Emulador 88110

Departamento de Arquitectura y Tecnología de Sistemas Informáticos
Facultad de Informática
Universidad Politécnica de Madrid

Índice general

Capítulo 1

Emulador e88110

1.1. Introducción

El microprocesador MC88110 es un procesador RISC superescalar que se encuadra dentro de la familia 88000 de Motorola. Es capaz de ejecutar hasta dos instrucciones en cada ciclo de reloj respetando el orden secuencial del programa a través del mecanismo de *pipeline* del secuenciador. El despacho de instrucciones se hace hacia diez unidades funcionales que trabajan en paralelo.

La memoria cache de datos y la memoria cache de instrucciones están separadas y son accesibles simultáneamente, teniendo cada una un tamaño de 8Kbytes.

La búsqueda y despacho de instrucciones se realiza en el secuenciador, el cual lee instrucciones de memoria, comprueba la disponibilidad de recursos y las posibles interdependencias entre instrucciones, dirige el flujo de operandos entre los bancos de registros y las unidades funcionales, despacha las instrucciones hacia las unidades funcionales y redirige el resultado de la ejecución de vuelta a los bancos de registros.

Las diez unidades funcionales que lo componen son las siguientes (figura 1.1):

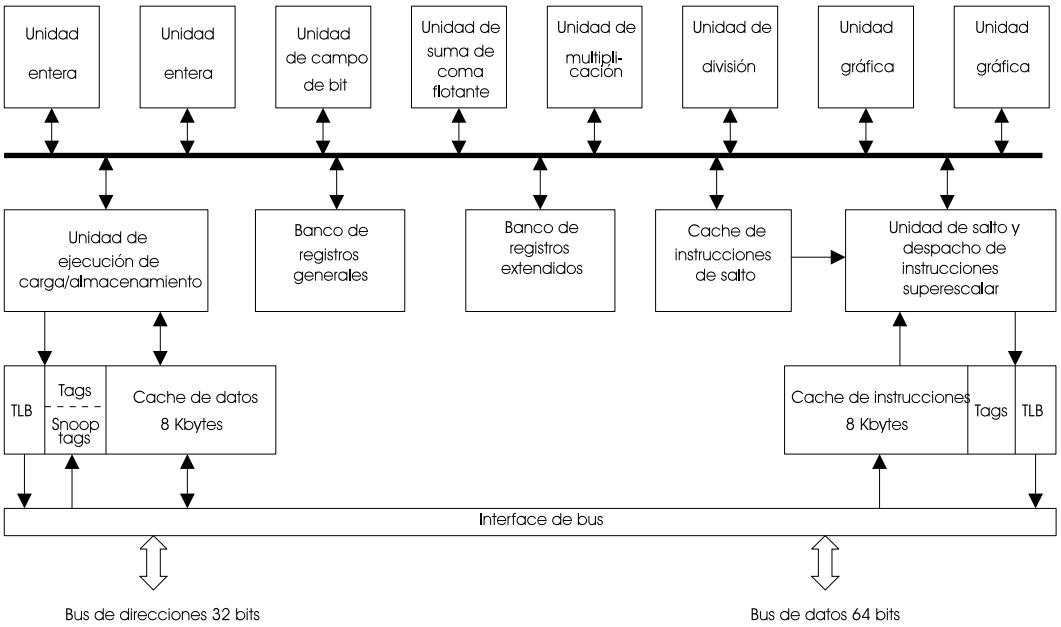


Figura 1.1. Diagrama de Bloques del microprocesador 88110

- **Unidades enteras:** Son unidades aritmético–lógicas que aceptan instrucciones lógicas y aritméticas de coma fija. La aritmética utilizada es complemento a 2.

- **Unidad de campo de bit:** Maneja instrucciones de manipulación de bits.

- **Unidad de multiplicación:** Ejecuta todas las instrucciones de multiplicación, tanto de coma fija como de coma flotante.

- **Unidad de suma de coma flotante:** Ejecuta instrucciones de suma, resta y comparación de coma flotante y conversiones entre enteros y números en coma flotante.

- **Unidad de división:** Realiza todas las instrucciones de división enteras y en coma flotante.

- **Unidades gráficas:** Ejecuta instrucciones referentes a gráficos, una maneja las instrucciones de empaquetamiento y la otra las operaciones aritméticas.

- **Unidad de salto:** Se encarga de la ejecución de todas las instrucciones que alteran el flujo del programa. Se sirve de una estación de reserva donde una instrucción de salto puede estar esperando, sin comenzar su ejecución, hasta que su operando esté disponible.

- **Unidad de carga/almacenamiento:** Ejecuta todas las instrucciones de transferencia de datos entre el sistema de memoria y los bancos de registros.

Los bancos de registros de que dispone son:

- **Banco general de registros:** tiene 32 registros de 32 bits que son accesibles de forma individual para realizar operaciones en precisión simple o por pares (en las instrucciones que utilizan doble precisión), de manera que se pueden almacenar operandos de 64 bits. De estos 32 registros hay dos que tienen un propósito específico, el registro cero (**r0**) contiene siempre la constante cero y el registro uno (**r1**) que se utiliza para guardar la dirección de retorno en las llamadas a subrutinas.

- **Banco extendido de registros:** tiene 32 registros de 80 bits que se usan únicamente por las instrucciones de coma flotante y soportan los formatos de precisión simple, doble y extendida del estándar IEEE–754. El registro cero contiene la constante +0.0E00.

Además de los registros anteriores, el MC88110 tiene el contador de programa (PC) y una serie de registros de control, entre los que cabe destacar el registro de estado del procesador (PSR).

Dada la finalidad académica de este emulador, no corresponde a una réplica exacta del MC88110 sino que constituye una herramienta que permita al alumno familiarizarse con el uso de un ensamblador en distintos niveles de complejidad: **serie** y **superescalar**.

No se han emulado todas las instrucciones del microprocesador real. Se han suprimido las instrucciones gráficas así como las unidades funcionales correspondientes. Se han descartado las instrucciones de excepciones. No se incluyen algunas opciones de las instrucciones de carga y almacenamiento.

Las operaciones de coma flotante que utilizan precisión extendida no se han incluido en este emulador, y por tanto no es necesario incluir el banco de registros extendido.

Se ha suprimido la cache de instrucciones de salto, el registro de control del emulador se ha simplificado respecto al original y se ha prescindido de los registros de control de coma flotante.

En el modo de ejecución serie el procesador actúa como un simple microprocesador secuencial que ejecuta instrucciones una a una sin *pipeline*. El modo paralelo es el modo de ejecución normal del 88110 en el que puede despachar hasta dos instrucciones cada ciclo de reloj, con el que se consigue el pleno funcionamiento del pipeline.

1.1.1. Memoria principal

La memoria principal es donde se almacenan todas las instrucciones y datos que el emulador necesita para su funcionamiento. La memoria se direcciona **a nivel de byte**, es decir, a cada byte se le asigna una dirección de memoria distinta. Las memorias cache del emulador intercambian bloques con la memoria principal a través de transacciones de 64 bits sobre el bus.

Puede accederse a palabras (32 bits) y dobles palabras (64 bits). El tamaño teórico de la memoria principal es de 4 GBytes, pero únicamente se han implementado 2^{18} direcciones, es decir, el rango de direcciones válidas son $[0, 0x3ffff]$.

1.1.2. Secuenciador

Puede trabajar en modo superescalar, emitiendo un máximo de dos instrucciones por ciclo de reloj, y en modo serie, en el que la búsqueda de una instrucción no comienza hasta que la anterior ha completado su ejecución.

1.1.3. Banco de registros

Dispone de 32 registros de 32 bits cada uno, estando el registro **r0** siempre con el valor cero y nunca se puede modificar su valor aunque **r0** sea destino del resultado de una instrucción. En el registro **r1** se guarda la dirección de retorno en las llamadas a subrutinas, no existiendo punteros de pila ni de marco de pila, debiendo reservar el programador los registros oportunos si desea funcionar con pila. Estos registros son accesibles por pares, de forma que se pueden utilizar operandos de 64 bits, por ejemplo, la instrucción

```
fadd.dsd r10, r15, r19
```

suma el número en coma flotante en precisión simple contenido en el registro 15 con el de doble precisión contenido en los registros 19 y 20, dejando el resultado, expresado en doble precisión, en los registros 10 y 11.

El registro de control de la máquina, de 32 bits, tiene el siguiente significado:

- Bit 0, indica si las excepciones están inhibidas o no. Un valor de 1 indica que están inhibidas.
- Bits 10 y 11, especifican el modo de redondeo en operaciones de coma flotante correspondiendo el valor 00 redondeo al más cercano, 01 redondeo a cero, 10 redondeo a $-\infty$ y 11 redondeo a $+\infty$
- Bit 28, es el bit de acarreo.
- Bit 30, indica el modo de ordenamiento de bytes en memoria. Si es 0 el modo es *big endian*, si es 1 es *little endian*.

  Para el caso en que se trabaje en modo serie:

- Bit 12, indica si se ha producido overflow en una operación entera. Un valor de 1 indica que se ha producido overflow. Este flag no aparece originalmente en el registro de control, puesto que se genera una excepción entera en caso de producirse desbordamiento. Este flag se ha incluido para detectar condiciones de desbordamiento entera en caso de que se ejecute el emulador con las excepciones inhibidas.
- Bit 13, un valor de 1 indica que se ha producido una división entera por cero.
- El resto de bits no se utilizan.

1.1.4. Excepciones

Las excepciones que pueden producirse son las derivadas de las instrucciones enteras y de coma flotante. Las excepciones enteras pueden ser: división por cero y overflow. Las excepciones de coma flotante son: operando reservado, división por cero, underflow y overflow. El emulador, al igual que el 88110, implementa un mecanismo de excepciones precisas presentando, con ayuda del almacén histórico de instrucciones, el estado correcto a la rutina de tratamiento de excepción.

1.1.5. Unidades funcionales

El funcionamiento de las dos unidades enteras, la unidad de manipulación de bits, la de multiplicación, la de suma en coma flotante y la de división es similar al de las unidades funcionales del 88110 y el tiempo de ejecución coincide con el original. Las unidades que tratan instrucciones de coma flotante siguen el estándar IEEE-754.

En la unidad de salto se ha prescindido de la cache de instrucciones de direcciones de salto, pero el secuenciador es capaz de proporcionar la dirección de salto de forma inmediata con lo que las prestaciones se igualan o mejoran. Igual que la unidad original del 88110, acepta instrucciones de salto aunque sus operandos no estén disponibles y ejecuta también saltos retardados. Los saltos incondicionales puede llegar a ejecutarlos en un ciclo de reloj y no provocan burbujas en el pipeline si sus operandos están disponibles. Los saltos condicionales, con predicción estática, también se ejecutan como mínimo en un ciclo si las condiciones son favorables.

La unidad de carga/almacenamiento mantiene almacenes separados para las instrucciones de carga y almacenamiento. Puede cargar hasta cuatro instrucciones de carga y tres de almacenamiento. No permite el reordenamiento de instrucciones por lo que el acceso se realiza en estricto orden de entrada a la unidad funcional. Cuando la política de escritura de la cache de datos es *copy-back* permite instrucciones con la opción *write-through with allocate* para actualizar incondicionalmente la memoria principal.

1.1.6. Cache de datos

La memoria cache de datos se ha diseñado de forma que pueda soportar distintas configuraciones. Puede especificarse el tiempo de acceso, la política de ubicación (asociativa, asociativa por conjuntos o directa), la política de escritura (*copy-back*, *write-through*), el número de bloques o líneas, el número de bytes por línea y en caso de ubicación asociativa por conjuntos, el número de líneas por conjunto. Todos estos parámetros pueden cambiarse para conseguir el sistema que se desee. A todo esto debe añadirse la posibilidad de desactivar la cache y acceder directamente a la memoria principal. El acceso a la memoria cache puede hacerse por palabras o dobles palabras. Cuando se quiere leer una doble palabra y estas caen en líneas distintas, la cache necesita hacer dos accesos distintos (de la misma forma que la cache del 88110).

Cuando se produce un fallo en la cache, el proceso de cambio de bloque desde memoria principal se realiza de forma automática y realizando peticiones a la memoria principal mediante accesos de dobles palabras. La primera palabra a traer cuando se produce un fallo es justamente aquella que lo provocó con lo que se optimiza el tiempo de respuesta de la memoria cache. El número de bytes mínimo permitido en cada línea es de ocho bytes y siempre debe ser múltiplo de ocho (una doble palabra).

1.1.7. Cache de instrucciones

Para la cache de instrucciones pueden aplicarse las mismas explicaciones que para la cache de datos, teniendo presente que la cache de instrucciones no permite operaciones de escritura. Por lo demás, los mismos parámetros configurables en la cache de datos son también configurables en la cache de instrucciones, exceptuando la política de escritura. Esta cache también permite desactivarse a voluntad.

1.1.8. Interfaz de usuario

El emulador también dispone de un sencillo mecanismo de comunicación con el usuario. La interfaz presenta un indicador en el que el usuario va introduciendo mandatos para avanzar en la ejecución del programa y ver el estado interno de la máquina. Es posible incluir puntos de ruptura para detener la ejecución. Los mandatos pueden ser diferentes cuando estamos en modo de ejecución serie o en modo de ejecución superescalar. Son los siguientes:

- `H`.- Presenta una ayuda. Lista todos los mandatos.
- `Q`.- Termina la simulación y vuelve al sistema operativo soporte.
- `P [+|- <esp_direccion>]`.- Si el mandato es `p` sin más, lista todos los puntos de ruptura activos. Si es `p + 24`, por ejemplo, incluye un punto de ruptura en la dirección 24 (expresada en decimal). Si es `p - 0x14`, desactiva el punto de ruptura de la dirección 14 (expresada en hexadecimal). El parámetro `esp_direccion` puede especificar una dirección (expresada en decimal o hexadecimal) o una etiqueta. Por ejemplo, si la etiqueta `BUCLE` está asociada a la dirección 0x1000, el comando `p + BUCLE` es equivalente a `p + 0x1000`.
- `D <esp_direccion> [<rango>]`.- Presenta un grupo de instrucciones en ensamblador a partir de la dirección especificada. El parámetro `esp_direccion` puede especificar una dirección de memoria o una etiqueta. Si se especifica el parámetro opcional `rango` (en decimal o hexadecimal) se desensamblan tantas instrucciones como se especifique en dicho parámetro. Por ejemplo el mandato `d 0x100 32` desensambla 32 instrucciones desde la dirección 100 (expresada en hexadecimal). Si no se especifica el parámetro `rango` se desensamblarán 15 instrucciones.
- `V [C] <esp_direccion> [<rango>]`.- Si no está presente al argumento `C`, se visualiza el contenido de tantas posiciones de memoria principal como indique el parámetro `rango` a partir de la dirección indicada. El parámetro `esp_direccion` puede especificar una dirección de memoria o una etiqueta. Los datos se muestran en hexadecimal y en palabras de 32 bits. Al igual que en el mandato `D` tanto la dirección como el `rango` pueden ir expresados en decimal o hexadecimal. El parámetro `rango` es opcional y si no se especifica se muestran 20 palabras de memoria.

  Si se pasa el parámetro opcional `C`, el comando muestra posiciones de la cache de datos (si está activada). Si el contenido de una palabra de la cache es un conjunto de espacios en blanco indicará que dicha palabra no está presente en la memoria cache.
- `E`.- Ejecuta el programa a partir de la dirección actual del contador de programa. La ejecución no se detendrá hasta que se encuentre un punto de ruptura o la instrucción `stop`.
- `T [<veces>]`.- Este mandato tiene significado distinto en modo de Ejecución serie y modo de ejecución superescalar. En ambos modos si se omite `veces` se supone uno. Si se está ejecutando en modo serie ejecutará `veces` instrucciones completas. En modo de ejecución superescalar ejecutará `veces` ciclos de reloj.
- `R [<numero> <valor>]`.- Presenta el contenido de los registros y estadísticas de acceso a las caches de datos e instrucciones si están activadas. Si este mandato va seguido de un registro del banco y un valor (decimal o hexadecimal) se modifica dicho registro con ese valor. Por ejemplo el mandato `r 12 0x4a` carga el valor hexadecimal 4a en el registro 12 del banco.
- `I <esp_direccion> <valor>`.- Modifica el contenido de una palabra de memoria principal `esp_direccion` (en decimal o hexadecimal) con el valor que le sigue `valor` (en decimal o hexadecimal). El parámetro `esp_direccion` puede especificar una dirección de memoria o una etiqueta.
- `B`.- Este mandato sólo funciona en modo superescalar y presenta el contenido del almacén histórico de instrucciones.

- `S`.- Este mandato presenta las estadísticas de acceso a las caches de instrucciones y datos (si están activadas).
- `C`.- Este mandato presenta los parámetros de configuración del computador emulado.

Para facilitar el uso de la interfaz de usuario del emulador se ha incorporado un histórico de mandatos que permite acceder a los mandatos tecleados anteriormente, editarlos y ejecutarlos de nuevo mediante las teclas de movimiento de cursor. Las particularidades de manejo que incorpora son las siguientes:

- La pulsación de la tecla ↑ muestra en la línea de mandatos el último mandato que se ha ejecutado. Si se pulsa sucesivamente esta tecla se va recorriendo el histórico de mandatos hacia atrás y pulsando ↓ se avanza en sentido contrario.
- La pulsación de las teclas ← y → permiten moverse a lo largo de la línea de mandato para modificarlo.
- La pulsación de la tecla `^D` permite eliminar el carácter situado debajo del cursor.
- La pulsación de la tecla `^E` avanza el cursor al final de la línea.
- La pulsación de la tecla `^A` sitúa el cursor al principio de la línea.
- La introducción del mandato vacío ("return") provocará que se vuelva a ejecutar el último mandato.

Esta interfaz, aunque simple, permite hacer el seguimiento de la ejecución de cualquier programa.

1.2. Programa Ensamblador

El programa ensamblador permite traducir instrucciones de ensamblador a instrucciones de máquina del 88110. Por tanto, a partir de un fichero fuente escrito en el ensamblador del microprocesador, se generará un fichero binario ejecutable por el emulador. Las instrucciones que admite el ensamblador desarrollado para este emulador son:

- **Instrucciones:** Son instrucciones del ensamblador que se traducen a una instrucción máquina del 88110.
- **Pseudoinstrucciones:** Son instrucciones del ensamblador que no se traducen en el código binario. Son órdenes que indican cómo se debe generar el código binario.

Las instrucciones que admite el ensamblador están exhaustivamente descritas en el capítulo 2. Las pseudoinstrucciones que se han implementado en este prototipo son las que se especifican en el estándar IEEE 694.

1.2.1. Pseudoinstrucciones

- **org:** Especifica en qué posiciones de memoria se ubicarán las variables o el código que aparece a continuación. Se pueden especificar tantas pseudoinstrucciones **org** como se desee, pero no pueden especificar zonas de código solapadas. Por ejemplo el ensamblado del siguiente fragmento de código generaría un error:

```
org 144
and.c   r1,     r2,     r3
and     r1,     r2,     r3
and.c   r1,     r2,     r3
and     r1,     r2,     3
org 140
add     r1,     r2,     3
bsr     1
stop
```

Obsérvese que a partir de la dirección 144 se incluyen 4 instrucciones, es decir, el rango ocupado es desde la dirección 144 hasta la dirección 159 (ambas inclusive). El segundo rango de direcciones ocupadas comprendería desde la dirección 140 hasta la dirección 151. Los dos rangos tienen una parte solapada. Un intento de ensamblar este programa (`ej`) se realizaría ejecutando el comando `88110e -e 144 -o ej.bin ej` y generaría los siguientes mensajes:

```
Compilando ej ...
Compiladas 10 lineas
Generando Codigo...
ERROR: Hay direcciones solapadas alrededor de la posicion 145
No pude generar programa
```

- **data:** Reserva e inicializa un conjunto de palabras en memoria que carga con un conjunto de datos especificados a continuación. La sintaxis de la pseudoinstrucción sigue dos posibles formatos:

  Formato 1: `data a, b, c, ...`

  Formato 2: `data "cadena de caracteres"`

  En el formato 1 `a, b` y `c` pueden ser de la forma:

  - 0xn: Es un valor expresado en hexadecimal.
  - n: Es un número positivo expresado en decimal.
  - –n: Es un número negativo expresado en decimal.

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

```
org 140
data -1, 2 , -9,  -1, 0, 0x80000000
```

  Como resultado se reservaría el rango de direcciones desde la 140 hasta la 163 (ambas inclusive) y se cargarían al inicio de la ejecución del programa con los valores especificados en dicha pseudoinstrucción. Si ejecutamos el emulador `mc88110` con el fichero binario generado por el programa ensamblador se obtendrán los siguientes resultados:

```
88110 > v 140
      128                                          FFFFFFFF
      144       02000000    F7FFFFFF    FFFFFFFF   00000000
      160       00000080    00000000    00000000   00000000
      176       00000000    00000000    00000000   00000000
      192       00000000    00000000    00000000   00000000
88110 >
```

  Como se puede observar en el ejemplo, aunque la configuración del emulador permite seleccionar el modo de trabajo del ordenamiento de bytes dentro de una palabra, se ha configurado para que trabaje en modo *little endian* (byte menos significativo en la dirección más baja).

  En el formato 2 la pseudoinstrucción data se utiliza para inicializar una zona de memoria con una cadena de caracteres, de tal forma que cada uno de los caracteres incluido en la cadena ocupará una dirección de la memoria del emulador. Para incluir caracteres especiales es necesario que vayan precedidos por el carácter `'\'` que indica que éste y el siguiente especifican un carácter. Se han definido los siguientes caracteres especiales:

  - `\"`: Indica que es un carácter comilla doble (carácter con código ASCII decimal 34).

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

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

```
org 100
data "1234\n"
data 3
```

Como resultado se cargarán los caracteres con códigos hexadecimales 0x31, 0x32, 0x33 y 0x34 en las posiciones de memoria 100, 101, 102 y 103 respectivamente. El carácter 0x0a se cargará en la posición 104. Obsérvese que como el número de caracteres reservado con esta pseudoinstrucción no es múltiplo de 4, la pseudoinstrucción data que le sigue almacenará el valor 3 en la dirección 108, puesto que la pseudoinstrucción data asociada a palabras exige que la dirección esté alineada a palabra.

- **res:** Reserva un conjunto de bytes en memoria que debe ser múltiplo de palabra. A diferencia de la pseudoinstrucción **data** las posiciones de memoria reservadas se cargan con un valor indefinido. A continuación se muestra un ejemplo del uso de esta pseudoinstrucción:

```
org 140
res 12
```

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

1.2.2. Llamada al programa Ensamblador

La sintaxis para invocar al programa Ensamblador es la siguiente:

```
88110e [-e <punto de entrada>] [-m{l|b}] -o <fichero de salida> <fichero fuente>
```

- **Opción -o:** Va seguida de un nombre de fichero. En este fichero se almacenará el programa binario generado por el programa ensamblador. Es una opción que debe aparecer obligatoriamente en la orden de invocación del programa ensamblador.
- **Opción -e:** Va seguida de un número entero o una cadena de caracteres. Si lo que sigue a la opción es un número decimal, éste indica el valor que se debe cargar en el contador de programa después de cargar el fichero en memoria. Si a esta opción sigue una cadena de caracteres, ésta debe ser una etiqueta válida en el fichero ensamblador que se desea ensamblar. La dirección representada por dicha etiqueta se cargará en el contador de programa después de cargar el fichero en memoria. Si no se especifica un punto de entrada se entenderá que se comienza la ejecución en la dirección 0 de memoria principal.
- **Opción -m:** Va seguida de una `l` o una `b`. Es opcional e indica cómo se deben generar las constantes que aparecen en sentencias `data`. La opción `l` indica que genere las constantes de dichas sentencias para el modo *little endian*. Si aparece una `b` generará dichas constantes en modo *big endian*. Si esta opción no aparece en la orden, se supone que se genera en modo *little endian*.

Por ejemplo la invocación del programa que aparece a continuación genera código en modo *little-endian* para el programa ensamblador que está almacenado en el fichero `practica.ens`, genera la salida en código máquina en el fichero `practica.bin` y se cargará el contador de programa con el valor 500 cuando se cargue dicho fichero con el emulador.

```
88110e -e 500 -ml -o practica.bin practica.ens
```

1.2.3. Macros

Al igual que en cualquier lenguaje de programación, en ensamblador se presentan situaciones que exigen que un conjunto de instrucciones se repitan a lo largo de un programa. Este conjunto de instrucciones suele ser reducido y en la mayor parte de los casos no merece la pena su encapsulado en una subrutina.

Una macro es un conjunto de sentencias a las que se asigna un nombre y un conjunto de argumentos. La sintaxis de una macro se presenta a continuación:

```
nombre_de_macro: MACRO(arg1, arg2, ..., argn)
                 conjunto de instrucciones
                 que componen la macro
ENDMACRO
```

nombre_de_macro es una etiqueta que debe aparecer obligatoriamente y por la que se conoce la macro. El conjunto de parámetros arg_i se utiliza para la construcción de la macro y serán sustituidos por el programa ensamblador por los parámetros utilizados en cada invocación de la macro. A modo de ejemplo se muestra la macro **swap** a la que se le pasarán dos registros. Esta macro intercambia sus contenidos utilizando un registro intermedio (**r1**). El código de la macro es el siguiente:

```
swap:   MACRO(ra,rb)
        or r1, ra,ra
        or ra,rb,rb
        or rb,r1,r1
ENDMACRO
```

En algún punto del código del programa se puede invocar esta macro para intercambiar el contenido de los registros **r5** y **r8**. mediante la sentencia:

```
swap(r5,r8)
```

A diferencia de las subrutinas, las macros se resuelven en fase de ensamblado. Cada invocación de una macro provoca su "expansión" en fase de ensamblado, es decir, el programa ensamblador sustituye cada invocación de la macro por las instrucciones que lo componen, sustituyendo los parámetros que aparecen en la declaración de la macro por los parámetros reales pasados en la invocación. La expansión de la macro anterior consistirá en incluir en el lugar donde se invoca la macro el siguiente conjunto de instrucciones:

```
or r1, r5,r5
or r5,r8,r8
or r8,r1,r1
```

Este proceso de expansión se repite para cada llamada a la macro. Una invocación a una macro exige que la macro esté definida previamente en el código.

Obsérvese que queda como responsabilidad del código que llama a la macro el asegurarse que los registros de almacenamiento intermedio que utiliza la macro (en este caso el registro **r1**) no contienen ningún dato útil.

En la utilización de las macros en el ensamblador `88110e` hay que tener en cuenta tres puntos:

- Una macro debe haber sido previamente definida antes de realizar cualquier invocación a la misma.
- Se permite la invocación de una macro dentro de la definición de otra, siempre que haya sido definida previamente.
- No se permite la definición de etiquetas dentro de una macro. Nótese que la expansión de la macro generaría a lo largo del programa tantas etiquetas del mismo nombre como invocaciones se hicieran. Esto generaría un error en la fase de ensamblado. Si se desea realizar bifurcaciones dentro de la macro hay que utilizar bifurcaciones con direccionamiento relativo al PC y expresar dicho desplazamiento en la instrucción.

1.3. Modos de Direccionamiento

El MC88110 tiene los modos de direccionamiento que enumeramos a continuación, incluyendo la nomenclatura contemplada por el estándar IEEE-694.

- **Direccionamiento inmediato.** El operando es de 16 bits y puede ser con o sin signo

  Ejemplo: `add rD,rS1,SIMM16`
  Descripción: rD ← rS1+SIMM16
  Nomenclatura estándar: `ADD .D,.S1, #SIMM16`

- **Direccionamiento de registro.** El campo de dirección de la instrucción indica el registro seleccionado. En el anterior ejemplo puede contemplarse este direccionamiento.

- **Direccionamiento relativo a contador de programa.** El desplazamiento se aplica al contenido del PC que apunta a la propia instrucción de salto. Los desplazamientos son de 16 bits en saltos condicionales y de 26 bits en los incondicionales.

  Ejemplo: `br D26`
  Descripción: PC ← PC+4*D26
  Nomenclatura estándar: `BR $dir`, siendo dir=PC+4*D26

- **Direccionamiento indirecto de registro.** El registro queda especificado en el correspondiente campo de dirección de la instrucción.

  Ejemplo: `jmp (rS2)`
  Descripción: PC ← rS2 (los dos bits menos significativos de rS2 a cero)
  Nomenclatura estándar: `JMP [.S2]`

- **Direccionamiento relativo a registro base.** El desplazamiento sobre el registro base puede ser un inmediato de 16 bits o el contenido de otro registro.

  Ejemplo: `ld rD,rS1,SI16`
  Descripción: rD ← (rS1+SI16)
  Nomenclatura estándar: `LD .D, #SI16[.S1]`
  Ejemplo: `ld rD,rS1,rS2`
  Descripción: rD ← (rS1+rS2)
  Nomenclatura estándar: `LD .D,[.S1,.S2]`

- **Direccionamiento de bit.** Algunas instrucciones contienen información para especificar un bit determinado de una palabra o un campo de bits de una palabra.

  Ejemplo: `clr rD,rS1,W5<O5>`
  Descripción: rD ← rS1 con los bits O5 a O5+W5-1 a cero
  Nomenclatura estándar: `CLR.IW5 .D,.S1,O5`
  Ejemplo: `bb0 B5,rS1,D16`
  Descripción: PC ← PC+4*D16 si el bit B5 de rS1 es cero
  Nomenclatura estándar: `BZ.I $dir,.S1,B5` siendo dir=PC+4*D16

1.4. Sintaxis del lenguaje ensamblador

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

```
Etiqueta: Mnemónico Operandos ;Comentarios
```

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

- **Etiqueta:** Es una cadena de caracteres que comienza por un carácter alfabético seguido por un conjunto de caracteres alfanuméricos. Este campo es sensible a las mayúsculas. Esto significa que las etiquetas `LISTA`, `Lista`, `LiStA` y `lista` son todas distintas. Esta cadena de caracteres deberá ir seguida del carácter :.

- **Mnemónico:** Es una cadena de caracteres que representa la operación que realiza la instrucción. En las tablas adjuntas se muestran los mnemónicos admitidos en este ensamblador y en capítulo 2 se hace una descripción exhaustiva de todas las instrucciones consideradas en este ensamblador. Este campo también es sensible a las mayúsculas. La instrucción `and` está admitida dentro del juego de instrucciones, pero no lo está la instrucción `AND`.

- **Operandos:** Es la lista de operandos sobre los que trabaja la instrucción. Estos operandos van separados por comas. Dependiendo de la instrucción se admitirán los diferentes modos de direccionamiento que se han explicado en la sección 1.3. En el caso de que el modo de direccionamiento involucre una etiqueta, se deberá tener en cuenta que el nombrado de las mismas es sensible a las mayúsculas. El nombrado de los registros tiene la misma particularidad: el registro `r12` existe en el banco de registros del emulador, pero no existe el registro `R12`. Al igual que en los mnemónicos, en el capítulo 2 se hace una descripción exhaustiva de los operandos y modos de direccionamiento que utiliza cada instrucción en particular.

- **Comentarios:** Es una cadena de cualquier tipo de caracteres precedida por el carácter `;`. Cuando el programa ensamblador encuentra el carácter `;` ignora el resto de caracteres hasta el final de la línea. Este campo para un programador novel puede parecer innecesario, pero es muy útil para incorporar aclaraciones en lenguaje natural a las instrucciones que componen el programa. Estos comentarios adicionales, ignorados por el ensamblador, se utilizarán por los programadores para hacer el programa más fácilmente mantenible en el futuro. Es altamente aconsejable la utilización de este campo especialmente en programas que no son triviales.

1.5. Ejemplo

Supongamos que se desea escribir un programa en lenguaje ensamblador, en el 88110 emulado, que suma **n** números representados en binario sin signo. Las variables sobre las que trabaja el programa son: **N** contiene el número de elementos enteros que tiene que sumar y debe ser distinto de 0; a partir de la etiqueta **SUMDOS** están almacenados de forma contigua todos los elementos que hay que sumar; en la variable **REST** se debe almacenar el resultado de la suma. Supongamos que la configuración del emulador se ha establecido con las siguientes características:

- Ordenamiento de bytes *little-endian*.

- Ejecución serie.

- Excepciones inhibidas.

El programa que resuelve este problema es el que se especifica en la figura 1.2. Los números que aparecen entre paréntesis al comienzo de cada línea se han incluido para hacer referencia a cada una de las instrucciones del programa y, por tanto, **no deben** aparecer en el programa fuente ensamblador. Este programa se debe crear utilizando cualquier editor de texto disponible en el sistema.

En este programa tiene especial relevancia la instrucción `(18)` (instrucción de salto condicional), que es equivalente a `bb1 4,r4,BUC`. El programa ensamblador permite especificar la condición de salto bien por el mnemónico de la condición usado en la instrucción `cmp` (véase la página 29), o bien por el bit del registro involucrado en la condición.

```
( 1)      org 0
( 2)ppal: or r20,r0,low(N)
( 3)      or.u r20,r20,high(N)        ;Se carga en r20 la direccion de N.
( 4)      or r21,r0,low(SUMDOS)
( 5)      or.u r21,r21,high(SUMDOS)   ;Se carga en r21 la direccion de los
( 6)      or r22,r0,low(REST)         ;datos y en r22 la direccion del
( 7)      or.u r22,r22,high(REST)     ;resultado.
( 8)      ld r5,r0,r20                ;Se inicializa el contador del bucle.
( 9)      xor r10,r10,r10             ;Se pone a cero r10.
(10)             ;INICIO DEL BUCLE.
(11)BUC:  ld r11,r0,r21               ;Se carga r11 con el dato a sumar.
(12)      add r21,r21,4               ;r21 apunta al siguiente dato.
(13)      addu.co r10,r10,r11         ;Suma nuevo dato al acumulado.
(14)      ldcr r4                     ;Carga r4 con el registro de control.
(15)      bb1 28,r4,OVF               ;Si hay acarreo hay desbordamiento.
(16)      subu r5,r5,1                ;Se decrementa el contador.
(17)      cmp r4,r5,0
(18)      bb1 gt,r4,BUC               ;Si r5 es mayor que cero se bifurca a BUC.
(29)             ;FIN DEL BUCLE.
(20)      st r10,r0,r22               ;Se almacena el resultado
(21)      add r30,r0,0                ;indicando que no ha habido error
(22)      br FIN                      ;y se salta a FIN.
(23)OVF:  add r30,r0,-1               ;Se indica que ha habido desbordamiento.
(24)FIN:  stop
(25)      org 1000
(26)N:         data 5
(27)SUMDOS:    data 0x80000000,24,0x2fffffff,0,3
(28)REST:      data 0
```

Figura 1.2. Suma de un número variable de enteros sin signo.

1.5.1. Ensamblado

Una vez creado el programa se debe crear su código máquina correspondiente. Esto se hace ensamblando el código con el Ensamblador proporcionado para la realización de la práctica (**88110e**). Teniendo en cuenta las restricciones que se han especificado para este ejemplo, el mandato a ejecutar sería el siguiente:

```
88110e -e0 -ml -o suman.bin suman.ens
```

Si el programa se ha tecleado tal y como aparece en este texto aparecerá el siguiente mensaje:

```
Compilando suman.ens ...
Compiladas 28 lineas
Generando Codigo...
Programa generado correctamente
```

suman.ens es el nombre de fichero que contiene el programa ensamblador. A partir de este fichero se crea un fichero (**suman.bin**) que contiene el código máquina generado. Este nombre de fichero debe seguir a la opción **-o** de la orden por la que se invoca el programa. La opción **-e0** indica que el punto de entrada del programa se sitúa en la posición 0 de memoria, es decir, cuando se arranque el emulador el valor que contendrá el PC será el 0. La opción **-ml** indica el ordenamiento de bytes que se está utilizando dentro de una palabra (en este caso es *little-endian*). Como se observa en el ejemplo el nombre del fichero que contiene el programa ensamblador debe especificarse el último en la orden (**suman.ens**).

Errores más comunes

Hasta este momento se ha expuesto el caso de que el programa se haya tecleado sin errores. Esta situación no es la habitual. Supongamos que se cambia la línea `2` por:

```
( 2)     or r20,r0,N
```

Se vuelve a teclear el mandato:

```
88110e -e0 -ml -o suman.bin suman.ens
```

En este caso el mensaje que aparece es el siguiente:

```
Compilando suman.ens ...
ERROR: Linea: 2 - parse error
```

parse error significa que el Ensamblador ha encontrado una línea que no sigue la sintaxis especificada en el manual. Si consultamos el manual de la instrucción `( 2)` tenemos que asegurarnos que la sentencia ensamblador de dicha línea sigue la sintaxis correcta. En definitiva se deben asegurar los siguientes aspectos:

- Si en la instrucción existe etiqueta, ésta está correctamente formada y va seguida del carácter `:`.
- El mnemónico de la instrucción existe en el juego de instrucciones. Por ejemplo la instrucción `OR` no está admitida en el juego. La que si lo está es la instrucción `or`.
- Los operandos están separados por comas (`,`).
- Los modos de direccionamiento de cada uno de los operandos están admitidos para la instrucción que se especifica. En nuestro ejemplo la instrucción or admite únicamente dos posibilidades:

  ```
  or rD,rS1,rS2          or rD,rS1,IMM16
  ```

  La primera de ellas realiza una operación or con tres registros (un registro destino, rD, y dos registros fuentes) y la segunda opera con dos registros y un dato inmediato de 16 bits. Puesto que la instrucción especificada no sigue ninguna de las dos posibilidades, el error está en el modo de direccionamiento que se ha especificado en el tercer operando. Un operando de 16 bits se puede especificar de forma numérica con un valor decimal o hexadecimal, o mediante los operadores `high` y `low`. El problema está en que no se ha especificado uno de estos operadores en el tercer operando.

Supongamos ahora que el programa varía en la instrucción `( 2)` que pasa a tomar el siguiente valor:

```
( 2)     or r20,r0,low(n)
```

De la misma forma que antes se teclea el mismo mandato. El mensaje que aparecerá en pantalla es el siguiente:

```
Compilando suman.ens ...
Compiladas 28 lineas
Linea 28 (3): El simbolo no esta definido (n)
Programa no generado
```

En este caso el mensaje es mucho más claro que el el caso anterior. La variable `n` no está definida en el programa pero sí lo está la variable `N`. La sintaxis del ensamblador especifica que el nombrado de símbolos es sensible a las mayúsculas y minúsculas y, por tanto, las variables `n` y `N` se tomarán como dos variables distintas.

1.5.2. Utilización del emulador mc88110

Supongamos ahora que se ha ensamblado con éxito el programa **suman.ens**. Como resultado se habrá generado un programa en código máquina **suman.bin** que es el que hay que pasar como parámetro al programa emulador **mc88110**. El emulador se invoca mediante el siguiente mandato:

```
mc88110 suman.bin
```

Entrada en el emulador

Los primeros mensajes que aparecen al comenzar la ejecución del programa son los que se muestran en la figura 1.3.

```
PC=0          or       r20,r00,1000    Tot. Inst: 0    << Ciclo : 0
FL=1 FE=1 FC=0 FV=0 FR=0
R01 = 00000000 h R02 = 00000000 h R03 = 00000000 h R04 = 00000000 h
R05 = 00000000 h R06 = 00000000 h R07 = 00000000 h R08 = 00000000 h
R09 = 00000000 h R10 = 00000000 h R11 = 00000000 h R12 = 00000000 h
R13 = 00000000 h R14 = 00000000 h R15 = 00000000 h R16 = 00000000 h
R17 = 00000000 h R18 = 00000000 h R19 = 00000000 h R20 = 00000000 h
R21 = 00000000 h R22 = 00000000 h R23 = 00000000 h R24 = 00000000 h
R25 = 00000000 h R26 = 00000000 h R27 = 00000000 h R28 = 00000000 h
R29 = 00000000 h R30 = 00000000 h R31 = 00000000 h
88110 >
```

Figura 1.3. Inicio de la ejecución del emulador.

La primera línea muestra la siguiente instrucción a ejecutar. Puesto que se acaba de arrancar el emulador, la siguiente instrucción a ejecutar es la que está ubicada en el punto de entrada del programa (en este caso la dirección 0). En nuestro ejemplo esta dirección contiene la instrucción `or r20,r00,1000` que corresponde a la instrucción `( 1)` de **suman.ens**. Además se muestra el valor del contador de programa (`PC` antes de ejecutar la instrucción). En este caso el PC contiene el valor del punto de entrada al programa. Por último en esta primera línea se muestra el número de ciclos de CPU consumidos por el programa.

La segunda línea contiene los valores de los bits del registro de estado más importantes:

- **FL=1** indica que el ordenamiento de los bytes de una palabra en memoria es *little-endian*.
- **FE=1** indica que las excepciones están inhibidas.
- **FC=0** indica que el flag de acarreo contiene el valor 0.
- **FV=0** indica que el flag de desbordamiento contiene el valor 0.
- **FR=0** indica que los dos bits que indican el modo de redondeo contienen el valor 0 (al más cercano). Las posibles alternativas a este valor son: 1 especifica redondeo hacia 0; 2 especifica redondeo hacia $-\infty$; y 3 hacia $+\infty$.

A continuación se muestra un conjunto de registros (en el ejemplo que nos ocupa se muestran todos) con los valores que contienen (en hexadecimal). La cadena de caracteres 88110 > es el *prompt* del emulador. Esto significa que el programa está esperando que se le proporcionen mandatos para ejecutar. Cada uno de estos mandatos se compone de una letra seguida de sus parámetros. A continuación se incluyen algunos ejemplos de los mandatos más utilizados y cuándo se deben utilizar.

```
 PC=24         ld       r05,r00,r20     Tot. Inst: 6   << Ciclo : 90
 FL=1 FE=1 FC=0 FV=0 FR=0
 R01 = 00000000 h R02 = 00000000 h R03 = 00000000 h R04 = 00000000 h
 R05 = 00000000 h R06 = 00000000 h R07 = 00000000 h R08 = 00000000 h
 R09 = 00000000 h R10 = 00000000 h R11 = 00000000 h R12 = 00000000 h
 R13 = 00000000 h R14 = 00000000 h R15 = 00000000 h R16 = 00000000 h
 R17 = 00000000 h R18 = 00000000 h R19 = 00000000 h R20 = 000003E8 h
 R21 = 000003EC h R22 = 00000400 h R23 = 00000000 h R24 = 00000000 h
 R25 = 00000000 h R26 = 00000000 h R27 = 00000000 h R28 = 00000000 h
 R29 = 00000000 h R30 = 00000000 h R31 = 00000000 h
88110 >
```

Figura 1.4.

```
 PC=28         xor      r10,r10,r10     Tot. Inst: 7  << Ciclo : 116
 FL=1 FE=1 FC=0 FV=0 FR=0
 R01 = 00000000 h R02 = 00000000 h R03 = 00000000 h R04 = 00000000 h
 R05 = 00000005 h R06 = 00000000 h R07 = 00000000 h R08 = 00000000 h
 R09 = 00000000 h R10 = 00000000 h R11 = 00000000 h R12 = 00000000 h
 R13 = 00000000 h R14 = 00000000 h R15 = 00000000 h R16 = 00000000 h
 R17 = 00000000 h R18 = 00000000 h R19 = 00000000 h R20 = 000003E8 h
 R21 = 000003EC h R22 = 00000400 h R23 = 00000000 h R24 = 00000000 h
 R25 = 00000000 h R26 = 00000000 h R27 = 00000000 h R28 = 00000000 h
 R29 = 00000000 h R30 = 00000000 h R31 = 00000000 h
88110 > v 1000
       992                                 05000000     00000080
      1008       18000000     FFFFFF2F     00000000     03000000
      1024       00000000     00000000     00000000     00000000
      1040       00000000     00000000     00000000     00000000
      1056       00000000     00000000     00000000     00000000
88110 >
```

Figura 1.5.

Ejecución paso a paso

El mandato de ejecución en modo traza (ejecución paso a paso) permite ejecutar una sola instrucción de tal forma que se pueden apreciar los datos sobre los que se opera antes y después de su ejecución. Supongamos la situación que se muestra en la figura 1.4. Esta figura muestra la situación del programa que aparece en la figura 1.2 inmediatamente antes de ejecutar la instrucción `( 8)`.

Obsérvese que se han inicializado los registros `r20` con la dirección de memoria ocupada por la variable `N` (1000), `r21` se carga con la dirección ocupada por el conjunto de números enteros representados por la etiqueta `SUMDOS` (1004) y r22 con la dirección ocupada por la variable `REST` (1024). La instrucción almacenada en la posición 24 de memoria (mostrada en la figura 1.4) carga el registro `r5` con el contenido de la posición de memoria contenida en el registro `r20` (variable `N`). Si en el *prompt* del emulador se teclea el mandato `t`, ejecutará la instrucción `ld r05,r00,r20`.

Al finalizar dicha ejecución aparecerán los mensajes que se incluyen en la figura 1.5. El registro `r5` se ha cargado con el valor 5 que es el que está contenido en la posición de memoria 1000. Para visualizar el contenido de posiciones de memoria utilizamos el mandato `v`. `v 1000` muestra el contenido de la posición de memoria 1000 y siguientes (véase la figura 1.5).

```
88110 > d 0
ppal:       0      or      r20,r00,1000
            4      or.u    r20,r20,0
            8      or      r21,r00,1004
           12      or.u    r21,r21,0
           16      or      r22,r00,1024
           20      or.u    r22,r22,0           88110 > d 56 8
           24      ld      r05,r00,r20                 56      cmp     r04,r05,0
           28      xor     r10,r10,r10                 60      bb1     04,r04,-7
BUC:       32      ld      r11,r00,r21                 64      st      r10,r00,r22
           36      add     r21,r21,4                   68      add     r30,r00,0
           40      addu.co r10,r10,r11                 72      br      2
           44      ldcr    r04                 OVF:    76      add     r30,r00,65535
           48      bb1     28,r04,7            FIN:    80      stop
           52      subu    r05,r05,1                   84      instruccion incorrecta
           56      cmp     r04,r05,0           88110 >
```

Figura 1.6.

```
PC=0          or       r20,r00,1000    Tot. Inst: 0    << Ciclo : 0
FL=1 FE=1 FC=0 FV=0 FR=0
R01 = 00000000 h R02 = 00000000 h R03 = 00000000 h R04 = 00000000 h
R05 = 00000000 h R06 = 00000000 h R07 = 00000000 h R08 = 00000000 h
R09 = 00000000 h R10 = 00000000 h R11 = 00000000 h R12 = 00000000 h
R13 = 00000000 h R14 = 00000000 h R15 = 00000000 h R16 = 00000000 h
R17 = 00000000 h R18 = 00000000 h R19 = 00000000 h R20 = 00000000 h
R21 = 00000000 h R22 = 00000000 h R23 = 00000000 h R24 = 00000000 h
R25 = 00000000 h R26 = 00000000 h R27 = 00000000 h R28 = 00000000 h
R29 = 00000000 h R30 = 00000000 h R31 = 00000000 h
88110 > p + 60
88110 > e
        Alcanzado punto de ruptura, direccion 60
PC=60         bb1      04,r04,-7       Tot. Inst: 15 << Ciclo : 248
FL=1 FE=1 FC=0 FV=0 FR=0
R01 = 00000000 h R02 = 00000000 h R03 = 00000000 h R04 = 00005998 h
R05 = 00000004 h R06 = 00000000 h R07 = 00000000 h R08 = 00000000 h
R09 = 00000000 h R10 = 80000000 h R11 = 80000000 h R12 = 00000000 h
R13 = 00000000 h R14 = 00000000 h R15 = 00000000 h R16 = 00000000 h
R17 = 00000000 h R18 = 00000000 h R19 = 00000000 h R20 = 000003E8 h
R21 = 000003F0 h R22 = 00000400 h R23 = 00000000 h R24 = 00000000 h
R25 = 00000000 h R26 = 00000000 h R27 = 00000000 h R28 = 00000000 h
R29 = 00000000 h R30 = 00000000 h R31 = 00000000 h
88110 >
```

Figura 1.7.

Establecimiento de puntos de ruptura

Hasta este punto se ha mostrado cómo se puede depurar un programa paso a paso. El principal problema es que la ejecución de un programa puede ser muy larga y esta forma de "ejecución controlada"puede llegar a ser muy lenta. El caso más habitual es que se haya cometido un error en el programa, pero se tenga acotado el momento en que se produce. Esta situación se ilustrará con un ejemplo.

Supongamos que en el programa de la figura 1.2 se sabe que hay un error, pero no se sabe exactamente en qué situación se produce, es decir, sabemos que se ha producido un error, pero no conocemos el momento

exacto en que se produce. En esta situación una de las formas más rápidas de encontrar el error es comprobar los datos parciales que se generan en cada iteración y decidir si son consistentes con los datos de entrada. Para ello debemos poder tomar control del depurador/ensamblador en cada iteración, pero no se desea ejecutar paso a paso (utilizando el mandato `t`). La forma razonable de depurar este supuesto es ejecutar sin que el simulador devuelva control hasta la instrucción `(18)`.

Un punto de ruptura consiste en especificar una condición que si se genera en una instrucción del programa, el emulador deja de ejecutar instrucciones y devuelve control al usuario para que introduzca nuevos mandatos. El emulador únicamente acepta puntos de ruptura de tipo *fetch*. En este tipo de puntos de ruptura se especifica una dirección de memoria y cuando el emulador lee de memoria la instrucción que ocupa dicha dirección devuelve el control al usuario. En la instrucción `18` pondremos un punto de ruptura (mandato `p`), de tal forma que cuando se alcance dicha instrucción el emulador deje de ejecutar el programa y devuelva control al usuario. El mandato `p` recibe como parámetro la dirección de la instrucción en la que queremos parar la ejecución. Para conocer dicha dirección desensamblamos el código (mandato `d`).

```
88110 > r 12 0x13
88110 > r
 PC=60         bb1      04,r04,-7       Tot. Inst: 15 << Ciclo : 248
 FL=1 FE=1 FC=0 FV=0 FR=0
 R01 = 00000000 h R02 = 00000000 h R03 = 00000000 h R04 = 00005998 h
 R05 = 00000004 h R06 = 00000000 h R07 = 00000000 h R08 = 00000000 h
 R09 = 00000000 h R10 = 80000000 h R11 = 80000000 h R12 = 00000013 h
 R13 = 00000000 h R14 = 00000000 h R15 = 00000000 h R16 = 00000000 h
 R17 = 00000000 h R18 = 00000000 h R19 = 00000000 h R20 = 000003E8 h
 R21 = 000003F0 h R22 = 00000400 h R23 = 00000000 h R24 = 00000000 h
 R25 = 00000000 h R26 = 00000000 h R27 = 00000000 h R28 = 00000000 h
 R29 = 00000000 h R30 = 00000000 h R31 = 00000000 h
88110 > i 2000 0x18
88110 > v 2000 4
      2000        18000000      00000000      00000000      00000000
88110 >
```

Figura 1.8.

El primer mandato (`d 0`) de la figura 1.6 desensambla 15 instrucciones a partir de la posición de memoria 0. La instrucción `(18)` no está entre esas 15 y hay que volver a pedir el desensamblado de las instrucciones que están almacenadas a partir de la dirección 56. Observamos que la instrucción `(18)` ocupa la posición de memoria 60 y es sobre esa dirección sobre la que hay que establecer el punto de ruptura. En la figura 1.6 se puede observar que cuando el emulador encuentra que el formato de instrucción no corresponde a ninguna instrucción admitida presenta el mensaje `instruccion incorrecta`.

Establecemos el punto de ruptura en la instrucción deseada (`p + 60`), y ejecutamos el programa (`e`) hasta que se alcance dicha instrucción. Los mandatos y mensajes que muestra el emulador se presentan en la figura 1.7. Al finalizar de ejecutar el mandato `e` se muestra la instrucción `bb1 04,r04,-7` antes de su ejecución. Puesto que tenemos control sobre el emulador podemos visualizar el contenido de los registros, comprobar que las direcciones de memoria contienen el valor correcto o, en general, teclear cualquier mandato que nos permita comprobar que la ejecución del programa ha sido correcta. Si no lo fuera, podemos modificar en el mismo emulador posiciones de memoria (mandato `i`) o registros.

La figura 1.8 muestra los mensajes que proporciona el emulador al modificar el registro 12 con el valor hexadecimal 13 y la posición de memoria 2000 con el valor hexadecimal 18. La modificación de esta posición de memoria sigue las reglas de ordenación de bytes en una palabra de memoria que especifique el registro de control. En este caso es *little-endian*.

Por último, si se desea eliminar el punto de ruptura y ejecutar hasta el final del programa se deben teclear los siguientes mandatos:

```
p - 60
```

```
e
```

La figura 1.9 muestra los mensajes que se generan al quitar el punto de ruptura, continuar la ejecución hasta el final del programa y salir del emulador.

1.6. Instrucciones

Clasificaremos las instrucciones en 6 grupos de acuerdo a su finalidad. Estas instrucciones se describirán de forma exhaustiva en el capítulo 2.

1.6.1. Instrucciones lógicas

Realizan las funciones lógicas **and**, **or**, **xor** con dos formatos de instrucción:

```
xor rD,rS1,rS2
xor rD,rS1,IMM16
```

En el primer formato se realiza la operación lógica (en el ejemplo **xor**) con los registros **rS1** y **rS2** y el resultado se almacena en el registro **rD**. En este formato las funciones lógicas se pueden realizar con la extensión **.c**, que complementa a 1 el operando rS2.

El segundo formato realiza la operación lógica con un dato inmediato de 16 bits **IMM16**. En este formato se pueden realizar las funciones lógicas con la extensión **.u**, indicando que la operación se realiza sobre los 16 bits superiores de **rS1**.

La instrucción **mask** utiliza el segundo de los formatos y es una operación **and** que deja a cero la parte no afectada de **rD**, ya que puede tener la extensión **.u**.

Los modos de direccionamiento son de registro e inmediato.

```
        Alcanzado punto de ruptura, direccion 60
 PC=60          bb1      04,r04,-7       Tot. Inst: 15 << Ciclo : 248
 FL=1 FE=1 FC=0 FV=0 FR=0
 R01 = 00000000 h R02 = 00000000 h R03 = 00000000 h R04 = 00005998 h
 R05 = 00000004 h R06 = 00000000 h R07 = 00000000 h R08 = 00000000 h
 R09 = 00000000 h R10 = 80000000 h R11 = 80000000 h R12 = 00000000 h
 R13 = 00000000 h R14 = 00000000 h R15 = 00000000 h R16 = 00000000 h
 R17 = 00000000 h R18 = 00000000 h R19 = 00000000 h R20 = 000003E8 h
 R21 = 000003F0 h R22 = 00000400 h R23 = 00000000 h R24 = 00000000 h
 R25 = 00000000 h R26 = 00000000 h R27 = 00000000 h R28 = 00000000 h
 R29 = 00000000 h R30 = 00000000 h R31 = 00000000 h
88110 > p - 60
88110 > e
        Fin ejecucion
 PC=84          instruccion incorrecta   Tot. Inst: 52  << Ciclo : 858
 FL=1 FE=1 FC=0 FV=0 FR=0
 R01 = 00000000 h R02 = 00000000 h R03 = 00000000 h R04 = 00005AA4 h
 R05 = 00000000 h R06 = 00000000 h R07 = 00000000 h R08 = 00000000 h
 R09 = 00000000 h R10 = B000001A h R11 = 00000003 h R12 = 00000000 h
 R13 = 00000000 h R14 = 00000000 h R15 = 00000000 h R16 = 00000000 h
 R17 = 00000000 h R18 = 00000000 h R19 = 00000000 h R20 = 000003E8 h
 R21 = 00000400 h R22 = 00000400 h R23 = 00000000 h R24 = 00000000 h
 R25 = 00000000 h R26 = 00000000 h R27 = 00000000 h R28 = 00000000 h
 R29 = 00000000 h R30 = 00000000 h R31 = 00000000 h
88110 > q
```

Figura 1.9.

1.6.2. Instrucciones aritméticas enteras

Realizan las operaciones **add** y **sub** (tanto con signo como sin signo) utilizan dos formatos de instrucción:

```
add rD,rS1,rS2
add rD,rS1,SIMM16
```

En el primer caso las operaciones pueden realizarse considerando el acarreo de entrada, el acarreo de salida, ninguno o ambos. Las operaciones con signo causan excepción de overflow.

Las operaciones **mul** y **div** (tanto con signo como sin signo) utilizan únicamente el primero de los formatos mostrados. En la multiplicación deja como resultado los 32 bits menos significativos del mismo. Las operaciones sin signo pueden realizarse en doble precisión. Las operaciones con signo causan excepción de overflow y las de división causan excepción de dividir por cero.

Los modos de direccionamiento son de registro e inmediato.

1.6.3. Instrucciones de campo de bit

Realizan las operaciones **clr**, **set**, **ext**, **extu** (sin signo), **mak** y **rot** (rotaciones a la derecha) sobre un campo de bits de un registro fuente, estableciendo un origen y una extensión para dicho campo de bits. El origen y extensión pueden estar definidos como operandos o como parte de un registro.

Los modos de direccionamiento son de registro y de bit.

1.6.4. Instrucciones de coma flotante

Realizan las operaciones en coma flotante **fadd**, **fsub**, **fmul** y **fdiv** que toman como operandos tres registros que pueden estar en simple o doble precisión. Causan excepciones de overflow, underflow, operando reservado y división por cero.

Las instrucciones **fcvt**, **flt** y **int** realizan el cambio de precisión en coma flotante y la conversión entre entero con signo y coma flotante (en precisión simple y doble) y viceversa. Utilizan como operandos dos registros.

Los modos de direccionamiento son exclusivamente de registro.

1.6.5. Instrucciones de control de flujo

Realizan bifurcaciones condicionales e incondicionales. Las condiciones observadas se refieren al valor de un bit del registro destino de una instrucción **cmp** (ver su descripción). El desplazamiento en el direccionamiento relativo se realiza sobre el contenido del PC que apunta a la propia instrucción de bifurcación. La extensión **.n** permite ejecutar la instrucción siguiente antes de efectuar el salto.

La llamada a subrutina se realiza almacenando la dirección de retorno en el registro **r1**.

Los modos de direccionamiento son relativo a PC e indirecto de registro.

1.6.6. Instrucciones de carga/almacenamiento

La dirección de carga/almacenamiento se obtiene por direccionamiento relativo a registro base, ya sea sumándole un valor inmediato o el contenido de otro registro. Los modos de direccionamiento son de registro y relativo a registro base. Se puede direccionar un byte con o sin signo, media palabra con o sin signo, una palabra y doble palabra.

A modo de ejemplo se muestra el siguiente programa que ejecuta un conjunto de instrucciones de carga/almacenamiento con distintos tamaños de objetos.

```
            org    0
            or     r10, r0, low(n)
            or.u   r10, r10, high(n)
            ld.b   r1, r10, 0
            ld.bu  r2, r10, 0
            ld.h   r3, r10, 0
            ld.hu  r4, r10, 0
            ld     r5, r10, 0
            ld.d   r6, r10, 0
            or     r20, r0, low(store)
            or.u   r20, r20, high(store)
            ld     r8, r10, 0
            ld     r9, r10, 4
            st.b   r8, r20, 0
            st.h   r8, r20, 4
            st     r8, r20, 8
            st.d   r8, r20, 12
            stop
            org    1000
n:          data   0xb0a09080, 15
            org    2000
store:      data   0, 0, 0, 0
```

Los valores iniciales de los registros y posiciones de memoria antes de ejecutar el programa son los que se especifican a continuación:

```
 PC=0          or       r10,r00,1000    Tot. Inst: 0    << Ciclo : 0
 FL=1 FE=1 FC=0 FV=0 FR=0
 R01 = 00000000 h R02 = 00000000 h R03 = 00000000 h R04 = 00000000 h
 R05 = 00000000 h R06 = 00000000 h R07 = 00000000 h R08 = 00000000 h
 R09 = 00000000 h R10 = 00000000 h R11 = 00000000 h R12 = 00000000 h
 R13 = 00000000 h R14 = 00000000 h R15 = 00000000 h R16 = 00000000 h
 R17 = 00000000 h R18 = 00000000 h R19 = 00000000 h R20 = 00000000 h
 R21 = 00000000 h R22 = 00000000 h R23 = 00000000 h R24 = 00000000 h
 R25 = 00000000 h R26 = 00000000 h R27 = 00000000 h R28 = 00000000 h
 R29 = 00000000 h R30 = 00000000 h R31 = 00000000 h
88110 > v 1000 6
       992                                 8090A0B0     0F000000
      1008       00000000     00000000     00000000     00000000
88110 >
```

Al final de la ejecución del programa los valores de registros y posiciones de memoria modificadas son:

```
88110 > e
        Fin ejecucion
 PC=68         instruccion incorrecta   Tot. Inst: 17 << Ciclo : 411
 FL=1 FE=1 FC=0 FV=0 FR=0
 R01 = FFFFFF80 h R02 = 00000080 h R03 = FFFF9080 h R04 = 00009080 h
 R05 = B0A09080 h R06 = 0000000F h R07 = B0A09080 h R08 = B0A09080 h
 R09 = 0000000F h R10 = 000003E8 h R11 = 00000000 h R12 = 00000000 h
 R13 = 00000000 h R14 = 00000000 h R15 = 00000000 h R16 = 00000000 h
 R17 = 00000000 h R18 = 00000000 h R19 = 00000000 h R20 = 000007D0 h
 R21 = 00000000 h R22 = 00000000 h R23 = 00000000 h R24 = 00000000 h
 R25 = 00000000 h R26 = 00000000 h R27 = 00000000 h R28 = 00000000 h
 R29 = 00000000 h R30 = 00000000 h R31 = 00000000 h
88110 > v 2000
      2000       80000000     80900000     8090A0B0     0F000000
      2016       8090A0B0     00000000     00000000     00000000
      2032       00000000     00000000     00000000     00000000
      2048       00000000     00000000     00000000     00000000
      2064       00000000     00000000     00000000     00000000
88110 >
```

Capítulo 2

Juego de Instrucciones del MC88110

A continuación se describe el juego de instrucciones del computador MC88110 que se han incluido en el emulador e88110.

2.1. Instrucciones lógicas

Instrucción and

Operación:

$$rD \leftarrow rS1 \text{ and } rS2$$

Sintaxis:

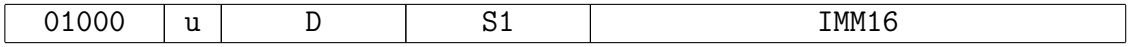
```
and     rD,rS1,rS2          and     rD,rS1,IMM16
and.c   rD,rS1,rS2          and.u   rD,rS1,IMM16
```

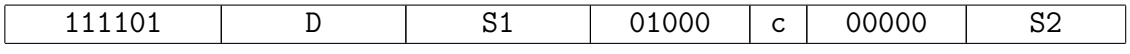
Descripción:

Para operación con tres registros, la instrucción **and** hace la operación lógica and del registro **S1** con el registro **S2** y el resultado lo almacena en el registro **rD**. Si se especifica la opción **.c**, el registro **S2** se complementa a 1 antes de hacer la operación.

Si la instrucción opera sobre un dato inmediato de 16 bits (**IMM16**) se hace la operación **and** con este dato y los 16 bits menos significativos del registro **S1**. Los 16 bits superiores de **S1** se llevan a **rD** sin ningún cambio. La opción **.u** especifica que la operación se realiza con los 16 bits superiores de **S1** y los menos significativos se llevan a **rD** sin cambios.

Codificación:





`u:`	0 - and IMM16 con bits 15-0 de S1
	1 - and IMM16 con bits 31-16 de S1
`D, S1, S2:`	registro destino, origen 1 y origen 2
`IMM16:`	operando inmediato sin signo de 16 bits
`c:`	0 - segundo registro no se complementa a 1
	1 - segundo registro se complementa a 1

Instrucción xor

Operación:

rD ← rS1 xor rS2

Sintaxis:

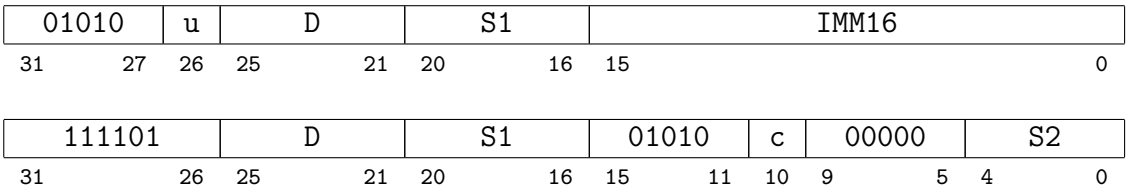
```
xor    rD,rS1,rS2                    xor    rD,rS1,IMM16
xor.c  rD,rS1,rS2                    xor.u  rD,rS1,IMM16
```

Descripción:

Para el formato de tres registros, la instrucción **xor** hace la operación lógica xor del registro **S1** con el registro **S2** y el resultado lo almacena en el registro **rD**. Si se especifica la opción **.c**, el registro **S2** se complementa a 1 antes de hacer la operación.

Si la instrucción opera sobre un dato inmediato de 16 bits (**IMM16**) se hace la operación **xor** con este dato y los 16 bits menos significativos del registro **S1**. Los 16 bits superiores de **S1** se llevan a **rD** sin ningún cambio. Si se especifica la opción **.u** la operación **xor** se hace con los 16 bits superiores de **S1** y los inferiores se llevan a **rD** sin cambios.

Codificación:



`u:`	0 - xor IMM16 con bits 15-0 de S1
	1 - xor IMM16 con bits 31-16 de S1
`D, S1, S2:`	registro destino, origen 1 y origen 2
`IMM16:`	operando inmediato sin signo de 16 bits
`c:`	0 - segundo registro no se complementa a 1
	1 - segundo registro se complementa a 1

Instrucción mask

Operación:

rD ← rS1 and IMM16

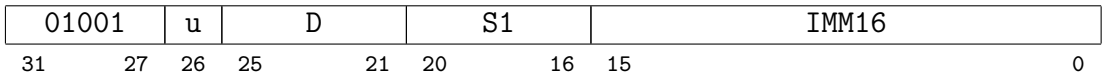
Sintaxis:

```
mask    rD,rS1,IMM16
mask.u  rD,rS1,IMM16
```

Descripción:

La instrucción **mask** hace la operación lógica and entre los 16 bits bits menos significativos de **rS1** y el valor inmediato sin signo especificado por **IMM16**. El resultado se almacena en los 16 bits inferiores del registro **rD**. Los 16 bits superiores de **rD** se ponen a cero. Si se especifica la opción **.u** la operación se hace con los 16 bits superiores de **rS1** y el resultado se salvaguarda en los 16 bits superiores de **rD** limpiando los 16 bits inferiores.

Codificación:



u:	0 - and IMM16 con bits 15-0 de S1
	1 - and IMM16 con bits 31-16 de S1
D, S1:	registro destino, origen 1
IMM16:	operando inmediato sin signo de 16 bits

Instrucción or

Operación:

$$rD \leftarrow rS1 \text{ or } rS2$$

Sintaxis:

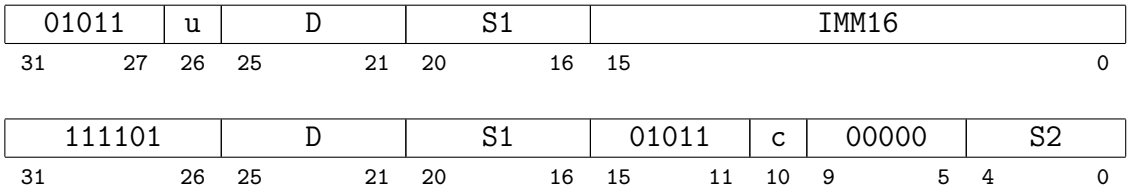
```
or    rD,rS1,rS2          or    rD,rS1,IMM16
or.c  rD,rS1,rS2          or.u  rD,rS1,IMM16
```

Descripción:

Para el formato de tres registros, la instrucción **or** hace la operación lógica or del registro **S1** con el registro **S2** y el resultado lo almacena en el registro **rD**. Si se especifica la opción **.c**, el registro **S2** se complementa a 1 antes de hacer la operación.

Si la instrucción lleva un dato inmediato de 16 bits (**IMM16**) se hace la operación or con este dato y los 16 bits menos significativos del registro **S1**. Los 16 bits superiores de **S1** se llevan a **rD** sin ningún cambio. Especificando la opción **.u** la operación se hace con los 16 bits superiores de **S1** y los menos significativos se llevan a **rD** sin cambios.

Codificación:



u:	0 - or IMM16 con bits 15-0 de S1
	1 - or IMM16 con bits 31-16 de S1
D, S1, S2:	registro destino, origen 1 y origen 2
IMM16:	operando inmediato sin signo de 16 bits
c:	0 - segundo registro no se complementa a 1
	1 - segundo registro se complementa a 1

2.2. Instrucciones aritméticas enteras

Instrucción add

Operación:

$$rD \leftarrow rS1 + rS2$$

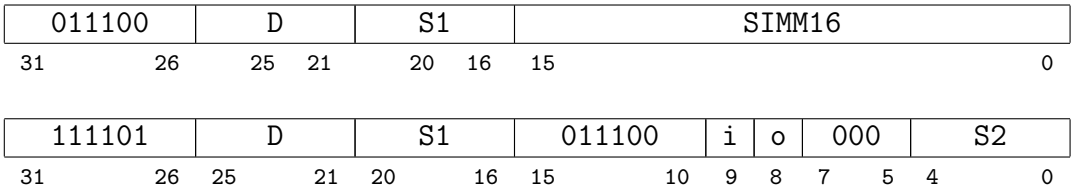
Sintaxis:

```
add      rD,rS1,rS2
add.ci   rD,rS1,rS2
add.co   rD,rS1,rS2
add.cio  rD,rS1,rS2
add      rD,rS1,SIMM16
```

Descripción:

La instrucción **add** suma el contenido del registro **rS1** con el contenido del registro **rS2** o el valor inmediato con signo **SIMM16** y almacena el resultado en el registro **rD**. Es siempre una suma con signo. La opción **.ci** hace que el bit de acarreo del registro de control se sume al resultado. La opción **.co** lleva el acarreo generado en la suma al registro de control. La opción **.cio** es la combinación de las dos opciones anteriores. Si el resultado de la suma no puede representarse como un número entero de 32 bits se produce una excepción de overflow y el registro destino de la instrucción (**rD**) no se modifica.

Codificación:



`D, S1, S2:`	registro destino, origen 1 y origen 2
`SIMM16:`	operando inmediato con signo de 16 bits
`i:`	0 - Desactiva acarreo entrante
	1 - Activa acarreo entrante
`o:`	0 - Desactiva acarreo de salida
	1 - Activa acarreo de salida

Instrucción addu

Operación:

$$rD \leftarrow rS1 + rS2$$

Sintaxis:

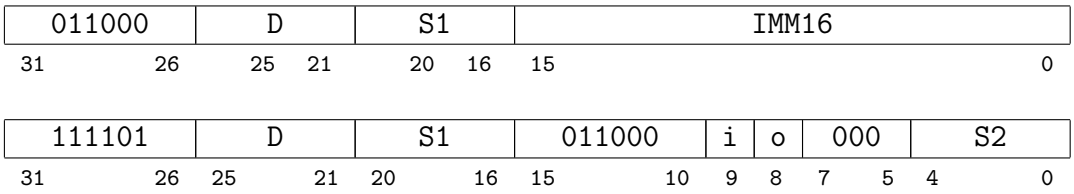
```
addu      rD,rS1,rS2
addu.ci   rD,rS1,rS2
addu.co   rD,rS1,rS2
addu.cio  rD,rS1,rS2
addu      rD,rS1,IMM16
```

Descripción:

La instrucción **addu** suma el contenido del registro **rS1** con el contenido del registro **rS2** o el valor inmediato sin signo **IMM16** y almacena el resultado en el registro **rD**. Es siempre una suma sin signo. La

opción **.ci** hace que el bit de acarreo del registro de control se sume al resultado. La opción **.co** lleva el acarreo generado en la suma al registro de control. La opción **.cio** es la combinación de las dos opciones anteriores. La instrucción **addu** no causa excepción si el resultado no puede representarse como un entero sin signo de 32 bits.

Codificación:



`D, S1, S2:`	registro destino, origen 1 y origen 2
`IMM16:`	operando inmediato sin signo de 16 bits
`i:`	0 - Desactiva acarreo entrante
	1 - Activa acarreo entrante
`o:`	0 - Desactiva acarreo de salida
	1 - Activa acarreo de salida

Instrucción sub

Operación:

$$rD \leftarrow rS1 - rS2$$

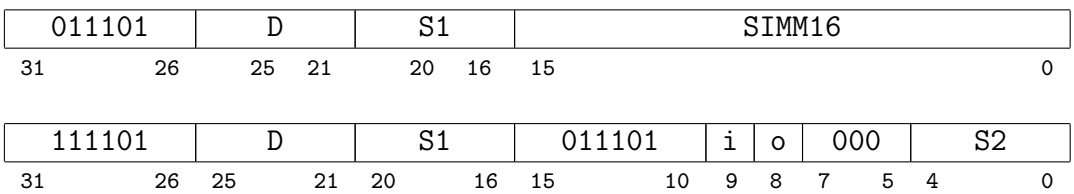
Sintaxis:

```
sub      rD,rS1,rS2
sub.ci   rD,rS1,rS2
sub.co   rD,rS1,rS2
sub.cio  rD,rS1,rS2
sub      rD,rS1,SIMM16
```

Descripción:

La instrucción **sub** resta el contenido del registro **rS1** con el contenido del registro **rS2** o el valor inmediato con signo **SIMM16** y almacena el resultado en el registro **rD**. Es siempre una resta con signo. La opción **.ci** hace que el bit de acarreo del registro de control se reste al resultado. La opción **.co** lleva el acarreo generado en la resta al registro de control. La opción **.cio** es la combinación de las dos opciones anteriores. Si el resultado de la resta no puede representarse como un número entero de 32 bits se produce una excepción de overflow y el registro destino de la instrucción (**rD**) no se modifica.

Codificación:



`D, S1, S2:`	registro destino, origen 1 y origen 2
`SIMM16:`	operando inmediato con signo de 16 bits
`i:`	0 - Desactiva acarreo entrante
	1 - Activa acarreo entrante
`o:`	0 - Desactiva acarreo de salida
	1 - Activa acarreo de salida

Instrucción subu

Operación:

rD ← rS1 - rS2

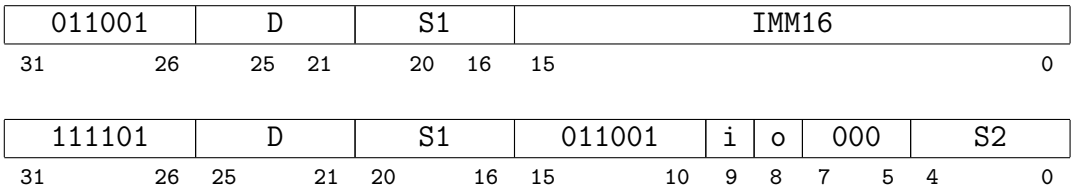
Sintaxis:

```
subu      rD,rS1,rS2
subu.ci   rD,rS1,rS2
subu.co   rD,rS1,rS2
subu.cio  rD,rS1,rS2
subu      rD,rS1,IMM16
```

Descripción:

La instrucción **subu** resta el contenido del registro **rS1** con el contenido del registro **rS2** o el valor inmediato sin signo **IMM16** y almacena el resultado en el registro **rD**. Es siempre una resta sin signo. La opción **.ci** hace que el bit de acarreo del registro de control se reste (resta con *borrow*) al resultado. La opción **.co** lleva el acarreo generado en la resta al registro de control. La opción **.cio** es la combinación de las dos opciones anteriores. La instrucción **subu** no causa excepción si el resultado no puede representarse como un entero sin signo de 32 bits.

Codificación:



`D, S1, S2:`	registro destino, origen 1 y origen 2
`IMM16:`	operando inmediato sin signo de 16 bits
`i:`	0 - Desactiva acarreo entrante
	1 - Activa acarreo entrante
`o:`	0 - Desactiva acarreo de salida
	1 - Activa acarreo de salida

Instrucción cmp

Operación:

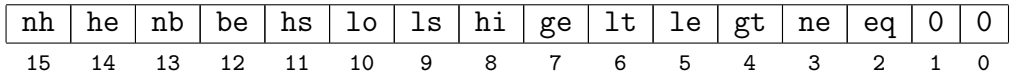
rD ← rS1 :: rS2

Sintaxis:

```
cmp   rD,rS1,rS2
cmp   rD,rS1,SIMM16
```

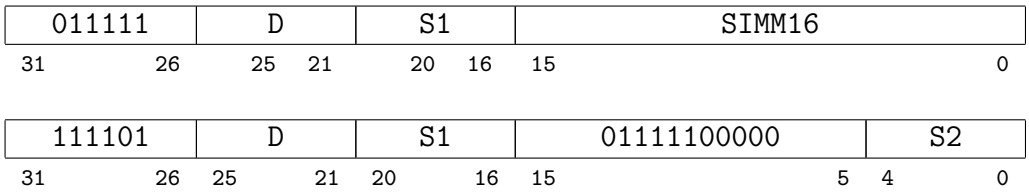
Descripción:

La instrucción **cmp** compara el contenido del registro **rS1** con el contenido del registro **rS2** o el valor inmediato con signo **SIMM16** y pone el resultado de la comparación como una cadena de bits en el registro destino **rD**. Los 16 bits más significativos de la cadena resultado (**rD**) se ponen a cero. Los 16 bits menos significativos representan 14 condiciones resultado de la comparación que se explican a continuación:



`eq`: 1 si y sólo si rS1 = rS2
`ne`: 1 si y sólo si rS1 ≠ rS2
`gt`: 1 si y sólo si rS1 > rS2 (con signo)
`le`: 1 si y sólo si rS1 ≤ rS2 (con signo)
`lt`: 1 si y sólo si rS1 < rS2 (con signo)
`ge`: 1 si y sólo si rS1 ≥ rS2 (con signo)
`hi`: 1 si y sólo si rS1 > rS2 (sin signo)
`ls`: 1 si y sólo si rS1 ≤ rS2 (sin signo)
`lo`: 1 si y sólo si rS1 < rS2 (sin signo)
`hs`: 1 si y sólo si rS1 ≥ rS2 (sin signo)
`be`: 1 si y sólo si rS1 y rS2 tienen algún byte igual
`nb`: 1 si y sólo si rS1 y rS2 no tienen ningún byte igual
`he`: 1 si y sólo si rS1 y rS2 tienen alguna media palabra igual
`nh`: 1 si y sólo si rS1 y rS2 no tienen ninguna media palabra igual

Codificación:



`D, S1, S2`: registro destino, origen 1 y origen 2
`SIMM16`: operando inmediato con signo de 16 bits

Instrucción muls

Operación:

rD ← rS1 * rS2

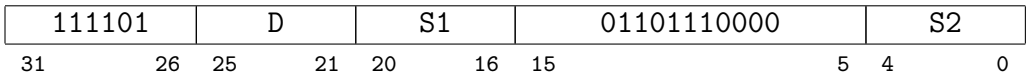
Sintaxis:

```
muls  rD,rS1,rS2
```

Descripción:

La instrucción **muls** multiplica el contenido del registro **rS1** con el contenido del registro **rS2** y pone el resultado en el registro **rD**. Es siempre una multiplicación con signo. Si el resultado de la multiplicación no puede representarse como un entero de 32 bits se produce una excepción de overflow.

Codificación:



`D, S1, S2:` registro destino, origen 1 y origen 2

Instrucción mulu

Operación:

$$rD \leftarrow rS1 * rS2$$

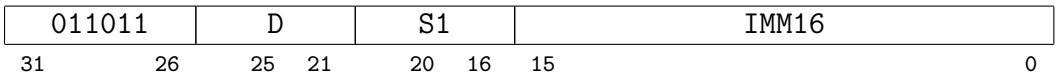
Sintaxis:

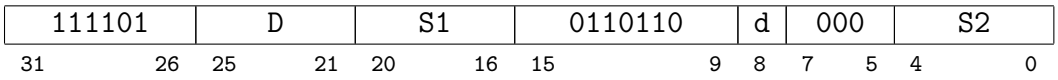
```
mulu    rD,rS1,rS2                    mulu   rD,rS1,IMM16
mulu.d  rD,rS1,rS2
```

Descripción:

Multiplicación entera sin signo. La instrucción **mulu** multiplica el contenido del registro **rS1** con el contenido del registro **rS2** o el valor inmediato sin signo **IMM16** y pone los 32 bits menos significativos del resultado en el registro **rD**. La opción **.d** especifica que los 64 bits del producto de ambos registros origen se almacenen en los registros **rD** y **rD+1**.

Codificación:





`D, S1, S2:` registro destino, origen 1 y origen 2
`IMM16:` operando inmediato sin signo de 16 bits
`d:` 0- destino en rD. Resultado en simple precisión
1- destino en rD y rD+1. Resultado en doble precisión

Instrucción divs

Operación:

$$rD \leftarrow rS1 \ / \ rS2$$

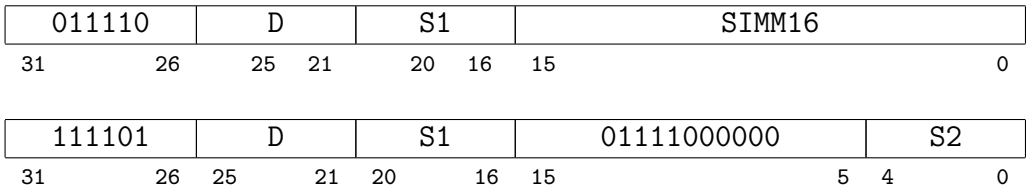
Sintaxis:

```
divs   rD,rS1,rS2
divs   rD,rS1,SIMM16
```

Descripción:

División entera con signo. La instrucción **divs** divide el contenido del registro **rS1** entre el contenido del registro **rS2** o el valor inmediato con signo **SIMM16** y pone el resultado de 32 bits en el registro **rD**. Si el contenido del registro **rS2** es cero provoca una excepción de división entera por cero o excepción de overflow y el registro destino de la instrucción (**rD**) no se modifica.

Codificación:



`D, S1, S2:` registro destino, origen 1 y origen 2
`SIMM16:` operando inmediato con signo de 16 bits

Instrucción divu

Operación:

$$rD \leftarrow rS1 \ / \ rS2$$

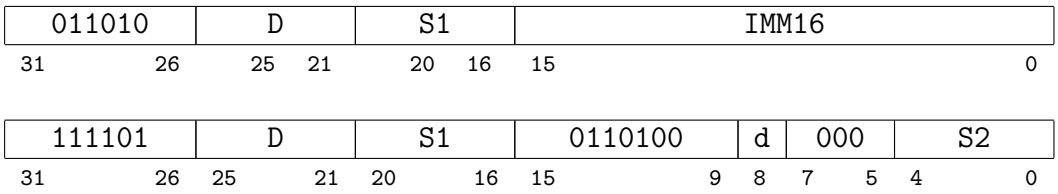
Sintaxis:

```
divu    rD,rS1,rS2                    divu  rD,rS1,IMM16
divu.d  rD,rS1,rS2
```

Descripción:

La instrucción **divu** divide el contenido del registro **rS1** con el contenido del registro **rS2** o el valor inmediato sin signo **IMM16** y pone el resultado de la división en el registro **rD**. Es una división sin signo. La opción **.d** especifica que los 64 bits del dividendo se toman de los registros **rS1** y **rS1+1** y el resultado de 64 bits se coloca en los registros **rD** y **rD+1**. Si el divisor **rS2** es cero se produce una excepción de división entera por cero y el registro destino de la instrucción (**rD**) no se modifica.

Codificación:



`D, S1, S2:` registro destino, origen 1 y origen 2
`IMM16:` operando inmediato sin signo de 16 bits
`d:` 0- destino en rD. Resultado en simple precisión.
1- destino en rD y rD+1. Resultado en doble precisión.

2.3. Instrucciones de campo de bit

Instrucción clr

Operación:

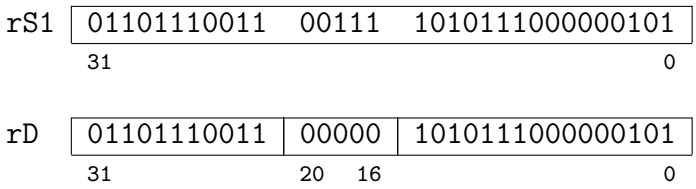
rD ← rS1 and (campo de bits de 0's)

Sintaxis:

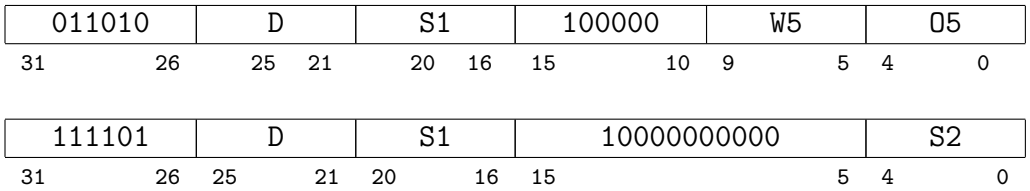
```
clr   rD,rS1,W5<O5>
clr   rD,rS1,rS2
```

Descripción:

La instrucción **clr** lee el contenido del registro **rS1** e inserta un campo de ceros en el resultado (**rD**). La longitud del campo de bit se especifica por el campo **W5** y el desplazamiento dentro de los 32 bits a partir del bit cero se indica por el campo **O5**. Para el formato de tres registros, los bits 9-5 y 4-0 del registro **rS2** se toman como los campos **W5** y **O5** respectivamente. Si el campo **W5** contiene el valor 0 se entenderá que la operación afecta a los 32 bits del operando. Por ejemplo si **W5** contiene 5 y **O5** contiene 16 el resultado será:



Codificación:



D, S1, S2:	registro destino, origen 1 y origen 2
W5:	5 bits que indican la longitud del campo de bit
O5:	5 bits que indican el desplazamiento

Instrucción set

Operación:

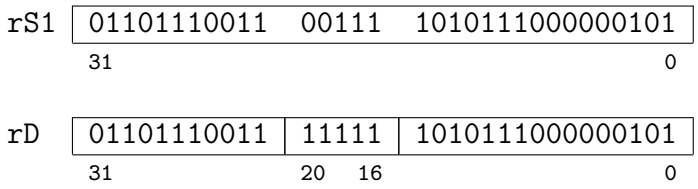
rD ← rS1 or (campo de bits de 1's)

Sintaxis:

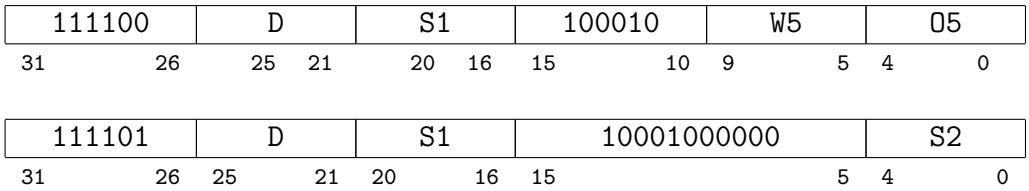
```
set   rD,rS1,W5<O5>
set   rD,rS1,rS2
```

Descripción:

La instrucción **set** lee el contenido del registro **rS1** e inserta un campo de unos en el resultado (**rD**). La longitud del campo de bit se especifica por el campo **W5** y el desplazamiento dentro de los 32 bits a partir del bit cero se indica por el campo **O5**. Para el formato de tres registros, los bits 9-5 y 4-0 del registro **rS2** se toman como los campos **W5** y **O5** respectivamente. Si el campo **W5** contiene el valor 0 se entenderá que la operación afecta a los 32 bits del operando. Por ejemplo si **W5** contiene 5 y **O5** contiene 16 el resultado será:



Codificación:



`D, S1, S2:`	registro destino, origen 1 y origen 2
`W5:`	5 bits que indican la longitud del campo de bit
`O5:`	5 bits que indican el desplazamiento

Instrucción ext

Operación:

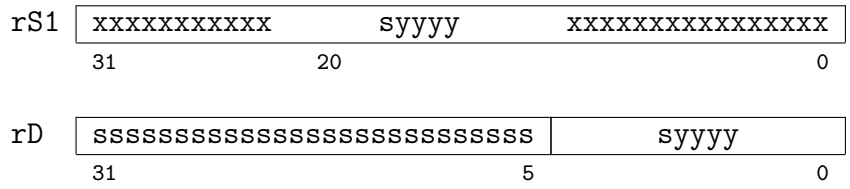
rD ← campo de bit con extensión de signo de S1

Sintaxis:

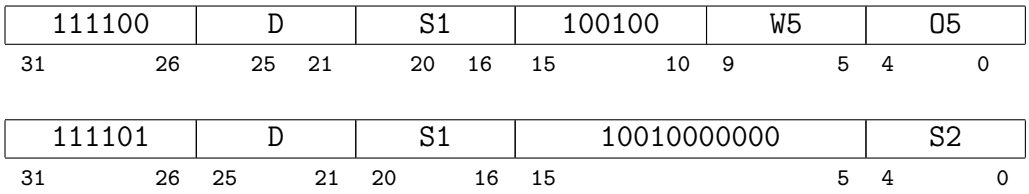
```
ext   rD,rS1,W5<O5>
ext   rD,rS1,rS2
```

Descripción:

La instrucción **ext** extrae un campo de bit del registro **rS1**. El campo de bit extraído es extendido con signo hasta completar los 32 bits que se introducen en el registro **rD**. La longitud del campo de bit se especifica por el campo **W5** y el desplazamiento dentro de los 32 bits a partir del bit cero se indica por el campo **O5**. Para el formato de tres registros, los bits 9-5 y 4-0 del registro **rS2** son tomados como los campos **W5** y **O5** respectivamente. Si el campo **W5** está a cero hay que realizar un desplazamiento aritmético múltiple a la derecha. El campo **O5** indicará el número de desplazamientos a realizar. Por ejemplo si **W5** contiene 5 y **O5** contiene 16 el resultado será:



Codificación:



`D, S1, S2:` registro destino, origen 1 y origen 2
`W5:` 5 bits indicando la longitud del campo de bit
`O5:` 5 bits indicando el desplazamiento

Instrucción extu

Operación:

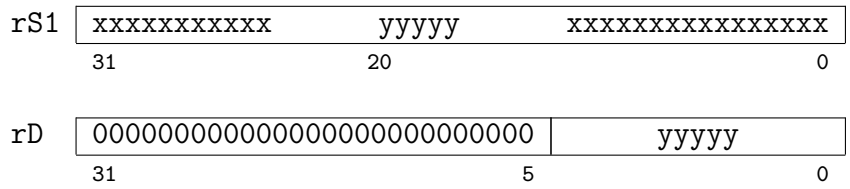
rD ← campo de bits sin extensión de signo de rS1

Sintaxis:

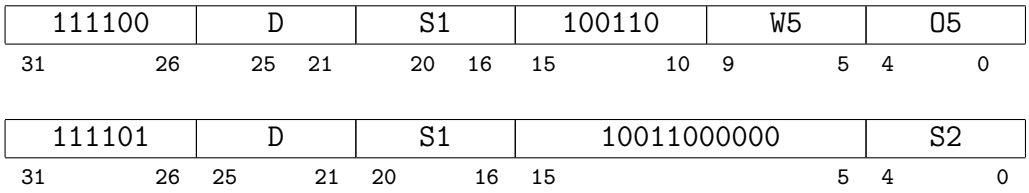
```
extu  rD,rS1,W5<O5>
extu  rD,rS1,rS2
```

Descripción:

La instrucción **extu** extrae un campo de bit del registro **rS1**. El campo de bit extraído es extendido con ceros hasta completar los 32 bits que se introducen en el registro **rD**. La longitud del campo de bit se especifica por el campo **W5** y el desplazamiento dentro de los 32 bits a partir del bit cero se indica por el campo **O5**. Para el formato de tres registros, los bits 9-5 y 4-0 del registro **rS2** se toman como los campos **W5** y **O5** respectivamente. Si el campo **W5** está a cero hay que realizar un desplazamiento lógico múltiple a la derecha. El campo **O5** indicará el número de desplazamientos a realizar. Por ejemplo si **W5** contiene 5 y **O5** contiene 16 el resultado será:



Codificación:



`D, S1, S2:` registro destino, origen 1 y origen 2
`W5:` 5 bits que indican la longitud del campo de bit
`O5:` 5 bits que indican el desplazamiento

Instrucción mak

Operación:

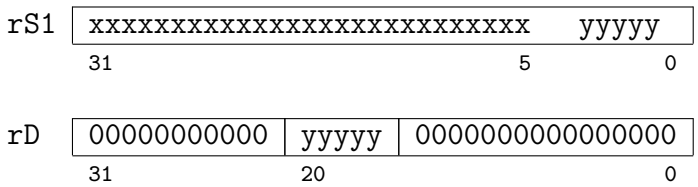
rD ← campo de bits de rS1

Sintaxis:

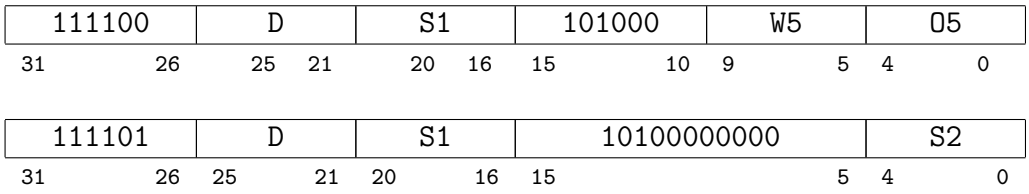
```
mak  rD,rS1,W5<O5>
mak  rD,rS1,rS2
```

Descripción:

La instrucción **mak** extrae un campo de bit del registro **rS1**. El campo de bit empieza en el bit menos significativo de **rS1** y se introduce en **rD** según un desplazamiento indicado. La longitud del campo de bit se especifica por el campo **W5** y el desplazamiento dentro de los 32 bits a partir del bit cero se indica por el campo **O5**. Para el formato de tres registros, los bits 9-5 y 4-0 del registro **rS2** se toman como los campos **W5** y **O5** respectivamente. Si el campo **W5** está a cero hay que realizar un desplazamiento múltiple a la izquierda y se introducen ceros por la derecha. El campo **O5** indicará el número de desplazamientos a realizar. Por ejemplo si **W5** contiene 5 y **O5** contiene 16 el resultado será:



Codificación:



`D, S1, S2:` registro destino, origen 1 y origen 2
`W5:` 5 bits indicando la longitud del campo de bit
`O5:` 5 bits indicando el desplazamiento

Instrucción rot

Operación:

rD ← campo de bits de rS1

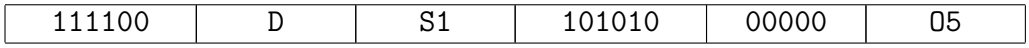
Sintaxis:

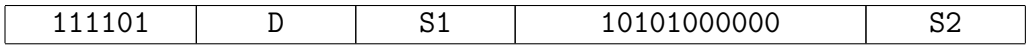
```
rot   rD,rS1,<O5>
rot   rD,rS1,rS2
```

Descripción:

La instrucción **rot** rota los bits del registro **rS1** hacia la derecha el número de veces indicado en **O5**. Para el formato de tres registros, los bits 4-0 del registro **rS2** se toman como el campo **O5**.

Codificación:





`D, S1, S2:` registro destino, origen 1 y origen 2
`O5:` 5 bits que indican el desplazamiento

2.4. Instrucciones de coma flotante

Instrucción fadd

Operación:

$$rD \leftarrow rS1 + rS2$$

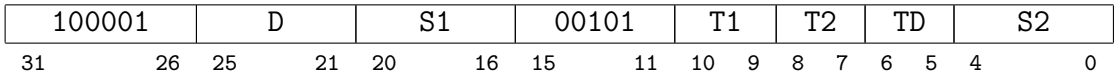
Sintaxis:

```
fadd.sss  rD,rS1,rS2        fadd.dss  rD,rS1,rS2
fadd.ssd  rD,rS1,rS2        fadd.dsd  rD,rS1,rS2
fadd.sds  rD,rS1,rS2        fadd.dds  rD,rS1,rS2
fadd.sdd  rD,rS1,rS2        fadd.ddd  rD,rS1,rS2
```

Descripción:

La instrucción **fadd** suma los contenidos de los registros **rS1** y **rS2** según el estándar IEEE 754 y almacena el resultado en el registro **rD**. Se puede indicar cualquier combinación de la precisión de los operandos (s-simple precisión, d-doble precisión) mediante los sufijos en ensamblador. Pueden darse excepciones de overflow, underflow y operando reservado.

Codificación:



`D, S1, S2:`	registro destino, origen 1 y origen 2
`T1:`	Precisión del operando S1
`T2:`	Precisión del operando S2
`TD:`	Precisión del operando destino
`Para T1, T2, TD:`	00 - simple precisión
	01 - doble precisión
	10 - no se usa
	11 - no se usa

Instrucción fcmp

Operación:

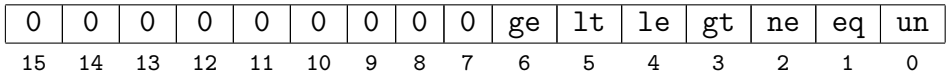
$$rD \leftarrow rS1 \text{ and } rS2$$

Sintaxis:

```
fcmp.sss  rD,rS1,rS2        fcmp.sds  rD,rS1,rS2
fcmp.ssd  rD,rS1,rS2        fcmp.sdd  rD,rS1,rS2
```

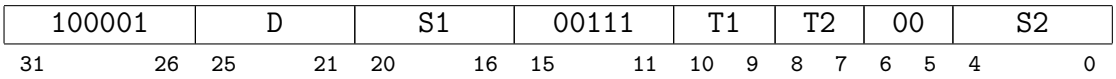
Descripción:

La instrucción **fcmp** compara los contenidos de los registros **rS1** y **rS2** según el estándar IEEE 754 y almacena el resultado de la comparación en el registro **rD** según la interpretación abajo indicada. Se puede indicar cualquier combinación de la precisión de los operandos (s-simple precisión, d-doble precisión) mediante los sufijos en ensamblador. Puede darse excepción de operando reservado. Los 25 bits más significativos de **rD** se ponen a cero. El resto de bits quedan como se indica en la figura:



`un:` 1 si un operando o los dos operandos son `NaN`
`eq:` 1 si y sólo si rS1 = rS2
`ne:` 1 si y sólo si rS1 $\neq$ rS2
`gt:` 1 si y sólo si rS1 $>$ rS2 (con signo)
`le:` 1 si y sólo si rS1 $\leq$ rS2 (con signo)
`lt:` 1 si y sólo si rS1 $<$ rS2 (con signo)
`ge:` 1 si y sólo si rS1 $\geq$ rS2 (con signo)

Codificación:



`D, S1, S2:` registro destino, origen 1 y origen 2
`T1:` Precisión del operando S1
`T2:` Precisión del operando S2
`Para T1, T2:` 00 - simple precisión
01 - doble precisión
10 - no se usa
11 - no se usa

Instrucción fcvt

Operación:

$$rD \leftarrow rS2$$

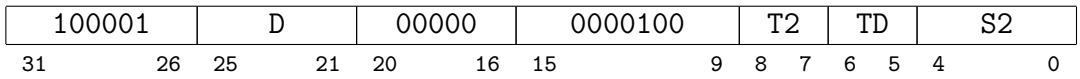
Sintaxis:

```
fcvt.sd  rD,rS2
fcvt.ds  rD,rS2
```

Descripción:

La instrucción **fcvt** cambia la precisión del número en coma flotante contenido en el registro **rS2** a la precisión indicada según el estándar IEEE 754 y almacena el resultado en el registro **rD**. Se puede indicar cualquier combinación de la precisión de los operandos (s-simple precisión, d-doble precisión) mediante los sufijos en ensamblador. Pueden darse excepciones de overflow, underflow y operando reservado.

Codificación:



`D, S1, S2:` registro destino, origen 1 y origen 2
`T2:` Precisión del operando S2
`TD:` Precisión del operando destino
`Para T2, TD:` 00 - simple precisión
01 - doble precisión
10 - no se usa
11 - no se usa

Instrucción fdiv

Operación:

$$rD \leftarrow rS1 \ / \ rS2$$

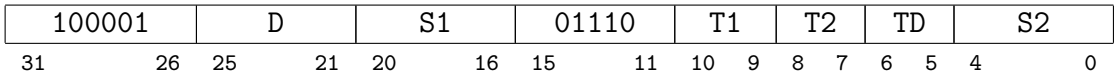
Sintaxis:

```
fdiv.sss  rD,rS1,rS2        fdiv.dss  rD,rS1,rS2
fdiv.ssd  rD,rS1,rS2        fdiv.dsd  rD,rS1,rS2
fdiv.sds  rD,rS1,rS2        fdiv.dds  rD,rS1,rS2
fdiv.sdd  rD,rS1,rS2        fdiv.ddd  rD,rS1,rS2
```

Descripción:

La instrucción **fdiv** divide el operando **rS1** entre el operando **rS2** de acuerdo al estándar IEEE 754 y almacena el resultado en el registro **rD**. Se puede indicar cualquier combinación de la precisión de los operandos (s-simple precisión, d-doble precisión) mediante los sufijos en ensamblador. Pueden darse excepciones de overflow, underflow, operando reservado y división por cero.

Codificación:



`D, S1, S2:`	registro destino, origen 1 y origen 2
`T1:`	Precisión del operando S1
`T2:`	Precisión del operando S2
`TD:`	Precisión del operando destino
`Para T1, T2, TD:`	00 - simple precisión
	01 - doble precisión
	10 - no se usa
	11 - no se usa

Instrucción flt

Operación:

$$rD \leftarrow Float\ (rS2)$$

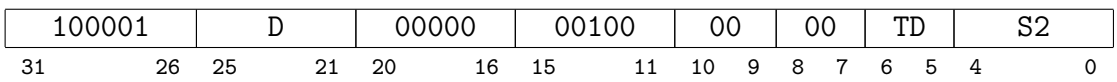
Sintaxis:

```
flt.ss  rD,rS2
flt.ds  rD,rS2
```

Descripción:

La instrucción **flt** convierte el entero con signo contenido en **rS2** a su representación en coma flotante y con la precisión indicada.

Codificación:



`D, rS2:`	registro destino y origen 2
`TD:`	Precisión del operando destino
`Para TD:`	00 - simple precisión
	01 - doble precisión
	10 - no se usa
	11 - no se usa

Instrucción fmul

Operación:

rD ← rS1 * rS2

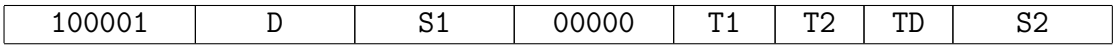
Sintaxis:

```
fmul.sss  rD,rS1,rS2        fmul.dss  rD,rS1,rS2
fmul.ssd  rD,rS1,rS2        fmul.dsd  rD,rS1,rS2
fmul.sds  rD,rS1,rS2        fmul.dds  rD,rS1,rS2
fmul.sdd  rD,rS1,rS2        fmul.ddd  rD,rS1,rS2
```

Descripción:

La instrucción **fmul** multiplica el operando **rS1** por el operando **rS2** de acuerdo al estándar IEEE 754 y almacena el resultado en el registro **rD**. Se puede indicar cualquier combinación de la precisión de los operandos (s-simple precisión, d-doble precisión) mediante sufijos en ensamblador. Pueden darse excepciones de overflow, underflow y operando reservado.

Codificación:



`D, S1, S2:`	registro destino, origen 1 y origen 2
`T1:`	Precisión del operando S1
`T2:`	Precisión del operando S2
`TD:`	Precisión del operando destino
`Para T1, T2, TD:`	00 - simple precisión
	01 - doble precisión
	10 - no se usa
	11 - no se usa

Instrucción fsub

Operación:

rD ← rS1 - rS2

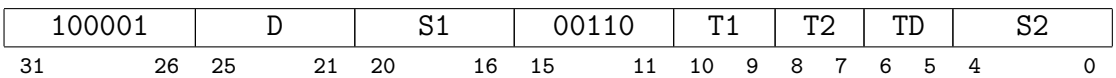
Sintaxis:

```
fsub.sss  rD,rS1,rS2        fsub.dss  rD,rS1,rS2
fsub.ssd  rD,rS1,rS2        fsub.dsd  rD,rS1,rS2
fsub.sds  rD,rS1,rS2        fsub.dds  rD,rS1,rS2
fsub.sdd  rD,rS1,rS2        fsub.ddd  rD,rS1,rS2
```

Descripción:

La instrucción **fsub** resta los contenidos de los registros **rS1** y **rS2** según el estándar IEEE 754 y coloca el resultado en el registro **rD**. Se puede indicar cualquier combinación de la precisión de los operandos (s-simple precisión, d-doble precisión) mediante los sufijos en ensamblador. Pueden darse excepciones de overflow, underflow y operando reservado.

Codificación:



`D, S1, S2:`	registro destino, origen 1 y origen 2
`T1:`	Precisión del operando S1
`T2:`	Precisión del operando S2
`TD:`	Precisión del operando destino
`Para T1, T2, TD:`	00 - simple precisión
	01 - doble precisión
	10 - no se usa
	11 - no se usa

Instrucción int

Operación:

$$rD \leftarrow \text{Round } (rS2)$$

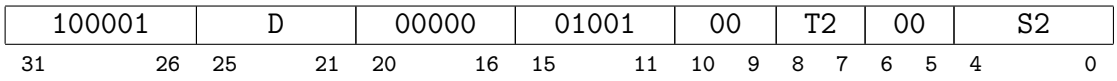
Sintaxis:

```
int.ss  rD,rS2
int.sd  rD,rS2
```

Descripción:

La instrucción **int** convierte el número en coma flotante contenido en **rS2** a un entero con signo de 32 bits usando el modo de redondeo indicado en el registro de control de la máquina. Puede generar excepciones de overflow y operando reservado.

Codificación:



`D, S2:`	registro destino y origen 2
`T2:`	Precisión del operando S2
`Para T2:`	00 - simple precisión
	01 - doble precisión
	10 - no se usa
	11 - no se usa

2.5. Instrucciones de control de flujo

Instrucción bb0

Operación:

$$\text{Si } rS1_{(B5)} = 0,\ \text{PC} \leftarrow \text{PC} + \text{D16} * 4$$

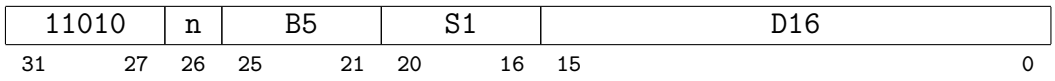
Sintaxis:

```
bb0     B5,rS1,D16          bb0     cnd,rS1,D16
bb0.n   B5,rS1,D16          bb0.n   cnd,rS1,D16
```

Descripción:

La instrucción **bb0** examina el bit del registro **rS1** especificado por el campo **B5**. Si el bit es cero se efectúa el salto. Para calcular la dirección de salto el desplazamiento indicado en **D16** es extendido con signo, desplazado dos bits a la izquierda y sumado a la dirección de la instrucción **bb0**. La opción **.n** hace que la instrucción siguiente sea ejecutada antes de efectuar el salto. El campo **cnd** es un mnemónico que indica una de las condiciones resultantes de la instrucción **cmp** (véase la página 29). La instrucción **bb0** indica la predicción estática de salto como un salto que probablemente no se tome.

Codificación:



`n:` 0 - siguiente instrucción no se ejecuta antes del salto
1 - siguiente instrucción se ejecuta antes del salto
`S1:` registro origen 1
`B5:` entero sin signo de cinco bits que indica un número de bit en el registro
`D16:` Desplazamiento de 16 bits con signo.

Instrucción bb1

Operación:

$$\text{Si } rS1_{(B5)} = 1,\ \text{PC} \leftarrow \text{PC} + \text{D16} * 4$$

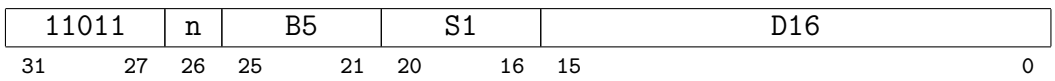
Sintaxis:

```
bb1     B5,rS1,D16          bb1     cnd,rS1,D16
bb1.n   B5,rS1,D16          bb1.n   cnd,rS1,D16
```

Descripción:

La instrucción **bb1** examina el bit del registro **rS1** especificado por el campo **B5**. Si el bit es uno se efectúa el salto. Para calcular la dirección de salto el desplazamiento indicado en **D16** es extendido con signo y desplazado dos bits a la izquierda y sumado a la dirección de la instrucción **bb1**. La opción **.n** hace que la instrucción siguiente sea ejecutada antes de efectuar el salto. El campo **cnd** es un mnemónico que indica una de las condiciones resultantes de la instrucción **cmp** (véase la página 29). La instrucción **bb1** indica la predicción estática de salto como un salto que probablemente se tome.

Codificación:



`n:` 0 - siguiente instrucción no se ejecuta antes del salto
1 - siguiente instrucción se ejecuta antes del salto
`S1:` registro origen 1
`B5:` entero sin signo de cinco bits que indica un número de bit en el registro
`D16:` Desplazamiento de 16 bits con signo.

Instrucción br

Operación:

$$PC \leftarrow PC + D26 * 4$$

Sintaxis:

```
br     D26
br.n   D26
```

Descripción:

La instrucción **br** provoca un salto incondicional. La dirección de salto se calcula desplazando 2 bits a la izquierda el valor D26 y se suma a la dirección ocupada por la instrucción **br**. La opción **.n** hace que la instrucción siguiente sea ejecutada antes de efectuar el salto.

Codificación:

`n:` 0 - siguiente instrucción no se ejecuta antes del salto
1 - siguiente instrucción se ejecuta antes del salto
`D26:` Desplazamiento de 26 bits con signo.

Instrucción bsr

Operación:

$$r1 \leftarrow PC + 4 \; ; \; PC \leftarrow PC + D26 * 4 \text{ si es } \textbf{bsr}$$

Operación:

$$r1 \leftarrow PC + 8 \; ; \; PC \leftarrow PC + D26 * 4 \text{ si es } \textbf{bsr.n}$$

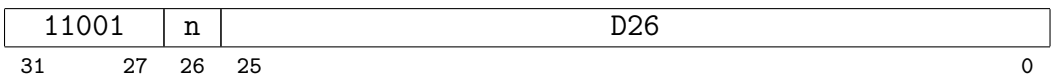
Sintaxis:

```
bsr     D26
bsr.n   D26
```

Descripción:

La instrucción **bsr** provoca un salto incondicional a subrutina. La dirección de salto se calcula desplazando 2 bits a la izquierda el valor D26 y se suma a la dirección ocupada por la instrucción **bsr**. La dirección de retorno se almacena en el registro **r1**. La opción **.n** hace que la siguiente instrucción sea ejecutada antes de efectuar el salto.

Codificación:



`n:` 0 - siguiente instrucción no se ejecuta antes del salto
1 - siguiente instrucción se ejecuta antes del salto
`D26:` Desplazamiento de 16 bits con signo.

Instrucción jmp

Operación:

PC ← rS2

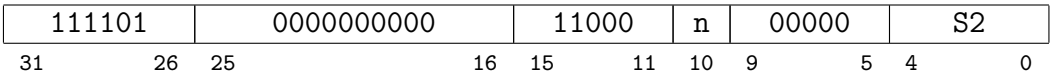
Sintaxis:

```
jmp    (rS2)
jmp.n  (rS2)
```

Descripción:

La instrucción **jmp** provoca un salto incondicional a la dirección contenida en **rS2**. Los dos bits menos significativos de esa dirección son puestos a cero para obtener una dirección alineada a palabra. Con la opción **.n** se ejecuta la instrucción siguiente a **jmp** antes de efectuar el salto.

Codificación:



`n:` 0 - siguiente instrucción no se ejecuta antes del salto
1 - siguiente instrucción se ejecuta antes del salto
`S2:` registro origen 2.

Instrucción jsr

Operación:

r1 ← PC + 4 ; PC ← rS2 si es **jsr**

Operación:

r1 ← PC + 8 ; PC ← rS2 si es **jsr.n**

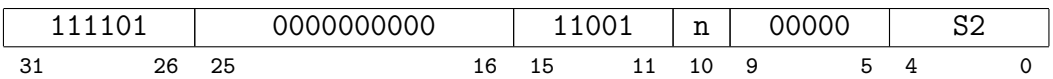
Sintaxis:

```
jsr    (rS2)
jsr.n  (rS2)
```

Descripción:

La instrucción **jsr** provoca un salto incondicional a la dirección contenida en **rS2** y guarda la dirección de retorno en el registro **r1**. Con la opción **.n** se ejecuta la instrucción siguiente a **jmp** antes de efectuar el salto. Los dos bits menos significativos de esa dirección son puestos a cero para obtener una dirección alineada a palabra.

Codificación:



`n:` 0 - siguiente instrucción no se ejecuta antes del salto
1 - siguiente instrucción se ejecuta antes del salto
`S2:` registro origen 2.

2.6. Instrucciones de carga/almacenamiento

Instrucción ld

Operación:

$$rD \leftarrow M(rS1 + SI16) \text{ ó } rD \leftarrow M(rS1 + rS2)$$

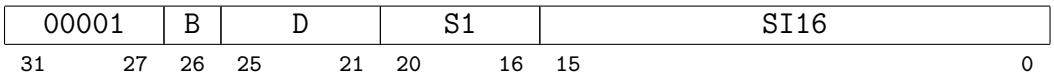
Sintaxis:

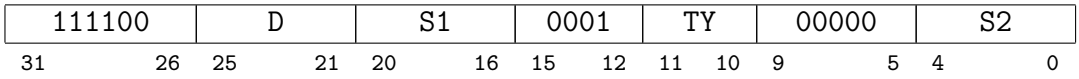
```
ld.b   rD,rS1,SI16          ld.b   rD,rS1,rS2
ld.bu  rD,rS1,SI16          ld.bu  rD,rS1,rS2
ld.h   rD,rS1,SI16          ld.h   rD,rS1,rS2
ld.hu  rD,rS1,SI16          ld.hu  rD,rS1,rS2
ld     rD,rS1,SI16          ld     rD,rS1,rS2
ld.d   rD,rS1,SI16          ld.d   rD,rS1,rS2
```

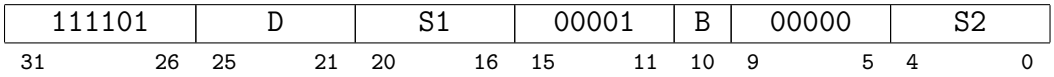
Descripción:

La instrucción **ld** lee datos del sistema de memoria y los carga en el registro destino. La dirección se obtiene sumando a la dirección base del registro **rS1** el entero con signo **SI16** o el contenido del registro **rS2**. La opción **.b** especifica que el objeto a transferir es un byte con signo, **.bu** un byte sin signo, **.h** media palabra (16 bits) con signo, **.hu** media palabra sin signo, **.d** es doble palabra (64 bits). Sin opción se entiende que el objeto a transferir es una palabra (32 bits).

Codificación:







`D, S1, S2:`	registro destino, origen 1 y origen 2
`SI16:`	Entero inmediato de 16 bits con signo.
`B:`	0 - Media palabra
	1 - Byte
`TY:`	00 - Doble palabra
	01 - Palabra
	10 - Media Palabra
	11 - Byte

Instrucción st

Operación:

$$M(rS1 + SI16) \leftarrow rD \text{ ó } M(rS1 + rS2) \leftarrow rD$$

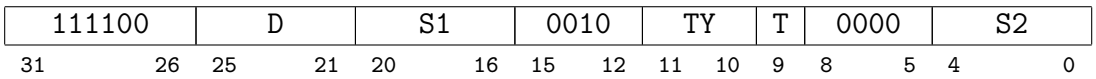
Sintaxis:

```
st.b   rD,rS1,SI16          st.b     rD,rS1,rS2
st.h   rD,rS1,SI16          st.h     rD,rS1,rS2
st     rD,rS1,SI16          st       rD,rS1,rS2
st.d   rD,rS1,SI16          st.d     rD,rS1,rS2
                            st.b.wt  rD,rS1,rS2
                            st.h.wt  rD,rS1,rS2
                            st.wt    rD,rS1,rS2
                            st.d.wt  rD,rS1,rS2
```

Descripción:

La instrucción **st** escribe el contenido del registro especificado en **rD** en el sistema de memoria. La dirección se obtiene sumando a la dirección base del registro **rS1** o bien el entero con signo **SI16** o el contenido del registro **rS2**. La opción **.b** indica que el objeto a transferir es un byte, **.h** media palabra (16 bits), **.d** es doble palabra (64 bits). Sin opción se entenderá que el objeto que se transferirá es una palabra (32 bits). La opción **.wt** indica que la escritura actualice incondicionalmente la memoria principal (*write-through*).

Codificación:



`D, S1, S2:`	registro destino, origen 1 y origen 2
`SI16:`	Entero inmediato de 16 bits con signo.
`T:`	0 - Almacenamiento normal
	1 - Write-through
`TY:`	00 - Doble palabra
	01 - Palabra
	10 - Media Palabra
	11 - Byte

Instrucción ldcr

Operación:

$$rD \leftarrow \text{registro de control}$$

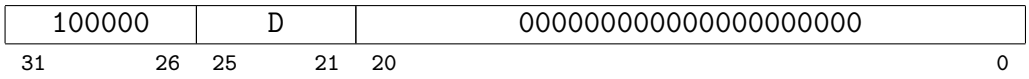
Sintaxis:

```
ldcr   rD
```

Descripción:

La instrucción **ldcr** mueve el contenido del registro de control del procesador al registro destino **rD**.

Codificación:



`D:` Registro destino

Instrucción stcr

Operación:

registro de control ← rS1

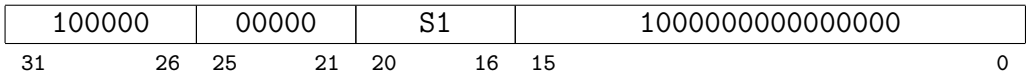
Sintaxis:

```
stcr  rS1
```

Descripción:

La instrucción **stcr** mueve el contenido del registro rS1 al registro de control del procesador.

Codificación:



`S1:` Registro origen

Instrucción xmem

Operación:

rD ↔ M(rS1 + rS2)

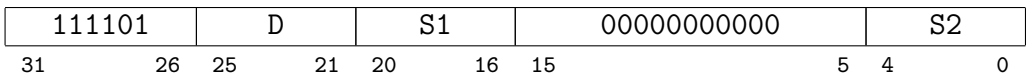
Sintaxis:

```
xmem  rD,rS1,rS2
```

Descripción:

La instrucción **xmem** intercambia el contenido del registro **rD** con la posición de memoria obtenida al sumar a la dirección base contenida en **rS1** la palabra con signo contenida en el registro **rS2**. Esta instrucción bloquea el bus del sistema durante toda la ejecución de la instrucción, es decir, el doble intercambio de información se realiza de forma atómica.

Codificación:



`D, r1, r2:` Registro destino, origen 1 y origen 2

2.7. Instrucciones especiales

Instrucción stop

Operación:

Finaliza la ejecución del programa de instrucciones y termina la simulación

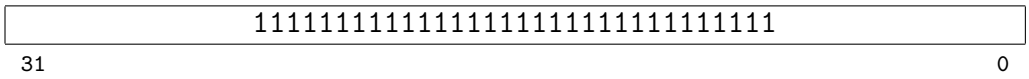
Sintaxis:

```
stop
```

Descripción:

La instrucción **stop** causa la terminación de la simulación y la ejecución del programa de instrucciones. El estado de la máquina se recupera para que coincida con el que tenía el procesador al terminar la ejecución de la instrucción anterior a la instrucción **stop**.

Codificación:



NEMÓNICO	OPERANDOS	DESCRIPCIÓN	31 26	25 21	20 16	15 10	9 5	4 0	COMENTARIO
and	rD, rS1, rS2	rD ← rS1 *op* rS2	111101	D	S1	010000	00000	S2	**.c:** Se complementa
and.c	rD, rS1, rS2					1			a 1 rS2
and	rD, rS1, IMM16	rD ← rS1 *op* IMM16 (los otros	010000			IMM16			**.u:** La operación se realiza con
and.u	rD, rS1, IMM16	16 bits sin cambiar)	1						los 16 bits superiores de rS1
xor	rD, rS1, rS2					01010			op es and, or ó xor
xor	rD, rS1, IMM16		01010						dependiendo de la instrucción
or	rD, rS1, rS2					01011			
or	rD, rS1, IMM16		01011						
mask	rD, rS1, IMM16	rD ← rS1 and IMM16 (los otros 16 a 0)	01001						
add	rD, rS1, rS2	rD ← rS1 ± rS2	111101	D	S1	011100	00 000	S2	**.ci:** Operación a través de
add.ci	rD, rS1, rS2						10		acarreo (del reg. de control
add.co	rD, rS1, rS2						01		**.co:** Lleva el acarreo al
add.cio	rD, rS1, rS2						11		bit del reg. de control
addu	rD, rS1, rS2					011000			**.cio:** .ci +.co
sub	rD, rS1, rS2					011101			Las operaciones add y sub
subu	rD, rS1, rS2					011001			causan excepción de OVF
add	rD, rS1, SIMM16	rD ← rS1 ± IMM16	011100			IMM16 ó SIMM16			
addu	rD, rS1, IMM16	u indica sin signo	011000						
sub	rD, rS1, SIMM16		011101						
subu	rD, rS1, IMM16		011001						
muls	rD, rS1, rS2	rD ← rS1 * rS2	111101	D	S1	011011	10000	S2	**.d:** Doble precisión (en los
divs	rD, rS1, rS2	rD ← rS1 / rS2				011110	00000		registros rD y rD+1)
divs	rD, rS1, SIMM16		011110			SIMM16			Las operaciones muls y divs
mulu	rD, rS1, rS2	u indica sin signo	111101			011011	00000		causan excepción de OVF
mulu.d	rD, rS1, rS2						1		Las operaciones divs y
mulu	rD, rS1, IMM16		011011			IMM16			divu causan excepción de
divu	rD, rS1, rS2		111101			011010	00000		dividir por cero
divu.d	rD, rS1, rS2						1		
divu	rD, rS1, IMM16		011010			IMM16			
clr	rD, rS1, W5<O5>	rD ← rS1 con bits O5 a O5 + W5 –1 a 0	111100	D	S1	100000	W5	O5	**W5:** Longitud del campo de bit
clr	rD, rS1, rS2		111101			100000	00000	S2	**O5:** Origen del campo de bit
set	Se opera	rD ← rS1 con bits O5 a O5 + W5 – 1 a 1				100010			Para las instrucciones de 3
ext	con los dos	rD ← los bits O5 a				100100			reg., W5 y O5 se toman
	tipos de	O5 + W5 – 1 de rS1 (con ext. signo)							los bits 9-5 y 4-0 de rS2
extu	formatos de	instr. ext sin extensión de signo				100110			respectivamente
mak	instrucción	bits O5 a O5 + W5 – 1 de rD ← bits				101000			
	utilizados	0 a W5 de rS1. resto a 0							
rot	en clr	rD ← O5 rotaciones dcha. de rS1				101010	00000		

NEMÓNICO	OPERANDOS	DESCRIPCIÓN	31 26	25 21	20 16	15 11	10 5	4 0	COMENTARIO
fadd.xxx	rD, rS1, rS2	rD ← rS1 *op* rS2	100001	D	S1	00101	T1T2TD	S2	**.x=.s:** Simple precisión
fsub.xxx	rD, rS1, rS2	T1, T2, TD son:				00110			**.x=.d:** Doble precisión
fmul.xxx	rD, rS1, rS2	00: Simple precisión				00000			.$\overline{x}$**=.d:** si x=s y .$\overline{x}$**=.s:** si x=d
fdiv.xxx	rD, rS1, rS2	01: Doble precisión				01110			fadd, fsub, fmul y fdiv causan
fcvt.x$\overline{x}$	rD, rS2	Cambio de precisión			00000	00001	00T2TD		excepciones OVF, UDF, de op.
flt.xs	rD, rS2	Coma flotante ← entero con signo				00100	0000TD		reservado y div. por 0. fcvt, int, sólo
int.sx	rD, rS2	entero con signo ← Coma flotante (s ó d)				01001	00T200		de OVF, UDF y de op. reservado
bb0	B5, rS1, D16	PC ← PC + 4*D16 si $RS1_{B5} = 0$	110100	B5	S1	D16			**.n:** Se ejecuta la instrucción
bb0.n	B5, rS1, D16		110101						siguiente antes de efectuar el salto.
bb1	B5, rS1, D16	PC ← PC + 4*D16 si $RS1_{B5} = 1$	11011						bb1, bsr y jsr admiten el sufijo
br	D26	PC ← PC + 4*D26 y	110000	D26					.n al igual que las instrucciones
br.n	D26	r1 ← PC + 4 si es bsr	1						que las preceden
bsr	D26		11001						bb0 predice que no salta
jmp	(rS2)	PC ← rS2 con $rS2_0 = rS2_1 = 0$ y	111101	00000	00000	11000	000000	S2	bb1 predice que salta
jmp.n	(rS2)	r1 ← PC + 4 si es jsr					1		
jsr	(rS2)					11001			
ld.b	rD, rS1, SI16	rD ← (rS1 + SI16)	000111	D	S1	SI16			**.b:** byte con signo
ld.h	rD, rS1, SI16		10						**.bu:** byte sin signo
ld	rD, rS1, SI16		01						**.h:** media palabra con signo
ld.d	rD, rS1, SI16		00						**.hu:** media palabra sin signo
st.y	rD, rS1, SI16	(rS1 + SI16) ← rD	0010						**.d:** doble palabra
ld.b	rD, rS1, rS2	rD ← (rS1 + rS2)	111100			00011	100000	S2	**.y** es .b, .h .d o sin extensión
ld.h	rD, rS1, rS2					1	0		**.wt** indica que la escritura
ld	rD, rS1, rS2					0	1		actualice incondicionalmente
ld.d	rD, rS1, rS2					0	0		la memoria principal
st.y	rD, rS1, rS2	(rS1 + rS2) ← rD				0010	00000		
st.y.wt	rD, rS1, rS2	(rS1 + rS2) ← rD					10000		
ld.bu	rD, rS1, SI16		000011			SI16			
ld.hu	rD, rS1, SI16		0						
ld.bu	rD, rS1, rS2		111101			00001	100000	S2	
ld.hu	rD, rS1, rS2						0		
ldcr	rD	rD ← reg. control	100000		00000	00000	000000	00000	
stcr	rS1	rS1 → reg. control	100000	00000	S1	10000	000000	00000	
xmem	rD, rS1, rS2	rD ↔ (rS1 + rS2)	111101		S1	00000	000000	S2	
cmp		VÉASE EL MANUAL							
fcmp		VÉASE EL MANUAL							