el ensamblado. En cambio, usar `%error` trata el mensaje como un error y detiene el proceso.

```
%warning "Este mensaje deja que el ensamblaje continue"
%error "Pero ESTE lo detiene"
MOV R0, 17   ; el ensamblador no alcanza esta línea
```

Incluir archivos

La directiva `%include` busca un archivo externo y añade su contenido en ese punto. Esto hace posible dividir nuestros programas en varios archivos o reutilizar subrutinas:

Archivo principal (program.asm)

```
%include "Routines/increment.asm"

; ahora podemos llamar a la rutina _increment_x
CALL _increment_x
```

Archivo incluido (Routines/increment.asm)

```
_increment_x:
  MOV R0, [14]   ; la variable x se guarda en la posición de memoria 14
  IADD R0, 1
  MOV [14], R0
  RET
```

La cadena que usa %include indica una ruta de archivo relativa al archivo principal, y si es necesario usará separadores de carpeta o “..” para ir a la carpeta padre, igual que cualquier otra ruta.

Una distinción importante: ¡No confundas esta inclusión de archivos con lo que hace la palabra clave datafile! Usar datafile incrusta un fichero en el binario del programa, tras ensamblarlo. En cambio %include incrusta un archivo en el código fuente del programa.

Puertos de control E/S

Las instrucciones `IN` y `OUT` usan los puertos de control E/S que exponen otros chips de la consola para comunicarse con la CPU. El lenguaje ensamblador de Vircon32 identifica estos puertos por su nombre. Esta sección enumera los nombres de los puertos de cada chip, junto con una breve descripción de su valor y el tipo de dato que espera.

Algunos puertos sólo aceptan una pequeña lista de valores válidos. Estos puertos indican su tipo de datos como "Enum". El nombre de esos valores también se mostrará aquí.

Puertos de control: GPU			
Nombre del puerto	Acceso	Formato	Descripción breve
GPU_Command	Escrit.	Enum	Puerto donde la GPU recibe comandos de dibujo
GPU_RemainingPixels	Lectura	Entero	Pixels que la GPU aún puede dibujar este frame
GPU_ClearColor	Lec/Es	Color	Color usado para borrar la pantalla
GPU_MultiplyColor	Lec/Es	Color	Color de modulación usado al dibujar regiones
GPU_ActiveBlending	Lec/Es	Enum	Modo de mezclado activo actualmente
GPU_SelectedTexture	Lec/Es	Entero	ID de la textura seleccionada actualmente
GPU_SelectedRegion	Lec/Es	Entero	ID de la región seleccionada (en la textura actual)
GPU_DrawingPointX	Lec/Es	Entero	Coordenada X en pantalla donde se dibujan regiones
GPU_DrawingPointY	Lec/Es	Entero	Coordenada Y en pantalla donde se dibujan regiones
GPU_DrawingScaleX	Lec/Es	Float	Factor de escalado en X al dibujar regiones [1]. Invierte la región en X si es negativo
GPU_DrawingScaleY	Lec/Es	Float	Factor de escalado en Y al dibujar regiones [1]. Invierte la región en Y si es negativo
GPU_DrawingAngle	Lec/Es	Float	Ángulo de rotación al dibujar regiones [2], en radianes. Si es positivo crece según las agujas del reloj
GPU_RegionMinX	Lec/Es	Entero	Coordenada X mínima de la región seleccionada
GPU_RegionMinY	Lec/Es	Entero	Coordenada Y mínima de la región seleccionada
GPU_RegionMaxX	Lec/Es	Entero	Coordenada X máxima de la región seleccionada
GPU_RegionMaxY	Lec/Es	Entero	Coordenada Y máxima de la región seleccionada
GPU_RegionHotspotX	Lec/Es	Entero	Coordenada X del hotspot de la región seleccionada
GPU_RegionHotspotY	Lec/Es	Entero	Coordenada Y del hotspot de la región seleccionada

NOTAS:

[1]: Estos factores de escala sólo se aplican si se dibujan regiones con escalado ON.
[2]: Este ángulo sólo aplica si se dibujan regiones con rotación ON.

Comandos de la GPU (para GPU_Command)	
Nombre del comando	Descripción breve
GPUCommand_ClearScreen	Borra la pantalla con el color de borrado actual
GPUCommand_DrawRegion	Dibuja región actual (escalado OFF, rotación OFF)
GPUCommand_DrawRegionZoomed	Dibuja región actual (escalado ON, rotación OFF)
GPUCommand_DrawRegionRotated	Dibuja región actual (escalado OFF, rotación ON)
GPUCommand_DrawRegionRotozoomed	Dibuja región actual (escalado ON, rotación ON)

Modos de mezclado de la GPU (para GPU_ActiveBlending)	
Nombre del modo de mezclado	Descripción breve
GPUBlendingMode_Alpha	Mezclado alfa: con transparencia pero sin cambios de color
GPUBlendingMode_Add	Mezclado aditivo: suma los colores, efecto de luz
GPUBlendingMode_Subtract	Mezclado sustractivo: resta los colores, efecto de sombra

Puertos de control: SPU			
Nombre del puerto	Acceso	Formato	Descripción breve
SPU_Command	Escrit.	Enum	Puerto donde la SPU recibe comandos de sonido
SPU_GlobalVolume	Lec/Es	Float	Multiplicador global de volumen para todos los canales
SPU_SelectedSound	Lec/Es	Entero	ID del sonido seleccionado actual
SPU_SelectedChannel	Lec/Es	Entero	ID del canal de sonido seleccionado actual
SPU_SoundLength	Lectura	Entero	Nº de samples del sonido actual
SPU_SoundPlayWithLoop	Lec/Es	Booleano	¿El sonido actual se reproducirá con bucle?
SPU_SoundLoopStart	Lec/Es	Entero	Primer sample del bucle del canal actual
SPU_SoundLoopEnd	Lec/Es	Entero	Último sample del bucle del canal actual
SPU_ChannelState	Lectura	Enum	Estado (parada/pausa/play) del canal actual
SPU_ChannelAssignedSound	Lec/Es	Entero	ID del sonido asignado al canal actual
SPU_ChannelVolume	Lec/Es	Float	Multiplicador de volumen del canal actual
SPU_ChannelSpeed	Lec/Es	Float	Multiplicador de velocidad del canal actual
SPU_ChannelLoopEnabled	Lec/Es	Booleano	¿El canal actual tiene activado el bucle?
SPU_ChannelPosition	Lec/Es	Entero	Posición de reproducción del canal actual

Comandos de la SPU (para SPU_Command)	
Nombre del comando	Descripción breve
SPUCommand_PlaySelectedChannel	Su efecto depende del estado del canal seleccionado: • Parado: empieza a reproducir su sonido asignado • Pausado: continúa la reproducción anterior • Reproduciendo: reinicia la reproducción
SPUCommand_PauseSelectedChannel	El canal seleccionado pausa la reproducción. Si estaba parado o ya pausado no tiene ningún efecto
SPUCommand_StopSelectedChannel	El canal seleccionado detiene la reproducción
SPUCommand_PauseAllChannels	Pausa todos los canales en reproducción
SPUCommand_ResumeAllChannels	Todos los canales pausados continúan la reproducción
SPUCommand_StopAllChannels	Detiene todos los canales pausados o en reproducción

Estados de canales de sonido (para SPU_ChannelState)	
Nombre del estado	Descripción breve
SPUChannelState_Stopped	El canal está parado
SPUChannelState_Paused	El canal está pausado y puede continuar la reproducción
SPUChannelState_Playing	El canal está actualmente reproduciendo sonido

Puertos de control: Controlador de mandos			
Nombre del puerto	Acceso	Formato	Descripción breve
INP_SelectedGamepad	Lec/Es	Entero	ID del mando seleccionado actualmente
INP_GamepadConnected	Lectura	Booleano	¿El mando actual está conectado al puerto?
INP_GamepadLeft	Lectura	Entero	Estado de la dirección izquierda del mando actual
INP_GamepadRight	Lectura	Entero	Estado de la dirección derecha del mando actual
INP_GamepadUp	Lectura	Entero	Estado de la dirección arriba del mando actual
INP_GamepadDown	Lectura	Entero	Estado de la dirección abajo del mando actual
INP_GamepadButtonStart	Lectura	Entero	Estado del botón Start del mando actual
INP_GamepadButtonA	Lectura	Entero	Estado del botón A del mando actual
INP_GamepadButtonB	Lectura	Entero	Estado del botón B del mando actual
INP_GamepadButtonX	Lectura	Entero	Estado del botón X del mando actual
INP_GamepadButtonY	Lectura	Entero	Estado del botón Y del mando actual
INP_GamepadButtonL	Lectura	Entero	Estado del botón L del mando actual
INP_GamepadButtonR	Lectura	Entero	Estado del botón R del mando actual

Puertos de control: Temporizador			
Nombre del puerto	Acceso	Formato	Descripción breve
TIM_CurrentDate	Lectura	Fecha	Fecha actual, en el formato interno de fecha
TIM_CurrentTime	Lectura	Entero	Hora actual del día, expresada en segundos pasados
TIM_FrameCounter	Lectura	Entero	Frames pasados desde el ultimo arranque o reset
TIM_CycleCounter	Lectura	Entero	Ciclos de CPU transcurridos del frame actual

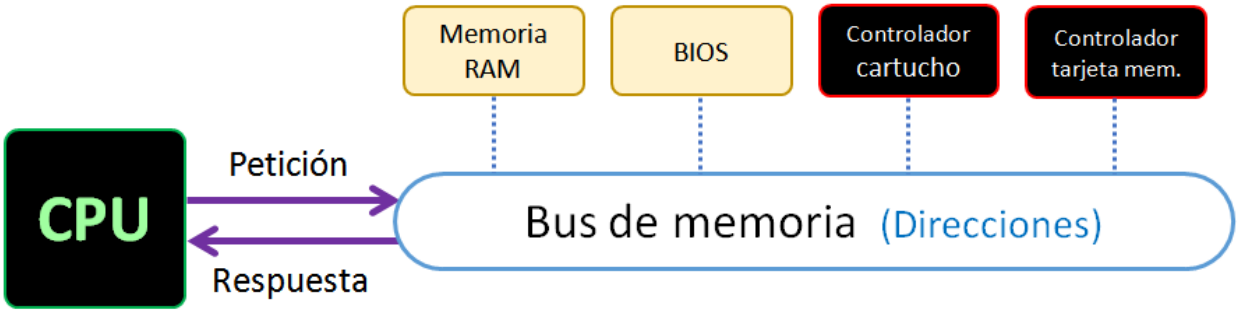
Puertos de control: Generador de números aleatorios			
Nombre del puerto	Acceso	Formato	Descripción breve
RNG_CurrentValue	Lec/Es	Entero	Valor/semilla actual de la secuencia pseudoaleatoria

Puertos de control: Controlador de tarjeta de memoria			
Nombre del puerto	Acceso	Formato	Descripción breve
MEM_CardConnected	Lectura	Booleano	¿Hay una tarjeta de memoria en la ranura?

Puertos de control: Controlador de cartucho			
Nombre del puerto	Acceso	Formato	Descripción breve
CAR_CartridgeConnected	Lectura	Booleano	¿Hay un cartucho en la ranura?
CAR_ProgramROMSize	Lectura	Entero	Nº palabras ROM de programa cartucho actual
CAR_NumberOfTextures	Lectura	Entero	Número de texturas del cartucho actual
CAR_NumberOfSounds	Lectura	Entero	Número de sonidos del cartucho actual

Mapeado de la memoria

En una consola Vircon32 puede haber hasta 4 dispositivos que tengan memoria accesible para la CPU. Cada uno de ellos se conecta al bus de memoria y la CPU puede pedirles leer o escribir en posiciones específicas de su memoria.



Esta tabla muestra el rango de direcciones de memoria para cada uno de los dispositivos, cuando están presentes. La memoria RAM y la BIOS siempre están presentes pero el cartucho y, especialmente, la tarjeta de memoria no siempre lo estarán.

Dispositivo conectado	Rango de direcciones
Memoria RAM	`0x00000000 – 0x003FFFFF`
ROM de programa BIOS	`0x10000000 – 0x100FFFFF (max)`
ROM de programa cartucho	`0x20000000 – 0x27FFFFFF (max)`
RAM tarjeta de memoria	`0x30000000 – 0x30003FFF`

Las memorias tipo RAM permiten acceso de lectura y escritura, mientras que las ROM son de sólo lectura. Ten en cuenta que el tamaño de las memorias de BIOS y de cartucho será variable. Los cartuchos son intercambiables, y diferentes sistemas Vircon32 pueden tener diferentes BIOS.

Para la memoria RAM es útil saber que la pila de la CPU empieza en la última dirección RAM y crece hacia direcciones inferiores. Así que en tus programas es una buena práctica usar la RAM desde la dirección 0 y crecer hacia arriba, para que no entren en conflicto.

Consejos generales

Esta última sección tiene algunos consejos prácticos que te serán útiles si estás aprendiendo a programar en ensamblador de Vircon32. Aunque este documento pretende ser una guía/texto de referencia más que un curso de ensamblador propiamente dicho, veremos algunos consejos que te serán útiles y te evitarán varios errores comunes.

Variables en memoria

Si conoces otros lenguajes de programación de alto nivel (¡que deberías antes de intentar el ensamblador!) puede que te hayas preguntado cómo se declaran las variables en este lenguaje ensamblador. La respuesta es: ¡no se declaran!

Tú mismo debes elegir dónde vas a almacenar cada una de tus variables, si en memoria o en registros. Pero como el contenido de los registros puede ser volátil (una subrutina puede haberlo modificado sin que cuentes con ello), lo habitual es que prefieras asignar a tus variables una dirección de memoria.

Aún puedes tener "pseudo-variables" usando `%define` para darles un nombre y dirección. Pero depende de ti manejar el tipo de cada valor (entero, flotante...). Y también recuerda que, mientras que una variable en un registro siempre se puede usar directamente, una variable en memoria necesita pasar por registros para manipularla. Como en este ejemplo:

```
%define Score  [0]             ; entero
%define BonusMultiplier  [1]   ; float

; esta subrutina hace: Score *= BonusMultiplier
_increment_score:
  MOV R0, Score            ; leer Score a un registro
  CIF R0                   ; convertir a float para poder multiplicar ambos
  MOV R1, BonusMultiplier  ; leer BonusMultiplier a un registro
  IMUL R0, R1              ; ahora R0 = Score * BonusMultiplier
  CFI R0                   ; para guardar de nuevo Score se debe convertir a entero
  MOV Score, R0            ; guardar el nuevo Score desde el registro
  RET                      ; hemos terminado, volver
```

Preservar los registros

Quizás vieras un problema en la subrutina anterior: los registros R0 y R1 se usan como variables temporales, pero el código que llama a esa subrutina puede no ser consciente de ello. Puede creer que la subrutina sólo maneja las variables. En ese caso, probablemente experimente errores: el código que llama puede estar guardando valores en, digamos, R0.

Para estar a salvo de esto es una buena práctica preservar los registros que usas en subrutinas. O al menos los que puedan ser problemáticos. Lo puedes hacer guardándolos en la pila antes de cambiar su valor, y recuperándolos cuando hayas terminado. Entonces el código que llama ya no verá cambiadas de repente sus variables de trabajo.

```
; esta subrutina calcula la raíz cuadrada de R0; devuelve el
; resultado también en R0, así que no necesitamos preservarlo
_square_root:
  PUSH R1           ; guardar R1 en la pila
  MOV R1, 0.5       ; ¡POW sólo puede operar con 2 registros! Necesitamos R1
  POW R0, R1        ; R0 = sqrt( R0 ) -> tenemos el resultado
  POP R1            ; restauramos R1 desde la pila
  RET               ; hemos terminado, volver (el resultado se da en R0)
```

La librería estándar de C

Si no estás seguro de cómo interactuar con la consola en lenguaje ensamblador, un buen sitio para buscar es la librería estándar del compilador de C de Vircon32. En los archivos de cabecera verás que muchas de las funciones están implementadas en ensamblador, o tienen alguna parte importante hecha con un bloque ensamblador.

El ejemplo siguiente procede del fichero de cabecera video.h. La sintaxis de llaves que se ve aquí es un mecanismo que permite capturar variables de C desde ensamblador. Al compilar, la variable nombrada se sustituye por su dirección.

```
// asigna el punto de referencia ("hotspot") de la región seleccionada
void set_region_hotspot( int hotspot_x, int hotspot_y )
{
    asm
    {
        "mov R0, {hotspot_x}"
        "out GPU_RegionHotSpotX, R0"
        "mov R0, {hotspot_y}"
        "out GPU_RegionHotSpotY, R0"
    }
}
```

Errores hardware

Como cualquier CPU, el procesador de Vircon32 puede encontrar errores en algunas situaciones, como dividir por cero. Cuando esto ocurra tu programa se abortará y la CPU activará un error hardware. La BIOS te mostrará un mensaje como este, y podrás ver cierta información del error. Pero tras esto tu programa no tiene forma de recuperarse.