1 SISA-F (Simple Instruction Set Architecture-Float)

Juan J. Navarro, Toni Juan, Roger Espasa, Jordi Tubella, Joan Manel Parcerisa, Carlols Álvarez, Pau Cabre, Christian Perez

1-Septiembre-2007

1.1 Resumen

En este documento se define el lenguaje máquina SISA-F. Esta basado en el lenguaje SISA-I y ambos han sido creados para la enseñanza y el aprendizaje del área de arquitectura de computadores. Los aspectos del SISA-F que hacen que no sea un super conjunto del SISA-I (cambios sobre SISA-I) y las extensiones o añadidos sobre el SISA-I son las siguientes.

1.1.1 Cambios sobre SISA-I

Los siguientes aspectos hacen que SISA-F no sea un super conjunto del SISA-I. Esto es, SISA-F no es compatible con SISA-I:

- La memoria separada de instrucciones y datos del SISA-I se unifica en una sola en el SISA-F.
- El direccionamiento de la memoria es a nivel de byte, la memoria tiene 2^16 bytes.
- Se modifica la semantica de las instrucciones `BZ` y `BNZ`. En SISA-F el campo de la instrucción de 8 bits, `N`, que codifica un desplazamiento en complemento a dos, se multiplica por 2 antes de sumarse al `PC` actualizado (`PC`+2).
- Se modifica la semántica de las instrucciones `LD` y `ST`. En SISA-F el campo de la instrucción de 6 bits, `N`, que codifica un desplazamiento en complemento a dos, se multiplica por 2 antes de sumarse a `Ra` para formar la dirección efectiva de memoria.
- La semántica y formato del resto de instrucciones de SISA-I se mantiene en SISA-F.

1.1.2 Extensiones respecto SISA-I

- Nueva instrucción para obtener los 16 bits de menor peso de la multiplicación de enteros y naturales (`MUL`). Nuevas instrucciones para obtener los 16 bits de mayor peso de la multiplicación para enteros (`MULH`) y para naturales (`MULHU`).
- Nuevas instrucciones de salto a través de registro: condicionales, `JZ` y `JNZ`, Incondicional, `JMP`, además de `JAL` para llamadas a subrutinas.
- Nuevas instrucciones load (`LDB`) y store (`STB`) de byte.
- 8 nuevos registros (`F0`,..., `F7`) de coma flotante de 16 bits (float[1]).
- Nuevas instrucciones de coma flotante para suma (`ADDF`), resta (`SUBF`), multiplicación (`MULF`) y división (`DIVF`).
- Nuevas instrucciones de comparación en coma flotante (`CMPLTF`, `CMPLEF`, `CMPEQF`).
- Nuevas instrucciones de load (`LDF`) y store (`STF`) a/de los registros de coma flotante (de 16 bits). En estas instrucciones, para formar la dirección efectiva de memoria, el offset de 6 bits se multiplica por 2 antes de sumarse al registro base.
- 8 nuevos registros con funcionalidades especiales (`S0`,..., `S7`)

  Nuevas instrucciones para acceder a los registros especiales S (`RDS`, `WRS`),
- Se pueden generar las siguientes excepciones:

  Instrucción ilegal.

  Dirección de memoria mal alineada.

  Overflow en operación de coma flotante.

  División por cero en operación de coma flotante.

  División por cero en operación de enteros, con o sin signo
- Nuevas instrucciones para la gestión de entrada/salida:

  Para gestión de interrupciones (`EI`, `DI`, `GETIID`, `RETI`),
- Nueva instrucción para parar el computador, (`HALT`).

1.2 Visión general de la arquitectura

1.2.1 Arquitectura RISC.

SISA-F es una arquitectura RISC de 16 bits, que puede operar con números en coma flotante pequeños (small-floats) de 16 bits. Destacamos los siguientes aspectos:

1. El conjunto de instrucciones es reducido. Solamente hay 49 instrucciones diferentes.
2. Todas las instrucciones de lenguaje máquina son de un tamaño fijo de 16 bits (2 bytes, 1 palabra).

1. Seria más preciso denominar small-float al tipo de datos de coma flotante de 16 bits que soporta SISA-F, para diferenciarlo de las tipos usuales en otras arquitecturas: float, de 32 bits y double-float de 64 bits. Pero como SISA-F sólo soporta floats de 16 bits, no usaremos el nombre small-float, si no simplemente float.

3. Todos los operandos fuente y destino de las instrucciones que efectúan cálculos aritméticos, lógicos, comparaciones, etc. tienen los operandos en registros del procesador y dejan el resultado en uno de esos registros. Estas instrucciones tienen dos registros fuente y uno destino que pueden ser distintos. Para mover datos entre memoria y registros existen instrucciones de tipo *load* (LDB, LD y LDF) y *store* (STB, ST, STF).

1.2.2 Estado del computador

El estado del computador lo forman la información contenida en todos los elementos de memorización del computador (registros, posiciones de memoria, registros de entrada/salida,...) que son visibles desde el nivel de lenguaje máquina (que pueden ser leidos y escritos mediante la ejecución de instrucciones de lenguaje máquina). En el computador posiblemente hay otros elementos de memorización (registros, biestables,...) que se usan para almacenar resultados intermedios durante las distintas fases de ejecución de una instrucción y que no forman parte del estado del computador. Podemos decir que la ejecución de una instrucción de lenguaje máquina provoca una modificación del estado del computador en función del estado del computador antes de su ejecución. Otra forma de diferenciar los elementos de memorización del computador que contienen el estado del computador de los que no lo contienen es la siguiente. Imaginemos una implementación secuencial del lenguaje máquina en la que no comienza la ejecución de una instrucción antes de que termine totalmente la ejecución de la instrucción anterior, como es el caso de las implementaciones SISP-I-1, SISP-I-2 y SISP-I-3. Si después de ejecutar una instrucción de lenguaje máquina y antes de que empiece a ejecutarse la siguiente instrucción, pudiéramos modificar de manera aleatoria el valor contenido en los elementos de memorización que no forman parte del estado del computador y no modificáramos el contenido de ningún elemento que forma parte del estado del computador, la ejecución del programa continuaría produciendo los mismos resultados que si no hubiéramos modificado nada.

El estado del computador de la arquitectura SISA-F lo forman el contenido de los siguientes elementos de memorización:

- Memoria: 2^16 bytes
- PC, registro contador de programa, de 16 bits.
- Registros generales: 8 registros de 16 bits cada uno.
- Registros de coma flotante: 8 registros de 16 bits cada uno.
- Registros especiales: 8 registros de 16 bits cada uno (uno de ellos contiene la palabra de estado del computador).
- Registros de entrada/salida: 2^8 registros de entrada y 2^8 registros de salida de 16 bits cada uno.

A continuación definimos con más profundidad cada uno de estos elementos de memorización.

Memoria

Esta arquitectura tiene un único espacio de direccionamiento de memoria, que se usa tanto para código como para datos. Una dirección de memoria consta de 16 bits y la unidad de direccionamiento es el byte (8 bits): se pueden direccionar hasta 2^16 bytes de memoria.

El acceso a datos se efectúa mediante las instrucciones tipo *load* (lectura) y *store* (escritura), en sus variantes de byte (8 bits: LDB, STB) y word o float (16 bits: LD, ST y LDF, STF). La instrucción LDB lee un byte de memoria y lo escribe en un registro general R, extendiendo el bit de signo hasta completar los 16 bits del registro R. La instyrucción STB escribe en un byte de memoria los 8 bits de menor peso de un registro general R. Las instrucciones LD y ST transfieren words (16 bits, 2 bytes) entre los registros generales R y memoria mientras que las instrucciones LDF y STF, que también transfieren 16 bits, 2 bytes, lo hacen entre los registros de coma flotante F y memoria.. La dirección lógica de memoria a la que se accede con estas instrucciones se obtiene según el modo de direccionamiento registro base más desplazamiento. Los accesos a datos deben estar alineados a su tamaño natural:

- Los accesos a palabras, con las instrucciones LD y ST, deben estar alineados a direcciones múltiplo de 2 (con el bit de menor peso de la dirección con valor 0).
- Los accesos a floats, con las instrucciones LDF y STF, deben estar alineados a direcciones múltiplo de 2 (con el bit de menor peso de la dirección con valor 0).
- Los accesos a bytes, con las instrucciones LDB y STB, siempre están alineadas por definición, puesto que el byte es la unidad de direccionamiento.

Los bytes consecutivos de memoria que forman un dato de tamaño word (16 bits, 2 bytes) o float (16 bits, 2 bytes) o una instrucción (16 bits, 2 bytes) se numeran, o direccionan, siguiendo el convenio little-endian.

- Como el byte es la unidad de direccionamiento (la mínima cantidad de memoria que se puede direccionar) no tiene sentido hablar de sistema de direcciones little-endian o big-endian cuando se trata de acceder a un dato de tamaño byte. Las únicas instrucciones que soportan datos de tamaño byte son la LDB, que carga un byte de memoria en un registro general, extendiendo el bit se signo para ocupar los 16 bits del registro y la instrucción STB que almacena en memoria los 8 bits de menor peso del registro general fuente.
- Un word en memoria lo forman 2 bytes contiguos con direcciones @ y @+1, estando la dirección @ alineada a word, esto es, con el bit de menor peso a cero, @<0> = 0. El word se direcciona con la dirección @ (la instrucción LD o ST calcula la dirección efectiva @. El byte contenido en la dirección @ es el byte de menor peso de la palabra (el que tiene los bits 7:0) y el de dirección @+1 el de mayor peso (bits 15:8).
- Un float en memoria lo forman 2 bytes consecutivos siguiendo el mismo convenio que para el word.*****aqui se podria comentar algo de en que byte esta el exponente etc...**********

Cualquier instrucción a ejecutar debe estar en memoria alineada a direcciones múltiplo de 2 (con el bit de menor peso de la dirección con valor 0, como un dato de tamaño word, o como un float).

Cualquier acceso a memoria mal alineado, tanto para instrucciones como para datos, genera una excepción del tipo “Acceso a memoria mal alineado”.

Secuenciamiento Implícito. Registro PC

SISA-F es una arquitectura con secuenciamiento implícito. Tiene un registro especial denominado PC (*Program Counter*) que direcciona la memoria para efectuar la búsqueda de las instrucciones. Todas las

instrucciones, después de su ejecución, dejan el valor del PC modificado. Después de ir a buscar a memoria la instrucción que indica el PC, y antes de decodificarla, se incrementa el PC en 2 unidades, ya que 2 es el tamaño en bytes de una instrucción, (PC PC + 2). Esta actualización del PC se hace sea cual sea la instrucción a ejecutar, ya que se efectúa antes de su decodificación. El valor del PC durante la ejecución, propiamente dicha, de la instrucción, indica cuál es la dirección de memoria donde se encuentra la siguiente instrucción a ejecutar. A este valor del PC, durante la ejecución de una instrucción, lo denominamos PC actualizado, PCup (*updated*).

Las instrucciones de ruptura de secuencia (saltos) son las únicas instrucciones que durante su ejecución pueden volver a modificar el valor del PC. Las instrucciones de salto que calculan la dirección destino del salto (dirección de la siguiente instrucción a ejecutar) de modo relativo al PC lo hacen sumando un desplazamiento al PC actualizado, PCup. El desplazamiento, que se encuentra en la instrucción codificado con 8 bits, se multiplica por 2, antes de ser sumado al PCup. Se multiplica por 2 para poder acceder a instrucciones más alejadas de la actual que si se sumara el desplazamiento tal cual, y porque las instrucciones siempre se encuentran alineadas en direcciones pares de memoria. Se puede acceder a instrucciones que están desde -127 a 128 instruccines de la instrucción de salto ((PC+2-128*2)/2,...,PC+2+127*2)/2).

En este documento, para no resultar repetitivo, al explicar qué hace cada instrucción al ejecutarse, no se especifica explícitamente cómo se incrementa el PC antes de la decodificación de la instrucción. Esto es, no se incluye la acción: PC PC + 2.

Bancos de registros: generales (R), de coma flotante (F) y especiales (S)

Existen tres conjuntos de registros: generales (R), de coma flotante (F) y especiales (S).

Registros generales. 8 registros de propósito general de 16 bits: R0, R1,...,R7. Estos registros se usan:

- como operandos fuente en instrucciones aritméticas y de comparación con números naturales y enteros, en instrucciones de operaciones lógicas bit a bit, en instrucciones de ruptura de secuencia condicional (para contener el booleano sobre el que se evalúa la condición), en instrucciones de almacenamiento en memoria STB y ST (para contener el dato que se desea almacenar) y en instrucciones de ruptura de secuencia a través de registro (para contener la dirección destino del salto);
- para almacenar el resultado en instrucciones aritméticas y de comparación con números naturales y enteros, en instrucciones de operaciones lógicas bit a bit, en instrucciones de comparación de coma flotante, en las instrucciones de carga de memoria LDB y LD y de carga de inmediato MOVI, MOVHI, y en instrucciones de ruptura de secuencia en las que se guarda la dirección de retorno; y
- como registros base para calcular la dirección efectiva en todas las instrucciones de acceso a memoria.

Registros de coma flotante. 8 registros para operaciones con números en coma flotante de 16 bits: F0, F1,..., F7. Estos registros e usan:

- como operandos fuente en instrucciones aritméticas y de comparación con números reales en coma flotante y en la instrucciones de almacenamiento en memoria `STF` (para contener el dato que se desea almacenar), y
- para almacenar el resultado en instrucciones aritméticas con números reales en coma flotante (no se usan para el destino de la comparación de dos operandos de coma flotante, se carga en un registro general `Ri`) y en la instrucción de carga de memoria `LDF`.

Registros especiales S. Los registros especiales sólamente se pueden acceder explícitamente mediante las instrucciones `RDS` y `WRS`. La instrucción `RDS` lee un registro especial S y lo escribe en un registro general R y la instrucción lee un registro general y lo escribe en uno especial. También las instrucciones `EI`, `DI` y `RETI` modifican el registro `S7`, aunque no aparezca como operando explícito de la instrucción. Por otro lado, como resultado de la ejecución de otras instrucciones se pueden modificar algunos registros especiales, por ejemplo si se produce una excepción. También se modifican cuando ocurre una interrupción externa.

Los registros especiales son 8 registros de 16 bits: `S0, S1,...,S7`. Su uso es el siguiente:

`S0.` Contiene la palabra de estado que tenía el sistema cuando se produjo una excepción (interna), o interrupción (externa). Cuando se produce uno de estos eventos, se salva en `S0` la palabra de estado actual, que se encuentra en `S7`, ya que acto seguido se cambia la palabra de estado actual, para inhibir las interrupciones (externas).

`S1.` Contiene la dirección de retorno después de una interrupción (externa), una excepción (interna). Cuando se produce uno de estos eventos, se salva en `S1` el `PC` actualizado, ya que acto seguido se copia en el `PC` el registro `S5`, que contiene la dirección del código de entrada de las rutinas de atención de interrupciones y excepciones.

S2. Contiene, en sus cuatro bits de menor peso, un código binario que puede tomar valores de 0 a 15 que identifica el tipo de evento que se ha producido: tipo concreto de excepción (interna) o interrupción (externa). El resto de bits de S2 están siempre a 0. El código es:

0: Excepción de tipo “Instrucción ilegal”.

1: Excepción de tipo “Acceso a memoria mal alineado”.

2: Excepción de tipo “Overflow en operación de coma flotante”.

3: Excepción de tipo “División por cero en coma flotante”

4: Excepción de tipo “División por cero en enteros o naturales”

5,...,14: No usadas en esta versión. Reservados para futuras ampliaciones.

15: Interrupción (externa)

`S3.` Contiene la dirección efectiva usada por una instrucción de acceso a memoria, cuando ésta provoca una excepción de acceso a memoria mal alineado.

S4. Puede ser usado por el programador como variable temporal.

S5. Contiene la dirección de memoria donde se encuentra la siguiente instrucción a ejecutar después de producirse una excepción (interna) o interrupción (externa). En esta dirección de memoria comienza el código de la rutina de servicio genérica (RSG) encargado de identificar la causa de excepción o interrupción y en cada caso llamar a la rutina de servicio de excepción (RSE) o a la rutina de servicio de interrupcón (RSI) específica correspondiente.

S6. Su uso se reserva para ampliaciones futuras del lenguaje.

S7. Es el `PSW` (Processor Status Word). Contiene el estado del procesador en todo momento. Cada bit tiene el siguiente significado:

Bit M (`PSW<0>`): Se reserva para ampliaciones futuras del lenguaje.

Bit I (`PSW<1>`): Bit que indica si las interrupciones están permitidas (en cuyo caso vale 1) o si están inhibidas (en cuyo caso vale 0). Este bit se modifica con las instrucciones `EI` (Enable Interrupt) y `DI` (Disable Interrupt), que lo ponen a 1 o a 0 respectivamente y la instrucción `RETI` que restaura la palabra de estado y retorna de una excepción o interrupción. También, se pone a 1 cuando se produce una excepción o interrupción.

Bit V (`PSW<2>`): Bit que indica si la excepción de overflow en operación de coma flotante está permitida (en cuyo caso vale 1) o si está inhibida (en cuyo caso vale 0).

Entrada/Salida

El espacio de direccionamiento de los registros de entrada y salida está separado del espacio de memoria. Existen dos espacios separados, uno para la entrada de datos, INPUT y otro para la salida de datos, OUTPUT. Cada uno de estos dos espacios contiene 2^8 registros, o puertos, de 16 bits cada uno. Las instrucciones `IN` (*input*) y `OUT` (*output*) realizan la lectura y escritura de los registros del entrada y salida, respectivamente. Son las únicas instrucciones para acceder a los espacios de entrada y salida. Una misma dirección de registro puede referirse a dos registros físicos diferentes, uno de lectura que se accede con la instrucción `IN`, y otro de escritura que se accede con `OUT`. No obstante, dependiendo de la implementación, una misma dirección puede referirse a un único registro físico de lectura/escritura, pudiéndose acceder tanto con la instrucción `IN` como con la `OUT`.

Además de poder realizar entradas/salidas por encuesta con las instrucciones `IN` y `OUT`, la arquitectura puede gestionar interrupciones.

1.2.3 Tipos de Datos

Los datos que son operandos fuente o destino de las instrucciones se encuentran en elementos de memorización del computador como son los registros del procesador, la memoria, etc. Incluso en el caso de operando fuente inmediato, el dato se encuentra en un campo de la propia instrucción.

Los tipos de datos manejados por las instrucciones de esta arquitectura son de dos tamaños posibles: byte (8 bits) y palabra o word (16 bits, 2 bytes) o float (16 bits, 2 bytes).

Los bits dentro de un byte, word o float en esta arquitectura se numeran, de 0 a `n`-1, siendo el bit 0 el bit de la derecha, que representa el bit de menor peso de un dato entero o natural y el bit `n`-1 es el bit de mayor peso en un número natural representado en binario y el bit de mayor peso y además bit de signo en un número entero representado en complemento a 2. También es el bit de signo en un número float.

Los tipos de datos, según los interpretan las instrucciones que los manipulan, son:

- Vectores de 16 bits sobre los que se efectúan operaciones lógicas bit a bit (bitwise).
- Valores booleanos TRUE y FALSE, que se codifican en binario en un vector de 16 bits mediante los valores 1 y 0 respectivamente.
- Números naturales codificados en binario con 16 bits (unsigned integers) sobre los que se hacen operaciones aritméticas y de comparación.
- Números enteros codificados en complemento a 2 (Ca2) con 16 bits (signed integers) sobre los que se hacen operaciones aritméticas y de comparación.
- Números reales codificados en 16 bits en el formato de coma flotante SISA-FLOAT16 sobre los que se hacen operaciones aritméticas y de comparación. El formato SISA-FLOAT16 es una simplificación del estándar ANSI/IEEE 754-1985 basic simple format adaptado a 16 bits.

El formato SISA-FLOAT16 codifica números reales del tipo:

$$2^{k}(1 + 0,f)$$

mediante 1 bit para el signo, 6 bits para el exponente `k` y 9 bits para la mantisa `f`. Siguiendo el modelo del estandar IEEE 754, el bit de signo se codifica 0 para los valores positivos y 1 para los negativos, el exponente se codifica en binario exceso 31 y la mantisa en binario natural, normalizada al primer bit significativo distinto de 0, con coma a la derecha y bit oculto (`1+0,f` siendo `f` la mantisa de 9 bits almacenada). El formato SISA-FLOAT16 mantiene, además, la codificación de dos ceros (+0 y -0) para los números de exponente y mantisa idénticamente iguales a 0, pero no se extiende a las demás codificaciones especiales, es decir, no admite codificaciones especiales para infinito, NANs, SNANs ni números denormales. Estos casos se tratarán mediante overflows o redondeos. El formato SISA-FLOAT16, además, solo admite un modo de redondeo por truncamiento, no admitiendo los diferentes modos del estándar ANSI/IEEE 754-1985.

Ej: El número decimal -16,25 se codificaría como sigue.

-16,25 = -10000,01 (en binario natural) = $-10000,01 * 2^0 = -1,000001 * 2^4$ (normalizado)

En formato SISA-FLOAT16 tendríamos:

signo = 1

exponente = 4 + 31 = 35 = 100011

mantisa = 0000 0100 0 (ocultando el primer bit)

-16,25 = 1 100011 000001000 (SISA_FLOAT16)

En memoria es posible tener datos de tamaño byte, por ejemplo para almacenar un carácter ASCII, pero al cargarlos en un registro general del procesador para poderlos manipular, se extiende el bit de signo a 16 bits, para que ocupe todo el registro.

En algunas instrucciones hay campos de 6 bits u 8 bits que codifican números naturales en binario o enteros en complemento a 2, que se usan como operandos fuente (inmediatos).

1.2.4 Modos de direccionamiento

Los operandos fuente y destino de una instrucción se encuentran en alguno se los elementos de memorización cuyo contenido forma el estado del computador. Denominamos modos de direccionamiento a las diferentes formas de especificar, en las instrucciones, donde se encuentran los operandos fuente y donde se debe escribir el resultado de una instrucción. También podemos decir que los modos de direccionamiento son las distintas formas que tiene el computador de obtener, para cada operando, el tipo de elemento de memorización de donde se debe leer, o en el que se debe escribir, y su dirección concreta si es en memoria o en un registro de entrada/salida o el número de registro si es en uno de los conjuntos de registros.

En general, cada modo de direccionamiento se especifica en la instrucción de una forma diferente. En algunas arquitecturas en las que un operando de una instrucción concreta puede usar distintos modos de direccionamiento, para cada operando existe un campo en la instrucción para especificar qué modo de direccionamiento usa. En la arquitectura SISA esto no es así, ya que para cada instrucción esta fijado el único modo de direccionamiento que se usa para cada operando. Por ello, la información de qué modos de direccionamiento se usan para cada operando esta implícitamente codificada en el código de operación de la instrucción. Así, el código de operación de la instrucción indica el formato de la instrucción, la funcionalidad (o parte de ella, ya que algunas instrucciones tienen otros campos para terminar de definirla) y los modos de direccionamiento para los operandos fuente y destino.

Los distintos modos de direccionamiento que aparecen en las instrucciones SISA-F se explican a continuación.

Modo Registro. El operando está (si es fuente) o debe escribirse (si es destino) en un registro. En la instrucción debe codificarse la información para saber en qué conjunto de registros está el operando (si hay varios) y qué registro es entre los del conjunto (número de registro). En SISA-F hay tres conjuntos de registros, R, F y S de 8 registros cada uno. El conjunto de registro está indicado en el código de operación de la instrucción. Los tres bits necesarios para especificar el número de registro de cada operando que usa el modo registro se codifican en campos específicos de la instrucción. El modo registro se usa en el SISA-F para los operandos fuente y destino de todas las instrucciones que efectúan cálculos aritméticos, lógicos, comparaciones, etc. Estas instrucciones no usan ningún otro modo de direccionamiento, excepto la `ADDI`, en la que el segundo operando fuente se encuentra en la propia instrucción (usa el modo inmediato, como se indica a continuación).

Modo inmediato. El operando fuente se encuentra codificado en un campo de la propia instrucción (sólo se usa para operandos fuente). En SISA-F hay dos instrucciones de movimiento que tienen un operando fuente en modo inmediato, `MOVI` y `MOVHI` y una de cálculo, `ADDI`, que tienen el segundo

operando fuente en este modo. Para `MOVI` y `MOVHI` el operando inmediato es un vector de 8 bits y para `ADDI` es de 6 bits.

Modo registro base más desplazamiento. Este es un modo de direccionamiento para acceder a operandos en memoria. Se calcula la dirección de memoria del operando sumando al contenido de un registro (que contiene una dirección de memoria) un desplazamiento, que puede ser positivo o negativo y que se codifica en la propia instrucción en complemento a dos con menos bits de los que requiere una dirección de memoria. Este modo permite acceder a datos que se encuentran a una distancia fija (codificada en la instrucción) en torno a una dirección de memoria que se encuentra en un registro (que hace de puntero, porque apunta a una dirección de memoria) que puede ir cambiando de una vez a otra que se ejecute la instrucción.En SISA-F todas las instrucciones de acceso a memoria tanto de lectura, *load* (`LDB`, `LD` y `LDF`), como de escritura, *store* (`STB`, `ST`, `STF`), calculan la dirección de memoria con este modo de direccionamiento. El registro base es uno cualquiera de los registros generales del procesador (`Ri`) y el desplazamiento está codificado en complemento a 2 en un campo de 6 bits en la propia instrucción, `N`. Una particularidad de este modo de direccionamiento en SISA-F es que antes de sumar el desplazamiento al registro base, se multiplica el desplazamiento por 1 o 2 según la instrucción sea de acceso a byte o a word/float, ya que los datos en SISA están alineados en memoria a su tamaño natural y con esta multiplicación conseguimos que con un desplazamiento de pocos bits se pueda llegar más lejos entorno al puntero que si no se realizara la multiplicación del desplazamiento. Esta variante del modo de direccionamiento se llama "*registro base más desplazamiento con escalado*".

Modo implícito. Cuando el operando está (si es fuente), o debe escribirse (si es destino), en un registro que no se encuentra codificado en ningún campo específico de la instrucción, porque el registro es fijo para esa instrucción, se dice que se usa el modo de direccionamiento *registro implícito*. El registro está implícitamente codificado en el código de operación de la instrucción y todas las instrucciones con ese código de operación usan el mismo registro. En SISA-F, como en prácticamente todos los lenguajes máquina, podemos decir que todas las instrucciones acceden a un operando fuente y destino en modo implícito. Este operando es el registro especial `PC`, que no se especifica en la instrucción, pero que es leído y escrito por todas las instrucciones, para indicar cuál es la siguiente instrucción a ejecutar. Además del acceso al `PC`, solamente hay una instrucción que tiene uno de sus dos operando fuentes que podemos denominar como implícito, porque no se especifica en la instrucción en un campo separado. Esta es la instrucción `MOVHI Rd, Ns`, en la que el registro `Rd` hace de fuente (implícito) y destino. Esta idea de modo de direccionamiento implícito puede aplicarse también a otros modos de direccionamiento que no se especifiquen en un campo de la instrucción. Por ejemplo, podemos decir que operan con una constante fija usan el modo inmediato implícito, pero esto ya es rizar el rizo,...

Por último, decir que también se usa el termino modo de direccionamiento como la forma de calcular la dirección de la siguiente instrucción a ejecutar en las instrucciones de ruptura de secuencia (esto es, cómo se obtiene la dirección que se carga en el `PC`). En SISA-F hay dos tipos de instrucciones de ruptura de secuencia, según el modo de direccionamiento usado. Las instrucciones `BZ` y `BNZ` calculan la dirección destino del salto (cuando se debe romper el secuenciamiento implícito) sumando al registro `PC` un desplazamiento que se encuentra codificado en complemento a dos en un campo de la propia instrucción. Podíamos llamarlo modo de direccionamiento registro base implícito más desplazamiento (pues el `PC` no se especifica explícitamente, siempre es el `PC` en estas instrucciones), pero se suele llamar **modo relativo al PC**.

1.2.5 Excepciones y Interrupciones

Cuando se produce una petición de interrupción (externa) ésta será tratada si las interrupciones estan globalmente permitidas: el bit I de la palabra de estado vale 1. Cuando se produce una excepción (interna), ésta será normalmente tratada, excepto en el caso de que sea enmascarable (sólo lo es la excepción de overflow de coma flotante) y no esté permitida: si el bit V de la palabra de estado vale 0. Si la excepción o interrupción está permitida, entonces

1. se guarda en `S0` la palabra de estado, que se encuentra en `S7`,
2. se guarda en `S1` la dirección del `PC` actualizado, `PCup`, que es la dirección de retorno, ya que se debe retornar a la misma instrucción que provoco el fallo,
3. se escribe en `S2` el código del tipo de evento ocurrido, para informar a la rutina de servicio (RSG) de lo ocurrido,
4. se escribe en S3, en el caso de una excepción de acceso mal alineado a memoria, la dirección efectiva impar utilizada por la instrucción que provocó la excepción,
5. se carga en el `PC` el contenido del registro `S5`, que tiene la dirección del único punto de entrada a la rutina de servicio de excepciones/interrupciones genérica (RSG),
6. se modifica la palabra de estado, registro `S7`, inhibiendo las interrupciones: `PSW<1> 0`, y
7. se pasa a ejecutar la instrucción que indica el `PC`, que es la primera instrucción de entrada a la RSG.

Para permitir que la RSG pueda ser interrumpida en caso de recibirse una petición de interrupción (externa), habría que permitir las interrupciones de nuevo, ejecutando EI, puesto que éstas se inhiben cada vez que se invoca la RSG. La correcta programación para admitir interrupciones anidadas requiere un diseño cuidadoso de la RSG que no se va a explicar aquí. Por otro lado, el código de las rutinas RSG, RSE y RSI debería estar siempre libre de excepciones, de modo que tampoco se pueda interrumpir su ejecución a causa de una excepción, lo cual invocaría de nuevo la RSG. No obstante, puesto que esa excepción podría ser causada de modo imprevisto por un código incorrecto, es conveniente que la RSG salve y restaure no sólo los registros Ri sinó también S0, S1 y S3 antes de llamar a una rutina RSE/RSI susceptible de producir excepciones. En resumen, la rutina RSG debería seguir la siguiente secuencia de pasos al ser invocada:

1. Salvar en la pila todos los registros Ri que vayan a ser modificados por la rutina, sin excepción.
2. Salvar en la pila los registros S0 (donde se guardó la palabra de estado de retorno) y S1 (donde se guardó la dirección de retorno), y S3 (donde se guarda la dirección efectiva mal alineada en caso de ser una excepción de este tipo).
3. Copiar `S2` (el tipo de evento que activó la ejecución de la rutina) en un registro R para examinarlo. Si S2 contiene un número entre 0 y 14, se trata de una excepción, y en tal caso se indexará con S2 en una tabla - el vector de excepciones - para obtener la dirección de la RSE específica correspondiente. Si `S2` vale 15, entonces se trata de una interrupción, y debe de ejecutarse la instrucción `GETIID` para solicitar y cargar en un registro el identificador del dispositivo (al ejecutarse `GETIID`, el dispositivo se entera de que se ha aceptado su interrupción y envía su identificador por el bus). Después hay que indexar con ese identificador en una tabla - el vector de interrupciones - para obtener la dirección de la RSI específica correspondiente.
4. Saltar a la dirección de la rutina específica de excepción (RSE) o de interrupción (RSI), y ejecutarla.

Al regresar de la RSE o RSI (aunque no siempre retornan), la RSG debe de efectuar la siguiente secuencia de pasos:

1. Restaurar los valores de S0, S1 y S3 que se salvaron en la pila
2. Restaurar los registros salvados en la pila
3. Ejecutar la instrucción RETI la cual restaura en el PSW, esto es en S7, la palabra de estado que tenía el programa cuando se produjo el evento, y que se encuentra en S0; y carga en el PC la dirección de retorno al código interrumpido, y que se encuentra en S1.

1.3 Instrucciones

En esta sección se especifica cada una de las instrucciones SISA-F agrupadas por familias. Las instrucciones de una familia comparten alguna funcionalidad y/o el formato de sus instrucciones. Para cada familia se indica el formato en lenguaje máquina (campos de bits de las instrucciones en binario y su significado), las instrucciones que la forman, su semántica (los cambios que produce la ejecución de la instrucción en el estado del computador) usando una notación muy compacta, la sintaxis en lenguaje ensamblador y finalmente una descripción textual de la semántica.

1.3.1 Notación usada

En adelante, para especificar la semántica de cada instrucción, o familia de instrucciones, se usa la notación que se indica a continuación. En esta sección, se efectúan explicaciones generales para algunos grupos de símbolos y luego, para cada símbolo, se denota a la izquierda la sintaxis usada y a la derecha su significado. También se dan ejemplos de uso.

Para cada familia, se indica el formato en lenguaje máquina, las distintas instrucciones de la familia según los códigos del campo que sirve para extender el código de operación, la sintaxis en lenguaje ensamblador, la semántica, usando una notación compacta y finalmente una descripción más amplia de la semántica.

Formato de las instrucciones y campos de bits

`I` Es el vector de 16 bits que codifica en lenguaje máquina la instrucción o familia de instrucciones que estamos tratando.

Si queremos destacar los distintos campos que forman la instrucción de acuerdo con su formato, separamos los campos por un espacio. Ejemplo:

I = `1000 110 111 010 110`

Para denotar los bits que forman cada uno de lo distintos campos de una instrucción específica o una familia de instrucciones, usamos letras diferentes. Además, para que resulte visualmente menos engorroso no indicamos los subíndices que marcan el peso de cada uno de los bits del campo (se entiende que el bit de la derecha es el de menor peso).

`cccc` Vector de bits $c_3c_2c_1c_0$ que codifica el código de operación de una instrucción. De hecho, para todos los formatos SISA, `cccc` = `I<14..11>`.

`e` Bit de extensión de código de operación, que tienen algunas instrucciones.

`ddd` Vector de bits $d_2d_1d_0$ que codifica en binario el valor `d`, que indica el número de registro que es el destino de la instrucción.

`aaa` Vector de bits $a_2a_1a_0$ que codifica en binario el valor `a`, que indica el número de registro de uno de los operandos fuente de la instrucción.

`bbb` Vector de bits $b_2b_1b_0$ que codifica en binario el valor `b`, que indica el número de registro de uno de los operandos fuente de la instrucción.

`fffff` Vector de bits F = $f_4f_3f_2f_1f_0$ que codifica las distintas funcionalidades de una familia de instrucciones. Este tipo de campo se puede dar en dos longitudes distintas, 3 bits (`fff`) y 5 bits (`fffff`).

`nnnnnn` Vector de bits N = $n_5n_4n_3n_2n_1n_0$ que codifica, en binario o en complemento a dos, un valor inmediato, un desplazamiento, o una dirección de entrada/salida, dependiendo de la instrucción en la que se encuentre. Este tipo de campo se puede dar en dos longitudes distintas, 6 y 8 bits.

Ejemplo:

El vector de 16 bits `I` = `1001 ddd aaa 001 bbb` denota el formato de la familia de las instrucciones de coma flotante. Para `fff` = 001 es la instrucción `SUBF Fd, Fa, Fb` cuya semántica es: `Fd  Fa - Fb`

`X<i>` Bit i del vector de bits `X` (para cualquier símbolo `X`, normalmente una letra mayúscula). Los bits del vector `X` se numeran del `0` al `n-1` empezando por la izquierda (por el bit de menor peso, si por ejemplo el vector representa un número natural en binario). Ejemplo:

Si `X` tiene 16 bits y representa un número entero codificado en complemento a 2, `X<15>` es el bit de signo y `X<4>` es el bit de peso 2^4.

`X<j..k>` Campo de bits del vector de bits `X` (para `j<k` y para cualquier símbolo `X`). En concreto, es el vector de `k-j+1` bits formado por los bits del `j` al `k`, ambos incluidos, del vector `X`. Ejemplo:

Si `I = 1001 ddd aaa 010 bbb` entonces `I<8..6> = aaa` (o más preciso, `I=`$a_2a_1a_0$).

Interpretación numérica de un vector de bits

Esta notación apenas se usa, pues especificamos las instrucciones como operaciones sobre vectores de bits, en vez de sobre los valores que representan los vectores de bits.

`Xu` Valor del número natural (unsigned integer) representado en binario por el vector de bits `X` (para cualquier símbolo `X`, normalmente una letra mayúscula). Ejemplo:

`X` = 10100, `Xu` = 20

`Xs` Valor del número entero (signed integer) representado en complemento a 2 por el vector de bits `X` (para cualquier símbolo `X`, normalmente una letra mayúscula). Ejemplo:

`X` = 10100, `Xu` = -12

`Xf` Valor del número real (float) representado en formato de coma flotante por el vector de bits `X` (para cualquier símbolo `X`, normalmente una letra mayúscula).

X = 1 000000 000000000, Xf = -0

Contenido de Registros, Memoria y Entrada/Salida

Registros

Cuando nos referimos al vector de bits contenido en el registro x usamos los símbolos: Rx, Fx o Sx, según se trate de un registro general, float o de sistema. El símbolo x puede ser un número del 0 al 7 (ya que es como se numeran los 8 registros de cada tipo) o la letra d, a o b. Cuando x es un número nos referimos a un registro concreto. Cuando es una letra, nos referimos al registro destino de la instrucción (cuando x es igual a d), o a uno de los registros que son operandos fuente de la instrucción (cuando x es igual a a o a b).

Para `i` = 0..7, `d`, `a`, `b`:

`Ri` Vector de 16 bits contenido en el registro general `i`.

`Si` Vector de 16 bits contenido en el registro de sistema `i`.

`Fi` Vector de 16 bits contenido en el registro de coma flotante `i`.

Memoria

$MEM_b[A]$ Vector de 8 bits contenido en el byte de memoria con dirección `A`, siendo `A` un vector de 16 bits.

`MEM[A]` Vector de 16 bits contenido en el word (palabra) de memoria con dirección `A`, según el convenio litle-endian, siendo `A` un vector de 16 bits.

Entrada/Salida

`IN[P]` Vector de 16 bits contenido en el registro P del espacio de entrada de datos (INPUT), siendo P un vector de 8 bits.

`OUT[P]` Vector de 16 bits contenido en el registro P del espacio de salida de datos (OUTPUT), siendo P un vector de 8 bits.

Funciones y operadores sobre vectores de bits

Funciones

`SEXT(X)` Vector de `16` bits formado por el vector X al que se le ha extendido el bit de signo por la izquierda hasta formar un vector de 16 bits. Por ejemplo:

```
SEXT(101101)= 1111111111101101
```

Operadores

Efectúan la operación que indica el operador sobre los dos vectores de bits que se especifican a la izquierda y derecha del operador (excepto para el oprador `NOT`, `~`, que sólo tiene un operando que se especifíca a la derecha del operador).

Los operandos y resultado de todos los operadores son siempre vectores de 16 bits (excepto en los operadores `sha` y `shl` en los que el operando de la derecha es de 5 bits).

Algunas operaciones, como por ejemplo la suma, pueden realizarse sobre distintos tipos de datos como números naturales en binario, enteros en complemento a dos o reales en formato coma flotante. En estos casos se usa el mismo símbolo de operador para los distintos tipos de datos, por ejemplo +, siempre y cuando pueda deducirse el tipo de datos de los operandos. Si uno de los operandos es un registro general, `Ri`, el tipo de datos puede ser natural o entero (o simplemente un vector de bits para los operadores que no interpretan el vector como un número como para operadores bitwise). Si uno de los operandos es un registro de coma flotante, `Fi`, el tipo de datos es un número real codificado en coma flotante. Por ejemplo, en el caso de la suma, no es necesario diferenciarla cuando actúa sobre registros `Ri`, ya que los 16 bits de menor peso de la operación son los mismos para ambas interpretaciones (y no se detecta desbordamiento ni para naturales ni para enteros). Cuando el resultado de la operación sea diferente para naturales que para enteros, se diferencian los símbolos del operador, como por ejemplo en el operador `*hu` y `*h` que calculan los 16 bits de mayor peso de la multiplicación de los operandos, considerando números naturales y enteros respectivamente.

Uno de los dos operandos se puede especificar como una constante entendiendose que el operador se aplica sobre el vector de bits que la representa en la misma codificación y número de bits que el otro operando.

Operadores Bitwise (bit a bit)

Efectúan la operación lógica bit a bit que indica el operador. El bit `i` del resultado es la operación lógica del bit `i` de cada uno de los dos operandos, para `i = 0,1,...` En el caso de la operación Or bit a bit sólo hay un operando, que se especifica a la derecha del operador.

`&` Bitwise And (And bit a bit).

`|` Bitwise Or (Or bit a bit).

`^` Bitwise Xor (Xor bit a bit).

`~` Bitwise Not (Not bit a bit), complementación lógica o complemento a 1

Ejemplo:

Si `X =1101101111000110`,

`Y = 0110000101001111`, entonces

`X & Y = 0100000101000110`

Operadores de desplazamiento

El vector a la derecha del operador tiene que tener 5 bits y se interpreta siempre como un número en complemento a dos. Valores positivos de Y indican desplazamiento a la izquierda y valores negativos desplazamientos a la derecha. Para los dos operadores de desplazamiento, en los desplazamientos a la izquierda se copian ceros en los bits de menor peso.

`sha` X shl Y es el vector de 16 bits que resulta de efectuar el desplazamiento aritmético del vector X a la derecha o a la izquierda tantos bits como indica el vector Y. En los desplazamientos a la derecha se copia el bit de signo en los bits de mayor peso. Otra forma de interpretar el operador `sha`, como operador aritmético, es que devuelve los 16 bits de menor peso[1] de la representación en complemento a 2 del número entero

$$Xs \times 2^{Ys}$$

Para Y_s negativo, la división por potencias de 2 es el cociente calculado considerando que el resto de la división siempre es positivo, independientemente del signo del dividendo.

`shl` X shl Y, es el vector de 16 bits que resulta de efectuar el desplazamiento lógico del vector X a la derecha o a la izquierda tantos bits como indica el vector Y. En los desplazamientos a la derecha se copian ceros en los bits de mayor peso. Otra forma de interpretar `shl` como operador aritmético es que

1. Para valores positivos de Ys los 16 bits de menor peso pueden no representar correctamente en complemento a 2 el valor entero $Xs \times 2^{Ys}$, pudiendose producir overflow.

devuelve los 16 bits de menor peso[1] de la representación en binario del número natural

$$Xu \times 2^{Ys}$$

Nota para los dos operadores `sha` y `shl`: Para valores positivos de Ys los operadores `shl` y `sha` realizan la misma función, ya que la multiplicación por potencias positivas de 2 es igual para naturales que para enteros, excepto en la detección de resultado no representable, que no se efectúa en ningún caso.

Operadores aritméticos

`+`	Suma.
`-`	Resta
`*`	Multiplicación
`/`	División
`*hu`	Mul-High-Unsigned, devuelve los 16 bits de mayor peso (*High*) del resultado de la multiplicación considerando números naturales codificados en binario (*Unsigned integers*).
`*h`	Mul-High, devuelve los 16 bits de mayor peso (*High*) del resultado de la multiplicación considerando números enteros codificados en complemento a dos.

Operadores condicionales

Evalúan la condición que especifica el operador sobre los valores representados por los vectores de bits que se especifican a izquierda y derecha del operador. El resultado de la evaluación es el valor 1 o 0, codificado en binario con 16 bits, según se cumpla o no la condición.

El operador de igualdad `==` se puede usar tanto para vectores de bits que representan números enteros como naturales. Para el caso de números codificados en coma flotante se usa el operados `==F` ya que existen dos representaciones del valor 0 en coma flotante (+0 y -0) y si se usara el operado `==` el resultado de la comparación sería incorrecto.

`==`	Igual (operandos naturales, enteros o vectores de bits sin interpretación)
`<`	Menor que (operandos enteros en complemento a 2)
`<=`	Menor o igual (operandos enteros en complementoa 2)
`<u`	Menor que (operandos naturales en binario –Unsigned integers)

1. Para valores positivos de Ys los 16 bits de menor peso pueden no representar correctamente en binario el valor natural $Xu \times 2^{Ys}$, pudiendose producir overflow.

`<=u`	Menor o igual (operandos naturales en binario –Unsigned integers)
`==f`	Igual (operandos reales en coma flotante)
`<f`	Menor que (operandos reales en coma flotante)
`<=f`	Menor o igual (operandos reales en coma flotante)

PC actualizado

`PCup`	PC actualizado (updated). Es el vector de 16 bits que se obtiene al incrementar en dos unidades el contenido del registro PC (Program Counter) cuando el PC contiene la dirección de memoria de la instrucción a la que nos referimos cuando se explica la funcionalidad de las instrucciones. Representa, en binario, la dirección de la siguiente instrucción en memoria respecto de la que estamos considerando.

Sentencias

	Sentencia condicional. `x` es una expresión condicional sobre vectores de bits que da
`if (x)`	como resultado el valor 1 o 0. Las sentencias `y` y `z` se expresan con la
`y`	notación que se explica en esta sección.
`else`	Si `x` es `1`, se efectúa la sentencia `y`, si no, se efectúa la sentencia `z.`
`z`	(el término "`else z`" puede omitirse).

Asignación

Asigna vectores de bits. Los vectores, fuente y destino, a derecha e izquierda de la flecha respectivamente, tienen que tener el mismo número de bits.

Ejemplos:

`Rd` $\leftarrow$ `M`$_w$`[Ra]`

`Rd<15..8>` $\leftarrow$ `I<7..0>`

`M`$_b$`[Ra]` $\leftarrow$ `Rb<8..0>`

1.3.2 Formato de las instrucciones

Todas las instrucciones del lenguaje máquina SISA-F son de tamaño fijo de 16 bits. Como se muestra en la Figura <$chapnum>.1 hay 3 formatos de instrucción distintos. Cada formato tiene asociada una "forma" de descomponer los 16 bits de la instrucción en distintos campos. Cada campo codifica una

información específica. Las instrucciones que necesitan codificar 3 registros usan el formato 3-Reg, las que requieren codificar dos registros usan el formato 2-Reg y las que sólo requiern uno usan el 1-Reg.

Todas las instrucciones tienen un campo de código de operación (`cccc`), situado en los bits 15 a 12 de la instrucción, `I<15..12>` = `cccc` (más formalmente: `I<15..12>` = $c_3c_2c_1c_0$). Además, existen campos que pueden verse como una extensión del código de operación, ya que codifican distintas funcionalidades para el mismo valor del campo `cccc`. Las instrucciones con formato 3-Reg tienen el campo `fff`, `I<5..3>` = $f_2f_1f_0$, que codifica 8 funcionalidades para el mismo código de operación. Además, las instrucciones del formato 1-Reg tienen el campo `e` de un bit, I<8>, que codifica dos posibles extensiones del código de operación.

Cada registro especificado en una instrucción se codifica en binario en un campo de 3 bits, ya que hay 8 registros de cada tipo (`R0`,…, `R7`; `F0`,…, `F7` y `S0`,…,`S7`). El campo `ddd` (más formalmente: $d_2d_1d_0$) codifica el registro destino de la instrucción (`Rd`, `Fd` o `Sd`, dependiendo del código de operación). Los registros que son operandos fuente se codifican en los campos `aaa` y `bbb`.

Más adelante se indica claramente qué información contiene cada campo de cada instrucción. Por ejemplo, el campo `I<11..9>` puede codificar un registro destino `Rd`, `Fd` o `Sd`; un registro fuente `Rb`, `Fb` o `Sb`; e incluso para algunas instrucciones puede no tener significado y debe contener tres ceros. Lo mismo ocurre para otros campos.

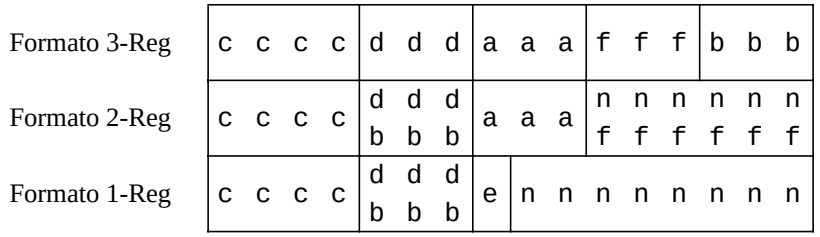

Fig. <$chapnum>.1 Formatos de las instrucciones SISA-F

1.3.3 Codificación de las instrucciones

La tabla de la Figura <$chapnum>.2 muestra el código de operación, campo `I<15..12>`, de todas las instrucciones SISA-F, afinando un poco más el significado del resto de campos de la instrucción. En los campos `ddd`, `aaa` y `bbb` se ha indicado el tipo de registro que se usa en esa instrucción, `R`, `F` o `S`. Las instrucciones se encuentran ordenadas por valor creciente del código de operación. Recuérdese que las instrucciones con código de operación `0000` a `0111` son las instrucciones del SISA-I (aunque con algún cambio semántico debido básicamente a que el SISA-I tiene el espacio de direccionamiento de memoria a nivel de word (16 bits)). Más adelante, después de explicar cada familia de instrucciones, se da una tabla con el formato y la codificación específica para cada una de las 55 instrucciones distintas del SISA-F. En la tabla, a la derecha de cada conjunto de instrucciones que comparten el mismo código de operación, se indica el nombre genérico del conjunto, o de la instrucción cuando solo haya una, y los

mnemotécnicos usados en ensamblador para referirnos a cada una de las instrucciones, en orden ascendente según el campo de función `fff`. La presencia de un guión, en vez de un mnemotécnico, indica que el código de función correspondiente no se usa (se reserva para posibles ampliaciones del lenguaje). Por ejemplo, de las instrucciones con código de operación `0000`, la que tiene codigo de función `fff` = `000` es la `AND`, la `ADD` tiene código de función `100`.

15 14 13 12	11 10 9	8 7 6	5 4 3	2 1 0		
0 0 0 0	d d d	a a a	f f f	b b b	Op. Lógicas y Aritméticas	AND, OR, XOR, NOT ADD, SUB, SHA, SHL
0 0 0 1	d d d	a a a	f f f	b b b	Comparación con y sin signo	CMPLT, CMPLE, -, CMPEQ CMPLTU, CMPLEU, -, -
0 0 1 0	d d d	a a a	n n n	n n n	Add inmediato	ADDI
0 0 1 1	d d d	a a a	n n n	n n n	Load	LD
0 1 0 0	b b b	a a a	n n n	n n n	Store	ST
0 1 0 1	d d d	0	n n n	n n n	Mover Inmediato	MOVI
		1				MOVHI
0 1 1 0	b b b	0	n n n	n n n	Salto condicional modo relativo al PC	BZ
		1				BNZ
0 1 1 1	d d d	0	n n n	n n n	Input	IN
	b b b	1			Output	OUT
1 0 0 0	Rd	Ra	f f f	Rb	Extensión aritmética	MUL, MULH, MULHU, - DIV, DIVU, -, -
1 0 0 1	Fd	Fa	f f f	Fb	Op/Cmp Float	ADDF, SUBF, MULF, DIVF CMPLT, CMPLEF, -, CMPEQF
1 0 1 0	Rb	Ra	0 0 0 f f f		Ruptura de secuencia modo registro	JZ, JNZ
	0 0 0	Ra				-, JMP
	Rd	Ra				JAL, -, -, -
			f f f f f f		Reservadas Fut. Ampl.	42 códigos
1 0 1 1	Fd	Ra	n n n	n n n	Load Float	LDF
1 1 0 0	Fb	Ra	n n n	n n n	Store Float	STF
1 1 0 1	Rd	Ra	n n n	n n n	Load Byte (8 bits)	LDB
1 1 1 0	Rb	Ra	n n n	n n n	Store Byte (8 bits)	STB
1 1 1 1			0		Reservadas Fut. Ampl.	32 códigos
	0 0 0	0 0 0	1	f f f f f	Instrucciones especiales	EI, DI, -, -
	0 0 0	0 0 0				RETI, -, -, -
	Rd	0 0 0				GETIID, -, -, -
	Rd	Sa				RDS, -, -, -
	Sd	Ra				WRS, -, -, -
						-, -, -, -
						-, -, -, -
	1 1 1	1 1 1				-, -, -, HALT

Fig. <$chapnum>.2 Tabla resumen del formato y codificación de las 47 instrucciones SISA-F.

1.3.4 Instrucciones de operación

Distinguimos tres subfamilias según el campo op.code: Aritmética básica y lógica, Extensión aritmética y Coma flotante.

Aritmética básica y lógica

Binario: `0000 ddd aaa fff bbb`

Código, Mnemotécnico, Funcionalidad y Nombre:

f f f	MNEMO	FUNC	Nombre
0 0 0	AND	&	Bitwise And
0 0 1	OR	|	Bitwise Or
0 1 0	XOR	^	Bitwise Xor
0 1 1	NOT	~	Bitwise Not
1 0 0	ADD	+	Add
1 0 1	SUB	-	Sub
1 1 0	SHA	sha	Shift Arithmetic
1 1 1	SHL	shl	Shift Logic

Semántica: NOT: `Rd ~Ra`

SHA y SHL: `Rd Ra FUNC Rb<4..0>`

Resto: `Rd Ra FUNC Rb`

Ensamblador: NOT: `NOT Rd, Ra`

Resto: `MNEMO Rd, Ra, Rb`

Descripción: Las instrucciones aritméticas ADD y SUB manipulan tanto datos enteros de 16 bits en complemento a dos como naturales de 16 bits en binario, ya que en ningún caso se detecta si el resultado es representable o no en 16 bits según la codificación usada.

Las instrucciones SHA (Shift Arithmetic) y SHL (Shift Logic) escriben en Rd el resultado de desplazar Ra a derecha o a izquierda tantos bits como indican los 5 bits de menor peso de Rb interpretados como un número en complemento a 2. Valores positivos indican desplazamiento a la izquierda y valores negativos desplazamientos a la derecha. En los desplazamientos a la izquierda se copian ceros en los bits de menor peso. En los desplazamientos a la derecha la acción depende de la instrucción: para SHA se copia el bit de signo en los bits de mayor peso, ya que la instrucción es Aritmética. Para SHL se copian ceros en los bits de mayor peso, ya que la instrucción es Lógica. Para valores positivos de Rb la instrucción SHL y SHA realizan la misma

función, ya que la multiplicación por potencias positivas de 2 es igual para naturales que para enteros, excepto en la detección de resultado no representable que no se efectúa en ningún caso.

La instrucción lógica `NOT` sólo tiene un registro fuente, codificado en el campo `Ra`. El campo `Rb` se codifica a `000` en esta instrucción.

Las instrucciones lógicas (`AND`, `OR`, `XOR` y `NOT`) operan bit a bit (*Bitwise Logical Operations*).

Extensión Aritmética

Binario: `1000 ddd aaa fff bbb`

Código, Mnemotécnico, Funcionalidad y Nombre:

f f f	MNEMO	FUNC	Nombre
0 0 0	MUL	*	Mul
0 0 1	MULH	*h	Mul High
0 1 0	MULHU	*hu	Mul High Unsigned
0 1 1	---	---	---
1 0 0	DIV	/	Division Signed
1 0 1	DIVU	/u	Division Unsigned
1 1 0	---	---	---
1 1 1	---	---	---

Semántica: `Rd  Ra FUNC Rb`

Ensamblador: `MNEMO  Rd, Ra, Rb`

Descripción: La instrucción `MUL` escribe en `Rd` los 16 bits de menor peso de la multiplicación de `Ra` por `Rb`. El resultado es correcto tanto para números enteros como para naturales. No se detecta overflow ni para enteros ni para naturales

La instrucción `MULH` escribe en `Rd` los 16 bits de mayor peso de la multiplicación de `Ra` por `Rb` interpretando los operandos y el resultado como números enteros codificados en complemento a dos (*signed integers* o simplemente *integers*).

La instrucción `MULHU` escribe en `Rd` los 16 bits de mayor peso de la multiplicación de `Ra` por `Rb` considerando números naturales codificados en binario (Unsigned integers).

La instrucción `DIV` escribe en `Rd` el cociente de la división entera de `Ra` entre `Rb`, interpretando los operandos y el resultado como números enteros codificados en complemento a dos (*integers*). El cálculo del cociente se redondea siempre hacia cero (se trunca). El resto de la división tiene el

mismo signo que el dividendo y su valor absoluto es menor que el del divisor. Si el divisor es cero, se produce una excepción de "división por cero". Si el dividendo es -2^{15} (0x8000) y el divisor es -1 (0xFFFF), el resultado en `Rd` es un valor indefinido, pero no se produce excepción.

La instrucción `DIVU` escribe en `Rd` el cociente de la división de `Ra` entre `Rb`, interpretando los operandos y el resultado como números naturales codificados en binario. El cálculo del cociente se redondea siempre hacia cero (se trunca), y el resto de la división es positivo y menor que el divisor. Si el divisor es cero, se produce una excepción de "división por cero".

Extensión de coma flotante

Binario: `1001 ddd aaa fff bbb`

Código, Mnemotécnico, Funcionalidad y Nombre:

f f f	MNEMO	FUNC	Nombre
0 0 0	ADDF	+	Add Float
0 0 1	SUBF	-	Sub Float
0 1 0	MULF	*	Mul Float
0 1 1	DIVF	/	Div Float
1 0 0	---	---	---
1 0 1	---	---	---
1 1 0	---	---	---
1 1 1	---	---	---

Semántica: `Fd  Fa FUNC Fb`

Ensamblador: `MNEMO  Fd, Fa, Fb`

Descripción: Las operaciones `ADDF` y `SUBF` realizan la suma y resta en coma flotante de los operandos `Fa` y `Fb` mediante el siguiente algoritmo: en primer lugar alinean ambas mantisas al mayor exponente, manteniendo una precisión máxima de 11 bits (incluido el bit oculto); a continuación suman o restan las mantisas (según la operación y el signo) y normalizan el resultado al primer bit significativo ajustando el exponente si es necesario; finalmente truncan la mantisa resultante a 9 bits (más uno oculto). Estas operaciones, en caso de dar lugar a un número no codificable por overflow generan la excepción correspondiente. Si se produce un underflow, el resultado queda truncado a 0.

Las operaciones `MULF` y `DIVF` realizan la multiplicación y división en coma flotante según el siguiente algoritmo: se multiplican los signos, se suman/restan los exponentes (teniendo en cuenta que ambos se encuentran codificados en exceso 31) y se multiplican/dividen las mantisas; a continuación se

normaliza las mantisa resultado y se ajusta el exponente si es necesario, truncando la mantisa resultante a 9 bits más uno oculto. Si cualquiera de estas operaciones genera un resultado no representable por desbordamiento del exponente se genera una excepción de tipo overflow en coma flotante (a diferencia del formato IEEE 754 que almacenaría la codificación de infinito). Si se realiza una división por +0.0 o -0.0 se genera una excepción de división por 0 en coma flotante.

1.3.5 Instrucciones de operación con inmediato

Sólo hay una instrucción de operación en la que el segundo operando fuente es inmediato.

Suma con Inmediato

Binario: `0010 ddd aaa nnnnnn`

Semántica: `Rd  Ra + SEXT(N)`

Ensamblador: `ADDI  Rd, Ra, Ns`

Descripción: Escribe en `Rd` los 16 bits de menor peso de la suma del contenido de `Ra` con el vector de 16 bits resultante de extender el bit de signo del campo `N` = `n n n n n n`.

El número entero `Ns` expresado en la sentencia ensamblador debe poderse representar en complemento a 2 con 6 bits, -32 <= `Ns` <= +31.

1.3.6 Instrucciones de comparación

Distinguimos dos subfamilias, comparación para números enteros y naturales comparación para números en coma flotante.

Enteros y naturales (con y sin signo)

Binario: `0001 ddd aaa fff bbb`

Código, Mnemotécnico, Funcionalidad y Nombre:

f f f	MNEMO	FUNC	Nombre
0 0 0	CMPLT	<	Menor que (enteros)
0 0 1	CMPLE	<=	Menor o igual (enteros)
0 1 0	---	---	---
0 1 1	CMPEQ	==	Igual que
1 0 0	CMPLTU	<u	Menor que unsigned (naturales)
1 0 1	CMPLEU	<=u	Menor o igual unsigned(naturales)

f f f	MNEMO	FUNC	Nombre
1 1 0	---	---	---
1 1 1	---	---	---

Semántica: `Rd Ra FUNC Rb`

Ensamblador: `MNEMO Rd, Ra, Rb`

Descripción: Cuando el mnemotécnico termina en `U` la comparación se efectúa considerando los operandos números naturales codificados en binario (*Unsigned integers*). En otro caso, las comparaciones se efectúan considerando los operandos números enteros codificados en complemento a dos (*signed integers* o simplemente *integers*).

La instrucción `CMPEQ` sirve tanto para números naturales como para enteros, ya que tanto en binario como en complemento a dos sólo existe una representación de cada número.

En todos los casos el resultado de las comparaciones es el correcto. Esto quiere decir que, por ejemplo, aunque la comparación de enteros y naturales se realice mediante una resta y el resultado de la resta no sea representable en 16 bits el resultado de la comparación que se escribe en el registro destino es el correcto.

Coma flotante

Binario: `1010 ddd aaa fff bbb`

Código, Mnemotécnico, Funcionalidad y Nombre:

f f f	MNEMO	FUNC	Nombre
0 0 0	CMPLTF	<f	Menor que float
0 0 1	CMPLEF	<=f	Menor o igual float
0 1 0	---	---	---
0 1 1	CMPEQF	==f	Igual que float
1 0 0	---	---	---
1 0 1	---	---	---
1 1 0	---	---	---
1 1 1	---	---	---

Semántica: `Rd Fa FUNC Fb`

Ensamblador: `MNEMO Rd, Fa, Fb`

Descripción: La comparación se efectúa considerando los operandos números reales codificados en 16 bits en el formato de coma flotante SISA16 (que es un subconjunto del formato IEEE para simple precisión).

La instrucción `CMPEQ` no puede usarse para el formato en coma flotante, ya que existen dos representaciones distintas del cero en el formato SISA16 (+0.0 y -0.0). Por ello para comparar si dos números reales son iguales se usa la instrucción `CPMEQF`. Todas las comparaciones en coma flotante consideran iguales ambas codificaciones del cero (es decir, -0.0 NO es menor que +0.0).

1.3.7 Instrucciones de accesos a memoria

Binario: *Loads*: `LDB`, `LD` y `LDF` `cccc ddd aaa nnnnnn`

Stores: `STB`, `ST` y `STF` `cccc bbb aaa nnnnnn`

Código, Mnemotécnico y Nombre:

c c c c	MNEMO	Nombre
1 1 0 1	LDB	Load Byte
0 0 1 1	LD	Load (word)
1 0 1 1	LDF	Load Float
1 1 1 0	STB	Store Byte
0 1 0 0	ST	Store (word)
1 1 0 0	STF	Store Float

Semántica:

```
LDB:      Rd    SEXT(MEMb[SEXT(N)+Ra])
LD:       Rd    MEM[SEXT(N)*2+Ra]
LDF:      Fd    MEM[SEXT(N)*2+Ra]
STB:      MEMb[SEXT(N)+Ra]    Rb<7..0>
ST:       MEM[SEXT(N)*2+Ra]    Rb
STF:      MEM[SEXT(N)*2+Ra]    Fb
```

Ensamblador:

```
LDB, LD:   MNEMO  Rd, Cs(Ra)
LDF:       MNEMO  Fd, Cs(Ra)
STB, ST:   MNEMO Cs(Ra), Rb
STF:       MNEMO Cs(Ra), Fb
```

Descripción: Para LDB y STB:

Se extiende el bit de signo de los 6 bits del vector N hasta completar 16 bits antes de sumarlos al contenido del registro Ra para obtener los 16 bits de la dirección de memoria a la cual se accede.

La instrucción LDB lee el byte de memoria cuya dirección se especifica en los campos Ra y N de la instrucción y lo escribe en los 8 bits de menor peso de Rd, extendiendo el bit de signo a los 8 bits de mayor peso de Rd.

La instrucción STB escribe los 8 bits de menor peso de Rb en el byte de memoria cuya dirección se especifica en los campos Ra y N de la instrucción.

Para LD y ST:

Se extiende el bit de signo de los 6 bits del vector N hasta completar 16 bits, se desplazan una posición a la izquierda introduciendo un 0 en el bit de menor peso y finalmente, los 16 bits de menor peso resultantes se suman al contenido del registro Ra para obtener los 16 bits de la dirección efectiva de memoria.

La instrucción LD lee la palabra (2 bytes) de memoria cuya dirección está especificada en los campos Ra y N de la instrucción -en little endian- y los escribe en Rd.

La instrucción ST escribe el contenido de Rb (2 bytes) en la dirección de memoria especificada en los campos Ra y N de la instrucción -en little endian-.

Para LDF y STF:

Se extiende el bit de signo de los 6 bits del vector N hasta completar 16 bits, se desplazan una posición a la izquierda introduciendo un 0 en el bit de menor peso y finalmente, los 16 bits de menor peso resultantes se suman al contenido del registro Ra para obtener los 16 bits de la dirección efectiva de memoria. (El cálculo de la dirección se efectúa igual que en las instrucciones LD y ST)

La instrucción LDF lee los 2 bytes de memoria cuya dirección está especificada en los campos Ra y N de la instrucción -en little endian- y los escribe en Fd.

La instrucción STF escribe el contenido de Fb (2 bytes) en la dirección de memoria especificada en los campos Ra y N de la instrucción -en little endian-.

Para todas las instruccionesde acceso a memoria:

Para todas las instrucciones de esta familia, el valor de la constante Cs que aparece en la especificación de la instrucción en ensamblador indica el número be bytes que hay que sumar al contenido del registro Ra para obtener la dirección efectiva de memoria a la que se accede. Para el caso de las

instrucciones LDB y STB el programa ensamblador debe codificar el valor Cs en complemento a 2 con 6 bits para formar el vector N, campo I<5..0> de la instrucción de lenguaje máquina. Para el resto de instrucciones, LD, ST y LDF y STF, el programa ensamblador debe dividir primero por dos la constante Cs antes de codificarla en complemento a dos con 6 bits para formar el campo N de la instrucción en lenguaje máquina. Así, Cs = Ns para LDB y STB mientras que Cs = Ns*2 para LD, ST, LDF y STF.

Con estas instrucciones se puede acceder a elementos de memoria que se encuentran a una distancia entre -32 y 31 elementos entorno a la dirección que contiene Ra. Aquí, un elemento es un byte para las instrucciones LDB y STB, un word para las instrucciones LD y ST y un float, para las instrucciones LDF y STF.

Los accesos a datos deben estar alineados a su tamaño natural: tanto LD y ST como LDF y STF deben estar alineados a direcciones múltiplo de 2 (con el bit de menor peso de la dirección con valor 0). LDB y STB, siempre están alineadas por definición, puesto que acceden a bytes, que es la unidad de direccionamiento. Cualquier acceso a memoria mal alineado genera una excepción del tipo “Acceso a memoria mal alineado”.

Los bytes consecutivos de memoria que forman un dato de tamaño word o float (16 bits, 2 bytes) se numeran, o direccionan, siguiendo el convenio little-endian.

1.3.8 Ruptura de secuencia (saltos)

Consideramos dos subfamilias según el modo de direccionamiento usado para calcular la dirección destino del salto: modo relativo al PC y modo registro. Las dos instrucciones con modo relativo al PC son además instrucciones de salto condicional. En las instrucciones con modo registro las hay condicionales e incondicionales y además hay instrucciones para llamada a subrutinas y a sistema operativo.

Saltos condicionales relativos al PC:

Binario: `0110 bbb e nnnnnnnn`

Código, Mnemotécnico y Nombre:

e	MNEMO	Nombre
0	BZ	Branch on zero
1	BNZ	Branch on not zero

Semántica: `BZ: if (Rb==0)PC PCup + SEXT(N)*2`

```
BNZ: if (Rb<>0)PC    PCup + SEXT(N)*2
```

Ensamblador: `MNEMO  Rb, label`

Descripción: Se extiende el bit de signo de los 8 bits del vector `N` hasta completar 16 bits, se desplazan una posición a la izquierda introduciendo un 0 en el bit de menor peso y finalmente, los 16 bits de menor peso resultantes se suman al contenido del PC ya actualizado para obtener los 16 bits de la dirección lógica que se carga en el `PC` en caso de salto tomado.

Con estas instrucciones de salto se puede romper la secuencia de ejecución y pasar a ejecutar una instrucción que se encuentra a una distancia de -127 a +128 instrucciones en torno a la instrucción de salto ((`PC`+2-128*2)/2,...,(`PC`+2+127*2)/2), ya que las instrucciones siempre se encuentran alineadas en direcciones pares de memoria.

El campo `label` que aparece en la sentencia ensamblador es el nombre de la etiqueta definida en la línea de código ensamblador donde se encuentra instrucción a ejecutar si se cumple la condición que provoca la ruptura de secuencia. El programa ensamblador codifica en complemento a 2 en el vector de 8 bits `N`, campo `I<7..0>` de la instrucción en lenguaje máquina, el valor que se optiene de restar el valor de la etiqueta label menos la dirección en la que se encuentra la instrucción de salto y dividir el resultado por 2.

Saltos a través de registro:

Binario:

```
JZ, JNZ:   1010 bbb aaa 000fff
JMP:       1010 000 aaa 000fff
JAL:       1010 ddd aaa 000fff
```

Código, Mnemotécnico y Nombre:

f f f	MNEMO	Nombre
0 0 0	JZ	Jump on zero
0 0 1	JNZ	Jump on not zero
0 1 0	---	---
0 1 1	JMP	Jump
1 0 0	JAL	Jump and link
1 0 1	---	---
1 1 0	---	---
1 1 1	---	---

Semántica: `JZ:  if (Rb==0)PC  Ra`

```
JNZ:      if (Rb<>0)PC    Ra
JMP:      PC    Ra
JAL:      Rd    PCup
          PC    Ra
```

Ensamblador: `JZ` y `JNZ`: `MNEMO  Rb, Ra`

```
JMP:      JMP  Ra
JAL:      JAL  Rd, Ra
```

Descripción: `JAL`: La instrucción Jump-And-Link se utiliza para llamada a subrutina. Para hacer un retorno de subrutina se puede usar `JMP`, ya que si se usa `JAL` se modificará un registro `Rd` que no será usado posteriormente.

1.3.9 Movimiento registro - Inmediato

Binario: `0101 ddd e nnnnnnnn`

Código, Mnemotécnico y Nombre:

e	MNEMO	Nombre
0	MOVI	MOV Immediat
1	MOVHI	MOV High Immediat

Semántica:

```
MOVI:     Rd    SEXT(N)
MOVHI:    Rd<15..8>    N
```

Ensamblador: `MNEMO  Rd, Ns`

Descripción: `MOVI`: El vector de 8 bits `N` se extiende a 16 bits para ser cargado en el registro `Rd`.

`MOVHI`: El vectror de 8 bits `N` se escribe en los 8 bits de mayor peso de `Rd`, y se mantienen inalterados los 8 bits de menor peso de `Rd`. Otra forma de especificar la semántica de la instrucción es decir que se escribe en `Rd` los 16 bits formados por los 8 bits del campo `N` a los que se les concatena por la derecha los 8 bits de menor peso de `Rd`. Esto quiere decir que esta instrucción tiene un operando fuente implícito, que es igual al operando destino, ya que `Rd` actúa como fuente (los 8 bits de menor peso) y como destino (los 16 bits).

El valor `Ns` especificado en la sentencia en ensamblador se codifica en complemento a dos con 8 bits en el campo `N` de la instrucción en lenguaje máquina, `I<7..0>`.

Ejemplo: Para cargar una constante arbitraria en un registro se puede usar una secuencia `MOVI` seguida de `MOVHI`, tal y como muestra el siguiente ejemplo: cargar 0x56ab en R4

```
MOVI    R4, 0xabc
MOVHI   R4, 0x56
```

1.3.10 Instrucciones de Entrada/Salida

Las instrucciones de entrada/salida se codifican con dos códigos de operación diferentes: `0111` para las instrucciones `IN` y `OUT` (que ya se encontraban en el SISA-I) y `1111` para el resto (el código `1111` también se usa para otras instrucciones especiales).

Entrada/Salida

Binario:
```
IN:  0111 ddd 0 nnnnnnnn
OUT: 0111 bbb 1 nnnnnnnn
```

Semántica:
```
IN:  Rd   IN[N]
OUT: OUT[N]   Ra
```

Ensamblador
```
IN:  IN   Rd, Nu
OUT: OUT  Nu, Rb
```

Descripción:

Los registros de entrada/salida son de 16 bits.

`Nu` es la dirección del registro de entrada o salida que se codifica en binario con 8 bits en el campo `N` de la instrucción, I<7..0>.

Resto de instrucciones especiales

Todas las instrucciones para la gestión de las entradas y salidas por interrupciones se encuentran englobadas dentro del código de operación `1111`, y vienen determinadas por el campo `fffff` tal y cómo especifica la siguiente tabla. Además comparten código de operación con estas instrucciones las de movimiento entre registros generales y especiales (`RDS`, `WRS`) y la instrucción especial de parada del computador (`HALT`). Los códigos no usados quedan libres para futuras ampliaciones del lenguaje; la ejecución de una instrucción con estos códigos genera una excepción de “instrucción ilegal”.

Binario:
```
EI, DI, RETI:       1111 000 000 1 fffff
GETIID:             1111 ddd 000 1 fffff
RDS, WRS            1111 ddd aaa 1 fffff
HALT:               1111 111 111 1 fffff
```

Código, Mnemotécnico y Nombre:

f f f f f	MNEMO	Nombre
0 0 0 0 0	EI	Enable Interrupts
0 0 0 0 1	DI	Disable Interrupts
0 0 1 0 0	RETI	RETurn from Interrupt
0 1 0 0 0	GETIID	GET Interrupt IDentification
0 1 1 0 0	RDS	ReaD System privileged register
1 0 0 0 0	WRS	WRite System privileged register
1 0 1 0 0	---	reservada para futuras ampliaciones
1 0 1 0 1	---	reservada para futuras ampliaciones
1 0 1 1 0	---	reservada para futuras ampliaciones
1 0 1 1 1	---	reservada para futuras ampliaciones
1 1 0 0 0	---	reservada para futuras ampliaciones
1 1 1 1 1	HALT	processor HALT

Semántica:

```
EI:       S7<1>   0     ;PSW.I = 1
DI:       S7<1>   1     ;PSW.I = 0
RETI:     S7   S0       ;restaura palabra de estado
          PC   S1       ;carga en PC dir. retorno
GETIID:   Rd   Data Bus
RDS:      Rd   Sa
WRS:      Sd   Ra
HALT:     Para el procesador
```

Ensamblador:

```
EI:       EI
DI:       DI
RETI:     RETI
GETIID:   GETIID  Rd
RDS:      RDS  Rd, Sa
WRS:      WRS  Sd, Ra
HALT:     HALT
```

Descripción: `EI`: Enable Interrupts, permite las interrupciones externas

`DI`: Disable Interrupts, Inhibe las interrupciones externas

`RETI`: Retorno de Interrupción o excepción.

`GETIID`: Activa la señal INTA para notificar al dispositivo que generó la petición que ésta va a ser atendida; y obtiene del bus de datos el identificador de interrupción enviado por el dispositivo, almacenándolo en el registro `Rd`.

`RDS`: Read Special Register. Permite leer el registro S especificado por el campo `Sa` y copiarlo en un registro de general `Rd`.

`WRS`: Write Special Register. Permite escribir en el registro S especificado por el campo `Sd` el valor contenido en el registro general `Ra`.

`HALT`: Para el procesador. Las acciones concretas dependen de la implementación.