Vircon32

CONSOLA VIRTUAL DE 32 BITS

Especificación del sistema

Parte 3: El procesador (CPU)

Documento con fecha 2023.12.18 Escrito por Carra

¿Qué es esto?

Este documento es la parte número 3 de la especificación del sistema Vircon32. Esta serie de documentos define el sistema Vircon32, y provee una especificación completa que describe en detalle sus características y comportamiento.

El principal objetivo de esta especificación es definir un estándar de lo que es un sistema Vircon32, y cómo debe implementarse un sistema de juego para que se considere conforme a él. Además, al ser Vircon32 es un sistema virtual, un importante objetivo adicional de estos documentos es proporcionar a cualquiera el conocimiento para crear sus propias implementaciones de Vircon32.

Sobre Vircon32

El proyecto Vircon32 fue creado de forma independiente por Carra. El sistema Vircon32 y su material asociado (incluyendo documentos, software, código fuente, arte y cualquier otro elemento relacionado) son propiedad del autor original.

Vircon32 es un proyecto libre y de código abierto en un esfuerzo por promover que cualquiera pueda jugar a la consola y crear software para ella. Para obtener información más detallada al respecto, se recomienda consultar los textos de licencia incluidos en cada uno de los programas disponibles.

Sobre este documento

Índice

La Parte 3 de la especificación define el procesador de la consola, ó CPU (Unidad Central de Procesado). Este documento describirá el comportamiento de este chip, sus elementos internos, la arquitectura de su conjunto de instrucciones y el proceso de ejecución.

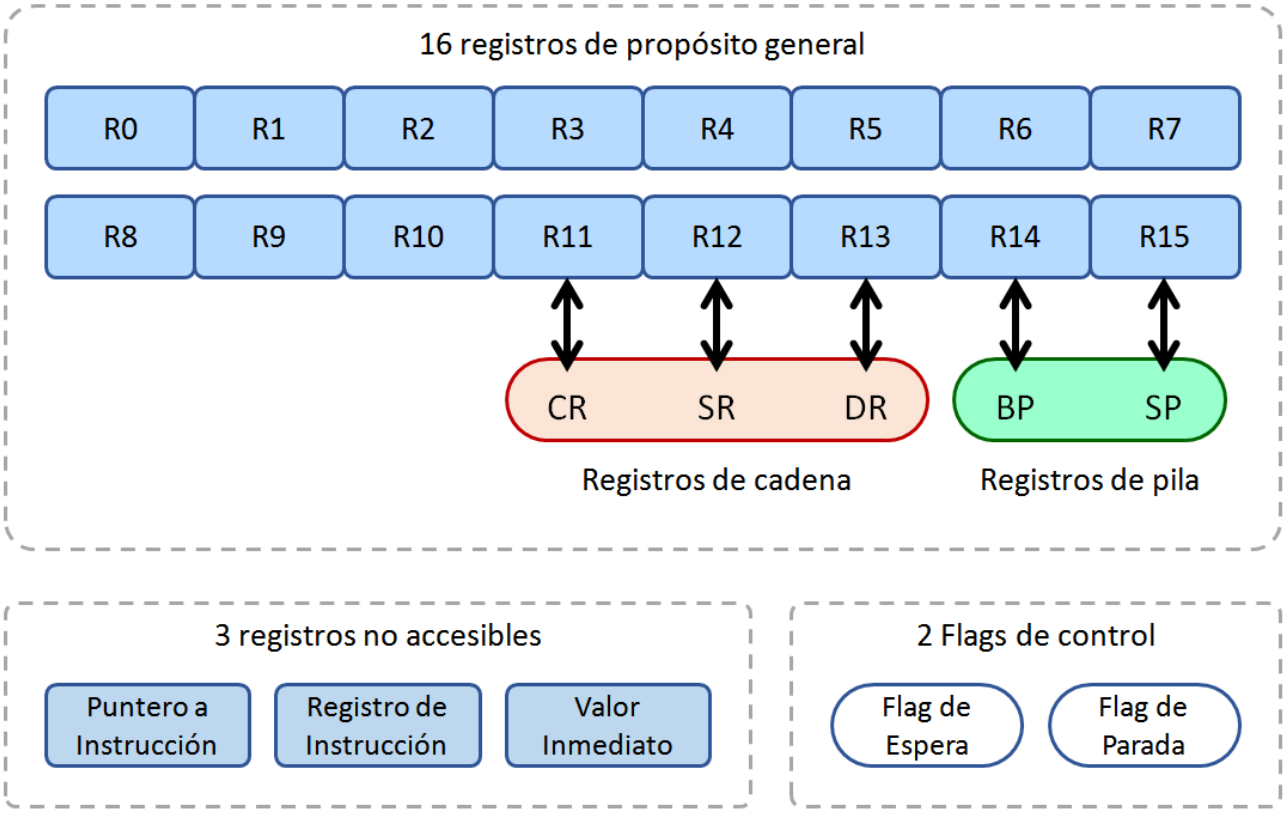
1 Introducción

La CPU es el chip principal de la consola Vircon32. Su función es leer las instrucciones del programa y ejecutarlas en secuencia para producir el comportamiento descrito del programa. Como parte de esta ejecución, la CPU también se comunica con otros componentes de la consola y envía comandos a otros chips para controlar sus funciones. Como la CPU es siempre el chip que inicia la comunicación, está conectado como dispositivo maestro tanto al bus de memoria como al bus de control.

Este procesador tiene una arquitectura de 32 bits pura: como todos los componentes de Vircon32, sólo puede trabajar con datos de 32 bits. Su juego de instrucciones está vagamente basado en la familia de procesadores x86, pero adaptado al conjunto de características de Vircon32. También se ha simplificado para que sólo haya 64 instrucciones. Éstas también tienen menos variantes que en otras CPUs.

1.1 Elementos internos de la CPU

Esta imagen da una visión general de los elementos internos que existen en la CPU. Las siguientes secciones explicarán cada uno de esos elementos, y mostrarán cómo algunos de los registros generales también reciben funciones específicas cuando la CPU opera con la pila (el área mostrada en verde) o hace procesado de cadenas (el área roja).



2 Registros de la CPU

La CPU dispone de un conjunto de registros de 32 bits. Su función es almacenar los datos necesarios para la ejecución. Los registros se pueden separar en 2 grupos diferentes:

2.1 Registros de propósito general

La misión de los registros de propósito general están es dar soporte a la ejecución de programas. Son los únicos registros directamente accesibles para el programador. La CPU tiene una matriz de 16 registros de propósito general, que se denominan R0 a R15.

Los últimos 5 de esos registros también se usan en ciertas funciones de la CPU (uso de pila y cadenas) que se cubrirán en los próximos capítulos. Debido a ello, también recibirán nombres alternativos que reflejan su papel en esas funciones.

2.2 Registros internos

Estos registros están reservados para el funcionamiento de la propia CPU, por lo que no son directamente accesibles al programa. Los 3 registros internos son:

Puntero a Instrucción

Contiene la dirección de memoria de donde la CPU leerá la siguiente instrucción del programa.

Registro de Instrucción

Guarda la última instrucción leída, es decir, la que se está ejecutando en ese momento.

Valor Inmediato

Si la última instrucción leída utiliza un valor inmediato, se lee y se almacena aquí.

3 La pila

La CPU implementa una pila hardware, que se usa para guardar y extraer valores. La política es sacar primero el último valor guardado. Su función principal es ser usada por las instrucciones de llamada y retorno para dirigir el flujo del programa. Los lenguajes de alto nivel también suelen utilizar la pila para almacenar "stack frames", que guardan los contextos de llamadas a funciones del lenguaje. El programador también puede insertar y extraer valores de la pila de forma explícita.

3.1 Funcionamiento de la pila

La pila se sitúa en las posiciones más altas de la memoria RAM y crece hacia abajo. El funcionamiento de la pila usa los 2 últimos registros de propósito general. Además de su nombre normal, también se les puede referenciar con estos alias por su papel en la pila:

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).

Proceso para guardar en la pila

Cuando la CPU guarda un valor en la pila, sigue estos pasos.
En siguientes secciones, esta operación se representará como: Pila.Guardar(Valor)

(1) Memoria[PunteroPila] = Valor;
(2) PunteroPila –= 1;

Proceso para sacar de la pila

Cuando la CPU saca un valor de la pila a un registro, sigue estos pasos.
En siguientes secciones, esta operación se representará como: Registro = Pila.Sacar( )

(1) PunteroPila += 1;
(2) Registro = Memoria[PunteroPila];

4 Operaciones con cadenas

Algunas instrucciones de la CPU están diseñadas para permitir que una única instrucción de programa opere sobre una serie de direcciones de memoria consecutivas. Algunas CPU se refieren a estos conjuntos de datos como "cadenas", aunque no representen texto.

Las operaciones de cadena usan los 3 registros de propósito general situados justo antes de los registros de pila. Además de su nombre normal, también se les puede referenciar con estos alias por su papel en las operaciones con cadenas:

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

Para permitir a una instrucción procesar varias direcciones, la CPU repite esa misma instrucción automáticamente hasta que el registro contador llega a cero. Así, la CPU implementa un rápido bucle interno que permite un procesado masivo más eficiente.

Cada vez que se procesa la instrucción el contador disminuye automáticamente, y se incrementarán hasta 2 registros puntero (para las direcciones de origen y destino).

La forma en que un programa usaría operaciones de cadena es: primero escribir las direcciones de origen y destino correctas en SR y DR, luego escribir el número de iteraciones en CR y, por último, utilizar la instrucción de cadena.

5 Flags de control

La CPU dispone de 2 flags de control que se usan para suspender el funcionamiento de la CPU en ciertas situaciones. Son de un solo bit cada una, y se interpretan como activadas cuando son 1, y no activadas cuando son 0. Su significado y uso es el siguiente:

Flag de parada

Cuando se activa, detiene la operación de la CPU hasta el siguiente reinicio o encendido.

Flag de espera

Cuando se activa, detiene la operación de la CPU hasta que empiece el siguiente frame.

El programador puede activar ambas flags para diferentes propósitos. Usar la flag de espera permite a los programas controlar su velocidad de ejecución. Por otro lado, si un programa ha terminado, puede simplemente parar la CPU como mecanismo de salida.

6 Respuestas a señales de control

Como todos los componentes de la consola, cada vez que se envía una señal de control, la CPU la recibe y produce una respuesta para procesar ese evento. Para cada una de las señales de control, la CPU responderá realizando las siguientes acciones:

Señal de reset:

- Las flags de parada y de espera se borran a 0.
- Todos los registros se borran al valor entero 0.
- La pila se reinicia apuntando BP y SP a la última dirección RAM: 0x003FFFFF.
- El puntero a instrucción se fija a la dirección de inicio de la BIOS: 0x10000004.

Señal de frame:

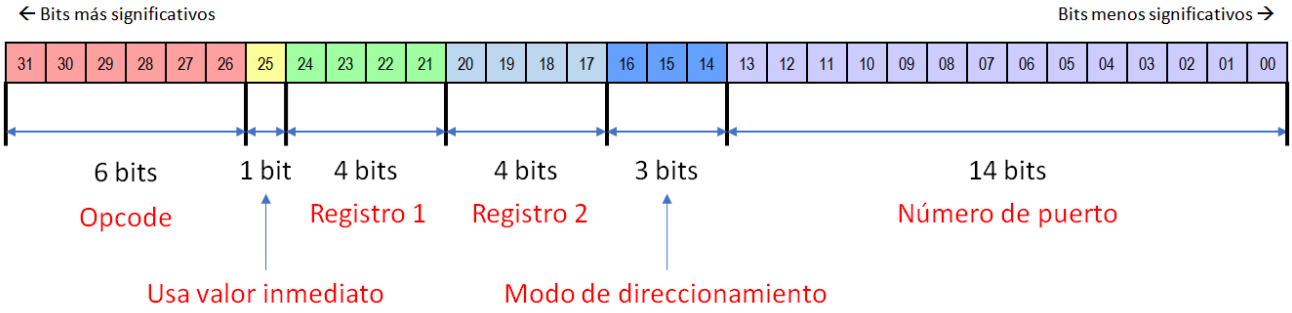
- El flag de espera se borra a 0.

Señal de ciclo:

- Si alguna de las flags de control está activada, la CPU no hará nada.
- De lo contrario, la CPU realiza un ciclo de procesado como detalla el capítulo 9.

7 Formato de las instrucciones

Las instrucciones de la CPU son valores individuales de 32 bits que la CPU interpreta como un conjunto de campos. Esta imagen muestra la posición y tamaño de cada campo.



Todos estos campos se interpretan como enteros sin signo. Los campos "Registro 1" y "Registro 2", cuando son usados por una instrucción, designan un índice de registro (de 0 a 15) dentro de la matriz de registros de propósito general de la CPU.

Si algún campo no se usa en una instrucción concreta debería ponerse a cero, aunque esto no influye en el correcto procesamiento de la instrucción.

Algunas instrucciones necesitarán usar una segunda palabra de 32 bits, llamada valor inmediato. Esto se indica poniendo a 1 el campo "Usa valor inmediato". En ese caso se leerá una segunda palabra de la memoria junto con la propia instrucción. Este segundo valor se guarda en el registro Valor Inmediato y se usará al procesar la instrucción.

Debe tenerse en cuenta que un sistema Vircon32 no está obligado a detectar instrucciones mal formadas. Procesar cualquier instrucción incorrecta producirá un comportamiento dependiente de la implementación.

8 Juego de instrucciones

Las distintas instrucciones que admite la CPU se representan con un código de operación (el campo Opcode de la instrucción), que es un número entero sin signo de 6 bits. La CPU de Vircon32 tiene exactamente 64 instrucciones, así que no hay códigos sin usar.

Muchas instrucciones usadas en esta CPU se basan en las de x86, y en muchos casos usan los mismos nombres. Las instrucciones de ensamblador se escriben usando la sintaxis de Intel, tanto en este documento como en el programa ensamblador de Vircon32.

Podemos distinguir varios grupos de instrucciones atendiendo a su finalidad. Las siguientes tablas enumeran las 64 instrucciones agrupadas por su función. Los códigos también se han numerado teniendo en cuenta esos grupos para una mejor comprensión.

Control de la CPU		
OpCode	Instrucción	Descripción breve
00	HLT	Parar la CPU hasta el siguiente reset
01	WAIT	Esperar al siguiente frame

Instrucciones de salto		
OpCode	Instrucción	Descripción breve
02	JMP	Salto incondicional
03	CALL	Llamar a subrutina
04	RET	Retorno de subrutina
05	JT	Saltar si verdadero
06	JF	Saltar si falso

Comparación de enteros		
OpCode	Instrucción	Descripción breve
07	IEQ	Igualdad de enteros
08	INE	Desigualdad de enteros
09	IGT	Mayor que entre enteros
10	IGE	Mayor o igual entre enteros
11	ILT	Menor que entre enteros
12	ILE	Menor o igual entre enteros

Comparación de floats		
OpCode	Instrucción	Descripción breve
13	FEQ	Igualdad de floats
14	FNE	Desigualdad de floats
15	FGT	Mayor que entre floats
16	FGE	Mayor o igual entre floats
17	FLT	Menor que entre floats
18	FLE	Menor o igual entre floats

Movimiento de datos		
OpCode	Instrucción	Descripción breve
19	MOV	Mover valor
20	LEA	Cargar dirección de posición de memoria
21	PUSH	Guarda valor en lo alto de la pila
22	POP	Saca valor de lo alto de la pila
23	IN	Lee valor de un puerto de E/S
24	OUT	Escribe valor a un puerto de E/S

Operaciones con cadenas		
OpCode	Instrucción	Descripción breve
25	MOVS	Mover cadena (memcpy por hardware)
26	SETS	Asignar cadena (memset por hardware)
27	CMPS	Comparar cadenas (memcmp por hardware)

Conversión de datos		
OpCode	Instrucción	Descripción breve
28	CIF	Convertir entero a float
29	CFI	Convertir float a entero
30	CIB	Convertir entero a booleano
31	CFB	Convertir float a booleano

Operaciones binarias		
OpCode	Instrucción	Descripción breve
32	NOT	NOT bit a bit
33	AND	AND bit a bit
34	OR	OR bit a bit
35	XOR	XOR bit a bit
36	BNOT	NOT booleano
37	SHL	Desplazar bits a la izquierda

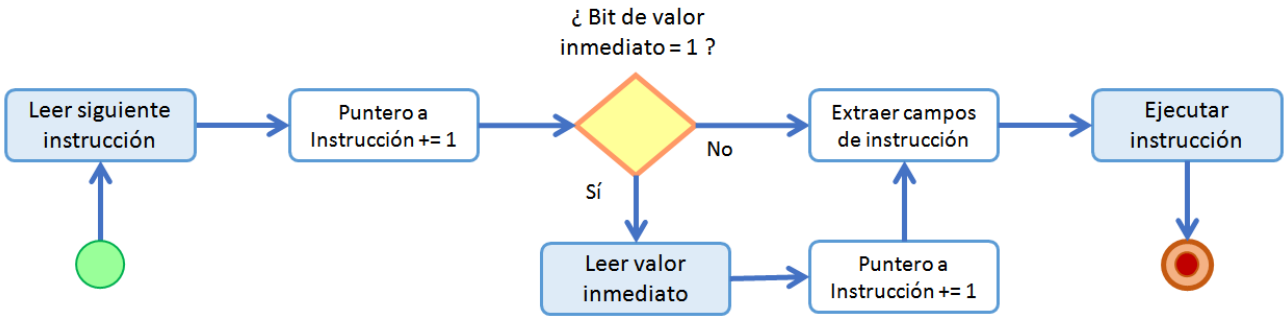
Aritmética con enteros		
OpCode	Instrucción	Descripción breve
38	IADD	Sumar enteros
39	ISUB	Restar enteros
40	IMUL	Multiplicar enteros
41	IDIV	Dividir enteros
42	IMOD	Módulo entre enteros
43	ISGN	Cambiar de signo un entero
44	IMIN	Mínimo entre enteros
45	IMAX	Máximo entre enteros
46	IABS	Valor absoluto de un entero

Aritmética con floats		
OpCode	Instrucción	Descripción breve
47	FADD	Sumar floats
48	FSUB	Restar floats
49	FMUL	Multiplicar floats
50	FDIV	Dividir floats
51	FMOD	Módulo entre floats
52	FSGN	Cambiar de signo un float
53	FMIN	Mínimo entre floats
54	FMAX	Máximo entre floats
55	FABS	Valor absoluto de un float

Operaciones extendidas con floats		
OpCode	Instrucción	Descripción breve
56	FLR	Redondear hacia abajo
57	CEIL	Redondear hacia arriba
58	ROUND	Redondear al entero más cercano
59	SIN	Seno
60	ACOS	Arco coseno
61	ATAN2	Arco tangente a partir de Y y X
62	LOG	Logaritmo natural
63	POW	Elevar a una potencia

9 El ciclo de procesado de la CPU

La CPU divide el procesado de su programa en ciclos. El propósito de cada ciclo es procesar una nueva instrucción individual del programa. Un ciclo de procesado de la CPU se describe mediante el siguiente algoritmo:



Los pasos marcados en azul son aquellos en los que podría darse un error hardware. En tal caso, este proceso se interrumpiría y se llevaría a cabo el procesado del error en cuestión. Las acciones de este ciclo de procesado se realizan del siguiente modo:

Leer siguiente instrucción:

La CPU lee la dirección de memoria a la que apunta el Puntero a Instrucción y guarda su contenido en el Registro de Instrucción.

Leer valor inmediato:

La CPU lee la dirección de memoria a la que apunta el Puntero a Instrucción y guarda su contenido en el registro de Valor Inmediato.

Extraer campos de la instrucción:

El valor almacenado en el Registro de Instrucción se divide en los diferentes valores enteros codificados en los bits de la instrucción. Este paso puede no ser necesario si la implementación es capaz de trabajar directamente con los campos de bits.

Ejecutar instrucción:

La CPU elegirá el procesado adecuado para la instrucción según su opcode. Después aplicará uno de los 64 casos posibles que se detallan en la siguiente sección.

Debe tenerse en cuenta que la CPU siempre debe respetar el campo "Usa valor inmediato", usando ese bit para determinar si debe leer otra palabra. Esto debe hacerse incluso si la instrucción no está bien formada. Procesar ese tipo de instrucción resultará en un comportamiento indefinido.

10 Procesado de instrucciones

Esta sección provee una especificación completa de cómo se estructura cada instrucción y su proceso de ejecución. Las siguientes subsecciones cubren los 64 opcodes posibles.

Es importante señalar que, salvo la instrucción MOV, todas las instrucciones tienen 1 o 2 variantes. En todos los casos, las instrucciones con 2 variantes determinan cuál se utiliza por el valor del bit "Usa valor inmediato".

---

00 Instrucción HLT (Halt)

Estructura y variantes:

HLT

Acciones de procesado:

FlagParada = 1

Descripción:

HLT activa la Flag de Parada de la CPU. Esto hará que la CPU detenga la ejecución hasta que la bandera se borre en el siguiente encendido o reinicio de la consola. Obsérvese que otros componentes seguirán funcionando: por ejemplo, si la SPU estaba reproduciendo música, seguirá haciéndolo.

---

01 Instrucción WAIT

Estructura y variantes:

WAIT

Acciones de procesado:

FlagEspera = 1

Descripción:

WAIT activa la Flag de Espera de la CPU. Esto hará que la CPU detenga la ejecución hasta que la bandera se borre, cuando el Temporizador señale el inicio del siguiente frame. Un reinicio o encendido también reanuda la ejecución de la CPU, ya que también provocan el inicio de un nuevo frame.

Cuando comience el nuevo frame, la CPU reanuda la ejecución siguiendo el orden habitual, es decir, procesando la instrucción directamente después de WAIT. Téngase en cuenta que los demás componentes seguirán funcionando.

02 Instrucción JMP (Jump)

Estructura y variantes:

(Variante 1): JMP { ValorInmediato }
(Variante 2): JMP { Registro1 }

Acciones de procesado:

(Variante 1): PunteroInstruccion = ValorInmediato
(Variante 2): PunteroInstruccion = Registro1

Descripción:

JMP realiza un salto incondicional a la dirección especificada por su operando. Después de procesar esta instrucción la CPU continuará la ejecución en la nueva dirección.

03 Instrucción CALL

Estructura y variantes:

(Variante 1): CALL { ValorInmediato }
(Variante 2): CALL { Registro1 }

Acciones de procesado:

(Variante 1):
Pila.Guardar(PunteroInstruccion)
PunteroInstruccion = ValorInmediato
(Variante 2):
Pila.Guardar(PunteroInstruccion)
PunteroInstruccion = Registro1

Descripción:

CALL realiza una llamada a la subrutina en la dirección especificada. Al procesar esta instrucción, el Puntero a Instrucción actual (que ya se había incrementado) se guardará en la parte superior de la pila y luego se sobrescribirá con la nueva dirección. La ejecución continuará en la nueva dirección.

04 Instrucción RET (Return)

Estructura y variantes:

RET

Acciones de procesado:

PunteroInstruccion = Pila.Sacar( )

Descripción:

RET vuelve de una subrutina llamada previamente. Al procesar esta instrucción, el Puntero a Instrucción se sobrescribe con el valor en lo más alto de la pila. La ejecución continuará entonces en esa dirección previamente guardada.

---

05 Instrucción JT (Jump if True)

Estructura y variantes:

(Variante 1): JT { Registro1 }, { ValorInmediato }
(Variante 2): JT { Registro1 }, { Registro2 }

Acciones de procesado:

(Variante 1): if Registro1 != 0 then PunteroInstruccion = ValorInmediato
(Variante 2): if Registro1 != 0 then PunteroInstruccion = Registro2

Descripción:

JT realiza un salto sólo si su primer operando es verdadero, es decir, distinto de cero cuando se toma como un entero. En ese caso se comporta igual que un salto incondicional. En caso contrario, no tiene ningún efecto.

---

06 Instrucción JF (Jump if False)

Estructura y variantes:

(Variante 1): JF { Registro1 }, { ValorInmediato }
(Variante 2): JF { Registro1 }, { Registro2 }

Acciones de procesado:

(Variante 1): if Registro1 == 0 then InstructionPointer = ValorInmediato
(Variante 2): if Registro1 == 0 then InstructionPointer = Registro2

Descripción:

JF realiza un salto sólo si su primer operando es falso, es decir, cero tomado como entero. En ese caso equivale a un salto incondicional. En otro caso no hay ningún efecto.

---

07 Instrucción IEQ (Integer Equal)

Estructura y variantes:

(Variante 1): IEQ { Registro1 }, { ValorInmediato }
(Variante 2): IEQ { Registro1 }, { Registro2 }

Acciones de procesado:

(Variante 1): if Registro1 == ValorInmediato then Registro1 = 1 else Registro1 = 0
(Variante 2): if Registro1 == Registro2 then Registro1 = 1 else Registro1 = 0

Descripción:

IEQ toma dos operandos interpretados como enteros y comprueba si son iguales. Guarda el resultado booleano en el primer operando, que siempre es un registro.

---

08 Instrucción INE (Integer Not Equal)

Estructura y variantes:

(Variante 1): INE { Registro1 }, { ValorInmediato }
(Variante 2): INE { Registro1 }, { Registro2 }

Acciones de procesado:

(Variante 1): if Registro1 != ValorInmediato then Registro1 = 1 else Registro1 = 0
(Variante 2): if Registro1 != Registro2 then Registro1 = 1 else Registro1 = 0

Descripción:

INE toma dos operandos interpretados como enteros y comprueba si son distintos. Guarda el resultado booleano en el primer operando, que siempre es un registro.

---

09 Instrucción IGT (Integer Greater Than)

Estructura y variantes:

(Variante 1): IGT { Registro1 }, { ValorInmediato }
(Variante 2): IGT { Registro1 }, { Registro2 }

Acciones de procesado:

(Variante 1): if Registro1 > ValorInmediato then Registro1 = 1 else Registro1 = 0
(Variante 2): if Registro1 > Registro2 then Registro1 = 1 else Registro1 = 0

Descripción:

IGT toma dos operandos interpretados como enteros y comprueba si el primero es mayor que el segundo. Guarda el resultado booleano en el primer operando, que siempre es un registro.

10 Instrucción IGE (Integer Greater or Equal)

Estructura y variantes:

(Variante 1): IGE { Registro1 }, { ValorInmediato }
(Variante 2): IGE { Registro1 }, { Registro2 }

Acciones de procesado:

(Variante 1): if Registro1 >= ValorInmediato then Registro1 = 1 else Registro1 = 0
(Variante 2): if Registro1 >= Registro2 then Registro1 = 1 else Registro1 = 0

Descripción:

IGE toma dos operandos interpretados como enteros y comprueba si el primero es mayor o igual que el segundo. Guarda el resultado booleano en el primer operando, que siempre es un registro.

11 Instrucción ILT (Integer Less Than)

Estructura y variantes:

(Variante 1): ILT { Registro1 }, { ValorInmediato }
(Variante 2): ILT { Registro1 }, { Registro2 }

Acciones de procesado:

(Variante 1): if Registro1 < ValorInmediato then Registro1 = 1 else Registro1 = 0
(Variante 2): if Registro1 < Registro2 then Registro1 = 1 else Registro1 = 0

Descripción:

ILT toma dos operandos interpretados como enteros y comprueba si el primero es menor que el segundo. Guarda el resultado booleano en el primer operando, que siempre es un registro.

12 Instrucción ILE (Integer Less or Equal)

Estructura y variantes:

(Variante 1): ILE { Registro1 }, { ValorInmediato }
(Variante 2): ILE { Registro1 }, { Registro2 }

Acciones de procesado:

(Variante 1): if Registro1 <= ValorInmediato then Registro1 = 1 else Registro1 = 0
(Variante 2): if Registro1 <= Registro2 then Registro1 = 1 else Registro1 = 0

Descripción:

ILE toma dos operandos interpretados como enteros y comprueba si el primero es menor o igual que el segundo. Guarda el resultado booleano en el primer operando, que siempre es un registro.

---

13 Instrucción FEQ (Float Equal)

Estructura y variantes:

(Variante 1): FEQ { Registro1 }, { ValorInmediato }
(Variante 2): FEQ { Registro1 }, { Registro2 }

Acciones de procesado:

(Variante 1): if Registro1 == ValorInmediato then Registro1 = 1 else Registro1 = 0
(Variante 2): if Registro1 == Registro2 then Registro1 = 1 else Registro1 = 0

Descripción:

FEQ toma dos operandos interpretados como floats y comprueba si son iguales. Guarda el resultado booleano en el primer operando, que siempre es un registro.

---

14 Instrucción FNE (Float Not Equal)

Estructura y variantes:

(Variante 1): FNE { Registro1 }, { ValorInmediato }
(Variante 2): FNE { Registro1 }, { Registro2 }

Acciones de procesado:

(Variante 1): if Registro1 != ValorInmediato then Registro1 = 1 else Registro1 = 0
(Variante 2): if Registro1 != Registro2 then Registro1 = 1 else Registro1 = 0

Descripción:

FNE toma dos operandos interpretados como floats y comprueba si son distintos. Guarda el resultado booleano en el primer operando, que siempre es un registro.

---

15 Instrucción FGT (Float Greater Than)

Estructura y variantes:

(Variante 1): FGT { Registro1 }, { ValorInmediato }
(Variante 2): FGT { Registro1 }, { Registro2 }

Acciones de procesado:

(Variante 1): if Registro1 > ValorInmediato then Registro1 = 1 else Registro1 = 0
(Variante 2): if Registro1 > Registro2 then Registro1 = 1 else Registro1 = 0

Descripción:

FGT toma dos operandos interpretados como floats y comprueba si el primero es mayor que el segundo. Guarda el resultado booleano en el primer operando, que siempre es un registro.

---

16 Instrucción FGE (Float Greater or Equal)

Estructura y variantes:

(Variante 1): FGE { Registro1 }, { ValorInmediato }
(Variante 2): FGE { Registro1 }, { Registro2 }

Acciones de procesado:

(Variante 1): if Registro1 >= ValorInmediato then Registro1 = 1 else Registro1 = 0
(Variante 2): if Registro1 >= Registro2 then Registro1 = 1 else Registro1 = 0

Descripción:

FGE toma dos operandos interpretados como floats y comprueba si el primero es mayor o igual que el segundo. Guarda el resultado booleano en el primer operando, que siempre es un registro.

---

17 Instrucción FLT (Float Less Than)

Estructura y variantes:

(Variante 1): FLT { Registro1 }, { ValorInmediato }
(Variante 2): FLT { Registro1 }, { Registro2 }

Acciones de procesado:

(Variante 1): if Registro1 < ValorInmediato then Registro1 = 1 else Registro1 = 0
(Variante 2): if Registro1 < Registro2 then Registro1 = 1 else Registro1 = 0

Descripción:

FLT toma dos operandos interpretados como floats y comprueba si el primero es menor que el segundo. Guarda el resultado booleano en el primer operando, que siempre es un registro.

18 Instrucción FLE (Float Less or Equal)

Estructura y variantes:

(Variante 1): FLE { Registro1 }, { ValorInmediato }
(Variante 2): FLE { Registro1 }, { Registro2 }

Acciones de procesado:

(Variante 1): if Registro1 <= ValorInmediato then Registro1 = 1 else Registro1 = 0
(Variante 2): if Registro1 <= Registro2 then Registro1 = 1 else Registro1 = 0

Descripción:

FLE toma dos operandos interpretados como floats y comprueba si el primero es menor o igual que el segundo. Guarda el resultado booleano en el primer operando, que siempre es un registro.

19 Instrucción MOV (Move)

Estructura y variantes:

(Variante 1): MOV { Registro1 }, { ValorInmediato }
(Variante 2): MOV { Registro1 }, { Registro2 }
(Variante 3): MOV { Registro1 }, [{ ValorInmediato }]
(Variante 4): MOV { Registro1 }, [{ Registro2 }]
(Variante 5): MOV { Registro1 }, [{ Registro2 } + { ValorInmediato }]
(Variante 6): MOV [{ ValorInmediato }], { Registro2 }
(Variante 7): MOV [{ Registro1 }], { Registro2 }
(Variante 8): MOV [{ Registro1 } + { ValorInmediato }], { Registro2 }

Acciones de procesado:

(Variante 1): Registro1 = ValorInmediato
(Variante 2): Registro1 = Registro2
(Variante 3): Registro1 = Memoria[ValorInmediato]
(Variante 4): Registro1 = Memoria[Registro2]
(Variante 5): Registro1 = Memoria[Registro2 + ValorInmediato]
(Variante 6): Memoria[ValorInmediato] = Registro2
(Variante 7): Memoria[Registro1] = Registro2
(Variante 8): Memoria[Registro1 + ValorInmediato] = Registro2

Descripción:

MOV copia el valor indicado en su segundo operando en el registro o dirección de memoria indicada por su primer operando. MOV es la instrucción más compleja de procesar porque tiene que distinguir entre 8 modos de direccionamiento diferentes.

La instrucción especifica en su campo "Modo de direccionamiento" cuál de los 8 modos utilizar, interpretando los posibles valores del modo siguiente:

Modos de direccionamiento de MOV		
Valor	Destino	Fuente
0	Registro 1	Valor Inmediato
1	Registro 1	Registro 2
2	Registro 1	Memoria[Valor Inmediato]
3	Registro 1	Memoria[Registro 2]
4	Registro 1	Memoria[Registro 2 + Valor Inmediato]
5	Memoria[Valor Inmediato]	Registro 2
6	Memoria[Registro 1]	Registro 2
7	Memoria[Registro 1 + Valor Inmediato]	Registro 2

20 Instrucción LEA (Load Effective Address)

Estructura y variantes:

(Variante 1): LEA { Registro1 }, [{ Registro2 }]
(Variante 2): LEA { Registro1 }, [{ Registro2 } + { ValorInmediato }]

Acciones de procesado:

(Variante 1): Registro1 = Registro2
(Variante 2): Registro1 = Registro2 + ValorInmediato

Descripción:

LEA toma una dirección de memoria como segundo operando. Almacena esa dirección (no su contenido) en el registro dado como primer operando. El caso más útil es cuando la dirección es de la forma puntero + offset, ya que la suma se realiza automáticamente.

21 Instrucción PUSH

Estructura y variantes:

PUSH { Registro1 }

Acciones de procesado:

Pila.Guardar(Registro1)

Descripción:

PUSH usa la pila hardware de la CPU para guardar el valor que contiene el registro dado en lo más alto de la pila.

22 Instrucción POP

Estructura y variantes:

POP { Registro1 }

Acciones de procesado:

Registro1 = Pila.Sacar( )

Descripción:

POP usa la pila hardware de la CPU para extraer un valor de lo más alto de la pila y lo escribe en el registro dado.

23 Instrucción IN

Estructura y variantes:

IN { Registro1 }, { NumeroPuerto }

Acciones de procesado:

Registro1 = Puerto[NumeroPuerto]

Descripción:

IN usa el bus de control para leer de un puerto de E/S en otro chip y almacena el valor devuelto en el registro especificado. Esta petición de lectura puede producir efectos secundarios dependiendo del puerto especificado.

24 Instrucción OUT

Estructura y variantes:

(Variante 1): OUT { NumeroPuerto }, [{ ValorInmediato }]
(Variante 2): OUT { NumeroPuerto }, { Registro1 }

Acciones de procesado:

(Variante 1): Puerto[NumeroPuerto] = ValorInmediato
(Variante 2): Puerto[NumeroPuerto] = Registro1

Descripción:

OUT usa el bus de control para escribir el valor especificado en un puerto de E/S de otro chip. Esta solicitud de escritura puede producir efectos secundarios dependiendo del puerto especificado.

25 Instrucción MOVS (Move String)

Estructura y variantes:

MOVS

Acciones de procesado:

Memoria[DR] = Memoria[SR]
DR += 1
SR += 1
CR –= 1
if CR > 0 then PunteroInstruccion –= 1

Descripción:

MOVS copia un valor de la dirección de memoria apuntada por SR a la apuntada por DR (como en un supuesto MOV [DR], [SR]). Luego implementa un bucle local para repetirse hasta que el contador en CR llegue a 0, mientras trabaja en direcciones consecutivas.

Obsérvese que incluso si se llama con un valor de CR igual o inferior a cero, MOVS siempre ejecutará el bucle descrito al menos una vez.

Esta instrucción es la única forma que hay en la CPU de Vircon32 de poder copiar directamente valores entre 2 posiciones de memoria sin pasar por un registro.

26 Instrucción SETS (Set String)

Estructura y variantes:

SETS

Acciones de procesado:

Memoria[DR] = SR
DR += 1
CR –= 1
if CR > 0 then PunteroInstruccion –= 1

Descripción:

SETS copia el valor en SR a la dirección apuntada por DR (como en un MOV [DR], SR). A continuación, implementa un bucle local para repetirse hasta que el contador en CR llegue a 0, mientras escribe en direcciones consecutivas.

Obsérvese que incluso si se llama con un valor de CR igual o inferior a cero, SETS siempre ejecutará el bucle descrito al menos una vez.

27 Instrucción CMPS (Compare String)

Estructura y variantes:

CMPS { Registro1 }

Acciones de procesado:

Registro1 = Memoria[DR] – Memoria[SR]
if Registro1 != 0 then terminar procesado
DR += 1
SR += 1
CR –= 1
if CR > 0 then PunteroInstruccion –= 1

Descripción:

CMPS toma como referencia el valor en la dirección apuntada por DR y lo compara con el apuntado por SR, con una resta. A continuación implementa un bucle local para repetirse hasta que el contador en CR llegue a 0, mientras lee direcciones consecutivas.

El resultado de la comparación se almacenará en el registro especificado, y será cero cuando sean iguales, positivo cuando algún valor en [DR] haya sido mayor, y negativo cuando algún valor en [SR] haya sido mayor.

Obsérvese que incluso si se llama con un valor de CR igual o inferior a cero, CMPS siempre ejecutará el bucle descrito al menos una vez.

28 Instrucción CIF (Convert Integer to Float)

Estructura y variantes:

CIF { Registro1 }

Acciones de procesado:

Registro1 = (float)Registro1

Descripción:

CIF interpreta el registro especificado como un valor entero. A continuación, convierte ese valor a una representación float y almacena el resultado en el mismo registro.

Debe tenerse en cuenta que, por la precisión limitada de la representación float, valores suficientemente altos de un entero de 32 bits provocarán una pérdida de precisión cuando se representen como float.

29 Instrucción CFI (Convert Float to Integer)

Estructura y variantes:

CFI { Registro1 }

Acciones de procesado:

Registro1 = (entero)Registro1

Descripción:

CFI interpreta el registro especificado como un valor float. Luego convierte ese valor en una representación entera y almacena el resultado en el mismo registro. La conversión no se realiza mediante redondeo, sino truncando (se descarta la parte fraccionaria).

Debido al rango mucho mayor de la representación float, valores lo bastante altos de un float resultarán en una pérdida de precisión al representarse como un entero de 32 bits.

30 Instrucción CIB (Convert Integer to Boolean)

Estructura y variantes:

CIB { Registro1 }

Acciones de procesado:

if Registro1 != 0 then Registro1 = 1

Descripción:

CIB interpreta el registro especificado como un valor entero. A continuación, convierte ese valor a su representación booleana estándar y almacena el resultado en el mismo registro. Esto significa que todos los valores distintos de 0 se convertirán a 1.

31 Instrucción CFB (Convert Float to Boolean)

Estructura y variantes:

CFB { Registro1 }

Acciones de procesado:

if Registro1 != 0.0 then Registro1 = 1 else Registro1 = 0

Descripción:

CFB interpreta el registro especificado como un valor float. Luego convierte ese valor en 0 (para el valor float 0.0), o en 1 (para cualquier otro valor) y lo guarda en ese registro.

32 Instrucción NOT

Estructura y variantes:

NOT { Registro1 }

Acciones de procesado:

Registro1 = NOT Registro1

Descripción:

NOT realiza un 'not' binario invirtiendo todos los bits del registro especificado.

33 Instrucción AND

Estructura y variantes:

(Variante 1): AND { Registro1 }, { ValorInmediato }
(Variante 2): AND { Registro1 }, { Registro2 }

Acciones de procesado:

(Variante 1): Registro1 = Registro1 AND ValorInmediato
(Variante 2): Registro1 = Registro1 AND Registro2

Descripción:

AND realiza un 'and' binario entre cada par de bits respectivos de los 2 operandos especificados. El resultado se guarda en el primero de ellos, que siempre es un registro.

34 Instrucción OR

Estructura y variantes:

(Variante 1): OR { Registro1 }, { ValorInmediato }
(Variante 2): OR { Registro1 }, { Registro2 }

Acciones de procesado:

(Variante 1): Registro1 = Registro1 OR ValorInmediato
(Variante 2): Registro1 = Registro1 OR Registro2

Descripción:

OR realiza un 'or' binario entre cada par de bits respectivos de los 2 operandos especificados. El resultado se guarda en el primero de ellos, que siempre es un registro.

35 Instrucción XOR

Estructura y variantes:

(Variante 1): XOR { Registro1 }, { ValorInmediato }
(Variante 2): XOR { Registro1 }, { Registro2 }

Acciones de procesado:

(Variante 1): Registro1 = Registro1 XOR ValorInmediato
(Variante 2): Registro1 = Registro1 XOR Registro2

Descripción:

XOR realiza un 'or exclusivo' binario entre cada par de bits respectivos de los 2 operandos indicados. El resultado se guarda en el primero, que siempre es un registro.

36 Instrucción BNOT (Boolean NOT)

Estructura y variantes:

BNOT { Registro1 }

Acciones de procesado:

if Registro1 == 0 then Registro1 = 1 else Registro1 = 0

Descripción:

BNOT interpreta el registro indicado como un booleano y luego lo convierte en el valor booleano opuesto. Esto equivale a utilizar primero CIB y luego invertir el bit número 0.

37 Instrucción SHL (Shift Left)

Estructura y variantes:

(Variante 1): SHL { Registro1 }, { ValorInmediato }
(Variante 2): SHL { Registro1 }, { Registro2 }

Acciones de procesado:

(Variante 1): Registro1 = Registro1 << ValorInmediato
(Variante 2): Registro1 = Registro1 << Registro2

Descripción:

SHL desplaza hacia la izquierda los bits del registro indicado. El segundo operando es un número entero de posiciones. Desplazar 0 posiciones no tiene efectos, y con valores negativos se desplaza a la derecha. El desplazamiento es de tipo lógico: al desplazar a la

izquierda se añaden ceros como bits menos significativos. Hacia la derecha se añaden ceros como bits más significativos. Los desbordamientos se descartan en ambos casos.

38 Instrucción IADD (Integer Add)

Estructura y variantes:

(Variante 1): IADD { Registro1 }, { ValorInmediato }
(Variante 2): IADD { Registro1 }, { Registro2 }

Acciones de procesado:

(Variante 1): Registro1 += ValorInmediato
(Variante 2): Registro1 += Registro2

Descripción:

IADD interpreta ambos operandos como enteros y los suma. El resultado se guarda en el primer operando, que siempre es un registro. Los bits de desbordamiento se descartan.

39 Instrucción ISUB (Integer Subtract)

Estructura y variantes:

(Variante 1): ISUB { Registro1 }, { ValorInmediato }
(Variante 2): ISUB { Registro1 }, { Registro2 }

Acciones de procesado:

(Variante 1): Registro1 –= ValorInmediato
(Variante 2): Registro1 –= Registro2

Descripción:

ISUB interpreta ambos operandos como enteros y los resta. El resultado se guarda en el primer operando, que siempre es un registro. Los bits de desbordamiento se descartan.

40 Instrucción IMUL (Integer Multiply)

Estructura y variantes:

(Variante 1): IMUL { Registro1 }, { ValorInmediato }
(Variante 2): IMUL { Registro1 }, { Registro2 }

Acciones de procesado:

(Variante 1): Registro1 *= ValorInmediato
(Variante 2): Registro1 *= Registro2

Descripción:

IMUL interpreta ambos operandos como enteros y los multiplica. El resultado se guarda en el primer operando, que siempre es un registro. Los bits de desbordamiento se descartan.

---

41 Instrucción IDIV (Integer Divide)

Estructura y variantes:

(Variante 1): IDIV { Registro1 }, { ValorInmediato }
(Variante 2): IDIV { Registro1 }, { Registro2 }

Acciones de procesado:

(Variante 1): Registro1 /= ValorInmediato
(Variante 2): Registro1 /= Registro2

Descripción:

IDIV interpreta ambos operandos como enteros y los divide. El resultado se guarda en el primer operando, que siempre es un registro.

---

42 Instrucción IMOD (Integer Modulus)

Estructura y variantes:

(Variante 1): IMOD { Registro1 }, { ValorInmediato }
(Variante 2): IMOD { Registro1 }, { Registro2 }

Acciones de procesado:

(Variante 1): Registro1 = Registro1 mod ValorInmediato
(Variante 2): Registro1 = Registro1 mod Registro2

Descripción:

IMOD interpreta ambos operandos como enteros y los divide. El resto de esa división se guarda en el primer operando, que siempre es un registro.

43 Instrucción ISGN (Integer Sign change)

Estructura y variantes:

ISGN { Registro1 }

Acciones de procesado:

Registro1 = –Registro1

Descripción:

ISGN interpreta el registro operando como un entero e invierte su signo.

44 Instrucción IMIN (Integer Minimum)

Estructura y variantes:

(Variante 1): IMIN { Registro1 }, { ValorInmediato }
(Variante 2): IMIN { Registro1 }, { Registro2 }

Acciones de procesado:

(Variante 1): Registro1 = min(Registro1, ValorInmediato)
(Variante 2): Registro1 = min(Registro1, Registro2)

Descripción:

IMIN interpreta ambos operandos como enteros. Luego toma el mínimo de ambos valores y lo guarda en el primer operando, que siempre es un registro.

45 Instrucción IMAX (Integer Maximum)

Estructura y variantes:

(Variante 1): IMAX { Registro1 }, { ValorInmediato }
(Variante 2): IMAX { Registro1 }, { Registro2 }

Acciones de procesado:

(Variante 1): Registro1 = max(Registro1, ValorInmediato)
(Variante 2): Registro1 = max(Registro1, Registro2)

Descripción:

IMAX interpreta ambos operandos como enteros. Luego toma el máximo de ambos valores y lo guarda en el primer operando, que siempre es un registro.

46 Instrucción IABS (Integer Absolute value)

Estructura y variantes:

IABS { Registro1 }

Acciones de procesado:

Registro1 = abs(Registro1)

Descripción:

IABS interpreta el registro operando como un entero y toma su valor absoluto.

47 Instrucción FADD (Float Add)

Estructura y variantes:

(Variante 1): FADD { Registro1 }, { ValorInmediato }
(Variante 2): FADD { Registro1 }, { Registro2 }

Acciones de procesado:

(Variante 1): Registro1 += ValorInmediato
(Variante 2): Registro1 += Registro2

Descripción:

FADD interpreta ambos operandos como floats y los suma. El resultado se guarda en el primer operando, que siempre es un registro.

48 Instrucción FSUB (Float Subtract)

Estructura y variantes:

(Variante 1): FSUB { Registro1 }, { ValorInmediato }
(Variante 2): FSUB { Registro1 }, { Registro2 }

Acciones de procesado:

(Variante 1): Registro1 –= ValorInmediato
(Variante 2): Registro1 –= Registro2

Descripción:

FSUB interpreta ambos operandos como floats y los resta. El resultado se guarda en el primer operando, que siempre es un registro.

49 Instrucción FMUL (Float Multiply)

Estructura y variantes:

(Variante 1): FMUL { Registro1 }, { ValorInmediato }
(Variante 2): FMUL { Registro1 }, { Registro2 }

Acciones de procesado:

(Variante 1): Registro1 *= ValorInmediato
(Variante 2): Registro1 *= Registro2

Descripción:

FMUL interpreta ambos operandos como floats y los multiplica. El resultado se guarda en el primer operando, que siempre es un registro.

50 Instrucción FDIV (Float Divide)

Estructura y variantes:

(Variante 1): FDIV { Registro1 }, { ValorInmediato }
(Variante 2): FDIV { Registro1 }, { Registro2 }

Acciones de procesado:

(Variante 1): Registro1 /= ValorInmediato
(Variante 2): Registro1 /= Registro2

Descripción:

FDIV interpreta ambos operandos como floats y los divide. El resultado se guarda en el primer operando, que siempre es un registro.

51 Instrucción FMOD (Float Modulus)

Estructura y variantes:

(Variante 1): FMOD { Registro1 }, { ValorInmediato }
(Variante 2): FMOD { Registro1 }, { Registro2 }

Acciones de procesado:

(Variante 1): Registro1 = fmod(Registro1, ValorInmediato)
(Variante 2): Registro1 = fmod(Registro1, Registro2)

Descripción:

FMOD interpreta ambos operandos como floats y los divide. Luego calcula el resto de esa división cuando se descarta la parte fraccionaria del resultado y lo guarda en el primer operando, que siempre es un registro.

---

52 Instrucción FSGN (Float Sign change)

Estructura y variantes:

FSGN { Registro1 }

Acciones de procesado:

Registro1 = –Registro1

Descripción:

FSGN interpreta el registro operando como float e invierte su signo.

---

53 Instrucción FMIN (Float Minimum)

Estructura y variantes:

(Variante 1): FMIN { Registro1 }, { ValorInmediato }
(Variante 2): FMIN { Registro1 }, { Registro2 }

Acciones de procesado:

(Variante 1): Registro1 = min(Registro1, ValorInmediato)
(Variante 2): Registro1 = min(Registro1, Registro2)

Descripción:

FMIN interpreta ambos operandos como floats. Luego toma el mínimo de ambos valores y lo guarda en el primer operando, que siempre es un registro.

---

54 Instrucción FMAX (Float Maximum)

Estructura y variantes:

(Variante 1): FMAX { Registro1 }, { ValorInmediato }
(Variante 2): FMAX { Registro1 }, { Registro2 }

Acciones de procesado:

(Variante 1): Registro1 = max(Registro1, ValorInmediato)
(Variante 2): Registro1 = max(Registro1, Registro2)

Descripción:

FMAX interpreta ambos operandos como floats. Luego toma el máximo de ambos valores y lo guarda en el primer operando, que siempre es un registro.

---

55 Instrucción FABS (Float Absolute value)

Estructura y variantes:

FABS { Registro1 }

Acciones de procesado:

Registro1 = abs(Registro1)

Descripción:

FABS interpreta el registro operando como un float y toma su valor absoluto.

---

56 Instrucción FLR (Floor)

Estructura y variantes:

FLR { Registro1 }

Acciones de procesado:

Registro1 = floor(Registro1)

Descripción:

FLR interpreta el registro operando como un float y lo redondea hacia abajo a un valor entero. El resultado, sin embargo, no se convierte a un entero y sigue siendo un float.

---

57 Instrucción CEIL (Ceiling)

Estructura y variantes:

CEIL { Registro1 }

Acciones de procesado:

Registro1 = ceil(Registro1)

Descripción:

CEIL interpreta el registro operando como un float y lo redondea hacia arriba a un valor entero. El resultado, sin embargo, no se convierte a un entero y sigue siendo un float.

58 Instrucción ROUND

Estructura y variantes:

ROUND { Registro1 }

Acciones de procesado:

Registro1 = round(Registro1)

Descripción:

ROUND interpreta el registro operando como un float y lo redondea al valor entero más próximo. El resultado, sin embargo, no se convierte a un entero y sigue siendo un float.

59 Instrucción SIN (Sine)

Estructura y variantes:

SIN { Registro1 }

Acciones de procesado:

Registro1 = sin(Registro1)

Descripción:

SIN interpreta el registro operando como un float y calcula el seno de ese valor. La función seno interpretará su argumento en radianes.

60 Instrucción ACOS (Arc Cosine)

Estructura y variantes:

ACOS { Registro1 }

Acciones de procesado:

Registro1 = acos(Registro1)

Descripción:

ACOS interpreta el registro operando como un float y calcula el arco coseno de ese valor. El resultado se da en radianes, en el rango [0, pi].

61 Instrucción ATAN2 (Arc Tangent 2)

Estructura y variantes:

ATAN2 { Registro1 }, { Registro2 }

Acciones de procesado:

Registro1 = atan2(Registro1, Registro2)

Descripción:

ATAN2 interpreta ambos registros operandos como floats y calcula el ángulo de un vector tal que Vx = Registro2 y Vy = Registro1. El resultado se almacena en el primer registro operando y se dará en radianes, en el rango [-pi, pi]. El origen de los ángulos se sitúa en (Vx > 0, Vy = 0) y los ángulos crecen al girar hacia (Vx = 0, Vy > 0).

62 Instrucción LOG (Logarithm)

Estructura y variantes:

LOG { Registro1 }

Acciones de procesado:

Registro1 = log(Registro1)

Descripción:

LOG interpreta el registro operando como un float y calcula el logaritmo base e de ese valor.

63 Instrucción POW (Power)

Estructura y variantes:

POW { Registro1 }, { Registro2 }

Acciones de procesado:

Registro1 = pow(Registro1, Registro2)

Descripción:

POW interpreta ambos registros operandos como floats y calcula el resultado de elevar el primer operando a la potencia del segundo operando. El resultado se almacena en el registro del primer operando.

11 Detección de errores hardware

La ejecución de un programa puede encontrarse con situaciones que el hardware no puede procesar, como dividir por cero. En Vircon32 esas situaciones se dan siempre como consecuencia de acciones de la CPU. Por eso el procesador debe ser el componente encargado de detectar las situaciones de error y disparar el mecanismo de error hardware.

En las siguientes subsecciones se enumeran todas las situaciones en las que puede darse un error de hardware, y se explica qué criterios usará la CPU para detectar cada error.

11.1 Errores de bus

Son los posibles errores que pueden producirse cuando la CPU hace una petición a uno de los 2 buses de comunicación. Su detección por parte de la CPU es automática, ya que el propio bus responderá a la petición con un indicador de éxito/fracaso.

Lectura de memoria no válida

Ha fallado una petición de lectura al bus de memoria.

Escritura en memoria no válida

Ha fallado una petición de escritura al bus de memoria.

Lectura de puerto no válida

Ha fallado una petición de lectura al bus de control.

Escritura en puerto no válida

Ha fallado una petición de escritura al bus de control.

11.2 Errores de pila

En su operación normal, el puntero a pila sólo debe apuntar a posiciones de dirección válidas dentro de la memoria RAM. Ese rango está entre 0x00000000 y 0x003FFFFF. Errores de programa o una pila de llamadas anormalmente grande pueden causar situaciones en las que la pila no puede seguir funcionando, y la CPU disparará un error de hardware como respuesta. Puede haber estas 2 situaciones de error:

Pila desbordada

Después de una operación de guardado, si el puntero a pila se vuelve negativo, se considera que se ha producido un desbordamiento de la pila. En este punto, la pila ha crecido hasta el final de la memoria RAM y no puede seguir creciendo.

Pila agotada

Después de sacar un valor, si el puntero a pila se hace mayor que su valor inicial de 0x003FFFFF (última palabra en RAM) se considera que la pila se ha desbordado. En este punto, la pila no puede seguir disminuyendo, ya que se ha agotado.

11.3 Errores matemáticos

En algunas instrucciones de la CPU que realizan operaciones matemáticas, existe una causa común de error cuando los argumentos recibidos quedan fuera del dominio de la función. Cuando se detecta este tipo de situación, se aborta el procesado de la instrucción para activar el error correspondiente. Los casos que se pueden encontrar en los que una función matemática no está definida son los siguientes:

Error de división

Este error puede ocurrir en 4 instrucciones diferentes: división y módulo, tanto en su versión entera como float. Ocurre cuando el segundo argumento (el divisor) es cero.

Error de arco coseno

Esta instrucción no puede procesarse si su argumento no está entre -1.0 y +1.0 (ambos extremos están incluidos en el rango válido).

Error de arco tangente 2

Esta instrucción no puede procesarse si ambos argumentos recibidos son cero.

Error de logaritmo

Esta instrucción no puede procesarse si su argumento recibido es menor o igual a cero.

Error de potencia

Esta instrucción no puede procesarse en el caso de que el primer argumento (base) sea negativo y el segundo (exponente) no tenga un valor entero.

12 Procesado de errores hardware

Cuando la CPU detecta una de las situaciones descritas en la sección anterior necesitará disparar el correspondiente procesado de error hardware. Las últimas etapas del procesado de errores las realizará la rutina de manejo de errores de la BIOS, pero antes de poder llamar a esa rutina la CPU necesita preparar cierta información para ella.

Los preparativos necesarios para que la BIOS procese un error consisten en identificar el tipo de error con un código y guardar los registros internos.

12.1 Códigos de error

Cuando se produce un error hardware, se representa con un código numérico el tipo de situación de error detectada. Los posibles códigos de error se enumeran en esta tabla:

Códigos de errores hardware	
Código	Tipo de error
0	Lectura de memoria no válida
1	Escritura en memoria no válida
2	Lectura de puerto no válida
3	Escritura en puerto no válida
4	Pila desbordada
5	Pila agotada
6	Error de división
7	Error de arco coseno
8	Error de arco tangente 2
9	Error de logaritmo
10	Error de potencia

12.2 Respuesta de la CPU a un error

Cuando la CPU detecta un error hardware, realiza las siguientes acciones:

- Escribe el código de error en R0.
- Copia el Puntero a Instrucción en R1.
- Copia el Registro de Instrucción en R2.
- Copia el Valor Inmediato en R3.
- Apunta BP y SP a la última posición de RAM (es decir, la dirección 0x003FFFFF).
- Apunta el Puntero de Instrucción a 0x10000000 (es decir, la dirección de la rutina de manejo de errores de la BIOS).
- Reanuda la ejecución normal desde esa dirección.

Después la CPU seguirá ejecutando la rutina de errores de la BIOS hasta que termine sus tareas, que son mostrar la información del error en pantalla y detener la ejecución.

Obsérvese que se reinicia la pila, descartando cualquier contexto de llamada previo. Esto es necesario, ya que el error puede haber sido causado por la propia pila. Además, el contexto previo será innecesario de todas formas cuando la BIOS obtenga el control.

(Fin de la parte 3)